Heriot-Watt University

The Augmented Reality Framework: An Approach to the Rapid Creation of Mixed Reality Environments and Testing Scenarios

Benjamin Charles Davis

April 2, 2009

Submitted for the degree of
Doctor of Philosophy in Electrical Engineering
on completion of research in the
Oceans Systems Laboratory,
School of Engineering and Physical Sciences.

Abstract

Debugging errors during real-world testing of remote platforms can be time consuming and expensive when the remote environment is inaccessible and hazardous such as deep-sea. Pre-real world testing facilities, such as Hardware-In-the-Loop (HIL), are often not available due to the time and expense necessary to create them. Testing facilities tend to be monolithic in structure and thus inflexible making complete redesign necessary for slightly different uses. Redesign is simpler in the short term than creating the required architecture for a generic facility. This leads to expensive facilities, due to reinvention of the wheel, or worse, no testing facilities. Without adequate pre-real world testing, integration errors can go undetected until real world testing where they are more costly to diagnose and rectify, e.g. especially when developing Unmanned Underwater Vehicles (UUVs).

This thesis introduces a novel framework, the Augmented Reality Framework (**ARF**), for rapid construction of virtual environments for Augmented Reality tasks such as Pure Simulation, HIL, Hybrid Simulation and real world testing. ARF's architecture is based on JavaBeans and is therefore inherently generic, flexible and extendable. The aim is to increase the performance of constructing, reconfiguring and extending virtual environments, and consequently enable more mature and stable systems to be developed in less time due to previously undetectable faults being diagnosed earlier in the pre-real-world testing phase. This is only achievable if test harnesses can be created quickly and easily, which in turn allows the developer to visualise more system feedback making faults easier to spot. Early fault detection and less wasted real world testing leads to a more mature, stable and less expensive system.

ARF provides guidance on how to connect and configure user made components, allowing for rapid prototyping and complex virtual environments to be created quickly and easily. In essence, ARF tries to provide intuitive construction guidance which is similar in nature to LEGO® pieces which can be so easily connected to form useful configurations.

ARF is demonstrated through case studies which show the flexibility and applicability of ARF to testing techniques such as HIL for UUVs. In addition, an informal study was carried out to asses the performance increases attributable to ARF's core concepts. In comparison to classical programming methods ARF's average performance increase was close to 200%. The study showed that ARF was incredibly intuitive since the test subjects were novices in ARF but experts in programming. ARF provides key contributions in the field of HIL testing of remote systems by providing more accessible facilities that allow new or modified testing scenarios to be created where it might not have been feasible to do so before. In turn this leads to early detection of faults which in some cases would not have ever been detected before.

Acknowlegdements

- Thanks to the EPSRC for funding this project.
- Thanks to David M. Lane and Yvan Petillot for providing supervision throughout the duration of this project.
- Thanks to Jonathan Evans for unofficial help in the begginning and support in pushing ARF into use.
- Many thanks to Chris Sotzing, Nick Johnson and Joel Cartwright for their continual usage of ARF and providing constant feedback.
- Many thanks to William Davis for help in proof reading this thesis, and Samuel Davis for providing wingman support at crucial times.

Publications

1. Davis, B.C.; Lane, D.M. (2008) *Enhanced Testing of Autonomous Underwater Vehicles using Augmented Reality and JavaBeans*: Underwater Vehicles, In-Tech Book, http://www.intechweb.org/books.php

2. Davis, B.C.; Patron, P.; Lane, D.M. (2007) *An Augmented Reality Architecture for the Creation of Hardware-in-the-Loop and Hybrid Simulation Test Scenarios for Unmanned Underwater Vehicles*: Oceans 2007

3. Davis, B.C.; Patron, P.; Arredondo, M.; Lane, D.M. (2007) *Augmented Reality and Data Fusion techniques for Enhanced Situational Awareness of the Underwater Domain*: OCEANS 2007 - Europe

Abbreviations

ALI - Abstraction Layer Interface

ARF - Augmented Reality Framework

AUV - Autonomous Underwater Vehicle

AR - Augmented Reality

AV - Augmented Virtuality

DIS - Distributed Interactive Simulation

DVE - Distributed Virtual Environment

DVL - Doppler Velocity Logger

GPS - Global Positioning System

GUI - Graphical User Interface

HMD - Head Mounted Display

HIL - Hardware-in-the-loop

HS - Hybrid Simulation

IDE - Integrated Development Environment

INS - Inertial Navigation System

JVM - Java Virtual Machine

JAR - Java Archive

OM - Online Monitoring

MHIL - Multiple Hardware-in-the-loop

OSL - Ocean Systems Laboratory

PS - Pure Simulation

ROV - Remotely Operated Vehicle

SIL - Software-in-the-loop

SLAM - Simultaneous Localisation and Mapping

UML - Unified Modelling Language

UUV - Unmanned Underwater Vehicle

VRML - Virtual Reality Modelling Language

VR - Virtual Reality

X3D - Extensible 3D

XML - Extensible Mark-up Language

XMSF - Extensible Modelling and Simulation Framework

XSLT - Extensible Stylesheet Language Transforms (XSL Transforms)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis addresses the problem of reducing the amount of real world testing required for a remote platform, in order to reduce costs, improve software quality and save time. This is achieved by concentrating on improving the available pre-real-world testing facilities, or more specifically providing a mechanism for testing facilities to be created quickly and easily. To achieve this, a generic extendible modular framework of components is required. The components of which can be arranged in a multitude of different ways allowing for many virtual environments and testing scenarios to be realised. This will enable scenarios to be constructed quickly, and as such significantly increase the quantity of available developmental testing resources. The first improvement over conventional testing methods is to allow the user to interpret data output by the remote platform in a variety of different ways and display this data in an intuitive form i.e. virtual reality representation. The developer can then identify problems more quickly because the virtual environment provides a 3rd person perspective making problems easier to spot. The framework will allow for user made components to be incorporated so that re-design is never necessary and so that new techniques for synchronising real and virtual data can be easily implemented and then used in a multitude of different applications.

In addition to reducing the required amount of real-world testing time, it is also vital to reduce the amount of time needed to actually create the testing scenarios so that adequate pre-real-world testing is not sidelined due to high testing harness creation time and consequently high expense.

1.1 Software Development

The software integration stage of software development projects can be tedious and time consuming. This is often due to the specification being too relaxed and not defined strongly enough. As a project is split up into its component parts specific information on how the components interact is not always known a-priori. Consequently, the interactions between different components may not adhere to exactly the same standard. Generally this is due to the data interfaces not being specified strictly enough in the design. This causes unforeseen errors at the integration stage where components are tested working together for the first time. These errors can be potentially dangerous and often go undetected until the latter phase of the testing cycle when more rigorous testing is carried out by end users, often doing things unforeseen and unanticipated by the programmers.

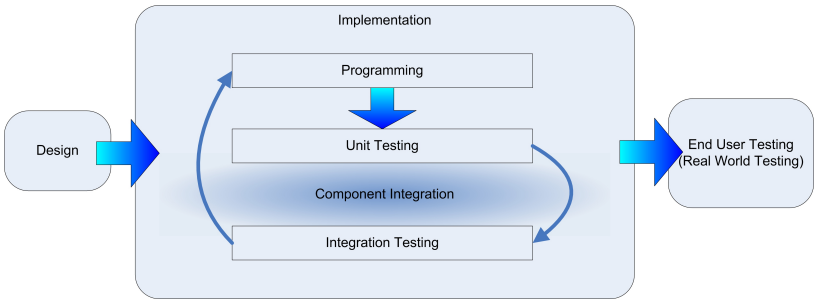


Figure 1.1: Software Development Cycle

This is particularly problematic when developing remote platforms; the platform may be inaccessible when it is in its intended environment, making the last phases of testing difficult, and consequently debugging very difficult and time consuming (Figure 1.1 shows the iterations through the integration testing phase). Often it is more costly to rectify problems during the later phases of testing, thus leading to larger costs which could have been avoided if testing facilities had been available throughout project development. A particularly applicable example being the development of Unmanned Underwater Vehicles (UUVs), and more specifically Autonomous Underwater Vehicles [1] (AUVs).

1.2 Autonomous Underwater Vehicle Testing

UUVs highlight a more specific problem; underwater vehicles are expensive due to water proofing to cope with the incredibly high pressures of deepest oceans (the pressure increases by 1 atmosphere every 10m). The underwater environment itself is both hazardous and inaccessible which increases the costs of operations due to the necessary safety precautions. Therefore, the cost of real world testing, the later phase of the testing cycle, is particularly expensive in the case of UUVs. Couple this with poor communications with the remote platform, due to slow acoustic communications, makes debugging very difficult and time consuming. This incurs huge expenses, or more likely, places large constraints on the amount of real world testing that can be feasibly done. It is paramount that for environments which are hazardous/inaccessible, such as sea, air and space, that large amounts of unnecessary real world testing be avoided at all costs. In a perfect world, different types of mixed reality testing facilities would be available to do most of the testing of the platform before hand. However, due to the expense of creating specific virtual reality testing facilities themselves, adequate testing is not always done before real world testing. This leads to failed software projects crippled by costs, or worse a system which is unreliable due to inadequate testing.

Testing facilities are themselves problematic to develop and as a consequence often ignored due to the expense of producing them in the first place. Simple tests created by the programmer serve their purpose but only test what the programmer expects to happen, which is often not the same as what is actually specified. Unexpected errors then arise during the integration of software components. Unexpected errors may be as simple as another component not following communications procedures strictly enough, or simply due to some misunderstanding, or more usually because documentation is inadequate or interface specifications were never specified correctly in the design phase. Consequently, as much integrated testing as possible should be carried out throughout module development so that these errors are detected early when they can be easily rectified, rather than during real world testing.

For these reasons testing facilities need to be made widely accessible to scientists

working on a multi-component project, so that continuous testing with both real and simulated components can be executed throughout project development. In order for this to be achieved a testing harnesses need to be quick and simple to create and highly flexible so that it can be used to test a multitude of different things.

1.3 Problem Statement

This thesis addresses the problem of reducing the amount of real world testing required for a remote platform, in order to reduce costs, improve software quality and save time. The problem remains of how to produce a generic pre-real-world virtual reality testing framework which allows for simulation components and real systems to be connected quickly and easily without having to program a new virtual environment from scratch each time. This thesis provides a framework which provides these features.

1.4 The Solution

This thesis presents a novel framework, the Augmented Reality Framework (ARF), for the rapid construction of virtual environments for use in Augmented Reality (AR) applications, such as the mixed reality testing of embedded systems operating in remote environments. ARF embraces a generic, extendable architecture based on JavaBeans [2] and Java3D [3] to provide a component based virtual environment which can be easily extended. ARF makes use of visual programming and introspection techniques to infer information about user's components, which is then used to provide guided construction for the interconnection of components. This allows for rapid prototyping enabling multiple testing combinations and virtual reality scenarios to be realised quickly for a multitude of different applications.

Mixed reality techniques can be used to render data visually in order to increase the situational awareness of the operator, and also provide simulation of systems and sensors for testing the remote platform. ARF provides the ability for all pre-real-world testing facilities to be realised. Consequently, increasing the rate at which errors are detected and hence developing a more robust system in less time.

ARF provides the required functionality by incorporating a set of visualisation tools, a distributed communication protocol and many simulated modules and sensors. Most importantly, ARF provides a generic architecture so that new simulated sensors and systems can be easily added to ARF's component base for use in potentially limitless applications such as in Hybrid Simulation (HS) [4, 5] (both real and virtual sensors working together). ARF incorporates the OceanSHELL [6] distributed communication protocol providing external communication allowing for the seamless substitution of real for virtual modules.

Most importantly ARF provides the ability for unlimited extension by allowing components to be easily created and added to ARF by the user. Therefore, if a particular component in ARF hasn't already been created, it is easily added since ARF is based upon the JavaBeans architecture. In effect, ARF is a JavaBeans development environment which is geared towards augmented reality and testing with the added enhancement of providing guided construction to the user. Guided construction is demonstrated by the analogy of LEGO®.

1.4.1 Guided Construction by Analogy: LEGO®

LEGO®[7] inherently provides intuitive guidance as to how it can be constructed. The guidance architecture provides small limitations on the things it can do, however, this makes it simpler for the user to use. For most purposes it can be used to create an approximation to a real structure, or make something just as good as. If the real structures cannot be created due to the limitations of the LEGO® pieces, the designers of LEGO® go back to their raw materials (a lower level) and produce a new higher level object. Take for example the LEGO® Tree. It is very hard to construct a sensible looking tree out of existing LEGO® pieces, therefore the designers dropped down a level of complexity and made a LEGO® tree from raw materials and made it have the same LEGO® connection interface so that it can be connected to other LEGO® objects. There are 3 main types of LEGO®: Basic, Normal and Technic. All LEGO® types have different components but can all be connected together. The way LEGO® can be interconnected and constructed is inherent in its design, the user can see how pieces can be fitted together and can therefore experiment and create

new structures by connecting them together in different ways.

The same can be said for Java and JavaBeans. Java uses class and interface types to specify which types of objects can be connected. However, although these allow objects to be connected, there is no particular way of showing the user how they can be interconnected in an intuitive fashion. Therefore, by combining JavaBeans with extra graphical technologies such as GUIs and JavaBeans Property editors, ARF is able to provide a guided medium which the user can utilise to link components together. The next step is to use Java3D to provide Java3DBeans, an extension to Java3D and JavaBeans created specifically for use in ARF, which provides the user with a virtual world where virtual components can be used to represent real world objects. ARF provides the necessary intuitive mechanisms that show the user how components can be linked together, not too dissimilar to LEGO®. ARF provides this via its guided construction mechanisms.

To go one step further a communications library called OceanSHELL[6] is also incorporated as a set of components which allows for the mixing of both virtual and real data. This flexibility means that virtual components can be swapped for real components, enabling a whole new set of uses for the architecture: for example HIL and other types of mixed reality testing. ARF now becomes incredibly applicable to the development of remote platforms such as UUVs, and meanwhile, maintains its application to normal integration testing and software development via JavaBeans.

The proof here is that by providing an intuitive interface, LEGO® makes construction of components easier to achieve. Therefore, since ARF provides intuitive user interfaces for connecting components, it should make constructing scenarios out of those components easier to achieve, since it is using a similar technique as LEGO® but through a slightly different visual mechanism.

1.5 Key Contributions

The primary focus of this thesis is improving the testing facilities for remote platforms. This is achieved by increasing the speed at which testing scenarios can be created. In turn this is achieved by providing a very flexible architecture which can be easily

extended and reconfigured by the user. In order to increase the speed of scenario construction, information about component interfaces is transformed by a guidance mechanism which gives the user options about how components can be connected, thus reducing the need to consult documentation or know which components are available. ARF is not restricted to testing applications for UUVs, since the virtual environment and generic architecture can be used to create virtual scenarios for many types of applications including Augmented Reality. However, the main focus of this thesis relates to applications to UUVs. Improving pre-real-world testing facilities through the use of virtual simulations of UUVs allows for higher level software modules to be continually tested throughout development, and consequently vastly reduces the amount of integration errors and problems occurring during real world tests and leads to a more mature, more reliable system. This thesis has improved the accessibility and flexibility of simulation, hardware-in-the-loop (HIL), Hybrid Simulation (HS) and real world testing facilities enabling them to be created quickly and easily. When some functionality is not available in ARF, due to some missing component, a component can be easily created and added by the programmer. This allows ARF to grow and be used for potentially many different uses. The time savings ARF provides can be easily converted to cost savings when applied to testing of new systems for AUVs. Real world testing is expensive and as such needs to not be wasted and kept to as little as necessary.

1.5.1 Key Applications

Apart from improving HIL testing facilities, ARF provides a facility which enables faults to be detected throughout testing and during online monitoring of the systems due to being able to quickly and easily add new feedbacks to the virtual environment to display outputs of systems running on the vehicle. ARF demonstrates how having the facilities to do this means that errors which would normally take a long time to debug can be found easily. An example of this is given in section 7.7.

1.6 Overview

The rest of this thesis is dedicated to explaining the various different parts of ARF including design, implementation, and examples of current and future usages and how ARF benefits them. Chapter 7 discusses use cases which demonstrate flexibility and performance of ARF. These are backed up by low level testing of ARF's architecture discussed in the testing Chapter 6. Firstly, some background literature about the concepts upon which ARF was conceived are discussed, and also the requirements and the niche which ARF fulfils are identified. Figure 1.2 gives an overview of the thesis structure illustrating which parts are unrelated and run in parallel.

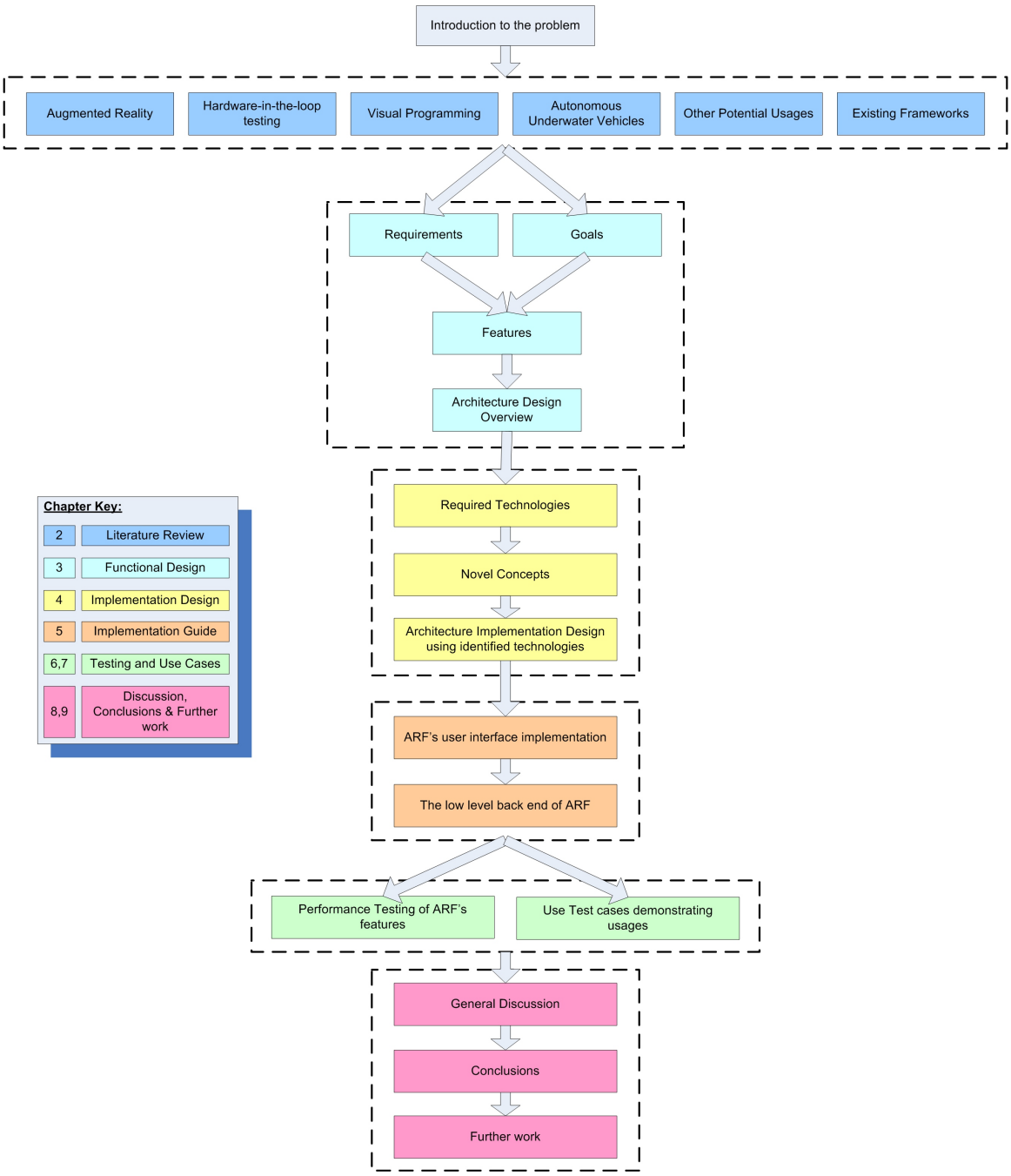


Figure 1.2: Thesis Structure

Chapter 2

Literature Review

This chapter provides an analysis of the relevant research in the areas of interest not limited to: augmented reality, underwater vehicles, simulation, visualisation, testing techniques and related technologies such as: multiple cooperating robots, mission planning, remote awareness and monitoring systems. Many key points and usages need to be considered when designing the Augmented Reality Framework. Since the problem being addressed by ARF is very broad, technologies and ideas are derived from many fields of research not limited to the underwater domain or even that of robotics. Topics such as visual programming are covered as this provides another method for constructing programs and potentially creating HIL testing scenarios and simulations.

For each area of research an overview will be given, and any relevant inferences which can be drawn from it will be discussed regarding their impact on the design of ARF.

To begin with this review will look into the field of Augmented Reality, as this has close ties with simulation and virtual environments which are used to provide testing facilities for underwater platforms.

2.1 Augmented Reality

Changing a persons/entities perception of the real world by adding virtual objects to the real world is defined as Augmented Reality (AR) [8, 9]. AR generally relies on

either the user wearing a special head mounted display, or the virtual objects being directly projected via a projector onto the real world. The latter of the two is a more rigid setup. AR is usually achieved using a see-through Head Mounted Display (HMD) [10], or by using a Virtual Reality (VR) headset (non see through HMD) and sensing the real world with video cameras before it is augmented with virtual objects. However, AR generally refers to human users having their real world view augmented with virtual data. If referring instead to some abstract entity, such as a robot, AR merely means altering that entities view of the real world, so the device used to alter the view of reality is specific to the way the entity perceives that reality. AR and VR are very useful for synthetic simulations of robots in their environments, and particularly AUVs as identified by Brutzman [11] who discusses the idea of a virtual world for an AUV.

AR is a more encompassing term than those used in testing of robotics, such as hardware-in-the-loop or hybrid-simulation, hence why this review focuses on AR as well as robotics related issues. The relevance of AR to robotics is discussed in more detail in section 2.3.

The framework being designed should provide applications for AR, but not be restricted to merely augmenting the perspective of some entity, such as a robot. For example, if a robot were driving around photographing corridors of a building, a virtual environment could be augmented with the photographic data from the robots cameras. Images could be positioned correctly in the virtual environment by keeping the virtual representation of the robot in synchronization with the robot's position in the real environment. Virtual objects also placed in the virtual environment could then be superimposed on the original images by simulating the cameras of the robot in the virtual environment. The virtual camera information would then be sent back to the robot's systems for processing, i.e. the robot would see the virtual objects superimposed on the real environment. This would be augmented reality from the robot's perspective, otherwise known as Hybrid Simulation (HS) (See section 2.3.3.1). The reverse of this is plotting the real sensor data in to the virtual environment for the operator to see. This is called Augmented Virtuality (AV). AV can be described as putting real data into the virtual world, in other words, augmenting the virtual view

with real data. Therefore a requirement is to provide both AR and AV simultaneously in order to accommodate all the potential usages. In general, a "reality" is being augmented for some entity, be it a virtual or real entity, hence the name given to the research of this thesis is the "Augmented Reality Framework".

2.1.1 Mixed Reality Continuum

Ronald T. Azuma[8] discusses Milgrim's Reality-Virtuality Continuum[12] of mixed reality (see figure 2.1) which describes the continuum from reality to virtual reality, and all the hybrid stages in between such as augmented reality and augmented virtuality. Any framework for testing of robots will certainly need to be able to provide facilities for all stages of the reality continuum. The required framework could technically be referred to as a mixed reality framework. Mixed reality inherently places the requirement for communication with external platforms.

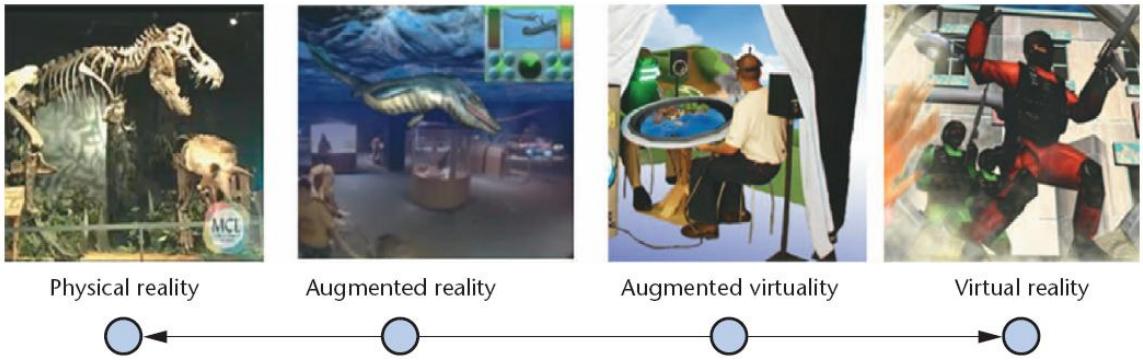


Figure 2.1: Milgrim's Technology Centric Continuum

2.1.2 Synchronisation Issues

Consider a person who is wearing a see through HMD; AR for humans can cause various problems disorientating the person wearing the see through HMD. These are mainly due to virtual and real world synchronization issues, e.g. by the time the computer has calculated where the user is looking and generated the graphics for the virtual objects in the correct place on the HMD, the user has probably moved a little more and therefore the images of virtual and real are not aligned. This can cause dizziness or motion sickness for the user. These are known as "dynamic errors"[13, 8].

One way of making sure that the real and virtual images are in synchronization is to use a non-see-through HMD. The real world is recorded by video camera and then the image from this is augmented with the virtual data. Therefore, the frames of the video camera and the frames of the virtual graphics are kept in synchronisation. However, the user may still experience lag from when they move to when they see the picture updated in their HMD as the real images are also delayed now as well as the virtual images. A greater loss is experienced due to lower resolution of the real world and the inability for the human to change where they are focusing, or move their eyes (unless eyeball movements are mimicked by the cameras).

These types of end-to-end delay errors reduce as technology improves and computer processors get faster, thus reducing lag time. Other techniques such as movement prediction can help to minimise the discrepancies between the real and the virtual worlds.

2.1.2.1 Registration Problem

The other main problem of all AR systems is that of Registration [14, 15] i.e. calculating exactly where the person is and where they are looking. Even the slightest error of a few millimetres can mean that the user can see discrepancies between the virtual view alignment and the real world. These errors can only really be reduced by more accurate methods of measuring the user's position, and by reducing the time between taking a position reading and displaying the data to the user.

The other types of errors are "static errors"[8] which are errors caused by inaccurate calibration of position sensing equipment and calibration of display for the user. It is quite difficult to calibrate a see-through HMD for a person because there are lots of different factors to take account of, for example the distance between a persons' eyes and their field of view. If not calibrated properly the user will see the virtual world not aligning correctly with the real world. This can cause motion sickness and is therefore not advisable. Non see-through HMDs are perhaps better for this reason but they have a problem of being incredibly low resolution compared with that of the human eye.

These problems do not necessarily need any attention when dealing with remote

environments because with remote environments there is always going to be a communication delay and sometimes no communication at all (as in the case of AUVs). Therefore the main problem is not that of synchronisation because the virtual environment will inherently be lagged behind the real anyway, so a few extra milliseconds lag will not make much difference, also virtual lag can be controlled and accounted for. However, registration and retrieving accurate position information is definitely required, but this is the job of the vehicle positioning systems and is not necessarily a problem which needs be addressed directly by this thesis since it should be handled by the robot's navigation systems. Most AUVs have a lot of expensive equipment for exactly calculating the orientation and position of the vehicle, although over time these are subject to drift. If the framework is being used for simulation and testing then it may be that a module is required to simulate the errors of the vehicle's navigation systems. However, this is up to the designer of the specific module to implement in their own way, as every navigation system will be different, and for most on land applications, there is usually relatively accurate navigation through GPS.

Robots, such as AUVs, also have problems with registration. Often when data is downloaded the same features in images will appear in different locations due to navigational drift. Systems which analyse this data on the fly can match the same features identified in different images to correct the robots position against navigational drift. This is called Simultaneous Localisation and Mapping (SLAM) [16]. These types of feature detection and alignment can be used in outdoor AR systems to help match the real geometry [14] of the surroundings to a virtual mock-up to help reduce alignment problems, or track the features to provide more accurate position information. This really identifies the requirement that no matter what the external system is, be it for human AR applications or autonomous robots, synchronisation of the real and virtual environments is required.

2.1.3 Applications of Augmented Reality

AR and AV are relatively new fields of research and only recently have graphics processors, computer processors and display technology become mature enough to enable the development of cost effective applications. AR has been used in education

and training as well as entertainment [17]. When a user is more involved in an experience, i.e. they are simply immersed in the scene in a natural way with the ability to interact with it, they actually learn a lot more from the experience. This is mainly due to there being no complicated user interface to learn in order to control movement and interactions. The user can simply look, move and interact with things in the same way as they would in real life making the task of observation and control very intuitive.

It is easy to see how this kind of technology would be very useful for an operator controlling a vehicle/robot in a remote environment. The third person perspective created by a virtual reality view of the environment allows the operator to gain better situational awareness about the surroundings of the robot, and thus make more informed decisions when deciding how to move the vehicle. The virtual environment can also be used to display the data the remote vehicle is gathering in an augmented virtuality manner. For instance, if the remote vehicle is an underwater vehicle, the data from the vehicle's sensors, such as sidescan sonar data which is rather like a photograph, or a bathymetric sonar which measures the bathymetry below the vehicle, can be rendered in the virtual environment in real-time. This gives the operator a better picture of the remote environment in which the vehicle is operating and increases their situational awareness. If the controls for the vehicle were intuitive, similar to a computer game, then anyone would be able to operate the vehicle within seconds. (To learn more about how computer games could influence ARF's design see section 2.4.4).

AR has already been used for remotely monitoring and operating underwater vehicles. A wearable[18] AR system has been developed which utilises a see-through HMD so that when operators are walking around the deck of a ship, they are able to see the position of the vehicle displayed on the HMD in relation to the real world, in a similar fashion to a Head Up Display (HUD) in a fighter plane. This is useful because vehicles are hard to see from the surface as they are underwater. There is no reason that the HMD of the operator could not be powered by the proposed framework, i.e. using it to generate the virtual graphics which are superimposed on top of the real world through the HMD. This places a requirement that the framework should

support plug-able modules so that new display devices and control devices can be used to view the environment and to move the virtual camera around in synchronisation with the real environment.

2.1.3.1 Augmented Scene Modelling

The principle concern of most AR applications is adding extra data to the real environment making it more understandable for a human and consequently increase the efficiency of working in that environment, e.g. highlighting areas of interest to the user. In environments such as underwater, where visibility is poor, highlighting and identifying certain items, such as structures, helps the operator orientate themselves. For example, the video camera feed from a robot could be augmented to show where the structures are in the distance. This is simple for an environment where the layout is known a priori, such as oil field structures and well-heads, but when in unstructured, unknown environments highlighting things of interest to the operator would be even more useful, but more difficult. A robot may come across some unknown structure and it may take readings from it with various different sensors, such as video and acoustic sensors, if underwater, to find out the distance and shape of the object. This data could then be combined and used to either highlight the structure using wire-frame (or something similar) or used to match the geometric properties of the structure to a known structure in a database[19]. This functionality would have to be part of a specific module. Again this highlights the need for the framework to support user modules which provide all these uses.

If the framework were to be extensively modular, a data interpretation and visualisation module could be created which looks at all the collected data, live or after the robot's mission is complete, and augment the virtual environment with the extra information and objects for the operator to see. One such example could be a module which allows the user to select and display various types of data, e.g. collected real data, virtual data added by a user, detected data or pseudo real data which represents identified structures or objects (chapter 7.7 discusses obstacle detection and avoidance. This shows how ARF is used to render data for the operator to see).

This concludes the AR section. Many of the concepts of mixed reality are dis-

cussed later in the applications of testing robots, and more specifically underwater robots. The next research area to be reviewed is underwater robotics. This will focus primarily on problems associated with underwater vehicles, and then more specifically how testing facilities can drastically improve reliability and maturity of underwater platforms.

2.2 Autonomous Underwater Vehicles

An Autonomous Underwater Vehicle (AUV) [1] is a type of unmanned underwater vehicle (UUV). The difference between AUVs and Remotely Operated vehicles (ROVs) is that AUVs employ intelligence, such as sensing and automatic decision making, allowing them to think for themselves. ROVs are controlled remotely by human operator with power and communications running down a tether. AUVs can operate for long periods of time without communication from an operator as AUVs execute a predefined mission which must be completed before returning to base. An operator can design missions for more than one AUV and monitor their progress concurrently. In contrast to ROVs which require at least one pilot per ROV to control them. Inherently the cost of using AUVs should be drastically reduced compared with using ROVs as long as the technology on the AUV is mature enough to do the task well.

AUVs have no tether or physical connection with surface vessels, such as boats, and therefore are free to move without restriction. Consequently, AUVs can be smaller and have lower powered thrusters than ROVs. This is partially due to not needing to drag a cable but also to conserve battery power. Consequently, AUVs are capable of going to greater depths far more easily and into highly structured environments which would prove hazardous to an ROV due to its tether.

In general, autonomous vehicles [20] can go where humans cannot or do not want to go, or in more relaxed terms they are suited to doing the "the dull, the dirty, and the dangerous" tasks. One of the main driving forces behind AUV development is automating tedious tasks which take a long time to do manually using an ROV; these can include oceanographic surveys, oil/gas pipeline inspection, cable inspection, clearing of underwater mine fields, etc. What these have in common is they are tedious

for humans to do or require expensive skills.

Figure 2.2: Nessie III and Remus AUVs deployed ready for mission start

Autonomous vehicles are capable of receiving initial mission data, moving to a different location, executing the mission, and then returning with the results of the completed task, be it survey data or just a status of as to how well the mission went. These vehicles are well suited to labour intensive or repetitive tasks, and can perform their jobs faster and with higher accuracy than humans. The ability to venture into hostile or contaminated environments is something which makes AUVs particularly useful. However, the main advantage of using AUVs over ROVs is that the amount of support required for an ROV is huge. ROVs require tether management, a large powerful boat which can sustain rough seas, a boat crew and highly trained ROV pilots. This causes the price per day of using ROVs to be very high. On the other hand, AUVs only require deployment and collection which allows the boat to do other tasks in the meantime (see figure 2.2). A much smaller deployment boat can also be used and therefore a much smaller crew. Realistically, AUVs cost more to develop, due to the intelligent systems onboard. However, once created many AUVs can be used in parallel by one operator and one ship, i.e. the benefits outweigh the initial development costs.

2.2.1 Autonomous Vehicle Architecture

Figure 2.3 shows an overview of the standard parts of an autonomous vehicle architecture. The majority of the intelligence of an AUV is contained in the *deliberative* and *executive* layers. This architecture is representative of a standard tri-level architecture which contains 3 main layers; the *deliberative*, *executive* and *reactive*. The *reactive* layer is called the *functional* layer in this diagram because it contains more systems than just manoeuvring (which is what *reactive* suggests). The general trend is that the *deliberative* layer contains the mission plan and the goals which are to be achieved by each mission. The *executive* layer takes the mission plan and assesses how the autonomous vehicle should carry out the mission. This then sends commands to the *reactive/functional* layer which controls the motion of the vehicle and its sensors. For example, a system such as obstacle avoidance is part of the *reactive* layer because it is not up to the mission executive to take account of obstacles and move around them. Therefore if an obstacle is detected the obstacle avoidance system would circumnavigate the obstacle to arrive at the waypoint issued by the mission executive. If a waypoint cannot be reached due to some reason it is up to the mission executive to realise this and execute a different part of the mission if possible and mark the failed leg as unachievable. More about mission planning and mission execution is discussed in section 2.4.3.

Intelligent systems, such as mission executive or obstacle avoidance, take time and lots of testing to develop to a high standard. Hence why a generic framework which is capable of providing all the functionality of the *synthetic environment* and *user interface* layers would be very useful for reducing development time. The ability to test different elements of the tri-level architecture using one testing framework would be advantageous. Therefore a testing framework needs to be able to be substituted easily for various real systems in the functional layer. For example, sensor simulation and vehicle motion could be provided by a synthetic environment so that the obstacle avoidance system, of the *functional* layer could be tested with the *executive* and *deliberative* layers running the system as a whole in virtual reality, rather than having to do expensive real world testing.

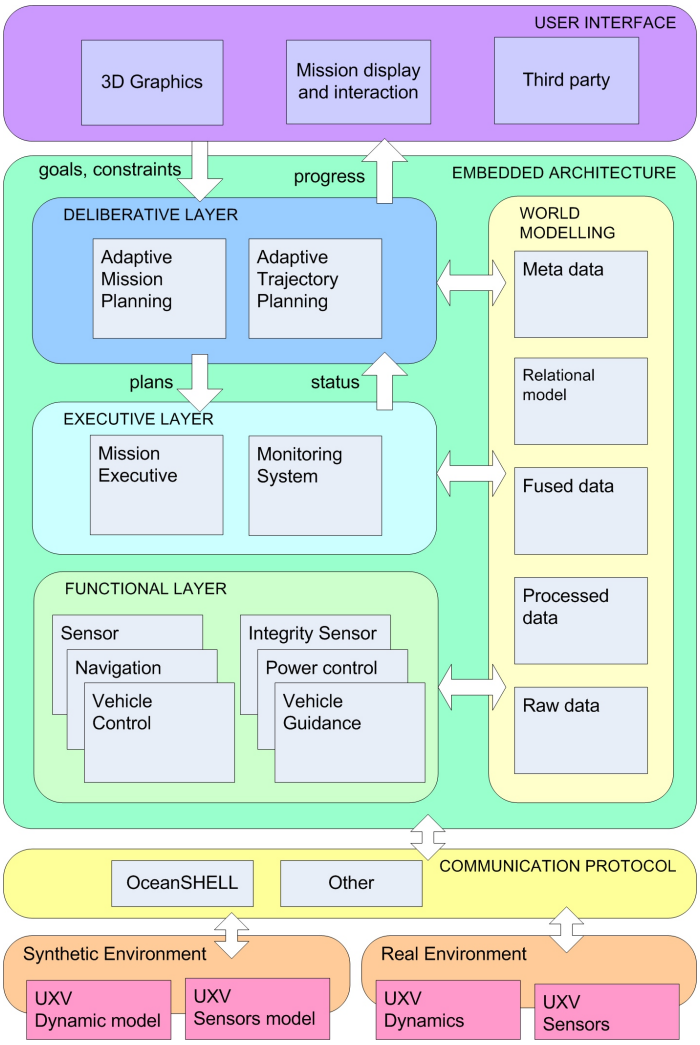


Figure 2.3: Autonomous Vehicle Generic Architecture Diagram

2.2.2 AUV Catagories

Many different sorts of AUVs exist and their purposes range tremendously:

- Voyager class (e.g. Gliders) - these tend to be quite small, torpedo shaped with fins, and very efficient because they use ocean currents as transport which helps save power. They usually have adjustable buoyancy which allows them to move vertically in water column allowing them to glide horizontally using their fins/wings. They have long endurance and are used for large scale sporadic surveying of the oceans.

- Survey class - These are usually torpedo shaped for good hydrodynamics allowing for long range missions. These are generally larger than gliders and size varies depending on how much endurance the AUV requires and how large the required payload sensors are. They are larger to accommodate the more accurate navigation systems and sensing equipment. They are used for shorter distance missions than gliders but can carry out detailed surveys in a timely manner (hours or days).

- Inspection class - These are usually capable of moving in 4 to 6 degrees of freedom which makes them incredibly useful for inspecting complex structures and navigating through cluttered environments.

For more specific AUVs and their applications see sections 2.8 and 7.

2.2.3 Communication Problems with AUVs

Communication with AUVs is problematic. Underwater acoustic communications are slow and unreliable. This causes problems when trying to test AUVs and monitor mission progress because real-time status information cannot always be retrieved from the AUV whilst in use. However, an AUV operator is much more comfortable if they monitor the AUV. Since this is not always possible due to communications problems, using simulation between acoustic updates helps provide tele-presence. If a simulated version of the same mission is kept in synchronisation with the real mission, the operator will have a good idea of AUV position and intent at any one time, as they

can see this from the virtual environment. This type of simulation helps install more trust in the operator of the AUV. Periodically, the simulated AUV will be updated by real data from the real AUV (This is known as Remote Presence). If acoustic communications are not available the AUV may come to the surface to exchange data and/or gain a GPS position update.

Therefore there is a requirement for the ability to synchronise real and virtual systems. This is a big problem since there could potentially be many different types of systems for a generic framework to support. Therefore, a mechanism which allows the user to provide special modules for synchronising internal and external systems needs to be included in the design. This also implies that the different types of external communication protocols should also be supported, in order to interact with different types of platform.

There are many applications for AR and AUV tele-presence which require the same functionality. This review now looks at the specific requirements of testing facilities for robotic platforms, specifically AUVs, and looks closely at the similarities to AR.

2.3 Hardware-in-the-loop and associated Testing Techniques

When creating robots for remote environments it is not always feasible to test the robot in its intended environment. This can be due to a number of reasons, the main ones being risk, expense and accessibility [21, 22]. However, it is very important that the systems are tested as a whole to make sure all the component parts interact correctly. The problem is that some parts of the software that produce output rely on the remote environment for input e.g. from sonar or video input. Therefore, it is the purpose of a synthetic environment to simulate the outputs of the sensors/systems which interact with the remote environment. These systems could include video, sound, navigation, environmental measurement sensors etc and their functionality must be replicated in a synthetic environment. The simulated sensors are plugged into the higher level systems of the robot to test that those higher level systems work correctly. This allows the robot to be tested in different environmental conditions

with varying features and types of terrain. Therefore the robot can be tested on many different scenarios which would be difficult to do in reality. This is very useful when designing software modules for the robot, which are specific to a certain sort of terrain/environment type because the environment can be tailored to the specific needs of the system being tested. Setting up complicated test scenarios in the real world may not be feasible even if the environment itself is easily accessible. For example, if testing tracking of multiple underwater oil pipelines, it may be to expensive to do this in the real environment and also too hard to diagnose any errors (see AUTOTRACKER in section 7.6). This type of testing is referred to as Hardware-in-the-loop (HIL) [23] or Robot-in-the-loop [24]. In essence a hardware-in-the-loop test bed is simply an extension to an existing distributed simulator, or Distributed Virtual Environment (DVE) [25]. This allows the robot to interact with the virtual environment rather than with the real environment i.e. placing the robot in virtual reality.

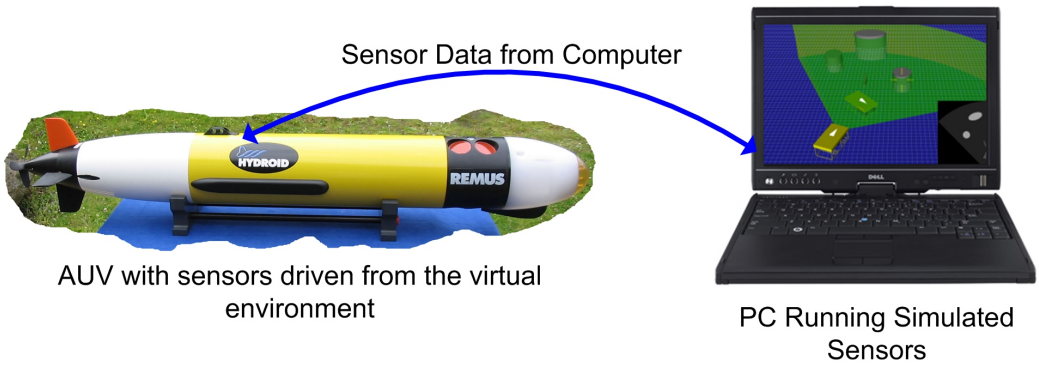


Figure 2.4: Hardware-in-the-loop

Hardware-in-the-loop testing requires that the hardware/sensors/modules of the robot being tested are simulated in order to provide the robot with the knowledge that it is interacting with the real environment (see figure 2.4). The problem then becomes the task of creating the virtual environment and all the simulated sensors required by the robot's higher level systems. In most cases as a new robot/environment combination is realised, a new simulator is designed in order to test it, or an older one is upgraded to accommodate the new robot/environment. However upgrading an older synthetic environment is not always that straightforward, and creating a new

simulator entirely from scratch is often just as uneconomical.

2.3.1 Considerations of Hardware-in-the-Loop Systems

Many HIL testing environments are currently available, however, most of them are platform or environment specific. For example, CoreSIM [23, 21] is a HIL testing facility for robots in a sub-sea environment. It discusses different approaches to HIL and also how their own "CoreSim" synthetic environment operates. The papers mainly discuss the synchronisation of modules via logical clocks, and the communication system which the modules use to communicate. One issue the author identifies is that some modules such as high fidelity sensor simulations are sometimes very hard to simulate accurately and may take longer, in terms of computation, than the real module. This creates the problem of how to synchronise the system, the simplest way being that the system runs as fast as the slowest module, or that the system is distributed over many machines to make use of parallel processing. One computer processor may not always be powerful enough to simulate the virtual world and all simulated sensors required. Therefore, any new architecture needs to be easily distributable over as many computers as required and provide the mechanisms for distributed processes to communicate (see section 4.1.5 on the OceanSHELL distributed communications protocol).

2.3.1.1 Distributed Virtual Environments

Since HIL is an inherently distributed concept, with some of the modules running on the robot's hardware and some within the virtual environment, it should not make any difference to the robot where the simulated modules are being executed. The robot doesn't know which machine the virtual environment is running on, therefore if (n) computers are creating the synthetic environment instead of 1 it will be transparent to the robot. As long as the modules/sensors have a mechanism for communicating their inputs and outputs, this means the simulated sensors could be distributed over different machines anywhere. This is known as a Distributed Virtual Environment (DVE) [25]. Inherently this functionality needs implementing by any virtual environment capable of HIL. Therefore, using multiple environments distributed over many

computers is merely an extension to the concepts of HIL.

2.3.1.2 Sensor Simulation

For robots, the sensor types can be split into 2 different categories:

- Proprioceptive sensors - measure information about the vehicle state, e.g. position, velocity, hull temperature, water leak sensor etc.
- Exterioceptive sensors - measure information about the surrounding external environment, e.g. sonar, video, echo sounder, x-ray vision, temperature sensors, ultrasonic rangers, compass (this senses magnetic field, however this is very easily simulated in software).

Since it is the external (remote) environment that is being simulated using virtual reality, some of these sensors become increasingly difficult to simulate accurately in real-time (i.e. the time spent by the real sensor to gather data). For example, sonar is computationally expensive to simulate due to the way sonar sound waves interact with the surrounding environment. Sonar works by producing a sound pulse in a certain direction, and then waiting for the reflections which bounce off the remote environment. However, in order to simulate this in a 3D virtual world each sonar pulse has to be traced through the virtual world and see where it intersects with the surrounding geometry, and then see how much would be reflected back. In computer graphics this technique is known as ray tracing and is very slow. A very realistic simulated sonar would also ray trace the reflected sound waves as well to see how they are reflected off the environment also. As a consequence this might produce double reflections of objects similar to real sonar images. In some circumstances the programmer can cheat if a less realistic output is required. For example, in the case of sidescan and bathymetric sonar the graphics processor can be used to output all the depths values of each pixel on the screen. This can then be modified to look like the sonar data. Therefore computationally expensive methods can be avoided if quality of simulation is not that important.

One problem with the CoreSIM [21] synthetic environment is that it does not provide the facility to test other non sub-sea environments straight off the rack; a

specific external module would have to be programmed and fitted into the architecture some how. This also applies to new sensors; they are not easily added to the simulation environment. Configurations have to be manually adjusted to allow the modules to interact. In essence all modules are standalone.

A new simulation framework should provide the flexibility for applications in any given domain. Most applications merely require a change in vehicle model, such as hydrodynamics in the case of underwater, terrain following with gravity etc, to simulate the interactions with the different environments. In essence, the only difference between any of these environments is how the simulated vehicle interacts with the virtual environment geometry. For example, in the virtual environment a land operated vehicle would implement some sort of terrain following algorithm because in the real world it would be affected by gravity which holds it on the terrain.

Each vehicle model type should simply be a component of a modular virtual environment. In order for a virtual environment to provide this it must have enough scope in the software design to accommodate all different vehicle specifications and provide the ability for them to interact with the virtual environment in different ways. Different sensors will be required for implementing the different vehicle models, e.g. a collision detection ray used for terrain following (see figure 2.5).

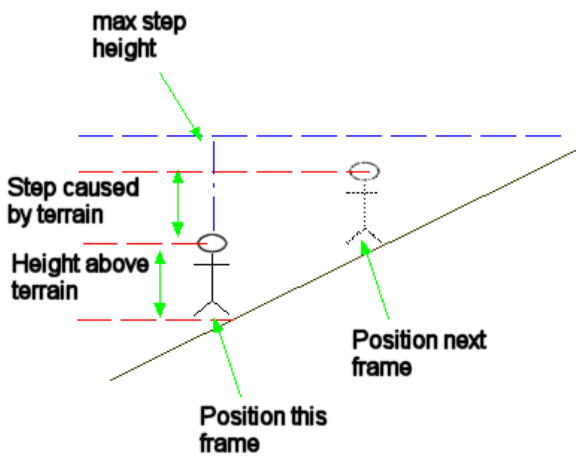


Figure 2.5: Terrain Following of an avatar up a hill - simulating gravity.

In essence the framework should simply be a virtual version of the real world, i.e. all things that can be done in the real world can be easily implemented in the virtual world, and thus all simulation will be able to be carried out in the virtual world. A

problem with such a generic architecture for building virtual environments is that a faster version of the same simulation could conceivably be created using a simpler model of the real world. For example the simulated sonars discussed earlier use ray tracing to trace the sounds pulse, however there are certain ways of ordering and pruning the 3D data which reduces the amount of intersection testing required and consequently improves performance. A generic set of modules may not always provide the method with highest performance because of the trade off between being generic and highly specialised. Therefore the framework architecture should itself be generic allowing for specialised, high performance, modules to be easy integrated. Then new user made components can be integrated which provide specific optimised functionality, e.g. a special 3D renderer could be used for ray tracing to provide sonar images. In order for a framework to be more useful it therefore needs to be easily extendible. Therefore, if a new component is particularly useful it can be shared amongst the scientific community for use in other simulations. New components should be easily implemented and added to the frameworks component base. For a simulation framework to be truly generic and be applicable to many applications, it needs to be able to naturally evolve without having to reprogram the whole environment. Extendibility needs to be at the core of its architecture.

2.3.2 A Taxonomy of Working Modes

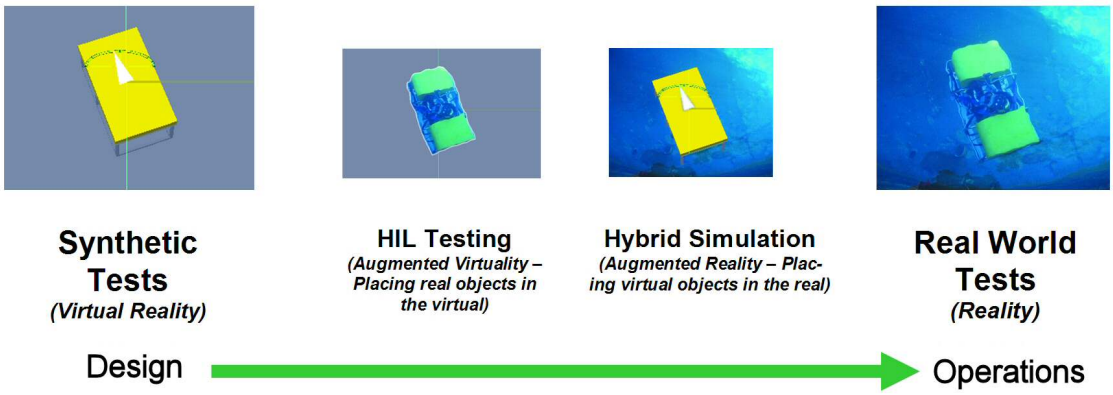


Figure 2.6: Reality Continuum combined with Testing Types

The hybrid stages between real and virtual are known as augmented reality [9]

and augmented virtuality [9]. Figure 2.6 shows how the reality continuum discussed earlier correlates with testing techniques. The hybrid reality concepts are built upon by the ideas of HIL testing and Hybrid Simulation. The NEPTUNE [4] system furthers the definitions of virtual environment uses via a taxonomy of 6 working modes. NEPTUNE provides methods for executing most types of testing and can therefore be considered to be the state of the art. However, it does not provide the extendibility and flexibility required for doing testing in all domains. The working modes are discussed are and correlated to augmented reality below:

- OFFLINE - Pure Simulation [4] (PS) - Simulated time may be faster or slower than real-time, e.g. if a simulated sonar takes a lot longer than the real thing then simulated time must be slowed down in order to let the sonar complete. This is used to do individual testing of each module before integration onto the intended platform.

- ONLINE - Pure Simulation (PS) - The simulation runs in real time and as a result puts performance requirements on the modules within the simulated system. Having a distributed architecture is one way of helping to make sure that every module can comply with the time constraints.

- HIL - Hardware-in-the-loop [23] is where most of the computation is actually done on the robot's hardware, in contrast to OFFLINE and ONLINE where all the systems run as part of the simulation. However, the robots actions are diverted to the virtual environment simulation rather than the real world systems. The robot may also receive its inputs from simulated sensors, such as sonars, running in the virtual environment. This is very useful for integration testing as the entire system can be tested as a whole allowing for any system integration errors to be detected in advance of real world trials.

- HS - Hybrid Simulation [22, 26, 4, 27, 28, 5] - This involves testing the platform in its intended environment in conjunction with some simulated sensors driven from a virtual world. For example, virtual objects can be added to the real world and the exterioceptive sensor data altered so that the robot thinks that something in the sensor dataset is real. This type of system is used if some

higher level modules are not yet reliable enough to be trusted to behave as intended. Consequently, fictitious data is used instead, augmented with current data, as inputs to these systems. This is so that if a mistake is made it does not damage the platform. An example of this would be obstacle avoidance system testing on an AUV (see example in section 7.7).

- OM - Online Monitoring - Real world testing - Can use the 3D virtual world for online monitoring of the robot in its remote environment. This is the last stage of testing i.e. all systems are trusted and the platform is ready for testing in the intended environment. Therefore, all integration errors should have been fixed in the previous stages otherwise this stage is very time consuming and therefore expensive.

- TR - Operator Training - This is where the operator interacts instead with a virtual world simulating a robot and learns how to control the robot without needing the real robot, i.e. training the operator, thus requiring less expensive real world testing time.

The NEPTUNE paper discusses lots of different simulators which each implement one or more of the working modes described above, and therefore those simulators will not be covered here since they are a lot less advanced than the NEPTUNE simulator. The NEPTUNE simulator employs all the working modes described above and is consequently a very useful virtual environment for those reasons. However, one problem with it is that it is particularly restricted to underwater applications in its present state and is far too specific to be extended easily. However, it identifies many uses for virtual environments such as sensor simulation, operator training and online monitoring. It has most of the constituent parts required for the building blocks for a good testing framework, but in its present state is not generic enough. One of the main focuses of this research is increasing the speed of the creation of virtual environments for any use, which NEPTUNE does not facilitate.

2.3.3 AR, Testing and UUVs

The different types of testing correspond to different stages of the reality continuum (See figure 2.6).

AR and AV refer to how the reality or virtual reality is altered respectively. In the case of AR, simulated data is added to the real world perception of some entity. For example, sonar data on an AUV could be altered so that it contains fictitious objects, i.e. objects which are not present in the real world, but which are present in the virtual world. This can be used to test the higher level systems of an AUV such as obstacle detection (see obstacle detection and avoidance example in Section 7.7). A virtual world is used to generate synthetic sensor data which is then mixed with the real world data. The virtual world has to be kept in precise synchronization with the real world. This is known as the *registration* problem discussed previously. The accuracy of registration is dependent on the accuracy of the position/navigation systems. Registration is a well known problem with underwater vehicles when trying to match different sensor datasets to one another for visualisation. Accurate registration is paramount for displaying the virtual objects in the correct position in the simulated sensor data.

AV is the opposite of AR, i.e. instead of being from a person's perspective it is from the virtual world's perspective - the virtual world is augmented with real data. For example, real data collected by real sensors on an AUV is rendered in real time in the virtual world in order to recreate the real world in virtual reality. This can be used for *online monitoring* and *operator training* as the operator can see how the AUV is situated in the remote environment, thus increasing situational awareness.

2.3.3.1 HS and AR

Hybrid Simulation is where the platform operates in its intended environment in conjunction with some sensors being simulated in real time using a synchronized virtual environment. Similarly to AR, the virtual environment is kept in synchronization using position data transmitted from the remote platform. Simulated sensors are attached to the virtual reality version of the remote platform and moved around in synchronization with the real platform. Simulated sensors collect data from the vir-

tual world and transmit the data back to the real systems on the remote platform. The remote platform systems then interpret this data as if it were real. It is important that the simulated data is very similar to the real data so that the higher level systems cannot distinguish between the two.

Other environments, such as Song's [22] and Neptune [4] provide hybrid simulation. However, Song makes the assumption that all simulation processing can be executed on one computer and thus relies on one computer having enough processing power to simulate all required sensors. This is because the modules communicate via shared memory and mutex locks rather than message passing over a network. This speeds up the performance of the simulator slightly, but makes it inflexible. Therefore, message passing is favoured due to the location transparency it provides.

In summary, the real platform's perception of the real environment is being augmented with virtual data. Hence HS is inherently AR. An example of a real scenario where AR testing procedures are useful is in the case of obstacle detection and avoidance in the underwater environment by an AUV in section 7.7.

2.3.3.2 HIL and AV/AR

HIL is another type of *mixed reality* testing. The platform is not situated in its intended environment, but is instead fooled into thinking that it is. This is achieved by simulating all required exterioceptive sensors using a virtual environment. Virtual sensor data is then sent to the real platform's systems instead of real data. The outputs of the platform's higher level systems, which rely on the simulated data, can then be relayed back and displayed in the virtual environment. This can help show the system developer that the robot is interpreting the simulated sensor data correctly and responding accordingly. In HIL all the sensors and systems that interact directly with the real environment are simulated. Vehicle navigation systems are a good example since these use exterioceptive sensors, actuators and motors to determine position. Using simulated sensors means that the developer can specify exactly the data which will be fed into the systems being tested. This is complicated to do reliably in the real environment as there are too many external factors which cannot be controlled. Therefore, HIL actually places the robot in virtual reality. Sometimes

a virtual environment is augmented with the feedback of the platform's data for observation, hence HIL is AV as well. Thus HIL and HS are both deemed to be *mixed reality* concepts.

2.3.4 Summary

It becomes clear that PS, HIL, HS are a subset of the main capabilities defined by AR, AV and VR. Therefore a new generic framework could support all of these capabilities and the *working modes* defined in section 2.3.2. This review now looks into other applications of virtual environments and existing implementations of testing frameworks.

2.4 Other Potential Usages and Requirements

This section looks at some background on other aspects of AUV development which need be considered when designing the framework. Although the framework should be flexible enough to be used on AUV unrelated tasks, it is impossible to conceive every possible application. Thus some of the main areas of AUV development are discussed and some other related topics are also discussed for their contributions towards this research. This shall start by looking at some of the uses of AUVs.

2.4.1 Applications of AUVs

The UUV Masterplan [29] is a document by the Department of Defence which outlines research topics that are seen as useful and vital to the further development of UUVs. This document is very specific to military applications but a lot of the techniques described can also be used for other, non-military, applications, e.g. inspection of underwater pipelines for the oil and gas industry. Most of the vignettes described in the UUV Masterplan require onboard intelligence on the AUVs for identifying objects with acoustic sensors such as sonar. This has applications in underwater obstacle avoidance, mine detection and ship hull inspection for use in sea port security. This sort of intelligence also has other applications in the underwater community, i.e. obstacle avoidance would be useful for any underwater activity be it military or

otherwise. Object detection can also be used by the oil and gas industry for finding pipelines (see AUTOTRACKER in section 7.6 for pipeline tracking and section 7.7 for obstacle avoidance) as well as being used by the military for detecting mines and enemy targets. The UUV Masterplan is mainly concerned with:

- Mine Counter Measures (MCM) - look for underwater mines and deploy a counter measure.
- Seaport security - concerned with guarding a port against a terrorist threat. For example, scanning the shipping lanes for explosive devices like mines, or, for scanning ships entering and leaving a harbour for foreign bodies which have been attached to their hull.
- Obstacle detection - apart from detecting mines, this is useful for detecting fishing nets and perhaps other enemy craft such as submarines.

These are just some of the popular areas of interest for AUVs, however there are far more uses particularly in offshore oil and gas and marine science industries. These include inspections of underwater structures, such as pipelines, and gathering of sea bed data for analysis. A number of topics have been researched in the Ocean Systems Laboratory (OSL) and have been the driving factor behind the research of this thesis. These include:

- Multiple vehicle coordination and cooperation which evolved into the DELPHÍS [30] system (see section 7.4).
- Increasing operator awareness - when an AUV is beyond communications distance, how do you keep the operator informed as of to what should be happening, and how do you explain when the vehicle has done something different.
- Creation of mission plans in an intuitive way.
- Hardware-in-the-loop testing approaches such as CORESIM [21] and now ARF [31].
- Integrating SLAM with inertial navigation to correct for drift over long periods of time.

- Purely visual based SLAM
- Mission plan adaptation when something goes wrong.
- Obstacle avoidance systems (See BAUUV section 7.7)
- Autonomous pipeline and cable tracking (see section 7.6
- Wall tracking
- Object detection, classification and inspection using Nessie III.

No matter what the application, the problems associated with keeping the operator/developer well informed about the AUV's operations in the remote environment are persistent. Improving situational awareness via tele-presence is one of the main issues. However, no matter what the application, a mission plan needs creating and often the systems need testing in simulation first. Similar facilities, but not the same, are used in each case. However, it should be possible to use the same environment to design the mission plans and then execute a simulation of that mission, i.e. same tools, different uses. Therefore a framework is required which provides varying levels of complexity allowing the creation of task specific tools, i.e. from simulation testing right through to observing real world data from real missions.

These are just some of the topics being research at the OSL, but most of them have one thing in common - they will at some point require testing in simulation, then hardware-in-the-loop and finally real world tests with real-time feedback. Hence the need for a robust, flexible and extendible testing and observation environment.

2.4.2 Testing and Visualisation of Cooperating Multi-Agent AUVs

In order to carry out certain tasks at sea, and indeed any environment, the use of multiple vehicles can reduce the overall time required to finish a task because some parts of a task maybe able to be subdivided into smaller tasks and carried out in parallel. It may also be the case there exists a pool of specialised vehicles which are in short supply. Tasks can then be delegated to these vehicles in the most effective way

so that energy is conserved and they are sent where they are most required. Therefore some way of coordinating the vehicles is required. As a result a method of monitoring and testing the cooperation of the different vehicles is essential. Multiple hardware-in-the-loop (MHIL) [32] testing may also need to be carried out and a facility for doing this will have to be created.

Testing of multiple agents in simulation is very useful because it allows developers of new coordination techniques to view how their system would perform in reality. One such way of coordinating agents is through the use of Blackboards [33] which works by every agent putting its data on a blackboard for all other agents to use. This task on the sea surface, on land or air is relatively straight forward, since the vehicles can always know where they are by using GPS and they can communicate easily with one another. However, in the underwater domain communication is slow, unreliable and relatively short range. Equipment for underwater acoustic communication is expensive so it is unlikely that each agent will have an acoustic communication device (acoustic modem), and therefore will have to surface to communicate. GPS is not available underwater either and more expensive navigation equipment is required, such as Inertial Navigation System (INS), Doppler Velocity Log (DVL) and fibre optic gyro. Unfortunately, even with this equipment the navigation is still subject to errors and drift over time. Therefore any system which coordinates multiple agents underwater has to take into account that it may not always have communication with each agent and the agents may not be able to communicate. In order to test this, taking into account the problem that communication underwater is basically unavailable, the designer of the algorithm for sharing AUV data would need to decided at what time this will occur so that all AUVs surface at the same time. Therefore, doing this task in simulation will show the programmer what the vehicles are doing and thus enable the designer to see that they are doing the correct thing. Lesser [34] describes multi-agents in general and is a good starting point for anyone who is interested in multi-agent coordination. Testing this kind of system in a real environment would be quite hard and expensive in practice. Therefore computer simulations are usually used to check the algorithms work correctly and visualisations are used to display what each agent is doing at any one time to the operator/developer.

Another approach to using multiple vehicles is that described in Borges de Sousa's [35] paper, where they generalise and say that all the vehicles involved in one task can be referred to as one Generalised Vehicle (GV). In essence the group of vehicles is assumed to be one vehicle with lots of spread out movable sensors. When collecting data from the vehicles for use in simulation, and also when doing HIL and HS, it maybe simpler to have one vehicle representation and instead have lots of different sensors. This may also lighten the computational load on the system. However, it does raise the issue of scaling up HIL and HS to work with multiple vehicles at anyone time, such as Multiple HIL (MHIL) testing and Multiple Hybrid Simulation as well as just multiple simulation. Papers [32, 22, 4] discuss their implementations of this concept. However a fuller solution is a modular environment which is capable of simulating lots of vehicles, and the simplest way of doing this is by running multiple simulations over many computers, each acting as one agent and communicating with a coordination module as well as other agents. As a result ARF's architecture must allow multiple vehicles, and the user interface has to be dynamic enough to allow switching between each vehicle, the best way being point-n-click *Command and Conquer* style (see section 2.4.4). Thus any architecture for testing must allow multiple entities and the simplest way to achieve this would be via a copy/paste mechanism which everyone who uses computers is familiar with.

The requirement for ARF highlighted by these papers is that the framework should be capable of simulating multiple agents in parallel, and under the same system, so that testing can be carried out on one computer. Each agent would effectively be a vehicle in the virtual environment and each one could have different sensing equipment, different dimensions and different manoeuvring capabilities in the same way as a single vehicle simulator might be configurable. This means that ARF needs to be very modular in design and have the ability to be distributed over many computers as simulation of multiple vehicles may require more processing power than one computer has at its disposal. This is mainly due to each vehicle requiring many different simulated sensors, and in the sub-sea environment these consist mainly of acoustic imaging devices, such as sonars, which require large amounts of computational power. It is imperative that there is an efficient underlying architecture that on which all sensors

are created and can be easily added to virtual vehicles for testing purposes. Research into this application using ARF is discussed later in combination with another system called DELPHÍS [30] *(See section 7.4 for more information).*

2.4.3 Mission Planning

Mission planning is important for AUVs, and can be part of the *user interface* and *deliberative* layers of the autonomous vehicle architecture shown in figure 2.3. Usually a graphical user interface is used to create mission plans, and as such is part of the *user interface* layer, however the mission plan being created then belongs *deliberative* layer.

Mission planning [36] is a very important process for all UUVs. Even with ROVs there might be control software on the surface, an autopilot, which drives the vehicle at some points in the mission (see SPINAV in chapter 7). Therefore some way of expressing the mission to the vehicle's control software, contained in the *executive layer*, is required, i.e. what the UUV is supposed to be doing and where. In order to create a mission for a vehicle the mission has to be written in a language that the vehicle understands and also expressed in a way the operator easily understands, i.e. is intuitive. The operator would benefit from an intuitive natural mission specification user interface where by they can see what they are telling the robot to do. For extra clarity the operator may want to simulate a mission, or playback a logged mission, so that they can see what the UUV will do, or has done. Mission planning is covered here because it has strong links with, and is often mixed with, simulation of the robotic platform for testing a mission. Therefore mission planning makes use of similar components and technology as HIL and AR techniques.

2.4.4 Computer Games and Mission Planning

The most intuitive form of expressing what the vehicle is supposed to be doing is by visualising the vehicle's environment and then displaying the mission executing in the virtual environment. Providing instant feedback gives the operator a visual check that the vehicle is executing the mission correctly. An intuitive mission planner or operator

control interface may take the same form as a "Command and Conquer" [37] game where by the user points-and-clicks a position on a map and the unit moves to that position, then the user can select a task to perform at this point. If the user's actions are logged, the log could then be converted to a mission script for a vehicle's control software. This mission could then be executed by any capable vehicle at any time. The user could select tasks for each vehicle to do in a similar way to the way a player would specify tasks to each army unit in Command and Conquer, e.g. surveillance (patrol an area). Obviously the tasks would not be the same as in a *Command and Conquer* style game, but would instead be tasks which the vehicle could carry out. For AUVs these may include, a surveillance pattern (sea port security), inspecting a pipeline or simply mapping the local terrain/bathymetry.

The current generation of people are generally very used to playing all sorts of computer games and thus used to certain user interface designs which have become standard. This exposure should be exploited so that mission planning is very similar to playing a computer game. For example, a computer game is set in a 3D environment which is similar to the real world, but for simplicity the view point is usually from a bird's eye view as this is more intuitive for observing and controlling movements. However, if the user has to do a more complex task (e.g. actually drive a vehicle) then the user assumes a non-bird's eye view such as a first-person view. Figure 2.7 is an example of different views which have different uses. When using a 3D environment it can be made to look 2D by using a plan or birds-eye view. This highlights the requirement that mission planning could actually be carried out in the same 3D environment as the vehicle will be operating so that the operator can see exactly what their vehicle will do.

The implication of this is that if mission plans are best created in a 3D virtual representation of the real world, then it can be done in the same 3D simulation environment as mission observation, hardware-in-the-loop testing and sensor simulation. This would be ideal because not only could missions be specified in the same environment to which they are tested, but also the user can see how the real mission differs from the simulated mission afterward. Once the real mission is completed the user can visualise the new data in the 3D virtual environment in conjunction with the

Figure 2.7: Observation and Plan View examples

previous mission plans. In some cases this allows the next mission plans to be made more efficient.

2.4.5 3D Mission Planning

3D mission planning is nothing new as this has been already carried out by the Naval Postgraduate School (NPS) in Zyda's [20] paper. They have a mission planner which has been created to use the same modules as their robot which means they can plan missions with it and also monitor missions as well. However, this is specifically designed software for use with their own AUV and has no other use apart from creating their mission plans. The different synthetic environment modules that make up the 3D environment feed into the planner, but these are again separate and not easily extendable. For example no sensor simulation or robot-in-the-loop [24] testing facility is built into this software, although the environment could, if redesigned in a different way, be used for all these purposes. Instead each part, the movement simulation, the sensor simulation etc, all run separately and therefore do not all run from the same virtual environment. A new framework could be only for testing and simulation, but it could also be used for the creation of mission plans, viewing current live missions and replaying existing missions. New missions could be created more efficiently by superimposing current mission data in the virtual world for the user to see. Mission planning would not be the primary focus of the framework, but there is no reason why a generic framework shouldn't be able to support a plugin which allows

the operator to create missions using a birds-eye view, in a similar way to the 2D mission planning facilities of SeeTrack [38] which only provides 2D mission planning. Again this highlights how the components which are used for mission planning could be a subset of some of the components required for testing facilities.

The framework must be very flexible to accommodate this requirement, and to go one step further and allow the user to be able to invoke their own modules, be it mission planning or something else, at the click of a button. All modules should interact and be configured within the same environment to save time and allow clearer more well defined virtual scenarios to be created.

An ideal mission planner would allow an operator to create a mission by adding virtual objects with rough positions, and then manually fly the mission in virtual reality first e.g. if the operator wants the robot to take a particular route, the operator can fly it them selves and the software could record the way points. This is much more intuitive than entering in waypoints in a LLA/LLD (Latitude, Longitude, Altitude/Depth) because the operator could simply point-and-click or manually drive like a computer game. Many mission planners still don't provide adequate facilities for simulating and testing missions.

2.4.5.1 AUV Workbench

One such planner which does provide facilities for simulating and testing missions is the AUV Workbench [36]. AUV Workbench is built to interact with the *Extensible Modelling and Simulation Framework* [39] (XMSF) and *Distributed Interactive Simulation* (DIS) [40] protocols. XMSF is a set of open standards which allow different platforms to communicate in the same language, such as XML [41] and other associated world wide web technologies. The DIS protocol uses UDP/packet based broadcast communication to enable communication outside the AUV Workbench to other modules. AUV Workbench provides the user with the ability to do Mission Planning and simulation and some HIL testing (through the use of DIS).

For mission plans, it uses an XML mission description language which is generated by the user interface. This XML mission specification is then interpreted and can be converted to the individual AUV's own scripting language used by its subsys-

tems. This is very similar to a mission description language called BIIMAPS [30, 42] (described in section 7.4) developed at OSL. XSLT [43] processors can be used to interpret XML missions produced by AUV Workbench and convert them to other XML document types for use in other systems, such as BIIMAPS.

The AUV Workbench uses independent modules running separately to provide the different functionalities, in a similar way that the OSL used to before ARF, e.g. the simulated sensors run in one location, the hydrodynamic model in another and the mission execution, perhaps, on the real robot. This kind of setup works well, but when wishing to change configurations and different levels of mixed reality and virtual reality, its more useful if the user can configure it all from the same user interface. Otherwise it becomes tedious and the user has to learn how to configure and run each individual system, and this cripples creativity and the ability to rapidly prototype different scenarios.

The framework should encourage users to provide their own components, and provide the mechanism to configure and save those component states. One method of doing this is by using introspection techniques (see JavaBeans in section 4.1.3) to ascertain the properties and property types of each component. The best solution would allow the user to customize each component using configuration sheets within the framework. Therefore they don't have to learn how to configure and start each component individually. Configuration files should be automatically generated to abstract the user away from unintuitive scripting. However, if configurations do need to be edited manually they should be in some easily understandable human readable format such as XML. Thus all configurations for components should be done in one virtual environment under the control of the user, but the user should be abstracted away from writing configuration scripts manually. The main requirement being a single point of creation within a single virtual environment, which allows for systems to be easily integrated together.

Another key requirement that the AUV Workbench highlights is the need for external communication to many different protocols and resources. Integration with both DIS and XMSF technologies would be incredibly useful. However, facilities to communicate with other protocols, such as OceanSHELL *(discussed in section 4.1.5)*,

would also be useful. Therefore the framework needs to have some generic internal communication mechanism which can easily be bridged to other external mechanisms. In essence, the framework needs to be able to support the addition of user components which provide communication to other external facilities.

The more uses one can think of doing with a virtual environment, the more requirements there are for an architecture capable of supporting all conceivable applications. A solution is required which is capable of providing all the modules needed for a fully simulated virtual environment, which allows extension using user modules, capable of interacting with the virtual environment, for specific tasks. Half the problem with existing software is that it is a black box and only the inputs and outputs can be used which it allows. If a new output is required, it might be very easy to program, but the programmer may not have access to the source code for that particular module, meaning that a whole new environment may have to be created just because the 3rd party software is unalterable. Hence the need for an extendible framework where all components can be configured and inputs and outputs from the virtual world are easily harnessed. In theory an application such as the AUV Workbench would use some of the same components as a full HIL testing platform. Therefore, a mechanism for creating and configuring these generic components for the different uses is required.

The main requirement identified here is that any new architecture needs to support standalone modules capable of doing anything. However, there needs to be a way of connecting the inputs and outputs of these modules easily. Visual programming could provide the mechanism for this. Section 2.7 on Visual Programming explains how. The review now looks into how virtual environments are used for monitoring systems and discusses the similarities to other applications.

2.4.6 Remote Awareness

Monitoring of remote environments is used in many different applications e.g. monitoring the status of a wind turbine on an offshore field. Sometimes the way this information is displayed to the user is not always the most intuitive. Consequently some important piece of information can be missed due to it not being obvious or due to some misunderstanding of a warning code. For example, when monitoring

platforms in remote environments an operator would be interested in seeing where a warning originates from, i.e. what piece of equipment is producing it. The operator then takes the appropriate action immediately. The contrast is a monitoring system which just indicates an error in equipment number (x), the operator then has to look up what (x) is and by which time something could have gone more wrong. The problem with many safety visualisation systems is that they are usually hard coded and therefore difficult to extend when more pieces of equipment are added to the system, and as a consequence an upgrade of all the software is usually in order.

A virtual environment could be used with 3D shapes representing the pieces of equipment being monitored. This is therefore another potential usage for the framework. Real world data is fed into a virtual world to display useful information in much the same way as monitoring autonomous vehicles. In a modular framework, special components could be used which change their status in response to some external stimuli.

The framework could be used as the visual front end to some remote monitoring systems, e.g. A remote health monitoring system such as a RECOVERY [44] which diagnoses faults and tries to recover from them. RECOVERY is the first example of diagnostic software that runs on systems in remote environments, this fits into the *executive layer* of the autonomous vehicle architecture shown in figure 2.3. To display feedback in a virtual environment, a simple plugin could have status lights such as red/amber/green to display if anything is wrong with a piece of equipment. Then depending on what sort of equipment it is, the operator could then select it with the mouse and invoke some action on it.

As a result ARF needs to provide a user interface for interacting with objects within the scene, and the user interfaces of these objects could be linked to real remote objects. For example, information about weather buoys out in the sea, which measure tides, could be displayed in ARF by changing the sea height in places governed by the buoy's GPS positions. Another use could be valves or other mechanical equipment on an oil pipeline, these could be monitored and information about each section highlighted in the virtual world, e.g. each oil/gas valves current flow rate. There is nothing specifically different about how these situations use a virtual environment

from any of the other uses discussed. There merely needs a specialised user module which takes input from some external system which in turn controls the appearance of certain virtual objects in the virtual environment. This highlights the requirement that modules for the framework must be able to easily interact with and control certain aspects of the 3D environment. Therefore a fairly simple 3D environment needs to be used so that creating modules that interact with it is straight forward.

The feature to be extracted from this is that objects can be placed in the virtual environment to represent the real objects and which interact with their diagnostic systems to display status information to the user. Therefore plugins could be developed to display information in the framework for any fault detection architecture as long as the framework is easily extendable and highly modular.

The review now looks at existing frameworks that have been created to address similar problems.

2.5 Existing Simulation Testing Frameworks

Aside from normal HIL testing environments which are tailored to one platform, frameworks exist which allow virtual environments to be created. These generally rely on some distributed communication protocol where by the different modules of the framework can communicate with each other. This allows modules to be swapped easily and also allows them to distributed over many machines. There have been attempts to provide configurable and flexible simulation frameworks which allow many vehicle/environment combinations to be realised and tested. One of the more recent developments comes from the Nasa Research Centre in Fluckiger's [45] paper which discusses the Mission Simulation Facility (MSF) test bed. This paper discusses a lot of the points that have already been raised in this review about the limited flexibility of most HIL environments, and that new simulators usually need developing for each separate application. As a result the MSF is intended to provide a configurable/flexible framework for the simulation of many robot/environment combinations. They provide an architecture and communications platform which allows the interconnection and substitution of different modules. However, they don't provide the ability to

build/extend existing modules quickly or provide guidance in connecting components together, therefore its still not that intuitive to actually use it.

In Gaskell's [46] paper they are developing a terrain server for generating specific types of topography such as that of Mars. This server, along with other modules, could then be plugged into the MSF as part of the synthetic environment with a lot of work interfacing with the MSF's architecture. Having merely an architecture is not really enough since this doesn't help to create new scenarios quickly. A user interface is required which allows the user to select quickly and easily the components they need and save the configuration at the touch of a button. This is one of the main requirements for ARF (see Visual Programming in section 2.7).

In essence a framework is required for creating the actual simulation environment for the needs of the application. The framework would also need to provide the flexibility of MSF, and would therefore require a similar communications network. However, in ARF the communication protocol needs to be flexible, and easily swapped, as many systems use different protocols so making ARF constrained to one protocol (say OceanSHELL) would make others less likely to adopt the framework for producing their own environments. Therefore a good communications protocol needs to be provided and also a generic interface which allows for 3rd party communications protocols to be easily integrated. The MSF identifies that certain modules may need to be "Hot-Swapped" for more accurate high fidelity modules in certain circumstances. However, high fidelity modules are more computationally expensive and for this reason distributing the computation of the environment over many computers makes it more feasible and reduces the need for any special hardware. In some cases modules may already have been created in another language for another purpose and the designer may not want to have to port the module to run in a new framework such as ARF. Instead, in ARF the user should be able to create dummy modules which communicate, via some communication protocol, with the stand alone module running on a different system. Therefore ARF needs to be very flexible and allow the user to create modules for practically any usage. For this reason ARF needs to be quite generic and not impose many requirements/restrictions on 3rd party modules.

The MSF uses Unified Modeling Language (UML) [47] to define communica-

tion messages which are passed between the different components of the framework down the communication backbone. Another method is using XML to define OceanShell/JavaSHELL (See section 4.1.5 for more information) messages, which can then be used for communication between external modules. However, there is no reason that a UML based protocol could not be implemented for use in ARF instead of OceanSHELL, since both would be special components available for the user to use.

Additionally, the MSF is being constructed incrementally, therefore only simple versions of modules are being implemented first and then higher fidelity modules introduced later once the base system is working well. This is the same technique that will be adopted when developing ARF and for developers using ARF. Since providing the core functionality, class structure/hierarchy and interfaces for ARF is vital so that more complicated specialised subclasses of the base modules can then be easily added at a later date. ARF is not only going to be for simulation and HIL, but also for remote awareness, mission planning and live mission observation. Thus ARF's base layer and base components have to be more generic than the MSF in order to provide this flexibility. However many of the components used for one purpose should be able to be used for another, hence why ARF is being developed as it will help re-use of modules for different applications.

2.6 Existing Augmented Reality Frameworks

The idea of component based frameworks for AR applications are not new. However, the definition of AR and the idea of what a framework is have not yet been standardised leading to many different interpretations of what a framework is. Therefore research into other areas has been necessary to underline the point that a general purpose framework for creating mixed reality scenarios for potentially any usage does not really exist.

The idea of a flexible, extensible, generic framework for AR has already been written about [48, 49] and discussed. The main problem with both of these approaches is that the research has not been into AR applications as a whole, but rather the more specific field of wearable AR devices for human applications, or Augmented

Reality displays of real environments. The point of a real generic framework has been missed by these researchers because AR can be used for robot testing, or mission observation as identified previously, not just for creating Augmented Reality views of a construction site which is merely one tiny application. Another fall back of these systems is that they don't provide a good high end user interface for creating the actual virtual scenario which will be augmented with real data, or used to augment a real environment view with virtual data. They still rely on configuration scripts which have to be created manually. Clearly this doesn't lead to fast creation or rapid prototyping.

For example, previous research [48] describes a framework for creating AR environments. What this research actually describes is a set of protocols and hardware interfaces which allow for different arrangements of equipment in a room/area to be configured as an augmented reality environment. Components are designed so that they communicate over a distributed communication protocol thus allowing them to be placed anywhere, i.e. provides location transparency making it easy to put sensors anywhere in an environment. The framework discussed has many good points such as XML configurable components, distributed communications protocol, standard interfaces defined for Navigation sensors etc. The main problem with this framework is that all the components can only be used to create an AR setup for a human viewing the environment either through a wearable setup or through cameras. The components themselves could easily be used for doing tests with AUVs, robots and such like since a lot of the navigation systems and visual sensors are the same. However, the problem being that this framework assumes that you want to create an AR environment and doesn't provide the scope for doing the many other similar tasks discussed earlier. The configurable nature is something which ARF would adopt, i.e. using XML to describe how components interconnect and how they are configured. However, the user won't actually author these files themselves, but instead they would be automatically generated by the user interface which helps to guide the user when connecting the available components. This is something which has been totally missed in this research and is one of the main reasons mixed reality environments are expensive to create because it takes the user a long time to learn how to use the architectures

available.

This leads the review onto the next section of *visual programming* since it is now apparent that ARF will need to provide mechanisms for helping the user connect components in meaningful ways.

2.7 Visual Programming

A subject which has been briefly mentioned is that of guided programming or guided construction, such as the LEGO® analogy given in section 1.4.1. One of the main problems for software programmers and designers which inhibits performance is whether or not they know how to use all of the existing software modules. More often than not computer software modules are poorly documented and do not provide example implementations on how to use them. This is particularly the case when projects are on a tight schedule with little capital backing them, meaning that there simply isn't the time or the money to spend on creating nicely commented code, documentation and tutorials. This problem is thus self perpetuating since each time a badly documented module is used, the programmer spends so much time figuring out how to use the module that they then have less time to document their own code.

Poor documentation is merely one aspect concerning software development which decreases productivity. Another problem for the programmer is that of simply knowing which modules are available and their functionality. Documentation can be consulted, however, as discussed earlier the documentation maybe poor or non-existent. Even if it exists it can still be hard for the programmer to find out exactly what they have to do to use the module, especially if no sensible examples are given. When a programmer knows exactly the functionality of a module they can create a program to use it with great speed. This needs to be harnessed, therefore better mechanisms need to be in place to help the programmer better understand which modules interact and how to configure them.

The combination of problems discussed means that programmers spend a lot of time re-inventing the wheel because existing modules are hard to locate, poorly documented or impossible to use. This problem is rife when it comes to producing virtual

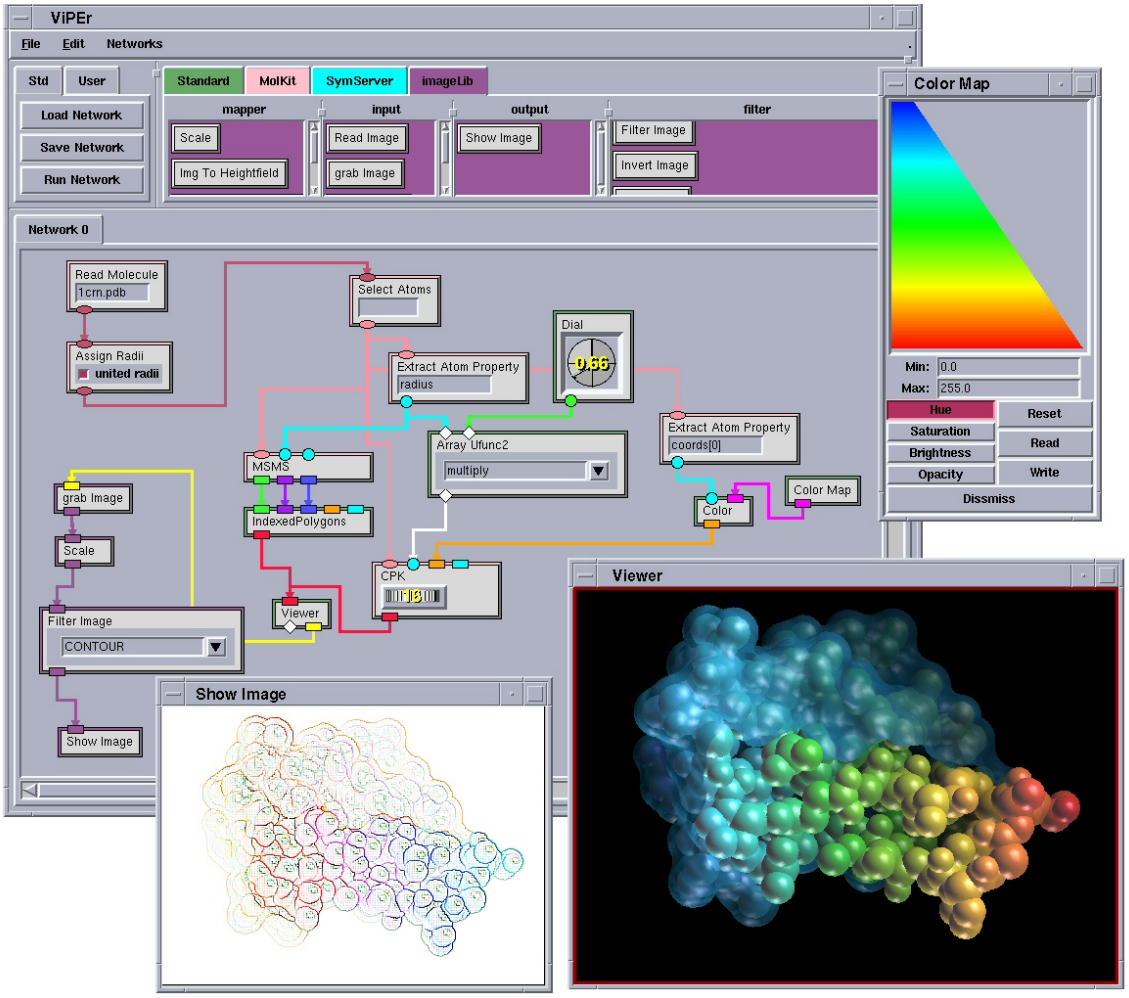


Figure 2.8: Michel Sanner's ViPEr - Visual dataflow system for structural biology

environments and simulated modules for testing robots, especially AUVs. This is usually due to environments being quickly "hacked up" to fulfil one purpose without considering the many other potential usages. Monolithic programming approaches then make reconfiguration and extension impossible limiting it to really only one purpose. More time spent making more generic modules with basic inputs and basic outputs in a configurable environment would make making changes later quicker and easier. However, when completely new functionality is required, even a configurable environment still has to be reprogrammed to incorporate new modules which can be difficult unless the environment is specifically designed to allow this.

Visual programming [50] provides some solutions to rapid module development and some ideas which can be harnessed to provide the basic idea behind a generic

architecture for creating virtual environments for AUVs. In general, visual programming is the activity of making programs through spatial manipulations of visual elements. It is not new and has been around since the very early 1970s when logic circuits were starting to be designed using Computer Aided Design (CAD) packages. Visual programming is more intuitive than standard computer programming because visual programming provides more direct communication between human and computer which given the correct visual cues makes it easier for the user to understand the relationships between entities and thus makes connecting them easier. Take the example of taking the ice cube trays out of the freezer and placing one in a drink. The human way of doing this is simply to look to locate the ice cubes, use the hands to manipulate the ice cube out of the tray and then drop it into the drink. Thus any interface which allows the user work in their natural way of doing things is going to make the job quicker. The other option, which is more like computer programming, is to have the human write down every single action required to do this, or use a keyboard to move the ice cube from the freezer. These methods are far more clunky and take much longer to do as its not obvious where to start. Therefore visual programming should exploit our natural instincts of how to move things around and join objects together, rather than writing code to do this. One such example of visual programming is ViPEr [51], shown in figure 2.8, which allows the user to visually connect components via data flows which then generate some visualisation. However, visual programming is not very useful unless components can be created very easily for use with the visual programming environment, and still remain standard programming objects which can be used in normal programming. Java is one such language which provides this unrestricting facility whilst still enabling visual programming. It is clear that if ARF is going to be very generic and be easy to use, it will require the use of visual programming methods to reduce reliance of documentation and speed up creation times.

This concludes the main part of the literature review. This chapter now continues but looks at technologies developed in Ocean Systems Laboratory and how they have influenced the research and design of ARF.

2.8 Ocean Systems Laboratory Technologies

This section covers the technologies that have been developed in the Ocean Systems Lab (OSL) as well as the other assets available. A brief history explains how the OSL has evolved and the implications this has had on the research towards ARF. The later sections discuss the main architecture behind all AUV technology in the OSL and how the new framework would fit into it.

The development of the platforms, discussed shortly, has meant that testing, simulation and observation facilities have been required in the OSL. The development of these platforms lead to the identification of a niche which this thesis addresses and many of the different requirements discussed in this literature review being identified.

2.8.1 Underwater Vehicles

The Ocean Systems Lab (OSL) has been developing underwater technologies in projects such as ALIVE, AUTOTRACKER which used the REMUS [52], Gavia [53] and Geosub [54] AUVs respectively. The OSL has also been developing its own autonomous platforms for its own research purposes namely RAUVER, NESSIE II, NESSIE III and RAUBoat. However, the OSL has other vehicles such as Hydroid's REMUS and HI-BALL which are also used for development purposes.

2.8.1.1 Nessie and RAUVER

RAUVER [55] is an AUV and has been developed for over 5 years. It has been through 3 major revisions. Its purpose is a test platform for all the technologies being developed in the OSL, and other institutions, when they become mature enough for real world trials. Figure 2.9 shows the RAUVER Mk2 and Nessie III. RAUVER was later stripped down and rebuilt so that the entire bottom layer disappeared. This was to maintain portability and keep the AUV below 200kg. However, for most small tests RAUVER was still too large. Therefore the idea came after the 1st Student Autonomous Underwater Vehicle Competition Europe [56] (SAUC-E) to build a very light weight hover capable inspection class AUV called Nessie [57] (see sections 7.2 and 7.3 for more information on Nessie's evolution and its usages with ARF).

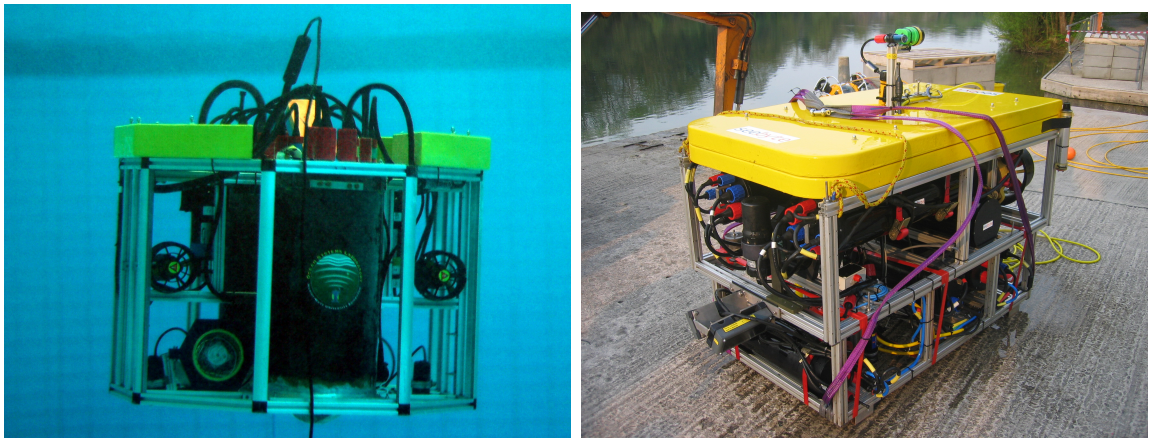


Figure 2.9: Nessie III and RAUVER Mk2

After Nessie I came Nessie II and Nessie III. Nessie III the newest addition to the operative AUVs of the OSL. The OSL also has a survey class AUV built by Hydroid called REMUS [52] (See figure 2.10). With both survey class and inspection class AUVs the use of multiple robots collaborating together is possible (see section 7.4) since all the AUVs have different capabilities meaning that cooperation between them is essential. The OSL has one autonomous boat called RAUBoat. The name lends itself to its inner workings as most of the technology came from RAUVER and is in essence an amalgamation of technology from RAUVER and NESSIE. The idea of RAUBoat is to provide surface surveillance and tracking of the AUVs. The second purpose is that RAUBoat act as a communications gateway from radio to acoustic and vice versa. Thus providing much shorter range acoustic communication and far more reliable data transmission from AUVs to the surface. RAUBoat will also be able to carry payload sensors such as sidescan and bathymetric sonar and underwater video cameras (see section 7.5 for more information on how this is used with ARF).

Figure 2.10: REMUS AUV

2.8.2 Implications for ARF

The OSL has evolved and now has many different software and hardware projects spread across the many researchers. There are only limited resources to do real world testing of the AUVs. The OSL has the benefit of two test tanks at its disposal; one tank which is a 3x4x2(LxWxH) metres and another wave simulation tank which is roughly 8x8x5 metres. The access to these resources is not always available, and this is only the first stage in real world testing. Consequently there is not enough real world testing time for hours and hours of debug. The fiscal cost of running tests in tanks rather than at sea can also be huge and consequently budgets won't stretch to large amounts of real world trials. For these reasons it is imperative that software be integrated together and all known bugs eradicated before any real world trials commence. Even known software problems can cause large amounts of wasted time because the AUV still has to be recovered to find out what the problem was. Then there is the setup time to do the same mission again, thus wasting large amounts of time. Even known software problems should be protected against and tested to make sure they cannot cause a mission fail. This is one of the reasons Nessie was developed as it provides a low cost test platform which can be easily used for real world trials. However, even then the costs are still high enough to mean that time cannot be wasted due to inadequate pre-real-world testing.

A flexible testing environment which uses the same architecture as the AUV is required. Often testing environments have been created for specific purposes, however, they are generally monolithic in design and not easily extendible to new tasks. Thus when the task changes slightly, it takes hours to factor in new parts to the testing program. More time is often wasted creating new test environments when the time would have been better spent designing an extendable architecture for doing multiple types of testing. The idea for ARF came from this very notion. A virtual environment was required which could accommodate far more than mere simulated testing i.e. could accommodate all mixed reality concepts which could be used to do potentially anything provided the architecture was both flexible and extendable. ARF needed to be able to accommodate new ideas, new communication protocols and new robots.

OSL is moving towards autonomy in all domains and is thus not so fixated with

underwater vehicles but more collaboration between all types of land/sea/air autonomous vehicles. Consequently, to test these systems it becomes more impractical and expensive to debug in the real world as resources simply are not available at the huge scale required for the complex scenarios. Real world tests are always required, but the architecture should be solid before a project's system makes its way on to the real platform. So that when a new problem occurs it is not masked by many other problems, as the problem with autonomous vehicles is finding out what caused the problem. Since real-time communications is often unavailable this makes debugging intermittent errors an absolute nightmare. Every single alteration made to some code could potentially take another real world trial just to see if it is fixed. Therefore testing all the systems running together in the way they would on the vehicle (HIL) is imperative to making sure that the robot is safe to do an autonomous mission for real.

2.9 Summary and Conclusion

This has review has covered the main areas of AR and HIL testing techniques. Other related uses such as the combination of mission planning and simulation have also been discussed to highlight the common elements between this and other associated technologies. Currently, there exists no frameworks which provide the different granular layers and abstraction required to provide all these capabilities. Existing frameworks provide no intuitive help for the user and do not provide straightforward extension making re-design quite often necessary.

When taking account of all the points and requirements discussed in this section it is clear that a gap exists for the creation of a framework of components that from which all identified uses from this chapter can be realised without having to design a new architecture for each one. The framework will have to be structured in such a way that provides the flexibility so that it can be used for all of the afore mentioned purposes and possibly many more. A general architecture is required which provides the basic building blocks for all types of environment. Then, providing the base architecture is structured and well defined, it should be possible to add extra functionality by

extending the current framework without having to alter the existing structure of the framework. The framework will have to be incredibly modular and highly object orientated so that many different combinations of its constituent elements, and user made components, can be combined to produce a useful virtual environment.

As mentioned earlier, all of the existing simulators, virtual environments, mission planners and testing facilities all have one thing in common and that is that they all use some sort of virtual environment. Therefore ARF will have to be generic enough to cater for all these environments individually, or all together, to provide a full virtual environment testing, planning and remote awareness facility. A summary of the functional requirements identified by this literature review is given in section 3.1.

Chapter 3

Functional Design

This chapter describes how ARF will meet all of the requirements identified in the literature review. The requirements are collated and explained. Then the design of ARF is explained.

3.1 Functional Requirements

The requirements identified in the literature review can be categorised as features of the architecture and the goals needing achieving. For example, speeding up scenario creation times is a goal, whereas providing guidance to the user as to how scenarios can be constructed is a feature of the architecture. The features of the architecture hopefully help achieve the goals. Most of these features can be summarised as to the general implications for the design of ARF. Therefore, the requirements are listed, then the goals and then the features of the architecture are extracted and explained.

3.1.1 Collated Requirements from Literature Review

- 3D virtual environment for displaying data and creating a simulation of the real environment.
- Provide AR, AV and VR i.e. all stages of the reality continuum.
- Domain independent (land, sea, air, space).
- Support all defined working modes (OFFLINE, ONLINE, HIL, HS, OM, TR).

- Provide AR and AV concurrently (for purposes of HS).
- Support MHIL.
- Copy/paste functionality for creating multiple simulations quickly.
- Support Distributed Virtual Environment with sensors and simulations running on different computers.
- Provide faster/slower than real-time simulations (OFFLINE).
- Provide external I/O for communicating with external modules.
- Provide location transparent external communications (important for HIL).
- Provide generic *event based/message passing* internal communication mechanism so that external communications can be easily bridged.
- Support 3rd party external communication protocols (DIS, OceanSHELL).
- Support mechanisms for synchronizing the virtual and real environment.
- Support modules for specific input and output devices such as HMDs.
- Allow modules to render data from external modules into the 3D environment (RECOVERY, increasing situational awareness).
- Easily add modules to the environment (e.g. for mission planning).
- Potentially unlimited fidelity for simulated sensors.
- Provide intuitive 3D control interface to allow the user to easily find their way around and use different views.
- 3D environment must be simple for 3rd party modules to control and interact with.
- Provide ability to load existing 3D mesh data formats as 3D geometry into the virtual environment (X3D, VRML, 3DS etc).
- Provide help for the user as to how components can be connected and configured.

- Provide intuitive user interface for configuring modules.
- Auto generate the module configuration scripts allowing the virtual environment to be automatically saved and loaded.

3.1.2 Collated Goals

- Intuitive visual user interface.
- 3D environment easy to build and manipulate.
- Increase situational awareness for operators of remote platforms.
- The ability to synchronise simulated systems with real systems.
- Provide a way for specialised modules for telepresence to be able to interpolate between sporadic communications from the remote environment.
- Modules easy to write and straightforward to integrate with ARF.
- Increase speed of virtual scenario creation time by guiding the user and reducing the need to consult documentation.
- Intuitive to connect the inputs and outputs of modules together.
- Intuitive to configure the module parameters so the user can't make mistakes or have to manually write configuration scripts.
- Support incremental development of the virtual environment alongside the project for which it is being created. Allow it to evolve at the same rate as the technologies it is being used to test.

3.1.3 Required Features

The features required of ARF can be defined as follows:

1. Generic - provide a generic low-level non-restrictive modular architecture. The lowest level of ARF must be generic in that it is not specific to underwater environments or robotics but is merely an architecture for building and constructing

scenarios which can be used for potentially any purpose i.e. no restrictions on what a component can or can't do.

2. Extendable - new components easily created and integrated with ARF i.e. able to build on the old, thus no re-inventing the wheel.

3. Flexible - although ARF is designed with testing of robotic systems in mind, it should not be restricted to only this i.e. the low level components could be used for many other purposes as well.

4. Distributed - able to communicate data to/from the real world systems allowing for various real and virtual data to be mixed for AR and HIL applications.

5. Guided - provide guided construction mechanisms which help the user choose how components are connected in order to construct scenarios more quickly and easily i.e. the analogy of LEGO®. Thus, alleviating the need for large documentation.

6. Intuitive/Easily Configurable - ARF should provide intuitive interfaces for configuring components and manipulating and using the 3D virtual environment.

7. Straightforward development - no new languages invented, the burden of development for the architecture is then eased; the developer requires no new skills.

8. Portability - varying uses, means varying systems and consequently different platforms will be running different systems; therefore ARF needs to be able to run on those systems.

9. Varying granularity of complexity - ARF must provide high level constructs for the end-user and also low level programming objects for the developer. ARF should support development at potentially any level, since if a component does not exist, the programmer can drop down a level to the programming layer and create it.

10. Polymorphism - allow components to have multiple interfaces so that they can be connected in many different ways for different uses. These interfaces could

be at a programming level or at an internal communication level i.e. accept many different sorts of internal communication types (event types).

11. Generic communication system - provide a generic internal communication system which can be easily bridged to external communication to the remote environment by 3rd party components which implement other protocols, such as OceanSHELL.

The generic nature of ARF will provide the flexibility. However, without extendibility the flexibility is short lived. Many of these concepts go hand in hand. It is important the design of ARF does not restrict either of these requirements as then the system could become obsolete due to the lowest level not being generic enough. It is difficult to never specialise a system for one purpose, however, specialising in ARF should merely be a case of producing specialised components, and therefore not restricting the architecture itself. A lower level of granularity needs to available for the programmer to create different architectures within ARF. Thus ARF will allow creation from the lowest level to the highest level. How this is actually implemented is discussed in detail in chapters 4 and 5. However, this chapter provides the functional design of ARF and how it achieves the requirements. It doesn't consider the technologies required, but instead outlines the required functionality of any technologies needed. The features identified are demonstrated through the use of specific testing and use test cases which are discussed in chapters 6 and 7 respectively.

3.2 Architecture Overview

The section provides a conceptual functional diagram which illustrates how the required components will interact within ARF to meet the requirements described earlier.

3.2.1 Component Interactions

Figure 3.1 shows how required components should interact. This diagram shows some of the components required by ARF to achieve the requirements. This is a general

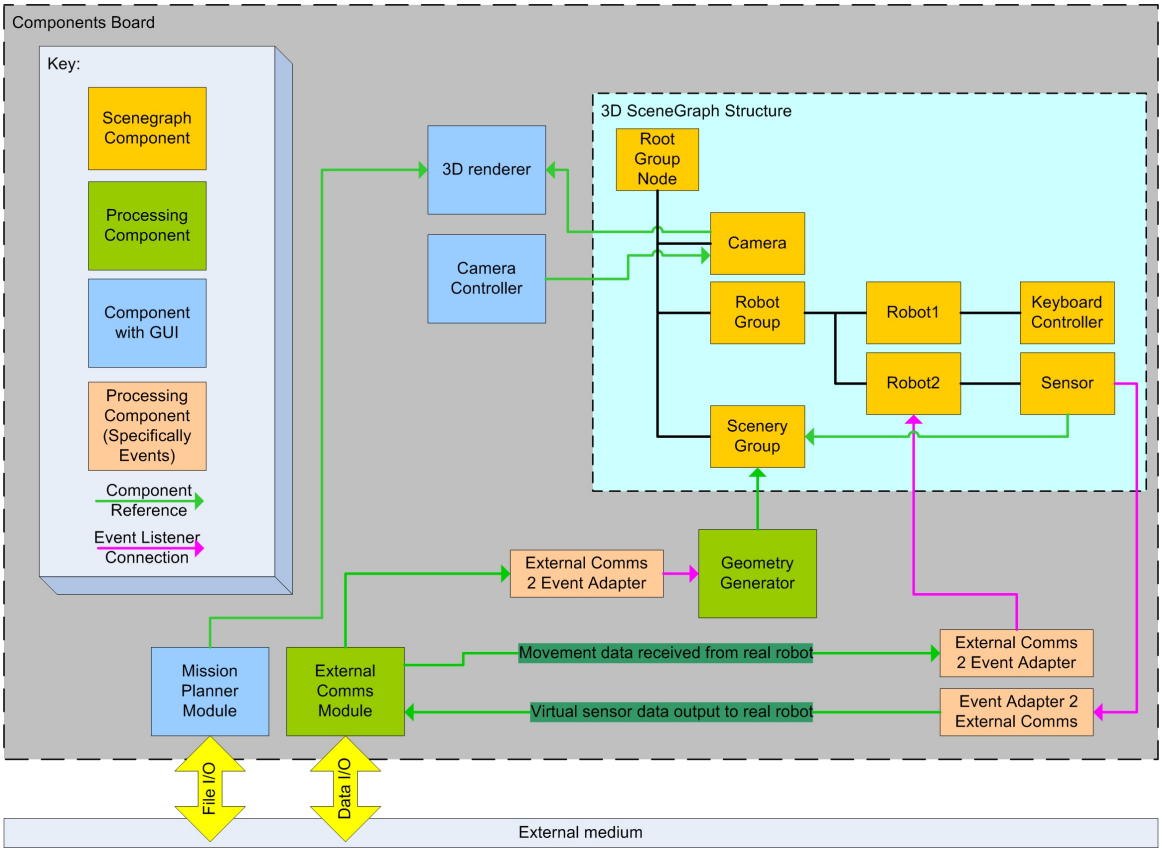


Figure 3.1: Entity Interaction Diagram

diagram which illustrates the types of interactions between the components and the structure used by ARF. To begin with, it should be noted that all the square boxes represent individual components. ARF will require that each component specify:

- Editable options - e.g. on/off, or visible/invisible, or colour etc.
- Component connections - what components are required as parameters for this component.
- Event listener connections - which event message types this component outputs. This will be used by the user interface to register other components as event listeners to this component.

The internal communication within ARF will either be via event messages, or via direct component reference. For example, a component will be able to use another component as a parameter and thus make use of that component in some way.

The diagram illustrates the different types of internal communication using arrows. The key demonstrates which colours of arrows and boxes refer to which type of component or communication type.

The diagram illustrates that there are 4 different types of component. In fact, there are only 2 distinct types of component: components which can be added to SceneGraph of the virtual world, and components which can't. ARF will use inheritance and polymorphism of components to provide the flexibility required. All SceneGraph components must be a subtype of a SceneGraph component (leaf, group etc). All non-SceneGraph components can do anything they like. However, ARF will use type interfaces to allow components to be recognised as more than one type. For example, some components may have graphical properties and need to be displayed on the screen, therefore these must inherit from an interface which specifies that they are graphical. Some components may implement event listener interface types; these allow the component to be identified as an event listener for a specific type of event. This information is used by the user interface to provide guided construction.

The diagram shows that any component can require a reference to any other type of component. Some components will require a reference to other components in order to control them and pass them data. For example, the *camera* SceneGraph component has a reference to the *3D renderer* so that it can send the information from the 3D environment to be displayed. The *camera controller* also needs a reference to the *camera* so that it can move the *camera* around the 3D environment in relation to movements made by the computer's mouse.

Sensors which simulate some real life sensor usually need to interact with the virtual environment. Therefore most sensors will be components which add to the SceneGraph so that they can sense the surrounding environment. This is why on the diagram the *sensor* has a reference to the *scenery group*. The *sensor* is thus slightly optimised because it will only sense the virtual environment which is connected to the *scenery group* sub-tree of the SceneGraph. This will allow the user to select which datasets each sensor will use.

The contents of the SceneGraph can also be made to be dynamic, i.e. driven by some external stimuli. For example, external communication can be implemented

by specific components. These components may listen to internal event traffic and convert this to messages which can travel on some external medium. Therefore components may also be used to merely provide a gateway to some external communication medium. This then requires other components which communicate directly with the gateway component. This is illustrated in the diagram by the *external comms module*. The *external comms module* would maintain a list of registered receiver components. Every time data is received it is passed to the registered components for processing. The user interface will allow the user to register components of correct interface type as receivers. For example, the diagram shows 2 registered receivers with the *external comms module*. These 2 components convert the incoming data to specific event types which are then sent to the registered event listener components. In this case the 2 registered event listeners to the two different converter components are *robot2* and *geometry generator*. As events are received by the geometry generator they are converted into virtual data which is rendered in real-time to the *scenery group*. At the same time the *robot2* moves around the virtual environment upon receipt of a movement event. Thus the event listener interfaces used by both these components might be a *GeometryDataEvent* and a *NavigationDataEvent* respectively.

Data is output to external communication mediums in a similar way. A special event receiving component converts that event type to data which is then sent to an *external comms module*.

This is merely an example of functionality which could be implemented with this style of architecture. ARF will provide the mechanisms for connecting the component references, registering event listeners and configuring component options. It will also manage which components are available and allow the user to import new components to the repository. The way it will do this is described in the next section which describes the user interface design.

3.2.2 Interaction with the User Interface

ARF will provide the functionality required by providing the means for connecting and configuring any sort of component. ARF will have to know information about the types of each component and what its options and connection requirements are.

This information will be inferred by examining the code of each component using introspection techniques, since no more requirements should be placed on the way components are programmed than necessary. Exactly how ARF does this is described in detail in the "Implementation Design" chapter 4, since it is quite specific to the technologies chosen to base ARF upon.

Figure 3.2 shows an overview of the user interface and how the various constituent parts will interact. The reader should notice that the components displayed on the *components board UI* and *3D SceneGraph structure UI* are the same components as discussed in the previous section. This diagram shows the different user interfaces required. There exists user interfaces for the SceneGraph and component board. There are also 2 other user interfaces: *graphical component display pane* and *selected component configuration*.

The *graphical component display pane* will allow for components of graphical type to be made visible by adding them to this pane and rearranging them. The *3D renderer* component and *camera controller* are graphical components. The 3D renderer is required to be displayed in order to see the through the *camera* component into the 3D virtual environment. The 3D virtual environment is added to, and controlled using the *3D SceneGraph structure UI*. This will allow the user to add specific SceneGraph components to the virtual environment, such as 3D geometry, robot simulations, animations etc. The *components board UI* maintains a list of all components currently being used within the project. The user can select components either here or on the *3D SceneGraph structure UI* to gain focus of their properties, component connections and event listeners; these are shown in the *selected component configuration* pane on the right of the diagram.

The *selected component configuration* is where ARF will provide intuitive guided construction. ARF will look inside the components and extract properties which the user can configure. This pane will show which connections are required to other components, various properties for the component and also allow the user to add event listeners of correct type. This is where the true force of ARF will lie. It will be able to show the user options as to which components can be connected, and which ones can be created if none are currently being used; the same for event listeners. The low

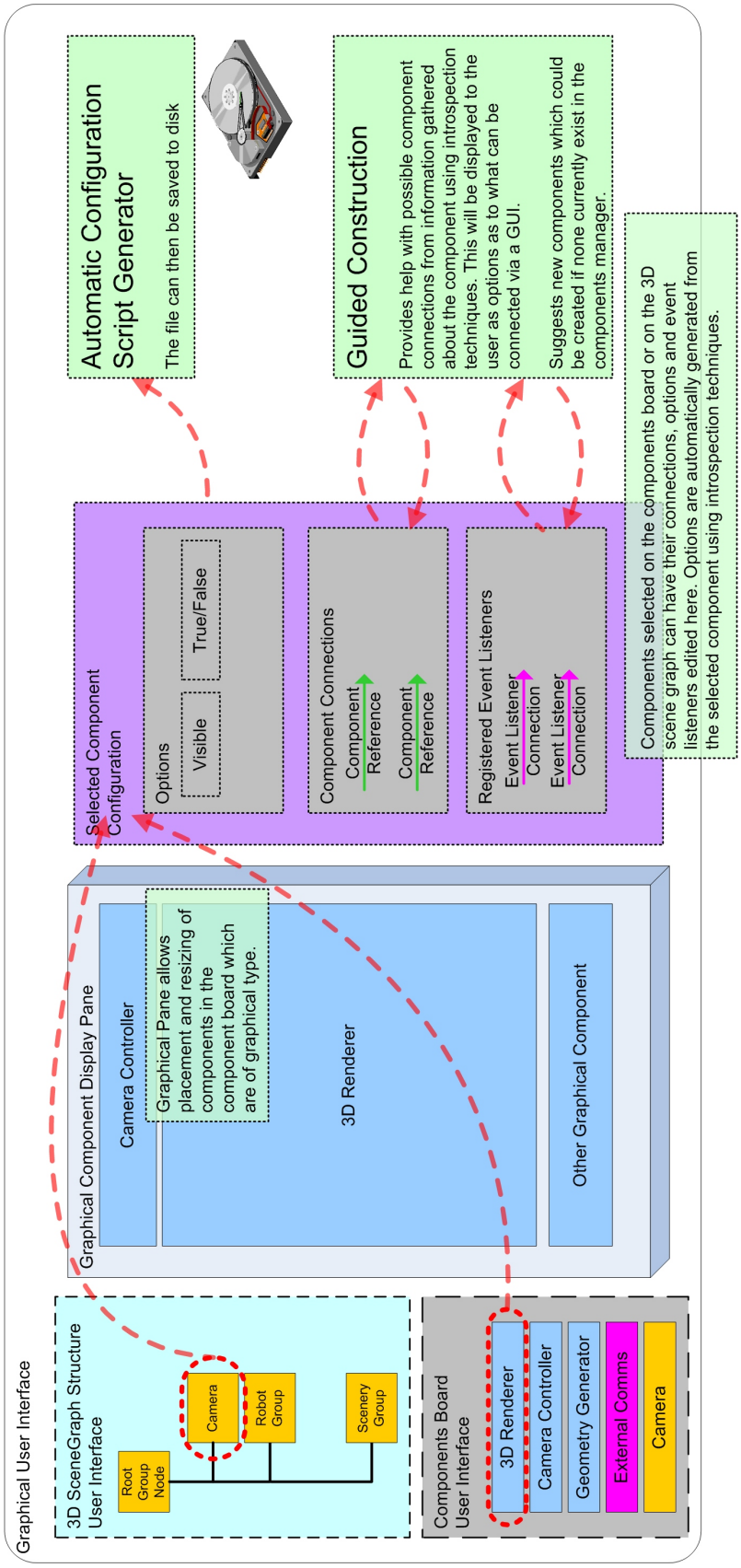


Figure 3.2: User Interface Interaction Diagram

level architecture ARF provides will allow many different uses through the evolution of new components. However, the implementation specific problem is how ARF will provide *Guided Construction*, and how it will automatically configure components; therefore abstracting the user away from ever having to manually edit configuration files. Everything will be contained within one framework allowing completely different scenarios to be loaded and saved quickly. The low level techniques used to provide guided construction are discussed in chapter 4.

3.3 Summary

It should become clear that ARF itself does not really need to have any particular focus on remote environments, communication protocols, virtual reality or HIL, since the architecture itself will be so flexible and generic that it can be used to configure and connect components for any particular usage. The design of the components and their interface types is what governs what they can be used for. ARF merely acts as the guiding hand when building the scenarios. However, ARF will be providing components such as the 3D virtual environment, sensor simulations and various external communications protocols to act as a starting point for the development of many different applications. The available components will evolve alongside the projects that they are being developed for. Due to the nature of this work ARF will provide many components for testing of robotic platforms and other associated augmented reality applications.

This chapter illustrated the functional design of ARF but did not elaborate how it was going to perform some of the complicated tasks, such as introspection, required to provide the generic and flexible nature required. This is because these techniques are largely dependent on the choice of the technologies which ARF will be based upon. The technologies used are discussed in the next chapter on implementation design.

Chapter 4

Implementation Design

The design of ARF is based on the requirements of many technologies in the underwater robotics domain, and more specifically the testing of these underwater robotics systems. It is important to keep in mind the reasons which are driving for the research and development of ARF, since these are the primary requirements. The literature review covers many diverse areas which all have links to underwater robotics and all have their own problems which need solving. However, most of these requirements are secondary and will in part be inherently provided by the modular design of ARF discussed in chapter 3. Therefore the key problem that ARF will address is providing hardware-in-the-loop and mixed reality testing scenarios for various different platforms. The main purpose of ARF is to reduce real world testing time, and consequently costs, by providing the ability to test modules, systems and platforms more easily and quickly than producing individual testing platforms for each system.

Providing an architecture for creating the scenarios is only part of the problem. The problem of reducing real world testing time of underwater problems relies on having ARF available to use, but also requires that the time spent creating the testing scenarios, and components needed for them, is also relatively small in comparison. Otherwise so much time is spent creating the virtual testing scenario that it could cost just as much as doing the real world tests. Therefore, mechanisms are required which guide the user as to how components can be connected, so that using ARF is much faster than manually programming the components together using an Integrated Development Environment (IDE). Various techniques using introspection are used to

provide information and guidance to the user in the form of "options" (See section 5.1.5 for more information on how this is actually implemented). Introspection is where actual programmer's source code is inspected to see which fields and methods can be identified as editable properties of that class and also what connections a class may require to other classes. However, in order to understand how this is achieved some background is given on the technologies which ARF will use.

This chapter covers the design of the architecture based on the technologies chosen. Chapter 5 describes the actual workings of ARF from a user interface level and from a programming level. Therefore this chapter discusses the design of ARF using the technologies which are available, without getting into low level detail of the code required for the mechanisms involved.

To begin with some background is given on the chosen technologies required for ARF.

4.1 Review of Required Technologies

This section looks at the technologies which would be useful in providing a generic base layer for ARF's architecture; firstly, the 3D virtual environment, then a more in depth discussion of JavaBeans is given since this is the technology which ARF will harness to provide the extendibility and flexibility demanded by the many different applications. A distributed communications protocol called OceanSHELL is then discussed since this will provide the location transparency required by so many mixed reality applications.

4.1.1 Why Java?

ARF will need to run on many systems; Java [58] offers portability of code which will enable ARF to run on any system capable of running a Java Virtual Machine (JVM). ARF also requires that it be able to peek inside the user created components and extract their properties; this is known as introspection. Java is perfect for this due to its Bytecode compilation layer. Bytecode is still easily interpreted and maintains all method names and type names. Therefore information can be easily extracted from

a Java class file without the programmer having to provide a special user interface to configure their component. In order for ARF to be widely used it must remain easy to program components for and not place lots of restrictions on the programmer. Ideally the programmer should be able to use existing modules with little modification to make components for ARF. More on this is discussed later. The other reason for choosing Java is that due to its flexible polymorphic nature, it allows classes to appear as different types through the use of interfaces and supports inheritance. Other languages such as C++ do this as well, but as mentioned earlier there is no way of introspecting C++ code due to its very low level compilation layer.

4.1.2 Java3D

Java3D [3] is a platform independent 3D language based on Java. Java3D uses a SceneGraph structure which is rendered using OpenGL or DirectX depending on the platform. Java3D allows the user to easily create nodes which can be added to the SceneGraph. These can include geometry, behaviour, sound and transform nodes. Java3D provides a feature rich set of libraries for the user to use. These allow for collision testing, picking using ray tracing and bounding shapes; these techniques are very useful for sensor simulation. Due to Java's extendable architecture, the user is able to subclass Java3D nodes and thus create their own custom implementations. ARF will provide many useful components which can be added to the virtual world by the user. The main advantage is that the SceneGraph structure is easy for the user to visualise and is precisely what is required by ARF's functional design (an example Java3D SceneGraph is shown in figure 4.1).

In addition to the flexibility of the Java3D architecture there exists many 3rd party developments which provide even more Java3D nodes and thus increase its usefulness. These include 3D Mesh loaders for objects and terrain of many different types. One such 3D mesh loader is X3D [59], or its more commonly known ancestor VRML [60]. X3D is based on XML [41] which makes it very expressive and easily usable. X3D is already used by the Naval Postgraduate School [20] in their AUV workbench [36] for displaying the virtual world and consequently there already exists vast amounts of X3D mesh data for scenery, buildings, AUVs and other real world objects. X3D

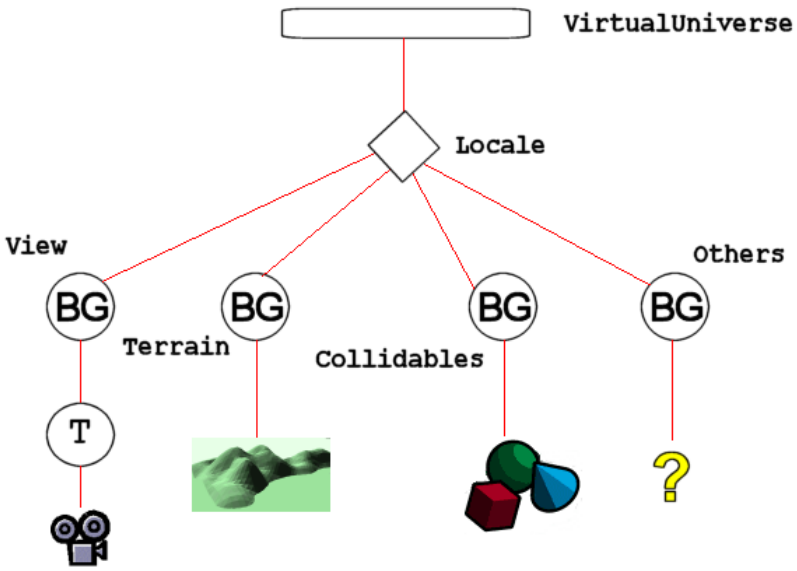


Figure 4.1: Example Java3D SceneGraph

provides a SceneGraph structure almost identical to Java3D. Thus X3D is easily represented and loaded onto a Java3D SceneGraph using the Xj3D [61] file loader. X3D can also be connected to the DIS simulation protocol in order to display external data, such as changing positions of objects. Thus having the ability to interface to DIS through the Xj3D loader would be an added bonus of using Java3D. The use of X3D allows for far more complicated visual scenary and animations to be loaded into a virtual environment and also allows for the environment's geometry to be created using another tool.

Apart from anything else, Java3D is very easy to learn for a Java programmer due to its SceneGraph and modular nature. This means that it is a very good architecture for providing virtual world extendibility for ARF.

4.1.3 JavaBeans

In section 2.7 visual programming was discussed as to how it can help increase the performance of software creation. Visual programming is a good idea, however, due to its visual nature it places lots of requirements on how a module is written and usually requires that the actual low level components themselves be programmed in a specially designed language. This leads to visual programming only being used

for more specific uses, such as connecting data flows using CAD packages. However, visual programming can become far more powerful if it places nearly zero restrictions on how the low level components are created. This means that in order for visual programming to be widely accepted it has to somehow be able to make use of all existing software components even if they are not designed to be used in this way. One such method of visual programming exists whereby components only have to implement a few simple "programming conventions" in order to be able to be used visually.

These special software components are called JavaBeans [2] and are based on Java. JavaBean visual programming tools work on the basis that a Java class object has been programmed adhering to certain coding conventions. Using this assumption the visual programming tool is able to use introspection techniques to infer what the inputs and outputs are to a Java class and then display these as identified properties to the user. Thanks to Java's bytecode compilation layer it is easy for a JavaBean processor to analyse any given class and produce a set of properties which a user can edit visually; thus removing all the need of writing code to configure the Java class object. In theory any software which adheres to software component theory [62] could be used, however, JavaBeans allows for properties to be identified rather than explicitly created by the programmer. Thus the programmer has little or no work to do when programming JavaBeans, as they are in essence just Java, as opposed to other special component based languages.

An example of a JavaBean visual programming tool is the *BeanBox* [63] which is what the JavaBean tutorial [64] recommends in for testing the user's own JavaBeans. However the *BeanBuilder* [65] is a full JavaBean visual programming tool. It allows the user to create custom applications using visual JavaBeans which are based on Sun's Abstract Windowing ToolKit (AWT) and the extended Swing Toolkit. These are generally GUI widgets and are used for creating more advanced GUIs. Non-graphical JavaBeans are also supported by the BeanBuilder and are wrapped up in a graphical widget so that they can have a visual position. All JavaBeans can be connected to one another's inputs and outputs using event listeners. The BeanBuilder allows the user to change the connections of these event listeners. Each JavaBean can

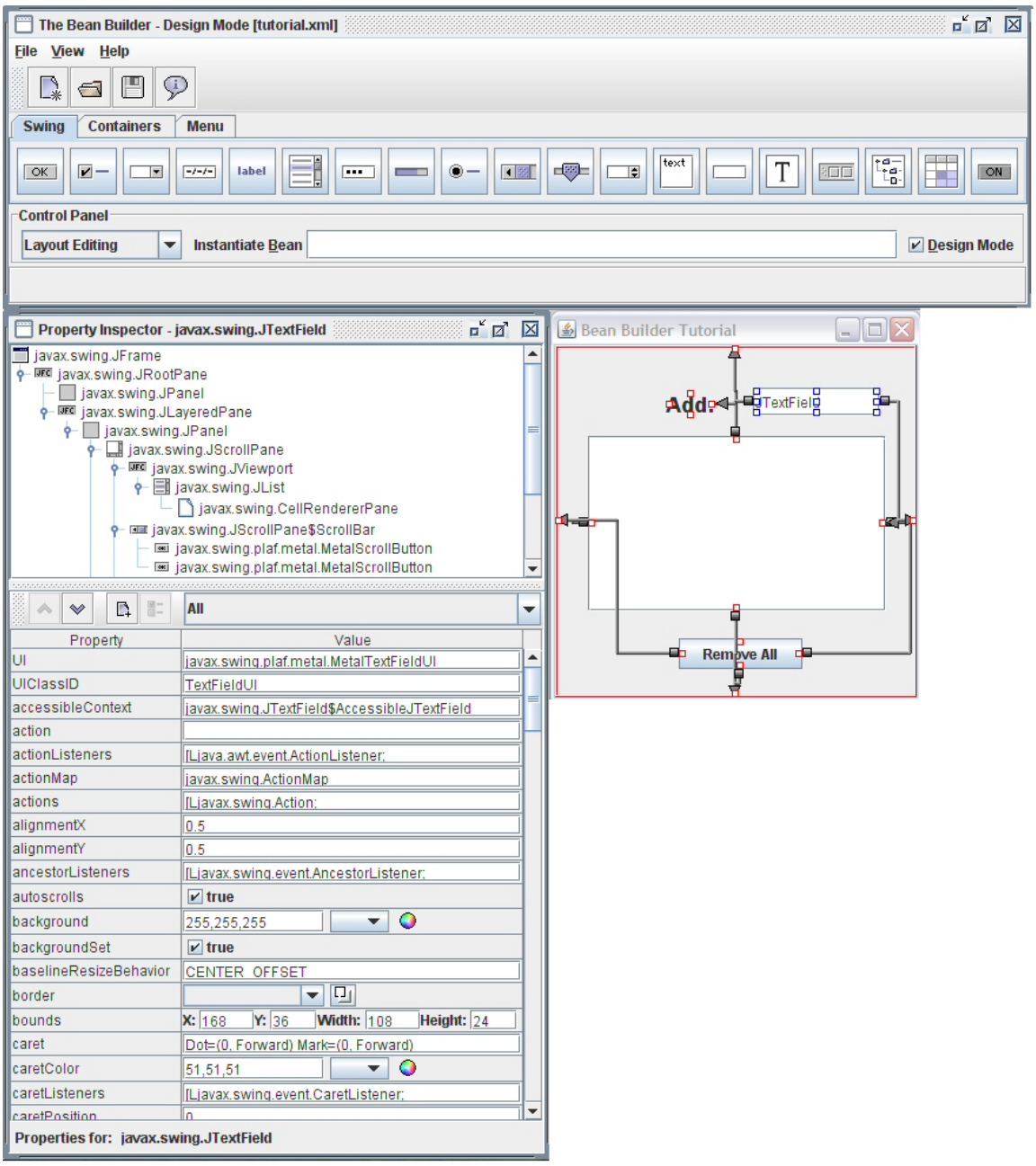


Figure 4.2: The Bean Builder visual programming environment

also be configured using a list of properties which are inferred from the JavaBean using introspection techniques. Figure 4.2 shows the BeanBuilder. On this picture the user should note a graphical design pane on the right and a hierarchical view of the components on the left. The currently selected component on the design pane has its JavaBean properties displayed on the properties pane below the hierarchical view. The property pane displays all the JavaBean properties in the form of a property sheet, each line refers to a different property. The JavaBean engine provides mechanisms for the programmer to match graphical property editors with the property types of the JavaBean properties. Thus for each property a graphical property editor is selected, and from which a list of editors is created and displayed as a property sheet for that JavaBean. The user is then able to use the editors to adjust the properties. It should be known that the property sheet generation is not part of Sun's JavaBean engine but is in fact part of the BeanBuilder program. Therefore it is up to the programmer of the property editor GUI to decide how they display these properties.

On the top of the picture the user should note a palette of buttons which refer to the graphical JavaBeans available for the user to add to their GUI. This representation can quickly get cluttered if there are too many JavaBeans and it shows all available components which doesn't help the user as some components cannot always be added to one another. Thus a more trimmed and refined set of options would be more helpful for the user when they are deciding what to place where.

4.1.4 BeanInfo Objects

In addition to using introspection techniques to gather information about the properties of a Java class, the programmer can also manually specify which are the key properties of the JavaBean. These properties are specified in a separate Java class called a *BeanInfo* object. The programmer may wish to do this if they want certain properties of a JavaBean to be kept hidden from the graphical user interface, but which they want to have methods which are the same as the JavaBean coding conventions (for use by other programmers directly using the class). If a BeanInfo object is present, the JavaBean introspection module will use the BeanInfo properties instead of the properties identified by introspection. The reader should read the *JavaBeans*

Trail [64] if they wish to learn more about the specifics of JavaBeans. However, the relevant parts of JavaBeans will be discussed where they are required throughout this thesis.

In summary, JavaBeans are incredibly useful because they can be created with minimal changes, if any, to the Java class. This means that as a visual programming platform, JavaBean environments are very easy to develop components for. JavaBeans are generally good for enforcing good programming practices as well since they force the programmer to think in an object oriented fashion, i.e. each object does a specific simple task but can be connected to other simple objects to do more complicated tasks. This helps enforce the generic nature of components which will be required by ARF.

JavaBean visual programming still has limitations. One such limitation is that guided programming is still in its infancy, i.e. giving the user options as to which objects can be used as parameters and which new objects could be instantiated if none currently exist. Therefore, looking at documentation is still necessity the first time using a Bean, or even identifying that the Bean is the right one. This is where this research fills a niche: improving guided programming of JavaBeans by providing the user with options for what they can do. This is useful when they have little knowledge of how particular software modules can be used, configured and connected to other modules (section 5.1.5 discusses the mechanisms ARF uses to provide guided construction).

4.1.5 OceanSHELL: Distributed Communications Protocol

HIL requires for real modules located anywhere to be swapped for similar simulated modules without the other systems knowing, having to be informed or programmed to allow it. The underlying communication protocol which provides the flexibility needed by the framework is OceanSHELL [6]. OceanSHELL provides distributed communications allowing modules to run anywhere, i.e. provides module location transparency. Location transparency makes mixed reality testing straight forward because modules can run either on the remote platform, or somewhere else such as a laboratory computer.

OceanSHELL is a C/C++ [66] software library implementing a low overhead architecture for organising and communicating between distributed processes. OceanSHELL's low overhead in terms of execution speed, size and complexity make it eminently suited for embedded applications. An extension to OceanSHELL, called JavaShell, is a portable version of OceanSHELL since it is built in Java [58]. Both JavaShell and OceanSHELL fully interact, the only difference being that OceanSHELL uses C structures to specify message definitions as well as the XML definitions which JavaShell uses. C structures are used in OceanSHELL to provide incredibly efficient casting of message types, however, this is less user friendly than the XML equivalent and requires for duplicate C structures which match the XML messages. OceanSHELL is not only platform independent but also language independent, making it fully portable.

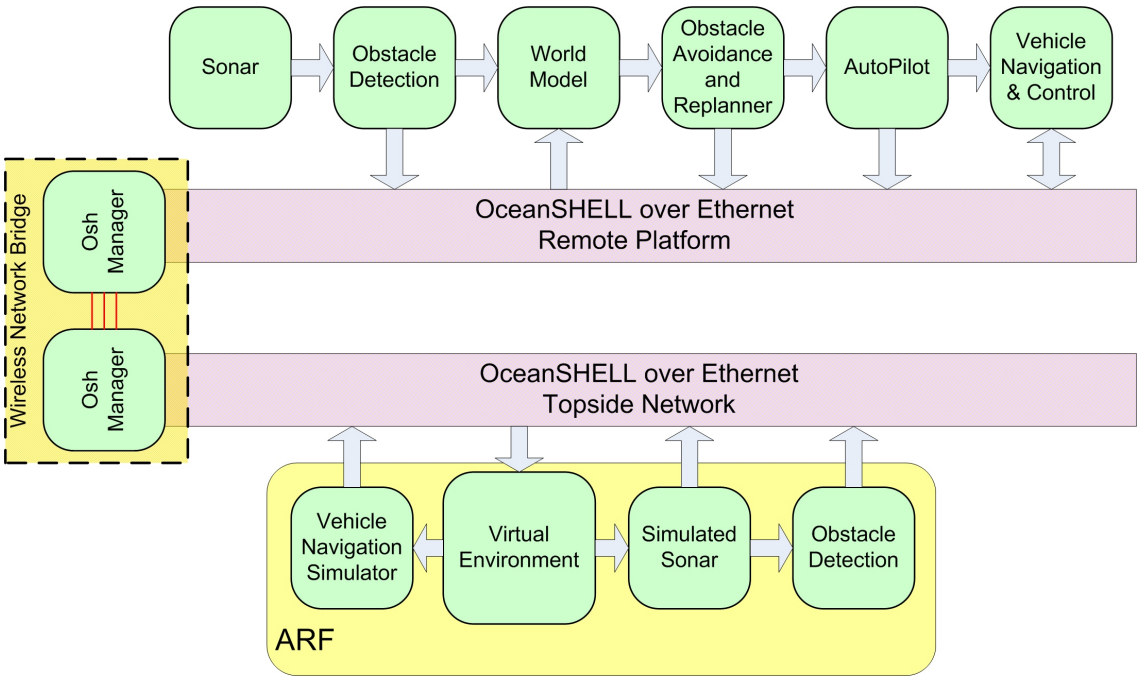


Figure 4.3: This diagram shows how OceanSHELL provides the backbone for switching between real and simulated (topside) components for use with HS/HIL.

4.1.5.1 Abstraction Layer Interface (ALI)

OceanSHELL provides the basis for embedded modules to communicate data to one another and also to higher level modules. Each robot then has its own message

definitions for information which needs to be transferred between modules. This information can be sensor information, status or navigational information etc. Since each robot has different capabilities, sensors and payloads it can be difficult to create software which is capable of working with any type of vehicle. Since there are many robots developed all over the world, there are many types of communication protocols. Even if everyone used OceanSHELL there would still be various different interpretations of what each message should have, e.g. a navigation message may contain a position and a rotation, however, another type may use GPS position in Long/Lat/depth where as another may use a local coordinate frame.

ALI stands for "Abstraction Layer Interface" and provides definitions of messages, and structure types within messages, in order to provide a base layer which every robot implements. A robot can be designed right from the beginning to work with ALI, or have a module which translates the vehicles control and capabilities to that of the ALI.

ALI defines standard interfaces for communicating with the robot. This includes the way to control the robot and ways to run missions. This means that any mission planning software only has to know how to use ALI in order to interface with many vehicles, similar to the DIS protocol used by AUV Workbench. ALI benefits means that ARF will require only one ALI interface module in order to do testing, simulation and online monitoring with many different types of underwater vehicle. Most of the ALI messages are domain independent as well, so can also be used for surface and air vehicles. ALI is very similar to Autonomous Vehicle Control Language (AVCL) [67]. AVCL provides a single archivable and validatable format for robot tasking and results that is directly convertible to and from a wide variety of different robot command languages. Therefore providing an add-on for AVCL would enable far greater scope for the immediate use of ARF.

4.1.6 Extensible Markup Language (XML)

Extensible Mark-up Language (XML) [41] is a mark-up language similar in form to HTML (Hypertext Mark-up Language). Unlike HTML, XML does not define any tag sets. XML only places requirements on the syntax of the document, allowing the

author to create their own tags where the names are meaningful to them. XML allows users to store data in a human readable and machine interpretable form. XML can be validated against an XML schema to check for errors.

There are many parsers for XML in most programming languages because XML is so easy to use and thus most programmers will have come across it. For this reason, XML will be used in ARF to store project information about component configurations and connections which define the scenarios created by the user. This allows the user to modify the project settings manually if they prefer, or if there is some problem. In addition to XML being so flexible, it is one of the ways of serializing the information stored in a JavaBean, i.e. a JavaBean can be automatically stored in an XML file. The settings of that JavaBean can then be easily modified by hand if needs be.

XML is also used for storing data about 3D objects. X3D [59] is an XML version of the VRML (Virtual Reality Modelling Language). OceanSHELL also uses XML as a means of describing the contents of messages which are used to communicate between distributed processes over a network. Therefore XML will be the method of choice for storing all information regarding scenarios created in ARF.

This chapter now looks at the novel concepts of ARF's design and also identifies how these concepts can be tested to show ARF's superiority over other methods of creating virtual environments. With this in mind the later sections look at how the technologies discussed will make up the backbone of the architecture.

4.2 Novel Concepts

A simple solution, but not very extendable would be to provide a set of generic components which can be connected in a variety of different ways to provide the testing scenarios required. This would fulfil some of the requirements enabling scenarios to be built quickly. However, the user still needs to know how these building blocks fit together. Further more, if the components are not generic enough then certain scenarios cannot be realised. This highlights the need for a architecture which provides multiple layers of generic granularity. This is required so that the architecture

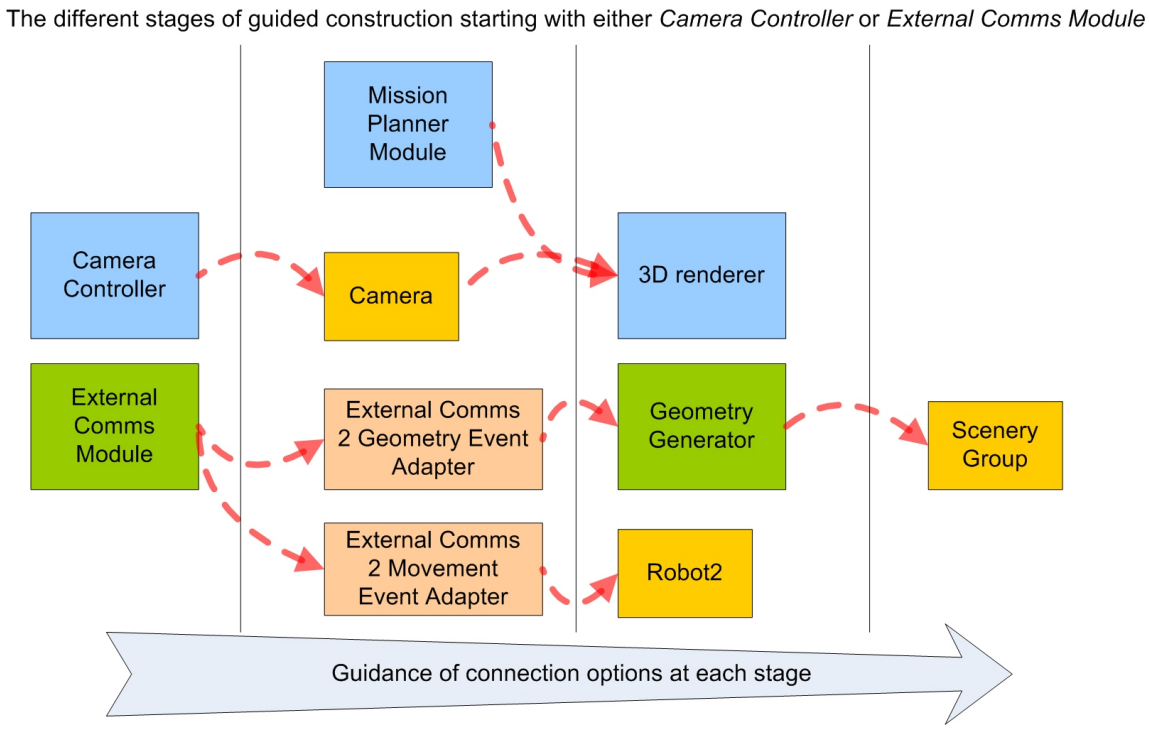


Figure 4.4: Example of the options guided construction can provide

is extendible and configurable on different levels of complexity, i.e. if the existing components do not provide a specific functionality and cannot be arranged in a different combination to perform that function, then there needs to be a simple way of creating and incorporating new components. Thus a programmer may have to drop down from the level of component connection to a lower generic layer which specifies how components interact and how they can be created. This then allows existing functionality to grow potentially forever. Without different levels of generic granularity, extendibility is compromised and thus it may be difficult to incorporate more functionality in turn leading to the development of yet another testing system which would defeat the point. Therefore, by having multiple levels of granularity in the architecture and enforcing conventions on the lowest layer, information can be extracted about the way components interact and this can be used to provide help to the user, of how to use the components, on the highest level.

Another way of looking at the problem is: How to provide guidance on abstract building blocks made by individuals which could have potentially any usage. This means that the architecture needs to know something about the building blocks which

the user doesn't. In order to be generic, and maintain extendibility, the architecture cannot place too many restrictions on components as this would mean it takes a lot of work to make components. Certain existing technologies must be harnessed which are already accepted and which provide the flexibility the architecture requires, i.e. which allow the architecture to peek inside the component and see how it interacts with other components, or, see what the component is made of an how it works, e.g. what properties it has. The architecture must be able to infer how a component interacts with another component so that this information can be passed on to the user in a way which they understand. This then becomes a representation problem. The information should be represented in such a way which allows the user to assemble the components in the way that they require, without having to have prior knowledge of the components and how they interact. If the user had a rough idea of a starting component the architecture could therefore guide them as to what their "options" are.

Providing "options" is key, since without options the user cannot make any sensible informed judgement, its just blind guessing. Providing options allows the user to try different combinations of component connections quickly and easily; enforcing the human standard of learning by trial and improvement. Figure 4.4 shows the different options guided construction could provide and how it evolves the further the user follows it. The same theoretical components are used from figure 3.1 in chapter 3.

If component names describe the purpose, and it has interfaces which describe how it can be connected, this information can be presented to the user in a meaningful manner, allowing them to connect components correctly without the need to trawl through pages of documentation as a programmer would have to. This enforces the concept of "a little bit of help goes a long way".

Layers of Granularity:

- Low Level - ARF's Java Interfaces and JavaBean coding conventions provide easy component creation.
- Middle - automatic generation of high level human readable XML representation of component configurations and connections.

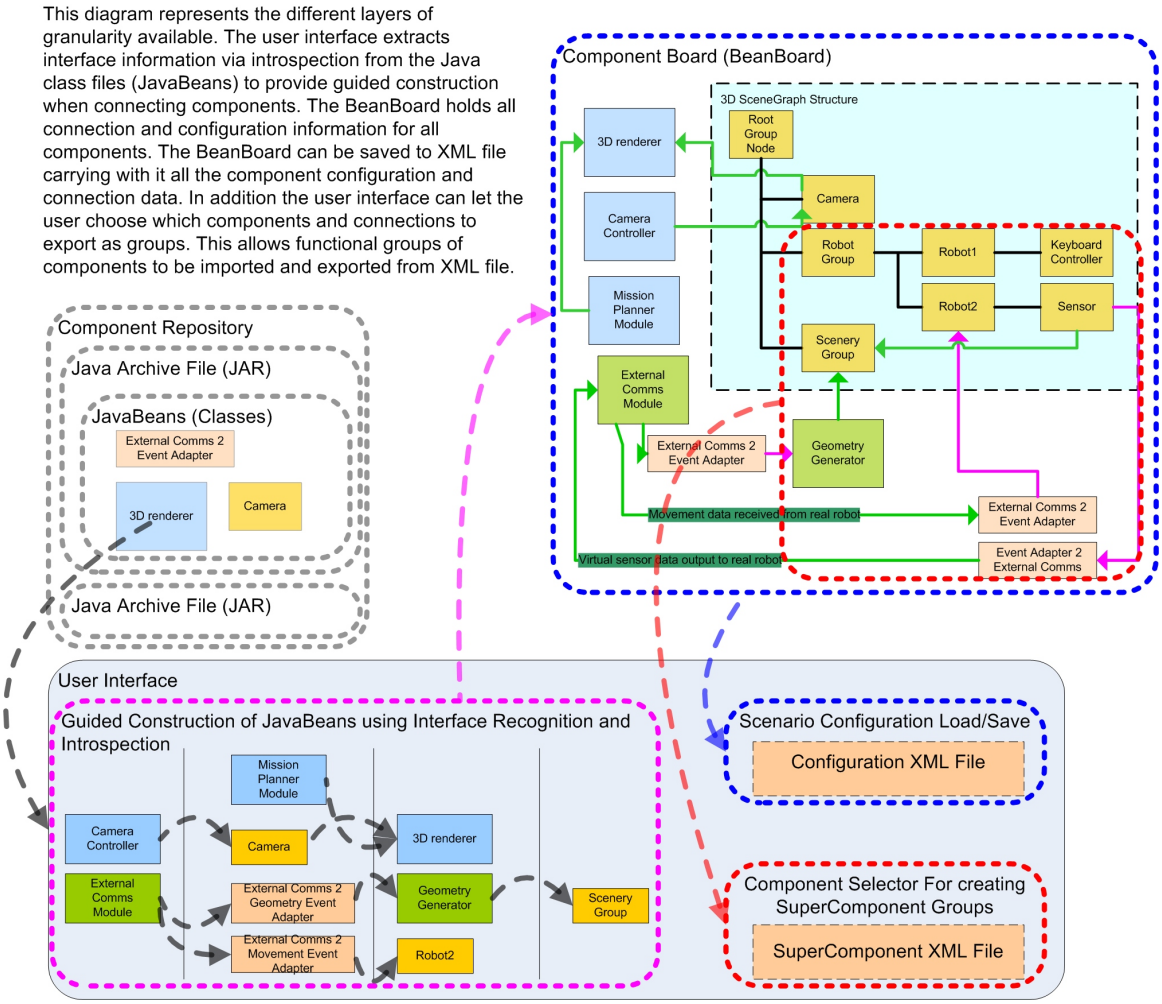


Figure 4.5: Layers of Granularity: viewed through the architecture

- High - introspection and Java Interface recognition to provide guided construction and thus aid the learning of the user.
- Top level - complex/SuperComponent creation. Allows multiple component configurations to be saved as a functional group which perform a specific task; allowing users to create specific names for configurations and groups of components.

From the lowest granular layer, a set of generic components can be created which fulfil many of the requirements for development and testing of underwater platforms. However, most these components can also be used for applications with nothing to do with underwater. Therefore any components written should be done so in a generic fashion which restricts the usage as little as possible. Consequently, each component is as flexible as possible. As the component repository of the architecture grows, so does the flexibility of the architecture. *Figure 4.5 shows the interaction between the granular layers.*

The most important aspect of the architecture is that it provides guidance on the highest layer by using introspection techniques to infer properties of a component. On the lowest level, coding conventions are imposed on any component created by the programmer. The more conventions they adhere to, and the more generic they make each component, and the more interfaces are used to communicate between components, the more flexible the higher level architecture becomes. If component properties use Java Interfaces as types, instead of actual Object types, it allows for greater flexibility of what can be used as a parameter and also makes creating new objects which can interact with others far easier. This is because Java classes can have many different types as they can inherit from many different interfaces. This polymorphism makes ARF more flexible and gives more options to the user. ARF can make inferences about property types and can then help the user by providing more informed options and provide some sort of on the fly help when constructing scenarios and connecting components, e.g. show the user which objects can be connected so that they can be easily pieced together like LEGO®.

One of the key aspects of Guided Construction using graphical property editors

to configure components is that it is very hard for a user to configure a component incorrectly. This is a definite advantage over standard programming methods. One of the most novel outcomes of such a generic method of connecting components is that useful component combinations for use in HIL testing, amongst others, can be quickly realised and used to carry out more thorough testing of a remote platform and as such detect and correct otherwise undetected faults before real world testing. This leads to a more mature and reliable platform. This is a key contribution of ARF, since without it testing facilities would not be created for such specific tasks. ARF's easily extendable generic architecture, with the help of guided construction, allow scenarios to be rapidly created for potentially any purpose.

4.2.1 Testing the Concepts

In order to measure the improvement of ARF due to its guided development nature, an informal study will be carried out, where users will be asked to create scenarios against the clock using existing components. A tutorial of ARF will be provided for background familiarisation. The time of creation using ARF will be compared to programming the same components manually using an IDE. The programming API for each component will be provided for the programmers reference. It is then the task for the programmer to figure out how it should be connected and to set the properties accordingly. The task will be timed. The next step will be to make a slight alteration to the scenario, and time how long that takes. Other tasks such as changing settings and creating different components with different configurations will also be undertaken.

The assumption is that the programmer is an expert in Java, so working as fast as possible, but that they will be a novice in ARF. The only experience of ARF will be from the tutorial. The tests should highlight how ARF can be used by a novice and still be faster than manually programming the components, since all the test subject is doing is creating the glue between the components and configuring them to some specification set out in the test. The IDE programming task will also just be creating the glue using the exact same components (Java programming classes). The test cases for each platform will need careful design to make sure it does not favour

one platform over the other. The final test will be for the user to create their own JavaBean component and integrate it with ARF to demonstrate the time overhead of using ARF. It should be noted that this is a one off fixed penalty. An outline of the testing metrics are given below, however, more information is available in chapter 6 on test design and the corresponding results.

Testing Types: To be completed on both platforms.

1. Construction Speed Test
2. Extendibility Speed Test
3. Alteration Speed Test
4. Saving Configuration, or multi-configurations speed test
5. Property Change Test - How long does it take to quickly adjust certain properties and run the program.

In addition to testing the performance achieved by the low level architecture, test cases will be demonstrated in chapter 7 to show ARF achieving all the requirements set out in chapters 2 and 3 such as AR, HIL, HS, OM etc.

4.3 The Architecture

This section describes the technologies which are going to be used to provide the functions of ARF's architecture. The actual implementation of components and the software interfaces used by ARF are discussed in chapter 5. However, ARF should eventually consist of components which are created by the user.

4.3.1 ARF Components

ARF will be based upon JavaBeans, as it is mature and provides many of the functions required by ARF. Creating JavaBeans is a straightforward process for a programmer and consequently porting real working modules into ARF components is easily done. JavaBeans will provide the low level back end to ARF and provide the introspection

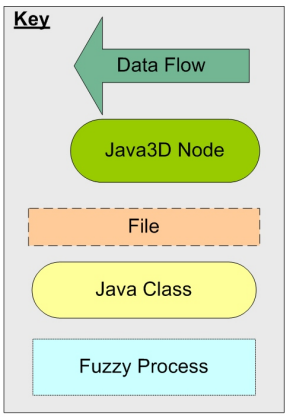
techniques which ARF will use to identify user editable properties in the components. Each property of a JavaBean has a *Java type* which is identified and cross-referenced in the property editor database in order to locate suitable GUI property editors that match the *Java type*. All this is provided by JavaBeans. However, the property editors themselves, and creating the property sheets for the user to use is not provided. Therefore ARF makes use of the BeanBuilder property sheet generation class which build a list of GUI property editors from the properties output by the JavaBeans introspection engine. This could have been re-written, and for a future upgrade a better property sheet could be provided. However, this already exists and only needing modifying slightly to provide the functionality required.

For each JavaBean property class type, a `PropertyEditor` subclass is needed so that the property can be edited by the user. If an editor is not found, ARF provides a mechanism for selecting another ARF component of the same type as a parameter for this property. This is where ARF's guided construction will help as it will show show the user which components they already have of correct type, and which components could be instantiated of correct type. How this is done is described in more detail in section 5.1.5. If a `PropertyEditor` does not exist, the programmer can provide one with their JavaBean and register it with Sun's JavaBean subsystem. This `PropertyEditor` can then be used by any other component's property sheet.

A way of easily importing user's own JavaBeans into ARF's database will be provided. The database is known as the `JarRepository`. It is called this because JavaBeans are generally wrapped up in Java Archive Files (JAR). ARF inspects the JAR files to see if there are any JavaBeans, if there are, the beans will be added to the JAR Repository database. Figure 4.6 shows how the different parts of ARF's architecture interact, i.e. the processes a JavaBean will go through from being imported into ARF, to being used in a project and then saved to XML storage.

4.3.1.1 ARF 3D Components

ARF will provide a virtual world which to all intents and purposes is the virtual version of the real environment, from which objects and sensors can be added. ARF will use Java3D to provide the virtual environment back-end. Java3D is chosen because it is



This diagram represents the information flow through ARF. This highlights the basic structure of the architecture. The user interface is able to instantiate beans from the JarRepository, then configure and connect them. The new beans are stored on the BeanBoard and/or added to the SceneGraph if they are Java3D JavaBeans. The Entire project can be saved to XML file. Individual components can be selected which make up a functional set and be exported as a SuperComponent. Projects can also be loaded from XML files and the JavaBeans and Java3DBeans loaded into the BeanBoard and SceneGraph respectively. SuperComponents can also be imported from XML and added to the BeanBoard and/or SceneGraph. A super component usually represents a collection of components which represent some useful configuration which will be needed multiple times in different projects. The JarRepository holds all JavaBeans available to use by ARF. JavaBeans are contained in JAR files which the repository indexes.

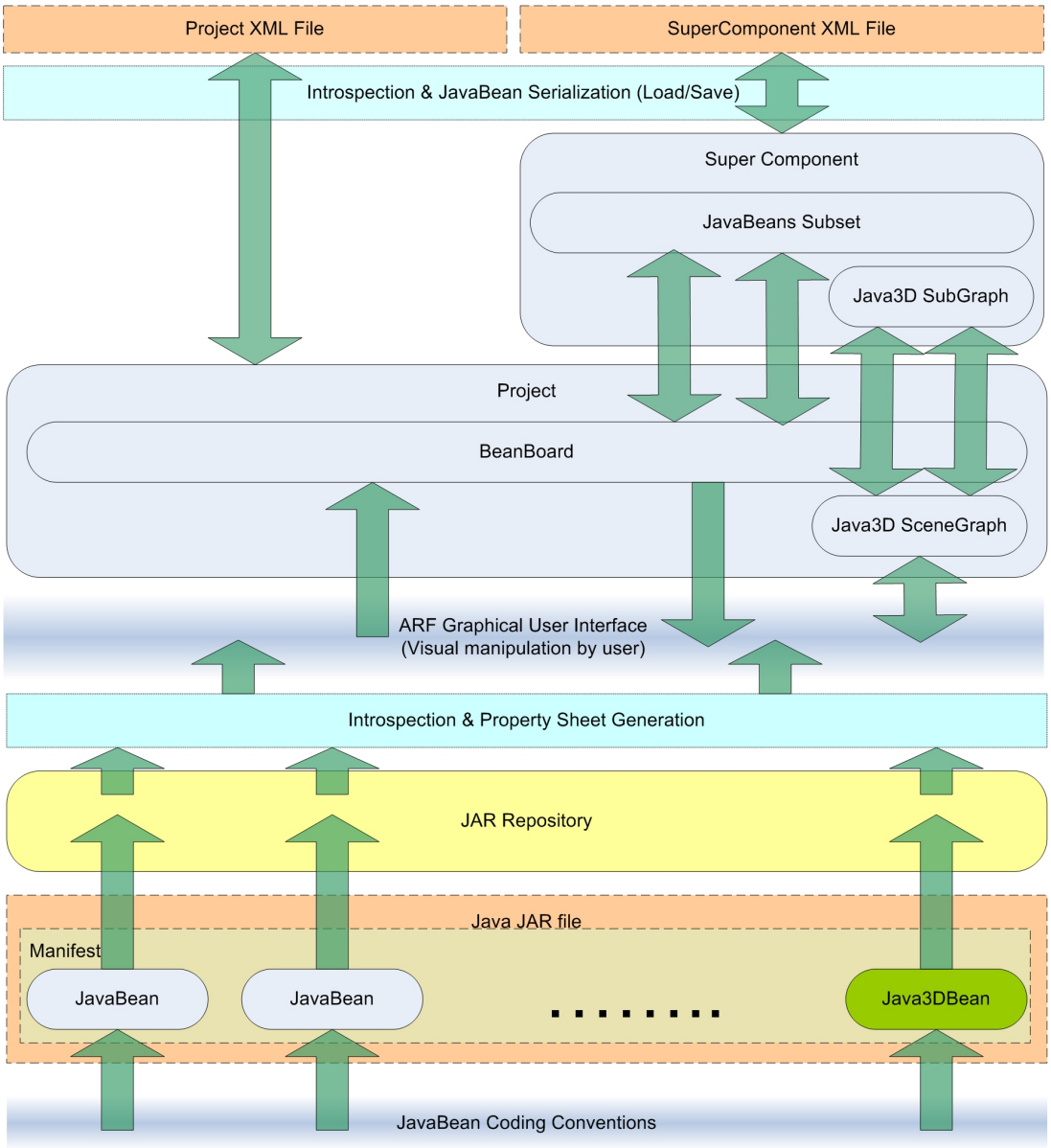


Figure 4.6: ARF Architecture Layers

object-oriented and easy to use and program. Java3D represents the virtual world as a SceneGraph, which is a very simple tree structure which allows nodes of the tree to represent physical objects (shapes), groups, or transformations in the 3D world. For example, a root Group node could be named a "universe", then another type of child, known as a TransformGroup node, can be added to this to represent groups which have positions and rotations relative to the parent group node. Thus a child TransformGroup could be named "Solar System". The tree could progress like this to contain other children such as "Venus", "Earth" and "Mars". On each TransformGroup node shapes (representing physical objects) can also be added, e.g. the "Solar System" node could have a shape which represents the "Sun" attached to it, and the "Earth" group node could have a shape which represents the "Earth" attached to it. Figure 4.7 represents the SceneGraph described above.

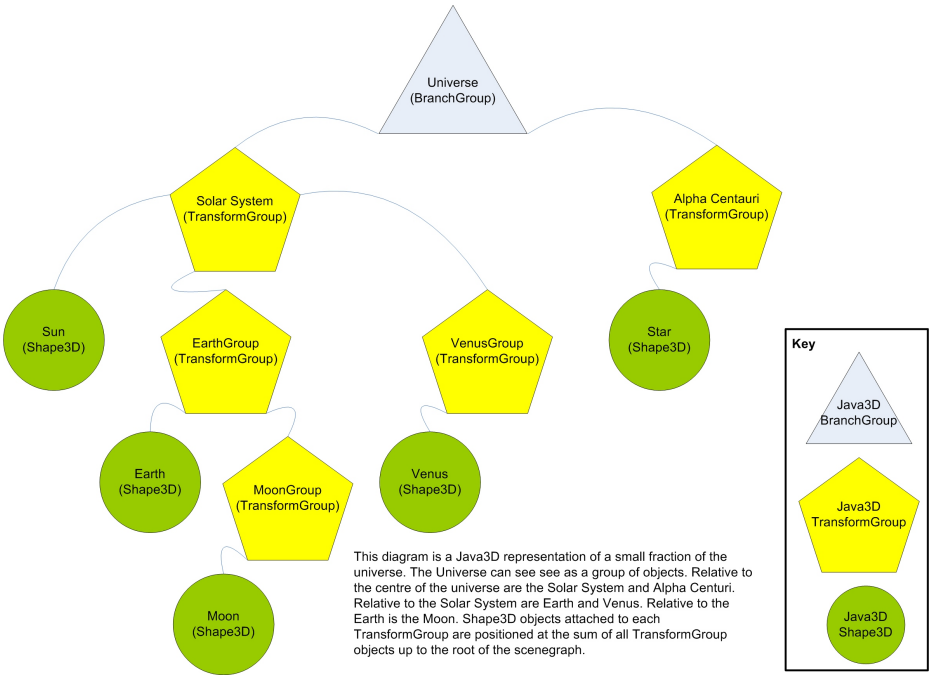


Figure 4.7: Java3D SceneGraph of the Universe

ARF provides an editable SceneGraph to which the user can add shapes, groups, transformations and behaviours to. Basically, any Java3D SceneGraph node can be added provided that there is a JavaBean in the ARF repository to represent it. Thus, ARF will provide many JavaBean versions of these Java3D SceneGraph nodes. From this point forth, these are known as "Java3DBeans" or "J3DBeans". In essence, ARF

has extended the Java3D classes to have JavaBean properties. ARF will provide mechanisms for adding and removing Java3DBeans from the SceneGraph. Therefore ARF will need to make a distinction between Java3DBeans and JavaBeans, because Java3DBeans represent 3D virtual components which are added to the SceneGraph capable of interacting with the virtual environment, and JavaBeans do anything be it 2D Graphical (Java's Swing) or merely processing/functional based with basic inputs and outputs. Obviously Java3DBeans can also be functional, but they are specific in that they don't actually do anything unless they attached to a SceneGraph which is currently being rendered in ARF. Therefore, JavaBeans and Java3DBeans will be treated differently in ARF. More on Java3DBeans is discussed in section 4.4.

4.3.1.2 Component Management

As mentioned in the previous section, Java3DBeans and JavaBeans have a fundamental difference. Java3DBeans have to be connected to the Java3D SceneGraph in order to actually function, yet JavaBeans do not. JavaBeans can be standalone non-graphical modules with basic inputs and outputs. Java3DBeans are in some senses a restriction to standard JavaBeans since Java3DBeans have to inherit from one of Java3D's SceneGraph node classes. However, being a Java3D SceneGraph node they are capable of far more than standard JavaBeans. They are capable of interacting with, and doing processing on, the entire Java3D SceneGraph, i.e. the entire ARF virtual environment can be sensed and manipulated by a Java3DBean.

ARF organises the JavaBeans into 2 data structures: Java3D SceneGraph and the BeanBoard. In ARF a JavaBean can only belong on the BeanBoard. However, a Java3DBean belongs on both the SceneGraph and the Beanboard. ARF's BeanBoard therefore keeps track of all user objects, their types, and consequently is used to find connections between different JavaBeans by inspecting each JavaBean's properties and event listeners. The BeanBoard is thus used for very powerful operations in order to aid the guided construction mechanisms. The BeanBoard can be queried as to what JavaBeans of a particular type/descendant are available for use as parameters to a property of another JavaBean. This information can then be presented to the user as options. The guided construction mechanisms also search the JarRepository

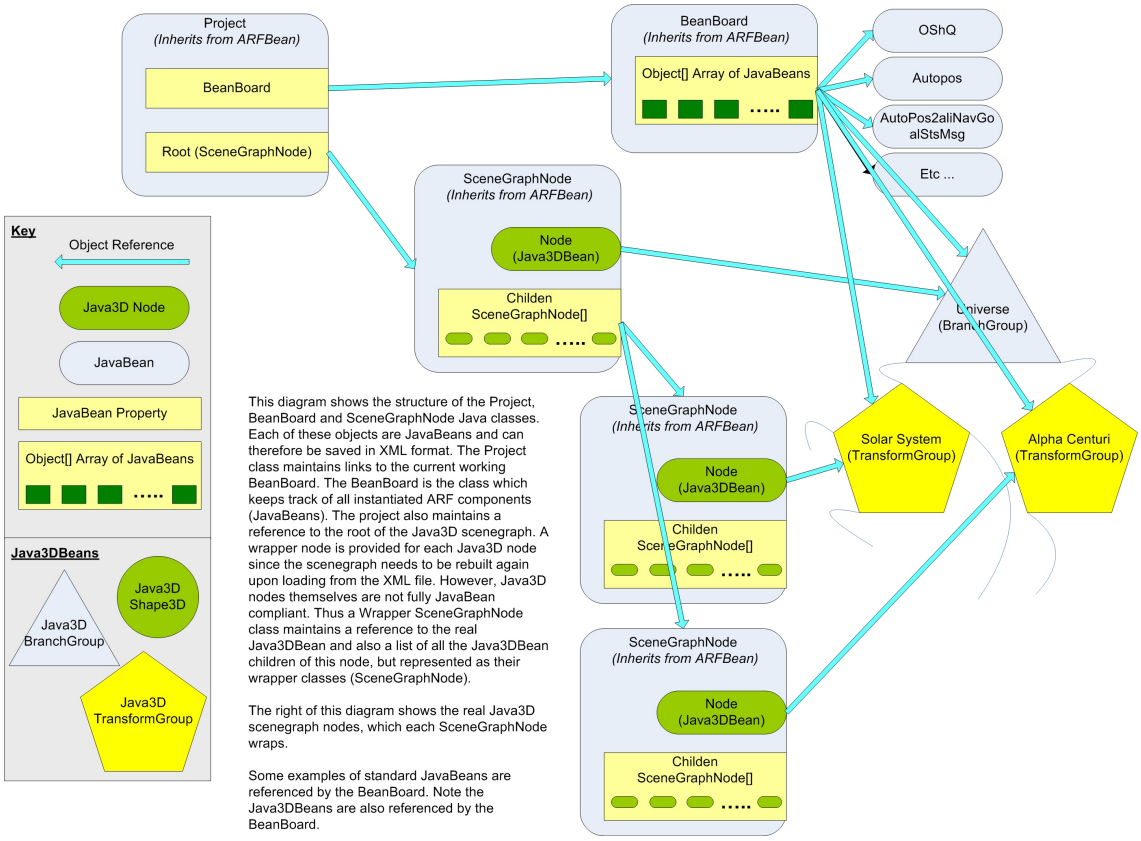


Figure 4.8: Class interaction diagram of Project, BeanBoard and SceneGraph

so see what other JavaBeans could potentially be created to be used as a parameter.

The BeanBoard and the Java3D SceneGraph are maintained by a Java class called `Project`. The `Project` instance can then be loaded and saved with all data on the SceneGraph and BeanBoards (see section 5.3.9). Figure 4.8 illustrates how the main `Project` class functions. It has two parameters: BeanBoard and the root SceneGraphNode. The SceneGraphNode is the root node which is at the top of the Java3D tree. All references to children nodes are stored inside the SceneGraphNode object. Therefore, the SceneGraph can be recursively traversed from the root node. All Java3DBeans and all JavaBeans currently in use are kept track of using a list in the BeanBoard object. The `Project`, `BeanBoard` and `SceneGraphNode` classes are all JavaBeans and can therefore be saved to disk easily merely by serialising the `Project` JavaBean. Since all the other JavaBeans are referenced either directly, or indirectly, from `Project` (more details in section 5.3.9).

4.3.1.3 Component Communications

Communication between components in ARF is via two main methods:

1. Direct method call - components have properties which require linking to other specific types of components, or interfaces. Only one object can be referenced, so this is a 1-to-1 connection.

2. Event passing - a component can implement a specific event listener interface allowing it to listen to certain event objects output by some components. A component can also maintain a list of event listeners. A component can then output data to many listeners which implement the specific event listener interface; this is a 1-to-many relationship. A component can therefore be a listener for many different types of event.

The use of event passing, and direct method call on interface types, allows many types of communication bridges to be substituted for event listeners. For example, a component outputs some event object, which is then transformed by some event listener to some other type of communication object (an OceanSHELL message). Figure 4.9 illustrates the 1-to-many relationship of event sources and listeners (from chapter

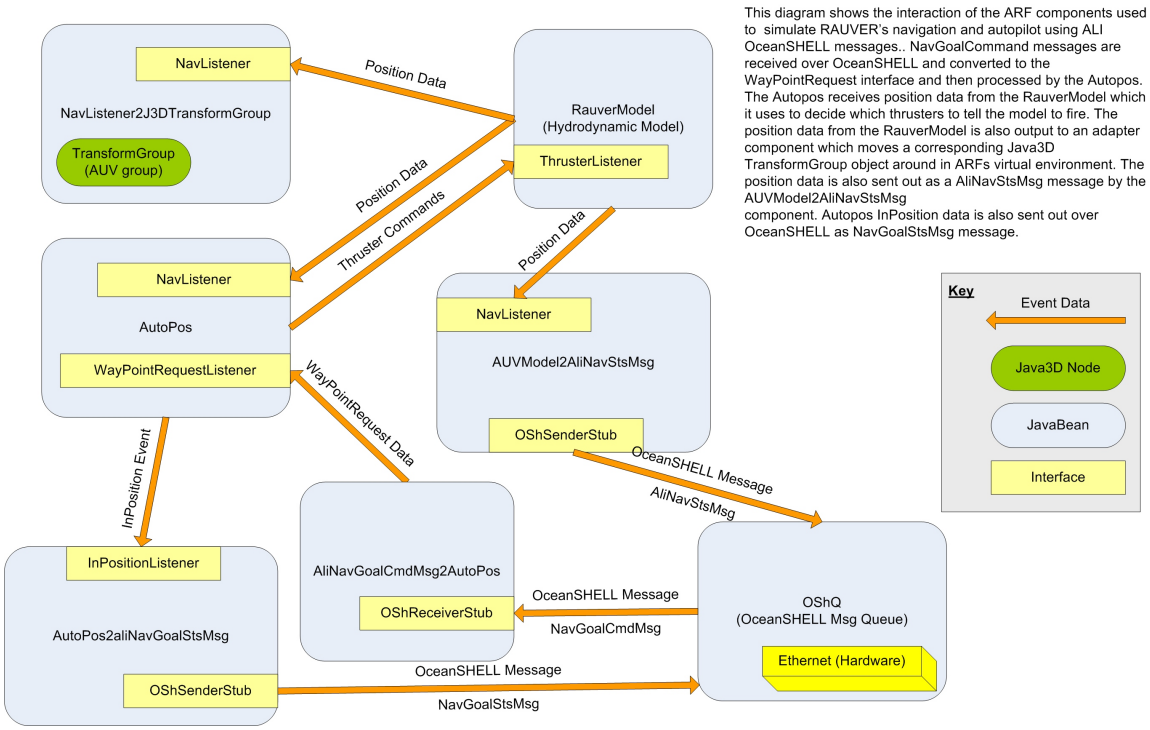


Figure 4.9: ALI Simulated Components Interconnection Diagram

7). Adding an extra layer such as OceanSHELL then enables the broadcast relationship where 1-to-many vehicle processes becomes 1-to-many intercommunicating vehicles.

Incorporating OceanSHELL into ARF is easily done by using special OceanSHELL message listener and transmission JavaBeans. The transmit JavaBeans will implement an event listener interface to convert from event objects to OceanSHELL messages. Another JavaBean is then used to convert from OceanSHELL messages back to the Java Events and inform all registered event listeners. Examples of this are given in the Use Cases in Chapter 7. The exact OceanSHELL components provided are discussed in section 5.3.7.

4.3.2 Architecture Structure

This section presents a series of diagrams to visualise the processes that provide ARF's capabilities. Figure 4.10 shows how all the main components discussed earlier will be represented to the user. These are the front end components, the Graphical User Interface (GUI) and are in effect the same components discussed in figure 3.2

in the previous chapter. The processes of the underlying parts of the architecture are demonstrated using examples, such as: JavaBean selection by the user; Property Sheet Generation for that JavaBean; and how the system provides help to the user with *Guided Construction*.

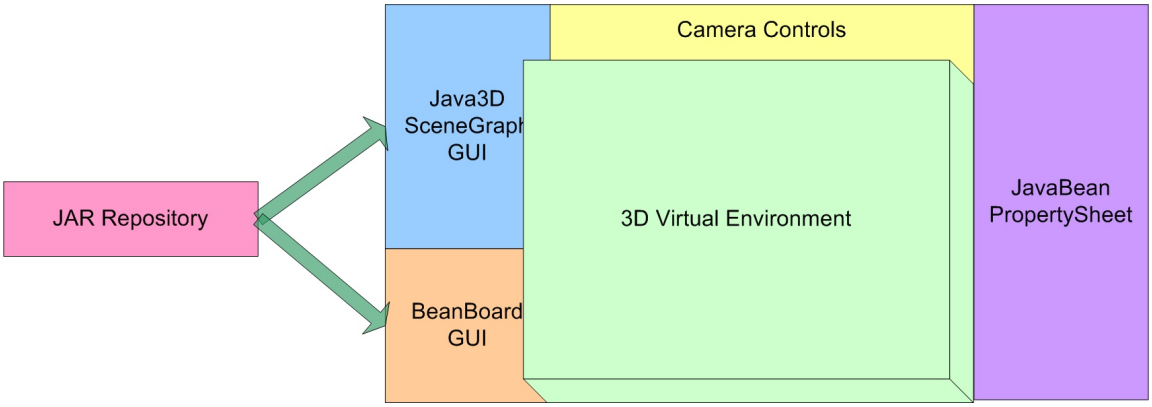


Figure 4.10: The User Interface Layout using the Architecture Parts

The main elements of the architecture are the BeanBoard, SceneGraph, PropertySheet and 3D virtual environment. These elements provide the ability to create the "Component" based scenarios needed by all the testing and visualisation techniques which inspired this project. These elements represent the low level core of ARF as they provide the functionality that allows the user to build the scenarios from the components available. These core elements allow the user to import their own components and provide the mechanisms for anyone to be able to use them in other scenarios.

4.3.2.1 Selecting a JavaBean and Creating the Property Sheet

The user can select JavaBeans, and hence view their properties, in a variety of different ways. Since all beans are added to the BeanBoard, the user can simply select a JavaBean from the list by clicking it with the mouse. The user is also able to select Java3DBeans on the BeanBoard. However, since Java3DBeans are structured the user will be able to select the Java3DBeans by clicking them on the SceneGraph view in the GUI as well. The user will be able to add new JavaBeans to the BeanBoard and SceneGraph by clicking the *right-mouse-button*. Clicking the *right-mouse-button* will show a list of all available beans in the JarRepository which can be added to the

BeanBoard or SceneGraph depending on where the user is clicking. When a bean is selected by the user a sheet of property editors is displayed on the property sheet GUI (right hand pane in figure 4.10). Figure 4.11 shows the high level process executed by ARF when a bean is selected, it shows the steps involved in displaying the bean's properties to the user.

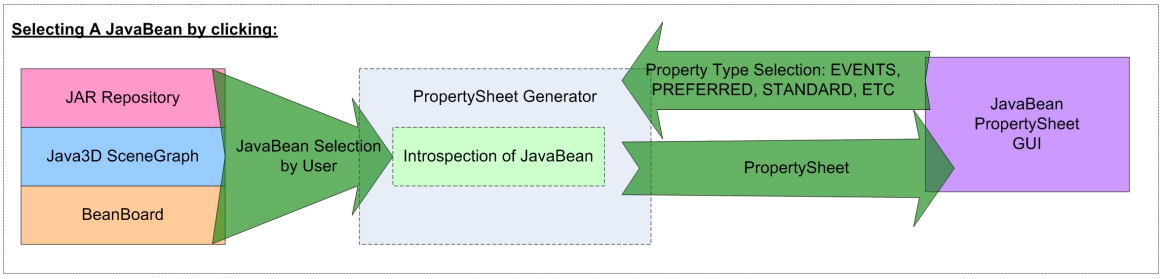


Figure 4.11: JavaBean Selection and PropertySheet Generation

A more detailed view of the property sheet generation is shown in Figure 4.12 which corresponds to the following method:

1. Select JavaBean

2. Perform introspection on attributes and methods of JavaBean class file.

3. Produce list of PropertyDescriptor classes which adhere to JavaBean coding conventions.

4. Filter out PropertyDescriptor classes which are not of visibility specified by the user in the PropertySheet GUI (*visibility* types include PREFERRED, STANDARD, EXPERT, EVENTS). If a JavaBean BeanInfo file is located with the JavaBean class then the *visibility* information is contained in that file.

5. Match Java types of the remaining PropertyDescriptors (String, Integer, Vector3d, Color etc) to registered types of a `PropertyEditor`. This is done by Java's JavaBean core classes. All registered `PropertyEditor` classes must register the Java class type which they provide the editor for. A lookup is required for each PropertyDescriptor of the JavaBean.

6. Add an instance of the `PropertyEditor` to a Graphical table, as a `PropertyEditor` is a mini GUI. Make sure the `PropertyEditor` is linked to the property descriptor and the JavaBean which it is to adjust. Add the name of the PropertyDescriptor to the Table as well so the user knows the property name.

7. Display the Generated `PropertyEditor` table (the PropertySheet) to the user.

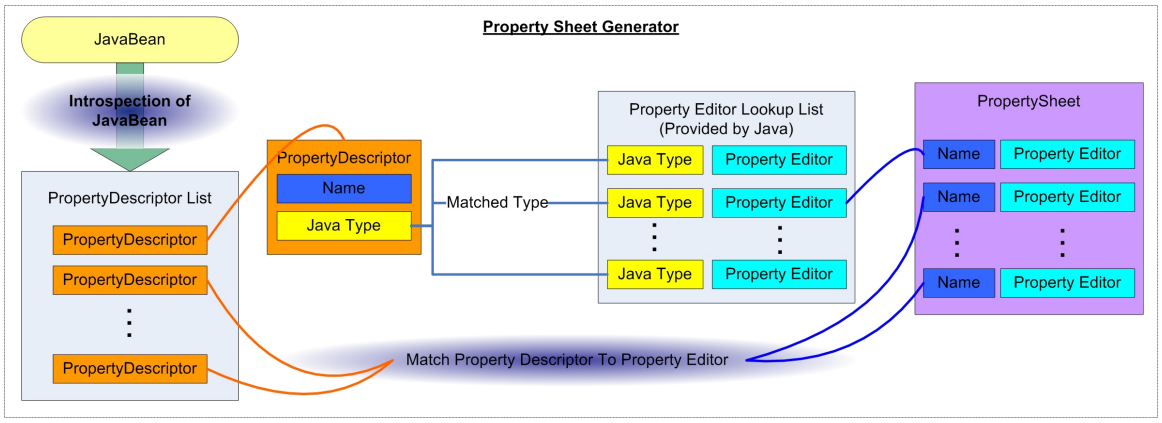


Figure 4.12: PropertySheet Generation using Introspection of a JavaBean

4.3.2.2 Guided Construction for Connecting and Instantiating JavaBeans

One of the main advantages of ARF is the ability to use introspection and gather information about the JavaBeans, such as their properties, and help the user learn quickly how the components connect together and as a result create scenarios quickly and efficiently. Figure 4.13 shows how the different parts of the architecture are used to provide options to the user when connecting and configuring component properties.

The method demonstrated by figure 4.13 is as follows:

1. User selects bean (see section 4.3.2.1)

2. The user then clicks a button on the `PropertyEditor` located on the PropertySheet to invoke the guided construction GUI.

3. The GUI will display two boxes, one for JavaBeans and one for potential JavaBeans not created yet. The JavaBeans and Potential JavaBeans have had

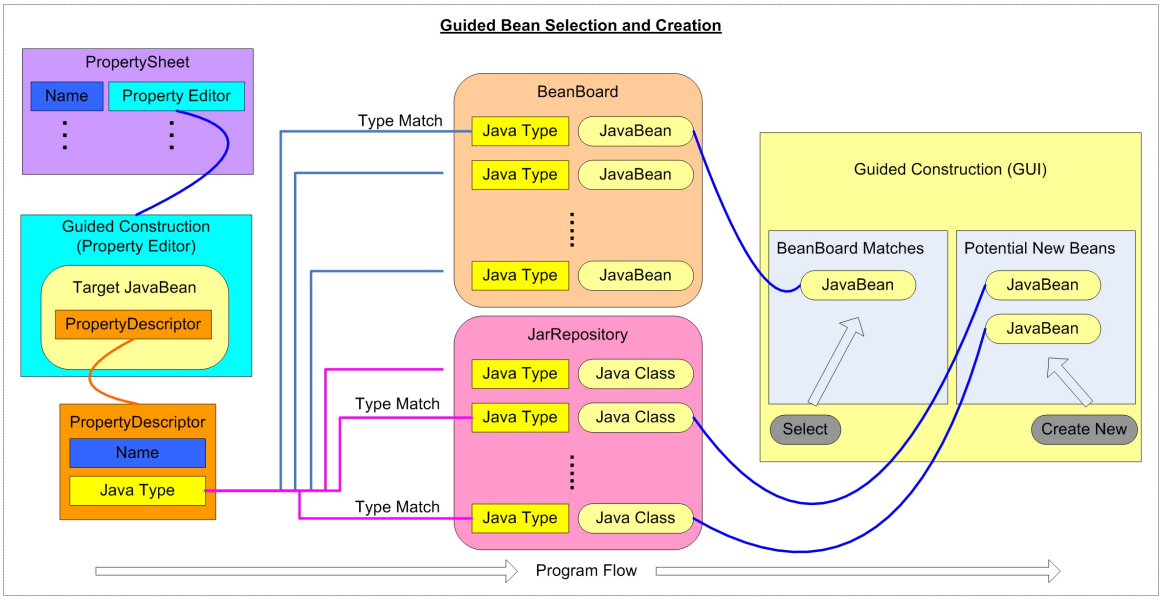


Figure 4.13: Guided construction using the BeanBoard, PropertyDescriptor and JarRepository

their types checked to make sure they are suitable as a parameter for this property. Guided construction is used when there is no specific property editor for a property's parameter type.

4. The `PropertyEditor` holds a reference to the selected JavaBean and the PropertyDescriptor which represents the property being edited.

5. Extract Java Type from PropertyDescriptor

6. Check for JavaBeans on the BeanBoard which match the type of the PropertyDescriptor.

7. Check for potential JavaBeans in the JarRepository which match the type of the PropertyDescriptor.

8. Fill the lists of the guided construction GUI with the JavaBeans which match the PropertyDescriptor type.

9. The user then selects either an existing JavaBean from the BeanBoard list as a parameter, or the user selects a JavaBean from the JarRepository list and

creates a new instance of the selected JavaBean. This new JavaBean can then be used as a parameter.

It is important that the BeanBoard and the JarRepository provide mechanisms for retrieving all JavaBeans of a certain type. The mechanisms can then be used for various different uses in the ARF GUI (See Chapter 5).

4.4 Virtual World Design

The virtual world interface for ARF is one of the main factors governing its usability. The virtual world has to be integrated into ARF in such a way that everything revolves around it, i.e. the main point of ARF is for mixed reality testing and observation. Thus it is important that the virtual world be easily navigable, easy to build and simple to interact with internal non-3d JavaBeans or external systems. Therefore ARF's virtual world requires standard navigation interfaces and many useful components for building the virtual world itself. The ARF architecture provides communication to other JavaBean objects via direct reference or Java events. Java3DBeans which make up the virtual world can thus be configured, connected and created in just the same way as any other JavaBean, except Java3DBeans have the necessity that they are connected to the SceneGraph. In essence the only fundamental difference between a JavaBean and a Java3DBean is that a Java3DBean has to be a subclass of a Java3D SceneGraph `Node` class and a JavaBean can be a subclass of anything. Therefore ARF will instinctively know by type checking, which components are Java3DBeans and guide the user accordingly as to how to add them to the SceneGraph. Section 5.2 discusses the user interfaces and the implementation of the Java3D virtual world in ARF.

Some simple components are required for the 3D virtual world to provide the functionality described above:

- MouseNavigation - Standard user interfaces in 3D virtual environments allow the user to pan the camera using a mouse. They can also zoom, rotate and click to view a point of interest.

- Camera - a virtual camera is required to be placed somewhere in the Java3D virtual environment so that the images can be rendered to the screen. However, the number of cameras is unlimited. The cameras should be easily controllable by the user, or be able to be dynamically attached to other moving entities (TransformGroups on the SceneGraph) in ARF's virtual environment.

- KeyboardControls - mouse navigation is useful, but is not very good for controlling the movement of any other objects in the environment other than the camera. The MouseNavigation can be used to move one camera around; the KeyboardControls could be used at the same time to drive some vehicle around the virtual environment. A second camera could also be placed on that vehicle to produce 2 different views of the virtual world.

These are just a few very important Java3DBeans which are required for the virtual world to be controlled and viewed in an intuitive manner. All other components are either geometry (3D objects), or sensors, listeners or behaviours more specific to the requirements of each application. For example, if haptics are required then specific nodes can be programmed by the user and added to ARF which provide the necessary inputs and outputs to/from the SceneGraph required for haptic feedback. ARF will provide standard 3D mesh loaders for loading well known 3D data formats such as X3D, VRML, DXF, 3DS, Lightwave etc, for scenery in the virtual world. Thus the construction of the graphical part of the 3D environment does not have to be carried out in ARF but instead can be created in some design tool such as 3D Studio MAX or similar and then imported into ARF via the mesh loader for Java3DBeans.

ARF will provide the ability to have more than one camera in the virtual world. This requires the user to be able to arrange the screen GUI components to accommodate multiple rendering canvases for the cameras to render their images onto. The canvas is the display panel used to display the image of a camera viewing the 3D world. The default view of ARF will be to have one camera controlled by the mouse and one large canvas that displays the camera's image. However, the canvas can then be substituted for a special panel JavaBean which will allow multiple canvas's to be added to it and sized accordingly by the user. Each camera object will therefore have

a JavaBean property which dictates which of the available canvases it renders images to.

4.5 Summary

ARF has 8 main constituent parts:

- ARF Graphical User Interface
- The Project
- The BeanBoard
- The SceneGraph (Virtual World)
- The JarRepository
- The JavaBean PropertySheet
- Guided construction
- Internal and external communication interfaces

These parts provide the base architecture of ARF. All other extensions can be built within the framework which ARF provides, i.e. all further requirements can be fulfilled by importing new ARF Components (JavaBeans) into ARF's repository. ARF therefore provides the mechanism to construct these components. The advantage of ARF compared with other JavaBean environments is that ARF provides much more help to the user. In addition ARF provides a virtual environment, which is in essence an extension to JavaBeans which allows JavaBeans to now act in a 3D space. Inherently JavaBeans are supposed to be 2D GUI components which can be connected together to create GUIs for various tasks. However, ARF provides the ability to still create 2D GUIs with standard GUI JavaBeans, but also provides the ability to not see a JavaBean at all, i.e. the JavaBean is simply a processing component with no graphical appearance. Java3DBeans remain the single most important advancement as they allow the creation of virtual entities and allow them to be represented in a

virtual world. This makes constructing a virtual environment much more straightforward since components can almost be joined together (built) in the same way as the real world, but by using a virtual interface. The virtual interface tries to capture the information which helps the user connect the components, e.g. in the way LEGO® provides an intuitive way of seeing which building blocks fit together. One further extension ARF provides is the ability to use "friendly" names for components. Users can name their components so that they are easily identified. In addition ARF components can have a description, which is available for the user to describe the purpose of the JavaBean's current configuration (section 5.3.6 gives more detail).

Chapter 5 covers in much more detail the actual implementation of ARF, using the technologies described, and provides a guide of how the different parts work.

Chapter 5

Implementation Guide

Chapters 3 and 4 concentrated on describing the constituent parts which make up the architecture of ARF. This chapter goes into much more detail on the implementation of the parts and describes how they are built, how they are linked to one another and the relationships between each part, and most importantly how user interaction is communicated through these parts to produce the outcome the user sees. Therefore, this chapter as much describes the GUI on top as the low level classes providing its functionality. This chapter starts by giving an overview of ARF's GUI and explains the different parts and how they are used. The later sections discuss in more detail how the low level modules provide ARF's capabilities. As well as the low level architecture, ARF provides many components which have been developed for use in the OSL which make up the component library. Most of the components relating to AUVs are discussed in the Use Cases chapter 7, where the projects which required the components are discussed. The later stages of this chapter discuss the various features of JavaBeans, as well as how to create JavaBeans for ARF using the many extra features and Java Interfaces which ARF provides for the programmer. The last sections discuss how the ARF Project class performs its tasks. For a full list of the ARF components and a full list of the Java classes required by ARF itself, see the Appendices C and C.1 respectively.

5.1 A Guide to the ARF User Interface

The Augmented Reality Framework brings together a number of different technologies which provide the basic entities which make up the framework. These include Java, Java3D, JavaBeans, XML and OceanSHELL. Java provides a programming platform capable of dynamic class (or module) loading and also provides portability allowing ARF to execute on any Java enabled platform. OceanSHELL is a platform independent distributed communications protocol which allows ARF components to be spatially distributed across multiple operating systems and program languages, for instance, C/C++ and Java all support OceanSHELL. Java3D provides a simple 3D API for the user. JavaBeans provide the conventions which make ARF components dynamically loadable and configurable on the fly. Finally, XML is used to save all data about all ARF components to a human readable configuration file. Thus ARF components, which are JavaBeans, can be saved in persistent storage in XML. Thus, this section shows how these different technologies have been harnessed to provide the intuitive user interface which is key to ARF's performance.

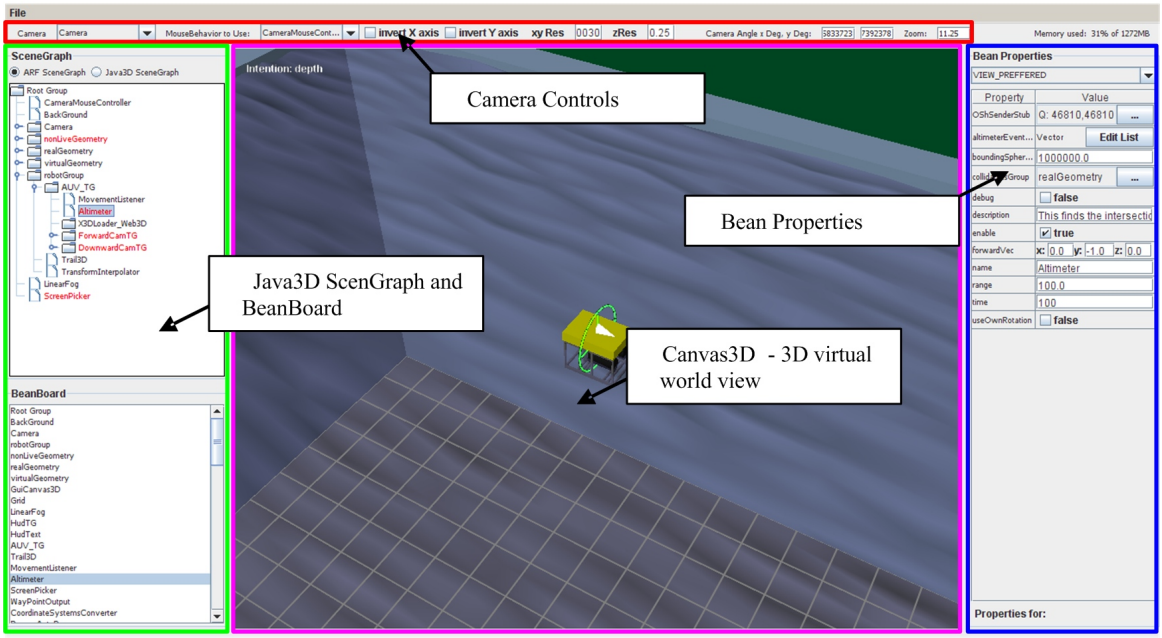


Figure 5.1: ARF GUI Overview

The main GUI of ARF displays 4 major parts of ARF's underlying architecture to

the user on the screen, as shown in Figure 5.1. The BeanBoard, SceneGraph, Virtual Environment and PropertySheet are all required by the user to construct a scenario. These are merely the visual front end to some of the main parts which make up ARF's architecture. Therefore, not all parts have a graphical front end as the user is not supposed to interact with them directly, such as the JarRepository.

5.1.1 The BeanBoard

As discussed in the previous chapter, the BeanBoard is for keeping track of JavaBeans which are currently in use; it keeps a list of all beans including Java3DBeans. The BeanBoard is represented in the ARF GUI as a list (see figure 5.2). This list displays the current state of the actual BeanBoard Java class. Both the BeanBoard and SceneGraph classes are members of a global manager class called `Project` (see section 5.3.9). The BeanBoard GUI provides a number of features which can be accessed by clicking the right mouse button:

- Add New Bean
- Delete - deletes the selected bean from the BeanBoard. Java3DBeans can only be deleted via the SceneGraph view.
- Set Main Component - Allows the user to change the 3D Virtual World view to a different sort of graphical component, such as a `ConfigurablePanel` component which can have more than one canvas added to it. The `GuiCanvas3D` is the component which acts as the window into the 3D universe of Java3D. Any JavaBeans that subclass the Java AWT or Swing classes are graphical in nature and thus can be set as the "main component", meaning that they replace the GuiCanvas3D as the main view.

The user can add JavaBean objects to the BeanBoard by clicking in the BeanBoard pane with the right mouse button and selecting "*Add New Bean*" - a list of all available beans in the Repository (excluding Java3DBeans) is shown (see figure 5.3).

Once a JavaBean is selected on the BeanBoard its ProperySheet is displayed on the right hand side of the screen. Sometimes editing the properties of the JavaBean

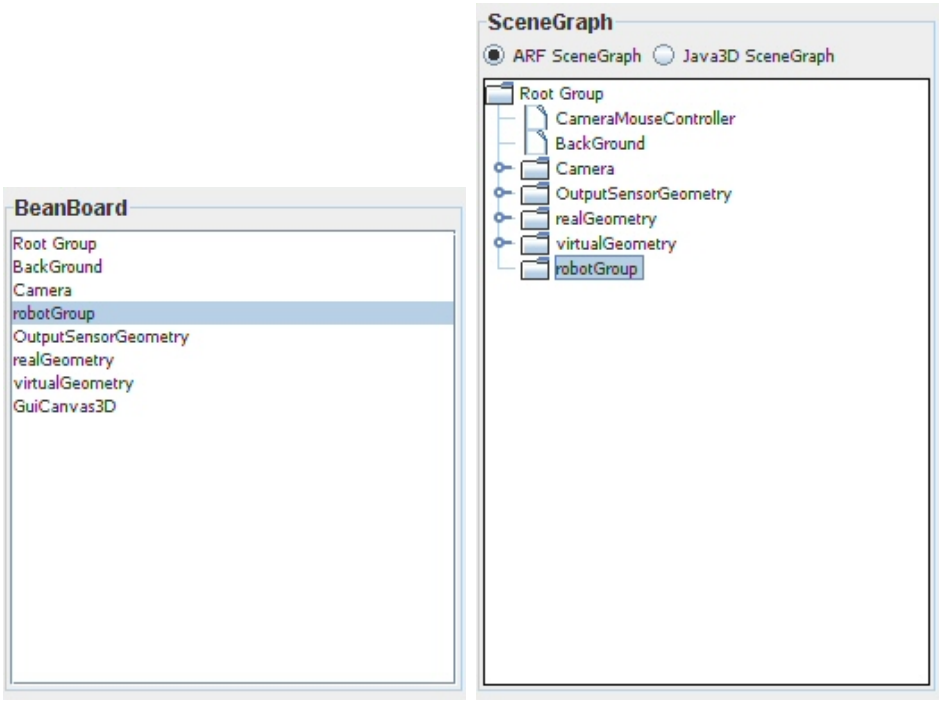


Figure 5.2: BeanBoard represented as a list, SceneGraph as tree

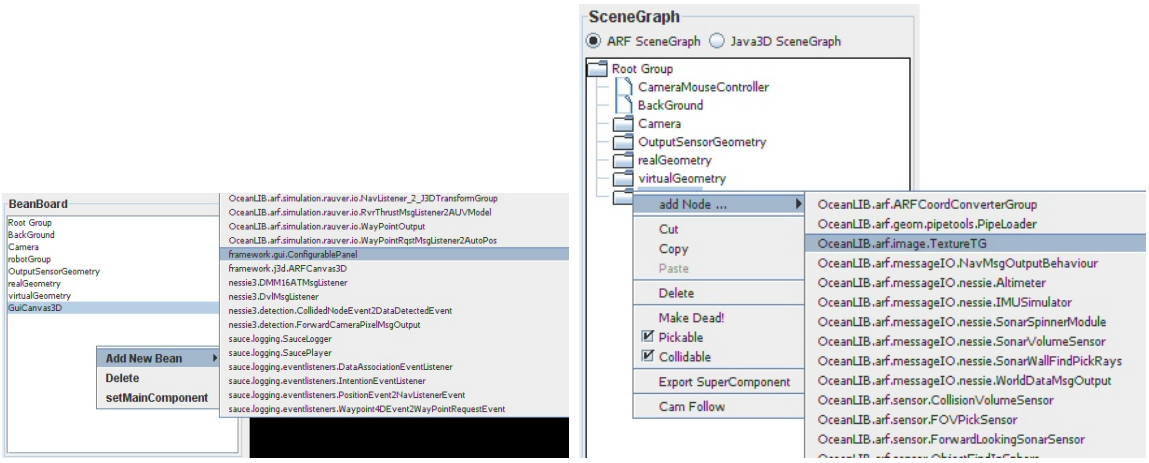


Figure 5.3: Adding to the BeanBoard or SceneGraph

via the PropertySheet is not what the creator of the JavaBean wants the user to do, because some tasks can not be easily changed into JavaBean properties. Therefore ARF provides a Java interface called `Menuable` which, if implemented by the JavaBean class, allows the user to access menus and dialogues created by the programmer for controlling the JavaBean (see figure 5.4) (section 5.3.6 discusses more Java interfaces provided by ARF). If a JavaBean has a menu it will appear in the list of options when the user *right-mouse-button* clicks the selected bean. Often this menu will provide extra modes and features or an options screen to control the JavaBean. This applies to both the BeanBoard and SceneGraph views.

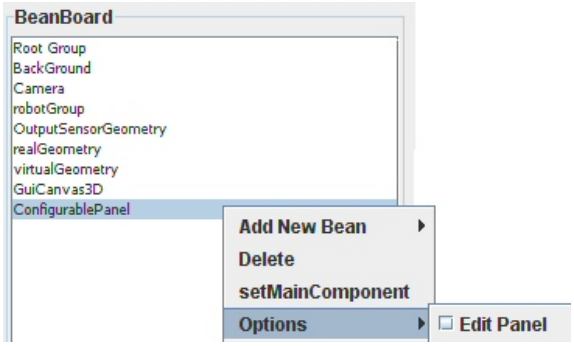


Figure 5.4: JavaBean which has its own options menu using the Menuable interface

5.1.2 The Java3D SceneGraph

ARF's SceneGraph editor (see figure 5.2) connects straight to the SceneGraph wrapper class, `SceneGraphNode` (see section 5.2.1), held by the `Project` class. The editor provides features which allow the user to directly manipulate the Java3D SceneGraph in an intuitive manner by interacting visually with the tree display. The SceneGraph works in much the same way as the BeanBoard in that the user uses the mouse's right button to bring up menus which allow them to manipulate the SceneGraph.

Java3DBeans have more options than normal JavaBeans because as well as providing some functional process, they also have to inhabit a position in 3D space and can be physically attached to other 3D objects as well. Therefore Java3DBeans have multiple properties which refer to how they interact in the 3D virtual environment such as:

- *add node ...* - add Java3D node to the selected node.
- *Cut, Paste* - allows user to move nodes/sub-trees around the tree onto different group nodes.
- *Copy* - allows entire sub-graphs to be cloned; this is especially useful for doing multiple vehicle testing.
- *Delete* - removes the node or sub-tree completely from the current project.
- *Make Dead/Make Live* - sometimes the user doesn't want to delete a node but simply wants to remove it from the virtual environment temporarily.
- *Pickable/Collidable* - similar to Make Live/Make Dead the user can select whether a particular sub-tree is pickable and collidable in the virtual environment. This is useful if the user wants an object to be visible but other objects unable to detect it. Pick testing and collision testing are similar things but Java3D treats them differently: collisions are automatically reported and therefore event based, where as picking is done when the programmer wishes it to be executed. Making an object not pickable or collidable means it cannot be detected by anything in the virtual environment.
- *Export SuperComponent* - the user can export sub-trees and other related JavaBeans from the BeanBoard as a SuperComponent. Generally this is a useful set of component configurations which the user will want to use again and again in different projects (see SuperComponents in section 5.1.7).
- *Cam Follow* - make the camera follow the currently selected SceneGraph node wherever it may be in the virtual universe.

The SceneGraph options available for Java3DBeans are shown in figure 5.3. The options menu is invoked by pressing the *right-mouse-button* on a node on the SceneGraph. In order to add a Bean to the J3DBeanTree the user must first select the node that they wish to add a child to, and then do the same as on the BeanBoard (see figure 5.3).

5.1.2.1 SceneGraph View Modes

The SceneGraph can display in 2 different modes; figure 5.5 illustrates the difference between these modes. The primary mode is to view ARF's SceneGraph which is a wrapper tree encompassing the Java3D SceneGraph. The wrapper SceneGraph makes constructing the Java3D SceneGraph much easier since it provides some convenience methods that Java3D doesn't normally provide (for more detail about the do's and don'ts that Java3D enforces see section 5.2.1). The other view is for the user to inspect the actual Java3D scenegraph, purely for observation purposes. In order to switch between the two modes a toggle switch is located at the top of the SceneGraph with two options "*ARF SceneGraph*" and "*Java3D SceneGraph*". Java3DBeans can still be selected from the "*Java3D SceneGraph*" view but none of the right mouse click options are available under this mode. The "*Java3D SceneGraph*" view shows hidden nodes which are are sometimes children of Java3DBeans, but which the user is not supposed to see or alter. Java3DBeans sometimes want to control the SceneGraph which lies below them, therefore anything below the Java3DBean in the SceneGraph-which is not added by the user-is unknown by the user since they did not add it from the JarRepository. ARF can simply ignore sub-graphs which are created and managed by Java3DBeans since this information is restored by the Java3DBean itself upon load. Therefore ARF doesn't have to specifically save/load the hidden sub-graphs when the ARF `Project` is loaded and saved.

5.1.2.2 Live and Dead SceneGraphs

Sometimes the user doesn't want to delete a node but simply needs to temporarily remove it from the virtual environment. Since Java3DBeans have to belong on the ARF SceneGraph in order to exist, a Java3DBean has two visibility modes: (Figure 5.6 shows how to make a SceneGraph *live* or *dead*).

1. *Live* - A Java3DBean is itself *live* when it is first added to the SceneGraph, however it may not actually be *live* if its parent is *not live*.

2. *not Live/Dead* - When the user wishes to stop rendering an entire portion of the SceneGraph they can set the node's status to *not live*. This means that this

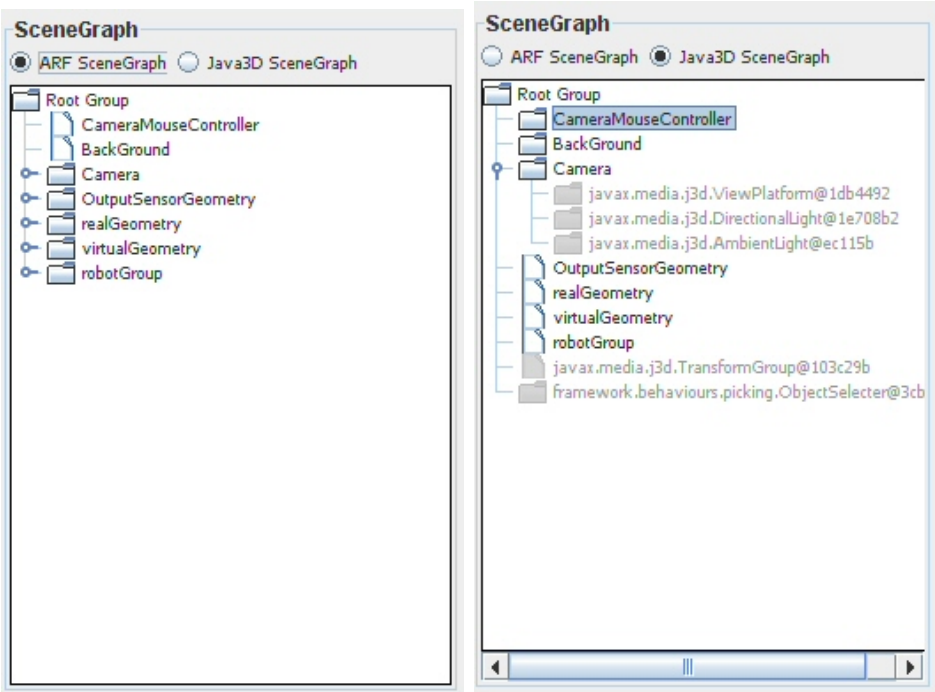


Figure 5.5: Java3D SceneGraph and ARF SceneGraph Views

node is actually disconnected from the Java3D SceneGraph but a reference is kept by the ARF SceneGraph wrapper so that it can be made *Live* again later if needs be.

In order to make the node *live* or *dead* the user can click the *right-mouse-button* on the selected node and select the *make live* or *make dead* option depending on the current state of the node. Each node which is set to *dead* is shown in red, and all child nodes which are *dead* as a consequence are shown in blue. In essence by making the parent node *dead*, the entire sub-tree is made *dead*. Thus, nodes which have their own status set to be *dead* are always red even if their parent also has its status set to *dead*.

5.1.3 The PropertySheet

When the user selects a Bean, from either the BeanBoard or the SceneGraph, the Bean's properties are displayed in the properties pane (PropertySheet) on right side of the ARF GUI. If a BeanInfo object is found with the JavaBean class in the JAR file, then only properties described in the BeanInfo object are displayed on the Prop-

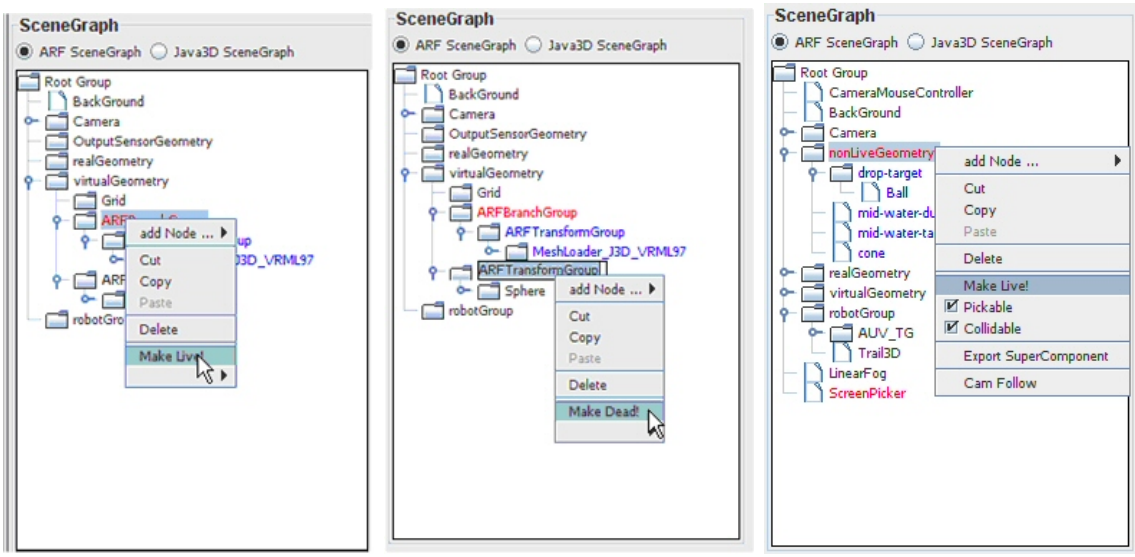


Figure 5.6: Live and Dead SceneGraphs

ertySheet. Otherwise introspection is used to detect which properties of the bean adhere to the JavaBean conventions and a property sheet is generated as previously shown in Figure 4.12, section 4.3.2.1.

The user can then edit the Bean's properties in the properties pane by using the individual property editors. Figure 5.7 shows the PropertySheet for an *ARFTransformGroup* Java3DBean with all the individual `PropertyEditors` displayed in a table. A JavaBean can have many properties of varying types, therefore in order to be able to edit a property there must be an associated `PropertyEditor` for the property's Java class type, e.g. String, Integer, Color etc. The `PropertyEditors` for each type are then placed in a table called a PropertySheet which is then displayed in the properties pane of ARF's GUI. The user can easily select which property to modify since `PropertyEditors` are GUIs; `PropertyEditors` understand the Java Class type of the property and provide an intuitive user interface to edit it.

BeanInfo objects, briefly mentioned earlier, allow the programmer to specify the level of visibility of the JavaBean properties of their class. This is very useful for the creator because some properties should maybe be hidden or only altered by an expert even though they adhere to the JavaBean coding conventions. JavaBean coding conventions are often used regardless of whether a class was meant to be a JavaBean. Therefore the creator needs the ability to hide or restrict visibility of some properties

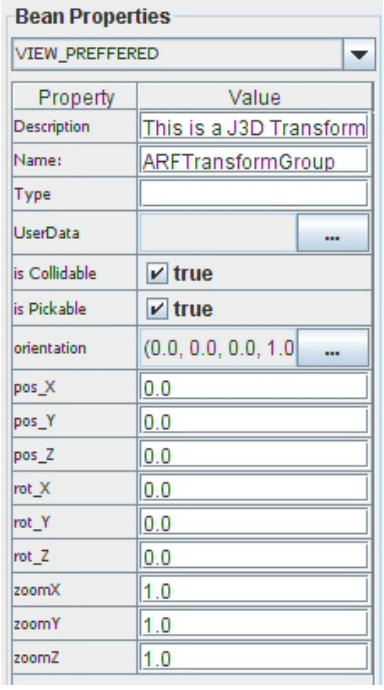


Figure 5.7: PropertySheet of an ARFTransformGroup Java3DBean

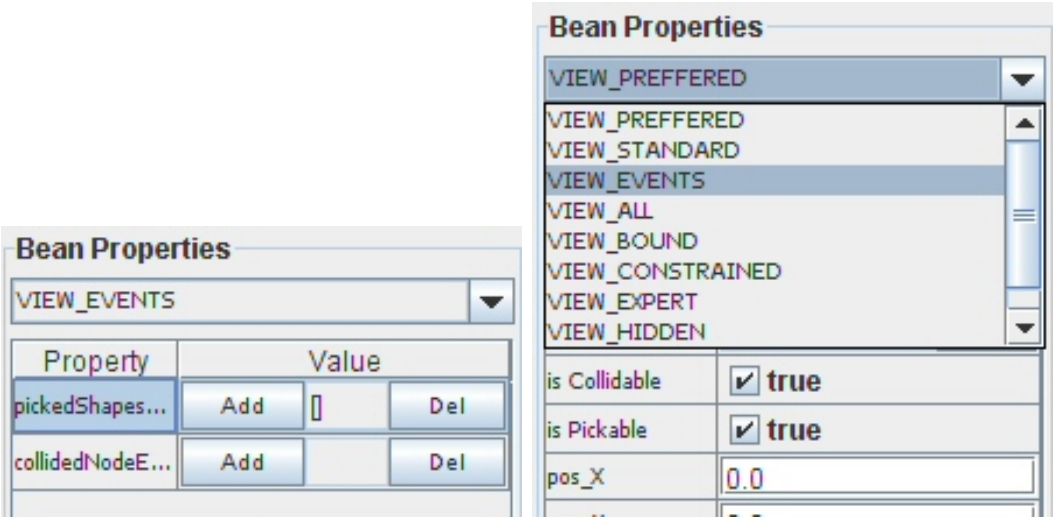


Figure 5.8: Changing PropertySheet visibility to event listeners only

to protect the user from invoking dangerous methods by editing the PropertySheet incorrectly. There are multiple types of visibility available to the JavaBean creator such as:

- *Preferred* - specifically highlighted by the programmer as properties the user can modify.
- *Standard* - this ignores BeanInfo and simply displays all properties which adhere to JavaBean conventions.
- *Expert* - properties that are potentially dangerous to adjust.
- *All* - displays all properties of any sort e.g. read only, expert etc.
- *Bound* - displays only Bound Properties described in BeanInfo object.
- *Constrained* - displays only Constrained Properties described in BeanInfo object.
- *Hidden* - shows properties specifically hidden by the programmer. Generally dangerous to try and modify.
- *Read Only* - properties which don't have a "Set" (see section 5.3.1) method and are thus only partially JavaBean properties.

In ARF the user can switch between visibility levels using the drop down box at the top of the PropertSheet (see figure 5.8). The default visibility level should either be "*Preferred*" or "*Standard*". In addition to changing visibility levels, this is also where the user can select to view the *event listener* properties of the JavaBean (see next section). Figure 5.8 displays the event listener PropertyEditors for the `FOVPickSensor` Java3DBean discussed in chapter 7. Section 5.3 discusses how JavaBeans can be created and used in ARF in much more detail.

5.1.4 Editing Event Listeners

Following on from the previous section, this section describes how ARF accommodates event listeners of JavaBeans. Figure 5.8 displays the event listener PropertyEditors

for the two types of event listeners of the `FOVPickSensor` Java3DBean discussed in chapter 7. The PropertyEditors for all event listener objects are the same.

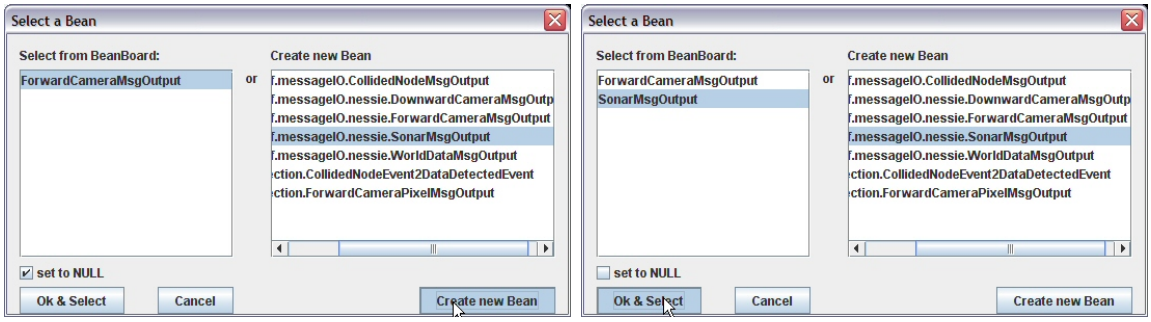


Figure 5.9: Selecting event listeners with guided construction

The PropertyEditor provides a user interface for adding/removing event listeners to/from a JavaBean. `FOVPickSensor` outputs two types of event object. If the user wants to connect an event listener or remove one they should click "*Add*" or "*Del*" respectively on the PropertyEditor. Both "*Add*" and "*Del*" cause a new menu to appear allowing the user to choose which JavaBean to connect or remove as an event listener.

When adding an event listener the *Guided Construction* dialogue appears, to help the user decide which event listener to connect or which event listener they can create that is of correct type (figure 5.9 shows the dialogue box). The user can set an event listener value to `null`; however, this is not recommended and can cause unforeseen problems as the JavaBean currently being edited may not check for `null` pointers; the best thing to do is select the "*cancel*" button if the user changes their mind about editing event listeners. The right hand pane of the dialogue shows all the potential JavaBeans which could be created which implement the event listener interface required. The user can select "*Create New Bean*" if they want to add a new JavaBean to the BeanBoard to use as an event listener. The event listener JavaBeans which already exist are displayed on the left pane of the dialogue box.

To remove an event listener simply click "*Del*" on the PropertyEditor on the PropertySheet and then select the listener to remove in the dialogue shown in figure 5.10.

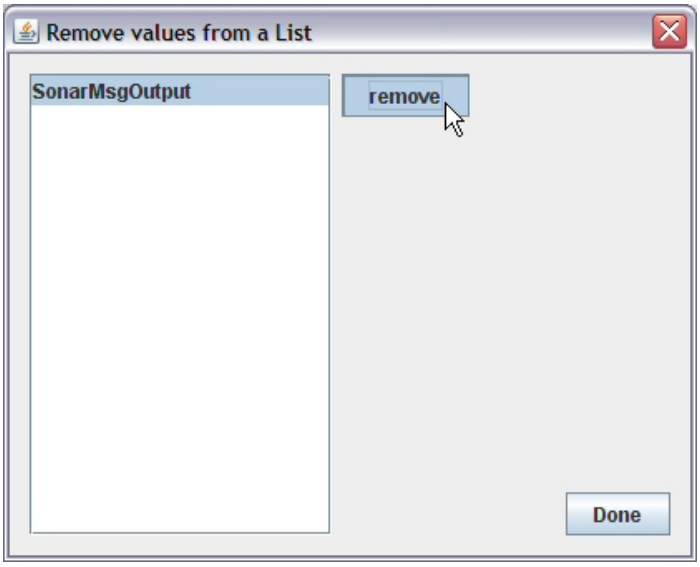


Figure 5.10: Removing an Event Listener

5.1.5 Guided Construction

The previous section describes how guided construction is used to help the user add event listeners to JavaBeans. However, this same interface can be used to connect JavaBean properties which have a type for which there is no registered PropertyEditor. Therefore for all properties which have no specific PropertyEditor the *Guided Construction* PropertyEditor is used. The *Guided Construction* PropertyEditor allows JavaBeans of the correct Java type to be used as reference properties, i.e. like pass by reference methods in C++ and Java. ARF provides standard property editors for all the Java base types: *String, Integer, Float, Double* etc, therefore the *Guided Construction* PropertyEditor is only used for reference types, i.e. pointers.

For example, the `RauverModel` JavaBean, described in section 7, requires a property connection to a `SimulationClock` type. The `SimulationClock` acts as a real-time clock for simulation components to work from. Thus, all simulation components must have a reference to a `SimulationClock` JavaBean. If all simulation JavaBeans are linked to the same `SimulationClock` instance then the time multiplier for all connected JavaBeans can be adjusted at the same time; speeding up or slowing down the rate of simulation.

Figure 5.11 shows the PropertySheet for the `RauverModel` JavaBean. The properties which have no associated PropertyEditor are highlighted in red. The Proper-

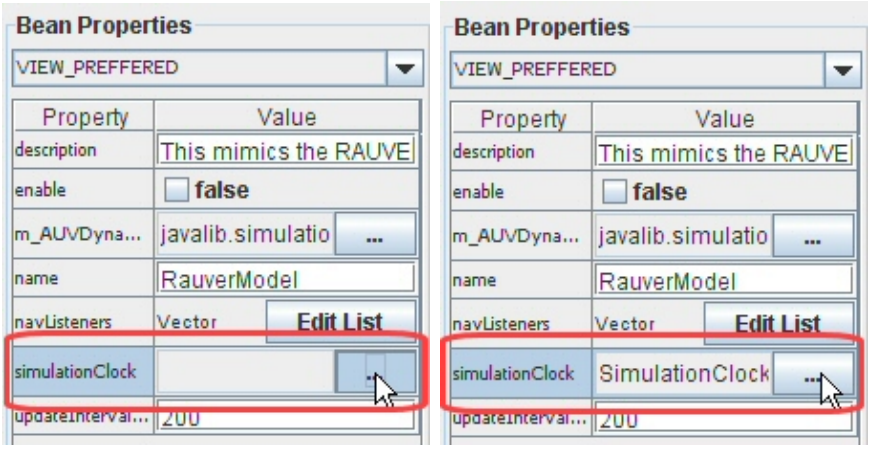


Figure 5.11: GuidedConstruction PropertyEditor before and after connection

tyEditor displayed for these properties is the GuidedConstruction PropertyEditor. If the property is currently blank then there is no JavaBean currently connected to this property. However, if there is a JavaBean connected it is shown next to the "..." command button. The user must click the "..." button in order to change the property reference to a different JavaBean.

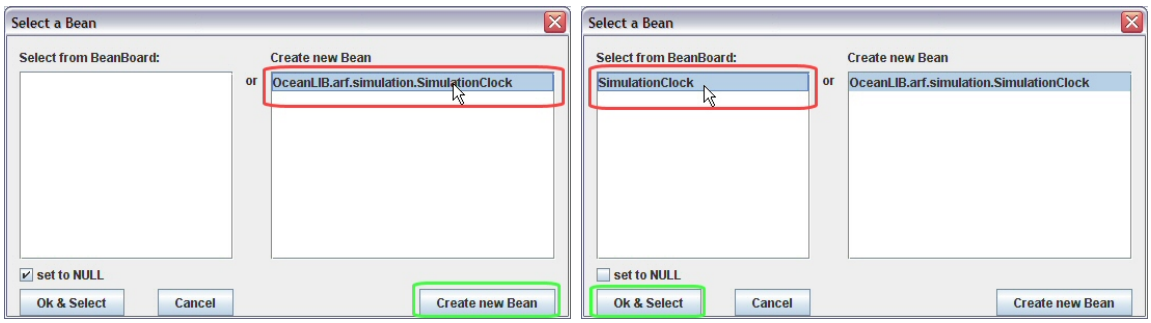


Figure 5.12: Instantiation and selection of JavaBean using Guided Construction PropertyEditor

The *Guided Construction* pane, shown in figure 5.12, allows the user to either select a JavaBean from the BeanBoard that has already been instantiated, or instantiate a new JavaBean of correct type from the JarRepository. The JarRepository and the BeanBoard both have pruning algorithms which search their database for potential candidates of the correct type for the property. The only current limitation of this *Guided Construction* pane is that it does not allow the user to create new instances of Java3DBeans since these have to be added to a specific location on the SceneGraph.

However, although the Java3DBeans are displayed as options the user is informed that they have to manually add the Java3DBean to the SceneGraph (see figure 5.13).

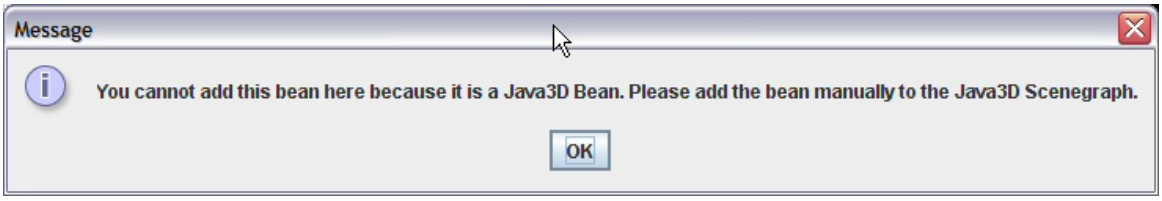


Figure 5.13: Java3DBeans can't be added to BeanBoard using guided construction

5.1.6 Loading and Saving a Project

Since ARF is based on JavaBeans, it makes serialising component's configurations and connections quite straight forward. The only thing that is really required in addition to standard JavaBean serialisation protocols is a method of saving and loading an entire project. The user interface is discussed here, however section 5.3.9 describes the `Project` class and how ARF actually loads and saves the JavaBean information and the state of the project. ARF provides functionality for loading, saving and creating new projects from the "*File*" menu shown in figure 5.14.

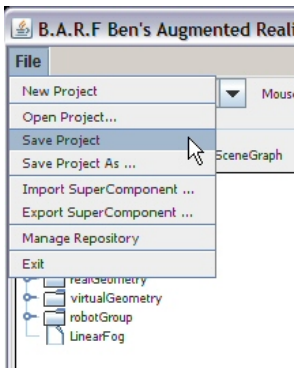


Figure 5.14: The File menu

Information about the project is saved as an XML file. Figure 5.15 shows the save dialogue which is in the same format as all file dialogues under Windows/Linux. Opening a project is done in the same way from the "*File*" menu.

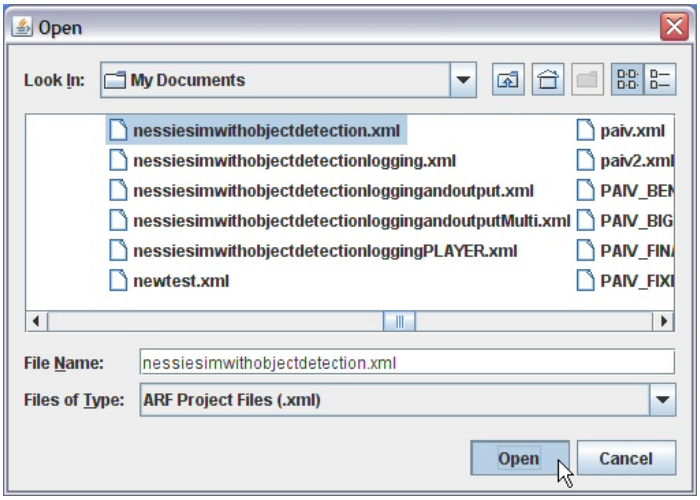


Figure 5.15: Saving a project

5.1.7 Importing and Exporting SuperComponents

ARF provides the ability for the user to create groups of components which together perform some useful task, or provide some specific functionality which the user may want to use multiple times in different projects; these are called SuperComponents. ARF provides a tool which allows the user to export certain groups of components, be them JavaBeans or Java3DBean SceneGraphs, as SuperComponents; the user can select multiple sub-graphs from the SceneGraph for export as well as any other JavaBeans from the BeanBoard which are needed for the SuperComponent's functionality. ARF provides automatic reference checking for the user so that all JavaBeans connected to a JavaBean selected for export are also added to the BeanBoard export list. Therefore the GUI which ARF provides for creating SuperComponents has two export lists, one for Java3D sub-graphs and one for JavaBeans from the BeanBoard. On import the user must decide where on their current Java3D SceneGraph the sub-graphs should be added.

In order to export a SuperComponent the user can select "*Export SuperComponent*" from the "*File*" menu (see figure 5.14). However, the user can also select to export a sub-graph of the SceneGraph by clicking the *right-mouse-button* on the current selected node on the SceneGraph pane (see Figure 5.16). This automatically adds the current Java3D subgraph (including all child nodes of the selected node) to the sub-graph export list of the export SuperComponent dialogue shown in figure 5.17.

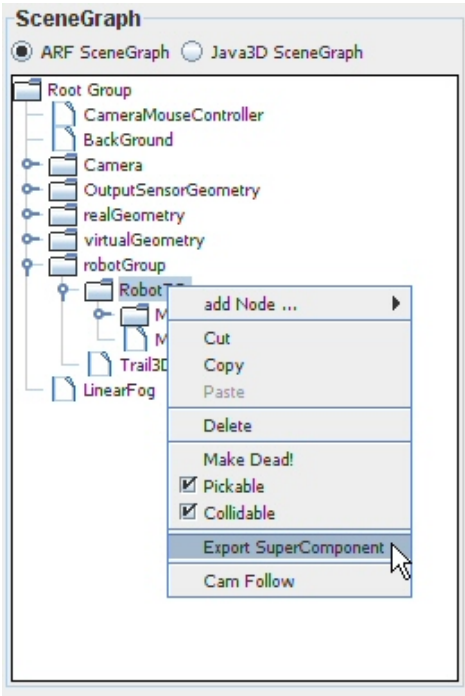


Figure 5.16: Exporting a SuperComponent from the SceneGraph view

The left side of the SuperComponent export dialogue shows the current SceneGraph and BeanBoard of the project. The user must select nodes from the SceneGraph and click the arrow button in order to add that sub-graph to the export list. Any dependencies on other JavaBeans/sub-graphs are also detected and added automatically to the export lists. JavaBeans from the BeanBoard are added to the BeanBoard export list in the same way. Once the user is satisfied that all the Java3DBeans and JavaBeans required for the SuperComponent are added to the lists then the user must click the "*Export*" button where they will enter a standard save-file dialogue.

The user can import as many SuperComponent objects into their project as they like. Importing is much the reverse of exporting. The user must first select "*Import SuperComponent*" from the "*File*" menu to access the import dialogue shown in figure 5.18. The user must then click the "*Import...*" button which brings up a file selection box allowing the user to choose which file to import components from. SuperComponent files end with "`.scp`" extension. The components which make up this SuperComponent are then displayed on the right hand side of the import dialogue. The user must individually choose where to add the Java3D sub-graphs to the existing SceneGraph. To attach one of the new Java3D sub-graphs to the SceneGraph the user

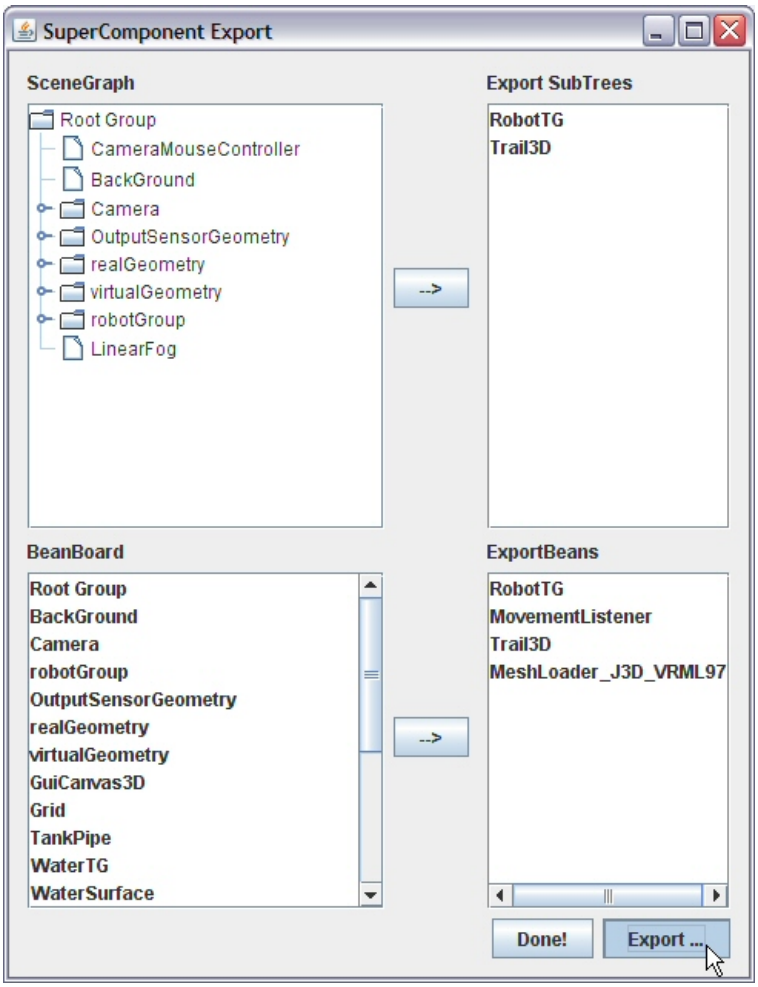


Figure 5.17: Exporting a SuperComponent

must select a node on the left SceneGraph pane, as this will be the destination parent. The user must also select on the right hand pane the sub-graph to add. To add the sub-graph click the "<-" button between the left and right panes. Standard JavaBeans are added to the BeanBoard in the same way, except the user doesn't have to choose their location. All sub-graphs must be attached to the SceneGraph before the user can exit the import dialogue.

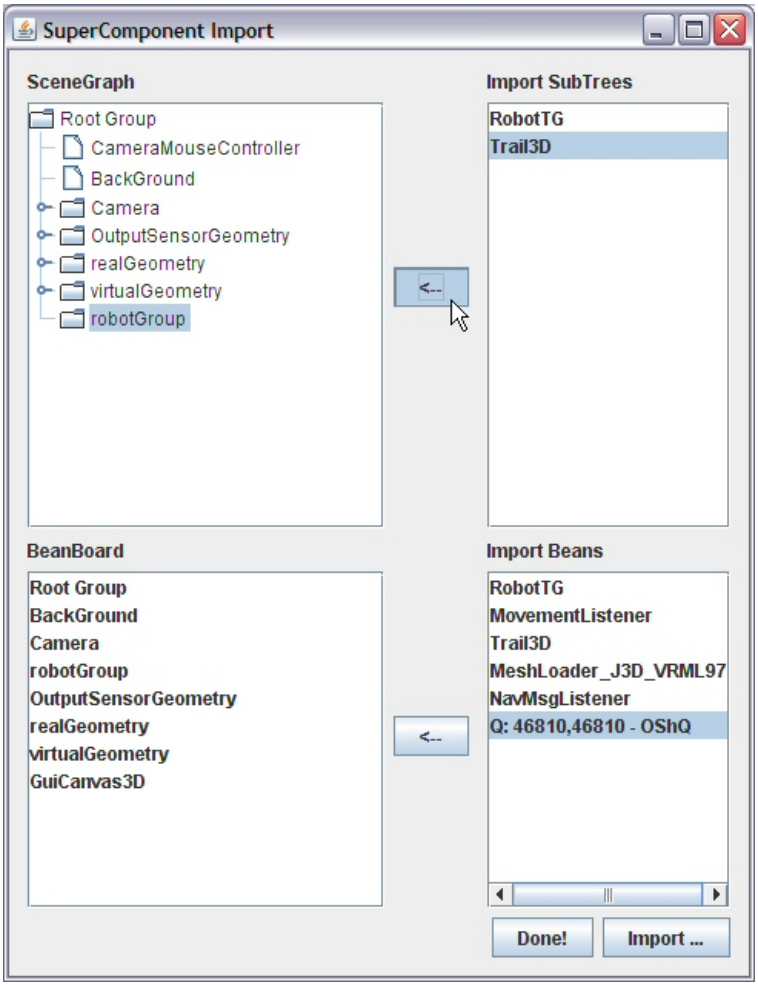


Figure 5.18: Importing SuperComponents

The addition of SuperComponents to ARF allows high end user modules to be created which perform a specific task required many times throughout different projects. The ability to do this could never be achieved using a monolithic virtual environment as it requires a very flexible and modular underlying architecture to be implemented.

5.1.8 Virtual world control and navigation

ARF provides an intuitive mouse navigation system for the virtual environment. The user is able to use the *right-mouse-button* and wheel to pan and zoom around the selected target. The left mouse button is used for pointing the camera at a specific object. The camera can also be made to follow an object from the SceneGraph by clicking the *right-mouse-button* on the node on the SceneGraph and pressing "*Cam Follow*". The camera angle and zoom can also be adjusted using the camera toolbar displayed in Figure 5.19.

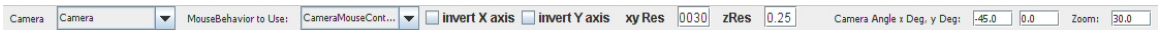


Figure 5.19: ARF virtual world navigation using camera controls

The camera toolbar enables the user to choose which camera to control as there may be multiple cameras on the SceneGraph which render to different canvases. The camera toolbar also associates a `MouseControlBehavior` with the current camera selected. This feature is so that custom camera control behaviours can be created for use with ARF and added to the SceneGraph.

5.2 Java3D Virtual World

Java3D provides the virtual environment for ARF. ARF provides an intuitive Java3D SceneGraph rendering system which makes it easy for the user to add Java3D objects to the virtual environment. The other advantage is that Java3D is Java and as a consequence can be made to be JavaBean compatible. ARF required a new type of JavaBean in order to provide Java3D functionality using JavaBeans; these are referred to in ARF as Java3DBeans/J3DBeans.

Unfortunately Java3D doesn't inherently provide JavaBean coding convention support, as many other Java packages do. Although some work has been carried out on some of the related Java3D packages, such as the `javax.vecmath` package, Java3D SceneGraph nodes still require additional methods and functionality to satisfy the configuration and persistence storage mechanisms of JavaBeans. Some of the core components of Java3D needed extending to provide the necessary functionality, i.e.

extending to adhere to JavaBean coding conventions and get round some restrictions imposed by Java3D.

To begin with this section will look at the Java3D SceneGraph structure and describe the classes required by the ARF backend to support the use of Java3D. Then the simplifications and modifications to Java3D are described in detail.

5.2.1 ARF Java3D Universe Structure

The Java3D SceneGraph is a simple tree structure. Some nodes are group nodes and some nodes can only be leaf nodes. Each group node has a list of child nodes and also a reference to its parent node. Unfortunately the accessor methods for the children list and the parent node do not conform to JavaBean coding conventions and consequently cannot be serialised to a file using Java's JavaBean introspection techniques. Rather than extending every type of Java3D SceneGraph node there is to provide JavaBean functionality, for which there are many, a simpler approach is to use a wrapper SceneGraph for the Java3D SceneGraph. The wrapper SceneGraph holds the extra information necessary to store the structure of the Java3D SceneGraph. In order to do this a wrapper SceneGraph object called `SceneGraphNode` is used. For every Java3DBean added by the user to the SceneGraph, ARF creates a wrapper node which holds a reference to the real Java3D node. As the user adds children and changes the parents of nodes, ARF captures this information and stores it in the wrapper SceneGraphNode object as well. This means that no alterations have to be applied to existing Java3D node classes in order to make them into JavaBeans since ARF stores all additional information on SceneGraph structure in wrapper node classes.

When a user adds a Java3DBean child to an existing SceneGraph node, ARF stores the parent/child relationships in the ARF wrapper SceneGraphNode class. This class is a JavaBean and can therefore be serialised to disk without any problems. On reloading of the SceneGraph from file, this class uses its saved parent/child node relationships to reconstruct the underlying Java3D SceneGraph to the state it was before it was saved. Figure 5.20 shows how the wrapper `SceneGraphNode` object interacts with the real Java3D SceneGraph made up of Java3DBeans.

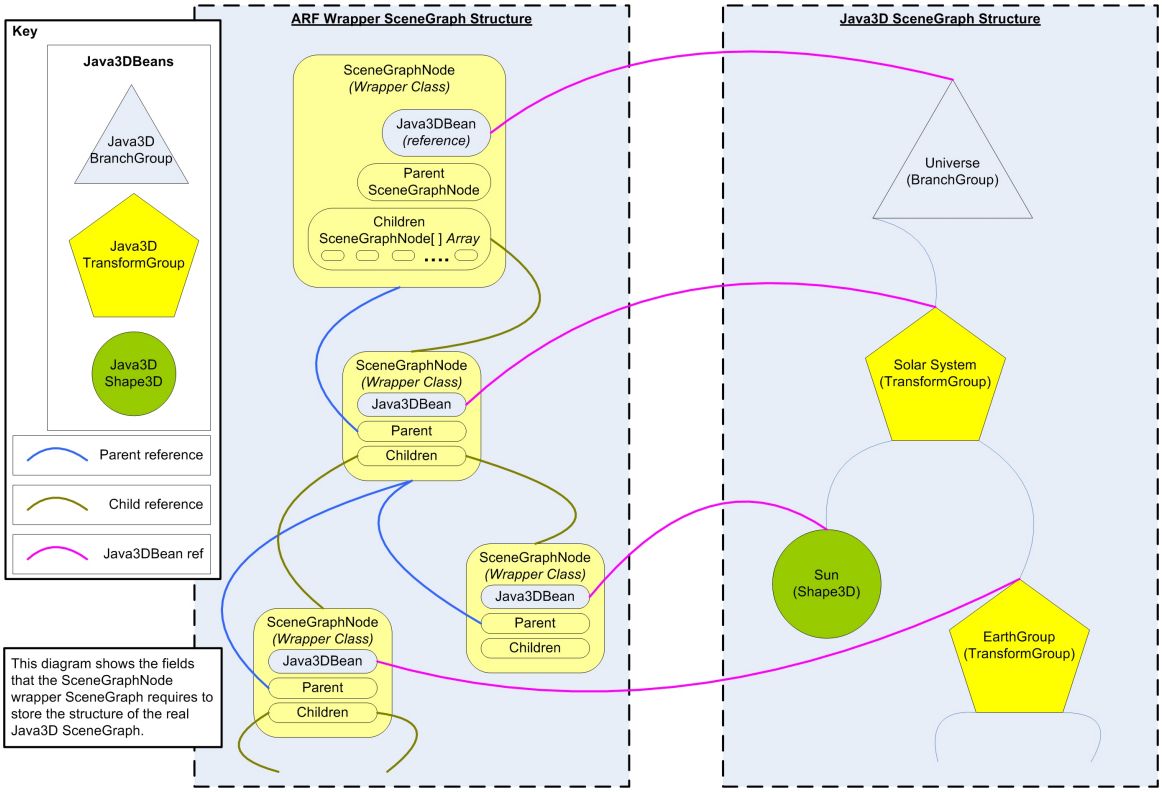


Figure 5.20: ARF's SceneGraph wrapper structure

The wrapper SceneGraph node can be traversed recursively just like the Java3D SceneGraph. In order that the SceneGraph Structure is saved, the `Project` class holds a JavaBean property of the root SceneGraphNode; the `Project` class can then be serialised. More details on the `Project` class can be found in section 5.3.9.

5.2.2 Java3D SceneGraph Modifications

In addition to using a SceneGraph wrapper to provide persistence for the Java3D SceneGraph structure, ARF has needs to circumnavigate some restrictions placed on SceneGraph alterations by Java3D. For example, Java3D limits which types of nodes can be added/removed to/from a *live* SceneGraph. A *live* SceneGraph is one which is currently attached to the renderer. Also, some properties of certain Java3D Nodes have to have special capability bits set to allow reconfiguration of their parameters whilst *live*. The capability bits have to be changed before a node is made *live*. Therefore, ARF either has to restrict the user in the types of SceneGraph node they can add/remove/modify, or, ARF has to find a way of implementing the required func-

tionality by getting around Java3D's restrictions. The Java3D restrictions are there so that, by restricting certain operations, Java3D can perform optimisation routines on the SceneGraph to make it render faster. However, there is not much to be gained by most of these optimisations on modern computer systems and therefore ARF can disable most of them and work around the others.

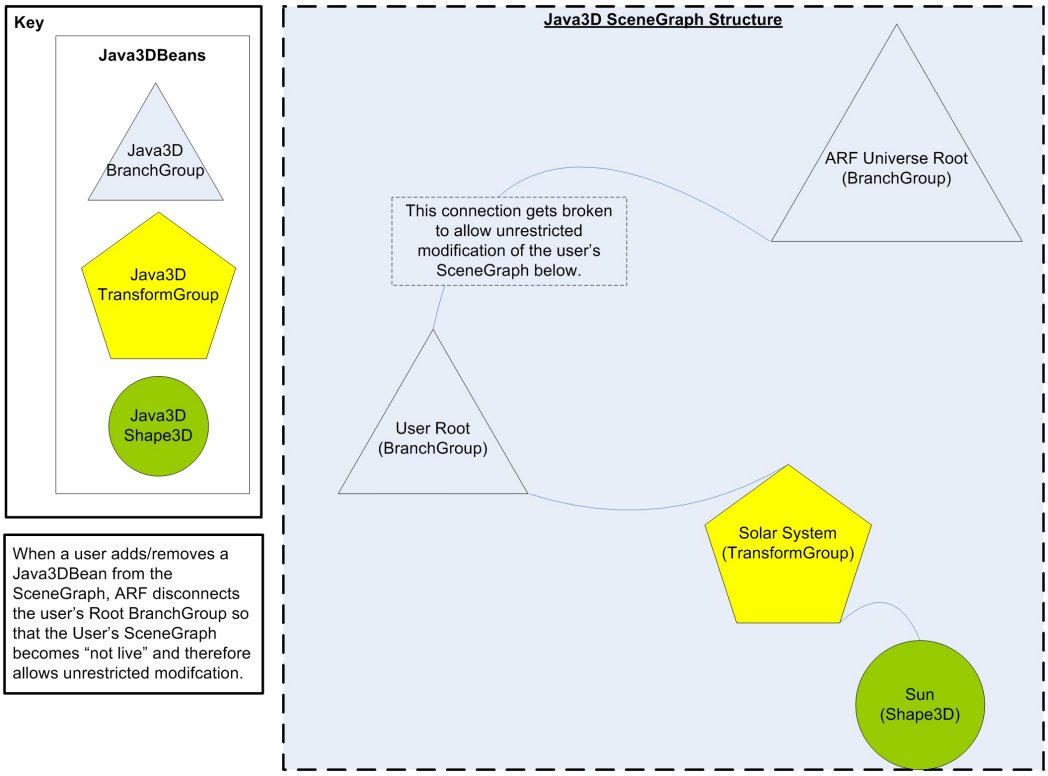


Figure 5.21: Working around Java3D SceneGraph Restrictions

The first major problem is that only BranchGroup nodes can be attached/detached to/from a *live* SceneGraph. However, the entire SceneGraph connected to the top level BranchGroup node can be modified if the user's top level BranchGroup is made *not live*. Therefore, ARF provides a work around which consists of providing a single BranchGroup node at the top of the SceneGraph which is visible to the user. When the user tries to add or remove a Java3D node, ARF momentarily disconnects the BranchGroup at the root of the user's SceneGraph from the higher level SceneGraph, thus making the lower level SceneGraph momentarily *not live*, then the lower level SceneGraph can be modified. This happens transparently and occurs when the user invokes a function such as: *add new Bean, cut/copy/paste, delete, make live/make*

dead. If the user deliberately makes a sub-graph *dead*, then the Java3D SceneGraph is severed, i.e. if they want to make certain parts invisible. However, the wrapper SceneGraph keeps its structure but changes a flag to indicate whether the Java3D node it represents should be connected to the SceneGraph or not. So the ARF wrapper SceneGraph makes a note as to whether a node is supposed to be *Live* or *Dead* but otherwise still keeps track of which nodes are children of which. Consequently, ARF is able to restore the correct state of the Java3D SceneGraph when a project is loaded from file by preserving the *live* or *dead* states of each node. Figure 5.21 shows ARF's internal top level Java3D SceneGraph which allows the SceneGraph below the user's "root node" to be modified without the restrictions of the Java3D subsystem.

5.2.2.1 Java3D Capability Restrictions

Java3D also places restrictions on each node type using capability bits which have to be enabled if those parameters need to be changed when the SceneGraph is *Live*. Only the programmer needs worry about these since it is they who will have to set the correct capabilities on their Java3DBean if they need to adjust some of its Java3D settings. ARF provides help by providing a convenient method for setting all the capabilities of common Java3D classes which the programmer can use if they wish. This class is called `J3DCapability` and is in package `framework.j3d` in the ARF4Auv core project (more detail on the class hierarchy of ARF is given in appendix C.1).

5.2.3 ARF Java3D Components

The 3D virtual world components of ARF would hopefully provide the same information for the programmer as real world inputs allowing straightforward substitution. By using JavaBeans combined with Java3D, a JavaBean can in essence be given a virtual physical appearance. Therefore, Java3DBeans can be thought of as real physical entities as well as processing components. Java3D *Behaviour* nodes provide animation, timing and event systems which can be used to animate and allow Java3DBeans to interact with their virtual surroundings. Java3DBeans can thus have a physical appearance, an interactive property and a processing capability as well as being used to create interesting visual animations. Consequently, Java3DBeans are very useful

for providing building blocks such as sensors, behaviours and visual aids which can be used to build interesting virtual environments for use in HIL testing scenarios. Many task specific Java3DBeans have been created for the example use cases detailed in chapter 7, and a full list of the core Java3DBeans needed by ARF is given in appendix C.

5.2.3.1 Sensor Interaction with Virtual World

The SceneGraph provides methods for doing collision detection and ray tracing (via Java3D picking). Most of the simulated sensors within the "Use Cases" chapter rely on collision/pick testing of the SceneGraph. For example, an altimeter sensor would do a collision test directly downwards to see if the pick ray (theoretical line in 3D space) intersects with any geometry. The distance of the nearest intersection is then reported as the altitude value. A simulated sonar would rely on ray tracing hundreds of pick rays in the virtual world to gather enough data to build up a realistic sonar image. A similar example is terrain following which was shown earlier in figure 2.5. It is up to the programmer to provide their own Java3DBeans which provide the functionality of the required simulated sensors, although many sensors for underwater applications are already provided and discussed in chapter 7.

5.3 JavaBeans Backbone

ARF's constituent parts that provide the JavaBeans functionality have been largely discussed in earlier sections. However, creation of JavaBeans and using them with ARF has not. The JarRepository hasn't been discussed, and it is this that loads Java JAR (Java Archive File) files and catalogues the available JavaBeans. The JarRepository is is a special JAR file loader which scans through a Java `manifest` file looking for entries for classes marked as JavaBeans. The `manifest` file is contained within the JAR file and is manually edited by the programmer to have entries indicating which classes are JavaBeans. In order to load programmer's JAR files, ARF provides a user interface for importing JAR files and adding their contents to the JarRepository.

This section provides some background on JavaBean creation and how JavaBeans

are imported into ARF. It then discusses ARF's extra features and Java interfaces which help the programmer create more advanced JavaBeans. Following on from this, section 5.3.7 focuses how to use Java Events with JavaBeans and how the programmer can utilise OceanSHELL for external communication of events. A section on the ARF `Project` class describes how ARF saves and loads the project's status and serialises the JavaBeans. This section also outlines how some of the Java interfaces from section 5.3.6 are used when loading and saving.

5.3.1 Creating JavaBeans

JavaBeans are Java classes which adhere to coding conventions that allow the class to be inspected using introspection techniques, using Java reflection, to see which fields and methods can be adjusted by the user. This is very useful because it allows Java object configurations to be changed using a graphical property editor. For each JavaBean a list of changeable features and their associated graphical property editors are displayed on the PropertySheet. JavaBeans also provide the functionality of dynamic loading. Dynamic loading allows the user to import Java JAR files (which contain many different Java classes some of which are JavaBeans) on the fly. These JavaBeans are then added to a repository allowing them to then be instantiated and inserted into an ARF project.

In order for a standard Java class to be used as a JavaBean it must conform to some programming conventions.

```
Class MyClass implements Serializable {
      YourClass theClass;
      public void setTheClass(YourClass theClass) {
            this.theClass = theClass;
      }
      public YourClass getTheClass() {
            Return this.theClass;
      }
      /** Zero argument constructor */
      public MyClass() {
            // do something
      }
}
```

Figure 5.22: Example JavaBean code

The standard programming conventions for a JavaBean are as follows:

- Class must have a zero argument constructor (unless a special way of instantiating the bean is specified in the bean's associated PersistenceDelegate - see JavaBean tutorial [64]).
- All bean attributes should have "*get*" and "*set*" methods and be of the form shown in figure 5.22.
- The class must implement the Java `Serializable` interface so that the Bean can be saved to persistent storage if needs be. Note: the XMLEncoder which saves the JavaBeans uses the BeanInfo and PersistenceDelegate objects associated with each JavaBean in order to save the data correctly and does not use the programmer's own custom serialisation methods since only JavaBean properties are actually saved (see section 5.3.3 for more information about PersistenceDelegates).

5.3.2 BeanInfo Objects

JavaBeans can optionally provide an extra class which allows the programmer to control which properties of the JavaBean are available for modification by the user. These files are called BeanInfo classes and they have a similar name to the real Java class except with BeanInfo appended to the end of the name. These can be used in conjunction with JavaBean coding conventions to provide a pruned subset of JavaBean properties for the user.

For instance, "`MyClass.java`" has BeanInfo object "`MyClassBeanInfo.java`" in the same package to describe what properties are available for alteration. BeanInfo classes also allow the user to specify different levels of visibility for bean attributes and methods.

BeanInfo classes are not easy to write by hand, however most IDEs, such as Netbeans, have a BeanInfo editor which allows the programmer to select which properties to include in the BeanInfo class and what their visibility should be, e.g. *Preferred, Expert, Hidden* etc.

The BeanInfo file is not necessary since JavaBean properties can also be inferred using the coding conventions. However, BeanInfo objects are useful because the user can also choose which fields not to display since some fields may be inherited from a super-class of which the programmer does not want some properties to be visible. *For a more detailed explanation of JavaBeans refer to Sun's JavaBean tutorial [64].*

5.3.2.1 Creating a Manifest

In addition to using the JavaBean coding conventions, the programmer must highlight which Java classes contained in a JAR file are JavaBeans. In all JAR files, there is a file called "`manifest.mf`" which the programmer must alter to include paths to the classes which are JavaBeans. This manifest should be included with the class files when building the JAR file. When using IDEs such as Netbeans, the manifest is generated automatically when the user instructs Netbeans to build a project. However, there is usually a separate manifest in the Netbeans project directory which the programmer can add their own entries to. This is then appended to the manifest created by the ANT build script and included in the JAR file. The manifest is usually located in the Netbeans project directory created by the user. Figure 5.23 shows what a manifest with JavaBean entries should look like. The *Name* corresponds to the relative path of the JavaBean in the Jar file. To indicate that this is a JavaBean a 2nd line under the first is required which says "JavaBean: True". When ARF loads JAR files it interprets this manifest to find out which classes are JavaBeans.

5.3.3 Custom Instantiation of JavaBeans

JavaBeans can be saved to persistent storage. This can be done by serialising the JavaBean properties since everything which defines a JavaBean should be represented as a JavaBean attribute. Therefore, in order to save a class all that has to be done is to save the individual JavaBean attributes of each class. These can be identified by using introspection or the BeanInfo objects discussed previously. JavaBeans do not use Java's standard serialisation routine for saving and loading the state of a class, since the classes are saved as XML. Thus the programmer must use a different method of custom instantiation than that of the standard `serializable/externalizable`

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: 1.5.0_06-b05 (Sun Microsystems Inc.)
X-COMMENT: Main-Class will be added automatically by build

Name: OceanLIB/arf/ARFCoordConverterGroup
Java-Bean: True

Name: OceanLIB/arf/messageIO/AutotrackerMultiBeamOutput
Java-Bean: True

Name: OceanLIB/arf/messageIO/CADCACMsgListener
Java-Bean: True

Name: OceanLIB/arf/messageIO/CollidedNodeMsgOutput
Java-Bean: True

Name: OceanLIB/arf/messageIO/NavMsgListener
Java-Bean: True

Name: OceanLIB/arf/messageIO/NavMsgOutputBehaviour
Java-Bean: True
```

Figure 5.23: Example JavaBean manifest

interfaces, since the `private void read/writeObject()` methods which the user can override using these interfaces are not called when using the JavaBean system. The JavaBean system circumnavigates the standard serialisation routines and uses its own method using introspection via the Java Reflection API.

XML is used to save the contents of a JavaBean to persistent storage. This is done using Java's `XMLEncoder` to save classes to file, and Java's `XMLDecoder` to load classes from file. A simple XML document for a JavaBean with one `String` field called "*name*" might look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0" class="java.beans.XMLDecoder">
     <object class="framework.j3d.ARFCanvas3D">
      <void property="name">
       <string>GuiCanvas3D</string>
      </void>
     </object>
</java>
```

When JavaBeans are saved/loaded to/from persistent storage the attributes of

each class are accessed directly, hence not using the associated *get* and *set* methods. Consequently any work which may be done by the class when these *set/get* methods are called is bypassed since the data is written directly to the class attributes. This can be a problem when more work needs to be done to ensure that a class's fields are initialised correctly, e.g. if a class representing a person has attributes height and weight, some extra work may be done to populate non-JavaBean fields such as density. If this class is loaded using `XMLDecoder` the density field will not be calculated since neither the `setHeight()` or `setWeight()` methods were called as the class attributes were modified directly. Therefore the programmer needs a special way of doing extra work when a class is loaded in order to initialise the class variables properly. If serialisation was used the programmer would create methods shown in figure 5.24 and perform some extra work there. However, when using JavaBeans there is another way, using PersistenceDelegates but ARF also provides an extra convenience method discussed in section 5.3.4.

```
private void writeObject(java.io.ObjectOutputStream out)
      throws IOException;
private void readObject(java.io.ObjectInputStream in)
      throws IOException, ClassNotFoundException;
```

Figure 5.24: Methods for custom serialisation and instantiation of Java classes

For more information on using PersistenceDelegates see the how-to [68], which provides detail on how to provide custom initialisation of JavaBean attributes. This provides a similar function to the `writeObject()` and `readObject()` methods which the user can use for serialisation and externalisation - see the Bean Persistence section in Sun's JavaBean tutorial [64] for information on this. Please note that these serialisation and externalisation methods are not called by the XMLEncoder when ARF loads or saves the Bean objects so the user must use PersistenceDelegates for custom instantiation of JavaBeans.

5.3.4 Initializable Interface

ARF provides another convenience method for doing this instead of using the more complicated PersistenceDelegates. ARF provides a special interface which should be

implemented by any class needing custom instantiation. It should be noted that JavaBeans used by ARF can use this interface, however it will not be supported by other JavaBean development environments. It is therefore a convenience method for ARF only.

The interface which provides this extra capability is the `Initializable` interface. The `void initiate()` method of this interface is called just after a Bean has been loaded (more information on this is in section 5.3.6).

More information on JavaBeans and further tutorials can be found on Sun Microsystems' website [69].

5.3.5 Importing JavaBeans using the JarRepository

In order to import JavaBeans into ARF the JavaBeans must be located in a JAR file with appropriate entries for the JavaBeans declared in the "`manifest.mf`" file within the JAR file. In order to import this JAR file into ARF, the user must select "*Manage Repository*" from the "*File*" menu of ARF. This displays a GUI, shown in figure 5.25, for adding and removing JAR files from the repository.

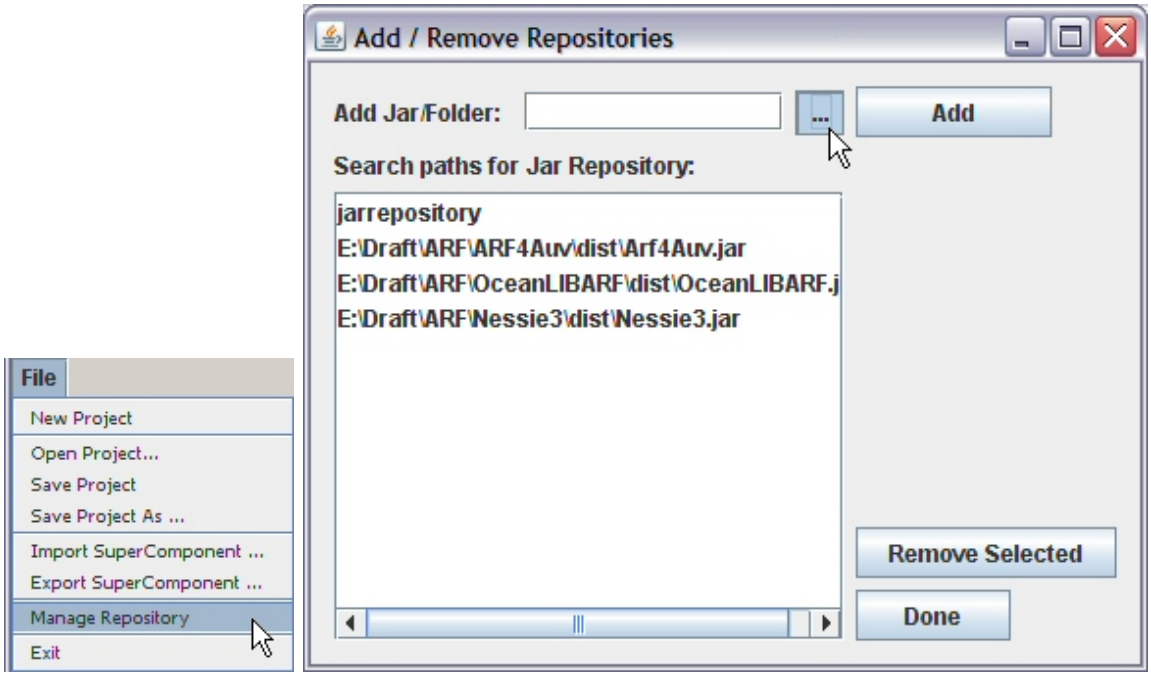


Figure 5.25: The JarRepository Manager

5.3.6 ARF Interface Extensions

In earlier sections it has been mentioned that ARF provides capabilities to programmers that go beyond the standard functions of JavaBeans. Many of these enhancements are to provide further flexibility over the standard JavaBeans PropertySheet functionality. JavaBeans classes can basically do anything, just like a normal Java class can. However, sometimes this can lead to problems such as when creating and destroying threaded JavaBeans continuously in ARF; this is perfectly likely behaviour of a user creating and changing a scenario in ARF. The difference between ARF and some other JavaBeans programming environments is that ARF uses "live" editing, meaning that whilst connecting the components together the components are in fact running like any normal instantiated Java class. Due to the nature of Java this can cause problems when JavaBeans use thread capabilities. This chapter gives an overview of some of the extra capabilities ARF provides, including how to use threaded JavaBeans appropriately. All the Java interfaces discussed are located in the ARF's Java package "`framework.interfaces`".

5.3.6.1 Threads

A thread in Java allows a class to execute a program loop continuously and concurrently with other classes. This is very useful for having components which do something periodically, e.g. an animation. However, Java uses garbage collection to manage and clean up its memory. Garbage collection is done by freeing memory which belongs to objects not in use anymore. Objects are determined to not be in use anymore when they no longer referenced by any other object. Therefore the garbage collector deletes that object instance from memory. The problem with threads is that they do not get freed by the garbage collector when they are no longer referenced by any other object, because they can still perform functions without needing to be called by another object, e.g. because the main loop of the thread is continually executing. Thus, if a user deletes a threaded JavaBean from the BeanBoard this will not automatically mean that it gets garbage collected, and as a consequence the thread will keep running in the background and could potentially interfere with resources needed by other JavaBeans running in ARF.

In order to avoid this problem ARF can do two things upon deletion of the threaded JavaBean from the BeanBoard:

- Kill the thread using potentially dangerous Java methods; this is not favourable since a thread could be killed whilst locking local computer resources and then cause other JavaBeans not to work properly if they access those same resources.
- Try and tell the JavaBean to stop running as a thread using a "soft" stop within the JavaBean itself. This is ideal since it leaves the original programmer to decide when and how to exit the thread cleanly, i.e. terminating from its "`run()`" method.

There is still a problem with both these methods; dangerous killing of the thread is only an option if the threaded object actually extends the Java `Thread` class and therefore allows ARF to call the `Thread.stop()` method. However, a class can instead implement an interface called `Runnable` which can then be executed by creating a `Thread` object to start it, ARF then has no access to the `Thread` object itself and therefore cannot call the "`Thread.stop()`" method for it; this creates a problem since threads are still able to run after they have effectively been deleted by the user from the BeanBoard. The only way this can be protected against is to provide a special Java interface for the programmer to implement when using threads which inherit from the `Runnable` interface instead of the Thread class itself. The interface ARF provides is called `StartStopable` and has one method:

```
void setEnabled(boolean)
```

The programmer must make sure that when `setEnabled(false)` is called that their thread code exits from its `run()` method and thus terminates. Calling `setEnabled(true)` should cause the thread to start again.

5.3.6.2 Initializable Interface

The `Initializable` Interface is located in package "`framework.interfaces`" and provides two methods:

- `void initiate()`

- `void aboutToSave()`

The ability to do work after loading and before saving is made easy for the programmer by implementing the `Initializable` interface. If the `Initializable` interface is implemented by the JavaBean class then the "`void initiate()`" method is called just before the Bean is added to the BeanBoard (just after it has been loaded) and the "`void aboutToSave()`" method is called just before the Bean is saved. These methods allow a Bean to do more work just after the data is loaded and just before the data is saved. However, the user should try and use the PersistenceDelegates method if they wish for their JavaBeans to be fully functional in any other JavaBeans environment as these interfaces are ARF-specific.

These methods are also called by the `Project` class when a project is loaded or saved. Therefore the programmer can use the `Initilizable` interface to make sure that their JavaBean loads and saves the correct information, i.e. the programmer can use these methods to populate certain data structures which need to be saved and then process these data structures again upon loading.

5.3.6.3 ARFBean

The "`framework.interfaces`" package also provides a standard interface called ARF-Bean which provides many extra functions used by ARF. These include a *name, description* and *type* of a JavaBean which can be altered by the user when using ARF. These make Beans more user friendly since they can be given more meaningful names than the standard Java class name. ARFBean also inherits from the `Initializable` interface and the `ARFInfo` interface which describes the *name, description* and *type* methods. The `ARFBean` interface is located in package "`framework.interfaces`" and provides the following methods:

- `void initiate()` - `Initializable` interface
- `void aboutToSave()` - `Initializable` interface
- `boolean isReady()` - this is used to show the user that the Bean has all its requirements met and is therefore ready to run. This must be implemented

by the programmer. This should return `true` if the JavaBean has all of its dependencies met.

- `void setDescription(String description)` - allow the user of ARF to give each individual JavaBean a description of its function or what it is used for.
- `String getDescription()`
- `String getName()`
- `void setName(String name)` - allows the user of ARF to give each JavaBean a friendly name so that they can easily identify it on the BeanBoard and when selecting it as a parameter during guided construction.
- `void setType(String type)` - this is a user definable *type* field. This allows the user to give each of their JavaBeans an associated friendly type which can be used by pruning algorithms to identify objects on the BeanBoard or the SceneGraph of a user type. This is used in section 7.2 for giving 3D virtual components a user type, so that a 3D object sensor can then report what type of object it has detected.
- `String getType()`

The programmer can either implement the ARFBean interface themselves or they can inherit from a convenience class which already implements the interface, such as "`SimpleARFBean`". `SimpleARFBean` is a fully functioning JavaBean which only provides a name, description and type property. The "`isReady()`" method always returns `false` so the programmer must override this method if they extend the `SimpleARFBean` class.

5.3.6.4 Menu Interface

The PropertySheet allows for JavaBean properties to be edited if there is a suitable propery editor. However, sometimes the programmer may wish to provide their own configuration GUI to provide extra help in configuring and controlling the JavaBean. The `Menuable` interface provides a mechanism for the programmer to provide extra

functions to the user through the use of the *right-mouse-click* menu on the BeanBoard. If a JavaBean implements the `Menuable` interface then the programmer's specific menu is added to menu of the BeanBoard for that Bean, as shown in figure 5.4. From the programmers own menu buttons they can invoke any functionality or even bring up their own seperate GUI for configuring the JavaBean. The methods of the `Menuable` interface are:

- `javax.swing.JMenu getMenu()` - The programmer must return their own instance of a `JMenu` class which has added to it `javax.swing.JMenuItem` type children. The user can then click these menu items to invoke special functions. The returned JMenu is appended to the *right-mouse-click* menu of the BeanBoard for this particular JavaBean.

5.3.7 Java Event Communications

JavaBeans can communicate by direct method call, i.e. having other JavaBeans as reference parameters which are connected using guided construction; or the other way is to produce event objects which are sent to registered event listeners. JavaBeans provide mechanisms for easily adding event listener functionality. In essence an event listener merely has to implement the interface methods for the type of event, e.g. a method similar to `void handleEvent(OurEventObject theEvent)`, the event producer then cycles through all registered event listeners and calls the `void handleEvent(OurEventObject theEvent)` method of that listener and passes the event object as a parameter to this method. The JavaBean tutorial [64] describes the JavaBean coding conventions that the programmer must follow in order for the event listener properties to be recognised by the JavaBean introspection techniques. If the JavaBeans producing the events adhere to the programming conventions then the event properties will be available for the user to register event listeners with as described in section 5.1.4. The user is then able to register event listeners of the appropriate type with the JavaBean which produces the events.

The most useful aspect of event listeners over direct method call is that there is no limit to the number of possible registered event listeners, and they can be chained together to provide an information flow which converts one type of event to

another. This chaining of event listeners is often used in ARF to convert an event to an OceanSHELL message which is then output onto Ethernet for other distributed processes to listen to, and vice versa, thus event producers can also be event listeners as well. The other advantage of event listeners is that there is no limit to the amount of event interfaces which a JavaBean can implement, so the same JavaBean can listen for multiple types of event.

5.3.8 OceanSHELL Communications

ARF provides JavaBeans and Java interfaces making interaction with OceanSHELL straightforward. As discussed in the literature review section 4.1.5, OceanSHELL provides packet based distributed communications over Ethernet. JavaSHELL is a Java version of OceanSHELL that uses XML message definitions to describe what is in the message packets. JavaBeans can use JavaSHELL to communicate by creating and populating messages or by listening to messages. ARF provides an extra layer to JavaSHELL which allows the user to choose which OceanSHELL message queues they connect their JavaBeans to. Since an OceanSHELL queue functions on a port between 1000 and 65535 there can be many potential queues which a JavaBean could connect to, so ARF provides some conventions which the programmer should use if they wish to use OceanSHELL. The programmer could use the standard JavaSHELL classes internally in their JavaBeans, but since the user would be unable to manually join receivers and senders to different message queues using ARF, this leads to JavaBeans being more complex and less flexible

To simplify the use of OceanSHELL, ARF provides a set of wrapper classes which extend the JavaSHELL classes. These classes are part of the `OceanLIB.arf.oceanshell` package. The user should consult documentation on JavaSHELL [70] since ARF's `OShQ` JavaBean is merely a subclass of JavaSHELL's `OShMessageTarget` class which has been altered to provide JavaBean functionality, i.e. it allows the user to join JavaBeans to OceanSHELL queues using the PropertySheet in ARF (in fact guided construction is used for this task).

OceanSHELL interfaces and message listeners:

- Two MessageQueue classes:

- `OShQ` - this provides the hardware interface to OceanSHELL. Each queue can transmit and listen for messages on a single port between 1000 and 65535 (ports less than 1000 are generally reserved for operating system use). This class implements the `OShReceiverStub` and `OShSenderStub` interfaces.
- `OShBridge` - this provides a communications software bridge within ARF, i.e. it works in place of an OShQ but does not transmit or receive external to ARF. This is useful if JavaBeans use OceanSHELL but the user wants to keep the ARF environment isolated and only allow internal OceanSHELL communications. This class also implements the `OShReceiverStub` and `OShSenderStub` interfaces.

- Four interfaces:
  - `OShReceiver` - an interface which must be implemented by JavaBeans wishing to receive OceanSHELL messages. This allows the JavaBean to be connected to an `OShReceiverStub` (OShQ/OShBridge) and receive OceanSHELL messages from it.
  - `OShSender` - this should be implemented by JavaBeans wishing to send OceanSHELL messages as this allows the JavaBean to be connected to an `OShSenderStub` (OShQ/OShBridge).
  - `OShReceiverStub` - this interface should be implemented by any JavaBean which provides OceanSHELL receiving capabilities similar to OShQ or OShBridge
  - `OShSenderStub` - this interface should be implemented by any JavaBean which provides OceanSHELL sending capabilities similar to OShQ or OShBridge

- Two example classes:
  - `SimpleOShReceiver` - this implements the OShReceiver interface as a JavaBean and provides the basic methods. The user should extend this class and override the `void handleMsg(OShMsg msg)` method.

– `SimpleOShSender` - this implements the OShSender interface as a JavaBean and provides the basic methods for connecting an OshSenderStub. The user should extend this class and send out the appropriate OceanSHELL message type by overriding the `sendMsg(OShMsg msg)` method.

OShQ provides a real OceanSHELL message queue for communication over Ethernet, where as OShBridge provides internal communication only. The *****Stub interfaces provide the functionality that any OceanSHELL implementation must implement. These interfaces provide a level of abstraction that allow for OceanSHELL messages to be transmitted over any medium and doesn't restrict it to just Ethernet like the `OShQ` JavaBean does. Only senders and receivers registered with the OShBridge will receive messages as they are not sent outside of ARF.

The OShReceiver and OShSender interfaces should be implemented by any class wishing to either receive data from an OShQ or to send data to an OShQ respectively. SimpleOShReceiver and SimpleOShSender provide example implementations of the JavaBean methods defined in the interfaces for use in the user's own JavaBean classes. The user's JavaBeans can then be connected to the OShQs using ARF. The example implementations also subclass ARF's `SimpleARFBean` JavaBean to provide simple naming and description conventions implemented by most JavaBeans in ARF (see ARF Interfaces section 5.3.6). Figure 5.26 shows the source code for SimpleOShReceiver abstract class.

5.3.9 The Project class

Figure 4.8 shows the how the "`Project`" class holds the data of the all the SceneGraph nodes and JavaBeans in ARF. The `framework.beans` package is where the `Project` and associated classes are located. The `Project` maintains all the references to the JavaBeans currently in use and some extra configuration data for the ARF GUI. However, for Java3DBeans simply keeping a reference is not enough. Java3DBeans are not full JavaBeans since they are incapable of recovering all their previous state information, because some of it is lost when the Java3DBean is saved. This is because the Java3D classes do not adhere to all the JavaBean coding conventions for prop-

```java
package OceanLIB.arf.oceanshell;

import framework.beans.SimpleARFBean;
import oceanshell.OShMsg;
import oceanshell.types.*;
/**
 * This is an example of what an OShReceiver class should implement
 *
 * @author Benjamin Davis
 */
public abstract class SimpleOShReceiver extends SimpleARFBean implements OShReceiver {
    protected long[] msgIDs;
    protected OShReceiverStub receiver;

    /** This method is called when a message is received of the appropriate type.
     * e.g.
     *  public abstract void messageReceived(OShMsg msg) {
     *    if (msg.GetMsgID().longValue() != 383) return; // double check the msg ID if
want
     *    // then do something exciting
     *  }
     *
     */
    public abstract void messageReceived(OShMsg msg);

    public void setOShReceiverStub(OShReceiverStub receiver) {
        if (this.receiver != null) this.receiver.removeListener(this);
        this.receiver = receiver;
        receiver.addListener(this, msgIDs); // here we register the message IDs with the
OShRecieverStub. This field should be set in the constructor - is an array of message IDs
which we are interested in.
    }
    public OShReceiverStub getOShReceiverStub() {
        return receiver;
    }

    public abstract void aboutToSave();
    public abstract void initiate();
}
```

Figure 5.26: Extending SimpleOshReceiver to provide specific OceanSHELL functionality

erties which need to be saved. The main problem is that the Java3DBeans do not restore the parent/child relationship between the nodes in the SceneGraph. For this reason, ARF has a wrapper SceneGraph that keeps track of the real Java3D SceneGraph structure. The wrapper SceneGraph is then able to be saved and restored to its previous state. For every Java3DBean, there is a separate "`SceneGraphNode`" wrapper class which keeps track of which children and parents each Java3DBean has. This wrapper SceneGraph is then saved to disk and all the parent-child relationships restored upon loading using the `Initializable` interface describe in section 5.3.6.

5.3.9.1 Loading and Saving the Project

The `ProjectUtils` class provides the mechanisms for saving the `Project` class and the associated BeanBoard and SceneGraph. The `Project` class is also a JavaBean which implements the `Initializable` interface. The main ARF GUI calls the `ProjectUtils` class to load the XML project file selected by the user in ARF. Once the `Project` instance has been loaded the `initiate()` method is called on it. The Project's own `initiate()` method then calls the `initiate()` methods for all the JavaBeans on its BeanBoard before ARF's GUI allows the user to start using the project. When the `initiate()` method is called on one of the `SceneGraphNode` wrapper classes, it rebuilds the real underlying Java3D SceneGraph as represented by the structure of the wrapper SceneGraph. Since the SceneGraph is a tree structure, each `SceneGraphNode` calls the `initiate()` method of its child nodes, thus recursively initialising and building the real Java3D SceneGraph.

The project class also maintains some other properties associated with the user interface of ARF which need saving so that they can be restored next time the Project is loaded from XML. These include:

- `GUICanvas3D` - The canvas which is currently controlled by the camera selection toolbar of ARF and the MouseController, i.e. the main canvas if there is more than one.

- `Camera` - The camera Java3DBean which is connected to the main canvas which the MouseController moves around.

- `MainComponent` - This is the Swing component which ARF adds to its main viewing window. This is usually a `Canvas3D` object, unless the user has added a `ConfigurablePanel` which enables them to add as many `Canvas3Ds` and other Swing components to it as they like, i.e. this property maintains which component is displayed in ARF's centre window.

- `BeanBoard` - A reference to the current BeanBoard and all associated JavaBeans.

- `RootNode` - The root node of the SceneGraphNode wrapper class which holds the construction information for the Java3D SceneGraph.

Due to the JavaBean nature of the `Project` class, by saving it to XML all the associated JavaBean instances referenced by the `Project` class are saved also. Then all the objects referenced by the JavaBeans are also saved. Thus the saving of the Project's JavaBeans is recursive with the `Project` JavaBean being the top level which references all the rest. Thus the `Project` class only has to maintain a single reference to the root SceneGraphNode JavaBean, since this recursively references all the rest via its children array.

The advantage of saving the `Project` as a generic JavaBean is that this can be loaded by any other Java application, meaning that the ARF surroundings, i.e. the GUI, can easily be replaced by a slightly different user interface. Therefore, if the programmer wishes, ARF can be used to construct scenarios and then the scenarios can be used in the programmer's own front-end application. Hence, ARF can be used for rapid prototyping of software as well.

5.3.10 BeanBoard and Scenegraph facilities

As described previously, the `BeanBoard` class is referenced by the `Project` class and is part of the same `framework.beans` package. The BeanBoard maintains an array of JavaBeans which has a getter/setter method so that all the beans in the array are stored when the `BeanBoard` class is serialised. Aside from keeping track of which JavaBeans are in use, the BeanBoard also provides functions for finding out which JavaBeans have dependencies on other JavaBeans. This functionality is useful when creating SuperComponents (see section 5.1.7) as JavaBeans selected for export require

that all referenced JavaBeans also be exported, and the BeanBoard provides methods for returning all JavaBeans referenced by a JavaBean which is to be exported. This list is then added to the JavaBean export list in the SuperComponent creation GUI. The method for finding linked beans is:

```
ArrayList getLinkedBeans(Object bean)
```

It works by building a stack of all JavaBeans which are referenced by the first JavaBean. JavaBeans are popped off the top of the stack and then recursively searched to find all JavaBeans referenced by each one on the stack. All JavaBeans found are added to a complete list. Duplicates are ignored if they already appear in the complete list and are therefore not added to the stack as they will get processed anyway, or have already been; this stops loops occurring.

5.3.10.1 SuperComponent Helper Algorithms

The BeanBoard also provides the methods which facilitate guided construction in ARF. Guided construction relies on being able to find JavaBeans which match the Java type of a property on the PropertySheet and then displaying a list of current possible JavaBeans and potential JavaBeans to the user. The method in the BeanBoard is the same as the one in the JarRepository: `ArrayList getInstancesOf(Class type)`. The `Class` class associated with every Java class instance is acquired by calling `instance.getClass()` which is a method of all instantiated Java objects. This `Class` instance is then used by the BeanBoard and JarRepository to check for subclasses in each of them which match the type of this Property. These simple methods provide much help since the user doesn't have to lookup which JavaBeans are of the correct type in a manual, thus saving a lot of time. It also allows the user to quickly create JavaBeans to try as parameters; if it's the wrong functionality the user can simply delete the JavaBean from the BeanBoard. This is so quick that it allows users to quickly try different ideas without having to worry about wasting any time.

5.3.11 Swing Component Addition

Adding extra Swing components was discussed in the Project section 5.3.9, but not fully demonstrated. Generally, ARF doesn't need to use visual 2D JavaBeans, i.e. those which extend the `javax.swing.Component` class, since ARF provides visual elements through the use of the 3D virtual environment using Java3DBeans. However, the user may still wish to create and arrange their own visual JavaBeans in the same way they would using a standard JavaBeans development platform, such as BeanBuilder[65]. ARF provides the ability to change the central 3D virtual world view for a `ConfigurablePanel` instead. Multiple visual JavaBeans can be added and arranged on this panel. This allows for displaying multiple canvas3Ds connected to different cameras in the virtual environment as well. It is therefore possible for the user to add their own JavaBeans which extend either the AWT or Swing GUI classes. These components can then either be displayed solely, instead of the canvas3D object, or a custom layout JavaBean, such as `ConfigurablePanel`, used to display multiple components in different locations. Graphical JavaBeans are added to the BeanBoard in the exact same way as any other Bean (see section 5.1.1).

5.3.11.1 Setting the Main Component

ARF has a concept of a "main component" JavaBean. This is the component which is at the highest level of the displayed AWT/Swing `Component` hierarchy, and is displayed in the main view area of ARF (like the canvas3D is by default). The user can choose any JavaBean, extending the `Component` class, to set as "main component", thereby filling the screen. By default the canvas3D object is set as the main component and thus fills the full middle section of ARF's GUI. However, if the user has their own component they wish to display instead, then the user must set their JavaBean component to be the "main component". To do this:

1. Select the JavaBean on the BeanBoard (which must extend `java.awt.Component`)

2. Click the *right-mouse-button* to bring up the menu.

3. If the bean the user has selected is a subclass of a `AWT/Swing` class then the user

will see an item in the menu called *setMainComponent*. This should be clicked in order to make this the "main" component.

By doing this the old main component will be replaced with the selected component and displayed, i.e. the canvas3D normally displayed will disappear and be replaced by the component the user chose to display.

5.3.11.2 Adding and Arranging Multiple Components

Quite often the user will need to be able to display both the canvas3D object and multiple other AWT/Swing components and arrange them on the screen. Using the *SetMainComponent* function of the BeanBoard only one component can be displayed at once. However, a special component is provided which allows the user to add multiple components and arrange them. This component is called `ConfigurablePanel` and must first be added to the BeanBoard and then set as "main component" in order to use it.

5.3.11.3 ConfigurablePanel How-To

Once the ConfigurablePanel has been set as the "main" component, the user should see a blank screen. This is because nothing has yet been added to the panel. In order to add components to the panel the user must first select the ConfigurablePanel on the BeanBoard and click the *right-mouse-button*. There will be a menu called "*options*" and under that menu will be a button to set the panel "*editable*" (see figure 5.27).

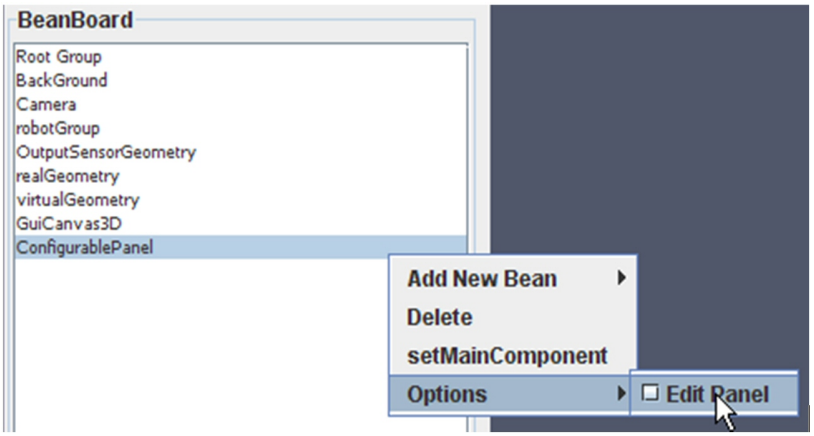


Figure 5.27: Setting ConfigurablePanel editable

In order to add another visual JavaBean to the panel, the user must *right-click* the mouse on the blank panel area on the screen; a menu will appear allowing the user to either "*add a component*" or "*remove a component*", for any component on the BeanBoard. The component to be removed will be the one the user has the mouse cursor hovering over. The user can then drag the new component to their desired position and resize it by selecting the bottom-right hand corner of the component and dragging it.

Once all components have been added and positioned the user should switch the ConfigurablePanel back to "*not editable*". This can be done by in the same way as making it "*editable*" described before.

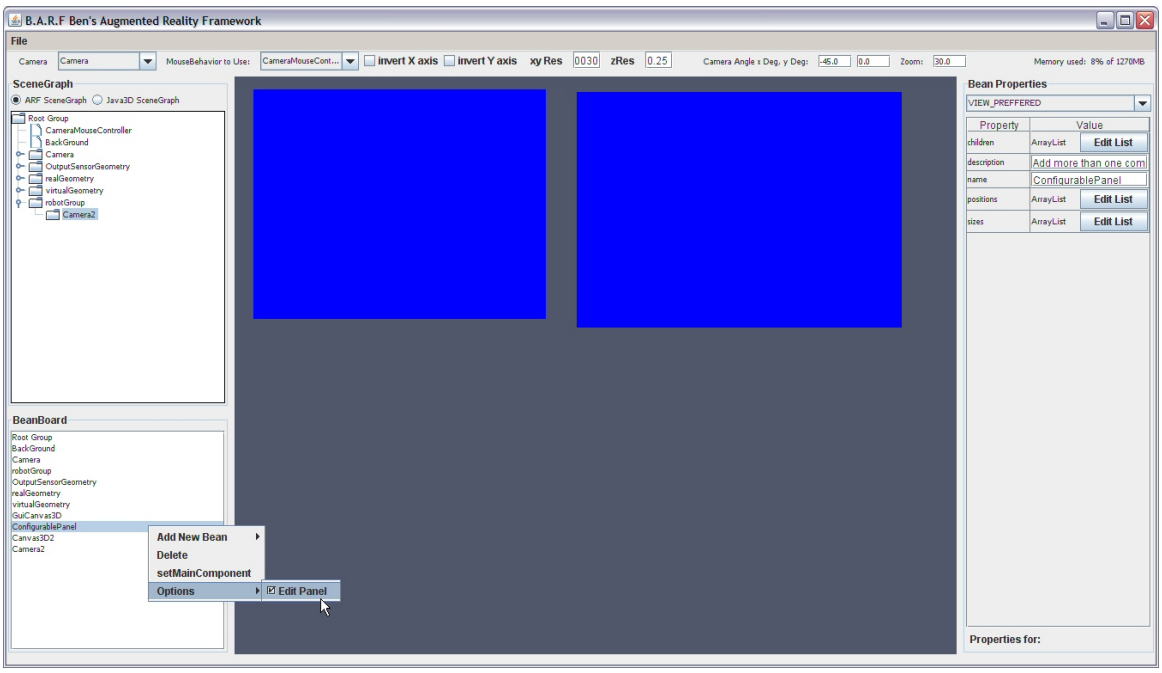


Figure 5.28: Setting ConfigurablePanel not editable

Figure 5.28 shows the ConfigurablePanel in edit mode, and shows the position of two canvas3D objects. The canvas3D objects need connecting to separate camera objects in order to render anything. This is done by selecting the camera objects and editing their "GuiCanvas" property on the PropertySheet to render to the desired canvas3D JavaBean. Figure 5.29 shows the ConfigurablePanel once editing is finished.

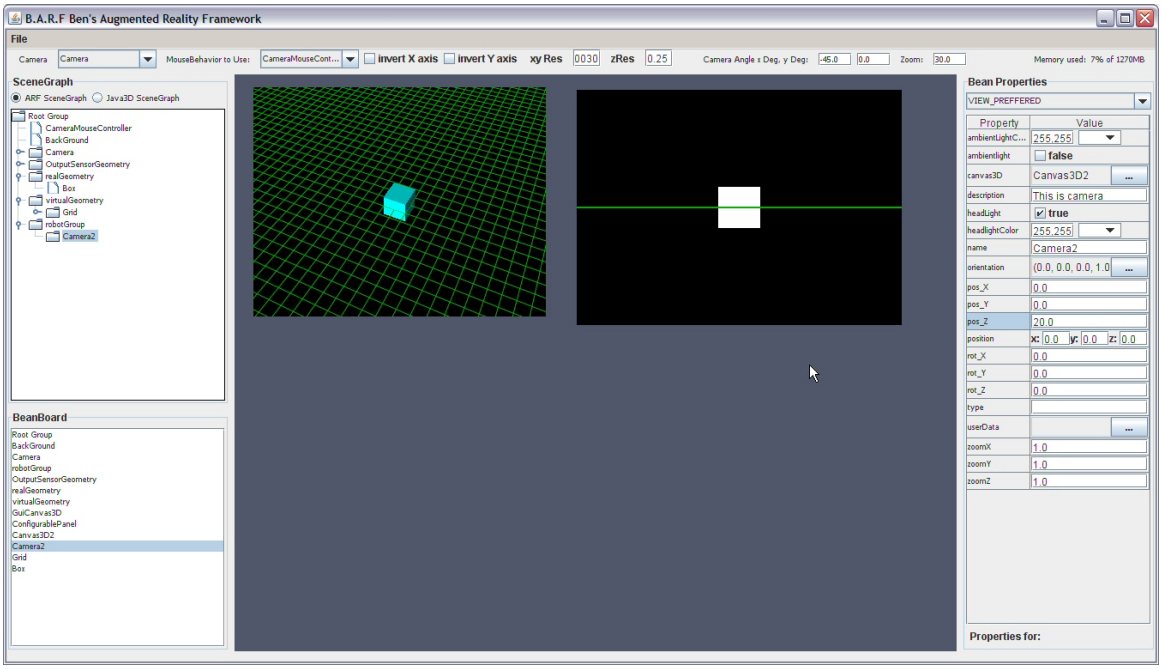


Figure 5.29: Two canvas3Ds

5.4 Summary

This chapter described how the various underlying parts of ARF's architecture are harnessed by the user interface and how they interact with each other beyond the detail given in the functional design and implementation design chapter. Some examples of how some of the JavaBeans and Java3DBeans work were touched upon; however more explanatory real examples of scenario types and the components required are given in the use cases in Chapter 7. These examples show how ARF's user interface and JavaBeans architecture really help improve testing and development of underwater platforms.

The main aspects of ARF's implementation were described here, however, there are far too many JavaBeans already implemented to be covered in this section and some of them are very application specific. A complete listing of JavaBeans already created for use with ARF is given in appendix C.1.

This section covers many of the programming classes and PropertyEditors used by ARF. At this point it should be clear how the constituent parts of ARF provide the features which give ARF its flexibility, extendibility and added performance over other platforms. It should also be evident that the underlying technologies have been

harnessed in such a way which provides an abstract layer for producing virtual environments for many different applications. The underlying JavaBeans architecture means that ARF is able to be used to create many conceivable types of virtual environment. The JavaBeans already created for ARF provide a base set of functions which provide many uses in the field of underwater vehicles as well as other domains as a by-product. The specific uses of the current JavaBeans created for ARF are discussed in chapter 7 which should be the reader's next port of call after reading the performance testing in chapter 6.

Chapter 6

Testing

There are various different categories of tests required which will demonstrate the different features of the architecture. For the most part testing will be carried out in a comparison fashion. i.e. test subjects will be required to undertake exercises using ARF and another competing method for a particular test. The tests will be designed having the ARF architecture in mind so as to exploit the features of the architecture without focusing too much on the possible individual uses of the architecture. The individual uses of the architecture will be demonstrated in section 7 which use these tests as a foundation for why high performence was acheived. However, it should be noted that this is an informal study to provide backing evidence of the performance increases achievable when using ARF. The performance increases indicated here are reflected in the use case scenario construction times in chapter 7.

6.1 Test Design

In order to test the fundamental principles of ARF, specific test exercises need to be undertaken by subjects in order to quantitatively highlight the advantages of using ARF over the standard method of creating a test scenario. As well as showing the benefits of ARF, the tests also need to highlight any overheads required to use ARF, and hopefully show them to be small in contrast to the overall gain. In order to provide such quantitative information, comparative tests against the standard way of programming a test scenario are required. These exercises will be undertaken by a

spectrum of people with different experiences and knowledge to provide a representative cut of the population of people who are likely to use ARF in their day to day operations. This will probably give varying results, though hopefully when ARF is contrasted to the standard programming approach on an individual basis, there will be a clear increase in performance.

Time will be used as the main testing metric as this will provide an overall guide to the complexity of the exercise. The user will be required to note start and end times for each exercise so that data can be analysed later. In addition to time, the user will be asked to rate the exercise on a number of other quantifiable qualitative factors such as complexity, ease of use of the interface, reliance on documentation, how good the documentation was etc. These will of course be specific to the test exercise itself. Furthermore the user will be asked for their comments about the exercises they performed in order to shed some light on the answers they gave to the statistical metrics, i.e. what they found difficult etc.

6.1.1 Performance Metric

In order to provide useful test results, there needs to be a metric which is used to measure the performance of ARF. Due to the nature of ARF, quantitative results are hard to obtain because it is difficult to measure how much "better" ARF is over something else. ARF is supposed to provide an intuitive interface to a low level architecture provided by JavaBeans and Java3D by abstracting the user from the code and displaying information in human readable form. In order to measure the performance gain of using ARF, a real user has to carry out simple tasks in both ARF and the standard approach to writing test programs, i.e. in an IDE (Netbeans). Thus performance is defined as how many times faster ARF is over using the Netbeans IDE.

$$Performance = 100 \times \frac{TimeTakenUsingIDE}{TimeTakenUsingARF}$$

The prediction is that ARF should be faster than Netbeans and consequently the performance metric for ARF should be greater than 100%. Over multiple test subjects and multiple exercises, an approximation for the average performance of ARF should be reached. Since there are multiple features in ARF which lead to the performance

increase, different tests will have to be carried out in both environments on specific areas in order to highlight how much effect each of ARF's features have over the traditional methods. It is important to keep in mind that ARF is a platform which is supposed to increase the flexibility and extendibility for the creation of virtual environments for mixed reality testing, so that mixed reality environments can be created, changed and extended efficiently, i.e. not having to start from scratch every time.

6.1.1.1 Quantitative Metrics

These are the quantifiable factors which will be measured by the performance tests:

1. Construction Speed Test - How long does it take to create a scenario.
2. Extendibility Speed Test - How long does it take to change the scenario.
3. Scenario Change test - How long does it take to change from one scenario to another.
4. Property Change Test - How long does it take to quickly adjust certain properties and run the program again.

6.1.1.2 Qualitative Metrics

Although time spent is a good overall indication of the performance of the architecture, it doesn't highlight usefulness of specific features of ARF, e.g. such as usability of the user interface. Therefore, as well as timing the specific exercises the user undertakes, the user will also be marked and give feedback on:

- Task completion
- Understanding of the task
- Amount of assistance required and reason for assistance
- Measures of usability of certain features which a specific exercise is testing.

These measures are quantified within a range e.g. task completion is measured from 1 to 5 where 5 is fully completed and 1 is nothing done. These measures will help to identify why a certain task took the time it did. These measures are required because the test subject may have varying knowledge and understanding of both ARF and Netbeans (Java) and thus anomalous results will need clarification. In order for these tests to be representative of the programming community, varying levels of expertise will be required for the test subjects. Therefore, candidates will be asked to rate their experience in various fields for correlation later with the results.

In addition to these measures the test subjects will be required to provide more quantitative feedback about the features of ARF and Netbeans in order to compare how intuitive, or easy to use, a feature is. This allows for another type of performance metric to be realised between Netbeans and ARF which provides a quantitative value for what is in effect a qualitative answer e.g. the usability of event listeners in ARF/Java. This is a qualitative measure but can again be quantified between the values 5 (very intuitive and easy to use) and 1 (not intuitive and very hard to use). This information can later be correlated with the timed performance metrics to explain why a task took a long/short amount of time.

6.1.2 Test Goal

The goal of these tests is to provide supportive evidence of the performance increases attainable by using ARF over the standard way of programming a test harness using an advanced Integrated Development Environment (IDE). The performance increases shown by these tests can then be used to help explain how ARF helped the various use cases described in Chapter 7.

In essence, these tests should show that by using ARF users are able to build scenarios quicker due to spending less time making mistakes and consulting documentation; consequently highlighting the effects of ARF's Guided Construction mechanisms, SuperComponents, PropertySheets and intuitive user interfaces. These tests should show that the user does not need as much expertise using ARF as they would if programming using an IDE to undertake the same task.

6.2 Test Design Analysis

In order to make sure that the test design was unambiguous, intuitive and not to time consuming, the test itself required testing. In order to do this, the test was tested by a few people to obtain constant feedback about the design of the exercises. As well as making sure that the test exercises were well written, it also served as a means to test the features of ARF and make sure that the exercises themselves were feasible bearing in mind that for most of the test participants this would be their first exposure to ARF. Thus, the features of ARF needed to be tested and any problems or bugs rectified before the test participants could undertake the test.

6.2.1 Analysis Findings

The analysis proved to be invaluable as many usability issues with the test were discovered. Mainly, there were errors performed by the test subject which occurred due to details of the exercise not always being very obvious. For example, one of the questions referred to some information given in another exercise, and because this information was not under the same exercise the reader forgot about the information. Since the exercises are supposed to be testing the features of ARF compared with those of an IDE, rather than the participant's reading ability, the exercises needed to be written in the most straight forward way making details of the task as obvious as possible. Therefore, some of the exercises needed re-writing due to these issues.

As well as highlighting problems with the exercises, problems with ARF itself were discovered. Mostly these were programming bugs, because participants were trying new things with certain ARF components which it had not occurred to the programmer to protect against. Other problems were highlighted with the usability of the user interface, the problems included:

- The Java3D SceneGraph displayed some of the available Java3DBeans in red - This was originally used to indicate which of the Java3DBeans were subclasses of the Java3D BranchGroup class as Java3D only allows BranchGroups to be added and removed to/from a *"live"* SceneGraph. However, ARF transparently provides a work around for this, but the red highlighting remained until it

became apparent that the user thought that red objects meant bad objects. Thus the colouring has since been removed.

- The user uses the left mouse button to select components on the Java3D SceneGraph and on the BeanBoard. However, when the user uses the right mouse button to display the options menu for that JavaBean they held the mouse over, it wouldn't select a new JavaBean even if the mouse was over a different JavaBean. This is not the same functionality as other applications employ as normally the right mouse button would also do the selection if the mouse was in a new position from where the user previously clicked with the left mouse button. This led to Java3DBeans getting lost as they were getting added to a different SceneGraph node than the user expected.

- The Export SuperComponent function was also added to the Java3D SceneGraph popup menu so that the user could quickly export sub-graphs easily as SuperComponents for use later.

- Improvements to the supplied API specification (JavaDocs) were required; better description of methods. In general, programmers have to put up with a lot of badly documented code so this wouldn't be anything out of the ordinary. However, in order for fairness the competing IDE should be as supported as possible. This doesn't necessarily affect ARF that much, however the JavaDocs for JavaBeans can still be useful.

6.3 Test Exercices

The actual test exercises themselves are included in Appendix D and a copy of the results sheet design is included in Appendix E. A breakdown of what each test is for is given below. Exercise 1 focuses on using ARF, Exercise 2 using Netbeans and Exercise 3 focuses on the overheads of creating a JavaBean for use with ARF.

1. (a) Build a virtual environment using ARF - demonstrate how quickly a virtual world can be created for a very simple purpose and how easy it is to do.

(b) Add sensor simulation and OceanSHELL message output capabilities - this test is to compare how quickly JavaBeans can be connected together with ARF due to its guided construction. This also demonstrates how easy JavaBeans are to configure using the PropertySheet editors. This is contrasted against programming by hand with NetBeans.

(c) Scenario save and SuperComponent export test - designed to show how quickly configurations can be created and saved as different projects, or be exported as SuperComponents for use later. This is contrasted with creating duplicate classes for different configurations using NetBeans.

(d) Making an alteration and saving the new configuration (speed test), contrasted against the manual approach in Netbeans.

(e) Create a scenario to use two robots instead of just one (as before). The idea is the user imports the SuperComponent they previously exported to demonstrate how quickly SuperComponents can be re-used.

(f) A simple test to see how long it takes the user to switch from one scenario to another using the load project feature.

2. (a) Connect the Java classes and configure them to the same specifications as in **1b** by programming using NetBeans. This can be directly compared to the performance of **1b**.

   (b) Make a new scenario with slightly different OceanSHELL configuration. This is designed to show the performance improvement in ARF for creating different configurations quickly. When many of these add up, the difference could be potentially huge.

3. (a) Convert the Java class created in exercise 2 into a JavaBean. This is designed to show the overhead of creating components for ARF.

   (b) Import the newly created JavaBean into ARF. This is also designed to show how much overhead there is in adding a JavaBean to ARF's repository.

   (c) Connect the JavaBean to the other JavaBeans required by it, using the PropetySheet. This is designed to show the user how easy it is to make

JavaBeans highly modular and assemble them with ARF. This should be a relatively quick task due to ARF's guided construction. This also tests that the JavaBean created in exercise 2 was programmed correctly.

6.3.1 Overheads and Requirements

In addition to the test exercises the test participants will be provided with extra material to help them throughout the test.

For ARF there was a tutorial provided which covers all the main features of how to use ARF. However, this only acts as a reference guide as there will not be time for the participant to work through the tutorial prior to the test. Instead a narrated video demonstration of how to use ARF is provided as this covers all the main features of ARF covered in the chapter 5. The ARF tutorial is on hand if the test subject needs to look up how to use something in ARF during the test.

For the programming part of the test, the user is expected to have used an Integrated Development Environment (IDE) such as NetBeans or similar. A brief introduction to the different parts of NetBeans is given by the examiner before the test starts. There will also be the JavaDoc API specifications for the classes/JavaBeans which the programmer is supposed to use during the test. Netbeans provides levels of help to the programmer such as displaying the JavaDocs on screen and providing automatic method name and variable name completion to aid them. In addition, if the test participant is completely stuck they can ask the examiner for help. However, all provided help is noted down for each exercise of the test to highlight which were most difficult and reasons are given in each case.

A stop clock will be started once the user has read each exercise and are confident they are ready to start. All results and feedback are recorded on an answer sheet displayed in Appendix E.

6.4 Results Discussion

In this section the results are compared, discussed and displayed by an appropriate visualisation. For a complete listing of individual results for all the test exercises see

Appendix F.

Generally, the averaged results are of main interest since it is the average performance increase which is the key indicator for ARF's success. Averages will be used primarily for the qualitative measures; however anomalous results will be discussed separately. The following section headings describe what is being tested, and where relevant refer to the feature being demonstrated from section 3.1.3.

6.4.1 Task Complexity

Task complexity is a qualitative measure which is decided by the test participant. It would be expected that the measure of task complexity should be proportional to task understanding and task completion. Figure 6.1 shows the time taken by each participant to complete the same programming task using ARF and using Netbeans. Figure 6.2 shows the respective complexity ratings for Exercises **1b** vs **2a** for ARF and Netbeans respectively, with complexity a measure from 0 (very straightforward) to 4 (very complex). This comparison shows that generally the task complexity was higher for Netbeans than for ARF. This is corroborated by the time extra time taken to complete the task in Netbeans.

The general trend depicted in figures 6.1 and 6.2 shows that for each participant the complexity of the task in Netbeans was either equal or greater than it was in ARF and the times of each participant in each task reflect this trend strongly. This correlation allows for emphasis to be put on qualitative feedback for which there is substantially more data. This proof of qualitative feedback allows for more generalised feedback results to be more useful and carry more strength. This is key since most of the proof for ARF's advantages exists in the form of the use cases in chapter 7.

Table 6.1 shows the averaged values for complexity, understanding, completion, time taken and usability of event listeners and object references for Exercise **1b** vs **2a** contrasting ARF and Netbeans respectively. The correlation between task completion, understanding and complexity shows clearly that the exercise was harder for the user when using NetBeans. From the feedback usability results it is clear that NetBeans was less helpful when connecting classes and using event listeners than ARF. This is due to the guided construction efforts in NetBeans being limited and often the

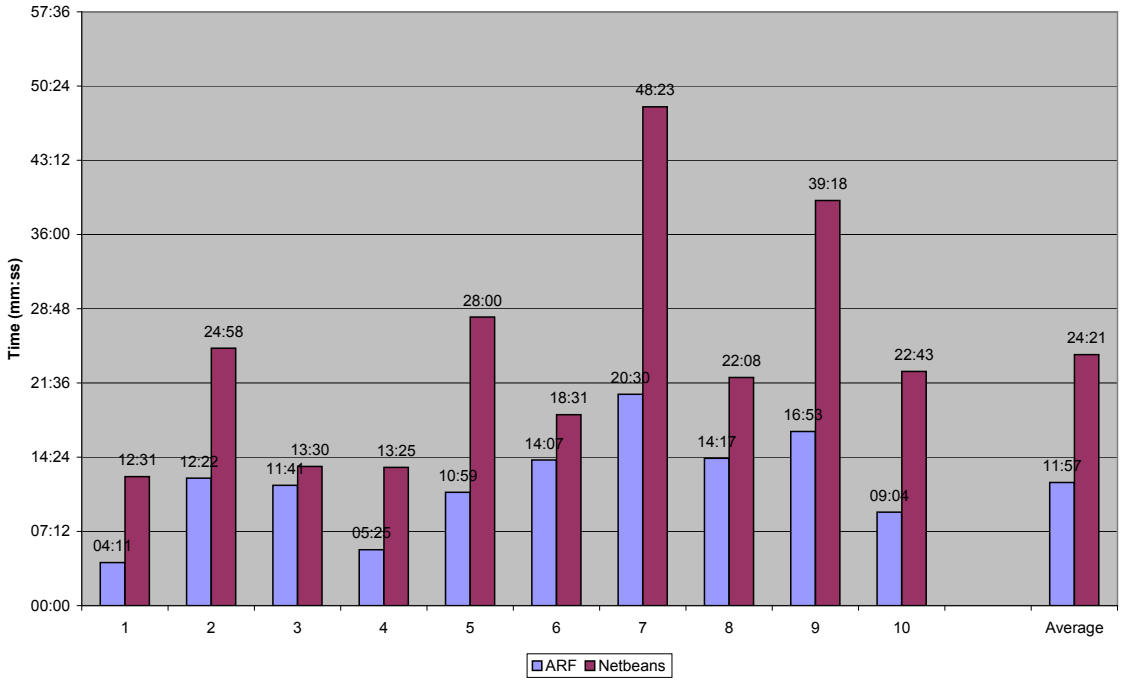


Figure 6.1: Construction Time Comparison between ARF and Netbeans

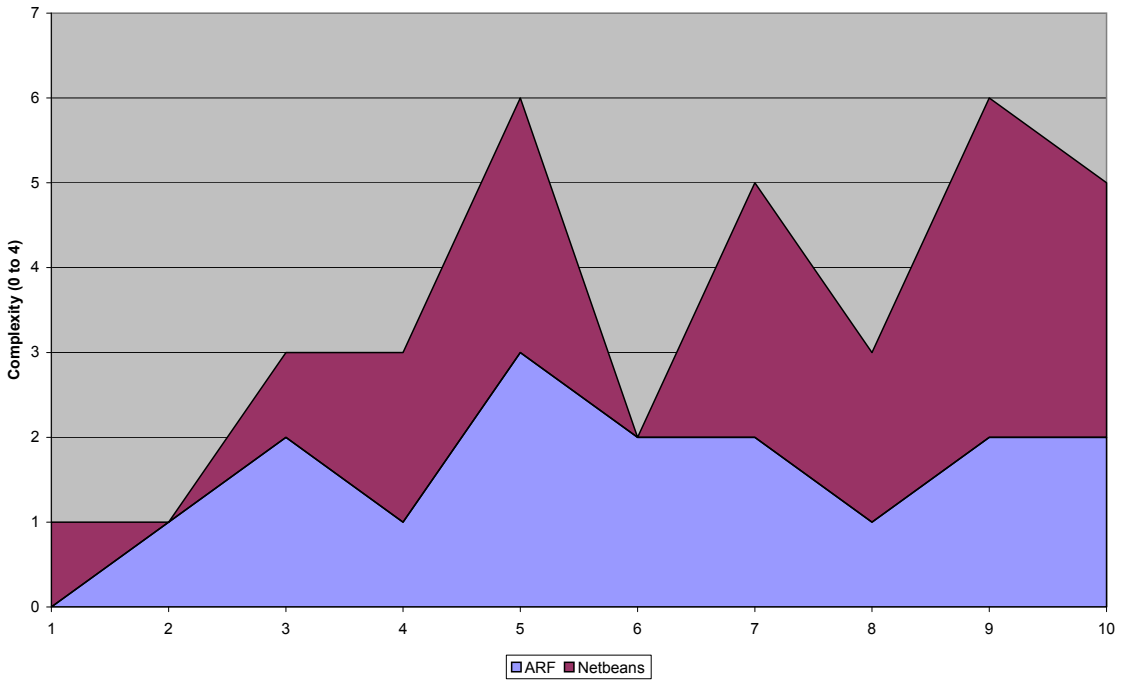


Figure 6.2: Perceived Construction Complexity Comparison between ARF and Netbeans

Task	ARF	NetBeans	Units
Average Complexity	1.6	1.9	0-4
Average Understanding	4.8	4.6	1-5
Average Completion	4.8	4.5	1-5
Event Listeners Usability	4.3	3.6	1-5
Object Reference Usability	4.7	3.2	1-5
Consulted Documentation	0	2.5	No. Occasions
Assistance Required	0.8	0.8	No. Occasions
Average Time	716.88	1460.74	Seconds

Table 6.1: Averaged Results for Exercises 1b and 2a

programmer would have to consult documentation of the classes themselves. The problems this causes are clearly reflected in the time taken, since ARF on average took less than half the time Netbeans did. Therefore, a preliminary estimate for the performance of ARF compared with NetBeans can be calculated:

$$Performance = 100 \times \frac{1460.74}{716.88} = 204\%$$

The contrast between exercise **1b** and **2a** is the main key performance indicator (KPI) for ARF since this exercise uses the exact same classes and documentation in both cases and the only difference being that ARF is used instead of NetBeans. It is expected that ARF be a lot quicker than NetBeans due to the help that it gives the user. Also, ARF is designed for constructing virtual test environments with dynamic configurations that can be changed quickly and easily without the overhead of re-design commonly occurring in test-bed creation. The other exercises highlight why ARF is more useful than the standard approach of re-design and re-program using an IDE.

6.4.2 Configurability and Extendability tests

In order for ARF to be truly useful it still has to meet the requirements of easily extendable and re-configurable. It is these concepts which are not provided by standard monolithic testing platforms which lead to huge development overheads, especially in

the case of mixed reality testing. Therefore ARF has to demonstrate that it provides these features in an easy to use and intuitive manner. JavaBeans provides the basics for allowing ARF to be easily extendable due to allowing users to create and use their own components. ARF provides the user interfaces which make these tasks more straightforward. In addition ARF provides other features such as SuperComponents which further the modularity and flexibility of virtual environment configurations, allowing users to create functional units which can be used time and time again. These features of ARF are now tested by comparing against the simplest equivalent approach. The equivalent approach normally undertaken is by a programmer, using an IDE, to make modifications to an existing test environment. This relies on the programmer knowing which modules provide the functionality they require, how to use them, and also knowing how to modify the existing monolithic program.

6.4.2.1 Configuration Speed Tests (feature 6 and 5)

Reconfiguration speed can be compared by contrasting exercises **1d** and **2b**. Figure 6.3 shows the time taken to make a simple adjustment and save as a new scenario, or a new class in the NetBeans case. Number 12 on the graph shows the averaged value for ARF and NetBeans. This shows that including the time to actually save the file ARF is still about twice the speed of NetBeans. The more alterations that need making at once, the more ARF's performance would increase further since adjusting the properties takes seconds, however the time it takes to save as a new file name takes longer. Figure 6.4 shows the save and load times for a project from exercises **1ci** and **1f** respectively, with the average save and load times being roughly the same at 17 and 15 seconds. If the average save time is removed from the average time to make an alteration and save, which is *37 seconds*. The average time to adjust a property can thus be calculated as *20 seconds*. The advantage of ARF is that when a property is changed, it is changed instantly and no recompilation has to occur. In contrast to programming manually where the software has to be stopped, make alterations, then recompile and then run the software again.

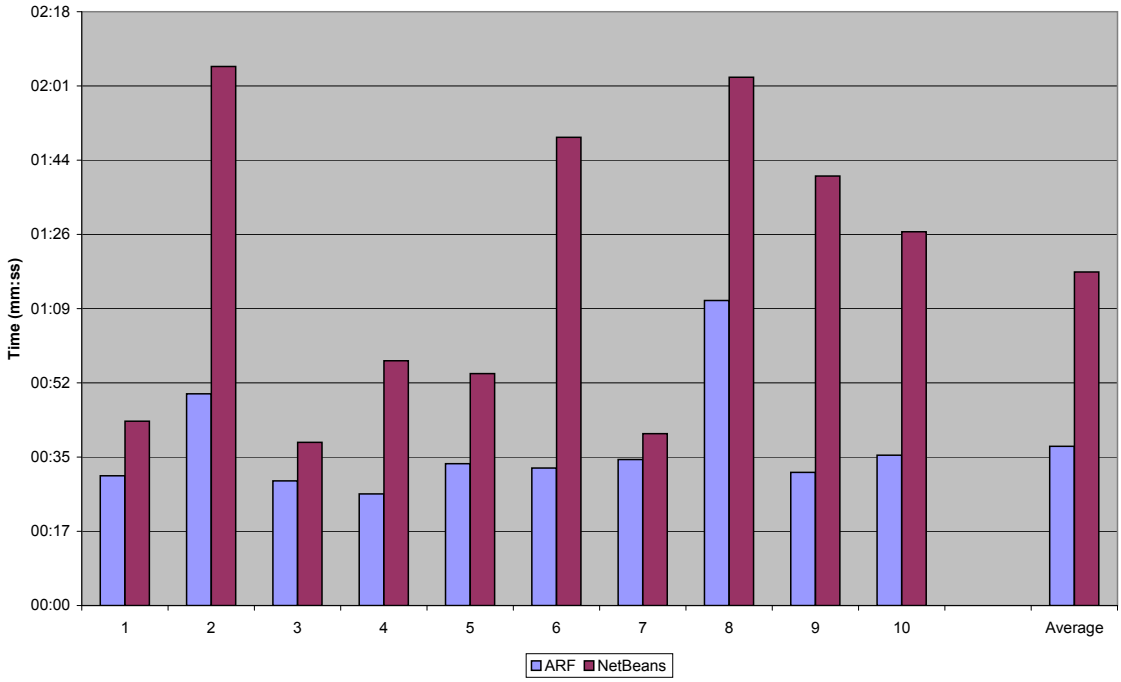


Figure 6.3: Reconfiguration Time Comparison

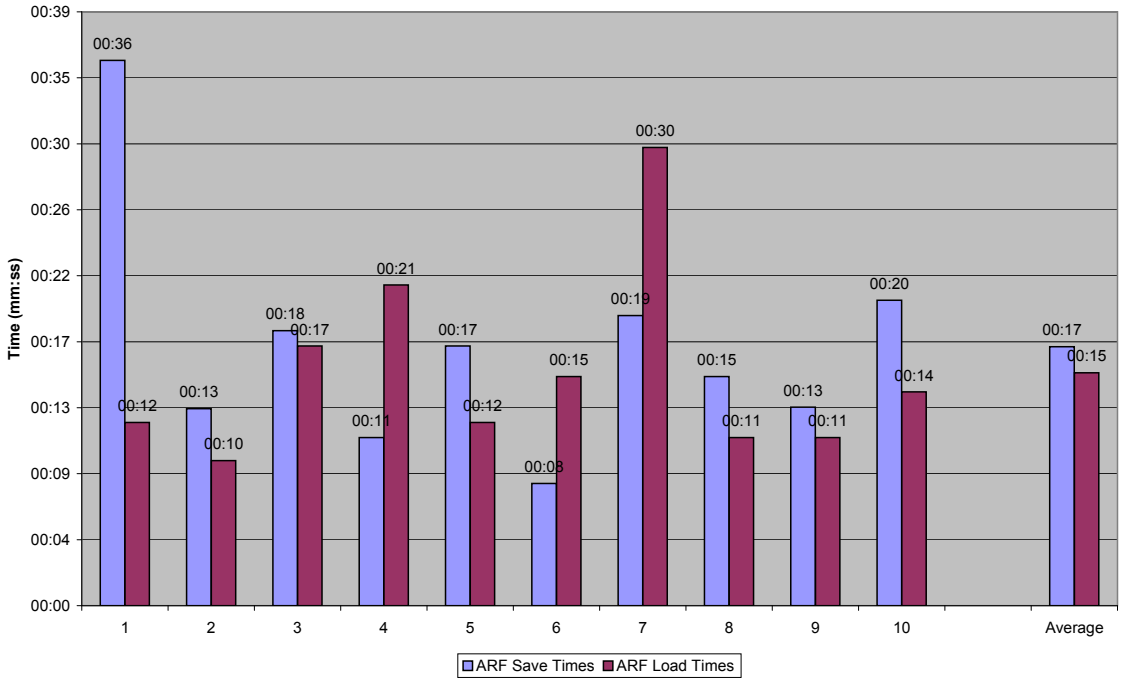


Figure 6.4: Save and Load times for an ARF Project

6.4.2.2 Extendibility Testing (feature 2)

In order for ARF to be truly useful as a testing environment for potentially all types of mixed reality testing scenarios, it needs to be easily extendable. Extendibility has been the main challenge for all testing environments. This is because some capability will evolve which means that a redesign of the software is needed due to a certain restriction. Due to ARF's JavaBean backbone it doesn't have any restrictions for usage, but is optimised through the use of Java3DBeans and OceanSHELL for the creation of AR environments which can be then used for sensor simulation and HIL testing. JavaBeans can be designed for potentially any usage and then communicate with other JavaBeans through the use of event passing, JavaBean object references or through an external communication protocol such as OceanSHELL. In order to extend ARF the user has to create their Java classes so that they adhere to the JavaBean conventions. This can be done when programming real modules for the vehicles so that they can then also be used on the simulated vehicles in the ARF virtual environment. If modules already exist they can either be ported to JavaBeans, or a JavaBean stub can be created for ARF which communicates to the externally running module. This communication can be done using OceanSHELL or any other third party communication mechanism. It is completely up to the programmer whether to use an existing communication protocol or write another one for use with ARF, since ARF places no restrictions. Therefore an external module can easily interact with ARF if needed. In essence this is exactly what HIL testing does.

Exercise **3a** tests how long it takes for the user to transform their normal Java class into a JAR file containing JavaBeans which can be imported into ARF. Figure 6.5 shows the varying times to create a JAR file, with one JavaBean entry, using NetBeans (*Entry 12* on the graph represents the average time). This test is designed to demonstrate the low overhead of creating a component for ARF. Bearing in mind this is the first time the test participants have ever undertaken this type of task, an average time of *198 seconds* is fairly good. Once one JavaBean entry has been added to the `manifest` then its far quicker to add all subsequent ones since the programmers knows exactly how to do it. It should be noted that this is a one time overhead, and from this point onwards the component could be used in ARF many times, making

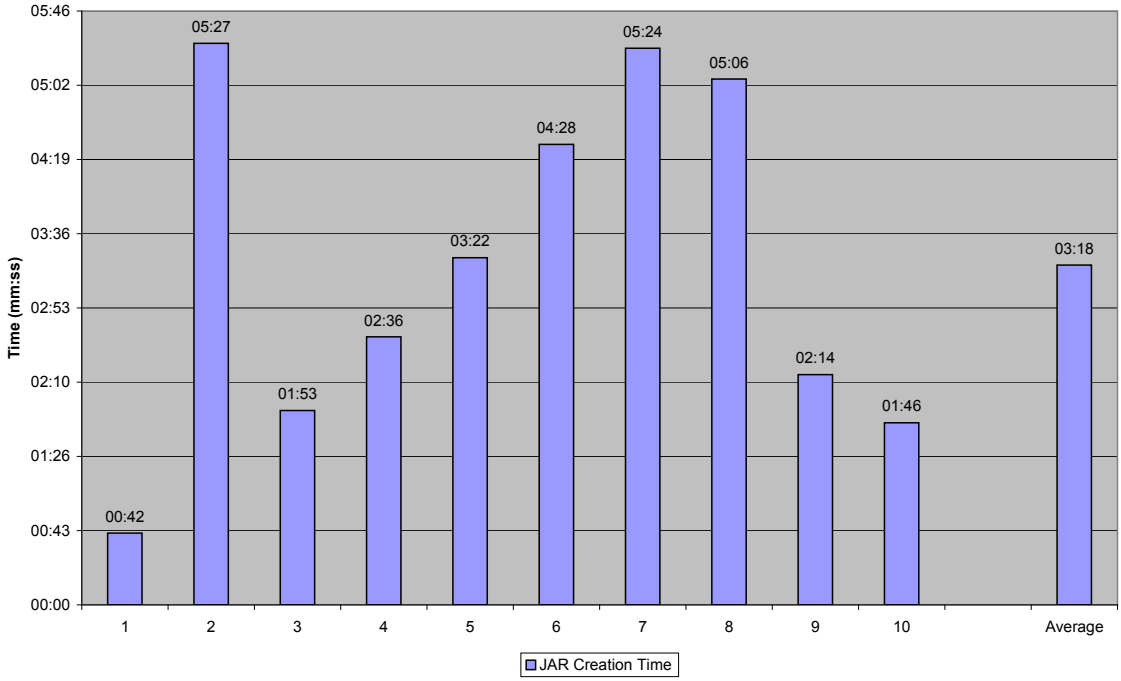


Figure 6.5: Time taken to create a JAR file containing JavaBeans

the overhead worth it.

In addition to JavaBean creation time, the time it takes to import the JavaBean into ARFs repository is also important. Importation is a one off task. All JavaBeans contained in the JAR files registered with the repository are scanned for new JavaBean entries when ARF starts. Figure 6.6 shows how long it took the participants to complete exercise **3b** which required them to import their new JAR file containing their JavaBean into ARF's Repository ready for use in ARF. *Entry 12* on the Graph represents the average time.

Once a JavaBean has been added to ARF's JAR Repository it can be used again and again by the user. The amount of time it took for the programmer to create the component and add to the JAR Repository is the total overhead time for using ARF. The time for the user to connect and configure their newly imported JavaBean using the ARF's PropertySheet, in exercise **3c**, is shown in figure 6.7 with *Entry 12* on the graph representing the average time.

The maximum overhead for using ARF can thus be calculated by adding the

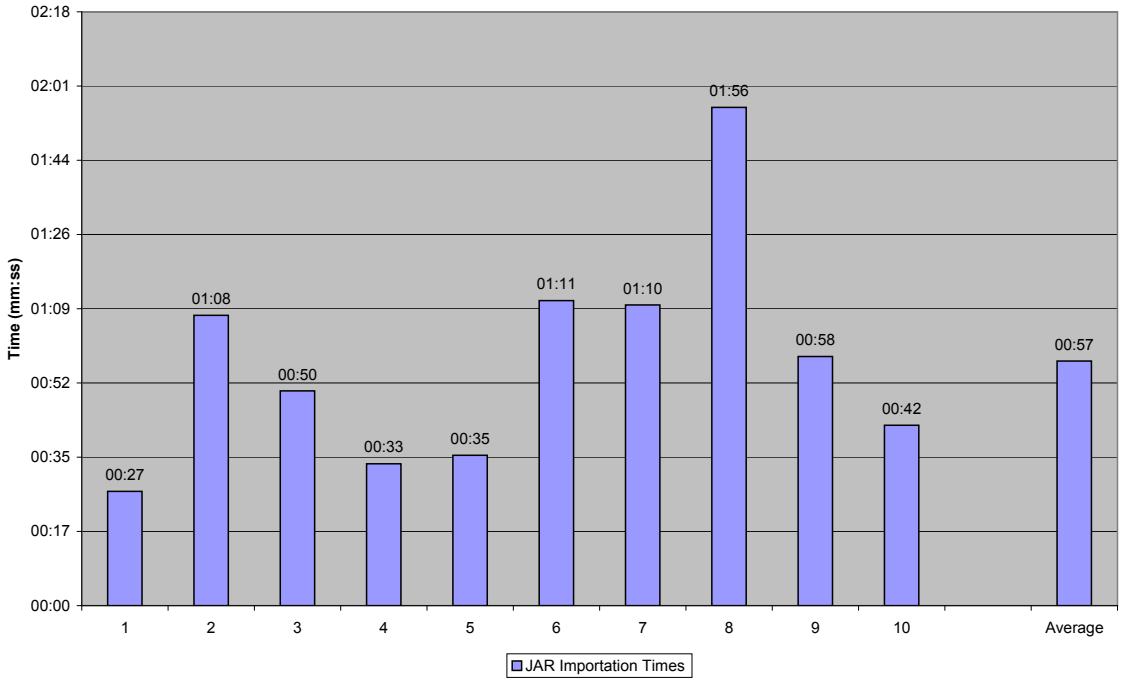


Figure 6.6: Time taken to add a JAR file to ARF's JAR Repository

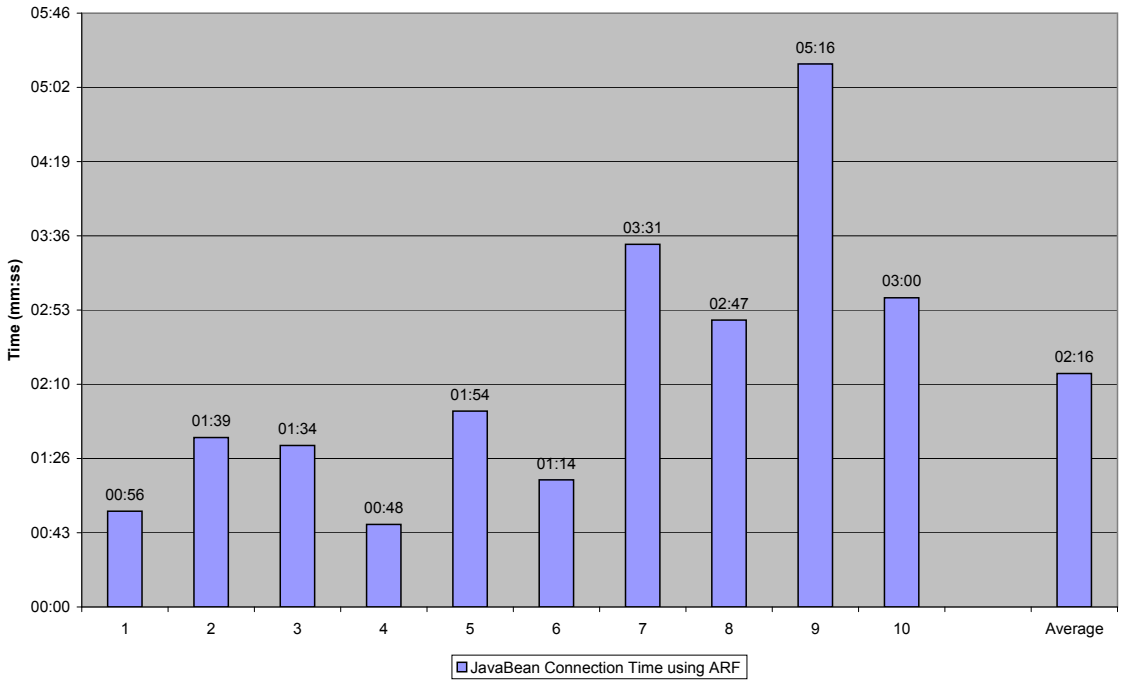


Figure 6.7: Time taken to configure the new JavaBean using ARF's Guided Construction

average times of exercises **3a,3b** and **3c**:

$$ARFoverheadtime = 197.77 + 56.92 = 255secs = 4mins15sec$$

However, since some of these tasks need doing only once this is definately a worst case estimate. Through repeated use of ARF this overhead becomes increasingly insignificant due to the time saved using ARF over standard programming methods.

6.4.2.3 SuperComponent Tests (feature 9 and 5)

Scenario construction time is small, and therefore easily repeatable by a user over and over again in different scenarios. However, ARF provides SuperComponents which increase the amount of code reuse by allowing the user to create their own useful component sets which can then be exported and imported many times into the same or different ARF projects. This allows for straightforward cloning of useful configurations that can then be instrumental when creating test scenarios for multiple robots, or other platforms, as the programmer only has to do the work of building the SuperComponents once. Exercises **1c(ii)** and **1e** measure the time to create a SuperComponent and the time taken to import that SuperSomponent back into the same project to create a duplicate simulated robot. The results are shown in figure 6.8 with *Entry 12* on the graph representing the average time.

SuperComponents reduce scenario construction times by providing a build once, use many times, facility. The export times are somewhat quicker than import times because on import the user has to decide where to add the Java3D sub-graphs and also sometimes has to rename certain JavaBeans to avoid having duplicate names.

6.4.3 Virtual Environment Creation using Java3DBeans

ARF provides guided construction mechanisms for JavaBeans in general. However, ARF also provides the user with many useful Java3DBeans (see Appendix C) for creating virtual worlds which can be controlled and interacted with. To illustrate that virtual world construction times are low, exercise **1a** required the test participants to build a simple 3D scenario with a moveable object. To do this by writing Java3D code by hand can take along time. This is due to the fact that the programmer has to

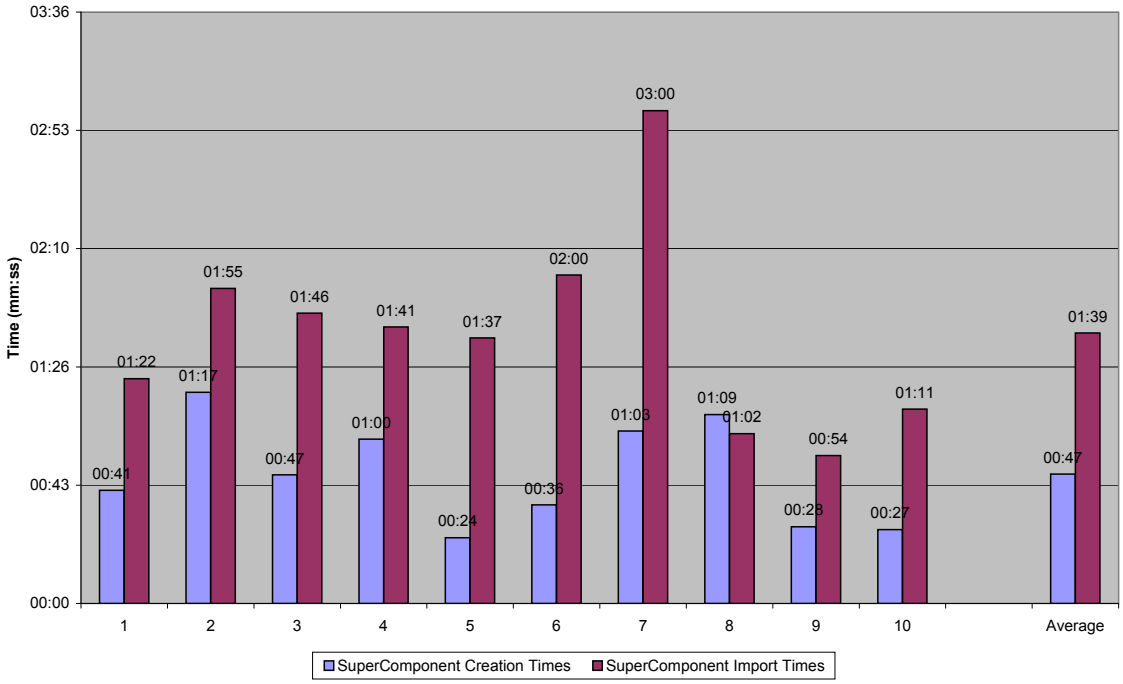


Figure 6.8: SuperComponent Creation and Import Times

know Java3D and then has to setup the Java3D universe manually, which ARF does behind the scenes, which is a large overhead. Figure 6.9 shows the resulting virtual world creation times. The times themselves are quite small and are good considering the test participants generally had no experience in Java3D or ARF. However, they were able to learn how to use ARF easily from the video demonstration. *Entry 12* is the average time.

The average creation time is *251 seconds* for the virtual environment itself. If this time is combined with the average time for exercise 1b, which is time taken to configure the JavaBeans using guided construction, then an average time for a complete scenario creation using ARF is obtained

$$AverageCreationTime = 716.88 + 250.63 = 968secs = 16mins : 8sec$$

This time is incredibly low for creating a virtual environment. Plus, this configuration can be reused through the saving of different scenario configurations or through the use SuperComponents for useful subsets of configured components. Remembering that these figures are based on participants who are novices in using ARF, a best case

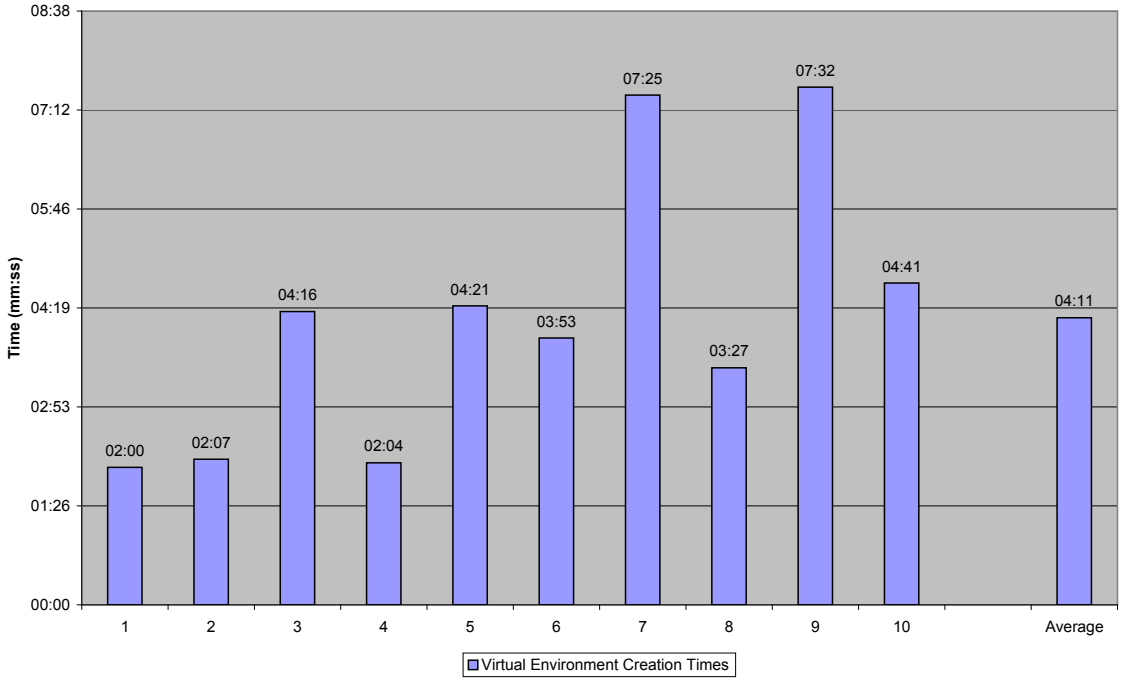


Figure 6.9: Virtual Environment Creation Times using ARF

could be:

$$BestCreationTime = 251 + 120 = 371secs = 6mins : 11sec$$

This value is far lower than the best time for merely joining the JavaBeans together using Netbeans which was undertaken by experts in Java and Netbeans and still took at best *12mins:31secs*. For further examples of full scenario creation times using ARF see the use cases chapter 7, since many of the examples give scenario creation times for real applications.

6.4.4 Participant Skills

It is easy to see from the test results that the test participants had a fairly even spread of abilities, with varying times and assistance requirements. The stacked bar chart in Figure 6.10 shows the skills possessed by the test participants in various programming languages. From this pie chart it is clear that the experience of the test participants is strongest in Netbeans and Java, which is mainly because this was a prerequisite of the test because otherwise it wouldn't have been a fair comparison to ARF. However,

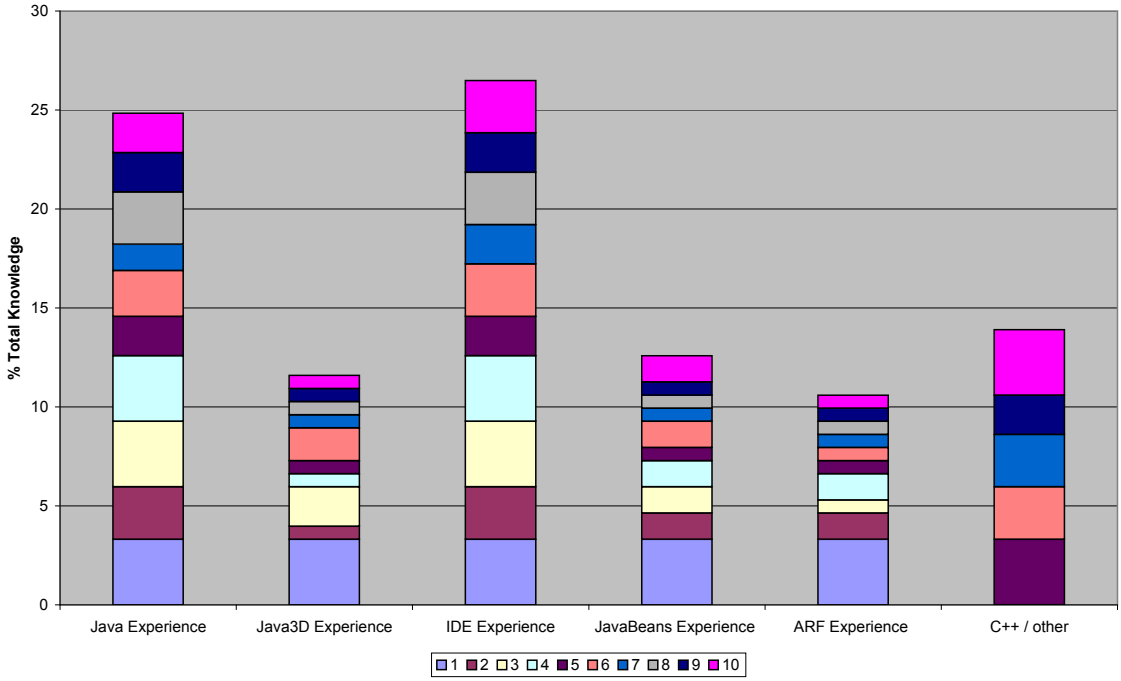


Figure 6.10: Test Participant Skills

despite the lack of knowledge of the test participants in ARF and Java3D, ARF still prevailed as the platform of choice and highest performance.

6.4.5 Assistance Requirements

The amount of assistance required using ARF was much less than amount required using NetBeans. This is because NetBeans provides minimal help. Sometimes the test participant did not know which Java methods to use and as such made mistakes. These would normally have taken them along time to debug and rectify, so instead they asked for assistance instead. Figure 6.11 shows the average amount of times per exercise that assistance was required, or documentation was consulted, for exercises using ARF and Netbeans where relevant. It shows that less assistance was required for exercises using ARF than for NetBeans, and also that the documentation was not needed as much with ARF. Assistance is inherently provided by the ARF user interface guided construction for the same reasons. Therefore it is expected that the user would find ARF easier to use and require less help. This is also corroborated by the complexity measures discussed earlier.

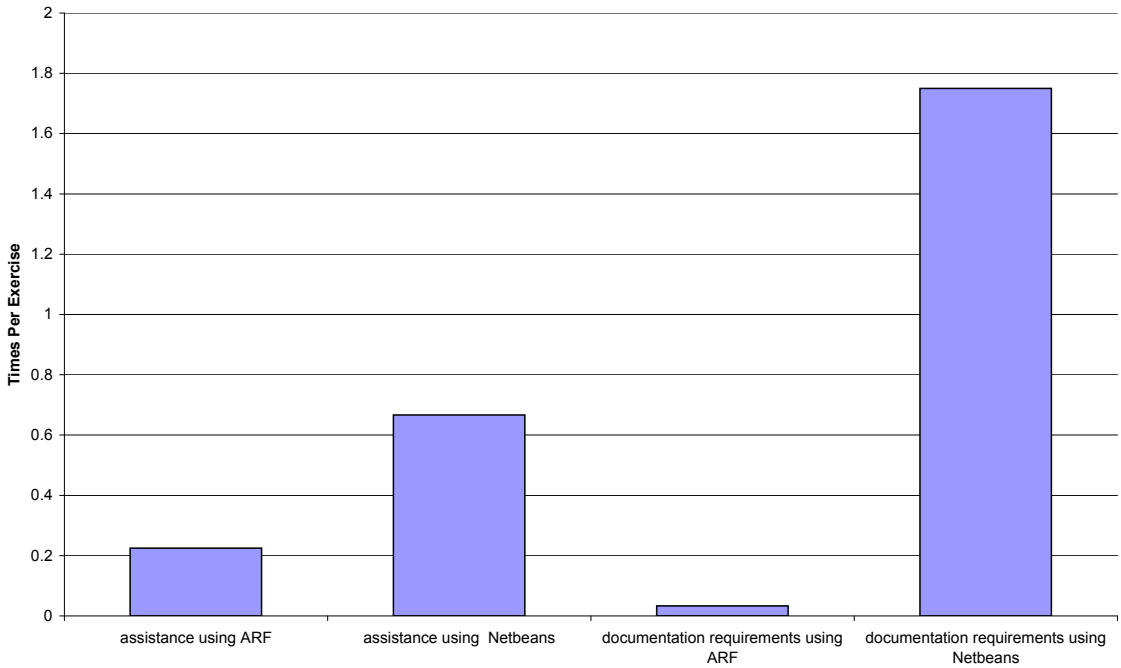


Figure 6.11: Assistance and Documentation Requirements

The reasons given by test participants for requiring assistance with either ARF or Netbeans are listed in tables 6.2 and 6.3 respectively. In general most problems with ARF were due to not consulting the tutorial for assistance first, and sometimes due to a misunderstanding on how to do something in ARF. With NetBeans the assistance was generally required due to programming errors, not being able to use documentation, or not consulting the documentation in the first place. The Netbeans exercise was harder than the ARF equivalent because it relied heavily on the programming skills of the individual, where as ARF provides a much more intuitive interface to the same task and thus makes the task easier, and can thus be carried out by less skilled individuals in programming.

6.5 Summary

The testing carried out by the ARF Performance Tests (APT) was designed to highlight the features of ARF which are very hard to prove via the use case examples alone (see use cases in chapter 7). The most important points to draw from the results discussion is that ARF provides an extendible architecture which is generic enough to

ARF Assistance Requirements
Bugs in ARF
Didn't understand Java3D coordinate system
Didn't fully understand hierarchical nature of Java3D TransformGroups
Misunderstood how to use the BeanBoard and SceneGraph
Didn't know how to use event listeners
Didn't understand which Field of view to alter in the simulated sonar
Didn't understand the Java3D scenegraph
Didn't export all required JavaBeans as a SuperComponent
Didn't understand that when importing SuperComponents that subtrees had to be added to specific locations on the scenegraph
Didn't read question properly or consult documentation

Table 6.2: Reasons user's required assistance when using ARF

NetBeans Assistance Requirements
Didn't see member variables of class they were editing
Couldn't find correct methods to configure sonar
Problems with syntax errors
More example classes in documentation would have helped
Forgetting to instantiate Java variables resulting in Null pointers
Found hard to locate correct methods
Didn't understand how to execute the exercise
Didn't know how to use certain features in NetBeans
Didn't consult provided documentation on creating JavaBeans

Table 6.3: Reasons user's required assistance when using Netbeans

provide the capabilities of all the different usages discussed in the literature review. In essence ARF is a one size fits all architecture which provides different generic granular levels thus providing flexibility for programmers and high end users. The second most important feature that the APTs highlighted is the considerable performance improvement over conventional methods of creating test scenarios. Furthermore, the trade off of using ARF is minimal due to being built on existing JavaBeans technology, which means that the programmer has less work to do to build components which interface with ARF. ARF scenarios can be altered quickly due to the use of project configurations, SuperComponents and JavaBean PropertyEditors.

The analysis of ARF's performance discussed in this chapter provides the fundamental evidence for why ARF is so powerful for higher level applications, such as creating HIL testing scenarios. The main increase being due to the many small time savings which ARF provides, through the use of guided construction, SuperComponents and PropertySheets, which propagate to massive time savings in more complex projects. Suddenly, scenarios no longer need to be reprogrammed from scratch, they can simply be extended, have new components bolted on and merged with functionality from completely different projects through the use of SuperComponents. As ARF's Java3DBean and JavaBean component base grows, the more quickly scenarios can be created with ever more useful SuperComponents. The run away effect of having a large ever increasing component base means that before long design of new components will hardly ever be necessary, making the small overhead of creating the components in the first place negligible.

In order to see how ARF has been used for many different real applications, see the use cases in chapter 7.

Chapter 7

Use Test Cases

It is very hard to measure the effectiveness of ARF using peformence testing alone, since these do not reflect how ARF is likely to be used, or how it meets the requirements of chapter 3. Many potential example usages of ARF were highlighted in the literature review; these examples are what ARF would be likely to be used for. Thus, the best way to test ARF is to see whether ARF accomodates these examples and therefore test as many different scenarios and usages and see if their goals are achievable. As it happens, ARF was able to accomodate these usages easily which therefore demonstrates the inherent flexibility and extendibility of ARF's architecture. The question therefore remains as to how much time was saved using ARF over conventional methods. This questions is answered in general by the performance testing carried out in chapter 6, but is corroborated by scenario creation times demonstrated in the following use cases.

In the following sections many different scenarios have been implemented using ARF. In most cases new components were required to provide the required capabilities. This is to be expected since the idea of ARF is that the component set grows and becomes more feature rich with time. What matters is that component creation time is kept to a minimum and scenario creation time is kept to a minimum. ARF's guided construction mechanisms help to reduce scenario construction time using the techniques discussed in chapter 4.

In each of the use cases an outline of the task is given, and how ARF is used to accomplish the task is described. In most cases creation times for components and

scenario construction are given to illustrate how quickly creating components for ARF and using ARF can be. However, for more statistics on component creation times and, importation times and scenario creation times, see the testing in chapter 6. Most of these uses were not planned specifically, but rather the requirements arose during the development of ARF. Therefore these uses were not set out like test cases for ARF, but rather have become illustrations of how ARF was able to be used on real projects. However, in each case areas of ARF have been identified as being very useful, and without ARF's flexible and extendible architecture none of these applications could have been easily catered for. Therefore these use cases are a tribute to ARF's success since they could not have been realised without it.

The use cases discussed can be categorised as to the features they demonstrate from the requirements; a table of the working modes/features for the different use cases is shown in figure 7.1. More thorough description of the working modes is given in section 2.3.2.

NAME	Features of ARF Demonstrated										
	OFF	ON	HIL	MHIL	SIL	HS	OM	TR	AR	AV	VR
(6.1) RAUVER	✓	✓					✓				✓
(6.2) Nessie II	✓	✓			✓						✓
(6.3) Nessie III	✓	✓	✓		✓		✓	✓		✓	✓
(6.4) DELPH´IS	✓	✓	✓	✓	✓		✓			✓	✓
(6.5) RAUBoat I							✓				✓
(6.6) AUTOTRACKER		✓			✓				✓		✓
(6.7) BAUUV	✓	✓			✓	✓	✓		✓	✓	✓
(6.8) SeeTrack Offshore	✓	✓			✓			✓			✓
(6.9) PAIV	✓	✓			✓		✓				✓

OFF(LINE) – Non-real-time pure simulation
ON(LINE) – Real-time pure simulation
HIL – Hardware-in-the-loop
MHIL – Multiple hardware-in-the-loop
SIL – Software-in-the-loop
HS – Hybrid simulation
OM – Online monitoring
TR – Training
AR – Augmented reality
AV – Augmented virtuality
VR – Virtual Reality

Figure 7.1: Applications of ARF demonstrated by the use cases

7.1 RAUVER AUV Simulator

RAUVER [55] was first AUV built in the OSL (See background section 2.8). Most AUVs built since have conformed to RAUVER's communication protocol, OceanSHELL (see section 4.1.5), and the message types and definitions used. Therefore, ARF implemented OceanSHELL I/O for most of the message types to/from corresponding simulated components in ARF for RAUVER. This enabled simulation testing to be executed using a hydrodynamic model of RAUVER through the same communication interface as RAUVER itself, meaning the RAUVER model could be substituted for the real RAUVER transparently to other software modules. Therefore the programmer only has to interface with RAUVER's OceanSHELL messages.

Since RAUVER a new more generic set of OceanSHELL message definitions have been adopted called ALI (see section 4.1.5.1 in literature review). Therefore, ARF implements interfaces for communicating using both the old RAUVER messages and the new ALI messages. The idea being that the new ALI messages are more flexible and make it easier to communicate with any ALI enabled robot, as before special messages were needed where the RAUVER messages weren't feature rich enough.

ARF implements many components for inputting and outputting OceanSHELL messages. These components enable many different HIL, HS and PS testing scenarios to be realised, as well as providing mission observation through visual feedback. The simulated components in ARF communicate using Java interfaces and events. However, the OceanSHELL I/O implements these same Java interfaces, so OceanSHELL can be potentially replaced by some other communication medium or by straight event passing between components in ARF. The RAUVER components support direct event passing in software using Java events as well as the ability to input/output OceanSHELL or any 3rd party communication protocol. Generally, information in ARF is passed between components by the chaining of event listeners from one component to another (see section 5.3.7). This provides the extendibility and flexibility required by different testing techniques since multiple data types can be input, output and converted from one type to another. This allows for connections to be easily made between different modules ready for testing.

7.1.1 RAUVER components

Figure 7.2 shows how some of the main RAUVER Messages are input and output from ARF, and the relationships between ARF components. The diagram simulates the navigation systems of RAUVER using the hydrodynamic model and has an autopilot which receives waypoint requests via OceanSHELL. Navigation data is output using OceanSHELL for the mission planner which monitors where the AUV is and issues more waypoints. This is an ideal setup to test an AUV mission planner. This scenario can be combined with object detectors and many other sensors to mimic a real AUV and therefore be used for doing much more complicated tasks such as obstacle avoidance (see sections 7.3 and 7.7). The configuration of components in figure 7.2 is fairly standard of a simple simulated AUV and is the base scenario for a lot of different uses of virtual environments for AUVs. Figure 7.3 displays the virtual environment representation of a RAUVER mesh with sea, bathymetry and a simulated sonar. This scenario can then be easily extended to have other 3D objects and sensors for different uses.

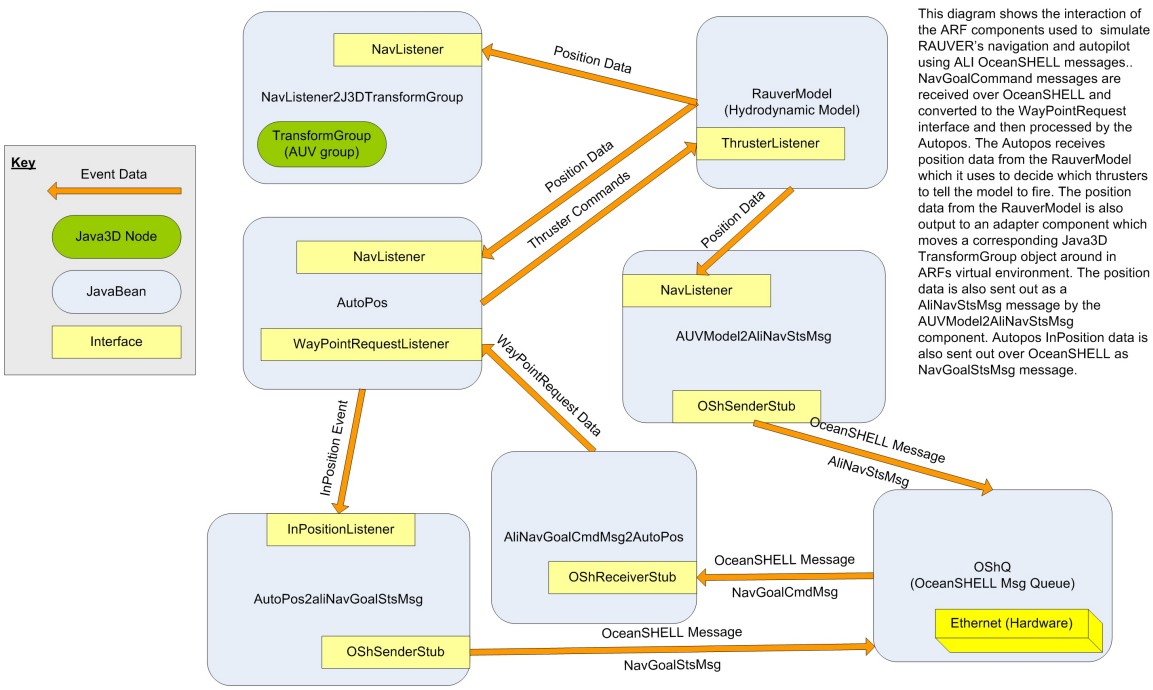


Figure 7.2: RAUVER Simulated Components Interconnection Diagram

The components required for interacting with and simulating RAUVER's low level

systems are as follows:

- `WayPointRqstMsgPlotter` (J3DBean) - displays the position of the waypoint in the virtual environment.
- `WayPointRqstMsgListener2AutoPos` - this listens for OceanSHELL *WaypointRequestMsg* types and converts them to a waypoint request event for the `Autopos` or any other class which implements the *WaypointRequestMsgListener* interface.
- `RvrThrustMsgListener2AUVModel` - this listens for OceanSHELL *ThrusterRequestMsg* types and converts them to a thruster event for the `RauverModel` or any other class which implements the *ThursterListener* interface.
- `NavListener2J3DTransformGroup` - this listens for a *NavListener* event from the `RauverModel` and moves a specified Java3D TransformGroup node around the virtual environment. Generally this is the node of the AUV.
- `AxisForceMsgListener2AUVModel` - axis force is another representation for a thruster command so this component does the same thing as the `RvrThrustMsgListener2AUVModel` component.
- `AutoPos2RvrAutoposInPositionMsg` - when the `AutoPos` has reached the desired waypoint it sends out an *InPosition* event. This event is converted to an OceanSHELL *InPositionMsg* by this component.
- `AUVModelNAV2RauverModelNAVMsg` - this listens for a *NavListener* event from the `RauverModel` and outputs a *RauverNavFeedbackV2* message over OceanSHELL.
- `RauverModel` - this listens for thruster requests and calculates the position of the AUV based on the hydrodynamic and thruster models internally.
- `RauverAutoPos` - this listens for waypoints such as position and attitude requests. This outputs thruster commands to the associated thruster listeners. This listens to navigation data which can be output by the `RauverModel` or could be be navigation data over OceanSHELL.

- `AUVModelDynamics` - this contains the maths for the hydrodynamic properties of the `RauverModel`. This can be replaced by a different set of hydrodynamics for a different vehicle if necessary.

- `ScreenPicker` (J3DBean) - this senses mouse clicks in the virtual world and translates them to positions in the 3D environment. This can be coupled with `WayPointOutput` to provide a point and click interface to output waypoints. These could then be sent directly to the `Autopos` to move the robot.

- `WayPointOutput` - this allows the user to send out waypoint requests. This can also be linked to the `ScreenPicker` component to listen for mouse clicks by the user and convert them to waypoints for the AUV.

- `SimulationClock` - this is a clock which links to any time dependent modules. This clock can be speeded up and slowed down to increase and decrease time.

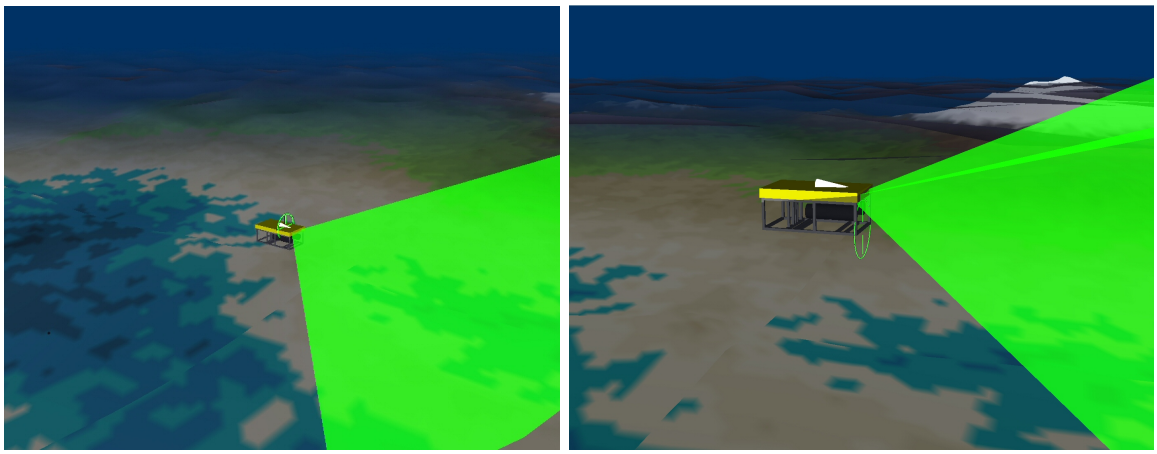


Figure 7.3: RAUVER Virtual Scenario

7.1.2 ALI components

The ALI components only need be used when the real AUV software uses ALI. However, this is just a different communication protocol over OceanSHELL. The extra components that need creating are the inputs to, and outputs from, the simulated modules of ARF. Components such as `HydrodynamicModel/Autopilot` can remain

the same as the RAUVER components. Figure 7.4 shows how the ALI communication components can be substituted into the RAUVER simulation diagram of figure 7.2. ARF provides the ability to have many listeners to any one event. Therefore a simulation scenario in ARF could input and output both RAUVER and ALI messages concurrently, or a mixture of the two.

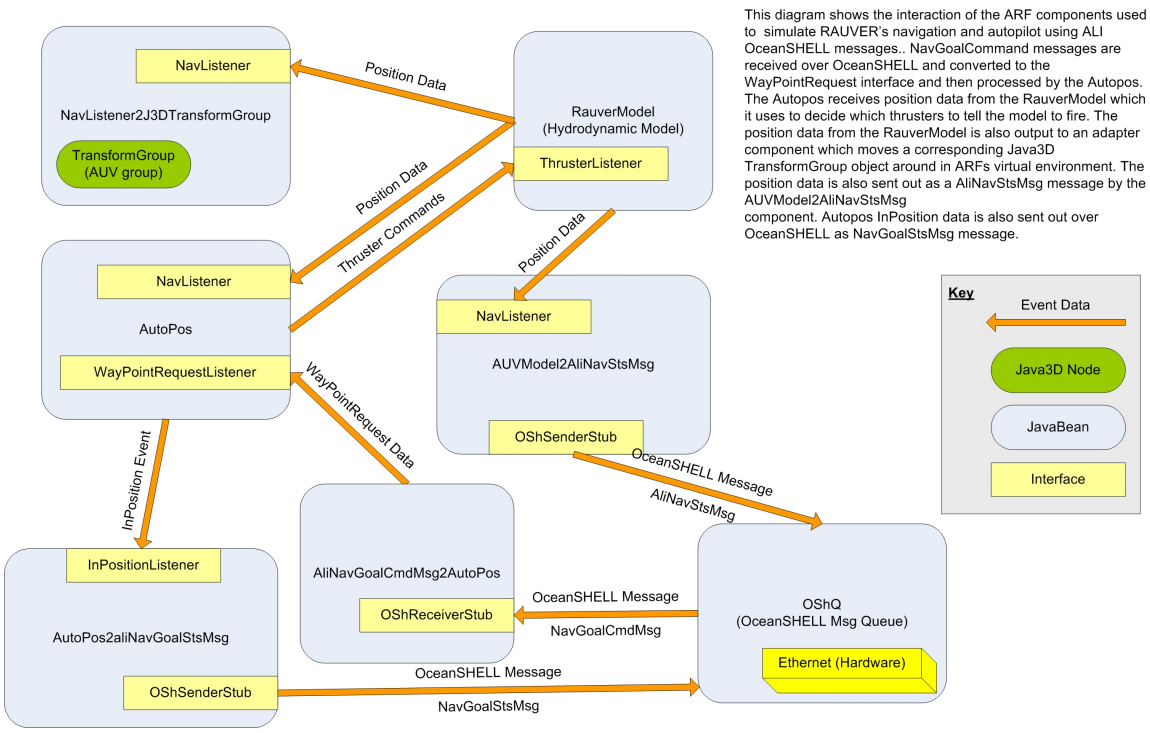


Figure 7.4: ALI Simulated Components Interconnection Diagram

ALI I/O components for ARF:

- `AUVModel2AliNavStsMsg` - similar to `AUVModelNAV2RauverModelNAVMsg` except this outputs an ALI message called *AliNavStsMsg*.
- `AliAxisForceMsg2AUVModel` - ALI axis force is another representation for a thruster command so this component does the same thing as the `RvrThrustMsgListener2AUVModel` component.
- `AliNavGoalCmdMsg2AutoPos` - a *NavGoalCommand* message is basically a waypoint request message. This component converts this to a *WaypointPointRequest* event and passes it to any registered `WaypointRequestMsgListener` objects, e.g. `AutoPos`.

- `AutoPos2aliNavGoalStsMsg` - When the AutoPos has reached the desired waypoint it sends out an *InPosition* event. This event is converted to an OceanSHELL *aliNavGoalStsMsg* message by this component, similarly to the `AutoPos2RvrAutoposInPositionMsg`.

7.1.3 Summary

These components provide the core modules required to simulate AUVs by providing basic simulated movement and control. Inputs and outputs from these components can communicate internally in ARF using events, or can be linked to external communication using the OceanSHELL I/O classes described. The ability to communicate with modules running on other platforms provides the most basic HIL/SIL functions. Real control modules running on the AUV's systems can be tested using simulated outputs from the virtual environment. For more complex testing, simulated sensors are required for detecting objects and mimicking real hardware on the AUV. These components only provide thruster and autopilot control. However, the OceanSHELL receiver modules can also be used to receive data from a real UUV and used to update ARF's virtual environment. For example, navigation and waypoint data from a real platform can be displayed in ARF, in real time. This has been used for *online monitoring* (OM) of the real vehicle running real missions where communications via wifi or tether are available (see SeeTrack Offshore in section 7.8 as an example of an intelligent ROV with full communications to the surface). This set of components enables for pure simulation of a basic AUV and enables higher level autonomous systems to be tested. The ability to do faster than real-time (OFFLINE) simulation is provided by the use of the `SimulationClock` JavaBean.

7.2 Nessie II AUV Developmental Testing

Nessie II (see figure 7.5) was the 2nd generation Nessie AUV designed for the SAUC-E (Student Autonomous Underwater Vehicle Challenge Europe) competition (see literature review section 2.8 for more background). Real world testing of Nessie was primarily focussed on low level systems as there was little real world testing time left.

This meant that testing of higher level systems, such as the mission planner, did not happen. Therefore, a complete simulation of Nessie II was required which provided the same inputs to the mission planner and which worked in the same way as the AUV.

Using ARF, a scenario was developed which used the core simulated components from RAUVER and many new ones, such as simulated sensors. Nessie 2 was largely based on the same OceanSHELL messages as RAUVER, however some were slightly different so copies of the RAUVER components were altered slightly.

There was only a matter of weeks before the start of the competition and the mission planner still hadn't been tested on anything resembling the real system. Only tests by the programmer had be done and the programmer was certain that there was nothing wrong with the module and it would work perfectly. The problem with designing static tests to test a section of code is that they are just that, "static". The problem is that the programmer can only test the module on what they expect the module will have as inputs and therefore cannot test outside the scope of their knowledge. Quite often modules are programmed to interact with other modules based on a specification of an interface; this causes problems when the definition of that interface is open to interpretation. The problem with Nessie II was that various interfaces had been designed before construction had commenced, and these had since evolved. However, due to poor communication during development, the different modules hadn't been tested working together and if they had they would not have worked. Therefore, the changing of interface specifications, and those very same interface specifications being open to interpretation, led to a mission planner that did not work at all.

7.2.1 Planner Testing Components

The task then was simple; gather interface information from all programmers modules and redesign the interfaces so that everyone was using the same ones. Once this was completed, the design of a virtual reality testing scenario in ARF was able to start. There were many specialised behaviours that Nessie 2 had which needed to be mimicked in simulation so that the mission planner was executing realistically. Below

Figure 7.5: Nessie II

are the ARF components which were required for Nessie's virtual environment:

- `NessieThrusterCmdMsgListener2RauverModelAdapter` - the Nessie 2 autopilot would eventually require testing. However, the thruster commands sent by the autopilot were different from that sent by the RAUVER AutoPos. This class converts from Nessie 2 thruster messages to thruster command events which the RAUVER hydrodynamic model can understand. The RAUVER hydrodynamic model was used because it had the same thruster configuration as Nessie 2 and it was already working and there was not enough time to create a new one specifically for Nessie 2. Also the exact motion of Nessie did not need to be replicated for the purposes of testing the mission planner. This component is very similar to the `RvrThrustMsgListener2AUVModel` component.

- `NessieWayPtRqstListener2AutoPos` - listens for Nessie 2 waypoint request messages and converts them to *WaypointRequest* events which can then be sent to the `AutoPos`. Does same thing as the `WayPointRqstMsgListener2AutoPos` component.

- `FOVPickSensor` (J3DBean) - this is an object detector which can be set up

to have the same characteristics as a video camera or sonar. The idea is that instead of simulating video or sonar precisely and then analysing those images for objects, simply run collision testing within a certain view volume and return a list of all the intersections with different shapes. Thus doing the sampling and detection phase all in one go and simplifying the simulation. This simulated sensor has variable field of view (FOV) and can run as often as the user desires, e.g. for a forward looking sonar, this could have an update rate of about 1Hz, for video detectors an update rate of 25Hz could be used. This sensor outputs a list of intersections and reports the name, pseudo type and Java type of the object. This event is sent to all registered listeners. `ForwardCameraMsgOutput` and `DownwardCameraMsgOutput` are potential listeners. These listeners then prune the data and filter out results they are not interested in. This technique speeds up object detection and can be used when the user is not concerned with simulating real world data realistically.

- `ForwardCameraMsgOutput` - this component listens for events from an associated `FOVPickSensor` which is configured to the field of views and update rates of the object detection systems running on Nessie 2. The detected objects are then queried as to their `ARFInfo` user type (discussed in the ARF interfaces section 5.3.6) and then the position of the objects are sent out in camera coordinate frame in an OceanSHELL message called *ForwardCameraMsg*.

- `DownwardCameraMsgOutput` - this does the same thing as `ForwardCameraMsgOutput` except that the downward camera was only capable of detecting certain objects so the results were filtered prior to sending out detections.

- `SonarMsgOutput` - Nessie 2 used sonar to detect objects which needed further inspection and classification. These objects would be inspected in turn and then categorised using the video camera sensors. The sonar also had another mode which was to send out 2 sonar pings at right angles to one another. One ping sent forward and one ping sent to the right of the AUV. This was to try and detect the tank walls. This information was then fed into the navigation system to try and localise the AUV in tank space. This was combined with

an IMU simulator and a compass, which combined with SLAM, done on object detections, could help correct the navigational drift of the AUV.

- `SonarStartMsgListener` - this was a special component which listened for messages from the mission planner which changed the mode the sonar should be in - full 360 degree search mode, or 2 pings wall detect mode.

- `SonarWallFindPickRays` (J3DBean) - this class was developed to do the 2nd mode of the sonar, to detect the walls. A 2nd sonar was used since this was simpler than reconfiguring the sonar every time the mission planner switched from search mode to wall ping mode; one of the sonars was simply switched off.

- `WorldDataMsgOutput` - the mission planner itself did not listen to the camera messages or the sonar messages. This is because the camera and sonar object detections were all in vehicle coordinate frame with coordinate axis flipped depending on orientation of the camera/sonar. A world model was used which was combined with the navigation systems which transformed the relative positions output by the camera and sonar to absolute coordinates in tank space. The mission planner listens to the output world model message for these object identifications. Upon receipt of the object positions the planner might change its task to something more specific. The world model and navigation systems were not ready at time of testing. Therefore this component relies on the fact that the positions output by the simulated sensors are correct, which they always are in simulation as no noise was added to the results.

This proved enough to test the mission planner in a SIL fashion, and iron out most problems. However, it also helped to iron out all the problems with the interfaces and standardise them as most of the interfaces hadn't been used yet. One of the main problems with Nessie 2 was that often the intelligence of the software was located in the wrong software modules making it hard to have one central control. For example, the forward camera module was supposed to make detections and output them. However, the mission planner programmer had assumed that once the camera was within striking distance of the "orange ball" that the detection module should send out a message saying it was ready to hit the ball. Really it should have been the job

of the mission planner to decide when was a good time to drive forward and hit the ball. However, this meant that the simulated modules required were not as clean as they might have been and in a lot of cases duplicated code that was present in the real detection modules.

Using ARF to test the mission planner forced all the other modules to start being integrated together. However, problems were surfacing until time ran out. Some of these could have been averted if more HIL testing had been carried out rather than just module by module simulated tests, which took longer than necessary due to not having been integrated properly. Testing all the real modules working together on the platform in HIL mode is the most important test since this shows whether an AUV is ready to run autonomously. An AUV should not be able to run an autonomous mission until all integration problems have been dealt with. AUVs cost a lot to produce and can be easily lost if one software module fails. Unknown errors can sometimes never be found because the AUV maybe unrecoverable making debugging impossible. As with Nessie 2, and many other AUVs, adequate testing facilities were not available to test the whole vehicle in HIL fashion. In the case of Nessie 2, there was simply not enough time to carry out lots of HIL testing and produce the extra modules for ARF. ARF was still in its infancy at this point and meant that alot of, now known to be, standard ARF components were simply not available for immediate use.

7.2.2 Nav System Testing Components

In addition to testing the mission planner, the navigation and world model module with integrated SLAM [16] was also ready and required testing. However, this module needed a few more inputs from different sensors which were currently not available in simulation. Therefore the following simulated sensors were created for ARF:

- `Altimeter` (J3DBean) - The altimeter is a key part of any AUV. Very simple to simulate in Java3D using a pick ray to test for intersections below the AUV. This component can be placed anywhere on the SceneGraph of any moveable object in the the 3D universe.

- `IMUSimulator` - Nessie 2 had an Inertial Measurement Unit (IMU) which pro-

vided roll/pitch/yaw rotation rate and attitude in roll/pitch/yaw. It also provided accelerations in roll/pitch/yaw. This sensor was used to help the navigation system. Although the accelerometers of the real sensor were inaccurate the headings given were still useful.

The Nav was able to output position relative to the corner of the tank using an amalgamation of information from:

- Sonar wall pinging - send 2 sonar pulses at right angles to one another to find out how far each edge of the tank was away.
- IMU - This was used as a compass to provide heading relative to a ficticious north which was aligned to the orientation of the tank (tank space).
- SLAM - This was used with detected object positions in both sonar and video camera to help correct the position of the vehicle when known features were re-detected.

Using ARF to test the SLAM navigation proved very useful since many bugs were detected related to timing issues due to old data and processing lag times. However, when it came to real world testing, the output from the sonar was far to noisy such that many false detections were made which meant that the SLAM system couldn't process the large amounts of false data quickly enough. Consequently, SLAM was eventually turned off and would need making far more robust with better filtering of data from the sensors. Use of a low quality sonar and acoustic noise in the test tank compounded the problem. ARF showed visually that slam worked in simulation because as the virtual robot passed the same object again and again the user could see the AUV jump to the new corrected position output by the SLAM. The simulated AUV's position was displayed and also was the SLAM's estimated position. The user was able to then see when the position estimated by the SLAM drifted and when it was accurately corrected as new object detections were made by the simulated sensors in ARF. This simple test shows that the SLAM works, but the abundance of large amounts of noise means that it takes a lot of computational power to find the correct features, thus this was unable to be used on Nessie II.

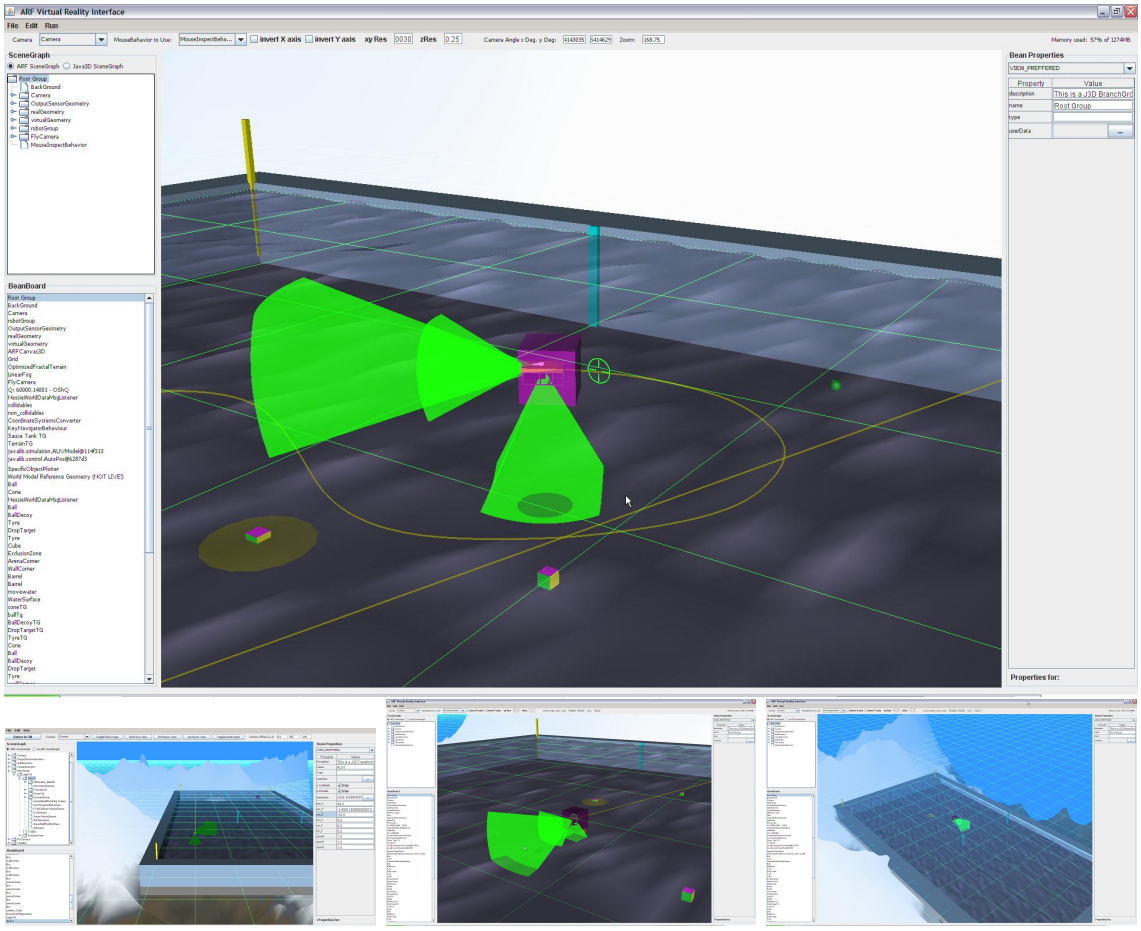


Figure 7.6: Nessie 2 Mission Tester

Figure 7.6 shows the Nessie 2 test environment for the mission planning and the SLAM. The green cones attached to the AUV represent the sonar and camera field of views. The path of the AUV was visualised using a yellow trail. Unclassified objects detected by the Sonar were represented by the multicoloured cubes, these then changed to actual object representations once the simulated cameras had classified them under closer inspection. The purple box around the AUV represents the position output by the SLAM system. This did not always match the true position of the AUV as the navigation was subject to drift. However, the purple box would re-align itself with the AUV when an object redetection caused the SLAM to recalculate the AUVs position.

7.2.3 Summary

This example, and also the Nessie III example, show how ARF components, if kept simple, can be used for many different purposes and that if something does not yet exist, new components can be easily created and integrated with ARF. This example demonstrated ARF being used for OFFLINE, ONLINE, SIL, VR, OM, and some basic HIL tasks. By switching components off/on and adding and removing components, ARF was used to test multiple systems in different ways using a variation of a base scenario. The use of SuperComponents allowed for a basic AUV configuration to be saved and then imported into the different scenarios required. The ability to turn on and off features of the SceneGraph enabled scenarios to be quickly adjusted to test new combinations of different 3D objects with the simulated sensors.

7.3 Nessie III AUV Developmental Testing

Nessie III [57] (shown in figure 7.7) is the new AUV based on the Nessie II hull design and software architecture. Nessie III involved a complete electronics redesign and new special purposed underwater connectors to eradicate the water leak problems caused by low quality connectors used on Nessie II. Nessie III was designed for salt water use, therefore water leaks were simply not an option due to the serious damage this can cause. In addition to hardware upgrades Nessie III required more stable

software modules for navigation, control and mission planning, i.e. most of it. Nessie III has 4 video cameras, an IMU (compass), an 802.11g WIFI antenna, a Doppler Velocity Logger (DVL) and various data/power tether options for use in ROV mode and testing.

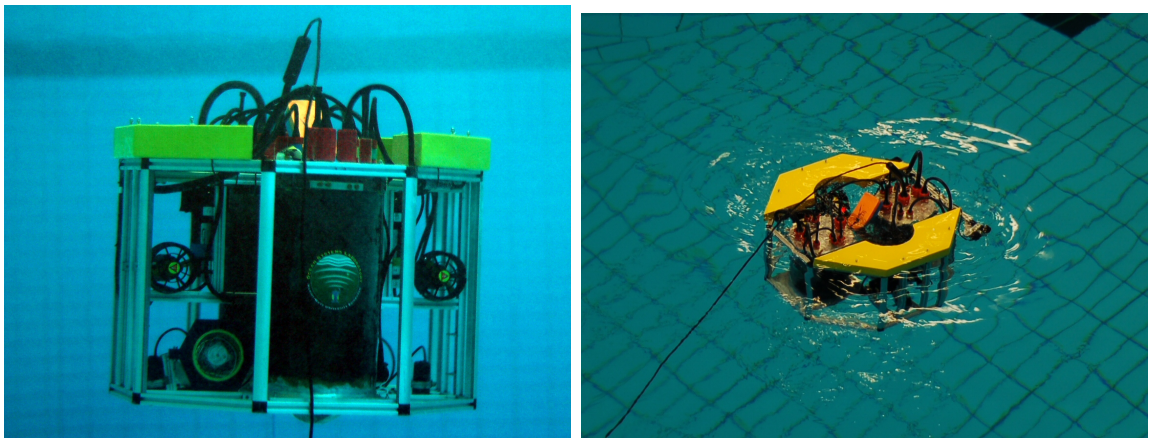


Figure 7.7: Nessie III

Nessie 2 relied heavily on SLAM systems and being able to detect the shape of the tank for navigation purposes. This proved error prone, so a different approach was adopted for Nessie III. The Nessie III platform uses visual tracking of the ground in order to estimate movement and absolute position. In addition, Nessie III is equipped with a DVL which, combined with a compass, provides accurate navigation when visual systems cannot be used. Visual systems rely on the water being clear and well illuminated, but this is rarely the case in real ocean and inland lake environments. However, the DVL is banned in the SAUC-E competition, but due to clear water the visual tracking was all that was required. Figures 7.8 and 7.9 show the hardware architecture and software architecture respectively.

The lower level navigation systems were the first systems requiring testing and implementing on the AUV in order to provide a stable platform for testing the higher level modules, such as control and mission planning. Both mission planning and control require lots of testing to make sure the software modules are stable and do what they are supposed to do. The more testing the better so that the operator has confidence that when the AUV goes on a real autonomous mission it will do what it is supposed to do without any unforeseen and errors. Thus the control and

mission planning modules were tested thoroughly in ARF using HIL and PS testing techniques.

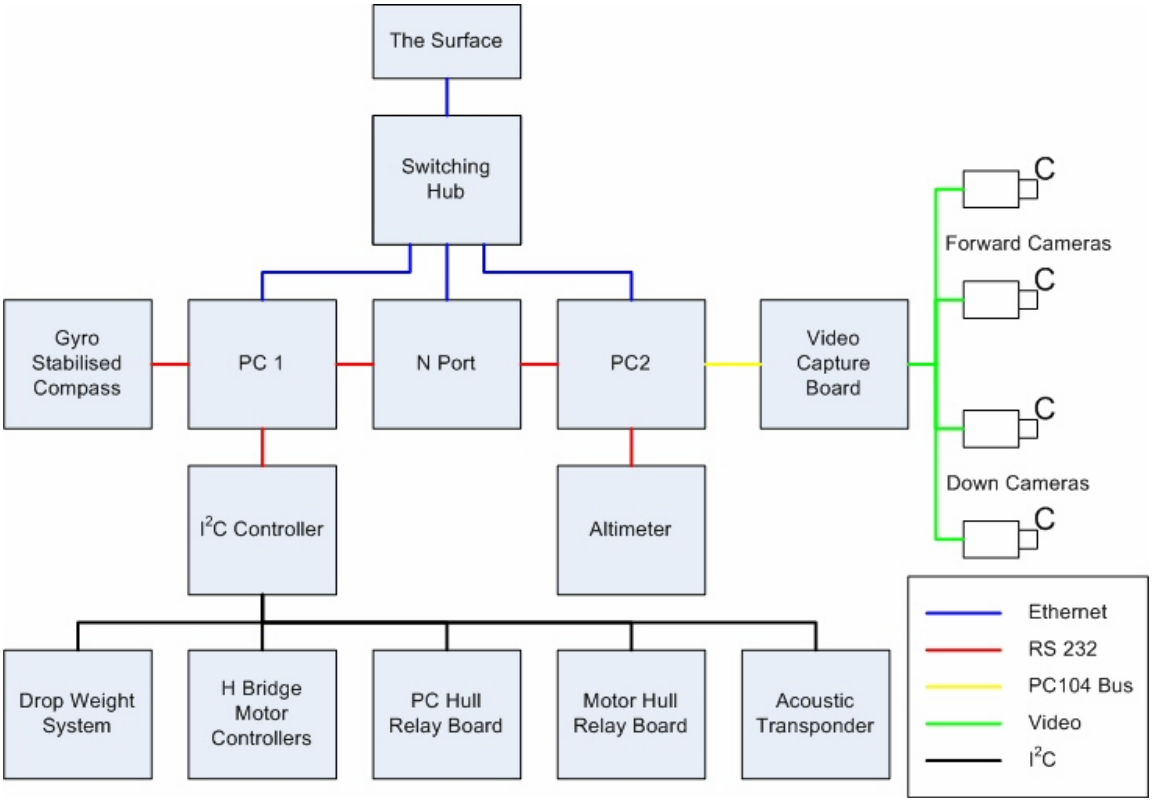


Figure 7.8: Nessie III Hardware Architecture

7.3.1 ProportionalIntegralDerivative (PID) Controller Tuning

A PID controller allows the robot to move to a specific position or rotation using its thrusters. In order to reliably do this the controller needs constant feedback of the AUV's position and velocity information. This is called a closed loop system. This allows for the PID controller to monitor whether it is going in the correct direction from the navigation information. On the Nessie AUV the navigation information is provided by the navigation module which uses multiple sensors, such as DVL and IMU, to give accurate position and velocity information.

A new autopilot was required for Nessie III, however Nessie III was still in construction and couldn't be used to tune and test the PID controller whilst it was

being developed. Using a real vehicle to continually develop and iron out bugs in the PID controller would prove harder than using a simulated vehicle, because accurate position and velocity information required by the PID controller would not be available until much later in the AUV's development. Instead, a virtual reality scenario in ARF was created which used RAUVER's hydrodynamic model JavaBean to produce navigation data and simulate the thrusters of Nessie. The software interfaces for Nessie would be the same OceanSHELL messages which were used for RAUVER, since Nessie uses the same thruster layout and configuration. This alleviated many of the problems of creating new OceanSHELL message definitions which Nessie II suffered. Consequently, anything developed for the simulated AUV is directly transferable to the real AUV.

The simulated AUV provided a test-bed and also a development tool. The programmer was quickly able to learn how the simulated AUV reacted to thruster commands. Once the PID controller was ready its gain variables were tuned to the dynamics of the simulated RAUVER hydrodynamic model. Once an accurate control was achieved using the simulated vehicle the PID module was ready for testing on the real AUV. Having access to a simulated, although hydrodynamically different, AUV meant that development of the autopilot could continue in parallel with Nessie III's construction. The difference in hydrodynamics between RAUVER and Nessie meant that the PID controller would have to be re-tuned for the Nessie AUV. However, all the learning of how the AUV moves and operates was done using the virtual vehicle and thus all obvious bugs removed early. Therefore integration onto the real vehicle was merely a case of a re-tune.

The task of tuning the PID controller for the Nessie AUV involved having a few other tasks finished as well:

- Navigation System - uses DVL/VVL and IMU (for compass heading).
- IMU or compass module integration and testing
- DVL module integration and testing (this gives velocities in vehicle X,Y,Z frame)
- Thruster module integration and testing - this converts thruster OceanSHELL messages to real thruster movement.

- Visual Velocity Log (VVL) - this tracks features in the downwards camera images to calculate distance moved over the ground and thus velocity.

The sensors also required fitting to the vehicle in hardware and checking to see if they all worked together on the vehicle.

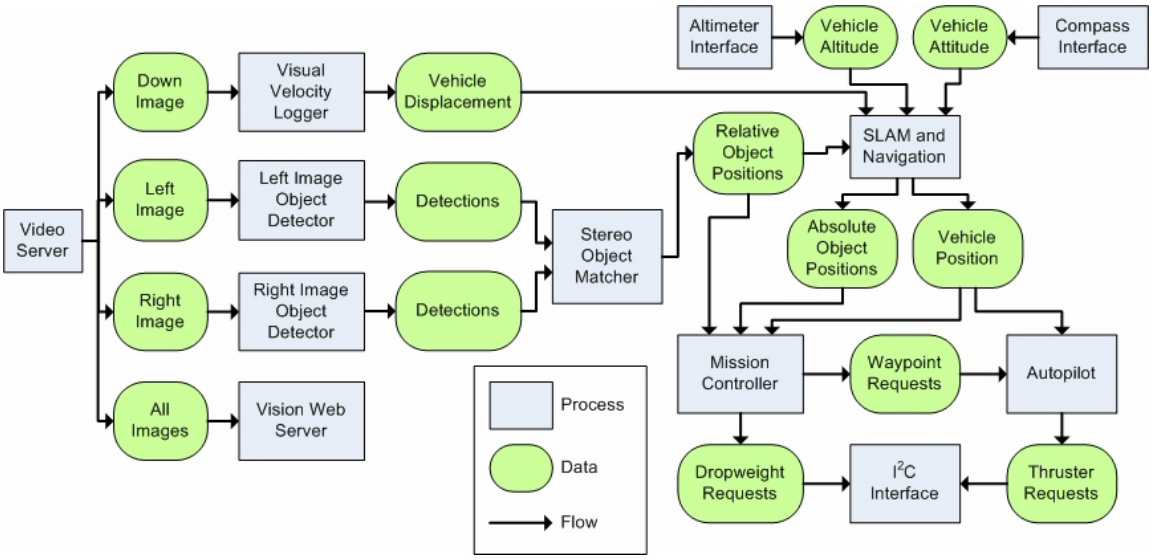


Figure 7.9: Nessie III Software Architecture

Once the PID Controller has been tuned using the navigation systems on board the AUV, the PID controller can be made to control the AUV with no position feedback, i.e. an open-loop system. This can be achieved by tuning a hydrodynamic model of the real AUV by measuring how the AUV's motion is affected by the individual thruster powers. The speeds of the vehicle and rotation rates were able to be measured using the output from the DVL and IMU sensors. Thus a mapping is generated from thruster powers to acceleration of the vehicle. This information can then be used by the PID controller in absense of the any accurate information from the AUV's other navigation sensors. The hydrodynamic model can then be used by ARF for simulation purposes so that other systems can control the thrusters and receive accurate navigation from the hydrodynamic model. This is very useful for testing the autopilot and mission planning systems.

Once the low level systems, such as autopilot, are ready, the vehicle is assumed to be in a functional but basic state. Thus, it can be used for accurately testing some of the higher level systems such as the mission planner. ARF allows for the real systems

to be fully tested before deployment onto the AUV and thus the testing time on the real AUV is more fruitful due to small problems being ironed out beforehand. In a perfect world, the only problems which could arise during integrating to the real AUV are those due to interfacing with real sensors. Thus problems become easier to diagnose at real-world test time since the designers have a better idea of where a problem is likely to be because they know which systems are stable.

7.3.2 Mission Planner Testing using Sensor Simulation

The SAUC-E competition requires that an AUV do the following tasks:

1. Dive a few metres and drive forward through a validation gate (a rectangle floating mid-water) - see figure 7.12 for virtual representation.

2. Find an orange ball and ram it (shown in figure 7.11).

3. Locate a target on the bottom and drop weights as close to its centre as possible.

4. Find either two traffic cones or two tyres on the bottom and surface the vehicle directly between them. Whether the AUV should surface above the traffic cones or tyres is specified just before the AUV enters the water.

The mission planner running on Nessie has to have special behaviours for some of these tasks, these include: searching for the ground targets (tyres, drop-target, cones) and searching for the mid-water targets (orange and green balls). Specific behaviour routines are needed for lining up with the drop target, for dropping the drop weights, and for lining up with the orange ball ready for colliding with it. This is easily done if all object positions are detected accurately in world frame coordinates. However, object positions are subject to inaccuracies due to sensor inaccuracy, false detections and navigational drift. To compensate for errors, once an object has been detected, take for example the orange ball, the mission planner will change its priority to completing the orange ball task. Rather than searching for all objects first and then going back for closer inspection later since positions may have changed. Positions of detected objects are stored by a world model so that once one task is complete, the

AUV goes to roughly where it thinks that the next object is, and starts a new search pattern to find it again if it has not already been redetected by the sensors.

All these behaviours need creating, testing and fine tuning so that they work in the most efficient way. This is time consuming to do on the real platform. Often real-time communications with the platform are needed for real-time debugging so that the operator can monitor what state the planner is in. Testing the behaviours in simulation first means that at least the planner will do the right thing when it receives the real data onboard the AUV, since the same messages will be used in simulation as on the AUV. The tuning of the actual object detection systems has to be done on the real AUV though since this relies on the many factors of the camera systems used to do it. However, much of this can be done off the AUV using a test rig with the same cameras as the AUV. The mission planner was able to be tested using the same ARF components designed for Nessie II. The only difference being that a new event listener was required to output a slightly different OceanSHELL message from the simulated cameras, i.e. FOVPickSensor was used again to do object detection.

7.3.3 Autopilot Visual Servoing for Object Collision Testing

In the previous section it discussed the problems with using absolute positions for detected objects due to sensor inaccuracies and navigational drift. When an object is detected by the AUV its position is stored in a world model. The navigation system uses SLAM [16] to help reduce the drift of navigation based on the detections and redetections of objects. However, these positions are still too inaccurate for close inspections of objects, so the AUV must rely upon continually detecting the object in order to move closer for more accurate tasks. In some cases the AUV may loose all navigation systems which would normally render the AUV useless because the autopilot has no navigation input, other than perhaps a hydrodynamic model which can only provide basic control at best. Therefore, if a point of reference is known, such as the continual detection of an object by a sensor, then this can be used so that the AUV can home in on it, or move relative to it. This requires the autopilot to have an open-loop control mode which can be controlled by a higher level system which takes into account the changing position of the detected object. This technique

is called visual servoing.

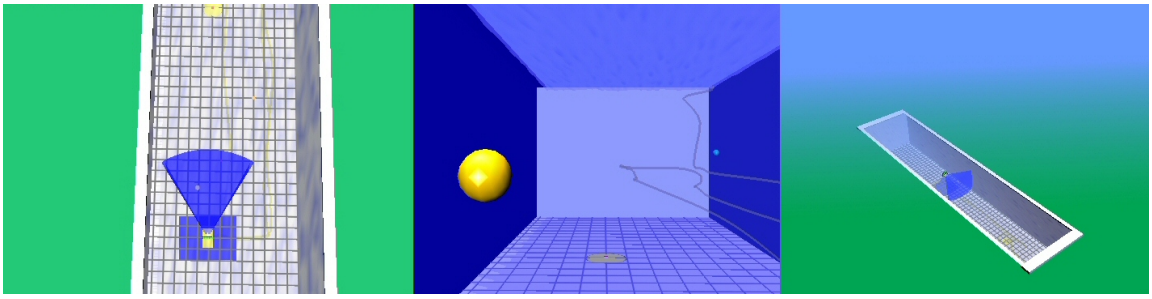


Figure 7.10: Simulated object detectors and hyrdrodynamic model allowing mission planning and autopilot to be tested.

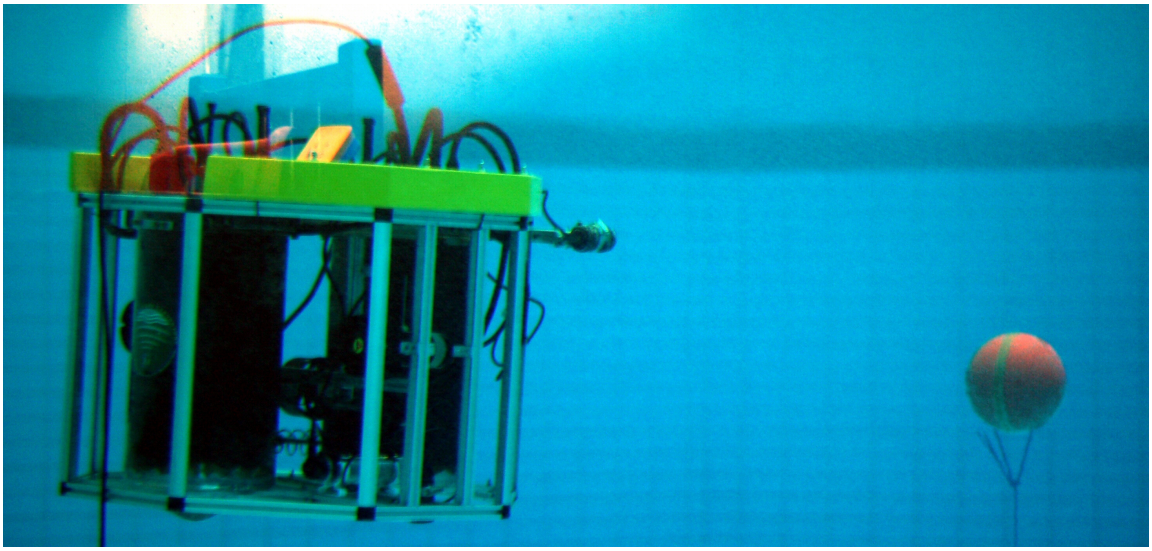


Figure 7.11: Nessie III doing real object detection for testing the mission planner and autopilot.

ARF was used to test that the open-loop control mode of the autopilot worked correctly. ARF simulated the AUV's hydrodynamic properties and also its video camera based object detection system. The outputs from ARF's object detectors were then fed into the mission planner, via OceanSHELL. The mission planner monitored the relative positions of the detected objects and used this to navigate the vehicle to the desired position using open loop control messages sent to the autopilot. In the case of SAUC-E this could be used to home in on the orange ball and drop targets if the navigation system failed. However, this was merely a back up plan and the

actual way the mission planner worked was by using the continual redetection of objects to calculate relative movement commands which were sent to the autopilot. The closed-loop control method of the autopilot proved to be accurate enough and therefore the open-loop method was not used (figure 7.10 shows the ARF virtual scenario which allowed visual servoing and mission planning to be tested). The field of views of the object detectors are shown in blue on the AUV. Figure 7.11 shows the real Nessie III doing the same tests in a test tank. A data tether was used to give continuous feedback of the AUVs systems during testing. ARF was used during these real tests to plot where Nessie thought the real objects were relative to the AUV. This proved to be a relatively good system for checking how accurate the 3D stereoscopic reconstruction was at calculating 3D positions of objects. This is a good example of Augmented Virtuality and mission observation since the virtual world was kept in synchronisation with the real world using a specific JavaBean which received data from the AUV's world model.

The last step of testing is then to execute completely autonomous missions, which should work in the exact same way as they did under tethered testing.

7.3.4 SAUC-E Mission Logging and Replay Suite

The SAUC-E competition required that every team who entered an AUV produce a log file of the AUV's current mission state, object detections, position updates and autopilot waypoints (if they used them). This log file would then be played back after the AUV completed the mission in order to validate that the AUV was doing the right thing and to show the audience and the judges what the AUV was thinking at each point in the mission so that they could see that it was indeed autonomous. The logged positions of detected objects were used to produce a map relative to the start of the mission. Points were then awarded for the accuracy of the map.

ARF was selected to be used as the mission playback environment since it already incorporated many of the required components required to show the dynamic environment of the competition pool. The requirements were to simply playback the log in real-time, plot and update object detections with visual representations of the objects in the tank, and finally display the state of the mission controller, i.e. what

the current intention of the AUV is. In a state based mission controller this is easy because mission state names can be given which correspond to the task to be done in the competition such as: going through validation gate, search for orange ball, colliding with orange ball, searching for drop target, dropping drop weights on drop target, finding cones and tyres which designate surface zone, and finally surfacing in surface zone.

In order for the playback environment to be created some extra components were required to read and playback the log files. The rest of the components had already been created for the sensor simulation and HIL testing environment needed for Nessie III. Therefore the visual representations of the tank, balls, validation gate, drop target, tyres and cones could be plotted in the detected positions logged in the log files. Figure 7.12 shows the various different states of the virtual environment from the Nessie III log file of the SAUC-E competition final. The pictures start from the top left and end at the bottom picture which displays the final view of the tank once the AUV finished. The last known positions of the detected objects represent the final state of the map.

The log files are written using XML, so as well as being human readable they are easily interpreted. An example XML document can be viewed in appendix G.

ARF was used to test the logging system as well. ARF used simulated object detection, vehicle navigation and the real autopilot and mission planner to test that the XML log files were correct and that ARF's interpreter could read and play them back in the same environment. Thus a `SauceLogger` component was created for ARF which allowed for simulated missions to be logged as well. This facility now enables simulated missions in ARF to be logged and therefore used for many different applications other than Nessie III and SAUC-E.

7.3.5 ARF Modules required for Nessie III and SAUC-E

In order to provide the necessary facilities for testing Nessie's systems and implementing the log playback facility, various components were created for ARF to allow real modules to be switched for simulated ones. Other components were also created to show information about current vehicle state which are to be used in mission

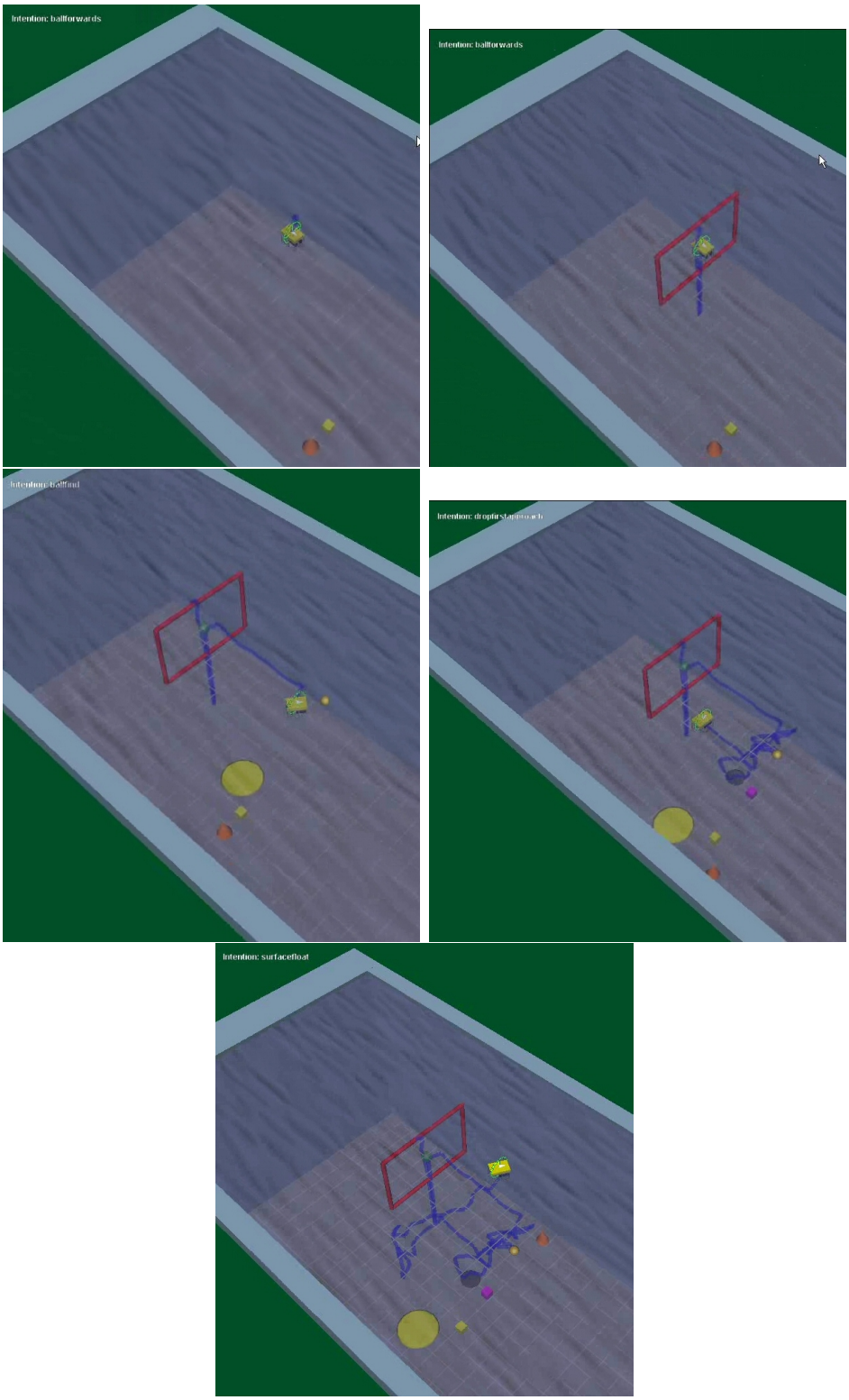


Figure 7.12: Nessie III log playback in ARF showing the state of the object detections made and path of the vehicle as time progresses.

observation and real world testing (using tether, WIFI or acoustic modem for communications).

Monitoring of vehicle state components:

- `Low level Systems Listener` module - Nessie's systems send out various messages regarding their state. For example, the general purpose I/O Card on Nessie's PC reads voltages from the depth, water alarm and on/off switch inputs. These are then displayed in ARF using widgets on the HUD. Water alarms are latching and the robot has special procedures it follows if it senses water, e.g. surface as quick as possible.

- `Alarm Gauge` and `PosNegBarGauge` are used to display alarms and quantities for systems on the HUD.

For Nessie all that is required is a simple OceanSHELL message listener which extracts the relative information and displays it in ARF. Once one component of this type is created it is very quick to create another one because it is usually just the message name and a few fields which need changing in the JavaBean class.

Simulation Components:

- Hyrdrodynamic Model (created for Nessie II)

- Object Detectors (created for Nessie II) - used to detect the virtual objects and output an event. Uses the same `FOVPickSensor` to simulate the sensor and image processing in one.

- `CollidedNodeEvent2DataDetectedEvent` - object detected events are converted to *DataDetectedEvents* which are listened to by the `WorldModel` Component

- `ForwardCameraPixelMsgOutput` - this listens to events from the object detectors and outputs the Nessie III *ObjectDetected* event which comes from Nessie's camera module.

Logging Components:

- `SauceLogger` - this takes inputs from the existing navigation, waypoint, object detected events and outputs an XML log.

- `SaucePlayer` - this opens an XML log file and links to listeners for each type of event, such as navigation, waypoint, object detected etc. These event listeners then alter the virtual environment accordingly, be it vehicle position, current waypoint marker, plotting objects in the virtual world or simply displaying the current intention of the mission on the HUD.

- `DataAssociationEventListener` - this handles detected object events from `SaucePlayer` which are then passed to the `WorldModel` component for displaying in the virtual world.

- `IntentionEventListener` - this listens for intention events. These events generally refer to the current intention of the mission planner. These are then read from the log file and displayed on the HUD.

- `PositionEvent2NavListenerEvent` - position events output by `SaucePlayer` are used to move a specified TransformGroup object in ARF's 3D environment. This would normally be the TransformGroup which represents the AUV.

- `Waypoint4DEvent2WayPointRequestEvent` - waypoint events output by `SaucePlayer` are changed to the standard *WaypointRequestEvent* which can then be interpreted by modules such as the `AutoPos` (autopilot) or `WayPointRqstMsgPlotter` which plots the current waypoint in ARFs virtual environment.

- `WorldModel` - This is used to keep track of detected objects in ARF. It gets input either from logged data via `SaucePlayer` or from object detectors running in ARF. The world model will plot the identified object in ARF. The world model is also capable of linking to `SauceLogger` to output detected objects to log file as well.

7.3.6 Time Analysis

Table 7.1 shows the times taken to create some of the components required. Also it shows how long the scenarios took to create in ARF for the various different purposes. Most of the scenarios were simply modifications of one of the other ones, so creation times were kept to an absolute minimum. The base scenario used to test the PID

Task	Time (HH,MM,SS)
Low level Systems Listener	1,10,00
Alarm Gauge and PosNegBarGauge	1,20,00
Observation Scenario	0,9,00
PID Testing Scenario	0,5,00
Autopilot Open-loop testing Scenario	0,10,00
Mission Planner Testing Scenario	0,20,00
Logger Testing Scenario	0,3,00
Log Playback scenario	0,6,00

Table 7.1: Nessie III Time Analysis: Time required to create components

system originally developed for Nessie was then altered to do much more complicated testing using simulated object detectors for testing the mission planner, open-loop mode of the autopilot, and then the logging and playback system. This demonstrates the extendibility of ARF and how scenarios can be easily modified in a short amount of time for different tasks.

7.3.7 Conclusions and Upgrades

In the future Nessie will have an acoustic modem which will help further with testing and mission observation using ARF. ARF will be able to be used for keeping the operator informed in between communications from the AUV by using interpolation of position, current status information and next waypoint data received from the robot. Nessie will be used for multiple robot cooperation and coordination tasks controlled by DELPHÍS [30] (see section 7.4). Simulated vehicles will also be provided by ARF to make up the numbers of vehicles in multiple vehicle testing. ARF will also provide Hybrid Simulation (virtual sensors) for real robots which do not posses the appropriate sensors required for testing the DELPHÍS system.

ARF helped tremendously for reducing testing time for the mission planner and autopilot for Nessie III. ARF provided facilities for software development to commence before the AUV was even built. This meant that software testing could run concurrently with the low level hardware testing of Nessie III.

7.3.8 Summary

Nessie III demonstrated many different features of ARF due to its numerous applications, these included:

- Operator Training (TR) - the first advantages of ARF were highlighted during in module development of the PID controller. The programmer was able to learn from the virtual AUV how it reacted and responded to thruster commands and then design a new PID controller to control it. This saved potentially many hours of in water testing time. Also the programmer now knew the basic systems of an AUV and how they respond. Although this is not operator training as such, it serves as development training, which in essence amounts to the same thing since they both serve to save time in water.

- OFFLINE, ONLINE, SIL, HIL - The mission planner was tested in SIL fashion, enabling new techniques to be tested. The whole AUV high level systems were then able to be tested in HIL fashion with ARF serving as simulated Nessie with the required simulated object detector sensors.

- AV was achieved during online monitoring of real missions for debugging purposes. The AUV's position and object detections were rendered in the virtual environment so that the system developer could see that, based on what the AUV thought it could see, it was doing the correct action. This served as real world testing for the mission planner and object detection systems.

- VR was demonstrated when testing of the mission planner and autopilot during HIL testing. Simulated object detection data was sent back to the AUV so that the navigation, mission planner and control systems could be tested in the lab.

7.4 Multi-vehicle Testing of DELPHÍS

DELPHÍS [30] is a system for coordinating multiple autonomous vehicles (agents). DELPHÍS agents are capable of executing different parts of a mission plan dynamically and concurrently. The mission is specified in such away that certain tasks can

be separated into smaller units that different agents can choose to execute. The system allows for agents to be placed in the mission at anytime to help execute the overall mission faster. The mission plan is specified by an operator in a language called BIIMAPS [42]. This language is a tree structure which is capable of describing goals, sub-goals, and dependencies between goals. The mission plan can also specify requirements of agents which are to execute a goal, e.g. a goal such as mapping of the seabed in a lawn mower fashion requires that only agents with a certain type of sensor can execute this task. The DELPHÍS system allows agents to dynamically choose tasks from the mission plan and execute them. Each agent periodically broadcasts its current task and its position. This allows other agents to keep their mission plan updated with what has already been done, and who is doing what. There are special algorithms to resolve task conflicts caused when multiple agents pick the same task; this is based on distance to next task, speed and other factors. The DELPHÍS system is capable of predicting which tasks will be chosen by which agents because its the same system on each agent and therefore all agents will pick their tasks in the same way. For example, an agent can assume that one agent will pick task (x) next because it is closest to it when they finish their current task, consequently this agent will pick a different task so there is no conflict between agents. Since each agent predicts in exactly the same way then there should be very little task conflict. An agent can add extra tasks to the current plan by broadcasting objects they have discovered which need further investigation and which may require a sensor which they do not have.

The DELPHÍS system is highly applicable for underwater applications because of its low bandwidth requirements and its ability to function in the absence of reliable communications due to the prediction facilities. Underwater communications are highly unreliable and have very low bandwidth. Therefore, a robust system is needed for coordination between vehicles which doesn't require large quantities of data to be exchanged and has the ability to run with severely limited communication. DELPHÍS was developed specifically for applications of this nature. A mission can be dynamically broken up into separate tasks which can be executed in parallel by separate vehicles. DELPHÍS has been tested in a variety of different ways using the ARF virtual environment. ARF is capable of simulating multiple vehicles quickly and

easily and was therefore very well suited to testing DELPHÍS.

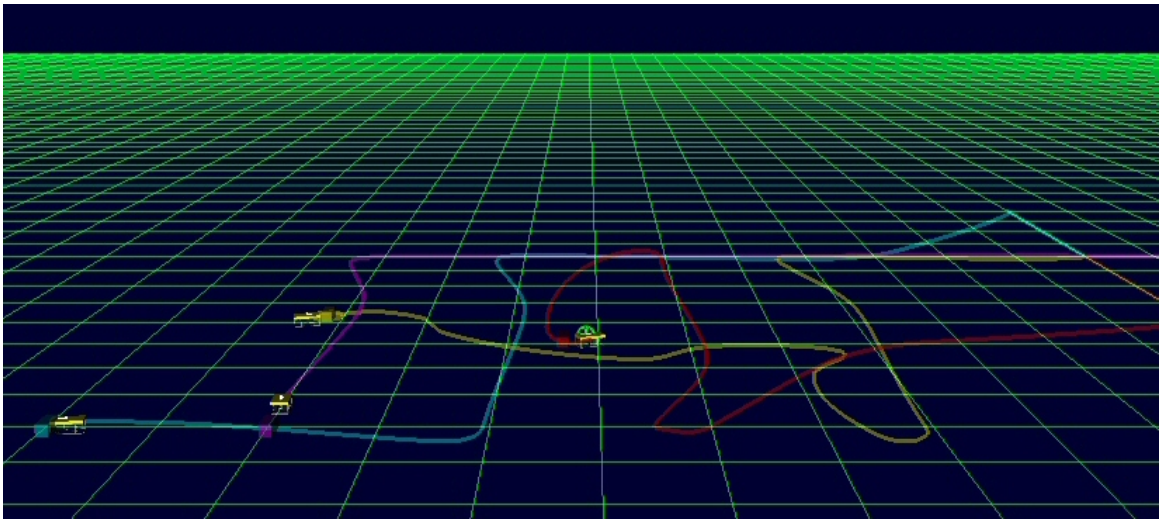


Figure 7.13: ARF Simulating 4 AUVs using DELPHÍS

7.4.1 Testing DELPHÍS

To begin with DELPHÍS was tested hundreds of times in simulation running many different missions with varying levels of communication loss. For most of the tasks the system was tested with increasing numbers of AUVs to see how much time was saved by the addition of another vehicle. However, there is a maximum number of vehicles where the efficiency of a mission cannot be increased further due to the mission not being complex enough, i.e. not enough tasks can be executed in parallel. The ceiling limit for the number of vehicles was generally around 4 AUVs for the missions which were tested. Therefore ARF simulated 4 AUVs which was simply achieved by copy and pasting the same AUV 4 times. Some of the AUVs simulated had different configurations so that certain agents were better at certain tasks, or some agents couldn't do specific tasks. For the cases discussed here, the two types of AUV which were simulated and used for real were Nessie III and REMUS.

ARF's primary purpose other than simulating the AUVs for testing the system, was for the designer of DELPHÍS to observe how the AUVs broke up the mission plan (figure 7.13 shows 4 simulated AUVs cooperating using the DELPHÍS system to break apart a lawnmower into many different legs so that the same area is surveyed in

less time, similar to a divide and conquer approach). ARF was used to view replays of missions where there seemed to be anomalies in the data, i.e. to see what the AUVs actually did since most testing was run faster than real-time.

7.4.1.1 Integrating DELPHÍS on REMUS

The simulated tests proved that DELPHÍS was very efficient in dividing up missions and allowing multiple AUVs to cooperate together in environments with poor communications. The next step was to prove that it worked using two real AUVs with acoustic communication capabilities. This required various levels of testing before the full scenario could be tested since the AUVs had to be upgraded to use acoustic modems. The REMUS AUV uses a WHOI acoustic modem. The same modem was purchased for Nessie and installed. Another modem was attached to a surface computer for monitoring REMUS via Hydroid's VIP software, and also for monitoring the state of DELPHÍS output by its broadcast mission state messages. In some of the testing a simulated vehicle was run on the topside computer communicating via the surface acoustic modem. This meant that one real AUV could be tested at a time, with one virtual AUV, to make sure it worked and communicated properly. This proved to be very useful since there were problems with the DELPHÍS system after being integrated onto the REMUS which would have been very hard to diagnose had there been another vehicle in the water introducing another possible source of errors.

The first line of testing to be executed was simply to test the REMUS being controlled by DELPHÍS. DELPHÍS was initially executed on Hydroid's own Remus simulator, as this was as close to hardware-in-the-loop that could be done with the REMUS. However, the simulator is supposed to have all the same interfaces as the real REMUS. This was before any acoustic communication was available, so it was simply to check that the DELPHÍS software could take control from REMUS at the appropriate point in its mission and start executing DELPHÍS's alternative mission. Since no other vehicles were running, REMUS completed all the mission goals. Upon completion of the DELPHÍS mission, control is handed back to REMUS so that it can return to its mission end point and wait for collection. Once DELPHÍS was tested and validated on REMUS, the next step was to integrate the Acoustic modem onto

Nessie and enable REMUS's acoustic modem interface so that DELPHÍS could use it.

7.4.1.2 Multiple Hardware-in-the-loop Testing of DELPHÍS

To begin with, Nessie was tested using DELPHÍS in the dry using a hydrodynamic model simulated in ARF. DELPHÍS was running on Nessie and acoustic communications from DELPHÍS were tested by placing the surface acoustic modem near Nessie's modem. The system was then tested using Nessie in hardware-in-the-loop fashion with its thrusters and navigation systems simulated by ARF, and also a simulated REMUS was provided by Hydroid's REMUS simulator. A third simulated vehicle was then introduced in ARF to check that communications could be sent down the surface acoustic modem to Nessie. Therefore, Nessie's broadcast communications had been proved in the dry. It was then merely a case of testing them in the wet.

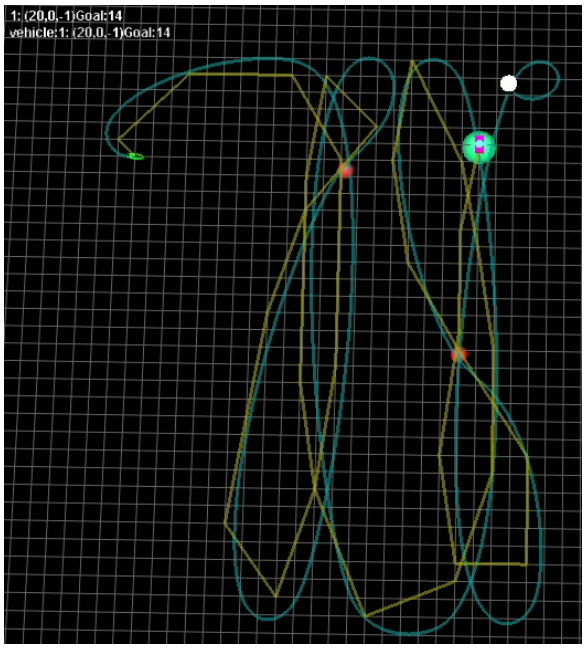


Figure 7.14: Real AUV Position Vs Acoustic Updated Position

ARF was used to compare the simulated vehicle path to the path created by updates from the acoustic messages from DELPHÍS. This showed that the update rate of the acoustic communications was sufficient to keep the operator situated and wasn't to far lagged behind. The HUD was used to display which goal ID each vehicle

was executing and a readable (x,y,z) value for its position. Trails were plotted for both acoustic updates and vehicle navigation to correlate the paths of each. This showed that delays of around 25 seconds between each acoustic transmission was a sufficient update rate for the operator. Figure 7.14 shows the acoustically updated vehicle and the real position. The acoustic updates are shown as yellow trails, and the real position shown in cyan trails. The real vehicle is depicted by a white circle and the acoustic position represented as a cyan circle with a pink rectangle. The red cone objects are the detected mines. The mission shows only one AUV executing a lawn mower and inspecting the detected mines.

7.4.1.3 Real world testing of Nessie III using DELPHÍS

The next stage of testing required that Nessie be tested in a real lake environment. This would be a test with a single vehicle to check that the acoustic communications sent out by DELPHÍS were correct. The next step involved running a simulated vehicle on the surface computer which also did some of the tasks. This tested the prediction part of the DELPHÍS system as the system accurately predicted which tasks would be executed by each agent. The test was carried out at a reservoir, just outside Edinburgh, called Harlaw. This test worked without any problems, thus all that remained was making sure REMUS's acoustic communications worked.

7.4.1.4 Real world testing of REMUS using DELPHÍS

REMUS had already been tested using DELPHÍS and this worked. Therefore, all that was left to do before a multi-vehicle mission could be executed was to test that REMUS broadcast its state correctly over the acoustic link. There was a small problem with the way REMUS allowed its acoustic modem to be used that meant that in order for REMUS to send an acoustic message it first had to receive an acoustic message asking for one (polled I/O). This meant that the surface computer had to continually send out a pole message so that REMUS would send an acoustic broadcast message which could be picked up by other AUVs. Once this problem was fixed, REMUS was sending out acoustic updates which looked correct.

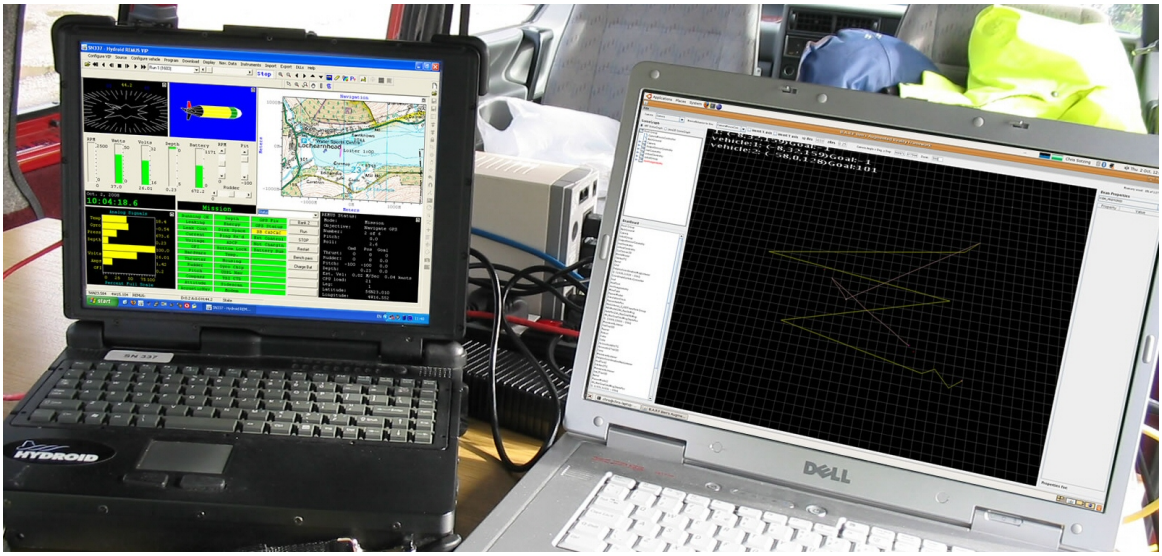


Figure 7.15: REMUS Vip Software and ARF viewing Acoustic Updates from Multiple AUVs

7.4.2 Multi-vehicle Trial using REMUS and Nessie III

The 1st real world trial of two AUVs actively collaborating and cooperating was visualised using ARF. ARF required 1 new component which listened for an OceanSHELL message which contained the location of each AUV, their current GOAL and whether they had detected any objects requiring investigation. The ARF scenario plotted the vehicle positions and the trails indicating where the vehicles had been. ARF also plotted the positions of any objects detected by the AUVs. The mission was executed by starting REMUS first, and then starting Nessie moments after. There were simulated sensors on REMUS to make fake object detections similar to what a CADCAC (Computer Aided Detection Computer Aided Classification) system would output. The exercise was similar to that of mine counter measures (MCM) vignette required by the Ministry of Defence (MOD). The REMUS AUV would select a goal to carry out a survey mission and map a large area with a lawn mower looking for mine like targets. The Nessie AUV would be used for inspection of identified targets because it is too slow to survey large areas quickly. Upon REMUS discovering new mine like targets, Nessie would receive the update to the plan and select one of the mine inspections as its goal. Nessie was unable to select the survey goal because it did not have the survey capability, i.e. not fast enough. REMUS would then detect another mine,

which Nessie would investigate also. REMUS would finish before Nessie had finished because there would be nothing left for REMUS to do, since the remaining task was being currently executed by Nessie and REMUS predicted that Nessie would get to the location of the mine before it could. Figure 7.15 shows the command and control setup. ARF was used to display which goal each AUV was currently executing and also where the vehicles were and where any detected mines were. Figure 7.16 shows the plots of the two AUV's trails and the detected mines in ARF.

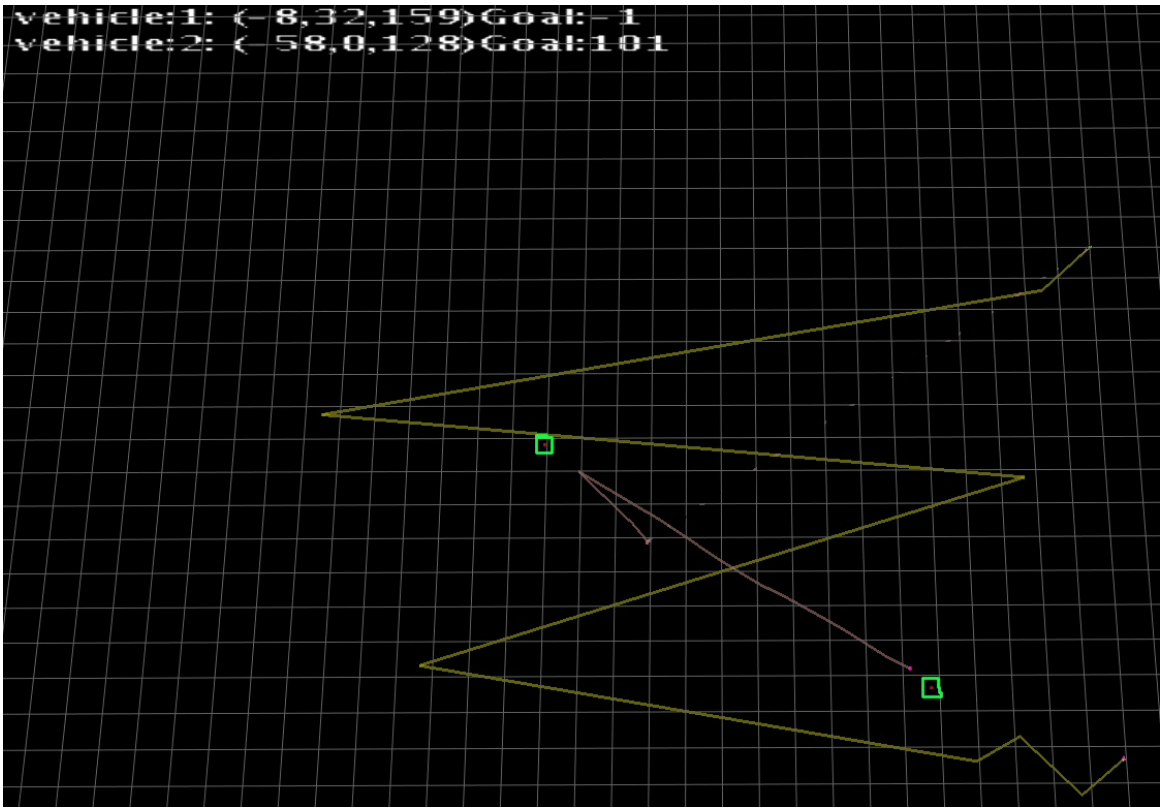


Figure 7.16: DELPHÍS Mission Status displayed in ARF

The pink trails represent Nessie. Nessie started off very close to the first mine detection. Nessie then goes to the next mine detection. The mine detections are marked with a red dot and highlighted with a green box. The trails do not appear to go all the way to the mines. This is due to the infrequent acoustic updates (once per 30 seconds or every couple of minutes if acoustic conditions are noisy). The reason Nessie doesn't appear to reach the last mine is because an update is not received whilst Nessie is at the target because the DELPHÍS system finishes and hands back control to Nessie; therefore no longer sends acoustic updates from DELPHÍS. The

Task	Time (HH,MM,SS)
DELPHÍS Acoustic Message Plugin	2,20,00
4 AUVs Simulation Scenario	0,10,00
ARF Observation Scenario	0,6,00
ARF Simulation and Observation Scenario	0,5,00

Table 7.2: DELPHÍS Time Analysis: Component and Scenario Creation

yellow trails represent REMUS's lawn mower.

ARF was used only for observation in this final test. Another simulated vehicle could have been added, but there wasn't enough time on this trial because Nessie's batteries went flat. However, this had been tested before at Harlaw loch with Nessie and a simulated REMUS.

7.4.3 Time analysis

Table 7.2 shows the amount of time required to produce the ARF components and the time taken to create the virtual scenarios. Most components were already available from previous projects, and as such were used to save time. ARF used the ALI abstraction layer, so no extra I/O components needed creating for ARF to simulate the AUVs. The only component requiring adding was one which listened for DELPHÍS *Mission Status* OceanSHELL messages. This required linking to a text label component on the HUD of ARF which then displayed data about each vehicle such as DELPHÍS Goal ID etc. This plugin was connected to the simulated Nessie `WorldModel` used to plot object identifications, including position of AUVs. Thus made re-use of components created for the SAUC-E competition.

When the requirements of the tests changed, ARF's scenario had to be changed. It was always a simple addition or alteration which needed making and therefore the whole scenario did not need redesigning. The ability to quickly add another simulated vehicle, or add feedback for another real vehicle was very useful as configurations could be tested quickly. In ARF it is rare that a complete scenario needs creating from scratch since so many uses have a common base which can be used throughout many scenarios, thus saving large amounts of time overall.

7.4.4 Summary

Apart from ARF being used for observation purposes, multiple AUVs have been simulated which helped tremendously for observing behaviours and testing of the DELPHÍS architecture (see section 7.10.1 for other applications of DELPHÍS). Basic object detection sensors provided a simple but effective method of outputting data from the virtual environment to DELPHÍS. Detections of certain types of objects meant that new goals were added to the plan. Generally these would be of the form: identify an object with one vehicle, then classify that object with another type of vehicle with the appropriate sensor. Thus some of the simulated AUVs were only capable of detecting the objects, whilst others were capable of inspecting and classifying those objects.

The use of ARF for real-world observation (OM) was quickly recognised as it provided a simple architecture that was easily extended for visualising multiple tasks at once. For DELPHÍS it was paramount that the operator be able to see visually what both AUVs were doing. No other software could provide the visualisation of multiple vehicles in a dynamic and changing environment. The plotting of targets meant that the operator always knew where the vehicle was heading and why. If more capabilities were required, the scenario can be easily extended to show more data. Testing and validation of DELPHÍS harnessed ARF's ability to provide flexible scenarios which were used for many different mixed reality and testing applications demonstrating features of: OFFLINE/ONLINE, HIL, MHIL, SIL, AV and VR.

The time taken to produce the scenarios required for DELPHÍS was minimal. The largest overhead was creating the OceanSHELL message listener which kept ARF updated. However, the time taken to create this was tiny in contrast to the time taken to do everything else relating to DELPHÍS. All other components required were already available from previous projects, thus demonstrating high code reuse. The time savings were attributable to ARF's flexible architecture allowing components to be easily interfaced with and used for many purposes. This highlights the flexibility and extendibility of ARF and its inherently useful applications in testing of systems for underwater applications.

7.5 RAUBoat I: Autonomous Surface Vehicle Testing

RAUBoat is a a maritime Autonomous Surface Vehicle (ASV). It is actually a boat rather than a land based surface vehicle. Its name lends itself to its inner workings as most of the technology came from RAUVER and is in essence an amalgamation of technology from RAUVER and Nessie. The idea of RAUBoat is to provide surface surveillance and tracking of AUVs. The second purpose is that RAUBoat act as a communications gateway from wifi/radio to underwater acoustic and vice versa. RAUBoat will follow the AUVs to maintain a short distance between itself and them. Acoustic communication is much faster and more reliable over short range. The acoustic communications can then be easily routed anywhere in the world via radio or Iridium Sat phone; thus providing much more reliable data transmission to the surface.

RAUBoat only requires limited sensors since navigation can be done via GPS and compass alone. RAUBoat will also have an underwater acoustic modem for communications with AUVs and a wifi antenna for surface communications back to operations base. RAUBoat will be able to follow AUVs keeping in communication with them, and thus helping de-risk operations.

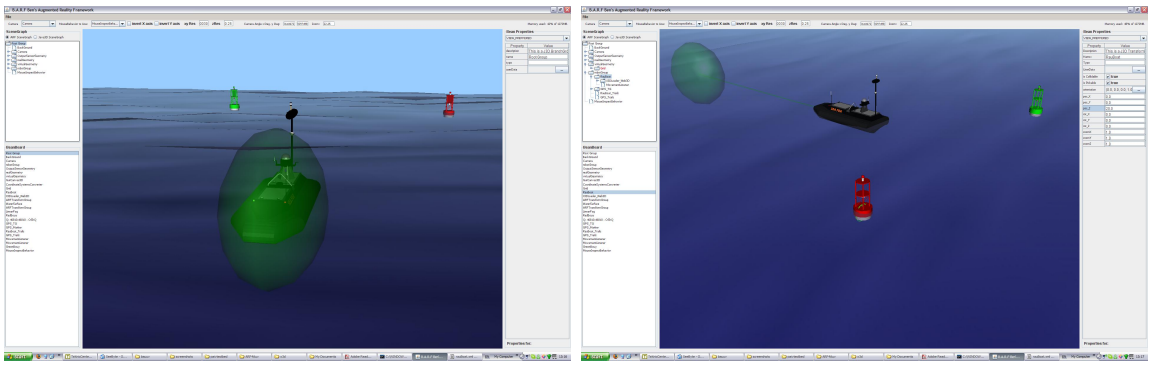


Figure 7.17: RAUBoat Mission Observation and GPS/Compass Testing

RAUBoat is equipped with 2 rear thrusters allowing for survey vehicle class operations as this ASV is not capable of lateral movement.

ARF was initially used as a testing mechanism for checking the GPS variance

Task	Time (HH,MM,SS)
GPS ARF Component	0,30,00
ARF Observation Scenario	0,10,00

Table 7.3: RAUBoat Time Analysis: Scenario and Component Creation

and compass data. However, ARF's purpose grows with the maturity of RAUBoat's capabilities. ARF will be used for a command and control platform using information relayed by RAUBoat about itself and other vehicles working within range of itself. Monitoring of multi-vehicle operations will become much easier. Figure 7.17 shows the ARF Scenario which shows the ASV's position and compares this with the readout given by the GPS/Compass. The green sphere in the image represents the GPS position and compass heading. Table 7.3 shows the amount of time needed to create a scenario in ARF for RAUBoat.

ARF mainly served as an observation platform (OM) for the output of the vehicles systems. However, using ARF enabled the developer to see what needed to be done to make the ASV better. This scenario showed that a vehicle dynamics model with filtering of the GPS and compass is all that would be necessary for the ASV's navigations systems.

7.5.1 Time analysis

Table 7.3 shows the amount of time required to produce the ARF components for the GPS feedback and for creating the scenario itself. Many components such as a compass listener have been used from other projects.

Using ARF meant that instant visual feedback was available whilst testing. This meant that technologies such as GPS and compass could be evaluated. The conclusions were that a basic autopilot could be tested fairly soon if some filtering was done on the GPS. Further trials involving the new systems will be tested in due course and then RAUBoat will be ready to fulfil its purpose as an AUV tracking communications gateway. The main feature demonstrated of ARF was the ability to easily re-use modules developed for other systems and new modules can be quickly and easily developed that provide extra feedback from the remote environment.

7.6 Autotracker: Testing of Autonomous Pipeline Inspection System

Oil companies are expressing interest in AUV technologies for improving large field oil availability and, therefore, production. It is known that Inspection, Repair and Maintenance (IRM) comprise up to 90% of the related field activity. This inspection is clearly dictated by the vessels availability. One analysis of potential cost savings is using an inspection AUV. The predicted savings of this over traditional methods for inspecting a pipeline network system are up to 30%.

Planning and control vehicle payloads, such as the AUTOTRACKER payload [71], can provide such capabilities. However, as mentioned, vessel availability and off-shore operation costs make these types of payloads a difficult technology to evaluate. ARF can provide simulated sidescan sonar data for a synthetically generated pipeline rendering on the sea bottom. These capabilities provide a transparent interface for the correct and low cost debug of the tracking technologies. Furthermore, potentially complicated scenarios, such as multiple pipeline tracking and junctions of pipes, can be easily created to test the pipe detection and decision algorithms of the AUTOTRACKER system (see figure 7.18). This could not easily be tested in the real environment as real-time debugging is not available and the potential for incorrect decision due to confusion about which pipeline to follow is high, therefore higher risk of loss of an AUV.

7.6.1 Additional Components Required

ARF required the addition of two new types of sensor, a multi-beam bathymetric profiler sonar and sidescan sonars. The multi-beam had basically been implemented already because it is the simplest type of sonar to simulate and therefore most of the work had already been done when creating the forward-look sonar for other projects such as BAUUV [72] (see section 7.7). The side scan sonar required a bit more work because it required gathering the angle of reflection as well as the distance of each intersection. Due to the nature of the ARF SceneGraph as many sensors can be added to the vehicle as required. For instance, for the sidescan, two identical sonars were

added facing in different directions, and then the output of these was amalgamated to produce one image (shown in figure 7.18).

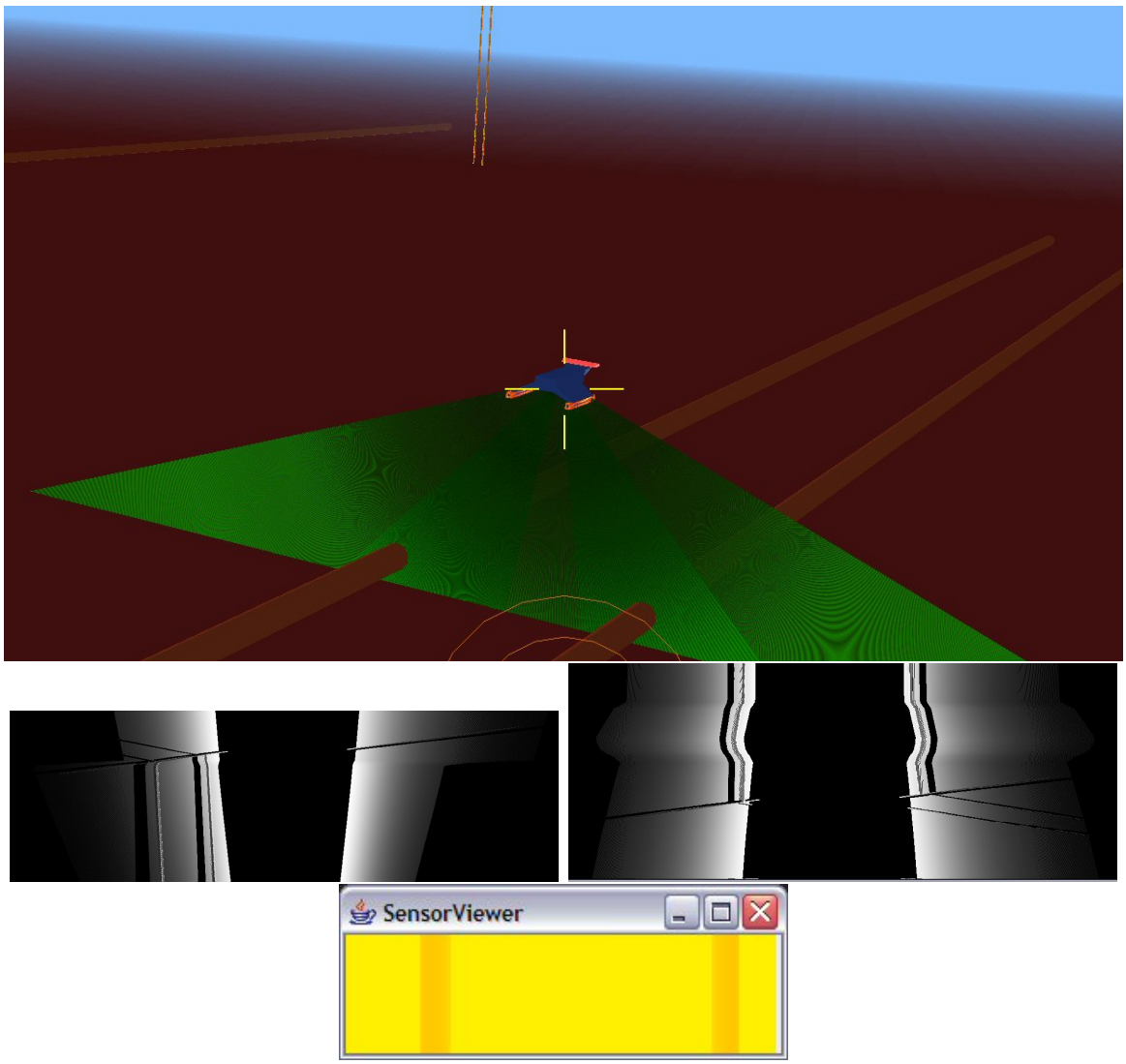


Figure 7.18: Simulated sidescan, multi-beam and corresponding simulator view

The project required two output OceanSHELL messages to be sent which carried the sidescan data and the multi-beam data. This meant that ARF could be substituted seamlessly for the real vehicle as it would output simulated navigation messages, provide an autopilot and output the necessary sensor messages to the tracking software. A plugin was created for ARF which read in the legacy data from file of an existing pipeline and plotted it over the current sea bottom/terrain of the virtual environment.

7.6.2 Summary

ARF enabled the AUTOTRACKER software to be tested on more complicated pipeline configurations, such as the one shown in figure 7.18. This scenario shows the AUV tracking two pipes, which would not have been possible until real sea trials and even then the particular test scenario may not have existed or be too dangerous. The main use for this project was for real-time simulation and SIL purposes as AUTOTRACKER will be fitted onto an AUV such as GeoSub [54], Remus [52] or Gavia [53]. The testing of the entire AUTOTRACKER system was done using an external GeoSub simulation which was synchronised with ARF, which provided the sonar sensor simulation. Simulated tests are vital to make sure that the AUTOTRACKER systems work before the real trials commence. Consequently, ARF helped to de-risk some of the scenarios before using a real AUV. Then the only remaining task is to integrate AUTOTRACKER onto the real AUV and perform HIL testing, then real trials. However, if the GeoSub simulator was an exact clone of the GeoSub systems, HIL testing is pretty much carried out through using their simulator.

7.7 BAUUV: Obstacle Detection and Avoidance Testing

One of the most common problems for unmanned vehicles is trajectory planning. This is the need to navigate in unknown environments, trying to reach a goal or target, while avoiding obstacles. These environments are expected to be in permanent change. As a consequence, sensors are installed on the vehicle to continuously provide local information about these changes. When object detections or modifications are sensed, the platform is expected to be able to react in real time and continuously adapt it's trajectory to the current mission targeted waypoint.

The Battle space Access for Unmanned Underwater Vehicles (BAUUV) [72] project was a project designed to address this exact problem, i.e. detect obstacles underwater and re-plan the current mission trajectory to avoid them. Obstacles could be anything such as nets, buoys, boats etc.

The real AUV would use a forward looking sonar to detect obstacles. Obstacles could in theory be anything, so the detection algorithm had to be quite generic and report shapes of obstacles rather than identifying what they are. In some senses this is more complicated than the pipe detection algorithm used in AUTOTRACKER because an obstacle could be any object in the way, but its important that only real objects are reported. However, the detection algorithm needed testing with the re-planning system and then would need testing HIL style on the real vehicle as well.

This was a perfect opportunity for ARF to be used. ARF provided a simulated forward looking sonar and outputted its image to a file which was continually read by the obstacle detection system. The obstacle detection system would then send OceanSHELL messages describing any detected obstacles to the re-planning system which would create a new course around the obstacle. ARF provided simulated navigation and autopilot using the same OceanSHELL messages as the real AUV. This meant that the trajectory re-planner merely had to send out OceanSHELL waypoint request messages to the simulated autopilot to make the simulated vehicle manoeuvre around an obstacle.

The sonar simulation came in very useful for this project, and also for lots of other projects. The generic object message output by the obstacle detection system could also be used for testing of SLAM systems being developed in the OSL. Without the sensor simulation, testing of BAUUV would have been no where near thorough enough and bugs would have gone unnoticed. Using ARF meant that anyone working on the project could quickly run the test scenario, load a mission into the external mission planner, and run as many tests as they wanted on different obstacles. This speeded up development and meant that many errors which would have been harder to detect after integrating onto the real AUV were detected before.

The extra requirement for this project of ARF was that the virtual world had to be configurable so that layout and type of obstacles could be changed. In order to make changing of obstacles quick and easy an already established virtual world modelling format was used to specify what appeared in the virtual world; known as X3D. X3D uses XML to define types of geometric objects, their positions and the vertex data to construct the object, as well as providing easy to use primitive shapes

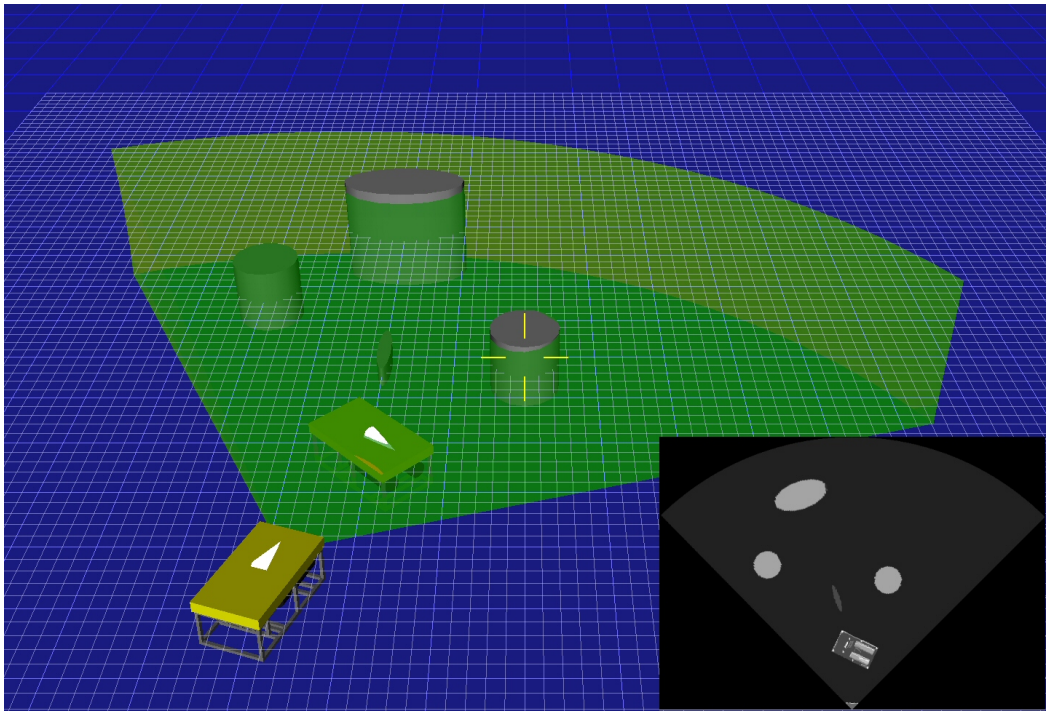


Figure 7.19: Simulated Sonar of Obstacles

such as box, sphere, tube and cone. The actual scenery was created in 3rd party 3D modelling software instead of using ARF's own primitive Java3D types. This meant that more complex scenery and obstacles could be created more easily if need be. Therefore, ARF required a loader component for importing X3D mesh files into ARF. A component was designed for ARF which used 3rd party Java3D loaders to load the X3D files as well as many other formats. The obstacle X3D files could then be loaded into ARF ready for the virtual AUV to detect with it's sonar. Figure 7.19 and 7.20 shows examples of a basic ARF scenario where obstacles have been loaded and forward looking sonar has been simulated. The sonar image shows that simulated sonar can be generated for any 3D object such as the Rauver 3D mesh and barrels shown.

To make the simulator more useful as a visualisation tool, special camera positions were created, such as birds eye view, and a free camera which the user could move with the mouse. Also the field of view of the sonar was displayed on the AUV so that the operator could instantly see which obstacles should be appearing in the sonar image. The output of the sonar was also displayed on the screen for a sanity check as

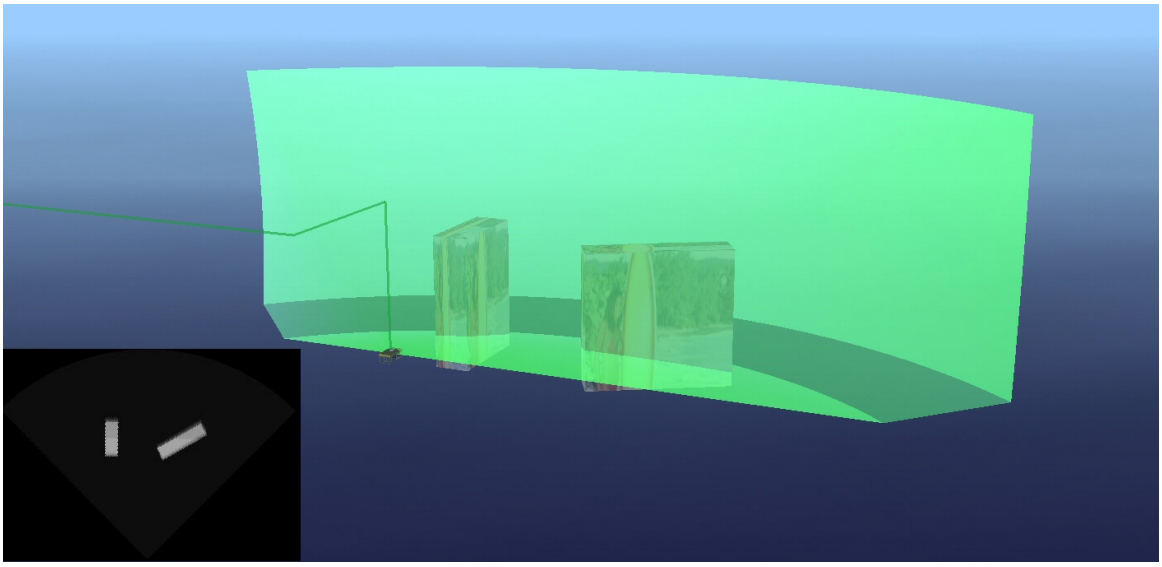


Figure 7.20: The BAUUV Pure Simulation Scenario: showing sonar field of view and simulated sonar image (bottom-left)

well as being sent to the obstacle detection module.

7.7.1 Hybrid Simulation

ARF proved very useful for pure simulation testing. ARF was also to be used as a visualisation tool for monitoring the real AUV and the obstacles it detected in the real environment. However, testing these kinds of adaptive algorithms requires driving the vehicle against man-made structures in order to analyse its response behaviours. This incurs high collision risks on the platform and clearly compromises the vehicle's survivability.

A novel approach to this problem uses ARF to remove the collision risk by using Hybrid Simulation. The approach uses simulated sonar for rendering synthetic acoustic images from virtually placed obstacles. The algorithms are then debugged on the real AUV whilst performing avoidance manoeuvres over the virtual obstacles in the real environment. Figure 7.21 shows the required framework components. In essence, the real sonar onboard the AUV is disabled and instead simulated sonar images are sent to the detection algorithms from ARF's virtual environment. ARF is kept in synchronisation with the vehicles position using OceanSHELL navigation messages from the AUV. If the vehicle has wifi or is on an Ethernet tether, ARF can be run

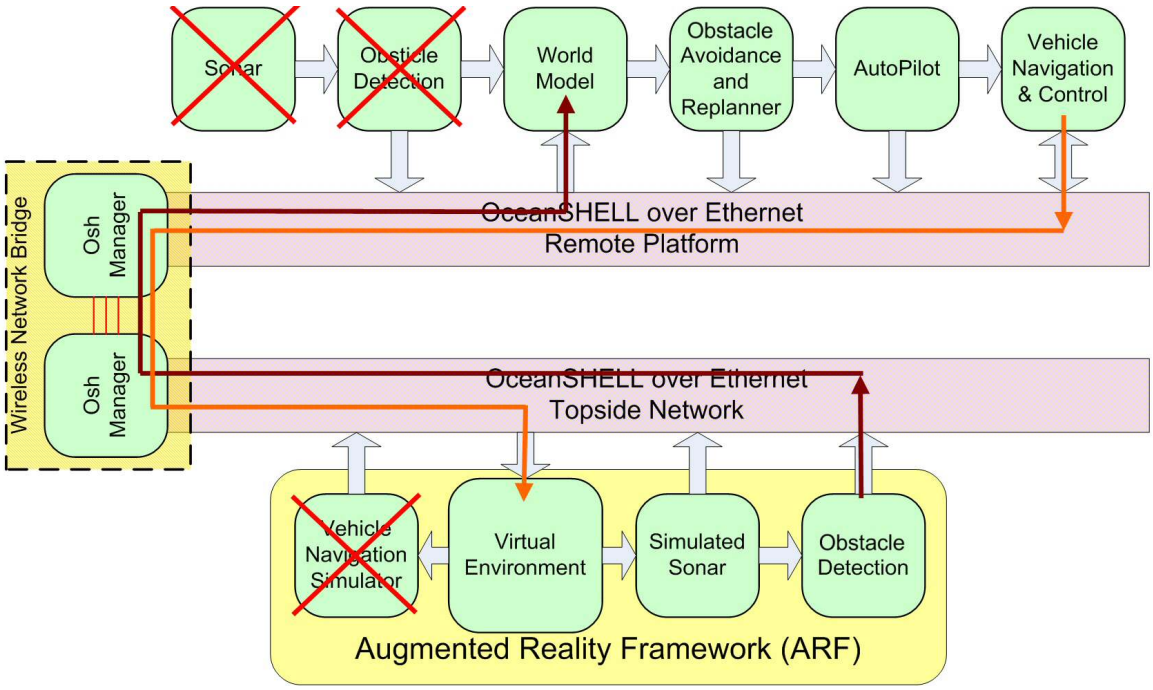


Figure 7.21: This diagram shows how OceanSHELL provides the backbone for switching between real and simulated (topside) components for use with Hybrid Simulation.

on a topside computer as shown in figure 7.21, however, ARF can also be run on the actual vehicle systems if there is enough processing power. The advantage of running ARF on the topside is that the operator can see the virtual environment, where the vehicle is and where the virtual obstacles are. This enables the operator to see if the real vehicle responds in time to the obstacles detected. If the vehicle doesn't respond in time it will appear to hit the virtual obstacle in the virtual environment. In addition to this feedback the virtual environment can also plot the positions of the obstacle detections from the obstacle detection OceanSHELL messages output by the real AUV. This helped the operator analyse the behaviour of the AUV. The AUV appeared to be giving the obstacles a very wide berth when avoiding them and also on occasion nearly running into them. This seemed to be a problem with lag time between sonar image being ready and obstacle detections being made such that the vehicle thought the obstacle was further away than it was. This was because when the sonar had produced an image it would log the position of the AUV so that any obstacle detections could be given in absolute world frame coordinates rather than vehicle frame. The problem was that the sonar took up to 1 second to produce an

image. This meant that the image was already 1 second old by the time the AUV position was logged. The problem being that a mechanical sonar takes time to scan. So if the AUV is moving forward then the obstacle in the image will appear closer on one side of the image than the other since the sonar scans from left to right as the vehicle is moving forward. This caused object detections to look skewed and slurred. It also meant that detected obstacles were plotted 1 second later, in AUV position terms, than when the sonar started its scan. This meant that obstacles were plotted slightly in the wrong position, i.e. detected further away than they actually were. This error would have been very hard to detect in real testing because the absolute position of the objects would not be known exactly. Thus when using simulated sonar, the plotted obstacles detections did not line up with the real obstacles all the time, unless the AUV was moving quite slowly. Rotation of the AUV also cause obstacles to be either squashed, rotating with the sonar scan direction, or stretched when rotating in the opposite direction to the sonar scan. Figure 7.22 shows how the moving AUV caused the detected obstacles to not exactly line up with the actual (simulated) obstacles during hybrid simulation. In the top picture the AUV is moving faster which is why the purple bounding box of the detected object appears slightly more behind the real object than in the bottom image.

Once the alignment problem was compensated for, ARF was used for observation purposes to see the real AUV avoid real world obstacles.

7.7.2 Summary

Without ARF, development and testing would have taken far longer. Due to ARF's flexible nature, the same scenario was able to be used for the various different types of testing, i.e. Pure Simulation (OFFLINE and ONLINE), Hybrid Simulation (HS) and online monitoring (OM). Consequently, both AR and AV were also demonstrated since the real AUV was having its perception of reality augmented by simulated data from the virtual environment, and at the same time object detection data was being relayed back from the AUV and the virtual environment augmented with this data. This shows ARF's ability to have potentially limitless numbers of modules which synchronise data to/from the real-world. As applications get more complicated

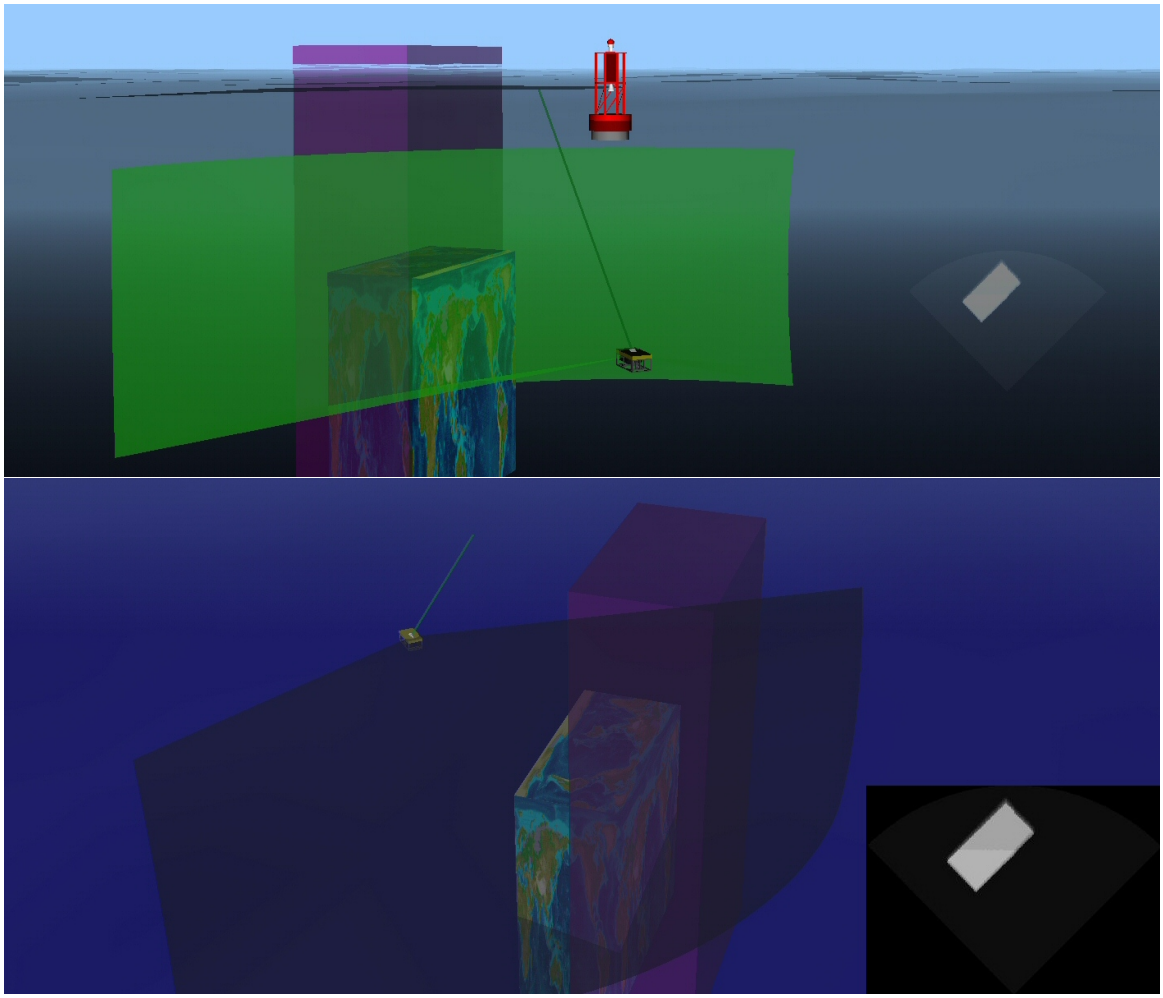


Figure 7.22: Detected obstacles did not always align with simulated obstacles when AUV was moving, thus facilitating algorithm visualisation and providing implementation debugging.

ARF is able to expand its component set and capabilities at the same rate allowing for very complex scenarios to be created which demonstrate many different concepts simultaneously.

For further information on BAUUV, a detailed description of the evaluation and testing of obstacle avoidance algorithms for AUVs can be found in [73] and [71].

7.8 SeeByte Ltd: SeeTrack Offshore Remote Awareness System Testing

SeeByte has a product for the offshore ROV market called SeeTrack Offshore (STOS). This product provides a Dynamic Positioning (DP) system which can be retro-fitted to existing ROVs. The minimum this product requires is an ROV with a standard joystick control used to control the ROV's thrusters. In addition a DVL is required in order to provide the surface STOS system with reliable and accurate bottom-lock velocity information. The DVL is sometimes an integral part of the ROV but can also be added to the ROV if it is not provided. The Workhorse DVL also contains a compass which can be used if the host ROV does not have one. The ROV must also output what is known as a survey string which contains information about the ROV, such as depth. Figure 7.23 shows a Seaeye Falcon ROV. This ROV was the first ROV used to test STOS. It required RDI's Workhorse DVL fitting to it to provide velocity, altitude and rotation information to the surface STOS computer.

STOS provides a surface system which takes the output from the DVL, which provides accurate velocity information, and uses this to work out where the ROV is. This information is then used by autopilot module of STOS to provide dynamic positioning by controlling the ROV's thrusters, i.e. the operator doesn't have to control the ROV's thrusters, only a waypoint is required and the ROV will go to it. A 2D virtual reality representation of the ROV and its environment are provided so that the pilot can see the actual position of the ROV in relation to other assets around it. A screenshot of the STOS user interface is shown in figure 7.24. The virtual world can also be used to control the ROV by providing a point and click control interface. The ability for the ROV pilot to control the ROV and not have to worry about sea

currents means that they can execute tasks much more quickly and efficiently. The STOS surface system takes care of adjusting the thrusters of the ROV so that the pilot has nothing to worry about.

Figure 7.23: Seaeye Falcon ROV with RDI Workhorse DVL fitted

7.8.1 Tuning Problems

This system can be retro-fitted to existing ROVs. It requires no modification to the ROV if the ROV already has a DVL. However, ROVs range in size, weight and manoeuvrability. Thus, each STOS system has to be tuned to the hydrodynamics and thruster limits of each particular vehicle. This can take quite a long time to get a fine tune which provides stable control for the ROV operator. In order to tune the vehicle an engineer must first be trained how to do the tuning process and this too can be time consuming.

7.8.2 Module Development

As mentioned earlier STOS provides a virtual representation of the ROV and surrounding objects in order to provide better visual queues to the operator to help them orientate themselves, i.e. providing intuitive visual feedback of the remote environment when the ROV's video cameras do not provide anything useful. Therefore, various visual objects are required to help the operator see where they are. These could include:

- Bathymetric chart - to give the operator an idea of the real world position of the ROV.
- Static installation positions - a standard data file containing geometry of known static installations could be loaded and displayed.
- Non-static object positions - these could be other moveable objects in the area such as ROVs or boats. It would be useful for the operator to see the positions of other equipment working in the vicinity of their own.

All of these are independent modules which could be easily developed separately from the core of STOS itself, meaning that STOS should not be required to test them. However, testing of the modules must then be done before integration with STOS, and this can only be done if there is a method of doing this without using STOS itself. Testing a modules full functionality using STOS itself is tedious due to the restrictions placed on the final user interfaces of the system, meaning that a module may not have been tested for all eventualities. Therefore something like ARF is required so the programmer can put the modules through their paces before being integrated with the overall STOS system.

7.8.3 System Architecture

STOS is built on the Java [58] platform and makes use of the Java3D [3] API to provide the virtual world GUI for the operator. STOS also uses OceanSHELL [6] for all of its inter-module communications. This means that modules made for STOS can be easily tested and integrated into ARF and vice versa since the base architecture is the same.

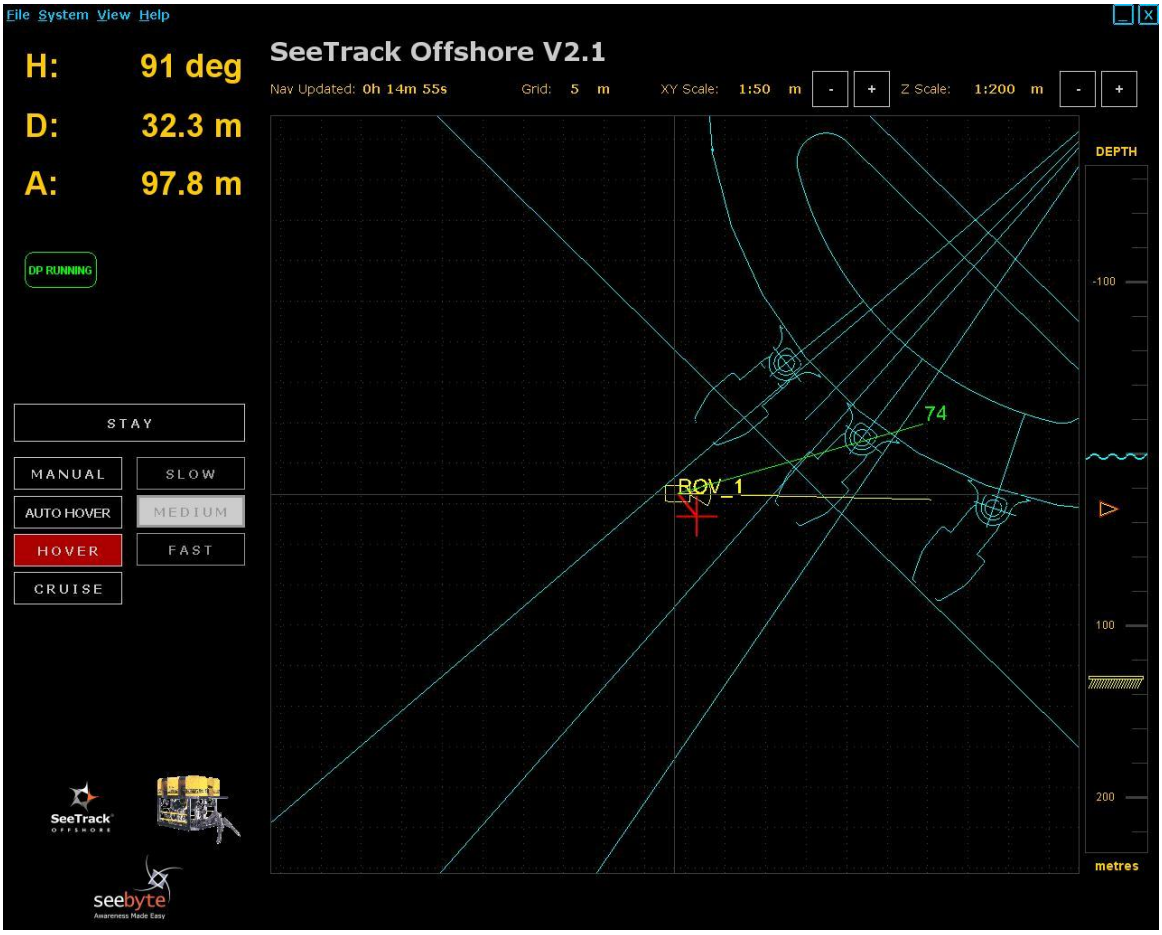


Figure 7.24: SeeTrack Offshore User Interface

Therefore, modules for STOS can be part of the ARF component library providing more functionality for ARF and vice versa making ARF components available for use in STOS. ARF could be used to fast prototype new interfaces and functionalities for STOS.

OceanSHELL is an integral part of both ARF and STOS. Consequently, HIL and PS testing of STOS using a simulated ROV provided by ARF is easily possible.

7.8.4 Advantages of using ARF to test STOS

It is possible that ARF could be used to dramatically reduce further development time of SeeTrack Offshore. Further more, ARF could provide a simulated virtual environment and simulated ROV for use with the STOS system in a software-in-the-loop fashion. The potential is therefore realised for training (TR) of persons on STOS without a real ROV. End users of STOS could learn how to use it without having to learn with an expensive ROV. Installation engineers could be trained on how to tune STOS to the hydrodynamics of the virtual ROV. Finally, there is the potential for being able to do a simulated tune for a vehicle if certain vehicle properties are known a-priori i.e. STOS could be tuned on a simulated ROV with the same hydrodynamic properties as the real ROV. Thus reducing the time required to do a tune for the real vehicle in the field.

7.8.5 Cost Savings

To hire all the necessary equipment to use a small ROV off the end of an on shore pier cost 1575 (GBP) per day. That doesn't even include the cost for the time of the people using the ROV. To use an ROV offshore costs nearer 10,000 (GBP) per day, and that is merely for the privilege of being on the control ship before the ROV is even used. For these reasons training people to use the systems on a virtual ROV would save a huge quantity of money which could be better spent on further research. In comparison to these costs, the cost of the time taken to develop a virtual scenario in ARF for STOS is negligible.

7.8.6 New ARF Components

In order to provide a suitable development and testing platform, extra simulated components need to be added to ARF to represent the virtual ROV. Some of the required components currently exist from other projects. The components which are still required are a benefit to ARF because they are generic to ROV and AUV applications and therefore will make an excellent addition to the ARF component library. Once all the components are created, the testing scenarios for many different STOS purposes can be created quickly and easily. Scenarios in ARF can be created in a matter of minutes, and therefore the cost overhead need not even be calculated. The required components are as follows:

- DVL - the DVL will need to simulate the data that a normal DVL produces. For this application, the DVL is only required for detecting the velocity in the (X,Y) plane. This relies on the DVL having bottom-lock, i.e. being able to track the sea bed.

- Magnetic compass - there are 2 kinds of compass which can be simulated for more realism. The magnetic compass provides a bearing to north which is accurate at best to a couple of degrees since the magnetic field of the Earth can be affected by the vehicle itself and also the surrounding environment. Therefore slowly changing noise should be able to be specified for the magnetic compass. Roll and pitch sensors are usually part of the same equipment, but they generally only operate within a +-30Degree range. In order to keep the components generic, a special adapter module will be able to be attached to the generic compass to simulate more specifically the correct behaviour of the real one, i.e. remove erroneous measurements which are out of range for each particular type of compass axis and introduce appropriate errors.

- GYRO - The GYRO is a device which measures rotation about the X,Y,Z axis. However, it has no concept of which direction is north and only reports how much the GYRO's orientation has changed, the delta orientation. This device is also prone to drift over a matter of hours due to the rotation of the Earth. This has to be compensated for. The navigation system would have to take account

of this and calibrate the GYRO from the magnetic compass. The GYRO is not required for the standard DP system of STOS. However, the PAIV Vehicle discussed later will require a GYRO because of the need for a more accurate navigation system.

- Depth sensor (survey string from ROV) - ROVs usually have a very basic set of sensors: a compass and a depth sensor. These sensors can be harnessed by STOS. The depth sensor is simply a pressure sensor which provides the depth below sea level. The altitude can be calculated from the DVL measurements. So the DVL also doubles as an altimeter.

- Hydrodynamic model - the joystick of an ROV outputs an (X,Y,Z) thruster vector. The vehicle model would take these percentages and turn them into a movement of the vehicle. This would best be a black box style implementation allowing the user to input the certain characteristics of the vehicle, such as acceleration, terminal velocity. Sometimes real thrusters have "dead zones" which are basically a percentage thrust which makes no difference to the actual thruster. These will still have to be collected when tuning the real vehicle as this information would not be easily measured by an ROV pilot. Collecting information about accelerations etc allows the STOS engineer to do a partial soft-tune using the simulated vehicle before requiring the real vehicle. This can save time when deploying the ROV as a basic control will be already available.

- Bathymetric data loader - this is basically a terrain data loader and there are many of these already in existence, so it would not be necessary to write one from scratch, but instead convert an existing one into a JavaBean ready for incorporating in ARF. ARF already provides loaders for DXF, X3D, 3DS mesh formats, so really does not need creating. The terrain loaded would be used by the altimeter to measure altitude of vehicle.

- Ocean currents - simulation of ocean currents is vital for testing the DP control of the vehicle thoroughly. Thus a data loader must be written for loading existing current data provided by nautical navigation charts; this could be 2D

or 3D data. The vehicle model will also have to take this data into account if it exists.

- Thermoclines and haloclines - both of these will affect the buoyancy of the vehicle and thus should be taken into account by the vehicle model. These can be implemented in a similar data map as the ocean currents are.

These components will allow for the fast creation of many scenarios speeding up development time for new modules. Scenarios can be created to provide training on how to use STOS, and provide more thorough testing which will in turn save time. Whilst STOS was being developed it took 7days of ROV testing to iron out most of the bugs in the software. Most of the bugs were user interface problems and could have been rectified with simulation testing using ARF and potentially saved thousands of pounds. However, ARF has been applied in on going testing since and is helping to test changes and improvements made to STOS. Another application of ARF was used in the SeeByte's PAIV project in section 7.9.

7.9 SeeByte Ltd: PAIV AUV Developmental Testing

PAIV is a new vehicle being designed by SeeByte and Subsea7. The purpose of this vehicle is to carry out the tasks that SeeTrack Offshore is being developed for, including: inspection, such as pipeline inspections, ship hull inspections etc. The software and sensors on the AUV are being developed by SeeByte, whilst Subsea7 are building the actual vehicle. The vehicle will be an ROV to begin with. Its intelligence will reside on the topside computers, similar to SeeTrack Offshore, and as the software becomes de-risked it shall be moved onto computers embedded on PAIV itself. Eventually the tether will be disconnected and it will be able to run as an AUV performing autonomous tasks as well as being used as an ROV. The main drive for PAIV is to be able to have an inspection AUV of relatively light weight which can go to 3000m depth and carry out autonomous inspection tasks.

The PAIV high level control systems will benefit from being able to be tested on

a simulated vehicle. There are various stages of development with small milestones which need to be achieved. Each milestone will require more parts of the PAIV vehicle to be simulated. Thus as the systems on PAIV evolve, then new simulated components will have to be created for use in ARF. The first milestone for PAIV is to test the control software and user interface control suite of the vehicle. This includes the end-user GUI systems and also the low level autopilot PID controller which will control the vehicles thrusters. In order to test these systems the PAIV simulator requires a hydrodynamic vehicle model which can use as input the OceanSHELL ALI Thruster messages and output an ALI navigation message. For this test the navigation system is being fully simulated by the vehicle hydrodynamic model as only the control of the vehicle is being tested. In later testing it might be necessary to test the real navigation systems with input from virtual sensors, such as DVL, GPS and compass.

7.9.1 1st Milestone: FAT (Factory Acceptance Tests)

In order to show that the development of PAIV was progressing SeeByte was required to demo the capabilities of the control software. This was do demonstrate that the AUV's autopilot was capable of basic movement. This was also to demonstrate the user interface for controlling an ALI enabled vehicle. Since ARF had recently implemented the necessary ALI communications interfaces, the PAIV control software did not require any extra integration. However, the ALI layer only took 4 hours to create the necessary ARF components for, which if taken to be part of the PAIV project is still a very small overhead to use ARF.

7.9.1.1 Requirements

The requirements were to use the existing hydrodynamic model provided by ARF and simulate the AUV with all the necessary OceanSHELL ALI interfaces in place to communicate with the PAIV control suite. This will allow the developers to run ARF and the PAIV scenario on their own computers whilst testing the higher level user interface systems and check that PAIV does the correct thing. The ARF components required for PAIV's scenario are:

- Hydrodynamic model with altimeter sensor input - normally, the hydrodynamic model only outputs depth. However, the autopilot requires an altitude input therefore a simulated altimeter is used.
- Autopilot (PAIV's autopilot communicating externally via OceanSHELL with ARF).
- OceanShell ALI Message I/O:
  - Hydrodynamic model to ALI *NavigationStatus* message output.
  - Hydrodynamic model ALI *AxisForce Thruster* message input - Requires, XYZ force to rauver thruster converter.
  - Autopilot ALI *NavigationGoalCommand* message input.
  - Autopilot to *NavigationGoalStatus* message output.
- Visual Scenery (VRML and X3D) - pipes, oil field christmas trees, bathymetry, water level and PAIV 3D mesh.
- Bread crumb trail behind the vehicle to show where it has been.

The ALI message components are described in more detail in section 4.1.5.1. All the rest of the components already exist from other projects. However, ALI component creation times are shown in table 7.4.

This first test is so that SeeByte can demonstrate that their higher level software works, as the real PAIV vehicle is not yet ready for testing. This test will show that they are on target for completion of PAIV development. Therefore this test is merely showing the virtual AUV in its environment and what it is doing. Therefore this is also being used as a visualisation tool, and thus the scenery will be decorated accordingly (see figure 7.25).

7.9.1.2 Results

The ARF framework proved most fruitful in the run up to the FAT tests. ARF allowed the high level PAIV control GUI to be tested with ARF's simulated vehicle and the Rauver Autopilot as soon as ARF's ALI message listeners and outputs had

been created. This was very useful because the real Vehicle Systems had not yet been converted to use the new ALI messages in time for testing the GUI. This meant that many bugs with the GUI were identified well before the GUI integration with the real components. ARF allowed the programmer of the GUI to see the output of their actions, and this highlighted many small bugs which would have been hard to spot had visual feedback not been available.

Problems identified with PAIV GUI by using ARF:

- OceanSHELL depth mode bit set incorrectly - Firstly this meant that the vehicle would be in "set altitude" mode instead of "set depth" mode. Secondly, sometimes the depth mode value would get set incorrectly on successive calls by the GUI. This was identified quickly and the problem fixed.

- After successive waypoint requests the GUI would stop accepting negative "east" values for the *(north, east, down)* waypoint. This was a problem due to GUI fields being reformatted incorrectly, which only materialised after many tests.

ARF proved very useful for visualising simple AUV missions around the scenarios. It allowed the assessors of PAIV to see exactly what they would be able to do with the real AUV and allowed them to easily evaluate the PAIV systems for the FAT tests. However, by using ARF an extra overhead was required for the creation of the ARF components and the building of the ARF testing scenario. A breakdown of the time requirements to use ARF are displayed in table 7.4. The additional overhead required to use ARF has to be weighed up against the amount of time saved by being able to use ARF. The total time spent integrating ARF was 5 hours 10 mins. However, this minimal expense will be recovered through every step of testing. In addition to this is the qualitative results that ARF helped achieve, i.e. if ARF had not been used it is clear that the PAIV GUI would not have been able to be tested because the PAIV embedded modules weren't finished until the last minute. Therefore all the errors with the GUI, and the ALI messages, would have shown up during the FAT tests and be severely detrimental to the performance during these tests. ARF was able to produce real-time software-in-the-loop (SIL) testing whilst demonstrating far more complicated AUV missions than would be possible without ARF. Also, the

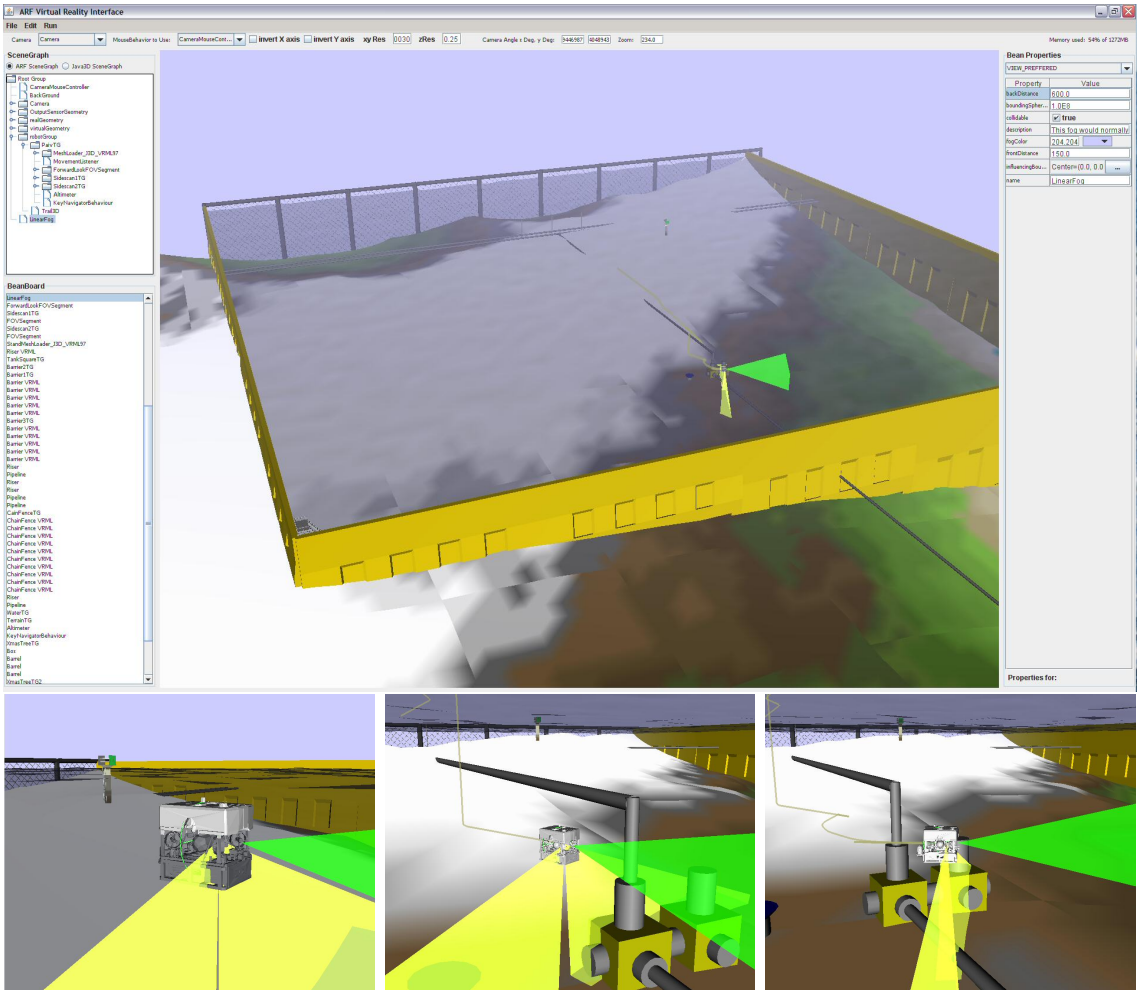


Figure 7.25: PAIV Oil Field Scenario 1: for simulating pipeline installation inspection tasks

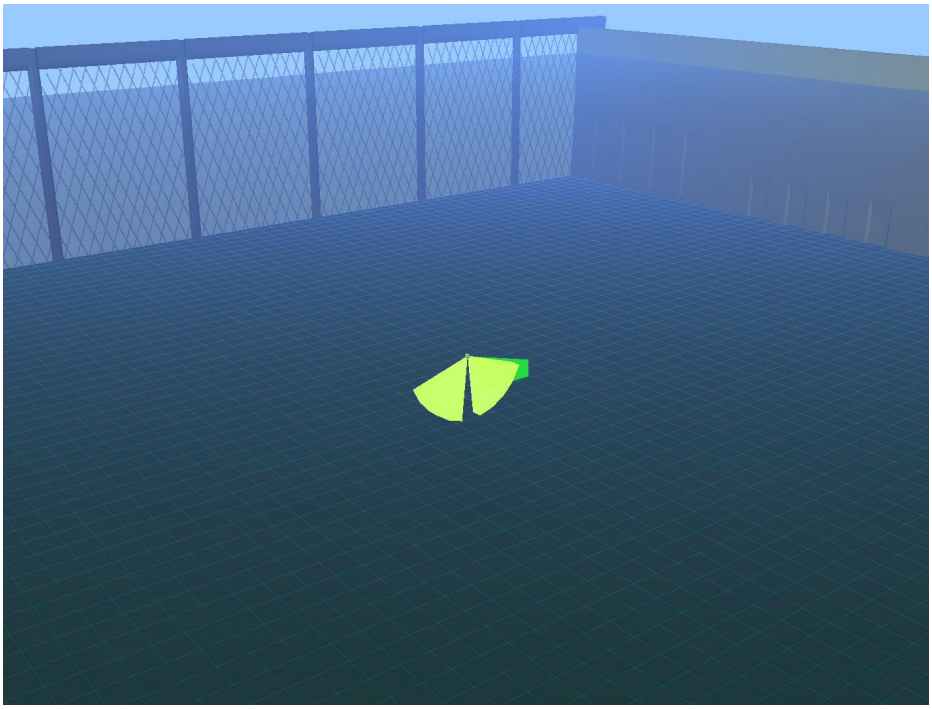


Figure 7.26: PAIV Big Tank Scenario 2

components created for ALI have far more applications other than that of PAIV and as such, due to ARF's guided construction, are used time and time again in different projects. If the testing of the GUI was delayed until real world tests of the AUV, even if the real world tests only took the same 5hours 10minutes as ARF, the costs alone of using the real ROV to do this test would be over 1000 GBP just for one day. This would also be the only testing available and programmers would have to make do with their own tests. Therefore ARF is definitely worth the small overhead due to its phenomenal time savings and cost difference.

The finished scenarios are shown in figures 7.25 and 7.26; these show the oil field virtual world and the big tank virtual world respectively. The large empty tank was used to demonstrate simple tests of the higher level control as the oil field scenario was used to demonstrate a mission which the final PAIV vehicle should be able to do autonomously. This would include autonomous pipeline, riser and installation inspection.

7.9.1.3 Problems detected during testing

The CCMM (mission script executor) hadn't been configured to use the new ALI *NavigationCommandStatus* message. However, the CCMM module did still accept the old *RauverInPosition* message which ARF is also able to output, so ARF output both message types as the higher level GUI being tested required the ALI message. This highlights the advantage of using event listeners, since more than one event listener can listen to the output of the hydrodynamic model and output different messages. ARF allows for the easy addition of event listeners on the fly, so this was a quick fix which enabled the GUI to be tested when the low level systems were not quite finished. This would not have been very straight forwards had ARF not been so flexible.

A bug in the altimeter component of ARF was highlighted which meant that it couldn't detected the ground past 10m. This was a simple problem and has since been rectified. A bug in the AUV model after many successive rotations meant that it output a *nan* (not a number). This needs fixing as it is probably a problem with a rotation value being un-capped allowing it to increase past 360 degrees. These bugs highlight the fact that even ARF components need thorough testing.

7.9.2 PAIV Integration Tests

After the initial trials of the software the real vehicle became available for adding sensors, such as DVL, forward-looking sonar, and fibre GYRO compass. The software was then due to be tested on PAIV in SubSea7's test tank in Aberdeen. In order to test the vehicle they required a virtual mock up of the tank so that they could visually check that the vehicle was doing the same in the virtual world as they could see it doing in the real world. This virtual scenario was also used to show what the AUV was actually doing and where it thought it was. Figure 7.27 shows the mock-up scenario in ARF. Table 7.4 gives creation times for the components and scenarios required.

For this step, all ARF had to do was listen for navigation data and waypoint request messages from the real AUV. This was used to simply display what the AUV was doing. This used the ALI OceanSHELL messages and therefore no additional

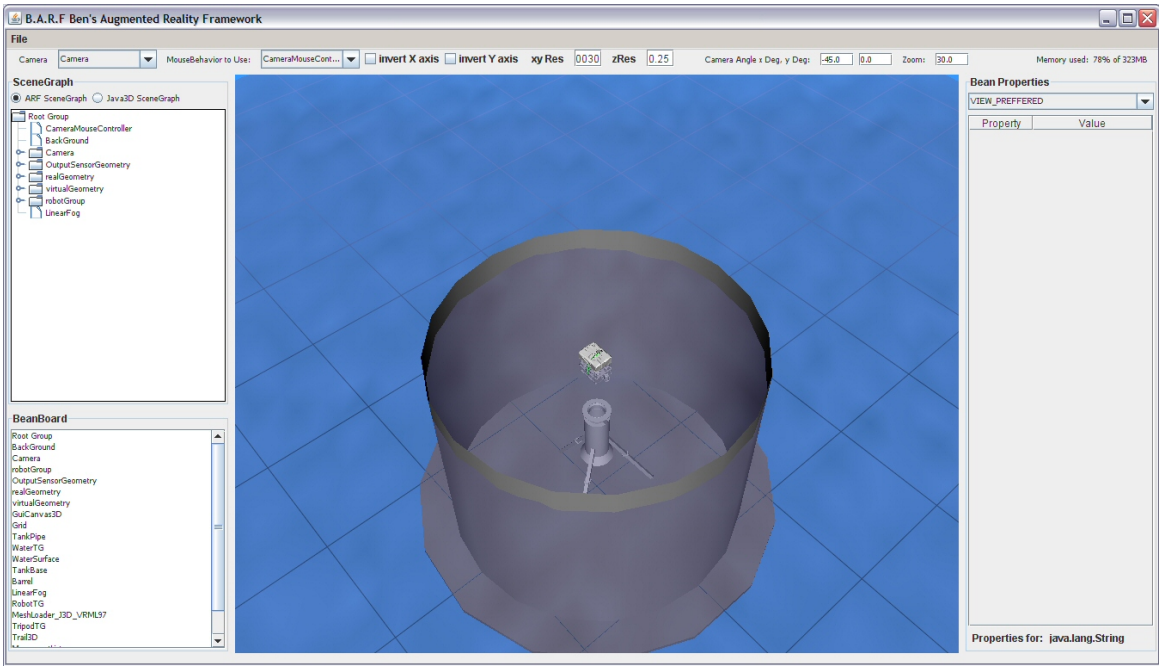


Figure 7.27: PAIV Test Scenario in Subsea7 Aberdeen

components were needed to be created for ARF.

7.9.2.1 New ARF Components

All of the STOS components are also required by a full PAIV vehicle test-bed (see section 7.8). This is but one example of how ARF can be used for different applications using the same components. Consequently, ARF scenario creation time can be drastically reduced by modifying existing ARF scenarios and using SuperComponents.

- GPS simulator - the GPS simulator must simply output the position of the vehicle in Longitude, Latitude, Depth/Altitude. However, a real GPS has errors, so a slowly oscillating error similar to the one produced by a normal GPS should be added.

- Altimeter - although a DVL can estimate altitude by taking the average of its 4 beams, this is not the real altitude exactly below the vehicle. Therefore a simple altimeter is required to output how far the vehicle is off the simulated tank bottom.

Task	Time (HH,MM,SS)
ALI Message I/O	4,00,00
Oilfield Scenario Design	1,00,00
Open Tank Scenario Design	0,10,00
Subsea7 tank Scenario Design	0,17,00

Table 7.4: PAIV Time Analysis: Scenario and Component Creation

7.9.3 Time Analysis

Table 7.4 shows the time it took to make the various PAIV components and scenarios. In particular attention should be paid to the amount of time it takes to create a scenario when all the required components are available. This is comparable to the time comparisons reported in the ARF performance tests in results section 6. If ARF did not exist but all the components did, it would have taken much longer to connect and configure all the components in raw programming language. Any alteration in ARF takes seconds, compared with finding documentation for required classes, programming it, compiling it, and then running it, and then hoping that the right thing has been adjusted and further alterations do not have to be made. The oil field scenario took longer to create due to the amount of scenery required.

7.9.4 Summary

ARF proved very useful for testing PAIV's high level systems in a SIL fashion by providing a simulated version of the real AUV. ARF also proved very useful for online monitoring of real trials whereby the operators used it to visually confirm that the robot thought it was doing the same thing that it was actually doing. Since these original trials further systems have been tested using ARF's PAIV simulator, such as: a wall and object tracking system used to make the AUV follow a ship hull or similar shape. A simple sensor was required by ARF which mimicked the behaviour of wall detection performed on sonar images. This enabled the higher level tracking algorithms and vehicle control to be tested. Figure 7.28 shows the PAIV vehicle and the associated breadcrum trail as the vehicle tracked the wall of a conical tank (made

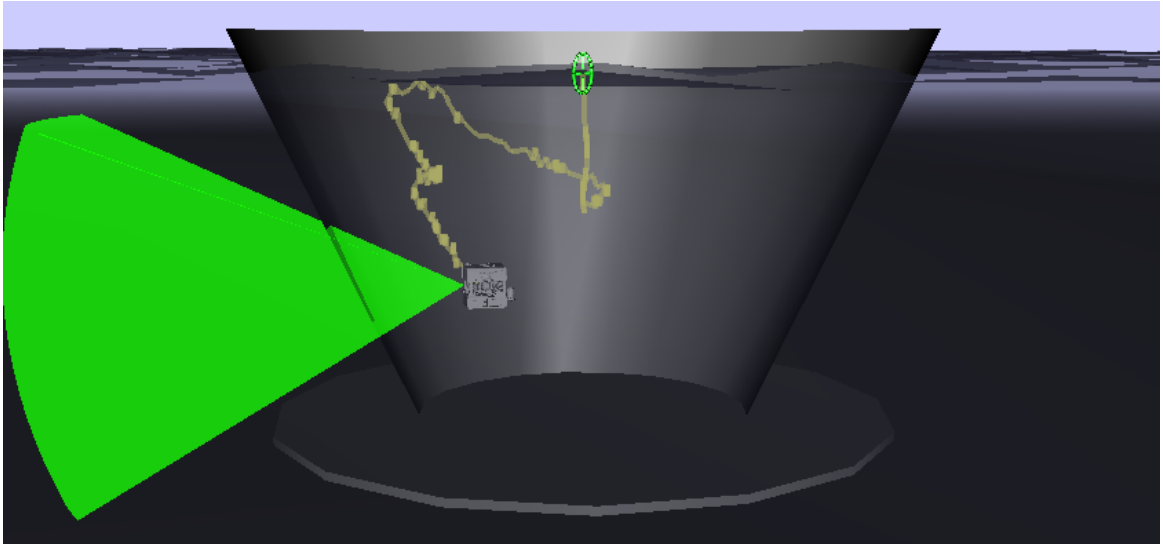


Figure 7.28: PAIV AUV tracking the walls of a conical test tank

of ebony).

7.10 Future Opportunities

This section looks at some opportunities for ARF which have been demonstrated but not fully implemented.

7.10.1 DELPHÍS: CopeHil Down Scenario

Other applications of the DELPHÍS multi-agent architecture have been demonstrated using ARF. These include a potential scenario for the MOD Grand Challenge. This involves both surface and air vehicles working together to find targets, in a village, and inspect them. DELPHÍS was tested using simulated air vehicles, rather than underwater vehicles, which executed a search, classify and inspection task. ARF provided the virtual environment, vehicle simulation and object detection sensors required for the identification of potential threats in the scenario. Figure 7.29 displays the virtual environment view of the Grand Challenge scenario. The top of the screen shows a bird's eye observation of the area with the bottom left and right views following the survey class Unmanned Aerial Vehicle (UAV) and inspection class UAV respectively. The red circles represent targets which the survey class UAV will detect upon coming

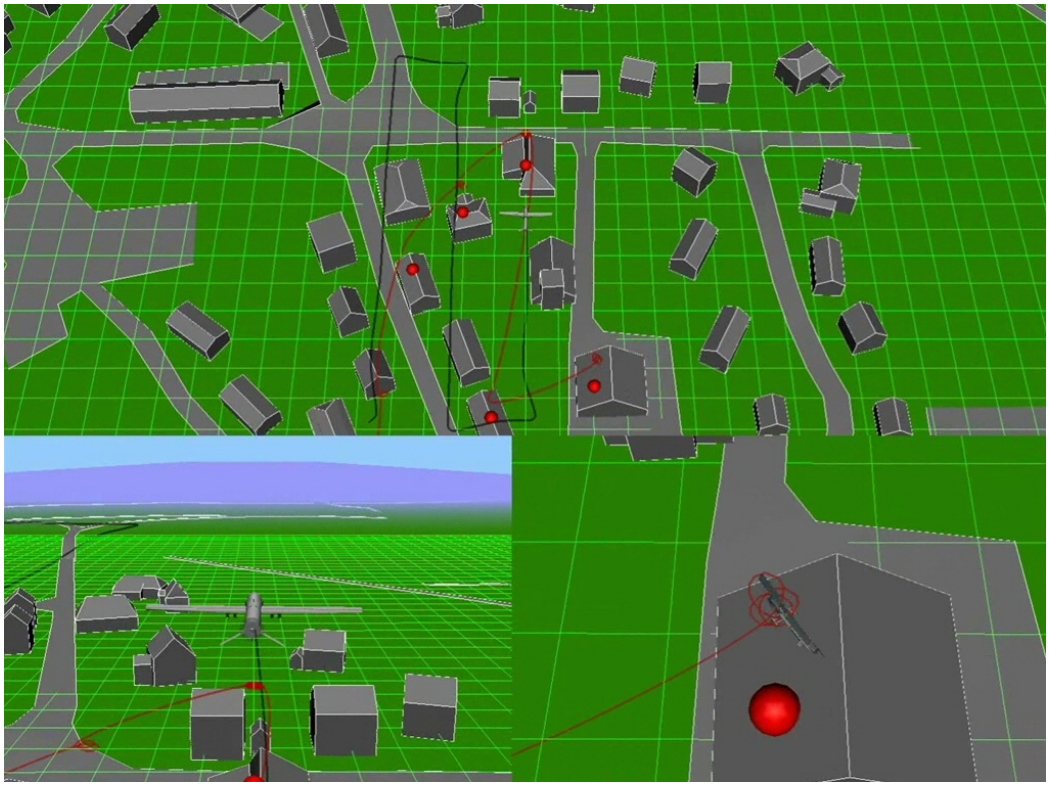


Figure 7.29: DELPHÍS: Being used to Coordinate UAVs

within range of the UAV's object sensor. Information regarding specific targets of interest is shared between agents utilising the DELPHÍS system. Vehicles with the required capabilities can opt to further investigate the detected objects and reserve that task from being executed by another agent. The DELPHÍS system executes this quicker than a single AUV since each AUV agent does a different part of the mission. A lot of missions in different domains can be categorised as the search classify scenario. The DELPHÍS system can be applied to any type of robot as long as they have communications. In the case of UAVs their communication mechanism will probably by radio, however, this is the only difference between DELPHÍS operating on an UAV rather than an AUV.

This scenario was merely a demonstration of possible applications of technologies in the OSL, however, it does demonstrate that ARF can be used for any domain and only requires that special domain specific ARF components be created where they are needed. Therefore ARF could easily be used for simulating multiple types of autonomous vehicle (UAV, AUV, ASV) all collaborating by different communication

methods using the DELPHÍS system to coordinate them.

7.10.2 SLAM Testing

In order to test Simultaneous Localisation and Mapping (SLAM) systems, ARF can be used to simulate basic object detection which outputs the relative position of an object and its bounding sphere size. In addition the simulation can be made more complex to eventually simulate video cameras, sidescan or forward looking sonar. The object detection systems themselves can then be trained to interpret this data and the output from which can then be fed into the SLAM system for helping reduce drift in the vehicle's navigation system.

7.10.3 Mine Counter Measures using Sidescan Sonar

One of the problems the Navy have is underwater mines. The Navy need to monitor and keep an area of sea clear of mines to keep shipping lanes open and to enable themselves to operate. AUVs are very useful for this task as they can survey an area looking for mines. The data can then be analysed on return home, or the data can be analysed on the fly and a counter measure deployed autonomously. However, training the systems to recognise the mines and check that identifications are correct can be hard using sidescan sonar since a mine can look different from many angles. Thus ARF can be used to see what different mine shapes look like in side scan sonar from different angles and therefore be used to train the system. The data from this system can also be used for the SLAM system.

Chapter 8

General Discussion

This chapter discusses the information obtained from the use cases and testing chapters to highlight the features of ARF, identified in section 3.1.3, which make it more efficient for the purposes of hardware-in-the-loop testing and other augmented reality applications. To begin with this chapter will look at the different use cases described in chapter 7 and correlate the results with data gathered in the chapter 6 to explain the advantages of ARF. It is important to keep in mind the design requirements for ARF discussed in chapter 3.1.3, since this discussion will attempt to provide the evidence for each of the requirements being met and what features of ARF's implementation provide them. It is therefore important that the reader be familiar with chapters 3, 4, 5, 6 and 7 before hand as many concepts will not be explained here.

8.1 ARF's Uses

The use cases, described in chapter 7, demonstrate the flexibility of ARF as a visual programming tool. They also exploit the augmented reality and distributed nature ARF by demonstrating various types of mixed reality environments. ARF has been utilised in many different ways. Most of these focus on AUVs or UUVs, however, there have been other uses such as the MOD Grand Challenge demo that show that not only ARF, but many other technologies in the OSL are adaptable to different domains other than underwater.

The Nessie II and Nessie III platforms required extensive testing scenarios for

testing the higher level software onboard the AUV. Both Nessie II and Nessie III had to complete a series of complex tasks in the SAUC-E competition. The tasks set out by the SAUC-E competition required the AUV to have autonomous decision making skills in order to assess the current information known about the environment and make an informed decision as to what to do next. Nessie II and III required extensive testing of the mission planner to evaluate search routines, task priorities and specific functions of the AUV, e.g. dropping the drop weights on the drop target having already aligned itself accordingly above the target. ARF became incredibly useful in evaluating and developing the mission planner because it enabled the programmer to continually create new behaviours and asses their performance and visually see how the algorithms controlled the AUV. For ARF to provide all the necessary inputs to the mission planner, various sensors required simulating such as sonar, video, altimeter, IMU etc. Most of these sensors could be simplified since what the mission planner was actually requiring was navigation information and object detections from the detection algorithms run on the sonar and video images. In ARF the object detection algorithms and sonar and video simulation were replaced by simple object detectors which output the detected objects in the same coordinate space as the real systems would. ARF required extending and due to its JavaBeans architecture this was easily done. New ARF components were created for exchanging OceanSHELL messages with systems running on the actual AUV and the simulated sensors. The extendibility of ARF lends itself to the powerful nature of JavaBeans, since JavaBeans were designed to make building programs far easier. Thus ARF reaps all the benefits.

Most of the use cases discussed in chapter 7 required ARF's component library to be extended due to ARF's relative immaturity and therefore only having a limited component base for underwater vehicle development at the time. However, later projects, such as PAIV and DELPHÍS required little or no component writing for ARF due to the component repository growing so rapidly. Many robot configurations created for one usage could be used time and time again through the use of SuperComponents. The OceanSHELL ALI interface has provided a standard for AUV systems communications, which ARF has been able to harness to its full extent since testing scenarios which work on one type of AUV can now be used to test another providing

it adheres to the ALI interface.

8.2 Flexibility

It is important to remember the evidence which backs up why ARF is so flexible and useful for different applications. The user interface provided by ARF includes features, such as Guided Construction and SuperComponent creation, which allow the user to create scenarios with little or no experience of the components they are using. The users require a starting point that from which it is merely a matter of connecting the components using ARF's guided construction mechanisms. The ability to save many different configurations quickly and easily without having to manually configure them by hand saves so much time. As such it is not easy for the user to make mistakes in configuration since the ARF GUI will not let them. Configuration files are always written correctly first time, in contrast to programming manually where compilation errors, null pointer exceptions and the usual problems of programming from scratch occur.

The APTs highlight the time savings of using ARF to make small alterations and create different configurations. The APTs show an average increase of double the performance when using ARF. However, this is due to the fact that creating a clone of a project in NetBeans with one parameter different can be done by copying and pasting. However, the more properties that are changed in between saves, the higher the performance of ARF, because in this test it was the save time overhead which made ARF only twice as fast as Netbeans. In ARF the PropertySheet increases the speed of making alterations since they can be changed instantly and, more importantly, on the fly. A property changed in source code, when programming by hand, means the code has to be recompiled and then run again, in ARF this overhead does not exist. Mistakes, such as syntax errors, can be made when programming by hand, this cannot happen using ARF.

The use cases demonstrate the flexibility of the underlying communication architecture of ARF since all the different components required to do external communication and sensor simulation were easily created. ARF is as flexible as JavaBeans for

what can be done with it. ARF's primary goal was to provide a framework that could be used to create testing scenarios quickly and easily for doing mixed reality testing of remote platforms such as AUVs. This is just one potential application of what can be achieved using ARF's JavaBeans backbone, as JavaBeans can be used for just about anything. However, the ARF user interface favours virtual reality, since this is its prime purpose, but there is no need for the user to stick to this. ARF allows for the removal of the Java3D Canvas from the main view of ARF and be replaced by any other graphical JavaBean.

8.2.1 Distributed Communication and Location Transparency

JavaBeans are inherently a non-distributed architecture. However, ARF provides special JavaBeans which implement OceanSHELL functionality which allow communication between different ARF environments running independently on different platforms and also allows communication with any other system implementing OceanSHELL functionality. This allows ARF to be distributed over many computers if a computational task is too expensive to run on one machine alone. In addition, the JavaBean event passing mechanisms can be coupled to ARF's OceanSHELL message queues to input/output event data to/from OceanSHELL messages. All the use cases rely on OceanSHELL to provide the location transparency needed for their various applications. Nessie II and III used OceanSHELL for visualisation of data from the AUV, to output simulated sensor data to the AUV's SLAM and Mission planning systems, and to input and output data from the AUV's autopilot to simulated thrusters and hydrodynamic models running in ARF. This was typically HIL testing.

The PAIV trials relied on ARF to provide visual feedback about AUV position to keep the operator informed. ARF was also used to test the PAIV user interface. ARF provided the simulated vehicle systems, such as hydrodynamic model, altimeter and compass. The PAIV GUIs were tested in a software-in-the-loop manner. The AUTOTRACKER scenario was used to test the AUTOTRACKER vehicle systems in the same software-in-the-loop manner since ARF was used to simulate the vehicle and the control systems whilst simulating side scan sonar data of virtual pipelines, and outputting this to the AUTOTRACKER pipe detection and control software.

The BAUUV scenario went one step further and used the real vehicle in the real environment in conjunction with some simulated sensor data from ARF's virtual environment. This was typically a Hybrid Simulation setup, however, the same scenario was also used with a simulated AUV before hand to check that this worked before real world tests commenced, i.e. PS and HS were both used for BAUUV. These examples demonstrate the mixed reality nature of ARF. ARF's interface to OceanSHELL provides the straightforward connection to remote platforms required for HIL and HS testing and for Online Monitoring (OM). ARF allows for instant switching of OceanSHELL data flows so that real systems can quickly be substituted for virtual systems. Each scenario can be easily saved, cloned and broken down into component parts which can then be imported into new more complex scenarios. ARF provides a ground up approach to designing complex mixed reality scenarios; each part can be created and tested separately and then turned into SuperComponents which can then be used to create more complex scenarios performing many tasks at once for a potentially limitless number of platforms.

These different uses demonstrate just one field of potential applications of ARF. It is ARF's ability to save/load projects quickly and easily, and create SuperComponents, that provide the flexibility and modularity. With JavaBeans supporting this architecture at the low level, it lends itself to be used for very complex scenarios which could not have been easily developed before because the individual chunks were not manageable. ARF can therefore evolve at the same rate as the projects which require its capabilities.

8.3 Extendibility

The scenarios discussed in the use cases in chapter 7 required for components to be created as JavaBeans and added to ARF's repository so that the user could create the scenario required. Creation of new components will always be needed, since new applications, systems and sensors will always be emerging. Creating components for ARF is as simple as it could be since only the JavaBean programming conventions need to be adhered to. The actual time it takes to produce a component is always

going to be the same time it takes to program that component for any environment, possibly a little bit quicker in ARF because no special language is required, and there are a lot of other components which can be extended or connected to. The advantage of ARF is that once the component has been created it can be used many times in different scenarios for different uses. The programmer also only has to focus on creating the functionality of their component, and doesn't need to worry about modifying the existing source code of ARF or having to resort to creating their own virtual environment every time. Once the component has been created, ARF provides help via guided construction of how to use it and configure it, so that it is easily used by others as well.

One of the ways ARF reduces scenario creation time is by reducing the amount of time required to glue the components together. In other architectures, components end up integrated into monolithic programs and become very hard to extract later, if they need to be used for something else. The extra time the programmer saves by not having to re-program the environment every time is better used creating more generic and flexible components, which can be used time and time again. High code reuse incurs huge time savings.

The advantage of ARF is that existing modules can be easily ported to become ARF components. If modules are already written in Java then the task is simply to make sure that the properties of that class, which should be modifiable by the user, are programmed adhering to the JavaBean coding conventions. If a module already exists and is programmed in a language other than Java then the programmer has two choices: if they wish for their module to be configured by ARF and easily used then it will require porting to Java which is not difficult if the original was programmed in C/C++, or, if the programmer simply wants their module to interact with other ARF modules they can simply create a simple stub component for ARF which uses a communication protocol, such as OceanSHELL, to communicate data with the remote module. This is the case for hardware-in-the-loop testing, since ARF has to input/output data to/from external modules. The autopilot testing of Nessie III is one such example, because data from the simulated hydrodynamic model was output to the real autopilot running externally, via OceanSHELL messages, whilst

thruster messages were input to the hydrodynamic model from the autopilot. Two ARF components were required to input thruster messages and to output navigation messages. Navigation messages were generated from the navigation event data output by the hydrodynamic model in response to the thruster control messages from the external autopilot.

One such component which is used time and time again, and for this reason is very useful to have as a JavaBean, is the hydrodynamic model. This is used to simulate the movement of underwater vehicles in response to thruster control messages. This component is so useful in day to day testing of AUV software, that having it allows for new modules, which require basic vehicle systems, to be tested very quickly and easily. If the component was simply a software module which runs separately (not in ARF), the programmer would have to program the interface manually every time they wished to use the component for a different purpose. This is where ARF provides far superior capabilities as it provides rapid creation of scenarios and encourages code reuse via SuperComponents and extending of existing scenarios. The APTs highlight the performance increase of using ARF to connect components over that of programming manually. In addition ARF accelerates code reuse by providing the ability to create modules, known as SuperComponents, within ARF. This provides varying levels of component complexity and allows for useful configurations to be saved and reused.

8.4 Scenario Creation

The Nessie, PAIV, DELPHÍS and RAUBoat examples provide information on component creation and scenario design times. Tables 7.1, 7.2, 7.4 and 7.3 show the respective times taken for these examples. The component creation times are the same for ARF as any other standalone module, so this is not an overhead specific to ARF, but is instead required for any testing software. In theory it should be quicker to program components for ARF since the programmer doesn't have to worry about programming special languages to allow the module to be used in visual programming. Scenario creation times using ARF tend always to be small because ARF allows the

components to be connected and configured very quickly and easily through its Guided Construction and PropertySheets. These times are corroborated by the APT exercises for 3D virtual world creation and component connection, which compare ARF against standard programming (see figures 6.9 and 6.1). One point to mention about the scenario creation times, demonstrated by the use cases, is that they are all very small. This is because if all the components are available, scenarios can be created so very quickly due to the GuidedConstruction provided and inherent code re-use. This is what makes ARF far more useful than just a configurable virtual environment, parts can be re-used for many purposes which may not have even been thought of yet, but ARF will provide the help to configure them when the time comes.

8.5 Summary

ARF enables the user to create scenarios quicker and easier than before. ARF provides help for the user via Guided Construction as to which types of components can be connected to others. An analogy which provides similar functionality to ARF is that of LEGO® (described in section 1.4.1). The importance of this example is that LEGO® provides the user with visual queues which enable the user to fit the LEGO® pieces together. This is what ARF tries to provide by helping the user know which components can be connected. LEGO® speeds up structure development by providing simple to use building blocks which can be quickly fitted together. This is inherently quicker than the user having to either mould an entire structure out of LEGO®, or create the building blocks from scratch. ARF provides some components and helps the user fit them together, in contrast to programming a testing platform from scratch where all the code is in one monolithic program which is very hard to alter or extend. The contrasting analogy to this is that if LEGO® bricks were glued together, it makes it very difficult to change the design to do something else, or modify the design. Thus ARF helps the user construct scenarios which can be easily rearranged and reconfigured. The proof here is that by providing an intuitive interface, LEGO® makes configurations of components easier to achieve. Therefore, since ARF provides intuitive user interfaces for connecting components, it should

Requirement	ID	Met how?
Generic	1	Implied in design using JavaBeans
Extendable	2	Use Cases and APTs
Flexible	3	Inherent through Use Cases applications
Distributed	4	Use of OceanSHELL demonstrated in use cases
Guided	5	Demonstrated through APTs
Intuitive	6	Demonstrated through APTs
Straightforward	7	APTs and implied in Use Cases
Portability	8	Inherent in Java and use of OceanSHELL
Varying granularity	9	JavaBeans, Projects and SuperComponents in APTs
Polymorphism	10	Use of Event Listeners in Use Cases
Generic communication	11	Use Cases: Event Listeners using OceanSHELL

Table 8.1: Summary of how requirements from section 3.1.3 are met.

make constructing scenarios out of those components easier to achieve, since it is using a similar technique to LEGO® but through a slightly different visual interface. This is also corroborated by the APTs which highlight how much ARF improves over the standard methods.

ARF demonstrates the functional requirements set out it chapter 3 through the use of APTs and use cases. It also demonstrates that it is more than capable of supporting the range of mixed reality and testing application working modes discussed in the literature review, i.e. OFFLINE, ONLINE, HIL, MHIL, SIL, HS, TR, AR, AV and VR. Examples of each case are demonstrated in the use cases chapter 7. A summary of how each requirement is met and how they are demonstrated is given in table 8.1. In essence most of the requirements are met by the technologies used by the design. However, ARF's applicability to specific usages, such HIL and HS, are inherently demonstrated through the use of the use cases.

Chapter 9

Conclusions and Recommendations

The discussion in chapter 8 describes how ARF meets the requirements set out in section 3.1.3. The use cases illustrate some of the potential uses of ARF. However, in order to really show that ARF is a better architecture for developing testing environments for remote platforms, performance testing of the features of ARF was required. These tests illustrated that ARF was at least twice as fast as compared to other methods. This figure is based on amateurs in ARF and experts in the rival. This lends itself to how intuitive ARF is, the ease of using it, and how flexible it is. The use cases demonstrated how ARF could be easily extended via its JavaBean architecture to accommodate many applications.

In a nutshell ARF provides a generic, extendable and flexible architecture which lends itself to distributed applications for use in augmented reality applications such as testing of remote platforms. Furthermore the intuitive user interface that ARF provides through mechanisms such as *Guided Construction* allows for scenarios to be developed for many applications quickly and easily in a way not too dissimilar to the intuitive interface LEGO® provides.

ARF is straightforward to develop due to its flexible JavaBean based architecture. Varying levels of granularity are available for development of ARF on the high end user level and the low programming level. It is the layers of granularity which will allow ARF to grow to produce ever more and more complex scenarios, one building on the work of the previous. As applications get more complicated ARF is able to expand its component set and capabilities at the same rate allowing for very complex

scenarios to be created which demonstrate many different concepts simultaneously. Underwater applications are but just one application of a framework of its nature. ARF itself doesn't impose any limitations on what it can be used for, but instead provides guidance for the user to help them achieve what they are trying to create. Concepts such as HIL, HS, OM, TR, MHIL, AR and AV are all definitions of related technologies; ARF allows those technologies to be connected to provide ever more powerful uses.

One of the main problems ARF set out to solve was improving pre-real-world testing facilities for remote platforms. ARF provides the ability to evolve its capabilities concurrently with the evolution of the application ARF is required to work with. Providing simple internal communications means that ARF can be easily extended to interact with remote systems on varying communication protocols. In essence, ARF allows for entire scenarios to be created and controlled within one program, and provides facilities for on the fly reconfiguration of those scenarios enabling rapid prototyping for many different usages. Again, this lends itself to the analogy of LEGO®.

The main goal ARF set out to achieve was improving pre-real-world testing facilities such as HIL. ARF does this by enabling different types of testing scenarios, such as HS, PS, OM, to be created using the same environment through the rearrangement and reconfiguration of components. ARF provides an object oriented and highly modular approach to building testing harnesses for robotic platforms. This stops the stove pipe development of simulations which have one usage and are consequently hard to alter for similar but slightly different usage. ARF provides extra help to the user when creating the scenarios which leads to a performance increase in scenario creation time as shown in the performance tests. ARF provides all the low level requirements necessary for HIL and other testing types and augmented reality applications (as listed in section 3.1.3). However, although these facilities are provided the whole problem specific to HIL testing is not solved since ARF provides all the base technologies such as distributed communications and 3D virtual environment but does not provide every conceivable component for all types of HIL testing since most would be application specific. Instead ARF provides a set of HIL testing components which have evolved through the usage of ARF for many HIL, PS, HS,

OM, TR applications demonstrated in chapter 7. These examples show how ARF is used to provide many different testing and augmented reality scenarios using the same subset of components. However, if a component does not exist ARF provides the ability for the user to incorporate their own.

ARF's component repository has no limits; it can grow and grow forever providing ARF with limitless functionality. As the repository expands, fewer new components will need creating for ARF, making creation of scenarios, for whatever purpose, faster and more efficient.

ARF is particularly suited to HIL testing purposes due to its existing mixed reality components, but also has immediate applications in simulation, remote awareness and operator training as demonstrated in the use cases. However, ARF can also be used as a standard JavaBean development environment for many applications, but with the added benefit that it provides guided construction no matter what the application.

In conclusion, ARF is very suited to creating testing scenarios for remote platforms and has demonstrated that it can easily provide the flexibility to do all the working modes and uses identified in the requirements, with the added benefit that it is capable of doing them all at once. In addition, ARF provides unrivalled scenario creation times, which are applicable to almost any usage, and more importantly stops the re-invention of the wheel when creating new virtual environments or modifying existing ones. ARF allows for more specific scenarios to be tailored quickly to the demands of the applications. This is attributable to ARF's unique guided construction mechanisms and easily extendable JavaBeans backbone. In future ARF will help produce more mature technologies than could have been achieved otherwise in less time and consequently costing less. ARF does this by providing more accessible and user friendly testing facilities which can be reconfigured and extended in minimal time to produce scenarios for many different testing and AR applications.

9.1 Further Work

There are many avenues one could take for further research of ARF. One of the main points of interest is further improving the intuitive interface the user uses to connect

components together via guided construction. The LEGO® analogy highlighted some key ideas which ARF could implement. One of these ideas is to visualise components as shapes. Properties which require a certain component as a reference can be represented as keyways which fit onto the shapes, i.e. if two shapes have a similar keyway, they can be connected together. This is in essence just the same as guided construction, but instead of showing the user a list of possible components they already have, or a list of components they could create, it displays on the screen which shapes fit the connection shape. Then the user simply has to move the shapes into contact, via spatial manipulation, to connect them. This type of visual user interface would be most useful when rearranging component connections. All the shapes could be displayed on the screen with their keyways showing how they are connected. The user could then simply drag the keyway connections onto different shapes which have the same keyway. Thus allowing connections between components to be severed and connected with great speed. ARF could highlight upon a mouse click which shapes have that same connection keyway, at which point the user would simply have to drag the shape keyway onto the other shape. This would be more like the building blocks found in LEGO®. This would be useful for allowing quick rearrangements of connections. Also the user could arrange the objects spatially into different areas on the screen to represent groups with different functionality, rather like SuperComponents. This would in essence be an addition to the BeanBoard and Guided Construction functionality of ARF.

However, this would be quite a radical change to ARF. There are some smaller steps which could be taken to improve on ARF's user interfaces before implementing the LEGO® approach. These include:

- Improve BeanBoard view in ARF - improve the ordering of the JavaBeans to make it easier for the user to locate them. Have the ability to place Beans in functional groups on the BeanBoard. Show status indicators of Beans, i.e. do they have null pointers or properties which need setting before they are ready.

- Improve the JarRepository - tt some point the repository will grow so large that JavaBeans will need to be sorted and categorised better according to usage. This

will require a tree structure instead of the current list view of possible JavaBeans. This can be used for guided construction and also when JavaBeans are added directly to the BeanBoard.

- Add a SuperComponent Repository which manages SuperComponents by name and description so that they can be added in a more convenient way to the project. This would allow SuperComponents to be added in much the same way as JavaBeans are to the BeanBoard. If this were combined with the improvements to the BeanBoard, i.e. group related components, this would make navigation of the BeanBoard much more intuitive.

- Allow addition of Java3DBeans to SceneGraph in guided construction pane. This means duplicating some functionality from the Java3D SceneGraph GUI and adding it to the guided construction pane.

Another quite radical region for further investigation is that of separating the core parts of ARF, such as the BeanBoard, J3DSceneGraph and PropertyPane, into JavaBeans which can then be constructed using a JavaBean IDE such as BeanBuilder. This would allow ARF to be incorporated into other programs, and in turn ARF can incorporate other programs into itself; similar to the chicken and the egg situation. The advantage being that parts of ARF such as the 3D virtual world could be substituted with say a 3D environment from NASA WorldWind [74]. Thus creating an ARF but geared more towards virtual worlds with WorldWind, rather than Java3D. This would allow programmers to develop different versions of ARF which could be geared more to one sort of application. Although ARF itself already allows JavaBeans to be added which could include a different 3D virtual world, it could be useful to allow the programmer to drop down a level and actually customize their own version of ARF. Further work could be carried out to incorporate the extra functionality provided by ARF into something like the BeanBuilder. However, this would mean adding the SuperComponent, GuidedConstruction and Project functionality of ARF into the BeanBuilder itself which would be difficult. The advantage of breaking ARF into separate JavaBeans is that different versions of constituent parts, take for example the BeanBoard, could be substituted for different representations which are more

appropriate to a specific usage of ARF. ARF itself could be modified so that it allows the user to choose which parts they want, e.g. the user could choose a different type of BeanBoard view depending on the purpose of their project. This information can then be saved as part of ARF's Project files.

Currently ARF uses the Java3D SceneGraph and as such a work around was required to make the Java3D SceneGraph nodes into JavaBeans (Java3DBeans). It would be beneficial if these enhancements were factored back into the actual Java3D SceneGraph classes so that SceneGraphs can be exported easily as JavaBeans by others.

9.1.1 Limitations of Work

Currently, ARF has many applications in the testing of robotic platforms due to the many different use cases which were demonstrated in chapter 7. Thus, many components were created that focused particularly on HIL testing. ARF could become a more feature rich facility specifically for creating HIL testing scenarios, however, more specialised components would need creating that focus on synchronising the virtual and the real world, since this is one of the main problems of HIL testing. ARF provides the facilities to support potentially limitless components for doing specific tasks related to HIL testing, and therefore it would be a very useful extension to ARF to provide more components for HIL specific tasks so that ARF can be easily used for many tasks without further component creation. Naturally, the component base of ARF will grow over time, so rather than further work this more of a future goal. The idea of Hybrid Simulation (HS) provides another problem: are there certain ideas which cannot be tested using HIL techniques? For example, if a real robot was navigating around a real environment but avoiding virtual objects (augmented reality from the robots perspective), how do you make the real robot collide realistically with a virtual object? This cannot be easily achieved since a totally realistic implementation would require the virtual environment to alter the physical properties of the real environment so that the robot actually collides. Another approach would be to simulate the collision by altering the outputs of the robot's motors so that it behaves as though it has collided. However, this implementation does not very accurately

simulate the physical laws governing the vehicle when colliding with a real object. This is a problem not specific to ARF, but for which ARF could easily be used to try and solve. ARF allows for components to be easily added to existing scenarios, a component could be added which provides the ability to control the systems of the remote robot to simulate how it would react when colliding with a virtual object. Various different implementations could be easily added as separate components to ARF and thus easily tested.

9.1.2 Future Applications and Research

ARF has enabled virtual environments to be created quickly and easily for many different uses, however, now that ARF is available it opens many doors to different types of research. One of the main areas of interest is that of testing fault detection systems. ARF can be used to simulate the various systems of an AUV, however instead of supplying correct data to the real modules in a HIL fashion, it can be used to generate incorrect, inaccurate or contradictory data. The developer of the fault detection system will then be able to see if the erroneous data output by ARF causes any states of deadlock within the system which need to be caught and dealt with. One of the key contributions of ARF is that it allows for normal faults to be easily spotted whilst testing modules in simulation and HIL testing. Without ARF scenarios take much longer to alter and create and as such simple additions to scenarios, which provide more visual feedback of the states of a system, are not carried out. The example of BAUUV (section 7.7) is a classic case of observing a fault which would be very hard to debug without being able to monitor both real and virtual systems concurrently i.e. the lag which caused objects to be detected in the wrong place would be hard to spot without the necessary feedback for the operator which shows the correlation between the virtual object and the detected position of the virtual object. However, this also highlights the fact that human-in-the-loop testing is vital, but in order to be effective a virtual environment is required which can be configured easily to show the necessary feedback and simulate the required sensors. This is where ARF provides a particularly useful application.

The flexibility of ARF opens enables new applications of HS and HIL testing. For

example, place ARF onboard the actual AUV when no external communication is necessary. This allows for doing HS/AR testing when an AUV is on an autonomous mission. The virtual environment could be kept in synchronisation with the real environment by rendering new data in to the virtual world. The virtual AUV is also kept in synchronisation and as such extra virtual sensors could be placed on the virtual AUV to provide pseudo realistic data to systems which may require some sensor which is not yet part of the real AUV. For example, bathymetric data collected by sonar could be rendered in the virtual world and then a simulated video camera used to provide images to some video processing system, i.e. for object detection and tracking systems which only work on video data. Using the same onboard virtual environment it would be possible to estimate things like water currents by running a hydrodynamic simulation of the vehicle at the same time as running the real vehicle.

$$waterMovement = realPosition - virtualPosition$$

This could also be used to find out the differences between the hydrodynamic simulation and the real hydrodynamics of the vehicle (if there are no water currents). This information can then be used in turn to provide a more accurate hydrodynamic model.

As shown there are many different avenues for the direction in which ARF could be taken. Wherever it may lead and whatever enhancements maybe carried out it is important to maintain ARF's generic nature and not restrict its usage, any enhancements should increase its flexible and intuitive nature rather than restrict it.

Appendix A

Accompanying CD-ROM

Many items have been put on the CD-ROM which accompanies this thesis. The CD-ROM contains the following:

- ARF Tutorial
- ARF JavaDocs
- ARF Source Code in Netbeans Projects
- Video demo of how to use ARF
- Videos of ARF usages e.g. Nessie 3, Sauce, DTC, Copehil Down etc
- Scenarios for ARF
- ARF itself

All the items on the CD-ROM are in folders which names match those described above.

Appendix B

Extra PropertyEditors

There are many standard property editors in package `framework.introspection.propertyeditors` which belong to other Java projects. These are only basic editors for Java primitive types such as Integer, Double, String etc. The following extra property editors are provided by ARF to enable intuitive editing of specific Java object types.

- `framework.gui.BensBeanSelectorEditor` provides the GuidedConstruction editor for Properties which reference an Object type with no specific associated PropertyEditor.
- `framework.gui.BensFileSelectorEditor` provides a property editor for selecting files for load or save, depending on what the JavaBean does.
- `framework.introspection.propertyeditors.Vector3dEditor` is a simple editor for javax.vecmath.Vector3d objects.
- `framework.introspection.propertyeditors.Vector3fEditor` is a simple editor for javax.vecmath.Vector3f objects.
- `framework.introspection.propertyeditors.Java3DColor3fEditor` is a simple editor for javax.vecmath.Color3f objects.
- `framework.introspection.propertyeditors.SwingEnumEditor` is a simple editor for Enumeration types in JavaBeans.

- `framework.introspection.propertyeditors.SwingListEditor` is a simple editor for all Java classes which implement the *List* interface.
- `OceanLIB.propertyeditors.T_LLDEditor` is an editor for specifying which Lat, long, depth coordinate types.

Appendix C

ARF Components

The current ARF component Library to date:

```
3D Mesh loaders:
framework.j3d.X3DLoader_CyberX3D
framework.j3d.MeshLoader_DXF_XTools
framework.j3d.MeshLoader_J3D_VRML97
framework.j3d.X3DScene

HUD:
framework.hud.HudText
framework.hud.AlarmGauge
framework.hud.ToggleGauge3D
framework.hud.BarGauge3D
framework.hud.PosNegBarGauge3D

3D Geometry:
framework.geometry.Sphere
framework.geometry.Barrel
framework.geometry.ExclusionZone
framework.geometry.Ball
framework.geometry.TargetMarker
framework.geometry.FOVSegment
```

```
framework.geometry.Pipe3D
framework.geometry.Trail3D
framework.geometry.Cone
framework.geometry.Cube
framework.geometry.Box
framework.geometry.Grid
framework.geometry.MoveableChangingTexture
framework.utils3d.noise.OptimizedFractalTerrain
framework.utils3d.noise.WaterSurface
framework.behaviours.SpecificObjectPlotter
OceanLIB.arf.geom.pipetools.PipeLoader
OceanLIB.arf.image.DlgBMPWChartLoader
OceanLIB.arf.image.TextureTG

Other Java3D SceneGraph nodes:
framework.j3d.Camera
framework.j3d.ARFBranchGroup
framework.j3d.ARFTransformGroup
OceanLIB.arf.ARFCoordConverterGroup

Java3D Effect Nodes:
framework.j3d.ARFBackground
framework.j3d.ARFFog

Java3D Behavior Nodes:
framework.behaviours.MovementListener
framework.behaviours.MouseInspectBehavior
framework.behaviours.FollowBehavior
framework.behaviours.KeyNavigatorBehavior
framework.behaviours.AccelNavigatorBehavior
framework.behaviours.TransformInterpolator
```

OceanLIB.arf.messageIO.nessie.SonarSpinnerModule

OceanLIB.arf.simulation.rauver.io.WayPointRqstMsgPlotter (OSh Rec.)

OceanLIB.arf.messageIO.NavMsgOutputBehaviour (OceanSHELL Output)

OceanLIB.arf.simulation.rauver.ScreenPicker

Java3D Simulated Sensors:

OceanLIB.arf.sensor.CollisionVolumeSensor

OceanLIB.arf.sensor.FOVPickSensor

OceanLIB.arf.sensor.ObjectFindInSphere

OceanLIB.arf.sensor.ForwardLookingSonarSensor

OceanLIB.arf.sensor.SonarRayTracer

OceanLIB.arf.sensor.ReflectanceRayTracer

OceanLIB.arf.messageIO.nessie.SonarVolumeSensor

OceanLIB.arf.messageIO.nessie.Altimeter

OceanLIB.arf.messageIO.nessie.SonarWallFindPickRays

Simulation Components:

OceanLIB.arf.messageIO.nessie.IMUSimulator (J3dBean)

OceanLIB.arf.simulation.rauver.RauverModel

OceanLIB.arf.simulation.rauver.RauverAutoPos

OceanLIB.arf.simulation.rauver.AUVModelDynamics

Swing 2D JavaBeans:

framework.j3d.ARFCanvas3D

OceanLIB.arf.sensor.RecordCanvas3D

framework.gui.ConfigurablePanel

OceanLIB.arf.sensor.RawSensorViewer

Miscelanious JavaBeans:

OceanLIB.arf.CoordinateSystemsConverter

OceanLIB.arf.simulation.SimulationClock

OceanLIB.arf.sensor.ImageFileWriter

OceanSHELL Core JavaBeans:

OceanLIB.arf.oceanshell.OShQ

OceanLIB.arf.oceanshell.OShBridge

OceanSHELL Data I/O:

OceanLIB.arf.messageIO.CADCACMsgListener

OceanLIB.arf.messageIO.PickedNodeMsgOutput

OceanLIB.arf.messageIO.AutotrackerMultiBeamOutput

OceanLIB.arf.messageIO.AxisForceMsgListener

OceanLIB.arf.messageIO.CollidedNodeMsgOutput

OceanLIB.arf.messageIO.NavMsgListener

Simulation RAUVER I/o:

OceanLIB.arf.simulation.rauver.io.RvrThrustMsgListener2AUVModel

OceanLIB.arf.simulation.rauver.io.WayPointRqstMsgListener2AutoPos

OceanLIB.arf.simulation.rauver.io.NavListener_2_J3DTransformGroup

OceanLIB.arf.simulation.rauver.io.AutoPos2RvrAutoposInPositionMsg

OceanLIB.arf.simulation.rauver.io.AUVModelNAV2RauverModelNAVMsg

OceanLIB.arf.simulation.rauver.io.WayPointOutput

Simulation ALI I/O:

OceanLIB.arf.ali.io.Ali_AxisForceMsg2AUVModel

OceanLIB.arf.ali.io.Ali_NavGoalCmdMsg2AutoPos

OceanLIB.arf.ali.io.AUVModel2Ali_NavStsMsg

OceanLIB.arf.ali.io.AutoPos2ali_NavGoalStsMsg

Nessie OceanSHELL I/O:

OceanLIB.arf.messageIO.nessie.SonarStartMsgListener

OceanLIB.arf.messageIO.nessie.WorldDataMsgOutput

OceanLIB.arf.messageIO.nessie.NessieIMUDataMsgListener

OceanLIB.arf.messageIO.nessie.ForwardCameraMsgOutput

OceanLIB.arf.messageIO.nessie.NessieWorldDataMsgListener

OceanLIB.arf.messageIO.nessie.SonarMsgOutput

OceanLIB.arf.messageIO.rauboat.GPS_Compass_MsgListener

OceanLIB.arf.messageIO.nessie.DownwardCameraMsgOutput

OceanLIB.arf.messageIO.nessie.NessiePICFeedbackMsgListener

OceanLIB.arf.messageIO.nessie.NessieThrusterCmdMsgListener2Rauver-ModelAdapter

OceanLIB.arf.messageIO.nessie.NessieWayPtRqstListener2AutoPos

nessie3.detection.ForwardCameraPixelMsgOutput

nessie3.DvlMsgListener

nessie3.DMM16ATMsgListener

DELPHIS Components:

delphis.DelphisCoordinationNavListener

delphis.DelphisCoordinationMsgListener

Sauce Logging and Nessie 3 Components:

sauce.logging.SaucePlayer

sauce.logging.SauceLogger

sauce.logging.eventlisteners.DataAssociationEventListener

sauce.logging.eventlisteners.IntentionEventListener

sauce.logging.eventlisteners.Waypoint4DEvent2WayPointRequestEvent

sauce.logging.eventlisteners.PositionEvent2NavListenerEvent

nessie3.detection.WorldModel

nessie3.detection.CollidedNodeEvent2DataDetectedEvent

C.1 ARF Programming

The ARF Class hierarchy is split across many different projects:

- ARFInterfaces - This is the project which contains all the core interfaces which ARF uses. User's components should implement these interfaces where necessary to gain use of ARF's extra features.
- Arf4Auv - This is the main project which contains all the ARF architecture, the GUI, the PropertEditors and some Java3DBeans for use in ARF.
- OceanLIBARF - This contains JavaBeans and Java3DBeans for use in ARF. This is mainly concerned with simulated components for underwater applications. It includes things like the Hydrodynamic model and OceanSHELL communications components.
- Nessie3 - This contains ARF components required for testing Nessie 3 and also fro creating and playing back log files in the SAUC-E XML format.
- DelphisARF - Contains the required ARF components to receive the DELPHÍS messages and render the data in ARF.

Most projects related to ARF will have dependencies on the ARF class libraries. Usually OceanLIBARF for OceanSHELL and ARF for other functionalities. Many of the features of these packages are discussed in Chapter 5 and also in the ARF Tutorial in Appendix A.

The JavaDocs display the class hierarchies of all these projects and provide descriptions of each classes functionality. These are descussed in Appendix A

Appendix D

ARF Performance Tests

ARF Performance Test (APT)

Introduction:
This test is designed to quantitatively asses the performance of ARF. Performance is defined as being the time it takes a programmer to perform a certain task in ARF compared to standard methods. The performance is measured for various different tasks to highlight where ARF is better/worse. The comparison of ARF will be carried out against a standard programming IDE (normally used to write quick test programs for modules written by programmers) which also has guided programming capabilities (NetBeans). Netbeans provides automatic JavaBean method creation, Automatic JavaDoc display for all methods and descriptions when using an object.

These tests are designed to measure the performance of: Creation, Changability / Flexability and Extendability.

From these tests, ARF's guided programming mechanisms, combined with the features of its low level JavaBeans arhcitecture, will hopefully show an increase in performance over conventional programming methods.

Performance Metrics:
Performance is measured from 0 to 4, 4 being the worst and 0 being the best. Thus overall the higher the score the worse the participant performed. However, this may reflective of specific problems with the test itself and thus will be highlighted.

Brief:
You are an embedded systems developer requiring a virtual reality testing suite to test a CADCAC (computer aided detection, computer aided classification) system for an autonomous underwater vehicle (AUV). Navigation information is output by a system running elsewhere. Note: this system could be real or simulated, but this makes no difference to the communication interface. The NAV system outputs Rauver Nav OceanSHELL Messages. The AUV will swim a lawn mower style recognisance of a 20mx20m square (0,0,1) to (20,20,1) (North,East,Depth) looking for obstacles which could be mines. Thus the virtual world should have features which are detectable in this region. Your task is to build a virtual reality scenario which listens for NAV messages and moves a 3D model, of an AUV, synchronously with the NAV system. The 3D model should also have attached a simulated Sonar module. The sonar should connect an OceanSHELL Object Detected message output. This is to test the "real" CADCAC software running elsewhere. The features would normally be detected by a forward looking sonar with a vertical (x) FOV (Field of View) of 30 Degrees (at a resolution of 0.5 degrees per beam i.e. 60 beams total), Horizontal(y) FOV of 90 Degrees (at a resolution of 0.5 degrees per beam i.e. 180 beams total), 20metres range, and scans every 1000milliseconds. Vertical and Horizontal FOV are specified as a rotation about the x axis and y axis respectively. A high fidelity sonar simulation is not required, instead a simple Field of View (FOV) Pick sensor can be used to simulate the detection of 3d objects within a specified FOV. In Java3D a pick Ray, which is basically a line in 3D space, can be used to check which objects collide with it. Thus an FOV pick sensor sends out a fan of multiple pick rays, like a sonar would. The user needs to set up the FOVPickSensor to the same specifications as the required sonar.

Before completing this test it is advisable that you have familiarised yourself with the ARF tutorial. Extra API information on JavaBean classes will be provided at the time of the test, but not before as this test is designed to test how long it takes to use these references if they are required.

1) Exercise 1: Building a Scenario in ARF

Java3DBeans: *FOVPickSensor, FOVSegment, Grid, Cone, Box, ARFTransformGroup*

JavaBeans: *CollidedNodeMsgOutput, OshQ, NavMsgListener*

In order to test the guided programming facility, you should aim to connect the supplied components (above in BLUE) together as quickly since possible as this part of the exercise will be timed.

a) Use a ARFTransformGroup with a Cone as a child and point the cone down the –Z axis to represent the vehicle (-Z is forward(North) in Java3D). Another ARFTransformGroup as a parent of the other will be necessary to move the Cone around. Thus the tree should contain: ARFTransformGroup *(to move AUV)* → ARFTransformGroup *(To orient cone)* → Cone

Fill out Answers:

b) The next task is to add the FOVPickSensor to the AUV TransformGroup. The FOVPickSensor can also be connected to an FOVSegment Shape which will display the field of view of the pick sensor in the 3D world. The FOVSegment should also be added to the AUVTransformGroup

- The FOVPickSensor added to the vehicle TransformGroup and its properties configured to the sonar specifications described in the ***brief***.
- Tip: The FOVSegment will automatically configure to same characteristics as FOVPickSensor when the FOVPickSensor's property called "FOVSegment" is connected to the FOVSegment. The FOVSegment should first be added to the vehicle's TransformGroup on the SceneGraph.

You should add the *CollidedNodeMsgOutput* JavaBean as an *event listener* to the FOVPickSensor. The *CollidedNodeMsgOutput* outputs an OceanSHELL message containing which nodes have been detected by the FOVPickSensor. The FOVPickSensor should have its *"collidablesGroup"* property set to a group which will eventually contain 3D objects which can be detected by the sensor (such as "collidablesGroup"). An OShQ will need to be created with *input* port *49002* and output port *49003.* The CollidedNodeMsgOutput should have its OShSenderStub property connected to the OShQ. The NAV messages will be output by a simulator running separately from ARF, these should be listened to by the NavMsgListener. The NavMsgListener should have its OshReceiverStub property set to the OShQ and the targetTransformGroup property set to the vehicles TransformGroup.

Fill out Answers:

c) This set of components can now be saved for use later. This can be done in 2 ways.

i) The entire project can be saved.

Fill out Answers:

ii) The required set of components to represent this AUV and its related JavaBeans can be saved as a super component. This allows this particular set of components to be imported into any other project and used countless times again. Meaning that the user doesn't have to re-create the same parts each time. Thus in order to export the super component, the user must export any required Java3D scenegraphs, and any other **required** JavaBeans from the BeanBoard. ARF helps by identifying dependencies between JavaBeans and adds any referenced JavaBeans from the BeanBoard to the export so that references don't get broken.

Fill out Answers:

d) Change the OceanSHELL input port to *48002*, and output port to *48003.* Then save this project under a different name from the previous.

Fill out Answers:

e) Consider a new scenario where two robots are being simulated doing the same task. What is the quickest way to add another robot to this scenario? The new robot should receive NAV OceanShell messages on port 49002 and send detected mine information on 49003. Add another AUV and associated JavaBean objects to this world to represent the new robot.

Fill out Answers:

f) Now you need to switch back to the first scenario with only one AUV. Load the old scenario. *This is to test the original setup.*

2) Exercise 2: Building Scenario in NetBeans

Java3DBeans: *FOVPickSensor, FOVSegment*
JavaBeans: *CollidedNodeMsgOutput, OShQ, NavMsgListener*

The task here is slightly different from before because you won't be creating the 3D world using java code, as this would only test your knowledge of Java3D and wouldn't provide any useful information as the task would take too long to do – Java3D is merely harnessed by ARF and therefore the user shouldn't need to know how to program in Java3D in order to program for, and use, ARF.

Instead, most of the required Java3D objects will already be passed to the class which you will help program. All java object references required are already defined in the class and have *"protected"* access. It will be up to the user to instantiate them if they haven't been already. The user must then set them up and connect them to each other accordingly. Obviously, the user will already have some knowledge of the objects from using ARF and will thus recognise some methods and know how to use them. ARF does some things that the user might not know about, so the user will have to take special care to configure each object carefully.

Hint: The objects can be initialised like JavaBeans by using the zero argument constructor for each class.

a) Using the Netbeans "Scenario.java" file. You must join the Objects just like in Exercise **1b**. You must enter your code inside the *"setupScenario(TransformGroup auvTransformGroup, BranchGroup collidables)"* method. This method has the TransformGroup of the AUV, and the BranchGroup which has collidable geometry attached, passed to it. You should instantiate the ***remaining*** JavaBeans and Java3DBeans, listed above, and configure them as per the original specification in the ***brief***. The scenegraph is constructed in a separate method *"setEnable(Boolean)"* and consequently you should not be concerned with how to construct the java3d scenegraph. You only need to connect the components together via their methods (JavaBean parameters). The Java3D scenegraph construction is done automatically elsewhere. Comments in the Java code describe in more detail what needs to be done. The required Java objects are already defined as member variables declared in the class body. These objects need to be instantiated and then connected to each other as follows.
   - The FOVPickSensor should be connected to the FOVSegment Shape which will display the field of view of the pick sensor in the 3D world.
     - The FOVPickSensor should be configured to the sonar specifications described in the brief.
     - Tip: The FOVSegment will automatically configure to same characteristics as FOVPickSensor if it is connected to FOVPickSensor before FOVPickSensor is configured.
   - The FOVPickSensor should have a *CollidedNodeMsgOutput* ***event listener*** added to it to output an OceanSHELL message containing which nodes have been detected by the sensor.
   - The FOVPickSensor should have its *"collidablesGroup"* property set to the supplied BranchGroup as this contains 3D objects which can be detected by the sensor.
   - An OShQ will need to be configured. The CollidedNodeMsgOutput should have its oshSenderStub property connecting to the OshQ. The NAV messages will be output by a simulator running separately from ARF on port *49002,* these should be listened to by the NavMsgListener which in turn moves the AUV TransformGroup. OceanSHELL messages from *CollidedNodeMsgOutput* should be sent on port *49003*. The NavMsgListener should have its oshReceiverStub property connecting to the OshQ.

Fill out Answers:

b) Make a new Scenario2 class with same configuration as before except change the OceanSHELL input port to *48002*, and output port to *48003.*

Fill out Answers:

3) Exercise 3: Creation of ARF Component and ARF integration

The class you programmed in exercise 2 needs to now be tested. This class has had special methods added to it to allow it to be a JavaBean and tested within ARF. The JavaBean properties of "Scenario.java" file allow for the *"collidables" Group node* and the AUV *TransformGroup node* to be specified by the user of ARF. There is an enable property which must be set to true, once the other properties are setup, in order to test the code written in exercise 2. If all the Objects have been instantiated, correctly configured and connected to one another, then when the user sets the *"enabled"* flag of this JavaBean to *true* then it should start receiving NAV messages, moving the robot around, and also sending out messages of discovered targets.

a) You must make your "Scenario.java" file into a JavaBean. The compiled *"Scenario.class"* file should be wrapped up in a Jar File. The built Jar file will also contain a file called "manifest.mf". Before building the Jar you should modify the "manifest.mf" file to include an entry indicating the full Java package and class name (e.g. "arfperformancetest/Scenario") of the "Scenario" JavaBean. The manifest file is located at *"E:\java\ARFPerformenceTest\manifest.mf"*. Your manifest should look something like:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: 1.5.0_06-b05 (Sun Microsystems Inc.)
X-COMMENT: Main-Class will be added automatically by build

Name: arfperformancetest/Scenario
Java-Bean: True
```

To build the JAR file select the project root, in the left hand navigation pane, and click right-mouse button and select "build". The Jar file is output to *"E:\java\ARFPerformenceTest\dist\ARFPerformenceTest.jar"*. The manifest will be wrapped in the Jar file during the build process. The entries in the manifest are used to identify which classes in the JAR file are JavaBeans (See tutorial on "JavaBean Creation")

Fill out Answers:

b) The next task is to import the Jar file into ARF. This can be done by going to File → Manage Repository → Enter the path of the Jar file and Click "add" → Click Done. JAR is located at *"E:\java\ARFPerformenceTest\dist\ARFPerformenceTest.jar"*.

Fill out Answers:

c) Open the ARF project located at *"My Documents/ModuleTester.xml"*.
   - Add the new *"arfperformencetest.Scenario"* bean to the BeanBoard.
   - Make sure the auvTransformGroup, collidables properties are pointing to the similar named Java3DBeans **already** created in this project. The "auvTransformGroup" property should connect to a TransformGroup which is used to move the vehicle. The "collidablesGroup" property should be connected to the BranchGroup node on which the "Box" object is connected to (this is the collidable objects group).
   - Then set *"enable"* on the propertySheet to *"true"*. The dos box should show a NavMsg debug output, and a detected object position. An FOVSegment shape should have also appeared in the 3Dworld attached to the AUV. If this hasn't happened then there will be an exception and a stack trace printed to the output which you should be able to find the error in and consult the moderator if assistance is required.

Fill out Answers:

Appendix E

APT Results Sheet Design

APT Results Sheet:

Name:
Date:

Experience

Experience of Java	5 (Expert), 4 (a lot of experience) ,3 (some experience), 2 (familiar with concepts), 1 (none)
Experience of Java3D	5 (Expert), 4 (a lot of experience) ,3 (some experience), 2 (familiar with concepts), 1 (none)
Experience of IDE	5 (Expert), 4 (a lot of experience) ,3 (some experience), 2 (familiar with concepts), 1 (none)
Experience of JavaBeans	5 (Expert), 4 (a lot of experience) ,3 (some experience), 2 (familiar with concepts), 1 (none)
Usage of ARF	5 (Expert), 4 (a lot of experience) ,3 (some experience), 2 (familiar with concepts), 1 (none)

1a.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Consulted API no. times?				
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of ARF	Didn't consult turorial when should have	Other
If other please state:				
Any Comments/feedback regarding the task				
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

1b.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Consulted API no. times?				
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of ARF	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
Event Listeners usability: 0 *(easy)* to 4 *(hard)*	0(Obvious/intuitive), 1, 2(Neither obvious or unobvious), 3, 4(Completely counter intuitive)			
JavaBean Reference Property Interconnection: 0 *(easy)* to 4 *(hard)*	0(Obvious/ intuitive), 1, 2(Neither obvious or unobvious), 3, 4(Completely counter intuitive)			
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

1ci.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of ARF	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

1cii.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of ARF	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

1d.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of ARF	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

1e.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of ARF	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

1f.

Time Taken (hh:mm:ss)	
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.

2a.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Consulted API no. times?				
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Didn't know how to use netbeans	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
Event Listeners usability: 0 *(easy)* to 4 *(hard)*	0(Obvious/easy), 1, 2(Neither obvious or unobvious), 3, 4(Completely unobvious)			
Object Property Interconnection: 0 *(easy)* to 4 *(hard)*	0(Obvious/easy), 1, 2(Neither obvious or unobvious), 3, 4(Completely unobvious)			
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

2b.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Didn't know how to use netbeans	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

3a.

Time Taken (hh:mm:ss)				
Consulted Tut no. times?				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of Netbeans	Didn't Consult documentation	Other
If other please state:				
Any Comments/feedback regarding the task				
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

3b.

Time Taken (hh:mm:ss)				
Consulted Tut no. times?				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of ARF	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

3c.

Time Taken (hh:mm:ss)				
Overall Task Complexity	0) Very Straightforward, 1) StraightForward, 2) Neither Complex or straightforward, 3) Complex, 4) Very Complex			
Required Assistance	0, 1, 2, 3, 4 *or more*			
If required assistance not zero, why?	Documentation inadequate	Unable to understand part of ARF	Didn't understand Task.	Other
If other please state:				
Any Comments/feedback regarding the task				
How hard was it to instantiate and configure the new JavaBean?	0 (easy), 1, 2, 3, 4 (hard)			
Understanding of Task	5) Completely, 4) Most of it, 3) about half of it, 2) understood some parts, 1) understood none of it.			
Task Completion 0 to 4:	5) Completely, 4) Most of it, 3) about half of it, 2) some parts, 1) none of it.			

Appendix F

RAW Results Data

Exercise:	1	2	3	4	5	6	7	8	9	10	Average
1a											
Time (seconds)	120	127.2	256.1	124	261	233	445	207	452	281	250.63
Complexity	0	0	1	0	1	0	1	1	1	0	0.5
Consult API	0	0	0	0	0	0	1	0	0	0	0.1
Assistance no.	0	1	0	0	1	1	0	1	0	0	0.4
UnderStanding	5	5	4	5	5	5	4	5	5	5	4.8
Completion	5	5	5	5	5	5	5	5	5	5	5
1b											0
Time	251	742.2	700.6	325	659	847	1230	857	1013	544	716.88
Complexity	0	1	2	1	3	2	2	1	2	2	1.6
Consult API	0	0	0	0	0	0	0	0	0	0	0
Assistance no.	0	1	1	1	1	0	2	1	1	0	0.8
Event Listeners	5	4	3	4	5	5	5	4	3	5	4.3
JavaBean references	5	5	5	5	5	5	4	5	3	5	4.7
UnderStanding	5	5	5	5	4	5	4	5	5	5	4.8
Completion	5	4	4	5	5	5	5	5	5	5	4.8
1ci											0
Time	35.7	12.9	18	11	17	8	19	15	13	20	16.96
Complexity	0	0	0	0	0	0	0	0	0	0	0
Assistance no.	0	0	0	0	0	0	0	0	0	0	0
Reason?											0
UnderStanding	5	5	5	5	5	5	5	5	5	5	5
Completion	5	5	5	5	5	5	5	5	5	5	5
1cii											0
Time	41.3	77.1	47	60	24	36	63	69	28	27	47.24
Complexity	0	0	0	1	0	0	0	0	0	0	0.1
Assistance no.	0	1	0	0	0	0	0	0	0	0	0.1
Comments											0
UnderStanding	5	5	5	5	5	5	5	5	5	5	5
Completion	5	4	5	5	5	5	5	5	5	5	4.9
1d											0
Time	30.2	49.3	29	26	33	32	34	71	31	35	37.05
Complexity	0	0	0	0	0	0	0	0	0	0	0
Assistance no.	0	0	0	0	0	0	0	0	0	0	0
Comments											0
UnderStanding	5	5	5	5	5	5	5	5	5	5	5
Completion	5	5	5	5	5	5	5	5	5	5	5
1e											0
Time	82.1	115.1	106	101	97	120	180	62	54	71	98.82
Complexity	0	0	0	0	0	0	0	0	1	1	0.2
Assistance no.	0	0	0	1	0	0	1	0	0	0	0.2
UnderStanding	5	5	5	5	5	5	5	5	5	5	5
Completion	5	5	5	5	5	5	5	5	5	5	5
1f											0
Time	12	9.5	17	21	12	15	30	11	11	14	15.25
Completion	5	5	5	5	5	5	5	5	5	5	5
2a											0
Time	750.9	1498.4	810.1	805	1680	1111	2903	1328	2358	1363	1460.74
Complexity	1	0	1	2	3	0	3	2	4	3	1.9
Consult API	2	3	1	0	0	4	1	4	5	5	2.5
Assistance no.	0	0	2	0	2	1	1	1	1	0	0.8
Event Listeners	4	5	5	3	4	4	4	3	1	3	3.6
Object references	4	5	5	2	2	4	2.5	2	2.5	3	3.2
UnderStanding	5	5	4	5	4	5	4	5	4	5	4.6
Completion	5	5	4	5	5	5	4	5	2	5	4.5

2b											0
Time	42.9	125.5	38	57	54	109	40	123	100	87	77.64
Complexity	0	0	0	1	1	0	0	1	1	1	0.5
Assistance no.	0	1	0	0	1	0	0	0	1	1	0.4
Comments											0
UnderStanding	5	5	5	5	4	5	5	5	5	5	4.9
Completion	5	5	5	5	5	5	5	5	5	5	5
3a											0
Time	41.8	326.8	113.1	156	202	268	324	306	134	106	197.77
Consulted Tut no. Times	1	2	1	1	1	1	1	1	0	1	1
Complexity	0	0	1	2	2	2	1	1	1	0	1
Assistance no.	0	3	1	0	1	1	1	1	0	0	0.8
UnderStanding	5	2	4	5	3	5	4	4	5	5	4.2
Completion	5	5	5	5	5	5	5	5	5	5	5
3b											0
Time	26.6	67.6	50	33	35	71	70	116	58	42	56.92
Consulted Tut no. Times	0	0	0	0	0	0	0	0	0	0	0
Complexity	0	0	0	0	0	0	0	1	0	0	0.1
Assistance no.	0	0	0	0	0	0	0	0	0	0	0
UnderStanding	5	5	5	5	5	5	5	5	5	5	5
Completion	5	5	5	5	5	5	5	5	5	5	5
3c											0
Time	55.7	98.6	94	48	114	74	211	167	316	180	135.83
Complexity	0	0	0	0	1	0	1	1	3	1	0.7
Assistance no.	0	0	0	0	0	0	1	1	1		0.3
Instantian & Configuring Javabean	5	5	5	5	5	5	5	5	5	5	5
UnderStanding	5	5	5	5	5	5	4	5	4	5	4.8
Completion	5	5	5	5	5	5	3	5	5	5	4.8

Appendix G

Nessie III log file example


```
- <data>
  - <log agent="AUV1" system="ALLSYSTEMS" time="0">
    - <event type="waypoint" agent="AUV1" time="500" relative="false">
      - <waypoint number="6" depthMode="false" relative="false" mode="0" global="false">
          <local-coordininate north="144.3453" east="866.8408" depth="0.0" altitude="5.0" />
          <direction-request roll="0.0" pitch="0.0" yaw="0.0" />
          <position-tolerence x="1.0" y="1.0" z="1.0" />
          <direction-tolerence roll="0.0" pitch="0.0" yaw="0.0" />
        </waypoint>
      </event>
    - <event type="position" agent="AUV1" time="1000" relative="false">
        <local-position north="0.0" east="0.0" depth="0.0" altitude="3.980546" />
        <orientation roll="0.0" pitch="0.0" yaw="0.0" />
        <local-speed north="0.0" east="0.0" depth="0.0" />
        <rotation-speed roll="0.0" pitch="0.0" yaw="0.0" />
      </event>
    - <event type="intention-started" agent="AUV1" time="1500" relative="false">
        <intention predicate="Going to Waypoint" />
      </event>
    - <event type="data-association-change" agent="AUV1" time="2000" relative="false">
      - <association>
          <key name="1" type="validation-gate" />
          <position north="0.0" east="0.0" depth="0.0" altitude="3.980546" />
        </association>
      </event>
    - <event type="position" agent="AUV1" time="2500" relative="false">
        <local-position north="0.0" east="0.0" depth="6.5952E-5" altitude="3.980546" />
        <orientation roll="0.0" pitch="0.0" yaw="0.0" />
        <local-speed north="0.0" east="0.0" depth="0.0" />
        <rotation-speed roll="0.0" pitch="0.0" yaw="0.0" />
      </event>
    </log>
  </data>
```


Figure G.1: Example of an SAUC-E XML log file

Bibliography

[1] A. Healey, A. Pascoal, and F. Pereira, "Autonomous underwater vehicles: An application of intelligent control technology," in *Proceedings of the American Control Conference, Seattle, Washinton June 21-23, 1995, pp. 2943-2949.* on IEEE, 1995.

[2] *JavaBeans*, Sun Microsystems, http://java.sun.com/products/javabeans.

[3] *Java3D*, Sun Microsystems, http://java.sun.com/products/java-media/3D.

[4] P. Ridao, E. Batlle, D. Ribas, and M. Carreras, "Neptune: A hil simulator for multiple uuvs," in *OCEANS '04. MTS/IEEE TECHNO-OCEAN '04 Volume 1, 9-12 Nov. 2004 Page(s):524 - 531 Vol.1*, 2004.

[5] S. Choi and J. Yuh, "A virtual collaborative world simulator for underwater robots using multidimensional, synthetic environment," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on Volume 1, 2001 Page(s):926 - 931 vol.1*, 2001.

[6] Ocean Systems Laboratory, "Oceanshell: An embedded library for distributed applications and communications," Heriot-Watt, Tech. Rep., 2008.

[7] *LEGO*, LEGO, http://www.lego.com.

[8] R. T. Azuma., "A survey of augmented reality," in *Teleoperators and Virtual Environments 6, 4 Pages 355-385*, August 1997.

[9] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre, "Recent advances in augmented reality," *Computer Graphics and Applications, IEEE*, vol. Volume 2, Issue 6, pp. 34 – 47, Nov.-Dec. 2001.

[10] K. Kiyokawa, Y. Kurata, and H. Ohno, "An optical seethrough display for mutual occlusion of real and virtual environments," in *Intl Symp. Augmented Reality 2000 (ISAR 00), IEEE CS Press, Los Alamitos, California*, 2000, pp. pp. 60–67.

[11] D. Brutzman, "A virtual world for an autonomous underwater vehicle," Ph.D. dissertation, Monterey (USA), December 1994.

[12] P. Milgram, H. Takemura, and et al., "Augmented reality: A class of displays on the reality-virutality continuum." SPIE Proceedings: Telemanipulator and Telepresence Technologies . H. Das, SPIE. 2351 : 282-292, 1994.

[13] B. MacIntyre and E. Coelho, "Adapting to dynamic registration errors using level of error (loe) filtering," in *Proceedings of Intl Symp. Augmented Reality 2000 (ISAR 00), IEEE CS Press, Los Alamitos, California.*, 2000, pp. pp. 85–88.

[14] R. Behringer, "Registration for outdoor augmented reality applications using computer vision techniques and hybrid sensors," in *IEEE Virtual Reality, IEEE CS Press, Los Alamitos, California*, 1999, pp. pp. 244–251.

[15] R. Holloway, "Registration error analysis for augmented reality," in *Presence: Teleoperators and Virtual Environments. vol. 6, no. 4, Aug. 1997*, 1997, pp. pp. 413–432.

[16] Z. Qiang, I. Tena Ruiz, and D. Lane, "Inertial 3d simultaneous localization and mapping," in *DTC Conference Proceedings*, 2008.

[17] C. Hughes, C. Stapleton, D. Hughes, and E. Smith, "Mixed reality in education, entertainment, and training," *IEEE Computer Graphics and Applications*, vol. Volume 25, Issue 6, pp. 24 – 30, Nov.-Dec. 2005.

[18] R. Damus, J. Morash, and C. Chryssostomidis, "A wearable vehicle interface for augmented reality in operating auvs," in *OCEANS 2003. Proceedings Volume 2, 22-26 Sept. 2003 Page(s):1154 - 1160 Vol.2*, 2003.

[19] A. Fusiello and V. Murino, "Augmented scene modeling and visualization by optical and acoustic sensor integration," in *Visualization and Computer Graphics,*

IEEE Transactions on Volume 10, Issue 6, Nov.-Dec. 2004 Page(s):625 - 636, 2004.

[20] M. Zyda, R. McGhee, S. Kwak, D. Nordman, R. Rogers, and D. Marco, "Three-dimensional visualization of mission planning and control for the nps autonomous underwater vehicle," *Oceanic Engineering, IEEE Journal*, vol. Volume 15, Issue 3, p. 217 221, July 1990.

[21] D. M. L. et al., "Mixing simulations and real subsystems for subsea robotic development specification and development of the core simulation engine," in *IEEE Int. Conference OCEANS'98, Nice France, 29 Sept-2 Oct 1998.* Sendai, Japan: Springer-verlag, 1998, lNAI 1316.

[22] S. F. et al., "Modeling and simulation of autonomous underwater vehicles: design and implementation." IEEE Journal of Oceanic Engineering, Vol. 28, Issue: 2, pp.283-296, April 2003.

[23] D. Lane, G. Falconer, G. Randall, and I. Edwards, "Interoperability and synchronisation of distributed hardware-in-the-loop simulation for underwater robot development: Issues and experiments." Amsterdam, Holland: Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on Volume 1, 2001 Page(s):909 - 914 vol.1, 2001.

[24] X. Hu., "Applying robot-in-the-loop-simulation to mobile robot systems," in *Advanced Robotics, 2005. ICAR '05. Proceedings., 12th International Conference on July 18-20, 2005 Page(s):506 - 513*, 2005.

[25] Z. Papp, M. Dorrepaal, and D. Verburg, "Distributed hardware-in-the-loop simulator for autonomous continuous dynamical systems with spatially constrained interactions," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International 22-26 April 2003 Page(s):8 pp.*, 2003.

[26] P. Ridao, M. Carreras, D. Ribas, and A. El-Fakdi, "Graphical simulators for auv development," in *Control, Communications and Signal Processing, 2004. First International Symposium on 2004 Page(s):553 556*, 2004.

[27] Y. Kuroda, K. Aramaki, and T. Ura, "Auv test using real/virtual synthetic world," in *Autonomous Underwater Vehicle Technology, 1996. AUV '96., Proceedings of the 1996 Symposium on 2-6 June 1996 Page(s):365 372*, 1996.

[28] S. Choi, S. Menor, and J. Yuh, "Distributed virtual environment collaborative simulator for underwater robots," in *Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on Volume 2, 31 Oct.-5 Nov. 2000 Page(s):861 - 866 vol.2*, 2000.

[29] "The navy unmanned undersea vehicle (uuv) masterplan," U.S. Navy, http://www.navy.mil/navydata/technology/uuvmp.pdf, Tech. Rep., 2004.

[30] C. Sotzing, J. Evans, and D. Lane, "A multi-agent architecture to increase coordination efficiency in multi-auv operations," in *OCEANS 2007 - Europe 18-21 June 2007 Page(s):1 - 6*, 2007.

[31] B. Davis, P. Patron, and D. Lane, "An augmented reality architecture for the creation of hardware-in-the-loop & hybrid simulation test scenarios for unmanned underwater vehicles," in *Oceans 2007*, 2007.

[32] Z. Papp, K. Labibes, H. Thean, and M. van Elk, "Multi-agent based hil simulator with high fidelity virtual sensors," in *Intelligent Vehicles Symposium, 2003. Proceedings. IEEE, Page(s):213 - 218*, 9-11 June 2003.

[33] D. D. Corkill, "Blackboard systems," in *AI Expert 6(9):40-47, September*, 1991.

[34] V. R. Lesser, "Cooperative multiagent systems: a personal view of state of the art."

[35] J. Borges de Sousa and A. Gollu, "A simulation environment for the coordinated operation of multiple autonomous underwater vehicles," in *Simulation Conference Proceedings, 1997. Winter 7-10 December 1997 Page(s):1169 - 1175*, 1997.

[36] J. Weekley, D. Brutzman, A. Healey, D. Davis, and D. Lee, "Auv workbench: Integrated 3d for interoperable mission rehearsal, reality and replay," in *Minwara Canberra Australia 2003*, 2003.

[37] *Command and Conquer*, Westwood Studios, http://www.westwood.com.

[38] SeeByte, *SeeTrack*, http://www.seebyte.com.

[39] D. Brutzman and M. Zyda, "Extensible modeling and simulation framework (xmsf) findings and recommendations report 2002," Modeling, Virtual Environments and Simulation (MOVES) Institute, Naval Postgraduate School (NPS), Monterey California, http://www.movesinstitute.org/xmsf/XmsfWorkshopSymposiumReportOctober2002.pdf, Tech. Rep., 2002.

[40] *Distributed Interactive Simulation (DIS) IEEE Standard 1278.*

[41] *XML - Extensible Markup Language ISO/IEC 8825-4:2002*, http://www.w3.org/XML.

[42] C. C. Sotzing, N. Johnson, and D. M. Lane, "Improving multi-auv coordination with hierarchical blackboard-based plan representation," in *Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*, 2008, pp. 110–117.

[43] *W3C XSL Transformations (XSLT)*, http://www.w3.org/TR/1999/REC-xslt-19991116.

[44] K. Hamilton, "Recovery," Ph.D. dissertation, Ocean Systems Laboratory Heriot-Watt University.

[45] L. Fluckiger and C. Neukom, "A new simulation framework for autonomy in robotic missions," in *Intelligent Robots and System, 2002. IEEE/RSJ International Conference on Volume 3, 30 Sept.-5 Oct. 2002 Page(s):3030 - 3035 vol.3*, 2002.

[46] R. Gaskell, J. Collier, L. Husman, and R. Chen, "Synthetic environments for simulated missions," in *Aerospace Conference, 2001, IEEE Proceedings. Volume 7, 10-17 March 2001 Page(s):7 - 3556 vol.7*, 2001.

[47] G. Booch, I. Jacobson, and J. Rumbaugh, *OMG Unified Modeling Language Specification*, version 1.3 first edition ed., http://www.omg.org/docs/, March 2000.

[48] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner, "Design of a component-based augmented reality framework," in *Augmented Reality, 2001. Proceedings. IEEE and ACM International Symposium on 29-30 Oct. 2001 Page(s):45 - 54*, 2001.

[49] A. Behzadan, H. Khoury, and V. Kamat, "Structure of an extensible augmented reality framework for visualization of simulated construction processes," in *Winter Simulation Conference, 2006. WSC 06. Proceedings of the 3-6 Dec. 2006 Page(s):2055 - 2062*, 2006.

[50] M. Hirakawa and T. Ichikawa, "Advances in visual programming," in *Systems Integration, 1992. ICSI '92., Proceedings of the Second International Conference on 15-18 June 1992 Page(s):538 - 543*, 1992.

[51] S. Coon, M. Sanner, and A. Olson, "Re-usable components for structural bioinformatics," in *9th International Python Conference*, March 2001.

[52] Hyrdoid, *REMUS*, http://www.hydroid.com.

[53] Hafmynd, *Gavia*, http://www.gavia.is.

[54] Subsea7, *GEOSUB*, http://www.subsea7.com/rov_geosub.php.

[55] *RAUVER*, Ocean Systems Laboratory, http://www.ocean-systems-lab.com.

[56] DSTL, *Student Autonomous Underwater Vehicle Competition Europe (SAUC-E)*, http://www.dstl.gov.uk/news_events/competitions/sauce/index.php.

[57] J. Cartwright, N. Johnson, B. Davis, Z. Qaing, T. Bravo, A. Enoch, G. Lemaitre, H. Roth, and Y. Petillot, "Nessie iii," in *The 10th Unmanned Underwater Vehicle Showcase*, 2008.

[58] *Java*, Sun Microsystems, http://www.sun.com/java/.

[59] *X3D (ISO/IEC 19775-1)*, Web3D Consortium, http://www.web3d.org.

[60] *VRML97 (ISO/IEC 14772-1:1997).*

[61] *Xj3D project of the Web3D Consortium*, Web3D Consortium, http://www.web3d.org/x3d/xj3d.

[62] Wikipedia, *Software Component Theory.* [Online]. Available: http://en.wikipedia.org/wiki/Component-based-software-engineering

[63] G. Voss, "Testing beans in the bdk beanbox," http://java.sun.com/developer/onlineTraining/Beans/Beans3/, Tech. Rep., 1997.

[64] *The Official JavaBean Tutorial*, Sun Microsystems, http://java.sun.com/docs/books/tutorial/javabeans.

[65] *The BeanBuilder*, https://bean-builder.dev.java.net/.

[66] *Programming Language C++ ISO/IEC 14882:2003.*

[67] D. T. Davis, "Design, implementation and testing of a common data model supporting autonomous vehicle compatibility and interoperability," Ph.D. Dissertation, Naval Postgraduate School, Monterey California, September 2006. [Online]. Available: https://savage.nps.edu/Savage/AuvWorkbench/AVCL/AVCL.html

[68] *Persistence Delegates Tutorial*, Sun Microsystems, http://java.sun.com/products/jfc/tsc/articles/persistence4.

[69] *The Official Tutorials Website*, Sun Microsystems, http://java.sun.com/javase/technologies/desktop/articles.jsp.

[70] *JAVASHELL*, Ocean Systems Laboratory, 2008.

[71] P. Patrn, J. Evans, J. Brydon, and J. Jamieson, "Autotracker: Autonomous pipeline inspection: Sea trials 2005," in *World Maritime Technology Conference - Advances in Technology for Underwater Vehicles, March 2006*, 2006.

[72] P. Patrn, B. Smith, Y. Pailhas, C. Capus, and J. Evans, "Strategies and sensors technologies for uuv collision, obstacle avoidance and escape," in *7th Unmanned Underwater Vehicle Showcase 2005*, 2005.

[73] C. Ptrs, Y. Pailhas, P. Patrn, Y. Petillot, J. Evans, and D. Lane, "Path planning for autonomous underwater vehicles," in *proceedings of IEEE Transactions on Robotics, April 2007*, 2007.

[74] NASA, *WorldWind*, http://worldwind.arc.nasa.gov.