A PDA Interface for 3D Interaction with an Outdoor Robot

Andreas Hedström

TRITA-NA-E03089
A PDA Interface
for 3D Interaction with an Outdoor Robot

Andreas Hedström

TRITA-NA-E03089

Master’s Thesis in Computer Science (20 credits)
at the School of Vehicle Engineering,
Royal Institute of Technology year 2003
Supervisor at Nada was Henrik Christensen
Examiner was Henrik Christensen
A PDA Interface for 3D Interaction with an Outdoor Robot

Abstract

When looking at a coarse map of an unknown area, it is difficult to quickly orient and get a good feeling for the surrounding environment. This is especially true on a tiny Portable Digital Assistant (PDA), where it is difficult to grasp the extent of the entire map due to the small format. I therefore got the assignment to develop a Graphical User Interface (GUI) for a larger PDA and to take advantage of this new format. The GUI is used for interacting with Pluto, a mobile multi-purpose autonomous map-making scout robot. The purpose of this project is to investigate the use of 3D models to visualize the generated (2D) map. Another objective was to find out if video can be integrated into the GUI. The thesis covers the GUI design issues and the problems with implementing 3D and video on the given PDA. It also provides an evaluation of different tools for GUI development targeting PDAs and a brief evaluation of the implemented GUI. The implementation indicates that even a simple wire-frame model increases the understanding of the map considerably.

The thesis can be used as a reference for the Pluto GUI project.

Ett PDA-gränssnitt för 3D-interaktion med en utomhusgående robot

Sammanfattning


Denna rapport kan användas som referens för Pluto-GUI-projektet.
Preface

This paper is the result of a Master’s project done by Andreas Hedström during the fall and winter of 2002/2003 at CAS – Centre for Autonomous Systems. CAS is a research center connected to the Department of Numerical Analysis and Computer Science, NADA, at KTH – the Royal Institute of Technology – in Stockholm, Sweden.

The project is part of a larger project called Pluto (an outdoor mobile multi-purpose scout robot). The Pluto project is a collaboration between CAS and the Swedish Defence Material Administration, FMV.

This thesis concludes my M.Sc. education at KTH. Supervisor (at CAS/NADA) and examiner was Prof. Henrik Christensen.

I would especially like to thank my supervisor Henrik for making this Master’s project possible, and I also would like to thank the following persons:

- John Folkesson, a Research Engineer at CAS, who is in charge of the Pluto project and helped me with the modifications to the robot server and answered all my questions about Pluto. He also helped me to test the GUI and provided useful feedback.
- Carl Lundberg, a Ph.D student at CAS, who was a co-writer of the “old” PDA GUI and with whom I discussed the ideas for the “new” GUI design.
- Anders Hedström, my brother, who read this thesis and provided useful feedback.
- Thierry Tremblay who wrote PocketFrog – a graphics library for PDA games on which I based my own graphics window class.
- Frank Zammetti who supplied me with a polygon filler routine for PocketFrog so I did not have to write one myself.
- Conan, Pam, Thierry, Frank, Refractor and the rest of the guys at PocketFrog forum at www.pocketmatrix.com.
Title: A PDA Interface for 3D Interaction with an Outdoor Robot

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Purpose</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Content Overview</td>
<td>3</td>
</tr>
<tr>
<td>2 Assignment, Problems, Goals and Restrictions</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Assignment</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Problem Definition</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Goal</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Restrictions</td>
<td>6</td>
</tr>
<tr>
<td>3 PDA Definitions</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Windows CE</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Graphics API</td>
<td>8</td>
</tr>
<tr>
<td>3.3 Siemens MOBIC T8 Specification</td>
<td>9</td>
</tr>
<tr>
<td>4 GUI Tools</td>
<td>10</td>
</tr>
<tr>
<td>4.1 Java Based Tools</td>
<td>10</td>
</tr>
<tr>
<td>4.1.1 SuperWaba</td>
<td>10</td>
</tr>
<tr>
<td>4.1.2 Ewe</td>
<td>11</td>
</tr>
<tr>
<td>4.1.3 Dog Gui</td>
<td>11</td>
</tr>
<tr>
<td>4.1.4 CyberVRML97</td>
<td>11</td>
</tr>
<tr>
<td>4.2 C++ Based Tools</td>
<td>12</td>
</tr>
<tr>
<td>4.2.1 WxWindows</td>
<td>12</td>
</tr>
<tr>
<td>4.2.2 ATL/WTL</td>
<td>12</td>
</tr>
<tr>
<td>4.2.3 PocketFrog</td>
<td>13</td>
</tr>
<tr>
<td>4.3 Summary</td>
<td>13</td>
</tr>
<tr>
<td>5 GUI Design</td>
<td>14</td>
</tr>
<tr>
<td>5.1 The Old GUI</td>
<td>14</td>
</tr>
<tr>
<td>5.2 The New GUI</td>
<td>15</td>
</tr>
<tr>
<td>5.2.1 Control Panel Window</td>
<td>16</td>
</tr>
<tr>
<td>5.2.2 Manual Drive Window</td>
<td>17</td>
</tr>
<tr>
<td>5.2.3 Explore Area Dialog</td>
<td>19</td>
</tr>
<tr>
<td>5.2.4 2D Map Window</td>
<td>20</td>
</tr>
<tr>
<td>5.2.5 3D Map Window</td>
<td>20</td>
</tr>
<tr>
<td>5.2.6 Video Camera Window</td>
<td>22</td>
</tr>
<tr>
<td>5.2.7 Change Speed Dialog</td>
<td>22</td>
</tr>
<tr>
<td>5.2.8 Settings Dialog</td>
<td>22</td>
</tr>
<tr>
<td>5.2.9 Other Components</td>
<td>22</td>
</tr>
<tr>
<td>5.3 Summary And Conclusions</td>
<td>23</td>
</tr>
<tr>
<td>6 Implementing the GUI</td>
<td>24</td>
</tr>
<tr>
<td>6.1 PocketFrog and ATL</td>
<td>24</td>
</tr>
<tr>
<td>6.2 Threads and System Resources</td>
<td>24</td>
</tr>
<tr>
<td>6.3 Synchronization</td>
<td>25</td>
</tr>
<tr>
<td>6.4 Communication and Bandwidth</td>
<td>25</td>
</tr>
<tr>
<td>7 Graphics Library</td>
<td>27</td>
</tr>
<tr>
<td>7.1 General Issues</td>
<td>27</td>
</tr>
<tr>
<td>7.2 Browser</td>
<td>27</td>
</tr>
<tr>
<td>7.2.1 Main Loop</td>
<td>28</td>
</tr>
<tr>
<td>7.2.2 DrawShape()</td>
<td>28</td>
</tr>
<tr>
<td>7.2.3 DrawPolygon() / DrawFilledPolygon()</td>
<td>30</td>
</tr>
</tbody>
</table>
## Contents

8 Evaluation.......................................................................................................................................................... 31
9 Summary .......................................................................................................................................................... 34
  9.1 Recap......................................................................................................................................................... 34
  9.2 Solutions..................................................................................................................................................... 35
  9.3 Conclusion .................................................................................................................................................. 36
  9.4 Future Work and Recommendations........................................................................................................ 37
10 References..................................................................................................................................................... 39
  10.1 GUI Tools .................................................................................................................................................. 39
  10.2 GUI Design............................................................................................................................................... 39
  10.3 Future Work and Recommendations........................................................................................................ 40

Appendix I - XML Protocol
  1 XML skeleton .................................................................................................................................................. 41
  2 Driver ............................................................................................................................................................ 42
  3 Goer .............................................................................................................................................................. 44
  4 Follower ........................................................................................................................................................ 47
  5 Explorer ....................................................................................................................................................... 48
  6 Get_map......................................................................................................................................................... 50
  7 Get_x3d_map............................................................................................................................................... 50

Appendix II - XML File Format
  1 XML Format.................................................................................................................................................. 51
  2 Data Tags...................................................................................................................................................... 51

Appendix III - X3D File Format
  1 X3D Definition ............................................................................................................................................ 54
  2 File Format .................................................................................................................................................. 54
  3 Tag Layout................................................................................................................................................... 55
  4 Tag Definitions............................................................................................................................................. 55

Appendix IV - VRML97 and Basic 3D Modelling
  1 Introduction ................................................................................................................................................ 62
  2 Modelling .................................................................................................................................................... 63
  3 Example....................................................................................................................................................... 69
  4 References.................................................................................................................................................. 70

Appendix V - Projectivities
  1 Perspective Projection to the Image Plane.................................................................................................... 71
  2 Homogenous Coordinates............................................................................................................................ 71
  3 Plane-to-Plane Projection............................................................................................................................. 72
Contents

4 Plane-to-Plane Mapping ................................................................. 73
5 References ...................................................................................... 73

User’s Manual
1 Introduction ...................................................................................... 74
2 Disclaimer ....................................................................................... 75
3 Start-up ............................................................................................. 75
4 Control Panel ................................................................................... 76
5 Manual Drive .................................................................................. 76
6 Explore Area ................................................................................... 77
7 Follow Me ....................................................................................... 78
8 Road Follow .................................................................................... 78
9 2D Map ........................................................................................... 79
10 3D Map ......................................................................................... 84
11 Video Camera ............................................................................... 87
12 Set Max Speed ............................................................................... 88
13 Change Settings .......................................................................... 88
14 Collision Avoidance ................................................................. 88
1 Introduction

This section provides the background information about the project and the abilities of the robot. It also describes the purpose of the project (i.e. why it was initiated) and finally it presents an overview of the content of this thesis.

1.1 Background

CAS\(^1\) has, in a collaboration with FMV\(^2\), developed a robot called Pluto. Its primary purpose is reconnaissance in an urban (outdoor) environment but it is constantly being upgraded and modified to handle different kinds of missions. Pluto is based on a 4WD\(^3\) robot platform from iRobot called ATRV\(^4\) (fig. 1), which is a rugged all-terrain robot with low centre of gravity, big knobby tires, high ground clearance and water-resistant enclosure (although some of the extra external equipment does not like water).

![Figure 1 - Pluto, a modified iRobot ATRV](image)

---

1. Centre for Autonomous Systems, NADA, KTH
2. Swedish Defence Materiel Administration
3. 4-Wheel-Drive
4. All Terrain Robot Vehicle
The robot can handle a payload of over 100 kg and is equipped with both tactile and sonar sensors as well as a laser scanner. Communication with the robot is done using a w-lan card (802.11). The robot runs a Linux server on a dual Pentium II 800 MHz computer. Beside the OS and all the programs that define Pluto’s behaviour and abilities, the server also hosts a web server which is responsible for relaying images (or frames) from the network digital video camera connected to, and mounted on top of, the robot. The robot is also equipped with a differentiated GPS system. For more detailed information, see the iRobot web site: www.irobot.com

Pluto can handle tasks like autonomous map-making of an unknown environment and autonomous navigation between two points using the internal map. The user can drive the robot directly using the PDA GUI (or a connected joystick). Pluto can also follow the user around autonomously (carrying equipment and/or supplies). Two functions currently being implemented in parallel to the GUI project is autonomous road following and a robotic arm. As mentioned earlier, the robot could be adapted to handle many different types of missions. For instance, adding chemical sensors it could be used to detect and locate toxic waste or other harmful substances. Given a trolley, it could transport wounded out of hostile environments, etc.

1.2 Purpose

The old PDA GUI, written in Personal Java¹, runs on a PocketPC² sized PDA with a resolution (i.e. screen size) of 240x320 pixels. The (low) resolution makes it difficult for a user to grasp the extent of an entire map. Therefore, one of the objectives of this project was to develop a new GUI using a different PDA with a bigger screen (and higher resolution) and take advantage of this new format.

The main purpose was to investigate if the modelling of the map as a 3D world could help the user to get a better understanding of the surrounding environment and the composition of the map. Is it possible to do 3D on a PDA in general, and on the given PDA in particular? There was also an interest to examine the use of the video camera. For instance: could it be used in parallel with the manual-drive control to enable the user to drive the robot when it is out of sight?

The project encountered several other interesting questions during development:

- What is the best tool for PDA GUI development regarding productivity, portability and performance?
- What are the general GUI design issues for PDAs?
- How should the GUI be designed for maximum utility and usefulness?
- What are the performance issues?

Developing software for a PC/laptop is a lot simpler than for a PDA. On a PC there are (almost) no performance issues and it has a (real) keyboard and a big screen. But the penalty for this extra performance and screen size is limited battery time, increased weight and reduced portability. Most laptops are also much more sensitive to shocks and blows than

¹. The specification for Personal Java 1.2 is compliant with JDK 1.1.8.
². Section 3.1 on page 8
PDAs. It is therefore important to investigate how to obtain a good balance between visualization and performance, and discover what compromises that need to be made.

1.3 Content Overview

This section briefly describes the content of the different sections/chapters found in this thesis.

1. Introduction
This section provides the background information about the project and the abilities of the robot. It also describes the purpose of the project (i.e. why it was initiated) and finally it presents an overview of the content of this thesis.

2. Assignment, Problems, Goals and Restrictions
This section describes the project assignment and defines the various problems within it. It also describes the goals and the restrictions/constraints I have made.

3. PDA Definitions
This section explains some of the terms used throughout this thesis and in the PDA world in general. Hopefully it will sort out some of the questions and confusions about PDAs. The last subsection defines the properties and drawbacks of the Siemens MOBIC T8 PDA.

4. GUI Tools
This section covers the evaluation of different tools for GUI development.

5. GUI Design
This section briefly describes the old GUI, its design goals, and the impact it had on the new design. Then the design of each component of the new GUI, the ideas behind the design and the problems with it are presented.

6. Implementing the GUI
This section covers the issues that occurred while implementing the GUI. It also discusses some system oriented design strategies.

7. Graphics Library
This section covers the design of a simple 3D browser and the graphics library used in the 3D Map window.

8. Evaluation
This section covers the evaluation of the GUI.

9. Summary
This section begins with a quick review of the problems. It is followed by a summary of how they were solved and the conclusions that could be drawn from this project. And finally some thoughts about future work and modifications are presented.

Appendix I - XML Protocol
This appendix covers the XML protocol used for communication between the PDA client and the robot server.
Appendix II - XML File Format
This appendix specifies the XML file format used by the 2D map.

Appendix III - X3D File Format
This appendix specifies the X3D file format used by the 3D map.

Appendix IV - VRML97 and Basic 3D Modelling
This appendix covers the basics of 3D modelling using VRML97.

Appendix V - Projectivities
This appendix covers the basics of projectivities, perspectivity and plane-to-plane mapping.

User's Manual
This appendix is written as a stand-alone document for the inexperienced user (or for reference). All the different functionality of the GUI is presented. The manual contains many screenshots that could be useful while reading this thesis.

NOTE: In the thesis I sometimes refer to “we” (i.e. we agreed upon) meaning me and my supervisor. In the GUI Design section, “we” is me and Carl Lundberg at CAS who worked with the old GUI. And in the Graphics Library section, “we” is used to explain the purpose of the code and do not refer to anyone in particular. I have tried to give credit to others for work I have not done and ideas I have not thought of myself.
2 Assignment, Problems, Goals and Restrictions

This section describes the project assignment and defines the various problems within. It also describes the goals and the restrictions/constraints I have made.

2.1 Assignment

The assignment was to write a new GUI for Pluto that could run on the new PDA platform that had been purchased by CAS (fig. 2). The GUI should be capable of utilizing all the current functionality of Pluto, including video relay which was not present on the old GUI. One of the objectives was to improve the manual-drive control to make it more intuitive and also to combine it with the video to further increase the utility (i.e. usefulness) of the GUI.

The main objective was to visualize the internal (2D) map as a 3D model. The hypothesis we wanted to try was that a 3D model would help the user to better understand the 2D map. If the visualization could not be done to satisfaction (i.e. it was to slow or could not be of use in any constructive way), then at least the 2D Map window should be fully functional and easy to use.

In order to solve these tasks the first thing to do would be to search for, and evaluate, different tools for GUI development for a PDA system. So we agreed upon making this task a part of the project assignment as well.

2.2 Problem Definition

The following questions need to be answered. The reason/origin for some of them are explained later in this report.

- Should the GUI be written in C++, Java or perhaps in a third language? This is a question of productivity, portability and performance.
- Can a good GUI tool be found that is easy to use, portable, and has the speed required to implement a 3D browser on a PDA?
- Is it even possible to implement a basic 3D browser on the PDA? Is it fast enough?
- What are the restrictions and limitations of the given PDA? What are the issues with PDAs in general?
- Can a 3D model increase the user’s understanding of the constructed map?
Since the 3D model is built from 2D data, the height needs to be approximated. Will a predefined height suffice in order to make the model more useful and realistic?

How accurate must the 3D rendering be to be considered useful? Will a simple wire-frame rendering suffice?

Can the 3D model be used instead of the 2D map all together?

Does video provide any useful information? Does it enhance the GUI?

How fast (measured in frames-per-second) must the video be updated to be useful?

How should the GUI be designed in general? And how should the manual-drive control be designed in particular? It is important that the GUI remains easy to understand and operate.

How slow can the GUI components get, before the user finds them to be badly designed and non responsive?

2.3 Goal

The goal is to answer and solve the questions and problems stated above, and in this process construct and implement a new GUI that is simple and intuitive to use and understand. The user should be able to perform the following tasks:

- Define an area to be explored (and tell the robot to explore it).
- Drive the robot manually to a new location.
- Send the robot to a new location using the 2D map.
- Toggle “Follow Me” mode and “Collision Avoidance” on/off.
- View the internal map in 2D and 3D.
- View the video camera stream.

A successful project includes a good analysis of the system and its components regarding the balance between 2D/3D visualizations and the quality of these visual components.

If possible, the GUI should be portable and relatively platform independent. Then it could at least be implemented on a laptop or another platform (without too much extra work) if the performance of the GUI was not adequate on the PDA.

2.4 Restrictions

A project like this can become very large and extensive unless restrictions are made. Here are the ones I have made:

- Only evaluate GUI tools that have potential to work and fulfil the requirements.
- Only implement the most basic features of the GUI in general. Specializations and improvements can be made later on if the design is sound and modular (i.e. follows the OOP\(^1\) standards).

---

1. Object Oriented Programming
• Restrict the 3D browser to a subset of VRML97/X3D nodes. There is no need to make full compliance with the VRML97 standard.

• Keep the 3D browser as simple as possible. This restriction is closely tied to the restrictions implied by making the GUI user friendly and retaining high performance on the PDA.

• Design the GUI without making a major initial survey and evaluate the GUI only by asking co-workers at CAS. A full scale analysis is outside the scope of this project and my masters thesis.

---

1. See Appendix IV for more information about VRML97.
3 PDA Definitions

This section explains some of the terms used throughout this thesis and in the PDA world in general. Hopefully it will sort out some of the questions and confusions about PDAs. The last subsection defines the properties and drawbacks of the Siemens Mobic T8 PDA.

3.1 Windows CE

The operating system Windows CE is not just a port of Windows for PDAs, it is a bottom-up rewrite. While it may look more or less like plain old Windows, it is completely different compared to Windows 3.x, Windows 9x, and Windows NT. The designers of CE decided early on to focus on portability and small size in the design of CE. Shrinking a Pentium machine to the size of a deck of cards and still remain reasonable battery-life is not possible, so Windows CE was written to work for alternative processors with very low power consumption (like the StrongARM, SH3/SH4, and MIPS). While Windows CE is not locked into any particular form factor and is capable of running on anything from embedded micro controllers to cell-phones, two main form factors have become predominant, the Handheld PC (HPC) and the PocketPC (PPC) (Hattan, 2001).

“The market for HPCs based on Windows CE unfortunately is in decline.” (Hattan, 2001). This statement would explain why there is not much out there for the HPCs. Practically all applications and tools are made for PPC today.

In response to the instant success of the Palm Computing platforms, Microsoft introduced the Palm-size PC but the platforms (manufactured by many different vendors) were savaged by critics for being overly complicated and clumsy to use. The standard Windows interface, while it worked well on larger screens, was tight on HPCs (640x240) and even became difficult on a small 240x320 screen. In the year 2000, Microsoft released a new version of the Windows CE interface, redubbed “PocketPC”. While internally it was basically the same Windows CE as the earlier versions, the user-interface was retooled to work better on a tiny screen (Hattan, 2001).

So, in short: There are two different versions of Windows CE. The HPC version which runs on slightly larger PDAs of various sizes where the most common screen size is 640x240. And the PPC version which only runs on the small pocket-sized PDAs with a resolution of 240x320. Although they are basically the same, some programs do not run on both versions. And most games do not run at all on the HPC because it lacks the GAPI library.

3.2 Graphics API

The Graphics API1 (GAPI) was written by Microsoft to allow the developers of PDA games to have direct access to the display frame buffer on PPC platforms and on some HPC platforms. The GAPI DLLs2 are only available for Casio, Compaq and HP3 devices however (Hattan, 2001).

1. Application Programming Interface
2. Dynamic Link Library. (External) code that can be executed dynamically from a program at run-time.
3. Hewlett-Packard
3.3 Siemens MOBIC T8 Specification

MOBIC stands for Mobile Industrial Communicator and is a Handheld PC running Windows CE 3.0. It has a 8.4” screen with a resolution of 800x600 pixels and 256 colours. The device itself is heavily protected and can withstand a two meter drop onto a concrete floor without breaking. It is also water splash-proof and dust resistant (Siemens, 2003).

With a weight of 1.7 kg\(^1\) this seemed like a good platform for the Pluto GUI project. And it would have been, if the goals of the GUI had not been to implement video and 3D visualization. The lack of GAPI combined with the large screen resolution resulted in very poor graphical performance. And besides this the overall system resources of this device was not as high as expected. Using the w-lan card reduces the already low system speed considerably. This PDA seems to have been designed for data sheets and basic graphics like diagrams and flow charts only.

Yet another problem is the touch screen that needs to be recalibrated frequently (between uses and users) and the fact that it sometimes fail to respond on taps. There is no particular way to get the screen to start registering taps again. It comes and goes which is very disturbing. If this is a fault or a flaw is hard to say, but we hope it is a fault that can be fixed.

The PDA also has very low legibility outdoors or in a bright room, but this is a common problem for most touch-screens (and TFT screens).

\(^1\) A Compaq iPAQ H3850 (PPC) weighs 190 g.
4 GUI Tools

Based on the given assignment I started to search on the Internet for tools that could help me build the GUI and a 3D browser. The old GUI was written in Java 1.1.8 (which means that it could only use the old AWT\(^1\) classes) so to avoid having to rewrite the whole GUI I first started to look for Java 3D solutions. Unfortunately, most of these solutions were based on OpenGL\(^2\) which is not present on any PDA device (at least not at present date). I did find a pure Java solution but its performance was, as one could expect, not sufficient. So I turned to C++ and started to look for GUI tools for the Windows CE platform, keeping platform independency in mind. Here follows a brief summary of the results and the decisions made, in a more or less chronological order.

4.1 Java Based Tools

Since Java3D was out of the question due to the lack of OpenGL drivers for my PDA (or any PDA for that matter) I started to look for fast and light Java solutions so that I could at least reuse most of the old GUI Java code. Writing GUI and network code in Java is also relatively easy since Java has an extensive collection of libraries. And, it is very portable and platform independent. So Java seemed like a good place to start, but unfortunately none of the solutions proved to be fast enough.

4.1.1 SuperWaba

Waba is a programming platform for small devices. Waba defines a language, a virtual machine, a class file format and a set of foundation classes (Wabasoft, 2001).

SuperWaba first began as Waba and with addition of new classes came the name change to SuperWaba. Waba was originally developed for cell phone interfaces and slowly became one of the dominate languages to use for programming PDAs. It is basically a very limited version of Java although it is not Java. SuperWaba has no relation with Sun Microsystems. Its syntax is a strict subset of Java. This makes for very easy programming for those who already know Java. SuperWaba is optimal for PDAs because of its design. SuperWaba’s libraries only include features that were deemed to be necessary to run applications efficiently on PDAs. (Catanzaro, 2002).

“After studying Superwaba for about 3 weeks I believe that Superwaba is a very versatile language for PDAs” (Catanzaro, 2002).

This seemed like a good and useful Java clone. Unfortunately I never got it to work on the PDA (although it should).

\(^1\) Abstract Window Toolkit, predecessor to the Java Swing classes. Contains Java GUI components.
\(^2\) OpenGL is a cross-platform standard for 3D rendering and 3D hardware acceleration.
4.1.2 Ewe

The Ewe system is a cross-platform, write-once run-everywhere programming system. It allows you to write programs that can be run unaltered on any Windows desktop system, Windows CE system (including Pocket PCs, Handheld PCs and WebPads), and any other system that supports a Java 1.2 run-time environment. (Brereton, 2002a)

On average, the Ewe VM starts three times faster and uses three times less memory than the PersonalJava VM (Brereton, 2002b).

Ewe is an extension of the original Waba VM (Brereton M, 2003). But unlike Waba and SuperWaba that target small devices, the Ewe VM is targeted at devices at the advanced PDA level and higher. That is to say, a 32-bit OS with at least a 160x160 touch screen, and at least 2 MB of available program memory (Brereton, 2002b).

Ewe, unlike SuperWaba, came with an easy to use program builder called Jewel (packed in a .jar file). With the Jewel GUI it was easy to build an Ewe project. One could even target a specific platform, like MIPS/HPC, and create a .exe file with the VM included which was a very nice feature. This tool looked very promising.

4.1.3 Dog Gui

Dog Gui is a lightweight, high-performance Java GUI toolkit. It is designed to replace the standard AWT user interface components such as buttons, textfields, and lists (Burdess, 1999).

“I have run several fairly thorough tests on the dog.gui componentry, comparing it against Swing and normal AWT components. It has proved to be 2-5 times faster than Swing to construct components, marginally faster to lay them out, and approximately 2 times faster to paint them. AWT components take a longer time to construct and lay out but paint faster.” (Burdess, 1999).

I tried this library and it works like any other Java library and could be a good complement to the AWT classes if the slightly different component design is acceptable.

4.1.4 CyberVRML97

CyberVRML97 for Java is a development package for VRML97/2.0 and Java3D programmers. Using the package, you can easily read and write the VRML files, set and get the scene graph information, draw the geometries and run the behaviours easily (Konno, 2002).

Using this library, and by modifying the sample 3D browser, I learned about the VRML97 structure and 3D coding. The stripped down pure Java browser (without shades and lighting properties) was tested both on a regular PC and on the HPC. The PC performance was acceptable on a 500 MHz Celeron but the browser literally froze on the PDA. This proved what I had feared all along, Java can not be used for graphics on a PDA.

I used the CyberVRML97 basic VRML browser as a skeleton for the GUI 3D browser.
4.2 C++ Based Tools

Since the pure Java solutions were too slow, I had to turn to C++ solutions. The OpenGL restriction still applied so I will not review any such tools although I tried a few on the PC platform. The main problem was that the device did not have support for GAPI so the task was narrowed down to finding a good GUI tool and a graphics library that wraps GDI\(^1\).

4.2.1 WxWindows

WxWindows gives you a single, easy-to-use API for writing GUI applications on multiple platforms. Link with the appropriate library for your platform (Windows/Unix/Linux/Mac) and compiler (almost any popular C++ compiler), and your application will adopt the look and feel appropriate to that platform (Smart, 2003).

This looks like a very promising tool for GUI building in C++. For instance, you can develop and then cross-compile Windows applications directly from Linux. Unfortunately the WxWindows for Windows CE 3.0 port has not yet (2003-03-21) been completed.

4.2.2 ATL/WTL

Without getting into too much details, there is only one way of creating windows in Windows, and that is with the Windows platform SDK\(^2\) also known as the Win32 API.

Programming using only the SDK becomes very difficult since it has no object design and you are basically on your own trying to find the correct window message to send to a component to achieve the wanted behaviour. Also, the code becomes hard to read and modify afterwards. That is why Microsoft created the MFC\(^3\) which was designed according to a “document-view” architecture. This architecture provides a logical separation between an application’s data and the representation of data. Using VC++\(^4\) the MFC AppWizard helps the user to create his application using a drag-n-drop interface. This might be acceptable for some people, but I think it is bad because you give up the design of the objects to MFC. There is also a price to pay for using the MFC architecture and that is that one must rely on (and link to) the MFC class library which makes it impossible to write a tiny program. Luckily there is another alternative, Active Template Library (ATL).

Although ATL is known primarily for its COM\(^5\) support, it also provides several classes that simplify Windows programming. These classes, like the rest of ATL, are template-based and have very low overhead (Park, 1999).

ATL only manages the most basic UI components and are considered too low-level for real windowing programming. But the same team that created ATL has also constructed the Windows Template Library (WTL) which extends ATL.

---

1. Graphical Device Interface, standard Windows graphics API.
2. Software Development Kit
3. Microsoft Foundation Classes
4. Microsoft Visual C++, the most common compiler for Windows.
5. Component Object Model, a software architecture to build component-based applications.
WTL provides a lightweight yet comprehensive application framework, which automatically furnishes applications based on it with many desirable facilities. The goal is something less than the impenetrable MFC framework, and something easier than starting to code WinMain manually i.e. using the SDK (ClipCode, 2000).

WTL also comes with a WTL AppWizard for those who like wizards so WTL/ATL is clearly a better choice than the old MFC. For more information about WTL, I suggest you read the documents referenced above and especially the ClipCode WTL guide.

ATL/WTL seemed like the only possible option for GUI development on WinCE and combined with PocketFrog to handle all the heavy graphics it turned out pretty good.

4.2.3 PocketFrog

“PocketFrog is *THE* game library to rapidly write blazing fast games on the Pocket PC platform. It is implemented in C++ with an object oriented design. If your application needs to harness the raw power of your Pocket PC graphics, PocketFrog is for you.” (Tremblay, 2002).

PocketFrog wraps GAPI if present and GDI otherwise and provides all the basic functionalities of a standard graphics library like:

- Graphics primitives (blit, line, rectangle, circle, etc.)
- Clipping rectangle
- Alpha blending and colour masking
- Image loading
- Etc.

PocketFrog supports all PPCs and HPCs as well as desktop PCs since it emulates GAPI when it is not present. No only does this have a positive affect on portability (among Windows platforms) but it also means that the application can be developed on the desktop and then recompiled for the target PDA for final testing. PocketFrog is based on ATL.

4.3 Summary

As it turned out, Java based tools could not be used to accomplish two of the goals of this project: a 3D browser and streamed video. The former because PDAs do not have hardware accelerated 3D support (i.e. OpenGL) and the latter because the platform was too slow. Since the Siemens platform also lacked GAPI support, many of the C++ GUI tools did not work. I finally found PocketFrog which works as a C++ GDI wrapper for all the devices that does not support GAPI, including regular Win32 platforms (i.e. desktops). This turned out to be very useful since then I could do all the testing and development on a desktop and later recompile to PDA for the final testing. Since PocketFrog is based on ATL it was easy to extend the library with regular ATL components and to modify the existing PocketFrog components.

1. At least not at present date. But there will be PDAs with special 3D chips available in the near future.
5 GUI Design

This section briefly describes the old GUI, its design goals, and the impact it had on the new design. Then the design of each component of the new GUI, the ideas behind the design and the problems with it is presented.

5.1 The Old GUI

The old GUI was partially developed by a student (Carl Barck-Holst) at NADA\(^1\), KTH\(^2\) as part of a “larger individual course” in computer science and a Ph.D student at CAS (Carl Lundberg). Their aim was to design a multi user-level interface and to make it as easy and intuitive as possible (Barck-Holst C, 2002). No previous knowledge should be required to operate the robot and once the user had established a certain amount of confidence about the system, he could switch to the advanced user mode and receive more detailed information. Information that would otherwise distract and scare the novice user. They worked according to an assumption that the robot would primarily be used by the military during joint international operations and therefore took some of the swedish military’s requirements for international service, as a user profile:

- Sex: Male or female.
- Age: 20-40.
- Education: At least some form of upper secondary school\(^3\) education.

From that profile they deduced that the user had some experience about computer systems. And by some experience they meant the ability to use Windows based applications like MS Office, surf the Internet with a browser and use web based e-mail applications like Hotmail. This implies that the user has an intuitive feeling for the following concepts:

- The function of common graphical components like windows, buttons and dialogs.
- The layout of these components and other frequently recurrent functionalities.
- The occurrence of delays in computer networks.
- The nature of transactions. Transactions here means the process of sending an action-command and receiving a confirmation about the outcome of that action.

Based on these conclusions they constructed a GUI written in Java primarily intended to be used on a Compaq iPAQ, which is a PPC\(^4\) PDA. They also designed an XML protocol\(^5\) to be used for communication between the PDA (client) and the robot PDA server. The PDA server is responsible for calling the correct robot module which in turn calls the appropriate control program through a CORBA\(^6\) interface. Each XML message sent to the PDA server is answered with an XML reply.

---

2. Royal Institute of Technology, Stockholm, Sweden.
4. Section 3.1 on page 8
5. See Appendix I for more info about the XML protocol.
The user feedback and evaluation of the old GUI resulted in the following remarks, conclusions and suggestions:

- In general, use more feedback from the system. Add more response to actions taken by the user. It is very important that the user feels that the GUI is working and processing the commands.
- Add some sort of speed indicator in the Drive\(^1\) module.
- Show progress during area exploration.
- In general, a GUI should adapt to the user’s experience level (novice/advanced/etc.).
- It is hard for the user to picture the surroundings just by looking at the generated map without some prior knowledge about the area.

### 5.2 The New GUI

The new GUI (from now on referred to only as “the GUI”) written by me is a complete rewrite of the old GUI. Since the GUI is written in C++ I could not reuse any of the old Java code. And since my object design\(^2\) and GUI design is totally different from the old GUI, I used the PDA server source code and the XML protocol specification as reference rather than the Java client code. So basically, the only impact of the old GUI project on the new is the use of the XML protocol and the PDA server. Of course, the old GUI and its feedback served as a starting point for the discussion that led to the new design. Since both GUIs use the same protocol there are many similarities but most of the underlying implementation as well as the (visual) design are different. And rightly so, since the old GUI targeted a small PDA (240x320) and the new GUI targets a large PDA (800x600). The need for speed and performance also affected the design, since it restricted the tools\(^3\) and GUI components that could be used. Both GUIs try to achieve three important things though: portability, extensibility and usability.

The major difference in design strategies is that I have not taken multiple user-levels in account. The reason for this is that some of the new functionality require more from the user than the old GUI did. And there is no point in having both simple windows and slightly more difficult (or slightly less simple) windows at the same time. But this does not mean that I have abandoned the demand for simplicity. In some cases the new GUI is even more simple to use than the old due to the added feedback and alternative design. As the GUI evolves over time it might be useful to add multiple user-levels again. For instance, there could be one debug user who receives maximum feedback, one normal and one advanced for those new and challenging functions that most users can do without. Therefore, the support for different user modes has been included in the settings object\(^4\) although it is not currently used anywhere.

The GUI should satisfy the goals\(^5\) of this project and try to improve and/or avoid some of the flaws discovered in the old GUI.

---

1. Corresponds to the Manual Drive window in the new GUI. Section 5.2.2 on page 17
2. The object oriented programming design of the GUI.
3. Section 4 on page 10
4. Section 5.2.8 on page 22
5. Section 2.3 on page 6
For information about using the GUI, read the User’s Manual which is the last Appendix of this thesis. There you will also find many screen shots of the GUI which might help you to understand this section better. The following subsections will cover the ideas and motivation for the design of the different GUI components. The graphics are discussed in the next section.

5.2.1 Control Panel Window

The Control Panel window (fig. 3) is the main window of the GUI. Its purpose is to unify all the different components and functions of the GUI and to be used as a launching platform.

The design is based on the following ideas, requirements and restrictions:

- The components should be grouped and easy to locate.
- The Control Panel window should be easy to extend if new components were added to the GUI, without affecting the old layout too much.
- The user should be able to control the GUI with his index finger and not be dependent on a tiny pencil (which might easily be lost in a real world situation) and a fine calibrated touch-screen.

The first two ideas imply that the components should have a tree layout. One example of a tree layout is a common menu bar, found in almost all window applications. Arranging the components hierarchically like this increases the overall utility/usefulness of the GUI and gives the user a schematic representation of how the components are related. Also, each component requires less individual description since it inherits the properties of the group. This design also solves the second requirement above: that the GUI should be extendable without major redesign and without compromising the users understanding.

The third idea can not be implemented with a menu bar because menus are to small and require a small pointer (i.e. a pen) to be used effectively. So I chose a layout consisting of labelled buttons and group boxes. The buttons where sized and placed/laid out in a way that the third requirement was fulfilled. The buttons where grouped together using group boxes to create the following three main function groups:

1. Section 7 on page 27
2. A group box is a graphical interface component consisting of a simple wire frame and a frame title.
• *Tools*, which represents all the robot functions.
• *Show*, which represents all GUI visualization windows.
• *Settings*, which represents all the customizing and settings components.

One could argue that sending the robot to a new location using the map is a robot function and should be in the *Tools* group. But the 2D *Map* also provides major feedback not only for the map itself but for robot status, robot position, screen shots, etc. and that is why it is located in the *Show* group.

### 5.2.2 Manual Drive Window

The *Manual Drive* window holds the interface for driving the robot directly, almost like an RC\(^1\) model. One of the goals was to increase the feedback of this control and make it more intuitive. The old GUI had, in advanced user mode, three text labels ([fig. 4](#)) to indicate the robot status.

The large screen gave me the ability to write the whole names. I also added a label with the maximum speed setting. This information could be grouped together in the middle of the screen ([fig. 5](#)) where it did not distract the user as much and this is one of the reasons why I dropped the user levels.

[![Figure 4 - Drive control (ill.)](#)](#)  

On the left side of the window is the video frame and a compass needle, because it seemed useful to know the current direction of the robot. All the components have been placed in

---

1. Radio Controlled
A PDA Interface for 3D Interaction with an Outdoor Robot

a way that maximizes the use of the large screen and so that the window as a whole feels symmetric. The video frame and all the buttons have a black border that gives the illusion that they belong to the window. The background has a neutral light grey colour, much like a regular dialogue window (but it is actually a PocketFrog\(^1\) window). The only reason why the colour of the text is white is because the same component for writing text is used in the 2D Map window which has a black background. Due to lack of time I did not write a label component with variable font colour, but for common dialogue consistency the text should have been black in this window.

This is one of the few windows of the GUI that assumes that the user is right handed. A possible extension of the GUI would be to add a right/left hand option to the settings object that alters the layout.

We discussed different solutions for solving the driving interface issue. Even though the speed is displayed explicitly on the screen (in digits), it might be a good idea to indicate it visually to increase the understanding and feedback of the control. We agreed to use partially filled arrow buttons, representing the zero speed with an empty arrow and maximum speed with a full arrow. This has now been implemented even on the old GUI.

Another idea that we discussed was the best way of controlling the robot. The following alternatives were considered:

1. Use the distance from the stop (centre) button to set a new speed in the given direction.
2. Use one horizontal and one vertical slider instead of the four buttons. The stop button is still needed for making a quick stop.
3. Use no buttons at all. The speed is relative to the distance from the centre.
4. Use four new diagonal arrow buttons.
5. Use the up/forward arrow to cancel/stop any current rotation/turning speed.

The first idea would have been very nice indeed. But due to the characteristics of the PDA's touch screen\(^2\) this would have created an interface that is more difficult to understand and control since it requires the screen to be fully calibrated. And besides this, the design would not fulfil the initial requirement that the interface should be usable without a pencil.

The second idea has its merits. It allows the user to quickly set a speed in the given direction and it can be controlled with the use of a finger. The only argument against this solution is that it might not (initially) be as intuitive as the five-button interface. And it could be a problem visualizing the feedback.

The third idea raised a lot of questions. For instance, how should the distance components be calculated. How should the interface, and the feedback, be visualized. It is an interesting idea that should have been evaluated with a user test group, but this was not within the scope of this project.

---

1. See Section 7 on page 27 for more info.
2. Section 3.3 on page 9
The fourth and fifth ideas were dropped because they would have created an interface that would initially be hard to understand. And furthermore useless if you consider the way most users tend to use the interface, which is: first they turn, then they stop, and then they go forward. With the new feedback system (filled arrows) it would be easy to initiate a turn and then stop/decrease turning without actually having to stop the entire robot. With the old design (without feedback) this was not the case since the user did not know how fast he was turning and either did not stop the turning fast enough or over compensated so the robot started to turn the other way.

The designs all have their pros and cons and the selected one (common five-button interface) was chosen primarily because it was considered to be the simplest and most intuitive one, but also because I thought it would look the best.

5.2.3 Explore Area Dialog

The Explore Area dialogue (fig. 6) helps the user to define an area for the robot to explore autonomously. Ideally, this area should be defined on top of the map in the 2D Map window, but since it was outside the scope of the project I choose the simplest implementation. Having it as a dialogue also provides a way to communicate with the user. Besides giving information about how to use the dialogue, it also explains how the exploration and the 2D Map window works during explore. It has been designed like a regular dialogue with the information on top, actions in the middle, and buttons at the bottom.
5.2.4 2D Map Window

The 2D Map window (fig. 7) will probably be the most used window of the entire GUI since it holds most of the robot functionality. Not only does it show the map, but also the current status, coordinates and given destination. From within this window, the user can tell the robot to go to a new location, explore an area, stop everything and/or resume the exploration. The user can also control the way the map is displayed, moving the map, zooming, centre on the robot or centre on any given point selected. All these actions are represented by button icons, located as a tool bar at the bottom of the window. Various information is displayed in the four corners of the window, where they take up the least attention from the rest of the map.

![Figure 7 - 2D Map (This screen shot has been converted to grayscale and then colour inverted)](image)

There is a conflict of design strategies here, which will be discussed at the end of the next subsection because it is also closely related to the 3D Map window.

5.2.5 3D Map Window

The 3D Map window (fig. 8) is a basic 3D browser, which renders the map in three dimensions according to VRML97 standards. Due to the poor performance of the PDA a compromise between window size and rendering speed had to be made. This resulted in an initial size of about 400x400 which later became 448x400 because I needed 14 buttons\(^1\) for the interface. The first button resets the browser, which is very useful. The second button stops the rotation of the word and is a requirement for the user to start to move around in the world. The key issue here is that the arrow buttons move the viewpoint (i.e. the position of the user) and not the world itself. The viewpoint can be moved along, or rotated around, the x, y or z axis. The rotation can be a bit hard to understand for the inex-

---

1. Which will be 15 buttons when the flat rendering is implemented. Each button is currently 32x28 pixels.
A PDA Interface for 3D Interaction with an Outdoor Robot

experienced user but it is standard in most 3D browsers. And with a little bit of practice the user will learn to navigate in the 3D world.

The design idea of the buttons was to make them as simple as possible. An arrow pointing to the left would mean that the viewpoint would move to the left. To move the viewpoint forwards and backwards required a complicated shape of the arrow that was not possible to create without increasing the button size. So I created a combined zoom and arrow button (fig. 9) that was also used in the 2D Map window which gave the 2D/3D windows some consistency even though the user is not actually zooming in the 3D world (although one could say that in the 2D Map the user is moving the viewpoint closer to the map when zooming).

The rotation buttons were designed to indicate around which axis the rotation would take place. Even though these buttons might not be intuitive to most people, they are sufficiently different so that the user will learn to distinguish between them with a little practice. And one can always test to see what will happen by pressing a button.

In order to keep the window size small (for better performance), the buttons were made small. This resulted in a conflict of design strategies because one of the initial goals was that the GUI should be possible to control without a pencil. But these buttons are so small that they require the PDA to be perfectly calibrated to be navigated with the tip of the index finger and not even then, depending on the size of the user's finger, is it guaranteed to work without accidentally pressing outside the button (sometimes this even happens with the pencil). For consistency, I used the same arrow buttons in the 2D Map window so the same problem arose there as well. The only thing that supports this design is the fact that both these windows can move the viewpoint to a new location by pressing somewhere on the screen. The 2D Map also has the Goto command which requires a set of coordinates. These coordinates are selected by pressing somewhere on the map and this should be done using a pencil to achieve some accuracy.
The design of the 3D browser and graphics library will be covered in Section 7 on page 27.

5.2.6 Video Camera Window

The Video Camera window simply shows the video stream and two buttons: one for taking a new snapshot and one for launching the snapshot viewer/organizer. The window has the same size as the video plus some extra space for the buttons, all in an effort to keep the frame rate as high as possible.

5.2.7 Change Speed Dialog

The Change Speed dialogue (fig. 10) allows the user to set a new maximum speed. The current maximum speed is displayed as a label in the top right corner and the new speed is displayed directly below. The speed is changed with a slider along the right side of the window.

![Figure 10 - Set Max. Speed](image)

This design assumes that the user is right handed, and does not cover any information with his hand while setting the new speed. As with the Manual Drive window, a possible extension would be to allow the design to change depending on a right/left hand option in the settings object.

5.2.8 Settings Dialog

The Settings dialogue has not been implemented yet, but should contain standard dialogue components to change all the options in the settings object (setting.ini).

5.2.9 Other Components

The remaining components include Collision Avoidance, Follow Me and Road Follow. The first two components do not require much user interaction and are basically just toggle on and off functions, hence they have been implemented as buttons in the Control Panel window. The feedback (i.e. the new state) is presented in a standard window message box.

Road Follow has not yet been implemented into the robot system and the interface has not yet been designed. How this feature will be implemented into the GUI remains to be seen, but some minor preparations have been made.
5.3 Summary And Conclusions

The old GUI was designed to be as simple and intuitive as possible and used multiple user-levels to achieve this. During its evaluation it turned out that the users wanted more feedback from the system to indicate if it was working or not. They also requested a larger screen since it was hard to grasp the full extent of the map in the small 240x320 format.

The new GUI tries to solve the problems with the old GUI, while adding a 3D browser and video relay, and still remain simple and intuitive. Unfortunately, due to the shortcomings of the platform, some sacrifices had to be made. To achieve acceptable performance of the 3D browser, I had to keep the window size small and this restricted the size of all the button icons. Then the characteristics of the touch screen made it hard to target those buttons unless the screen had been perfectly calibrated. This also affected the design of the manual drive interface since any form of relative speed input, where the speed is relative to the distance from the centre, would be very difficult to operate without accurate input.

Therefore the GUI did not turn out as good as I had hoped. The “Evaluation” on page 31 and the “Future Work and Recommendations” on page 40 will go deeper into this.
6 Implementing the GUI

This section covers the issues that occurred while implementing the GUI. It also discusses some system oriented design strategies.

6.1 PocketFrog and ATL

The GUI is a mixture of ATL, WTL and PocketFrog. The only WTL component used is the slider in the Set Max. Speed dialogue. ATL is used for the dialogs and the main Control Panel window. The windows that required graphics support were created with PocketFrog which is based on ATL. I extended and modified one of the PocketFrog classes called Game, which is part of the PocketFrog framework, and created the PFWindow class which basically is an ATL window with its own thread to handle the window updates using PocketFrog as a GDI wrapper.

6.2 Threads and System Resources

One of the initial ideas was to create a threaded GUI, to allow greater flexibility and more feedback. It soon became evident that threads was a requirement for modifying PocketFrog to work in a client window. PocketFrog’s Game class was designed to make game programming on PPCs easy, and to be used in full-screen mode only. The class had its own window message loop and this prevented the use of the Game class as a client window together with other windows since it never released the main UI thread until the window had been closed. The modified PFWindow class however, was designed to be used as a client window letting the main program provide and handle the message loop. And since each PFWindow provides (and starts) its own thread, they can co-exist with the rest of the GUI.

Unfortunately the system resources on the PDA in question were very limited. This restricted both the number of threads that could run simultaneously and their total workload. Just using the w-lan PCMCIA card reduces the speed of the PDA considerably. I had to find a good compromise between window update frequency, robot communication frequency and general user response and feedback. This worked well in some windows where the demand was not as high (like in the 2D Map window) while in others (like the Manual Drive window) it did not work at all.

The GUI consists of these five threads:

- PFWindow thread. One for each PocketFrog window. Responsible for updating the window graphics.
- SpeedUpdate thread. Used by the Manual Drive window to communicate with the robot, setting a new target speed and receiving the current actual speed.
- StatusUpdate thread. Used by the 2D Map window to receive updates of the current robot position and status.

---

1. GAPI can not be used in windowed mode.
2. Personal Computer Memory Card International Association, expansion cards.
VideoStream thread. Used by the Manual Drive and Video Camera windows to get new frames from the video camera. The thread is responsible for communication with the HTTP server\(^1\) as well as decoding the frames.

Main UI thread. This is the thread of the main program and it is the one responsible for all the functions launched by the user, including windows, dialogs and buttons etc. The main thread is created by the OS.

### 6.3 Synchronization

Because the GUI uses multiple threads, it needs to synchronize them or undefined behaviour and so called race conditions will occur. A race condition occurs when two (or more) threads tries to access the same object at the same time and the behaviour of the code changes depending on the true access order of those threads.

For instance: Two unsynchronized threads are using the same global (integer) variable. The first thread is modifying it (by adding a number) and the other thread is using it (displaying it on the screen). Then the program will behave differently depending on the access order of those threads. The first time the program might display 3, 5, 6, 8 and the next time 2, 4, 5, 6. The behaviour is undefined.

There are several ways to synchronize threads in Windows using C++. I choose to use a critical_section object but could just as easily have used a mutex object. Critical_sections are supposed to be faster than mutex objects on average (Richter, 1998).

### 6.4 Communication and Bandwidth

In order not to stress the robot server there is a short delay between each command sent to Pluto. And since the graphics is slow (on the PDA), there is no need to update the status faster than it can be displayed. But regular communication is not the problem. The small XML messages are sent to and from the robot at a fraction of a second. And the large maps are only loaded once in a while.

Video however, needs to be transmitted and processed as fast as possible to be of any use. And video requires much bandwidth and CPU power. Even when the video jpeg frames have maximum compression, it takes between one to two seconds to send and process the first image although the video is updated at approximately one frame per second. The update frequency is acceptable although it is not optimal but the delay is far from satisfactory. With the delay and low frame rate, the robot could be as far as two seconds away from the current video image. And with a maximum speed of 1.5 m/s that means it could have moved as much as three meters.

The solution to this problem is to send smaller images (i.e. smaller file size). Since the video frames already use maximum compression, they need to be decreased in (image) size. I tested the video stream with a regular web server on my desktop computer providing 176x144 images through a custom ASP\(^2\) page and it worked very well (or at least much better). Because not only is each frame one quarter of the original size and thus transmits

---

1. The Pluto HTTP server provides the images from the network video camera through a CGI script.
2. Active Server Pages, a dynamic html server page standard.
faster but they are also decoded much faster on the PDA. Stretching and blitting\(^1\) a small image to double its size takes practically no extra time at all compared to blitting a normal image. And not only would the smaller frames be faster, they would present a better result since they would not have to be heavily compressed. Unfortunately the Axis camera on Pluto only has four image formats: 352x288, 704x576 in PAL and 352x240, 704x480 in NTSC, so there is no way of increasing video performance as it is.

But perhaps one could write a program that runs on the robot server and subsamples the images from the video camera and feeds them to the GUI on demand. Extending the XML protocol to include video would make it easier for other programs as well. And perhaps the program could save the jpeg frames in another format (gif) that is faster to decode on the PDA if this would increase performance without reducing video utility. The Siemens PDA has a maximum colour depth of 8 bits (256 colours) and so does the gif format.

---

\(^1\) Blit, copy the image data to the display buffer (screen).
7 Graphics Library

This section covers the design of a simple 3D browser and the graphics library used in the 3D Map window.

7.1 General Issues

Because speed and performance was a major factor, the design goal was, from the very beginning, to start with a very simple 3D browser and then build from there. This means no textures, no lighting, no shades or even surfaces. The world was build by so called wire-frames, which are just the outlines or corners of each object.

Normally when developing a 3D application you want to take advantage of the 3D hardware on the graphics card or at least some sort of 3D library that utilize the graphics accelerator or provides optimized software rendering. But since all PDAs lack 3D capabilities (at least at present date) and most of the PDAs do not even have a separate graphics accelerator, there are no pure 3D libraries like OpenGL\(^1\) and Direct3D\(^2\) available. For PPCs however, there is a library called PocketGL that uses GAPI to provide optimized software 3D rendering as well as basic primitives and fast fixed-point arithmetic for vectors and matrices.

As stated earlier, our PDA lacks GAPI support and can therefore not use the PocketGL library. This infers some tight restrictions on what can be accomplished with the PDA. Luckily I found the PocketFrog library which works as a GDI wrapper and provides some basic 2D primitives and takes care of all the clipping\(^3\).

I decided early on to create and work with a data structure similar to the VRML97 structure\(^4\) but restricted to the subset of VRML97 nodes used in the 3D map. The benefit of this is easy parsing, easy debugging and extensibility. Naturally I have modified some of these nodes to be more efficient within the browser, like for instance: instead of going through the IndexedFaceSet node’s coordIndex array and match each index with the coordinate in the Coordinate array, I just go through an array of polygon objects. The polygon objects are created when the 3D map is parsed.

7.2 Browser

This subsection will cover the most important parts of the GUI 3D browser and how they work. For those who are interested in exactly how the code was implemented and how the browser and VRML97 structure was designed, there is a documentation and reference manual on my web catalogue: www.nada.kth.se/~t98_hes/cas/gui/html/index.html

---

1. An open source graphics library and graphics standard developed by Silicon Graphics.
2. Part of Microsoft’s DirectX library. The counterpart to OpenGL.
3. Clipping is when, for instance, a line ends/starts outside the window surface and needs to be cut of.
4. See Appendix IV for more info.
7.2.1 Main Loop

The main loop, which is responsible for updating the window graphics, first checks if the world is currently rotating or if the user has stopped it by pressing the pause icon/button. If the world should continue to rotate, a predefined rotation matrix is applied to the current rotation matrix. This matrix is used later on when drawing/rendering all the objects.

Then the button icons are drawn and the clipping rectangle is set so that the icons will not be overdrawn by any 3D objects. The browser has a handle to the SceneGraphNode which is the first node (or root node) of the whole VRML97 structure. After validating the SceneGraphNode the TransformNode, ViewpointNode and the two ShapeNodes are retrieved (and validated) using the node data structure. The TransformNode defines the transform/projectivity that applies to all the shapes, the ViewpointNode defines the viewpoint perspective transform/perspectivity which is responsible for how the world is rendered (i.e. how the user sees the world), and the ShapeNode defines a set of shapes. The first ShapeNode is the Ground node which defines a single polygon. The second ShapeNode is the Walls node which defines all the wall polygons of the 3D Map. If the ShapeNodes are valid, they are drawn with the DrawShape() function.

7.2.2 DrawShape()

The DrawShape() function is responsible for rendering the current shape. It starts by retrieving the IndexedFaceSetNode, gathering some information about the current window size and creating the transform matrix and viewpoint matrix from the TransformNode and ViewpointNode. The function then goes through all the polygons defined in the ShapeNode and for each polygon, it goes through all the points/vertices of that polygon. After applying the transform matrix but before applying the viewpoint matrix, the world model is rotated by applying the current rotation matrix. This is a bit ad hoc, since the rotation could have been specified in the general transform node/matrix.

The function now follows one of two paths depending on whether the model is rotating or not. As long as it is rotating we want the browser to be as fast as possible and therefore we sacrifice some accuracy and simply interpret all points that are closer to the screen surface than one pixel (in the z-coordinate) as if it was exactly one pixel away. So all points \((x,y,z)^T\) where \(z > -1.0\) are interpreted as \((x,y,-1.0)^T\). This is necessary because the perspective projection from 3D to 2D is undefined when the z-coordinate (i.e. the depth) is zero. The coordinate system is aligned so that the z-axis is pointing outwards from the screen (towards the user), the x-axis points to the right and the y-axis points upwards.

The perspective projection equation looks something like this:

\[
\begin{align*}
    x &= -f X / Z + cx \\
    y &= f Y / Z + cy
\end{align*}
\]

where \((x,y)^T\) is the screen coordinate, \((X,Y,Z)^T\) is the world coordinate, \(cx\) and \(cy\) are constants to position the coordinate system at the centre of the window and \(f\) is a scale factor that is calculated once to determine the initial depth of the model. The scale factor is set so that the entire model/map is visible on start-up, determined by the relation between map

---

1. The rotation speed can be set in the setting object (settings.ini).
and window size. And at that distance, no part of the map has a \( z > -1.0 \) so avoiding the interpolation is relatively safe. The minus sign is needed to compensate the sign of the \( z \)-coordinate generated by the coordinate-axis transformation from the world to the screen where the \( x \)-axis is pointing to the right but the \( y \)-axis is pointing downwards (and hence, by the right hand rule, the \( z \)-axis is pointing inwards).

If the world is not rotating, however, we can spend some time calculating a viewing volume\(^1\) and interpolating all the points that have a \( z \)-coordinate greater than -1.0 to a new coordinate where \( z = -1.0 \). To achieve this, all the transformed points of the polygon are tested and added to a point list. If \( z \leq -1.0 \) then the point is simply added to the list (or array) without modification. But if \( z > -1.0 \) however, then we need to calculate a new location for that point depending on the location of the previous and/or next point in the polygon. There are four scenarios:

1. The next point has \( z \leq -1.0 \).
2. The previous point has \( z \leq -1.0 \).
3. Both the previous and the next point have \( z \leq -1.0 \).
4. Neither the previous nor the next point have \( z \leq -1.0 \).

The first three scenarios have almost the same solution. If the next (or previous) point lies in front of the screen (i.e. \( z \leq -1.0 \)), then we use the equation for the line between these two points and locate the point on that line where \( z = -1.0 \). The equation of a line between two points \( u \) and \( v \) looks like this:

\[
x = u + (v - u)t
\]

where \( t \) is an arbitrary scale factor. Solving for \( x_3 = -1.0 \) yields \( t \). And when we know \( t \), we can use the equation again to calculate \( x \) and \( y \) (i.e. \( x_1 \) and \( x_2 \)) for the point where \( z = -1.0 \) on that line, or in other words: we have made a linear interpolation of the two points and calculated a set of new coordinates for the point behind the screen. The new point is then added to the point list.

If both the previous and the next points have \( z \leq -1.0 \) then we make two interpolations and add two new points to the polygon point list.

If neither the previous nor the next point have \( z \leq -1.0 \) then interpolation is useless. And furthermore, the point will not affect the rendering of the map so it can be dropped safely. If all points have \( z > -1.0 \) then they are all dropped and the polygon is never rendered, as one would expect since it is fully behind the screen. After processing all the points, the polygon is rendered using the new (possibly interpolated) point list.

After completing the interpolation of the \( \text{Ground} \) polygon we calculate the plane-to-plane mapping\(^2\) used to transform screen coordinates into map coordinates for user input. This can be used either to move the viewpoint around or to send commands to the robot using the \( 3D \ Map \) as input instead of the \( 2D \ Map \). The matrix can also be used to indicate in the \( 3D \) model where the robot or other important markers are located. Currently the only use

---

1. The slice of 3-space that is visible from the current viewpoint. Should be used to sort the polygons.
2. See Appendix V for more info.
of the mapping matrix is to centre the viewpoint on top of the selected coordinate\(^1\), facing down. The matrix is calculated using the original, non-interpolated, points.

To render the polygon, \texttt{DrawPolygon()} is called and depending on the user settings it will either use wire-frame or flat rendering mode. Since no textures, lights, shades or shadows are used and all the walls have the same colour, the polygons can be drawn in any order as long as the ground polygon is drawn first. Otherwise the viewing volume should be calculated to select only the visible polygons and then sort them according to their distance from the current viewpoint (i.e. the screen). The polygons that are furthest away should be drawn first.

\subsection*{7.2.3 \texttt{DrawPolygon()} / \texttt{DrawFilledPolygon()}}

The \texttt{DrawPolygon()} function either simply connects the polygon by drawing lines between its points/vertices creating a wire-frame or calls \texttt{DrawFilledPolygon()} to handle flat rendering. That function constructs a table/array of starting and ending points for the edges of the polygon. Only the x-coordinate is stored since the y-coordinate is used as an index. After processing all the edges, the polygon is created using the table to draw a series of horizontal lines between the starting and ending points of each row, from the smallest y-coordinate occupied by the polygon to the largest.

The flat rendering part has not been completed and needs a lot more work. More information about this in the next section.

\footnote{See User's Manual for more info.}
8 Evaluation

I let some of my co-workers at CAS try the GUI and their overall impression was that the GUI looked good and worked well. They all thought the GUI was easy to manage and that the 3D visualization was a good complement to the 2D map. They had, however, some additional remarks and suggestions:

- Use fewer buttons in the 3D view.
- Where is the robot on the 3D map?
- Where is the user on the map?
- Could the rendering be done any differently?
- There should be some sort of feedback from (some of) the 2D Map buttons.
- Do not overlap two windows unless they are both updating.
- Why is there no “Collision Avoidance” status in the Manual Drive window?
- The response of the Manual Drive buttons must be faster. The robot should stop almost immediately when pressing the stop button for instance.

The evaluation group consisted of people who are working with the Pluto project and have a lot of experience with the robot, as well as with the old GUI. As mentioned earlier it is important to get feedback from a broader group of users, including those with little or no knowledge about the robot and those who are not technicians.

One thing that was not mentioned during the evaluation, but that I noticed while observing the users (and while using the GUI myself) is that some of the buttons are difficult to target. Especially the tiny OK button on all the message boxes. They require that the user has calibrated the screen and uses a pencil. To save window space on these tiny devices, Windows CE has its own window design, and so the OK button is on the window caption/title bar next to the close-this-window button. Since these are standard components defined by the platform SDK the only solution is to write your very own message box for CE with a large OK button on the bottom like normal message boxes. For the same reason, all windows should have an explicit exit/close button.

Fewer buttons in the 3D browser is a very good remark. It is pretty obvious that the normal user does not have much experience with 3D browsers and rotating objects in 3D space. So limiting the buttons to only translations would probably be more intuitive to the naive user. But what is a browser without the ability to rotate. One possible solution is to have macro buttons. For example, if the viewpoint is located in a birds-eye-view position, pressing the explore button would set off a series of viewpoint movements to move the user down to the ground plane and rotate 360 degrees before moving back to the original position. Another idea is to allow only one pair of right/left rotations with different results depending on the current viewpoint mode. When in birds-eye mode, it would rotate the viewpoint around the local z-axis and when standing on the ground (in the xy-plane) it would rotate the viewpoint around the local y-axis. There are, of course, many many more solutions, and it seems like the advanced user would benefit from having as much abilities as possible. Hence, the GUI should change appearance depending on the user’s experience level just like the old GUI did.
An indicator of where the robot is located in the 3D map could be useful. But it requires a communication thread to update the robot position and this would steal precious CPU cycles from the browser. The thread could however be idle most of the time, updating only once in a while and/or only when the map is not rotating (or browser is running a macro). So it is definitely possible. Indicating the user’s position requires a GPS enabled PDA and would also be a nice feature.

The wire-frame rendering could be a bit distracting sometimes because there are too much details (lines and walls behind other walls). But sometimes this is something positive, sort of like an x-ray feature. The flat rendering, as it is now, has the same colour on all the walls causing them to blend into each other. This actually looks worse than the wire-frames, and the colours have not been optimized for a relaxing view yet either. Enhancing the contrast can be done either by shading or by filling in the visible edges with a darker colour. The shading of each wall is determined by the relation between the wall’s normal vector and the vector of the incoming light rays. Shading has nothing to do with shadows.

The buttons of the Manual Drive window are designed as common dialogue buttons and work exactly like a button should behave. But the buttons in the 2D Map and 3D Map windows are icons that can not be pressed down. In the 3D Map this is not a problem, since every button/icon immediately generates a visible action. If the user taps the Pause button the rotation stops. If he taps the screen or one of the arrows, the viewpoint will be moved. The response is instant and does not need to be enforced by marking the button as “tapped”. But in the 2D Map window, not all buttons generate a visible action. Most of the buttons though will either move the viewpoint or display a message box, but the GoTo, Stop and Resume Explore buttons does not. The Stop and Resume Explore buttons will most likely affect the status of the robot so for instance, pressing the Stop button while the robot is exploring will change the status indicator from Exploring to Idle. The GoTo button however requires a destination to send the robot to and hence a follow-up tap somewhere on the map. This is not apparent and should be marked somehow, possibly by drawing a new border around the button in some contrast colour. Displaying a message box would become annoying after a while but possibly useful for the novice user.

The original idea was to combine video feedback with the map and the driving interface. But due to the poor performance of the PDA I designed the windows so that only the one which currently had the focus would be updated. Having the Video Camera window overlapping the 2D Map window could give the user conflicting input. For example: in the video window the robot is moving, but in the unfocused map window the robot remains in his old position until the window receives focus again. Either both windows should update the robot position, or only one window should be visible to avoid confusing the user.

There is no reason why the “Collision Avoidance” status should not be present in the Manual Drive window. This should definitely be implemented in an updated version. The reason why there is no toggle-collision-avoidance or set-max-speed buttons in the window is because I tried to keep the interface as clean and simple as possible. Gathering all the settings in the Control Panel window is much better.

To enhance the feedback of the driving interface I decided to use buttons (instead of icons) with partially filled arrows to indicate speed and a video frame. All these things required a constantly updating window. As mentioned before the system resources on this PDA is very low and when using the w-lan card together with a graphics window and two other
threads, to handle speed update and the video stream, something has to give. A simple test using a TP\textsuperscript{1} wire to connect to a custom server gives a hint of how bad it gets. Running the PC version of the GUI on the server as reference shows that the code works as intended. The speed is updated 3.3 times per second which is about what I wanted. But on the PDA with video running, the speed is updated 1.8 times per second. Without the video, the speed gets better, 2.8 times per second. What this means is that when pressing the stop button it takes a maximum of 0.3 seconds and 0.15 seconds on average for the robot to get the command using the PC. On the PDA with the video stream enabled the maximum delay is 0.55 seconds using the built in ethernet card. Using the w-lan card in a real situation would result in even slower performance since it is slightly slower than a direct connection by wire plus that the card slows down the PDA. This would explain why it sometimes takes up to a second before the robot stops and this is of course not satisfactory. Disabling the video helps a bit, but the main bottle neck here is the PDA. Faster graphics (GAPI support) and more resources would have made a huge difference. A reasonable question would be why the speed update thread is not prioritized and the answer is once again low system resources. But a quick response is very important so at the expense of the video, the speed update thread has been given a higher priority and is now updating 3.7 times per second on the test with (or without) video. This is even faster than the PC reference with the old thread priority. Considering that the video was slow already this might seem fair.

---

1. Twisted-Pair, the standard used in regular LAN networks.
9 Summary

This section begins with a quick review of the problems. It is followed by a summary of how they were solved and the conclusions that could be drawn from this project. And finally some thoughts about future work and modifications.

9.1 Recap

The assignment was to write a new GUI for Pluto using a PDA with a larger screen size and to take advantage of this new format. The GUI should be so simple and intuitive that even an inexperienced user could control the robot. The focus of the GUI (and the project) was to investigate the usage of 3D models to visualize a map for user-robot interaction with a PDA and try to answer the following questions:

- Will the 3D model help the user to get a better understanding of the surrounding environment and the composition of the map?
- Is it even possible to do 3D on PDAs in general and on the given PDA in particular?
- How fast and accurate must the 3D rendering be to be considered useful?
- Can a 3D map replace the 2D map all together?

Another objective was to use the video camera mounted on top of the robot to investigate how it could be utilized in the GUI.

PDAs are interesting because they are small and portable, with a good battery life-time. However, they are not as powerful as a laptop. It is therefore important to investigate how to obtain a good balance between visualization and performance, and discover what compromises that need to be made to achieve this balance on a PDA.

With this thesis I also try to answer the following related questions:

- Can a good GUI tool be found that is easy to use, portable, and has the speed required to implement a 3D browser on a PDA?
- What are the restrictions and limits of the given PDA? What are the issues with PDAs in general?
- Since the 3D model is built from 2D data, the height needs to be approximated. Will a predefined height suffice to make the model useful and realistic?
- Does video provide any useful information? Does it enhance the GUI? How fast must the video be to be considered useful?
- How should the GUI be designed in general? And how should the manual drive controls be designed in particular?
- How slow can the GUI components get, before the user finds them to be bad and non responsive?

No formal evaluation (i.e. user studies, interviews, etc.) of the GUI has been made to determine its usability and efficiency regarding the intended purposes. The only evaluation

1. Larger compared to the standard PPC format, 240x320 pixels. In this case, 800x600.
done has been based on the spontaneous comments and feedback from the co-workers at CAS using the GUI.

9.2 Solutions

PDAs in general are slow devices compared to laptops and desktops. And they lack both an FPU\(^1\) and a graphics accelerator\(^2\). Clearly they are not designed for 3D applications although 3D can be achieved given a fast PDA with GAPI\(^3\) support (i.e. a PPC\(^4\) device). Normal 2D games and mpeg/avi video playback runs smooth on most PPCs so it is just a matter of writing efficient code.

Given the information above, using Java for 3D on PDAs is out of the question although it is possible for simple 3D modelling on a laptop/desktop using OpenGL. On a PDA however one has to rely upon C++ and instead of OpenGL one could use either PocketGL or write a very own 3D graphics library.

The given PDA, a Siemens MOBIT8 HPC, lacks GAPI support and is therefore not an ideal platform for graphics applications. The large screen (800x600) is nice though, but it takes a long time to update. And on top of this, the PDA is very low on system resources making it hard to do more than one thing (run more than one thread) at the same time. Basically, the project was doomed from the start but we continued anyways to see just how it would turn out and if we could draw some conclusions from it.

There were no 3D libraries available for this platform so I had to start from scratch. Fortunately I found a good 2D graphics library so at least I did not have to worry about that too. I decided early on to use the VRML97\(^5\) standard for the modelling of the map. It is one of the most frequently used 3D modelling languages and there are a lot of (free) tools available that support VRML97. The GUI’s 3D browser only supports a subset of the VRML97 nodes but the internal data structure is easy to expand with more nodes, like lighting and appearance nodes, if needed.

The browser is very simple and has not been fully optimized. Instead, it has a well thought out OOP design to allow the browser to expand if the simple wire-frame browser should work well enough. And it did. The 3D visualization of the 2D map truly gives a better understanding of the map and its composition even with the wire-frame rendering.

If the 3D map could have been rendered in a full screen window, then the 2D map would be rather useless since it is simply a birds-eye-view perspective (or projection onto the xy-plane) of the 3D map. The plane-to-plane projectivity\(^6\) provides a two way mapping between screen coordinates and map coordinates so there are no problems with user input and world/robot feedback. But extra care must be taken to design new buttons and combining the functionality of the 2D map with the 3D browser in a logical and intuitive way.

---

1. Floating-Point Unit, a co-processor that handles floating-point arithmetic.
2. Some of the new PDAs have a graphics accelerator chip now days and special 3D chips are on the way.
3. Section 3.2 on page 8
4. Section 3.1 on page 8
5. See Appendix IV for more info.
6. See Appendix V for more info.
This is one of the benefits of having the two separated, since then the 3D Map is only a representation, and the 2D Map provides the user-robot interaction.

The video showed great potential. Not only while driving the robot when it is out of sight, but also for feedback on what the robot is doing and possibly for taking snapshots of important locations. The 2D map only gives the position of the robot and the detected obstacles. Video gives the user a view of the actual surroundings. Unfortunately the video suffers from the same platform specific problems as the 3D browser. Slow CPU and graphics combined with low system resources made it impossible to get good performance for the video stream. The poor performance (i.e. low frame rate) made the usage of video together with the manual driving controls useless. The video is still useful as a complement to the 2D and 3D maps though, running in a separate window.

The large screen size had a huge impact on the GUI design since it gave me the ability to design everything large. Buttons, components, text, etc. Instead of having short abbreviations, the whole word could be written making it easier for the inexperienced user to understand. And the GUI could be designed for user interaction using the index finger instead of a pen. The only thing that restricted window (and component) size was poor graphics performance so a compromise had to be made.

9.3 Conclusion

It is very important that a GUI is responsive so that the user knows that it is working. The GUI must never stall or freeze without telling the user about the possible delay first and then confirming the action later. Users are in general very impatient. If the GUI is too slow, the user assumes it is not working properly or becomes agitated because the GUI has been poorly designed. But if the user knows an action might take a while to execute, he will have more tolerance.

If you need to write small and fast applications that use graphics, then ATL/WTL combined with some graphics wrapper (like PocketFrog) is the way to go. One could use PocketFrog altogether, but standard UI components not only saves a lot of time, they are also more familiar to the user. If the emphasis is not on high performance graphics, then Ewe is highly recommended.

The most important thing to check before developing a graphical application for a PDA, or before purchasing a PDA intended to be used primarily for multimedia applications, is to make sure it supports GAPI and has good memory bandwidth. Accessing the video buffer directly through GAPI is much faster than using the GDI interface.

Modelling a 3D world from a 2D map by approximating all the obstacles as a wall with predefined height works fine. And the model only needs to use basic polygon shapes to be useful, since this is the way the 2D map is built. Using only wire-frames works pretty good but it could be confusing with all the extra lines. Using flat rendering with a single colour has a tendency to clog the map. For flat rendering to be useful it requires light and shades to enhance the contours and separate the walls from each other. Other alternatives are to use different colours for adjoining walls or to draw the corners with a different colour.

1. Section 4.2.3 on page 13
2. Section 4.1.2 on page 11
As this project showed, the speed and details of the 3D world does not need to be very high to be useful. A simple wire-frame increases the understanding of the map considerably. And even though the rotation of the map is a bit slow and choppy it is still valuable. The important thing is that the interface (and hence the rendering) should respond to the user input as fast as possible. The delay should be no more than a few tenths of a second. But as long as the world is changing (i.e. moving or rotating) detail is not that important, so one could use wire-frames for active rendering and filled polygons with different shades for passive rendering. Combining wire-frame and flat rendering is also useful since the wire-frames enables the user to see through walls.

A 3D model can definitely be used instead of the 2D map provided that it can be rendered in a full screen window and remain reasonable speed.

For the video to be useful as guidance while driving the robot, it must be fast and with as little delay as possible. At least one frame per second. The quality of the video is not as important as long as the user can distinguish and identify obstacles. As a way of getting information about the world in front of the robot the video speed does not matter as much, as long as the quality of the images is reasonably good. For snapshots, the speed is not important at all but the quality must be good.

9.4 Future Work and Recommendations

The focus of this project has been to evaluate if a simple 3D browser could be implemented on the given platform and if this would help the user to understand the map and the surroundings better. Therefore the GUI is far from optimized and there are several areas that could be extended and modified. As mentioned in the evaluation, the GUI should be evaluated by more users to provide more feedback about the system. The only current feedback is from people who are technicians and are very familiar with the system.

There are a few simple things that could increase the utility of the GUI. The buttons in the 3D Map and 2D Map windows must be enlarged and/or redesigned because the current ones are to small for the GUI to be navigated using only the index finger. For the same reason, all the windows should have an explicit exit/close button. The settings dialogue should be constructed so that all the variables of the settings object can be modified (and applied) from within the GUI and not only from an external text editor. There should be an indicator of the current “Collision Avoidance” status in the Manual Drive window. Also, adding a right/left hand layout option in the settings would be nice for those who are left handed. These are all minor alterations that I did not have the time to implement.

As mentioned in the evaluation, the GoTo button needs some sort of feedback to indicate that a second tap is required and that the first tap has been registered.

The design of the buttons and the colours used throughout the whole GUI should be tested to achieve the highest contrast in an outdoor environment.

A useful feature that has been prepared for is the ability to take snapshots from the video camera and store them in memory (or to disk). The position of these screenshots could be indicated in the 2D Map window by blue robot icons. Pressing an icon will launch the snapshot viewer displaying time and location of the snapshot.
As the functionality of the GUI grows, it should take advantage of the multi user-level option built into the system but not currently used. As mentioned in the evaluation, the 3D browser would benefit from a few macro buttons in normal mode and all possible buttons in advanced mode, providing the speed of the browser is not compensated.

The speed of the wire-frame rendering while rotating the map is acceptable. And when the scene is not moving, the interpolation and flat-rendering algorithms work fine. One could even consider adding light and shading to increase the reality factor. But after that I think it would be hard to maintain acceptable performance on this PDA since it lacks the GAPI\textsuperscript{1} library. Textures, shadows and complicated shapes would require a faster machine. To increase 3D performance further on this PDA though, one could rewrite the vector and matrix transformation functions to use fixed point arithmetic, since all PDAs lack a FPU (Bikker, 2001). The PocketFrog graphics library uses the 22.10 fixed-point math standard and is already optimized.

To increase video performance one could try passing the video frames through a server-side pre-processor that subsamples each frame and saves it in a smaller format. Since the server is much faster than the PDA, this trick to reduce bandwidth and image size could work providing that the pre-processor does not interfere with any of the other server-side programs.

We have discussed taking the GUI to the new TabletPC platform, which is like a small laptop without the keyboard that runs standard desktop applications. This would be a huge step forward since this platform does not have any of the limitations of the PDA and it is only slightly larger (but lighter) than the Siemens Mobic. On the other hand, it is not as rugged and the battery life-time is not as long (although probably long enough). On this new platform, the 3D map could be the basis for all of the users input and feedback making the 2D map useless. But the core of the browser needs to be rewritten to take advantage of a 3D library like OpenGL running in full screen mode with wall and ground textures, shading, shadows and lighting. This opens up a lot of possibilities for what could be done using a 3D interface to control a mobile robot.

\textsuperscript{1} Section 3.2 on page 8
10 References

10.1 GUI Tools

Brereton M. 2002a. What is Ewe?
http://www.ewesoft.com/WhyEwe.htm (verified 2003-03-21)

Brereton M. 2002b. Comparison of the Ewe VM with a PersonalJava VM.
http://www.ewesoft.com/VersusPersonalJava.htm (verified 2003-03-21)

http://bluezoo.org/knife/gui.html (verified 2003-03-21)

Brereton M. 2003. Ewe Homepage.
http://www.ewesoft.com/ (verified 2003-03-21)

http://csgrad.cs.vt.edu/~ccatanza/superWaba.html (verified 2003-03-21)

ClipCode. 2000. Chapter 1 - Overview of WTL.
http://www.clipcode.net/content/wtl_guide/chapter1.htm (verified 2003-03-22)

http://www.gamedev.net/reference/articles/article1312.asp (verified 2003-03-21)

http://www.cybergarage.org/vrml/cv97/cv97java/index.html (verified 2003-03-21)

Park M. 1999. ATL 3.0 Window Classes: An Introduction.

http://www.siemens.com/mobic/ (verified 2003-03-21)

Smart J. 2003. WxWindows Homepage.
http://www.wxwindows.org/ (verified 2003-03-21)

http://www.pocketfrog.com/ (verified 2003-03-22)


10.2 GUI Design


10.3 Future Work and Recommendations

http://www.flipcode.com/tutorials/tut_toofast02.shtml (verified 2003-03-27)
Appendix I - XML Protocol

The XML protocol, used for communication between the client (GUI) and the server (Pluto), can be divided into sub categories depending on the purpose of the command, or to be more precise, on the doer.

1 XML skeleton

All XML commands (or messages of you will) have the following form:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<command>
    <doer>#doer</doer>
    #data
</command>
```

where #doer is one of the following:

- driver
- goer
- follower
- explorer
- get_map
- get_x3d_map

and #data is doer specific XML tags.

All commands (except get_map and get_x3d_map) generate an XML reply of the following form:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<reply>
    <doer>#doer</doer>
    #data
</reply>
```

where #doer is like before, and #data is doer specific XML tags.

The get_map and get_x3d_map commands simply returns the map in an XML format.

Issuing a command that changes the doer will cause the robot to abandon/abort the action of the current doer. For instance, if the user has previously sent an explore command and then sends a goto command, the goer object will try to get sole possession over the robot and tries to stop the explorer object. If the goer succeeds, the goto command will be executed and the current doer will be set to goer. The only exception to this, is queries (or empty commands) where the current status is sent back to the client without affecting the doer at all.
2 Driver

The *driver* commands are used by the client/user to move the robot and has the following *data* tags:
- speed
- turn
- stop
- avoidance

*Speed* and *turn* tags can be sent together in one single command. *Stop* will always take precedence over *speed* and *turn*. *Avoidance* should be sent as a single command for clarity although it could be mixed with the others.

All the *driver* commands generate the following reply:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<reply>
  <doer>driver</doer>
  <speed>#float</speed>
  <turn>#float</turn>
  <obstacle_detected>#int</obstacle_detected>
</reply>
```

where *#float* is a number between -1.5 and 1.5, representing either m/s or rad/s, and *#int* is either 1 or 0, representing *true* or *false*.

2.1 Speed

Tells the robot how fast it should move in the current direction.

```xml
<speed>#float</speed>
```

where *#float* is the speed in meters per second (-1.5 to 1.5).

Positive speed is forward, negative speed is backwards.

2.2 Turn

Tells the robot how fast it should turn.

```xml
<turn>#float</turn>
```

where *#float* is the turning rate in radians per second (-1.5 to 1.5).

Positive turning rate turns left, negative turning rate turns right.
2.3 Stop

Tells the robot to stop.

<stop></stop>

2.4 Avoidance

Tells the robot to turn on/off the collision (or obstacle) avoidance. It is enabled by default, but in some situations it must be turned off for the robot to be able to move. For instance, when driving the robot through a narrow doorway.

<avoidance>#int</avoidance>

where #int is either 1 or 0 (on or off).
3 Goer

The *goer* commands are used by the client/user to move the robot while using the map. It can also be used to get the current location and status of the robot. The *goer* command has the following *data* tags:

- *goto*
- *go*
- *go_straight*
- *pose*

These tags can not be combined with each other into a single command.

All the *goer* commands generate the following reply if the current *doer* is *goer*:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<reply>
  <doer>goer</doer>
  <pose>
    <x>#float</x>
    <y>#float</y>
    <theta>#float</theta>
  </pose>
  <obstacle_detected>#int<obstacle_detected>
</reply>
```

where *x* and *y* are the current coordinates measured in meters, and *theta* is the current orientation measured in radians from north. *Obstacle_detected* is a flag set when the robot has encountered an object in its path. *Obstacle_detected* is either 1 or 0, representing true or false.

Positive *theta* means CCW\(^1\), negative *theta* means CW.

If the current *doer* is *explorer*\(^2\), the *obstacle_detected* tag is replaced by a *status* tag (and of course, the *doer* tag data is changed from *goer* to *explorer*).

---

1. Counter Clock Wise
2. Section 5 on page 48
3.1 Goto

This is the most frequently used command in the GUI while using the 2D map. It tells the robot to go to a new set of coordinates (x, y).

<goto>
  <x>#{float}</x>
  <y>#{float}</y>
</goto>

where #{float} is measured in meters.

3.2 Go

This command enables the robot to move freely while using the map. It corresponds to the speed and turn commands of the driver\(^1\) object.

<go>
  <linear_velocity>#{float}</linear_velocity>
  <angle_rate>#{float}</angle_rate>
</go>

where #{float} is a number between -1.5 and 1.5 representing either meters per second or radians per second.

Positive angle rate turns left, negative angle rate turns right.

3.3 Go_straight

Tells the robot to start moving in a specific direction.

<go_straight>
  <linear_velocity>#{float}</linear_velocity>
  <theta>#{float}</theta>
</go_straight>

where linear_velocity is the robot speed (-1.5 to 1.5) in meters per second and theta is the angle of the robot (or direction of movement) measured in radians from north.

Positive theta means CCW, negative theta means CW.

Note: This command can be used to rotate the robot to a desired orientation if the linear_velocity is zero.

---

1. Section 2 on page 42
3.4 Pose

Tells the robot where he is (or where he should be according to the map).

```
<pose>
    <x>#float#</x>
    <y>#float#</y>
    <theta>#float#</theta>
</pose>
```

where \( x \) and \( y \) are the new coordinates measured in meters, and \( \theta \) is the new orientation measured in radians from north.

Positive \( \theta \) means CCW, negative \( \theta \) means CW.
4 Follower

This command enables the robot to follow an object (the user) and act as a mule, carrying heavy equipment. The follower command only has one data tag:

- follow

The follower command generates the following reply:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<reply>
  <doer>follower</doer>
  <status>#int</status>
</reply>
```

where #int is either 1 or 0, representing follow mode on or off (default).

4.1 Follow

Tells the robot to enable or disable follow me mode.

```xml
<follow>#int</follow>
```

where #int is either 1 or 0, representing follow mode on or off.
5 Explorer

The explorer commands are used by the client/user to explore the world and has the following data tags:

- explore
- abort
- save
- resume

These tags can not be combined with each other into a single command.

All the explorer commands generate the following reply:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<reply>
  <doer>explorer</doer>
  <pose>
    <x>#float</x>
    <y>#float</y>
    <theta>#float</theta>
  </pose>
  <status>#int</status>
</reply>
```

where x and y are the current coordinates measured in meters, and theta is the current orientation measured in radians from north.

Positive theta means CCW\(^1\), negative theta means CW.

Status can have one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Saving the map</td>
</tr>
<tr>
<td>0</td>
<td>SLAM thread not active</td>
</tr>
<tr>
<td>1</td>
<td>SLAM running</td>
</tr>
<tr>
<td>2</td>
<td>SLAM ready to run</td>
</tr>
<tr>
<td>3</td>
<td>Map saved, SLAM ready to run</td>
</tr>
</tbody>
</table>

SLAM = Simultaneous Localization And Mapping (i.e. the explorer object)

---

1. Counter Clock Wise
5.1 Explore

Tells the robot to (autonomously) explore an area of a given size.

\[<\text{explore}>
  \text{<width> #float</width>}
  \text{<height> #float</height>}
\] <explore>

where #float is measured in meters. The width and height specifies a rectangle in front of the robot, where width defines the baseline (with the robot on the centre) and height defines the length from the current position and forward in the facing direction.

When the robot has covered the entire area, it returns to the starting position.

5.2 Abort

Tells the robot to abort explore and stop moving. This will also save the map.

\[<\text{abort}>\] <abort>

5.3 Save

Tells the robot to abort explore and save the map.

\[<\text{save}>\] <save>

5.4 Resume

Tells the robot to resume any previously aborted explore.

\[<\text{resume}>\] <resume>
6 Get_map

This command tells the robot server to send back the XML representation of the internal map. It doesn't have any data members.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<command>
  <doer>get_map</doer>
</command>
```

For more info about the XML map file format, see Appendix II.

7 Get_x3d_map

This command tells the robot server to send back the X3D (3D) representation of the XML (2D) map. It doesn't have any data members.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<command>
  <doer>get_x3d_map</doer>
</command>
```

For more info about the X3D map file format, see Appendix III.
Appendix II - XML File Format

The internal (2D) map of the robot is converted into an XML file which can be sent to the client on demand. The format can also be used for loading external maps to the GUI.

1 XML Format

The XML file format follows the general XML specifications, or more precisely:
- The file begins with `<xml version="1.0" encoding="UTF-8"?>`
- The file has a root tag (i.e. `<map>`)  
- All opening tags are followed by a closing tag.
- All tag names are lower case, XML valid characters.

2 Data Tags

The main (or root) tag is `map`. The member (or data) tags can be divided into header tags and body tags although strictly speaking, they are all body tags.

2.1 Header Tags

The following tags define global map properties and are therefore considered as header tags.

2.1.1 Name

The `name` tag specifies the name of the map.

```
<name>#name</name>
```

where `#name` is any valid character expression.

2.1.2 Units

The `units` tag specifies the distance between two units.

```
<units>#units</units>
```

where `#units` can be any of the following three: `mm`, `m`, `km`. Any unknown `units` value is interpreted as meters, which is the default unit measure.
2.1.3 North

The *north* tag specifies the map offset from north.

```xml
<north>#float</north>
```

where *#float* is the offset measured in radians (-π to π). Since the robot always generates maps with zero offset, the GUI ignores this tag at the moment. This could change later on though, to support external maps.

2.1.4 X_min

The *x_min* tag specifies the smallest x coordinate found in the map.

```xml
<x_min>#float</x_min>
```

2.1.5 X_max

The *x_max* tag specifies the largest x coordinate found in the map.

```xml
<x_max>#float</x_max>
```

2.1.6 Y_min

The *y_min* tag specifies the smallest y coordinate found in the map.

```xml
<y_min>#float</y_min>
```

2.1.7 Y_max

The *y_max* tag specifies the largest y coordinate found in the map.

```xml
<y_max>#float</y_max>
```

2.1.8 X_offset

The *x_offset* tag specifies the map offset in the x direction, which for some reason is exactly the same as the *x_min* tag.

```xml
<x_offset>#float</x_offset>
```

2.1.9 Y_offset

The *y_offset* tag specifies the map offset in the y direction, which for some reason is exactly the same as the *y_min* tag.

```xml
<y_offset>#float</y_offset>
```
2.2 Body Tags

The following tags define local properties about the map and are therefore considered as body tags.

2.2.1 Points

The points tag defines all the points found in the map. A point is the basic building block for walls and other structures. Pluto, and the GUI, only uses walls.

```xml
<points>
  <length>#int</length>
  <point>
    <p_key>#int</p_key>
    <y>#float</y>
    <x>#float</x>
  </point>
  ...
</points>
```

where length defines the number of point tags, p_key defines a unique point identifier and x,y is the coordinate of the point.

2.2.2 Walls

This is the only meta (or structure) tag used by Pluto and the GUI. It defines a wall, or a solid object of some sort.

```xml
<walls>
  <length>#int</length>
  <wall>
    <w_key>#int</w_key>
    <start>
      <p_key>#int</p_key>
    </start>
    <end>
      <p_key>#int</p_key>
    </end>
  </wall>
  ...
</walls>
```

where length defines the number of wall tags, w_key defines a unique wall identifier, start defines the starting point of the wall and end defines the end point.

All the p_key values must be present in the points tag.
Appendix III - X3D File Format

The 3D map is constructed from the 2D XML map\(^1\) which represents the internal map of the robot. Since this map has no clues about the third dimension what so ever, all walls (or obstacles) are assumed to be of the same predefined height. The world (or 3D map) created uses a subset of the X3D standard and is sent to the client on demand. The 3D maps are fully X3D compliant and can be viewed in any external X3D browser. The GUI 3D browser however is not fully X3D compliant and can only parse and display worlds written in the special X3D format defined within this Appendix.

1 X3D Definition

X3D is both a file format based on XML and a 3D modelling standard based upon VRML97\(^2\), another 3D modelling standard and currently the most popular one. A simple interpretation would be that the X3D specification defines the XML tags (DTD\(^3\) rules) that define the X3D file format. On top of this ambiguity the GUI 3D map is constructed from a subset of X3D/XML tags.

So, to avoid confusion the following is assumed for the rest of this Appendix

* Any reference to “the X3D file” is a reference to the robot 3D map.

2 File Format

The X3D file format follows the general XML specifications, or more precisely:

* The file begins with `<xml version="1.0" encoding="UTF-8">`
* The file has a root tag (i.e. `<X3D>`)  
* All opening tags are followed by a closing tag, unless the opening tag is closed (i.e. `<tagname/>`).
* All tag names are composed of XML valid characters, not necessarily all lower case.
* All attribute values are quoted (i.e `<tagname attribute="value">`)

And, obviously, the X3D file format follows the X3D specifications.

\(^{1}\) See Appendix II for more info about the XML file format.
\(^{2}\) Virtual Reality Modelling Language. See Appendix IV for more info about VRML97.
\(^{3}\) Document Type Definition, defines the legal building blocks of an XML document.
3 Tag Layout

The 3D map has the following tag layout:

```xml
<X3D>
  <Scene>
    <Viewpoint/>
    <Transform DEF="World">
      <Shape DEF="Ground">
        <IndexedFaceSet>
          <Coordinate/>
          <Color/>
        </IndexedFaceSet>
      </Shape>
      <Shape DEF="Walls">
        <IndexedFaceSet>
          <Coordinate/>
          <Color/>
        </IndexedFaceSet>
      </Shape>
    </Transform>
  </Scene>
</X3D>
```

The 3D browser looks for the presence of the following nodes

- Viewpoint
- Transform (with the attribute DEF="World")
- Shape (with the attribute DEF="Ground")
- Shape (with the attribute DEF="Walls")

If all these nodes are found, the map is considered valid. The file can contain any number of nodes as long as it remains a valid XML file, but these four nodes must always be present. The 3D browser only knows how to parse and display IndexedFaceSet nodes, so if the shapes are made of other 3D elements they will not show in the GUI.

4 Tag Definitions

This section will define the properties of all the nodes listed in the layout above. It will also describe if and how the parser parses the nodes. The parser will assume that the file is X3D compliant, which it is if it is constructed by Pluto, and will not try to verify this. Therefore some nodes are never parsed and the number of nodes parsed is not counted or verified. The definitions will be restricted to our subset of X3D elements. For more information about the tags, see the X3D DTD and Schema at [www.web3d.org/x3d](http://www.web3d.org/x3d)
4.1 Common Attributes

Some of the attributes are defined and reused in many different tags. These are covered here.

4.1.1 DEF

The DEF attribute defines an id that can be used by another tag instead of a tag definition. This enables frequently used tags to be defined only once.

This attribute is only parsed for the Transform¹ and Shape² tags.

4.1.2 USE

The USE attribute specifies an id that it should use instead of a tag definition. This attribute tells the tag it should reuse a previously defined tag.

This attribute is not parsed.

4.2 X3D

This tag is the main (or root) tag of the X3D/XML file. It has no individual properties³ and must contain one Scene node.

This tag is not parsed.

4.3 Scene

This tag defines a SceneGraph node, which simply is the world itself. It has no individual properties and can contain any number of the following tags:

- Viewpoint
- Transform
- Shape

plus some other tags that are not part of our subset.

This tag is not parsed.

---

¹ Section 4.5 on page 58
² Section 4.6 on page 59
³ Actually, X3D does have two attributes but both are optional and not used by the GUI.
4.4 Viewpoint

This tag defines a Viewpoint node, which is a location in space from where the user can view the world. It has the following properties (and default values):

- description (“”)
- fieldOfView (“0.785398”)
- position (“0 0 10”)
- orientation (“0 0 1 0”)
- centerOfRotation (“0 0 0”)
- jump (“true”)
- DEF (“”)
- USE (“”)

and can’t contain any tags.

Only the first four attributes are being parsed and used by the browser. The remaining attributes are defined here for compliance with the X3D standard.

4.4.1 Description

The description attribute defines a name (or description) of the viewpoint. A scene graph can have several viewpoints and this is one way of distinguishing between them. Pluto names the default viewpoint “v1” but the 3D browser simply selects the first one.

4.4.2 FieldOfView

The fieldOfView attribute defines how much of the world that is visible from the current viewpoint (fig. 1). The angle (α) should be more than zero and less than π.

![Figure 1 - Field Of View](image)

4.4.3 Position

The position attribute defines the initial position of the viewpoint.

4.4.4 Orientation

The orientation attribute defines the initial orientation of the viewpoint. The first three values defines a vector drawn from origo, and the fourth defines the angle the viewpoint will rotate around that vector.
4.5 Transform

This tag defines a Transform node. The node affects the properties of all its child nodes. Multiple Transform nodes can be nested. The node has the following properties (and default values):

- `bboxCenter` ("0 0 0")
- `bboxSize` ("-1 -1 -1")
- `center` ("0 0 0")
- `rotation` ("0 0 1 0")
- `scale` ("1 1 1")
- `scaleOrientation` ("0 0 1 0")
- `translation` ("0 0 0")
- `DEF` ("")
- `USE` ("")

and can contain any number of the following tags:

- Shape
- Viewpoint
- Transform

plus some other tags that are not part of our subset.

4.5.1 BboxCenter

The `bboxCenter` attribute defines the centre of the bounding box. A bounding box is a box that encloses the children of the Transform node. This value is currently unused by the browser since it assumes that all maps/worlds are centred at origo.

4.5.2 BboxSize

The `bboxSize` attribute defines the size (x, y and z) of the bounding box centred at `bboxCenter`. If `bboxSize` has the value "-1 -1 -1" then `bboxSize` is undefined.

4.5.3 Center

The `center` attribute defines a translation offset from the origin of the local coordinate system. This value is parsed but never used. The browser assumes that scale, rotation and translation are relative origo.

4.5.4 Rotation

The `rotation` attribute defines a rotation of the coordinate system. This is the second operation applied to the total transformation matrix of the GUI 3D browser.
4.5.5 Scale

The scale attribute defines a scale of the coordinate system. This is the first operation applied to the total transformation matrix of the GUI 3D browser.

4.5.6 ScaleOrientation

The scaleOrientation defines a rotation of the coordinate system that only applies to the scale operation. This value is parsed but never used. The browser assumes that the scale is relative to the local coordinate system at origo.

4.5.7 Translation

The translation attribute defines a translation of the coordinate system. This is the third and final operation applied to the total transformation matrix of the GUI 3D browser.

4.6 Shape

This node defines a Shape node, and has the following properties (and default values):

- bboxCenter (“0 0 0”)
- bboxSize (“-1 -1 -1”)
- DEF (“”)
- USE (“”)

and can contain the following tags:

- IndexedFaceSet

plus some other tags that are not part of our subset.

4.6.1 BboxCenter

The bboxCenter attribute defines the centre of the bounding box. A bounding box is a box that encloses the Shape node’s geometry.

This value is not parsed.

4.6.2 BboxSize

The bboxSize attribute defines the size (x, y and z) of the bounding box centred at bboxCenter. If bboxSize has the value “-1 -1 -1” then bboxSize is undefined.

This value is not parsed.
4.7 IndexedFaceSet

This tag defines an IndexedFaceSet node, which is a list of faces/polygons defined by several index lists. It has the following properties (and default values):

- colorIndex
- coordIndex
- ccw ("true")
- colorPerVertex ("true")
- convex ("true")
- creaseAngle ("0")
- normalPerVertex ("true")
- solid ("true")
- normalIndex
- texCoordIndex
- DEF ("")
- USE ("")

and can contain the following tags:

- Color
- Coordinate

plus some other tags that are not part of our subset.

Out of all the attributes, only the first two are parsed and used by the 3D browser. The remaining attributes are defined here for compliance with the X3D standard.

4.7.1 ColorIndex

The colorIndex attribute is a (space separated) list of indexes that each assigns one of the colours defined by the Color node, to one of the polygons defined by the coordIndex attribute. The first index in the list corresponds to the first polygon in the coordIndex list. The Color node list is zero indexed.

4.7.2 CoordIndex

The coordIndex attribute is a (space separated) list of indexes that each assign a coordinate defined by the Coordinate node to a polygon face set. Each set is separated by an index value of -1 or the end of the list. An IndexedFaceSet node should have at least one polygon, and that polygon should have at least three vertexes. The Coordinate node list is zero indexed.
4.8 Color

This tag defines a Color node. It has the following properties:

• color
• DEF
• USE

and can not contain any tags.

Only the color attribute is parsed and used by the 3D browser.

4.8.1 Color

The color attribute defines a (comma separated) list of RGB colours. The RGB values range from zero to one (black to white). The colours are assigned to the polygon faces by the colorIndex attribute in the IndexedFaceSet tag.

4.9 Coordinate

This tag defines a Coordinate node. It has the following properties:

• point
• DEF
• USE

and can not contain any tags.

Only the point attribute is parsed and used by the 3D browser.

4.9.1 Point

The point attribute defines a (comma separated) list of coordinates. The coordinates are used to define polygon faces by the coordIndex attribute in the IndexedFaceSet tag.
Appendix IV - VRML97 and Basic 3D Modelling

This appendix tries to explain basic 3D modelling using VRML97. Only the nodes and features relevant to the GUI will be covered. For more information about building VRML97 worlds I suggest reading Floppy's VRML97 Tutorial (2002).

1 Introduction

1.1 VRLM 1.0, 2.0 and 97

VRML stands for Virtual Reality Modelling Language. Originally, there was VRML 1.0, which was a first attempt at an Internet 3D language and after a while it was revised into VRML 2.0, which changed the file structure and added a lot of new features such as animation. VRML 2.0 was then submitted to the ISO/IEC for approval and standardization. VRML97 is the outcome of that process, and is an ISO/IEC standard.

1.2 X3D

As noted in Appendix III, X3D extends VRML97, or to quote the X3D Task Group:


X3D is relatively new so there are not as many tools available for X3D as for VRML97.

1.3 Browsers

There are a lot of VRML97 compliant browsers out there. The most popular ones are Cortona and Cosmo. Both work as a plugin for Internet Explorer or Netscape. There are currently no X3D plugin and hardly any browsers.

1.4 Editing

To create VRML97 worlds all that is required is a normal text editor, but special editors and authoring tools exist. VRML97 is case-sensitive and the normal extension for VRML97 files is .wrl.

1.5 Nodes, Fields and the Scene Graph

A VRML97 world is made up of nodes, which specify an object type. The properties of the object are defined in fields inside the node. Fields can be anything from the size of a box to another node inside the first. Nesting nodes gives a kind of hierarchy sometimes referred to as the scene graph.
2 Modelling

2.1 Scene Graph

Scene graphs are treelike data structures used to store, organize and render 3D scene information (Walsh & Gehringer, 2002). In VRML97, each .wrl file defines a scene graph. Multiple scene graphs can then be combined into a virtual universe. The scene graph consists of nodes for shapes, lighting, materials, textures, behaviours, viewpoints, etc. A scene graph can be a single shape or an entire world.

2.2 Undefined Nodes and Fields

If a field is not defined in the node, it will be assigned default values by the browser. So there is no reason to explicitly define values for nodes and fields if they are not used.

2.3 Coordinate Systems and Axes

All distances in VRML97 are measured in metres. This does not matter in your own world, as long as you are consistent, but if you want to link to other people's worlds or use external files, it is a good idea to keep to the standard. The VRML97 coordinate system works as shown below (fig. 1). The x-axis is horizontal, y is vertical, and z is coming out of the screen towards you.

Rotations in VRML97 are performed using the right-hand rule. Imagine wrapping a hand around one of the axes, with the thumb pointing in the positive direction. Then the direction of positive rotation is the same as the direction of the fingers wrapped around the axis.
2.4 DEF / USE

For simplicity and flexibility, each node can be defined (i.e. given a name) by the DEF parameter. DEFined nodes can then be reused in other worlds or on multiple locations in the current world by the USE parameter. For example, if you want to reuse the appearance of one shape in another, it might look something like this:

```
DEF BOX Shape {
    appearance DEF APP1 Appearance {
        material Material {
        }
    }
    geometry Box {
    }
}

DEF SPHERE Shape {
    appearance USE APP1
    geometry Sphere {
    }
}
```

By defining the shapes they can also be reused to create multiple shapes of the same type.

2.5 Transform

All the shapes, viewpoints, spotlights etc. are defined using a local coordinate system centred at origo. Normally a world consists of more then a single object, so they each need to be moved to a new position (or they would all be centred at origo). This is where the Transform node comes in. It is a Group node, meaning that is can hold other nodes, including Transform nodes. The transformations within a Transform node apply to the children of the node. This is called nesting, where a node can have any number of child nodes. The syntax for this is shown in the example below:

```
Transform {
    scale 2 1 1
    rotation 0 1 0 0.78
    translation 1 1 1
    children [
        USE BOX
    ]
}
```

Within a single Transform node, the transformations are carried out in strict order: first scale, then rotate, then translate. So in order to perform a translation followed by a rotation, one needs to nest two Transform nodes.
2.5.1 Scale

This field defines how much the object should be scaled in each direction. If scale is 1 1 1 the objects remain unchanged, but if the scale is 2 1 1 then the object will be twice as large in the x direction only.

2.5.2 Rotation

The first three parameters defines a vector in space centred at origo. This is the rotation axis. The fourth parameter is the rotation angle in radians. Positive rotation is defined using the right-hand rule.

2.5.3 Translation

This field defines how much the centre of the object should be translated/moved in the given direction. If translation is 0 0 0 then it remains unchanged.

2.6 Viewpoint

The viewpoint tells the browser where to position the user in the world at start-up. A scene graph can have more than one viewpoint, in which case the user can switch between them. Most browsers allow the user to move and rotate the viewpoint to investigate the world from different positions and different angles. If no viewpoint is specified, it is up to the browser to position the user and define a starting viewpoint. Usually, the default viewpoint is positioned somewhere on the z-axis pointing in the negative z direction. The node looks something like this:

```
Viewpoint {
    position 0 5 5
    orientation 1 0 0 -0.39
    fieldOfView 0.39
    description "Cam1"
}
```

which would position the viewpoint/camera at (0,5,5)^T looking slightly down at 22.5 degrees, with a narrow FOV and a description of “Cam1”. The description can be used to identify the viewpoint. Field of view (FOV) defines how much of the world is visible from the current viewpoint (fig. 2). The default angle (α) is 0.78 radians (45 degrees) giving a “normal” field of view.

![Figure 2 - Field Of View](image)
2.7 Shape

The Shape node defines a single shape. Each shape consists of an appearance field, containing an Appearance node, and a geometry field containing some kind of geometry node. See Section 2.4 on page 64 for an example.

2.7.1 Appearance

The Appearance node defines the look and feel of the shape. It can contain a material field or a texture field. A material field can contain a Material node, and that node defines how the surface of the shape will behave (shading, glow, colour, etc.). The texture field can contain some kind of texture node, like ImageTexture, MovieTexture or PixelTexture where ImageTexture ought to be the most useful one.

2.7.2 Geometries

There are a lot of different geometry nodes although there is no actual Geometry node. They all have different properties and could be divided into basic geometries (Box, Cylinder, Sphere, Cone, etc.) and complex geometries (IndexedFaceSet, IndexedLineSet, PointSet, ElevationGrid, etc.).

2.7.3 IndexedFaceSet

This node defines a set of polygons by defining all the vertices of these polygons. With an IndexedFaceSet node you can create practically any given shape. This node is not always the best choice for constructing a generic shape but it is the one I used for the GUI because it defines the coordinates of the vertices. A shape could look like this:

IndexedFaceSet {
  coord Coordinate {
    point [  -2 0 2, 2 0 2, 2 0 -2, -2 0 -2
             -2 4 2, 2 4 2, 2 4 -2, -2 4 -2]
  }
  coordIndex [0 4 7 3 -1
               1 2 6 5 -1
               4 5 6 7 -1
               2 3 7 6 -1 ]
  solid FALSE
}

The Coordinate node defines a set of coordinates in the world, and the coordIndex field defines a set of polygons using these points. Each polygon ends with an index value of -1, so the first polygon consists of the following vertices: (-2,0,2), (-2,4,2), (-2,4,-2), (-2,0,-2). It is very important how these polygons are constructed, i.e. in what order the vertices are specified. Not only because you normally do not want intersecting polygons (i.e. bow ties) but also because it determines the normal of the polygon. The normal affects how different
lights affect the surface and how it is rendered. Every polygon surface has two faces, one on the front and one on the back, and they are defined by the order of the vertices (fig. 3).

The normal is always pointing outwards, perpendicular from the front face. If the constructed object is not closed, then the solid flag needs to be false. This tells the browser to draw both faces. If the object is solid however, there is no point in rendering faces no one can see.

2.8 Lighting

To achieve lighting effects, we need to define some light nodes which can be either DirectionalLight, PointLight or SpotLight. All lights have the following common fields: color, ambientIntensity, and intensity. The color is simply the color of the light. Intensity ([0,1]) defines how bright the light is, and ambientIntensity ([0,1]) tells us how much light it contributes to the general ambient light in the scene. Ambient light is light that shines on every surface in the scene, simulating light scattered from other objects.

2.8.1 DirectionalLight

This is a light that shines from a uniform direction, like a set of parallel rays. This gives an effect like that of the sun, where everything is illuminated from the same direction. The DirectionalLight node does not have a location in the worlds space, it simply exists.

2.8.2 PointLight

A PointLight is light that emanates from a particular point in space, spreading equally in all directions. Unlike the DirectionalLight, this node actually has a position in space specified in its location field. The radius of the light is the maximum distance at which objects will be illuminated. Within this radius, the intensity of the light is affected by a set of attenuation constants according to the following formula:

$$1 / (attenuation[0] + attenuation[1]*r + attenuation[2]*r^2)$$

2.8.3 SpotLight

The fields of the SpotLight node have exactly the same effect as described above, giving a position, a maximum radius, and an attenuation within this radius for the light source. The difference between the SpotLight and the PointLight is that this node shines in a particular direction.
2.9 Behaviours

VRML97 also supports behaviour nodes that can tell the browser how to animate the world and how to interact with the user. This is handled by events. Most nodes contain events which are divided into two types, in- and out-events. In order to do useful things with events, they need to be connected somehow. This is known as routing. For example, to route a touchTime out-event to a startTime in-event (for instance to play a sound on a mouse click), the route could look like this:

ROUTE SENSOR.touchTime TO SOUND.startTime

This bit of code will route the touchTime event from a TouchSensor (capable of sensing mouse clicks on an object) to a startTime event in a sound node. Therefore, when the TouchSensor is clicked, the sound will play. Each node that is routed to or from needs to be DEFined so that it has an individual name. So, the TouchSensor and Sound nodes would be defined like this:

DEF SENSOR TouchSensor {
}
DEF SOUND Sound {
}

The VRML97 standard also supports scripts for even more complicated behaviours.

For more information about scripts, events, routs, sensors, etc. take a look at Floppy’s VRML97 Tutorial (2002) or some other VRML97 resource.
3 Example

The following code has been extracted from a VRML97 file generated from one of the 2D maps used by the robot. The IndexedFaceSet node defining all the obstacles/walls has been removed because it is similar to the Ground shape. The viewpoint has been calculated to start in a position where the whole map can be viewed at an angle of 45 degrees from above. The last two nodes define a timer and an animator responsible for rotating the world around origo. All fields have been defined for clarity, although some are never used.

---File Begin---
#VRML V2.0 utf8
Viewpoint {
  fieldOfView 0.785398
  position 0.0 -220.6 156
  orientation 1.0 0.0 0.0 0.785398
  description "v1"
}
DEF World Transform {
  translation 0.0 0.0 0.0
  rotation 0.0 0.0 1.0 0.0
  scale 1.0 1.0 1.0
  scaleOrientation 0.0 0.0 1.0 0.0
  center 0.0 0.0 0.0
  children [
    DEF Ground Shape {
      geometry IndexedFaceSet {
        ccw TRUE
        colorPerVertex FALSE
        normalPerVertex TRUE
        convex FALSE
        creaseAngle 0.0
        solid FALSE
        coord Coordinate {
          point [
            130.0 90.0 0.0,
            -130.0 90.0 0.0,
            -130.0 -90.0 0.0,
            130.0 -90.0 0.0
          ]
        }
      }
      color Color {
        color [
          0.0 1.0 0.0
        ]
      }
      coordIndex [
        0, 1, 2, 3
      ]
      colorIndex [0]
DEF Timer TimeSensor {
cycleInterval 15.0
enabled TRUE
loop TRUE
startTime 0.0
stopTime -1.0
}
DEF Animator OrientationInterpolator {
key [
  0.0
  0.5
  1.0
]
keyValue [
  0.0 0.0 1.0 0.0,
  0.0 0.0 1.0 3.14,
  0.0 0.0 1.0 6.28
]
}
ROUTE Timer.fraction TO Animator.fraction
ROUTE Animator.value TO World.rotation
---File End---

4 References

http://web3d.vapourtech.com/tutorials/vrml97/ (verified 2003-03-27)


http://www.web3d.org/x3d.html (verified 2003-03-27)
Appendix V - Projectivities

The idea of a projective plane can be applied to n-dimensional space in order to define projective n-space. Of particular interest is projective 3-space. Transformations within and between projective spaces are called projectivities and are the fundamental concern of projective geometry (Beardsley, 1997).

1 Perspective Projection to the Image Plane

Perspective projection is a projectivity from projective 3-space to the projective plane. It has the form:

\[
\begin{bmatrix}
    x_1 \\
    x_2 \\
    x_3
\end{bmatrix} = P
\begin{bmatrix}
    v_1 \\
    v_2 \\
    v_3 \\
    v_4
\end{bmatrix}
\]

where \((x_1,x_2,x_3)^T\) are the homogeneous coordinates of a point on the image plane, \(P\) is a 3-by-4 matrix, and \((v_1,v_2,v_3,v_4)^T\) are the homogeneous coordinates of a point in the world. Perspective projection is a particular type of projectivity called a perspectivity, in which all rays of projection pass through a single point - this puts constraints on the form of the matrix \(P\) (Beardsley, 1997).

2 Homogenous Coordinates

The equations for perspective projection to the image plane are non-linear when expressed in non-homogeneous coordinates, but are linear in homogeneous coordinates. This is characteristic of all transformations in projective geometry, not just perspective projection (Beardsley, 1997).

Homogenous coordinates utilize a mathematical trick to embed 3-dimensional coordinates and transformations into a 4-dimensional matrix format. As a result, inversions or combinations of linear transformations are simplified to inversion or multiplication of the corresponding matrices. Homogenous coordinates also make it possible to define perspective transformations (Woods, 2001).

Instead of representing each point \((x,y,z)\) in 3-dimensional space with a single 3-dimensional vector \((x,y,z)^T\), homogenous coordinates allow each point \((x,y,z)\) to be represented by any of an infinite number of 4-dimensional vectors \((\lambda x, \lambda y, \lambda z, \lambda)^T\).

The 3-dimensional vector corresponding to any 4-dimensional vector can be computed by dividing the first three elements by the fourth, and a 4-dimensional vector corresponding to any 3-dimensional vector can be created by simply adding a fourth element and setting it equal to one.
3 Plane-to-Plane Projection

A projectivity from a projective plane to a projective plane is called a plane-to-plane projectivity. It acts on, and generates, a homogeneous 3-vector and is therefore a 3-by-3 matrix (Beardsley, 1997).

To see how such a projectivity arises, consider two images taken from different viewpoints of a plane in a scene (fig. 1). The mapping of points on $\pi$ to the corresponding points in image 1 is described by a projectivity $T$. Similarly, the mapping of points on $\pi$ to the corresponding points in image 2 is described by a projectivity $S$. An important property of projectivities is that they form a group. It follows that there is a projectivity $R$ which describes the mapping of the image of $\pi$ in image 1 to the image of $\pi$ in image 2, where $R = ST^{-1}$.

![Figure 1 - Plane-to-Plane projectivities. Two images are taken of the plane $\pi$. There is a projectivity between each image and the plane, and between the image of $\pi$ in image 1 and the image of $\pi$ in image 2.](image)

Computation of a projectivity such as $T$ requires four points on $\pi$ together with their mapped points in the image. Let the projectivity $T$ map point $p$ to point $q$. These points $p$ and $q$ are normalised by scaling them so that their third components are equal to one. The scale of $T$ is arbitrary and is set by making $T_{3,3}$ equal to one. Then the mapping of $p$ to $q$ is described by

$$
p = \begin{bmatrix} p_1 \\
p_2 \\
p_3 \end{bmatrix} = T \begin{bmatrix} q_1 \\
q_2 \\
1 \end{bmatrix} = \begin{bmatrix} a & b & c \\
da & e & f \\
g & h & 1 \end{bmatrix} \begin{bmatrix} p_1 \\
p_2 \\
p_3 \end{bmatrix}
$$

where $p$ is a scale factor.

The matrix equation yields three linear equations, and eliminating the unknown scale factor leaves two linear equations in the eight unknowns $a, b, ..., h$,

$$
ap_1 + bp_2 + c - gp_1q_1 - hp_2q_1 = q_1
$$

$$
dp_1 + ep_2 + f - gp_1q_2 - hp_2q_2 = q_2
$$

Thus four pairs of points $p_i, q_i$, $i=1..4$, give eight linear equations in the eight unknowns and it is possible to solve for $a, b, ..., h$. The condition for linear independence of the equations
is that no three of the four points are collinear\(^1\). Given more than four pairs of points, the system can be solved using a least-squares method such as the pseudo-inverse.

4 Plane-to-Plane Mapping

Since the map is nothing more than a 2-dimensional plane in 3-space and the window is simply an image of the perspective projection of the map, the mapping between window/screen coordinates and actual map coordinates is analogous to the mapping between points on \(\pi\) and points on the image of \(\pi\) in image 1 in the example above (fig. 1). When the ground surface polygon (i.e. the map) is projected onto the screen, the location of the four corners are stored in memory. These four points, together with the original coordinates of the corners form the four pairs \(p, q\) necessary for solving the projectivity \(T\). The linear-equation system is solved using a LU factorization and back-substitution algorithm. Once solved, the plane-to-plane transformation matrix can be used to map any screen coordinate to its equivalent map coordinate by applying the inverse of \(T\) (i.e. \(p = T^{-1}q\)).

5 References

Beardsley P. 1997. *Important concepts from projective geometry.*

http://bishopw.loni.ucla.edu/AIR5/homogenous.html (verified 2003-03-26)

---

1. Three or more points are said to be collinear if they lie on a single line.
1 Introduction

This user’s manual is written as a brief walk-through of the most basic functions and presents the typical usage of the Pluto GUI. The manual is written as a stand-alone document which means that you do not have to have any prior knowledge about the Pluto project. But in order to understand the GUI better the following background information might be useful:

Pluto is a mobile, 4-wheeled, robot designed primarily for outdoor usage in an urban environment but could just as easily be used for indoor assignments. Pluto is constantly being upgraded and modified to handle different kinds of missions. Pluto currently supports the following functions:

- Autonomous map making of unknown environments.
- Manual driving.
- Sending Pluto to a destination, using internal map coordinates.
- Following an object around autonomously.
- Provide video images of the area in front of the robot (from the on-board camera)

The GUI provides the following functions:

- 2D representation of the internal map (which can be used to move the robot)
- 3D representation of the internal map.
- Capture the video image stream and show it to the user.
- Interface for driving the robot manually (with or without video relay).
- Interface for defining a new area to explore.
- Ability to toggle follow-me mode on/off.

The following extensions to Pluto are under progress:

- A robotic arm with a grappling device.
- Ability to follow a road from the current location to a given coordinate.

Possible extensions to the GUI are listed below:

- Interface for controlling the robotic arm.
- Ability to toggle road-follow mode on/off.
- Take and display snapshots from different locations.
2 Disclaimer

The screen shots in this user’s manual are optimized to look good in print. They might turn out grainy or distorted depending on the zoom of your PDF browser and/or the resolution of your screen. All screen shots can be found at my web catalogue: www.nada.kth.se/~t98_hes/cas/gui/screenshots/

The 2D and 3D Map screenshots has been converted into grayscale and then inverted (black becomes white, white becomes black) to save toner/ink when printing.

3 Start-up

When launching the GUI you will be asked the following question:

Depending on your answer, the following will happen

- If you answer yes, the GUI will try to verify that a connection to the robot can be made. This is important, because we do not want to try and send messages to Pluto if we are not connected. Trying to connect to the robot server will freeze the GUI for several seconds while waiting for the time-out call. The following info message is displayed to inform the user of this.

If the GUI could make a connection to Pluto an online flag is set to unlock all the GUI features.

- If you answer no, or if the network connection test returned false, the GUI will not set the online flag and most of the features will be locked. The purpose of running the GUI offline is to ease debugging, but also to view locally saved maps and/or screen shots (not implemented yet).
4 Control Panel

This is the main window of the GUI. From here you can launch all the tools, display all the windows and change the settings. At the bottom of the window you see the network status flag.

5 Manual Drive

This window enables you to drive the robot manually with the interface shown below.

(The quality of the video will normally not be as high as this screen shot indicates due to performance issues. In order to keep the frame rate at a maximum the video frame compression needs to be set relatively high resulting in grainy images but this would look bad in print so I chose to drop the image compression for this screen shot. The compression ratio can be changed in the settings.)

On the left side of the window we have the video feedback. This is very useful if the robot has stopped moving and is out of sight. The video can be toggled on/off by pressing the grey button under the video frame. Above the video frame you see a white compass arrow indicating the orientation (i.e. the direction) of the robot. Again, this could be useful if the robot is out of sight or if you want to align the robot before a new explore mission.

On the right side we have the driving interface. It is designed as a regular RC$^1$ controller used for RC model racing cars. Only instead of two one-DOF$^2$ levers, one for up/down
and one for left/right, they are combined into one two-DOF controller. To move forward, press the up arrow. To move backwards (or slow down), press the down button. To turn left/right press the left/right arrows. Unlike a normal RC car however, the robot can rotate on the spot and does not require any forward (or backward) speed to turn. Another thing that is different is that you do not have to hold down the buttons for the robot to move. It will register how many times the button has been pressed and show the current speed (depending on the robot reply) as progressively filled arrows. To stop the robot you simply press the (red) centre button.

In the middle you see the information. Maximum speed (changed through the Control Panel\textsuperscript{1} window), current speed (forward) and current turning rate. The turning rate can be a bit confusing at first, since it is measured in radians per second and this is not something normal people can relate to. The forward/backward speed is in meters per second and this is pretty obvious. Another thing about the turning rate is that it is defined as positive for CCW\textsuperscript{2}, which means that rotating/turning to the right will give a negative speed. The turn rate is only presented to complete the interface. For most of the users the filled arrows will suffice.

6 Explore Area

This dialog window allows you to send the robot out to explore the world.
The little (red) square symbolizes the robot. And the big (white) square is the area about to be explored. You can change the scale of the area by using the +/- buttons. Once you press the Explore button, the dialog will close, an explore command will be sent to Pluto and the 2D Map\textsuperscript{1} window will be launched. The dialog window also provides some information about how the exploration works, like for instance:

- To view changes in the map you need to press the Reload Map button.
- Saving the map will take several minutes during which the robot will remain frozen.
- Pluto will continue to explore until it either has covered the entire area, is stopped by the user, or receives a goto command from the 2D Map window.
- Explore can be resumed at any time (unless while saving the map) by pressing the Resume Explore button.

7 Follow Me

You can activate or deactivate (default) the Follow Me mode of the robot. When activated, Pluto will try to follow the object right in front of him. This enables Pluto to be used as a mule, carrying heavy equipment for you. One of the two following messages will be displayed to indicate if the mode is active or not.

8 Road Follow

This feature has not been implemented yet.

---

1. Section 9 on page 79
9 2D Map

This is what you see when you first launch the 2D Map.

The map has been centred on the robot (the little red icon) and its current (internal) coordinates and orientation is displayed in numbers in the top-left corner. In the bottom-left corner you can check the status of the robot. The (white) arrow in the top-right corner is a compass arrow, pointing towards north (relative to the map, not the PDA). By using the two rotation buttons you can rotate the map and align it to your current orientation in the real world, almost like if you had a real map in your hand. This feature has not been implemented yet though. In the bottom-right corner you see a scale indicator, helping you to determine distances in the map. The only thing that does not follow scale is the robot which is drawn at approximately three times its natural size in order to show the position and orientation more clearly.

9.1 Buttons

On the bottom of the screen, you have all the buttons. They are all explained (in left-to-right order) below:

9.1.1 Centre On Robot

This button will centre the map on the robot.
9.1.2 Zoom In / Zoom Out

These two buttons allow you to change the resolution of the map. If you zoom out too far the robot icon will become a red square, as you can see in the following screen shot.

The reason for this is that the robot should always be visible.

9.1.3 Move Viewpoint Left/Up/Down/Right

Using these buttons, you can move the viewpoint a small increment in the given direction. Remember that it is the viewpoint (i.e. your head) that is moving and not the map. The map appears to move in the opposite direction. The design follows the standard of common web browsers and other text viewing tools, where the slider moves the viewpoint and not the page (think about this for a second or try it yourself).

You can also move the viewpoint by pressing somewhere on the map, forcing the map to re-centre on this position.

9.1.4 Rotate Viewpoint CCW/CW\(^1\)

This feature has not been implemented yet.

9.1.5 GoTo

Pressing this button once, and then pressing somewhere on the map, you will send Pluto on a mission to reach the given destination. As the following screen shot shows, the goto

\(^1\) Counter Clock Wise / Clock Wise
coordinates are marked as a tiny (red) square, and written in numbers at the bottom-left side of the screen. The status is also updated.

Once the robot receives a goto command, it will stop doing whatever it is doing (unless it is saving the map. Saving the map has the highest priority of all robot tasks.)

9.1.6 Stop

This button will send a stop command to the robot, telling it to stop moving. The command will also have the side effect of aborting any currently active exploration.

9.1.7 Reload Map

This button will first send a save map command to the robot, resulting in that the robot will abandon whatever it was doing and start to save the map instead. This can take several minutes depending on the size of the map and how long ago it was since the map was last saved. During this process the 2D Map window can not be used. Before saving the map, you will see the following message:

and once the map has been successfully saved the following message will be displayed:
Once you dispose of the message box, the robot will resume any previously aborted explore and the GUI will load the newly saved map and continue to update the robot position.

**9.1.8 Resume Explore**

Pressing this button will send a *resume* command to Pluto, telling it to resume any explore that has been previously aborted or stopped, for instance while moving from one position to another (see GoTo\(^1\)).

**9.1.9 Explore Area**

Pressing this button will launch the *Explore Area*\(^2\) dialog.

**9.1.10 Show Snapshot**

By pressing this button once and then on any of the snapshot markers (shown as blue robot icons on the map) a new window will be launched, showing the snapshot image and information.

This feature has not been implemented yet.

---

1. Section 9.1.5 on page 80
2. Section 6 on page 77
9.1.11 Set New Pose

If you zoom down to 1 meter on the scale indicator, a new button will appear in the bottom-right corner as the following screen shot illustrates.

This button allows you to reposition the robot on the map (if it does not know where he is). As for now, you can only set the coordinates, and not the orientation, so the robot must face north. When you press the button, a message box will inform you of this as well.

Use the Manual Drive\(^1\) window to drive the robot to a place you know and can locate on the map. After you have dismissed the message box, point on the map to reposition the robot. If the position you pointed at is sufficiently close (~15 cm) to the actual robot position, Pluto will understand this and auto update its new position to the actual position.

---

1. Section 5 on page 76
10 3D Map

This is what you see when you first launch the 3D Map.

The model will rotate in a moderate pace so you can view the whole map and get an intuitive feeling of how the world is connected. For now, there is only wire-frame rendering (that is, only the corners of the walls are drawn) but flat rendering will be available in a future version.

10.1 Buttons

On the bottom of the screen, you have all the buttons. They are all explained (in left-to-right order) below:

10.1.1 Reset

If you press this button, the viewpoint (which is your position as an observer in the 3D world) will be reset to its initial settings and the world will begin to rotate (again).

10.1.2 Pause

If you press this button the world will stop rotating. This must be done before you can begin to move around in the world (i.e. move the viewpoint).

10.1.3 Move Viewpoint Left/Up/Down/Right/In/Out

Using these buttons, you can move the viewpoint a preset increment in the given direction. Unlike the 2D Map\(^1\) window, where the movement is relative to the current zoom level, the increment is always the same. This is a major drawback if you are moving in the xy-plane, following a street for example. The reason for this is that constant movement is the simplest solution. There is no good measure that covers every possible situation. For

---

1. Section 9 on page 79
instance, you want to be able to move fast (large increment) if you are far away from the target point, regardless of your current z position.

Remember that it is the viewpoint (i.e. your head) that is moving and not the world/map. The world appears to move in the opposite direction. Just like the 2D Map the design follows the standard of common web browsers and 3D browsers. In the following screen shot the viewpoint has been moved inwards.

You can also move the viewpoint by pressing somewhere on the map, forcing the viewpoint to enter a birds-eye point-of-view centred over the pointed position.

The browser will remember the current height (z-axis value) and rotation (z-axis rotation angle) so if, for instance, you move the viewpoint closer and then click on the map again it will re-centre the viewpoint using the same height and rotation. All the other DOF\(^1\):s will be reset.

---

1. Degree Of Freedom
10.1.4 Rotate Viewpoint Around The Z-axis

The z-axis is the axis pointing outwards from your screen surface. Rotating CCW\(^1\) has the same effect as if you tilted your head to the left in the real world.

10.1.5 Rotate Viewpoint Around The X-axis

The x-axis is the horizontal axis pointing to the right of your screen. Rotating CCW has the same effect as if you leaned your head backwards (to look at the stars perhaps?) in the real world.

10.1.6 Rotate Viewpoint Around The Y-axis

The y-axis is the vertical axis pointing upwards from your screen. Rotating CCW has the same effect as if you turned your head to the left in the real world.

The following screen shot shows a viewpoint that has been moved inwards and downward and then rotated around the x-, y- and z-axis.

---

1. Counter Clock Wise
11 Video Camera

This window shows the video frames as fast a possible, unlike the Manual Drive\textsuperscript{1} window where bandwidth for driving interface is also considered. This is what the window looks like.

(As with the Manual Drive window, the quality of the video will normally not be as high as this screen shot indicates due to performance issues. In order to keep the frame rate at a maximum the video frame compression needs to be set pretty high resulting in grainy images but this would look bad in print so I chose to drop the image compression for this screen shot. The compression ratio can be changed in the settings.)

11.1 Buttons

On the bottom of the screen, you have all the buttons. They are all explained (in left-to-right order) below:

11.1.1 Take Snapshot

If you press this button, the video stream will freeze for a few seconds while the current frame is being saved in memory together with the robot information at the present time. Once a snap shot has been taken, the window continues to show the video stream. The snap shots can be viewed by pressing the Show Snapshots button.

This feature has not been implemented yet.

11.1.2 Show Snapshots

Pressing this button will launch a new window, responsible for showing all the available snap shots stored in memory.

This feature has not been implemented yet.

\footnotesize
1. Section 5 on page 76
12 Set Max Speed

This dialog allows you to set the maximum speed of the robot.

The speed can not be higher than 1.5 m/s. The settings will take affect immediately.

13 Change Settings

This feature has not been implemented yet. It is suppose to be a dialog window where all the settings can be changed. For now, all the settings must be set using a text editor, modifying the settings.ini file in the application root directory. But at least this gives you the ability to change the settings as oppose to not at all.

14 Collision Avoidance

Setting the collision avoidance is very important and is activated by default. But in some situations you need to turn it off or you will not be able to move the robot. For instance, driving through a doorway can not be done with collision avoidance activated. You will get one of the following two messages indicating whether collision avoidance is activated or not.