A Managing Agent for Sharing Multiple Modalities

Multimodal user interfaces promise more natural man-machine communication, as well as improved accessibility for disabled people. When extending multimodal user interfaces beyond single application scenarios, providing a multimodal interface in a desktop, applications have to share access to input and output devices. This requires an agent that arbitrates between applications and low-level drivers. This thesis proposes a design for communication between driver, manager and application, and discusses a prototype implementation and preliminary testing results.

Graduation Committee:
Prof.Dr. H. Koppelaar
Drs.Dr. L.J.M. Rothkrantz
Prof.Dr.Ir. E.J.H. Kerckhoffs

Boi Sletterink,
Delft,
August 1999

Acknowledgements

This report is the thesis for my masters degree in Computer Science from the Knowledge Based Systems group at Delft University of Technology, headed by Prof. Dr. H. Koppelaar. The research and work itself was performed at the CAIP center at the Rutgers University, the state university of New Jersey, in the USA.

I would very much like to thank Dr. Attila Médl, assistant professor at CAIP, who has done a great job in being my mentor. I enjoyed working with him, and his advice and support has been really valuable. Discussions with him have been very insightful and interesting.

Many thanks to Prof. Dr. James L. Flanagan, director of CAIP, for his hospitality and support at CAIP center. It has been a pleasure to work in his center. Also many thanks to my former colleagues (you know who you are), who have been very pleasant to work with.

I would like to thank the people at Delft University who have helped me, and especially Dr. Drs. Leon J.M. Rothkrantz, who has been most supportive and stimulating. I am very grateful that he arranged such a great chance to study abroad, and for guiding me through the last year of my study.

I would also like to thank CAIP center, Delft University of Technology, the faculty of Information Technology and Systems, and the Fundatie van de vrijvrouwe van Renswoude te Delft for providing financial aid.

Last but not least, I would like to thank my family, friends and roommates (no matter where you are now) for being nice in general and putting up with me in times of stress in particular.

Delft, August 1999

Boi Sletterink

Glossary

Terms

multimodal using two or more modalities in a serial or parallel fashion.

modality
1. in human factors studies, a sense; examples are sight, sound and touch.
2. in application designer's and programmer's parlance, an input or output device.

interface
1. in HCI, the point where interaction occurs between the human and the computer.
2. in programming, the boundary between two modules. Also used as the set of externally visible operations on a module (esp. in Java).

Package a set of (Java) classes that form a unit and share a common package name.

Abbreviations

MIM Modal Input Manager

MM Modality Manager

NLI Natural Language Input

HCI Human-Computer Interaction

GUI Graphical User Interface

AWT Abstract Windowing Toolkit (Java's old GUI package)

JNI Java Native Interface (tools for calling native [non-Java] code fom within Java)

JDK Java Development Kit

Contents

Chapter 1

Introduction

1.1 Problem Statement and Motivation

Pervasive digital networking promises to bring the power of distributed computing to a broad sector of society. Concomitant is the extension of human intellect through sophisticated communication and computer-mediated collaboration. But mass deployment depends upon ease-of-use – being able to communicate with machines in ways the human finds convenient. In natural human-human communication, sensory modalities of sight, sound, and touch are favored. Hence, there is strong incentive to emulate the naturalness of face-to-face information exchange in the design of user interfaces.

Individual technologies for human-machine communication, though imperfect, have advanced to a stage where they can be profitably integrated to produce capabilities that go substantially beyond traditional mouse and keyboard. Eye tracking, visual gesture, and "region-of-interest" segmentation provide new dimensions of control in the sight domain. Automatic speech recognition and text-to-speech synthesis give interactive conversational ability to machines. And, tactile force-feedback and manual gesture enable natural grasp and movement of icons in shared workspaces. Intelligent software agents are becoming adept at fusing these simultaneous sensory inputs into reliable estimates of user intent – as the human does in face-to-face exchanges.

Therefore, we find it important to investigate multimodal man-machine communication by building a prototype system, in which multiple interface modalities are implemented for sight, sound and touch communication. The system incorporates an automatic desk-mounted eye tracker for visual control of screen cursors; a microphone array for "hands free" sound capture; speech recognition and synthesis for conversational interaction; and a tactile force-feedback glove for gesture and grasp-control of virtual objects (figure 1.1).

A major challenge in designing a multimodal interface is making it sufficiently advanced so that it can handle a large number of modal inputs, simultaneously and in combination, in a desktop environment with multiple applications. In order to achieve this, we need a multimodal input manager. This manager should be able to control how applications interpret events triggered by the multiple modal input objects. It should also be able to decide which application has control, and send all modal inputs to that application. To further simplify matters, each modal input object should be defined and implemented by a common specification used by all input objects.

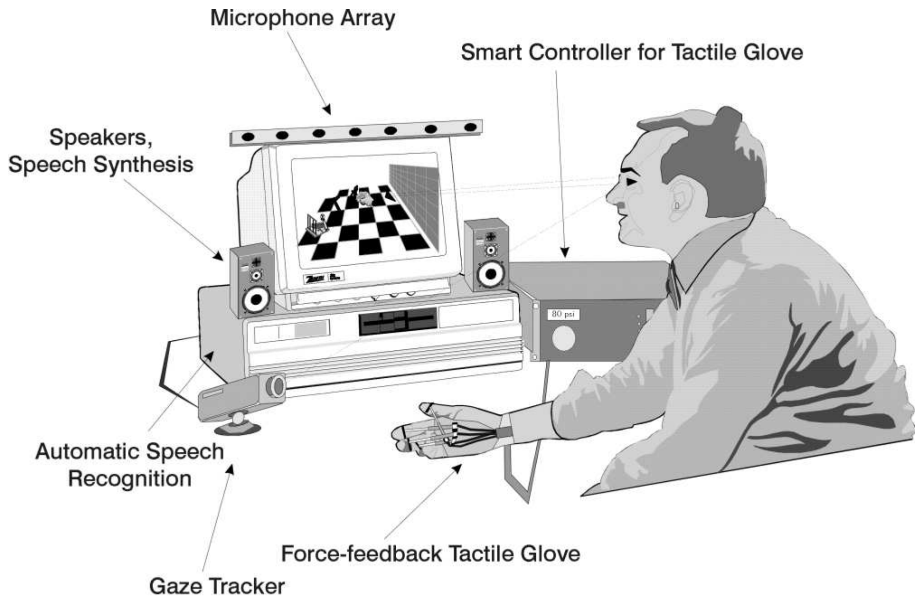

Figure 1.1: The multimodal workstation concept

In order to demonstrate real-time operation of the system, we need collaborative applications using multiple modalities. We also find it crucial to collect data from human factors evaluations and compare them against baselines for mouse and keyboard.

The goal of this thesis work is therefore:

1. to establish both software and user interface requirements for a multi-modal collaborative computing environment within a legacy GUI setting;
2. to establish a flexible prototype programming environment to develop the applications and the modality management agent in;
3. to develop prototype applications and a prototype modality manager, and gain insight in problems related to modality management;
4. to employ the prototype for user testing and gain insight in human-computer interaction in multimodal collaborative environments.

The rest of this introduction discusses background information, project status at the beginning of the thesis work, and an overview of related research. Chapter 2 discusses the requirements and the conceptual design, chapter 3 discusses the design of the programming environment, and chapter 4 discusses the implementation of the multimodal managing agent. Chapter 5 briefly discusses the implementation of multimodal extensions in example applications, as well as user test results. Finally, chapter 6 summarizes the results of this thesis work.

1.2 Background

This thesis is to obtain a Master of Science degree in Computer Science from Delft University of Technology (DUT). The work described here was made possible by the collaboration of DUT and the Center for Computer Aids for Industrial Productivity (CAIP) at Rutgers University.

1.2.1 Knowledge Based Systems Group, Delft University of Technology

The Knowledge Based Systems (KBS) Group at Delft University of Technology is part of the Department of Mediamatics. This department is part of the faculty of Information Technology and Systems, which also has departments in the fields of Computer Science, Mathematics and Electrical Engineering.

The KBS group is headed by Prof. Dr. Henk Koppelaar and is active in several AI fields, including spoken language processing, non-verbal communication, case based reasoning and AI techniques for financial modeling.

1.2.2 CAIP Center, Rutgers University

The Computer Aids for Industrial Productivity Center at the Rutgers University (the state university of New Jersey), is an Advanced Technology Center that conducts research in the application of computing technology and collaborates with industry and government to transfer the result of university research to societal benefit. CAIP is supported by the Rutgers University, the New Jersey Commision on Science and several industrial member including Lucent Bell Labs, AT&T Labs, Sun Microsystems, IBM and other leading information technology companies.

The center is headed by Prof. Dr. James L. Flanagan and has groups working in several fields, including human-machine interfaces, computer vision and multimedia information systems.

1.3 The STIMULATE project

The STIMULATE project [13] is a NSF funded project (contract number IRI-9618854) at CAIP to explore issues in human-computer interaction. STIMULATE stands for Synergistic Multimodal Communication in Collaborative Multiuser Environments.

The project description in the original proposal is:

> *The objective of this NSF-sponsored STIMULATE project is to establish, quantify, and evaluate techniques for designing synergistic combinations of human-machine communication modalities in the dimensions of sight, sound, and touch for collaborative, multi-user environments.*

For the collaborative multi-user environment, we use work from the DISCIPLE project [12] [26], which is another project at CAIP. Re-using work from another project saves us time and since that work is done at CAIP as well, we already have all the knowledge about the environment. In turn, the DISCIPLE project benefits from our work.

Figure 1.2: The multimodal workstation in action

1.3.1 Multimodal workstation

To have a testbed for designing multimodal applications and performing measurements, we developed a multimodal workstation (figure 1.2). In June 1998, this multimodal workstation was composed of the following components:

- A Sun Ultra-1 running our Java program, the collaborative whiteboard.
- A microphone array and a set of speakers for use with the speech recognition and synthesis software.
- an Intel-based PC for running the speech software.
- A tactile glove with position detection. This is the Rutgers Master II system developed by dr. Burdea [6], who is also involved in the STIMULATE project.
- A gaze tracking device to measure eye movement developed by ISCAN, Inc. [21]

1.3.2 Software

In June 1998, we used the following off-the-shelf software:

- Solaris 2.5 for the Sparc.
- SunSoft's JDK 1.1.6 for Solaris.
- Microsoft Windows NT 4.0 with service pack 3 for the PC.
- Microsoft Whisper [19], based on SpeechSDK 3.0. This is the speech recognition and synthesis software.

Our own software consisted of three drivers and the application. The drivers interface with the hardware or, in case of the speech subsystem, with the speech library. The drivers communicate with the application through TCP/IP connections. This is an easy, effective and flexible way to have the modalities available on different platforms.

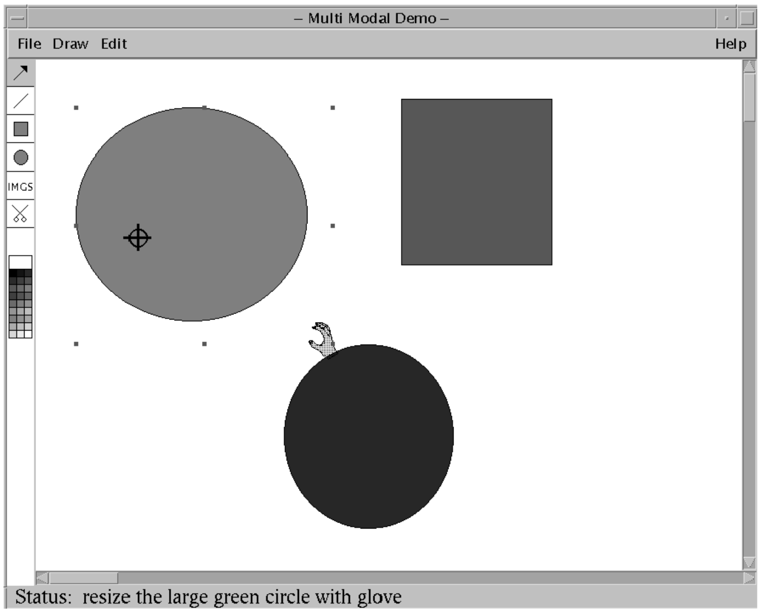

Figure 1.3: Snapshot of the whiteboard application

1.3.3 Application

The application is a collaborative whiteboard (Figure 1.3), with extensions for military mission planning (Figure 1.4) [3] . Users can place lines, circles, rectangles and military icons on a background map. Objects can be moved or resized. The application can be used with the mouse or with a combination of the speech recognizer, gaze tracker and tactile glove.

The whiteboard uses a slot-filling mechanism to interpret the commands. Commands can be generated by keyboard, speech or with gestures of the glove. All commands are natural sentences, gestures are translated into text. Objects to perform the command on can be indicated with keyboard, speech, gaze or glove.

The multimodal fusion is "hardwired" into the parser. Application specific knowledge is condensed as Java code into the parser. The parser and understanding module is therefore written specifically towards the whiteboard application. It is tightly coupled with the other modules.

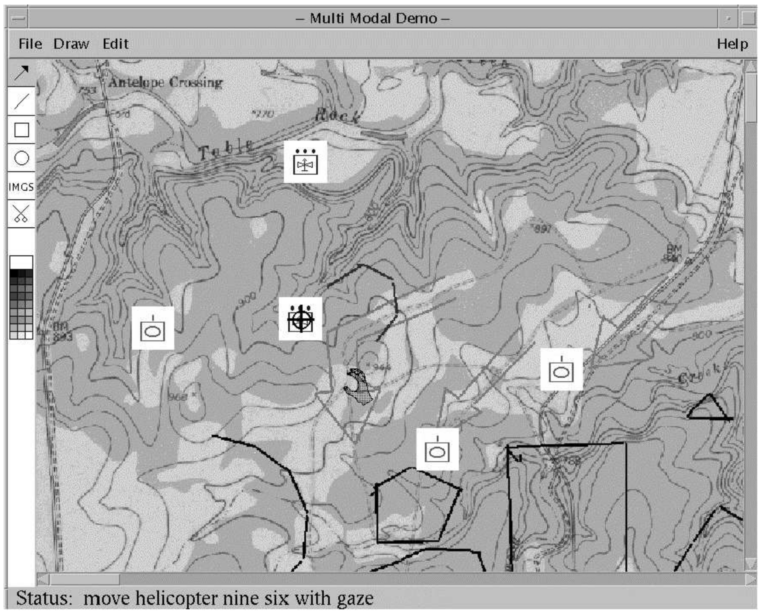

Figure 1.4: Snapshot of the whiteboard with military map extensions

1.3.4 Improvements

There were several ways in which the multimodal workstation could be improved. Following is a brief discussion of several issues.

Improved multimodal fusion The multimodal fusion could be improved by studying human-human communication and trying to incorporate more behavior in the fusing agent. This leads to more natural communication.

This is a relatively new and unexplored area with several groups active in this field. The advances in computing power and problems with speech-only systems have increased interest in this subject. Most approaches for this problem only involve one other modality besides speech, and are very application specific.

Improving the design of the natural language interface To make the natural language interface more natural, a more radical approach would be needed. A design using a number of agents, coupled by a blackboard could be used. Using advanced AI techniques, they make decisions based on their own inputs and knowledge, as well as information they get from other agents. This systems should also allow for a more conversational type of interaction.

Work in this area is performed by many groups. It is a very complex issue with many unresolved problems. It is a classic AI problem, and as such, there are several methods for attacking this problem. Still, there is no be-all end-all solution to this.

Porting to the new DISCIPLE platform The DISCIPLE project was moving towards a new collaborative platform, which supported multiple applications in a desktop other than just a library supporting one application. It was necessary to port or write a similar application for the new environment.

This would be a relatively straightforward programming exercise. However, one of the ideas of the desktop environment, running multiple applications, is limited by this; with that setup, only one application could be multimodal.

Support for multiple multimodal applications Another idea for bringing the multimodal technologies to the desktop environment would be to implement a managing system for handling the modalities. With such a system, multiple applications can use the same modalities. This is conform the idea of windowing systems, where a window manager handles user interface resources, such as the mouse, the keyboard, and the screen.

Although the concept is similar to existing window managers, it is very different with respect to the way multiple modalities complement each other and can (partially) replace each other. There is not much work done in this area, so there are no well-known methods for dealing with this, other than extrapolating techniques for conventional window managers.

Minor improvements Some less radical improvements were desired as well. These include 3D position and direction support for the glove, haptic feedback for the glove, a better and cross-platform speech recognizer, more accurate gaze tracking and easier startup procedures.

Considering the needs and what was practically possible, we chose to work towards the new DISCIPLE platform and implement a multimodal manager, along with some minor improvements, and adapt some test applications to use the manager.

1.3.5 Comparison with Other Research

Integration of multiple modalities in human-machine interfaces has for a long time been viewed as a means of natural communication with machines [4, 22]. Individual technologies for more natural human-machine communication have only recently matured enough to be employed in freeing computer users from the constraints of the keyboard and the mouse [14]. Properly combined in selected applications, these human-machine techniques can extend the capabilities of human users [16, 15, 33]. Some examples of recent work in multimodal human-machine interfaces include integration of two modalities such as speech and gesture [10, 30, 32], text and images [27] or that aim to combine modalities of speech recognition and language understanding, gaze tracking, lip reading and gesture recognition [37, 40]. However, the proposers' work [2, 17, 28], presently evolving under the NSF-STIMULATE sponsorship, has established technology for synergistic fusion of multimodal data from *gaze, speech, and tactile* inputs. It is believed that this evolving capability is at the forefront of combining sight, sound and touch modalities for natural-like communication between the human user and the complex information system that is serving the human.

Researchers at MITRE have developed the "Virtual Interactive Environment" (VIE) which uses the Immersive Workbench table-top stereo display manufactured by Fakespace Co. [41]. VIE was used for visualization in a C2 center for Air Defense Operations in the Cheyenne Mountains. Interaction is mediated by a 3D stylus, or optional Pinch Gloves also sold by Fakespace Co. The user is thus able to select, "grasp" and manipulate objects in 3D but *without force feedback* or display of related information in pop-up windows. There is *no* voice pickup, *nor* gaze tracking. The Immersive Workbench has also been integrated in a C2 system at Navy Research Laboratory for Marine maneuvers planning over the terrain of the Marine Corps Air Ground Combat Center [29, 31]. In this application, interaction was through a joystick which projected a laser pointer on the map, allowing zooming and rotation by pressing a button. One drawback, as pointed out by the Marine commanders that used the system, was the lack of haptic interaction.

The Carnegie Mellon University does research on multimodal systems [37, 38, 39, 40]. They developed a multimodal interface for an appointment scheduling task on a computerized calendar. The user can use a combination of spoken input, gesturing with a pen on a touch-sensitive screen, or handwritten words to interact with the system. Their vocabulary is restricted and they did not integrate gaze tracking into the system [9].

The Human-Computer Interaction Laboratory at the University of Maryland focuses mainly on visual interfaces rather than on multimodal systems [34, 35, 36]. They developed a user interface for highway traffic management that provides round the clock monitoring of highway traffic. The decision features of the system could also be used in military applications [20].

The role of natural language is investigated by researchers at the Oregon Graduate Institute of Science and Technology [10, 30]. But they provide little support for identifying objects not on the screen, for specifying temporal relations, for identifying and operating on large sets and subsets of entities, and for using the context of interaction. On the other hand, these are precisely the strengths of natural language. They tested speech and gesture integration in a military planning and simulation scenario.

The SRI Multimedia Interfaces Group is currently building a series of prototype map-based applications that accept handwritten, verbal and gestural requests [7, 8]. These applications are distinguished by comparatively rich natural language capabilities, access to existing data sources including the World Wide Web, and a mobile handheld interface. Their system does not integrate and use gaze or tactile glove as inputs.

The Beckman Institute at the University of Illinois conducts basic research in human-computer interaction studying input-output techniques and human factors involved in the interchange [32, 33]. They developed an accurate eye-tracker for small eye movements, a lip-reader for enhancement of speech recognition and a speech interpreter. Scientists there can collaborate on developing new visualization techniques within a virtual reality environment called the CAVE, maintained by the National Center for Supercomputing Applications.

Chapter 2

Requirements and Concept

This chapter discusses the requirements on the design and implementation of the modality manager. It is split into four parts. Three parts each view the requirements from a different perspective: the project's, the user's and the programmer's point of view. The requirements are presented in plain English rather than in a formal description language, since most of them are really hard to specify in a formal language. The last part describes the resulting design concept.

2.1 For usability in the project

The manager should work in the collaborative desktop environment that was (at the time) to be developed. This means that the manager should work in a Java environment that has the notion of a workspace with multiple applications.

It should not force every application in the workspace to be written specifically for an environment with the manager. On the other hand, the modality management interface for applications should also be such that modality management-aware applications can work without the manager. In that case, they will just not have the alternative modalities available.

2.2 From the user's point of view

From the user's point of view, the manager should be a component that directs the modal inputs and outputs to the application components in a natural and consistent fashion. It must be configurable, active all the time but not encumber or inhibit performing tasks in the old-fashioned "mouse-keyboard-GUI" style.

This implies the following constraints:

- Pointer events (motion and actions) must be delivered to the components in which they occur (in Java/AWT terminology, they must be sent to the source).

- Modalities that do not have a specific screen position as part of their state or events, need a `focus`. For example, the keyboard does not have a screen coordinate as part of its output. GUIs have solved this problem by having a focussed component that gets the keyboard input. This same concept also applies to other non-pointer based modalities.

  All non-pointer modalities should share the same focus. If they would all have their own focus, the user has to remember where they are or look at the screen to find out what component is focussed for modality X. This is confusing. Also, multiple visual markers will clutter the screen. This causes unnecessary stress for the user.

- The manager must have a visual representation. This representation must allow the user to change settings, turn modalities on and off etc. Since our environment is speech-enabled, the manager should also have voice commands and feedback.

- The visual representation must be on the screen at all times, so the user can adjust the manager's behavior at all times. It should not take too much screen space nor should it alter the appearance of other components.

- The new modalities are *alternatives* to existing modalities, not necessarily complete replacements. The mouse and keyboard must still be usable.

2.3 From the programmer's point of view

Programmers would like to have a simple and clean interface, a decent level of abstraction and the flexibility to handle modal inputs as they see fit. They should not be forced to handle every possible human-computer interface device in existence, but they should be able to do so if wanted.

Applications should be ported easily, preferrably with an add-on component such that it will run without trouble on a system without the manager or even the libraries involved.

It must be possible to correlate different inputs of different modalities in the time domain. This is because not all modalities are completely real-time. For example, the speech recognizer may sometimes need a second or two to find the match.

These features would imply the following requirements:

- For programming flexibility, Java `interfaces` should be used instead of `classes` at most times, since you can inherit multiple interfaces but you cannot inherit multiple classes. This is because Java does not support multiple code inheritance, only multiple interface inheritance.

- The implementation should be split into two parts: a light-weight core library which defines the interactions between the modalities, manager and applications, and a separate implementation of the actual manager and modality interfaces. This allows applications to implement their multimodal support without depending on the actual implementation. It should also allow on-demand loading of the manager.

- There should be support for faking a pointer-based modality by sending fake mouse events. This makes it a lot easier to add multimodal support to existing applications.
- There should be timestamps on all events, and a history buffer for events to support interaction between events that are not synchronized.
- There should be a mechanism to find out what modalities are available and what their properties are.

2.4 Conceptual design

The concept for the design of the Java classes is shown in figure 2.1. Round boxes are libraries, shadowed boxes are implemented objects. Objects and libraries inside libraries indicates dependency. Arrows indicate methods being called by an object.

Everything is built with the Java2 library (a.k.a. JDK 1.2). This is the latest standard Java class library (at time of writing). The DISCIPLE desktop (not shown in the figure) and the applications are written with the Java2 library but not with the MM (modality management) library. All classes that are part of or deal with the MM subsystem use the MM library.

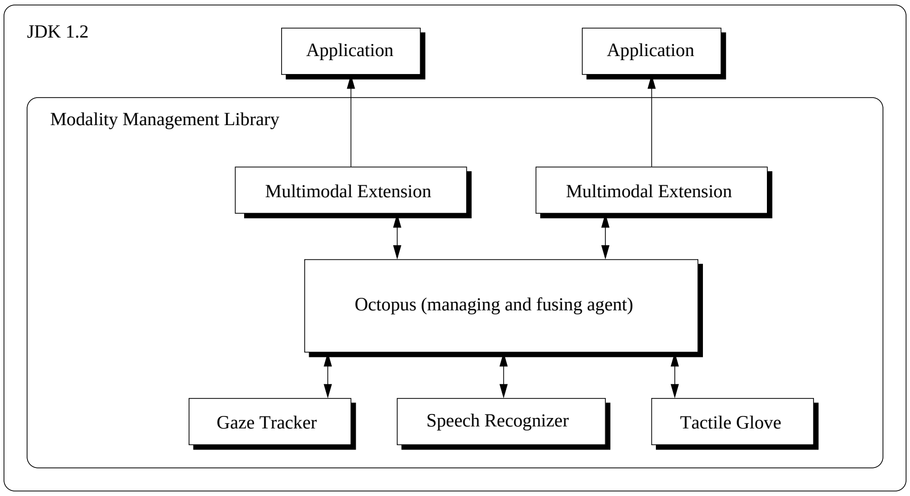

Figure 2.1: Conceptual view of components and libraries

The desktop nor the applications need to implement the multimodal extensions. Those can be implemented in separate classes and even separate packages. The main advantage of this split setup is that the desktop, which was almost fully developed at the time, and the applications, which were being developed in parallel, could develop separately from the development of the design and the implementation of the MM library and manager.

Work for this thesis was therefore confined to the design and implementation of the MM library, the managing and fusion agent (the Octopus), and the multimodal extensions (everything within the MM library box in figure 2.1). Building and improvement of the desktop and the applications was done by other people in our team. For writing the multimodal extensions, the respective authors were also involved.

Since the modality management library and the actual implementation are separated, they are discussed separately. Chapter 3 deals with the design of the modality management library and chapter 4 deals with the implementation of the Octopus.

Chapter 3

Design and Implementation of the Modality Management Libraries

This chapter discusses the design of the Modality Management libraries. The libraries provide a framework for the modalities, manager and applications to communicate with each other. The library does not perform any of this communication by itself; it is more like a protocol between the different components. The implementation of the manager and the modality interfaces (based on this library) is discussed in chapter 4.

Since it has to work with the DISCIPLE desktop, it has to be written in Java with version 1.2 of the class library (JDK 1.2). The multimodal library makes only a few assumptions, being:

- The manager works in an environment with multiple applications.
- There is a visual component for every application, containing all other visual components of that application.
- All applications use Java's latest visual component library, Swing, instead of the older AWT library.

These assumptions are consistent with the implementation of the DISCIPLE desktop at designtime.

As described in chapter 2, the basic idea and most of the features were defined in Java classes and interfaces, as a library, before the actual implementation of the manager, modality interfaces and applications. However, during the implementation, we found that some features were missing or insufficient and changes were made in the library to accomodate these necessities.

3.1 Notation and conventions

This section discusses notation and conventions used to describe Java structures in this thesis.

Classes and packages

For those not familiar with Java's concepts of modularity: classes can be gathered in a `package`. Each package has a name, consisting of a series of words separated by dots (a.nice.package). A class' full name is then the package name, another dot, and the class name (a.nice.package.ExampleClass).

Classes that are not in the class' own package have to be `imported` before they can be referred to by their short name. All classes in a foreign package can be specified by appending an asterix: `package.*`. The official naming convention for packages is as follows:

java.* for all packages that are part of JDK 1 and above.

javax.* for packages that are official and supported Javasoft extensions of the JDK, such as Swing and Java3D. Some packages may actually be part of JDK 1.2 and up, like Swing.

organization.internalname.* for anything else. The organization should be the reversed internet domain name.

In our case, this leads to the package basename `EDU.rutgers.stimulate` for the entire collection of multimodal packages. The package name for the multimodal management library is `EDU.rutgers.stimulate.MIM`. If the package name for a class is not mentioned, this package is assumed. For all classes that are not in the package, the complete class name is explicitly mentioned.

Class names are always capitalized on the first character. Subsequent uppercase characters denote characters of an abbreviation or word breaks.

Methods

Methods are always denoted by appending a pair of matching parenthesis behind its name, like setGrammar(). Generally, the method parameters and return type are not specified; the full specification can be found in the sources or the JavaDoc documentation. The absence of those should therefore not be interpreted as the method being void or not having any parameters.

Contrary to class names, method names always start with a lowercase character. Word boundaries within the method name are denoted by capitalized characters like with class names.

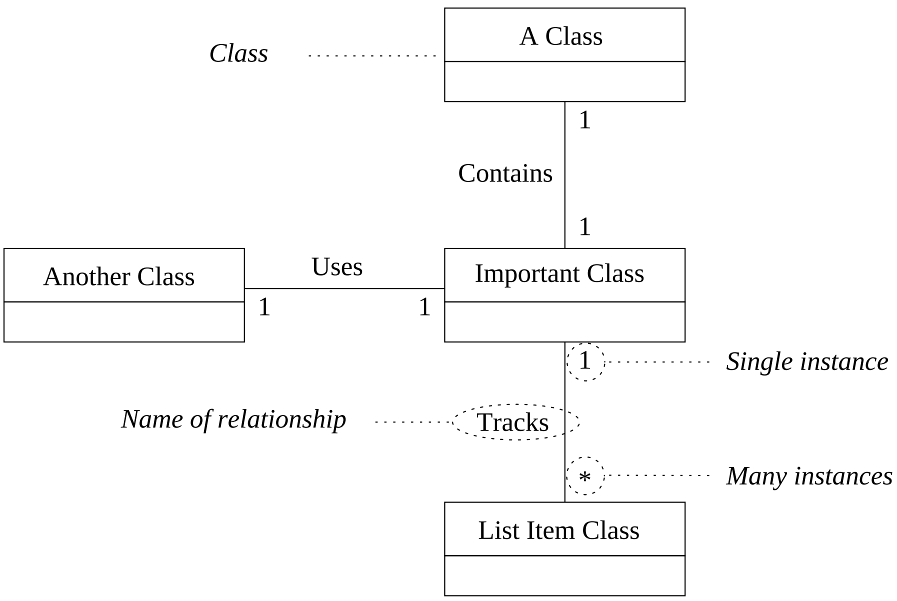

Figure 3.1: Brief explanation of UML relationship diagrams

Diagrams

Class relationship diagrams (figure 3.1) and sequence diagrams (figure 3.2) are based on UML (Universal Modelling Language). UML is a widely recognized industry standard for modelling object-oriented design aspects. UML provides a platform with a set of primitives with well-defined semantics. For more information on UML, see Booch, Rumbaugh and Jacobson [5] or visit Rational's UML website at `http://www.rational.com/uml`.

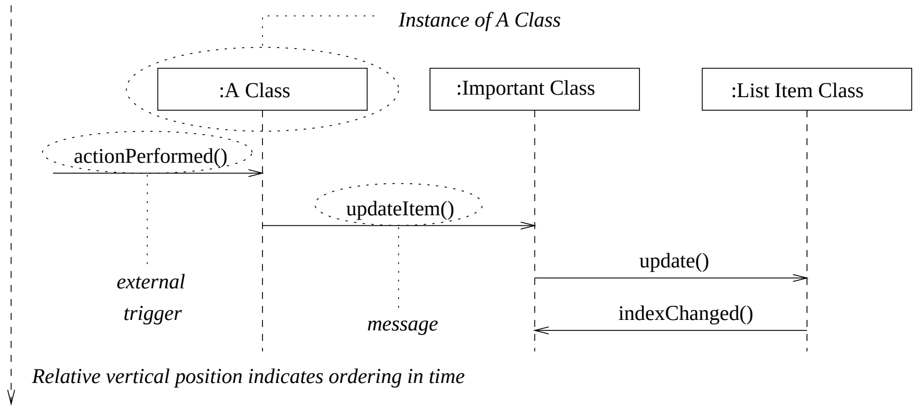

Figure 3.2: Brief explanation of UML sequence diagrams

Usually, diagrams are slightly simplified towards a specific point of view for clarity. As a result, diagrams in different parts of the text may show a different picture, since elements not shown are considered too much detail for the context. The complete class relations diagram would be quite large and complex.

3.2 Library design

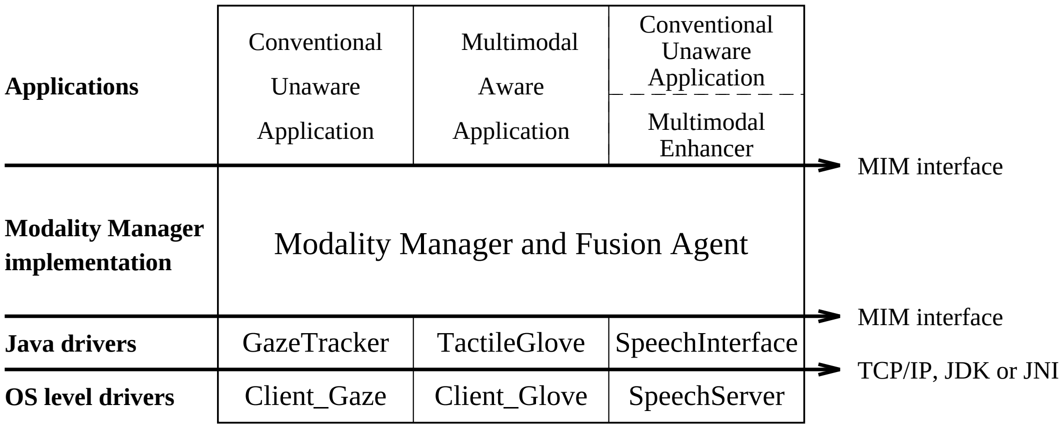

Figure 3.3: Overview of the layers

Figure 3.3 shows a basic overview of the different layers and interfaces between them. The main idea is that the manager is between the applications and the Java "drivers" for the modalities. This way, the manager can be a mediator between them and provide the necessary level of abstraction for the applications, while the Java-level drivers can be kept simple and re-used if someone decides to write a new manager.

3.2.1 Application interface design

The interfaces between the applications and the manager is such that an application can either be aware or unaware of the manager. In case of MM unaware applications, they can be 'enhanced' by using an enhancer class. This class is specific for each application, since it requires detailed knowledge of the innards of the application.

In case an application is unaware and there is no enhancer, it could still be used (although limited) by the new modalities by sending mouse events. This is up to the specific implementation of the MM. However, it severely limits the naturalness of the user interface; the manager is simply not able to retrieve enough information from the application, nor can it send the application information about the new events.

If the application is MM aware, it has a small but powerful set of interfaces available. This interface allows the application to get information about the available modalities and request events for specific modalities. The application, in turn, must provide objects that can receive those events.

3.2.2 Modality interfaces design

The modality interfaces abstract the manager from the problem of interfacing with the specific system-level driver. The interface is mainly concerned about getting information from or about the device.

How the driver communicates with the system-level driver, or even if it needs to do so, is not important to the manager. Possible methods for communication are existing APIs (for example for keyboard and mouse), the Java Native Interface, which can link to machine architecture and OS specific (native) code, or TCP/IP connections.

3.3 Application interface description

A basic outline of the application part of the modality management definition is given in figure 3.4. The figure shows the three types of components that make up an application: the `multimodal extension`, the `graphical component` and the `modality event listeners`.

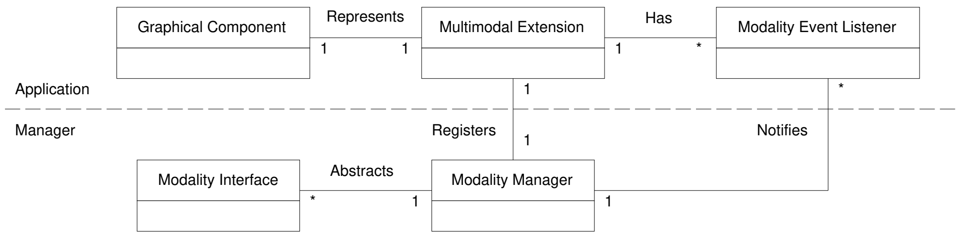

Figure 3.4: Relationship between application and manager

The multimodal extension is the main intermediary between the application and the manager. It implements the interface `MIApp`, providing the manager with a method to retrieve the toplevel graphical component (`getToplevelComponent()`). It also has methods to notify the application of the starting and stopping of management services (`notifyManagerStarted()` and `notifyManagerStopped`, respectively), and the addition of a new modality (`notifyMIAvailable()`).

The (toplevel) graphical component is the outermost container object, containing all other graphical components in that application. This is usually the `content pane` of the window containing it. The content pane is the object representing the entire area inside the window's borders. It has to be a Swing component, which means that it has to descend of `javax.swing.JComponent`.

This component is not a special type of object for the modality manager; it is always there if the application has a visible component. The multimodal extension merely makes it explicitly available to the manager.

The manager needs to know what the graphical component is to find out which application is focussed. This is used to route events to the focussed application.

For applications that were not written with a multimodal interface, there is support for `multimodal enhancers`. These enhancers contain the application specific knowledge to add the multimodal user interface in the form of Java code. They piggyback on the applications, acting as an agent between the modality manager and the application. This is only possible for well-written applications with extensibility in mind.

Contrary to applications, which are instantiated by the workspace, enhancers are instantiated by the manager as soon as it detects a new instance of an application that the enhancer supports. Besides the MIApp interface, Enhancers must also implement the `MIEnhancer` interface, which only defines one method. This method, `hookupWith()`, is called by the manager after it instantiates the enhancer to point it to its application.

The modality event handlers are the application's event handler classes. In Java terminology, these are called `listeners`. When the multimodal extension is notified of the availability of a modality, it can register a listener with the manager for that specific type of modality. The manager will then route incoming events to the appropriate handlers, when an application has the focus.

There are two types of event handlers, one for natural language events and one for pointer events. This is to reflect the different nature of both in a strongly typed way. Events are described in section 3.6.

3.4 Modality interface description

The basic relationships between modality interfaces and the manager is shown in figure 3.5.

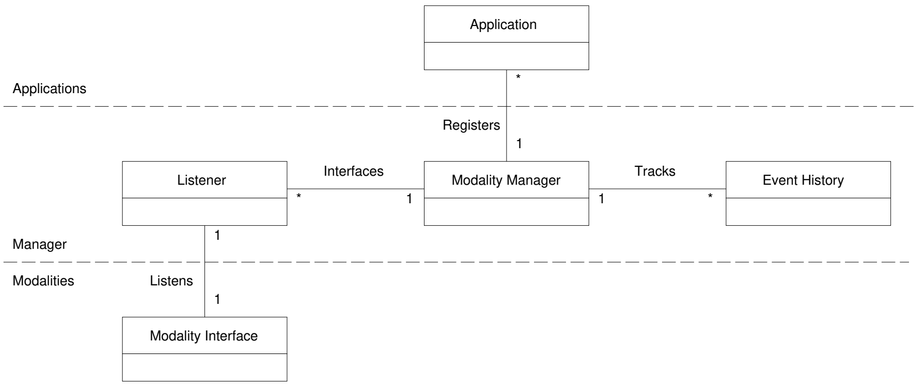

Figure 3.5: Relationship between modality interfaces and manager

A modality interface implements the `MI` (Java) interface. This interface has methods for retrieving name (`getModalInputName()`), version (`getModalInputVersion()`) and capabilities (`getCapabilities()`) of the modality. It also has methods for manipulating the modality, such as supplying customizers (`loadCustomizer()`) and switching between them (`selectCustomizer()`), and changing settings (`getControlPanel()`).

MI includes the recognized flags for capabilities; a device must be described with a combination of these flags. New devices that have capabilities that do not have an appropriate flag, cannot be used to the fullest. The modality interface cannot express these and the capability can therefore not be used. A workaround for this would be to define custom flags. This is possible but the manager and applications will have to be written towards these custom flags; the existing manager and applications cannot use this capability.

Current flags for input devices include flags for 2D and 3D pointer devices, directional information for pointing devices, finger flexing, and timestamped and score annotated natural language input (text).

If the device represented by the modality interface is some sort of pointing device, it should implement `MIPointer`, which inherits MI and provides specifics for pointers, such as keeping track of dragging motions (`get/setDraggingXXX`) and making the cursor visible or invisible (`setCursorVisible()`).

When the modality interface registers with the manager, it receives a listener for the modal input events. When events are sent, it is the manager's responsibility to store them in the history buffer, perform fusion when possible and send events to the applications. The modality interfaces are just the lowest layer in the hierarchy, like device drivers and do not have anything to do with multimodal fusion or disambiguation.

When a modality interface can also send feedback, it should set the appropriate capability flag and implement the complementing interface. Currently, there are only two forms of feedback:

1. Natural language feedback, indicated by the `NL_FEEDBACK` flag in the MI interface. The accompanying interface to implement is `NLFeedback`.
2. Force feedback, indicated by the `FORCE_FEEDBACK` flag. The accompanying interface to implement is `ForceFeedback`.

Java interfaces turn out to be very useful here; they provide an easy, intuitive, and type-safe method for having different implementations.

3.5 Modality manager description

The interface definition for the modality manager is named `MIManager`. It is the central part of the library, through which all other parts communicate. Its comprehensive but powerful set of methods are discussed below, grouped according to their use.

Informational methods

There are two methods that retrieve name and version information of the manager. These are `getMIManagerName()`, `getMIManagerVersion()`.

Methods for modality interfaces only

The methods for the modality interfaces are relatively simple: for registration, there are `registerNLIModalInput()` and `registerPointer2DModalInput()` which return the listener the manager assigned it. To unregister, there is unregisterModalInput(). All other communication between modality interfaces and manager go through the listener and modality interface object.

Methods for applications only

There are quite some more application methods. This is necessary to allow the application to instruct the manager on it's needs with respect to the modal interfaces.

To register an application, there is registerApp() and registerEnhancer(). The latter one will register the application's class to look out for and the accompanying enhancer to instantiate for it. The methods unregisterApp() and unregisterEnhancer() complement this for undoing the registration.

To get the events, the application must register listeners for the desired modalities. For this purpose, the methods addNLISentenceListener(), addNLIStringListener() and addRawMIPointer2DListener() are available. To stop the event delivery, removeNLISentenceListener(), removeStringListener() and removeRawPointer2DListener() are used.

For grammar switching, the method setGrammar() is available. This method must be called every time the application needs to update its grammar. For simple applications, this is only at startup, to set the initial grammar. It must be assumed that the grammar will be augumented with other constructs. Depending on the implementation of the manager, it may result in the application getting natural language input events that are not within its grammar.

To facilitate the addition of a multimodal interface to existing applications, the method setPointerMotionInput() can be used. This method allows the application to set a different pointing device as its mouse input, while receiving the events as if they were regular mouse events.

Fusion related methods

There are several methods to support fusing modality events. They mostly focus on finding related events or converting desired pointer events to mouse events, during or after semantic interpretation. These events do not solve any problems such as the actual fusion, redundancy, and ambiguity, but can be used in the process. This is left to the actual implementation of the manager and applications.

To trigger pointer-based actions, generateClickedEvent(), generatePressedEvent() and generateReleasedEvent() generate false mouse events for a specific pointing device at a certain moment. Since these actions are usually only triggered after speech recognition, they use the pointer's history buffer to find the position of the event closest to the desired moment.

To start and stop a dragging motion on any pointing device, startDragging() and stopDragging() are used. This will lock said device for use for the specified application only, generating mouse dragging events until it is released.

To find an event in the history buffers, findEvent() will find the event closest in time to the specified moment for the specified modality. This relieves applications of having to keep track of every event themselves.

Other required shared methods

To retrieve all registered applications and modalities, the methods `getMIApps()` and `getModalInputs()` can be used. These methods return a list of applications and modal interfaces, respectively. The list is a `java.util.Enumeration` object.

Feedback methods

The two supported types of feedback are natural language feedback (for displaying as plaintext or pronunciation through a text-to-speech system) and force feedback for a haptic interface. The corresponding methods are `sendNLFeedback()`, which accepts a `java.lang.String` with the text, and `forceFeedback()` which accepts a float in the range [0..1] to specify the amount of feedback.

These feedback methods are primarily for the applications. However, like fusion methods, the manager might use them internally since they provide a level of abstraction.

Absence of an MIListeners class

There are no specific classes to denote the low-level modality listeners that interface the modality interfaces. Instead, it uses the same listeners that are also used for applications (which are discussed below). This avoids duplication of functionality. There is no need for typesafety here; applications cannot get access to events before the manager, since the modalities ask the manager for a listener. Applications, on the other hand, offer the manager a listener. Modalities will never communicate with an application directly.

3.6 Events

Events are updates about the state of input devices. These can occur any moment during runtime, at a high rate (gaze tracker, glove) or at a slow rate (speech). Due to computational, device-to-system, or network lag, events are not necessarily delivered sequentially. Therefore, they are timestamped. However, this does not mean that the manager is obliged to order the events and deliver them sequentially. Again, this is left to the actual implementation.

Events are divided into two main categories: pointer events and natural language events. The distinction is there because they have very different characteristics.

However, both share a common characteristic: they both have timestamps, indicating when the event occurred. This is reflected in the common base-class, `MIEvent`. This class only defines a method for getting the timestamp, `getWhen()`. This is used in the history buffer class, which is independent of the type of event. Every modality should have its own history buffer object, but there is no need to make separate classes for every modality.

Pointer events

Events carrying pointer information are stacked in a hierarchy, each layer adding an additional level of information. Here are the events in order from the simplest to the most complex event:

RawPointer2DEvent contains a single integer 2D position.

SimplePointer3DEvent adds a floating-point 3D position. Applications receiving this event will also have the 3D position adjusted for their position in the desktop.

Directional3DEvent adds a floating-point 3D vector, denoting the direction.

HandMotionEvent adds finger flexing data in a floating point vector. It has five elements, one for each finger, normalized to a range of 0.0 (finger extended) to 1.0 (finger contracted).

To receive a pointer event, the receiver must provide a listener. All pointer events go through the same type of listener, defined by the `MIMPointer2DEventListener` interface. The listener is usually tailored for a specific type of pointer event, but can inspect the recieved type by using Java's `instanceof` operator or by introspecting the `java.lang.Class` object (using the reflection API).

Natural language events

There are two types of natural language events. The only difference is the level of detail provided about the external occurrance that triggered the event.

The simplest event, `NLIStringEvent`, only contains a string with the sentence that was sent by the user. This might be through the speech recognizer or the keyboard interface. Applications that simply match whole strings may prefer to use this type of event.

The more complex event, `NLISentenceEvent`, contains much more information. It is especially geared towards speech recognition output. First of all, it contains all possible alternatives that the speech recognizer found. Then, for each alternative, it contains the timestamps for the beginning and the end of the proposed sentence, timestamps for individual words (if provided by the recognizer), and a confidence number.

This event is to be used internally in the manager, since it contains most information. Applications may also use it if they need to analyze possible alternatives or timestamps for specific words. Modality interfaces will always have to supply this complete version; missing information will have to be filled out.

To obtain the NLIStringEvent for applications that use the simpler event, the manager has to decide which alternative is the correct one, then construct a new NLIStringEvent itself.

3.7 Initialization

When an application starts, it needs to establish contact with the manager. There are at least three methods to achieve this:

- using a special constructor for every application. This would require modification of the DISCIPLE desktop to call that constructor instead of the default parameterless constructor (this constructor is required to make a class a bean).
- using a static variable. This is easy to program, since it can be initialized to null when the class containing it is loaded, and when a manager is instantiated, it can assign itself to it.
- using a network connection. This is how the os-level drivers communicate with the Java modality interfaces, so a minimalistic design might use this. However, since our applications always run on the same computer, and even in the same Java virtual machine, this is very inefficient.

Since it is was the idea to keep the multimodal management separate from the desktop, as an add-on module, the first option is not acceptable. The third option is more complex than necessary and too inefficient, since the traffic between the manager and the applications is expected to be quite intense and latency adds up quickly. The second option seems to be the most effective and efficient.

However, when the application is instantiated, the manager may not be instantiated yet. Therefore, a simple solution with a single static variable is not sufficient because it would require too much code in every application to keep track of the manager's appearance into the desktop.

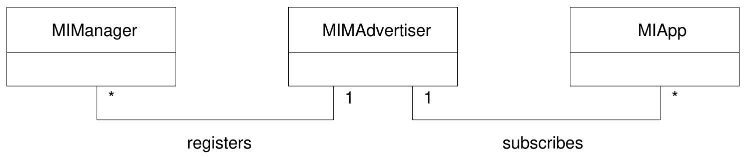

Figure 3.6: Relationship between manager, advertiser and application

To solve this problem, the `MIMAdvertiser` was introduced. This class, containing all static variables and methods, is part of the library and thereby loaded at startup. It acts as an advertisement agency for modality management services, keeping track of the appearance of the manager and applications. A class relationship diagram is shown in figure 3.6.

The advertiser allows any sequence of sign-ons and sign-offs by both manager and applications, requiring only a single registration and de-registration call. After registration, applications will be informed of the presence of a manager when it is available. At de-registration of either the manager or the application, the application will be notified with that management services have been cancelled. Figure 3.7 illustrates this. It shows the method calling sequences for two possible sign-on orders and the two sign-off orders.

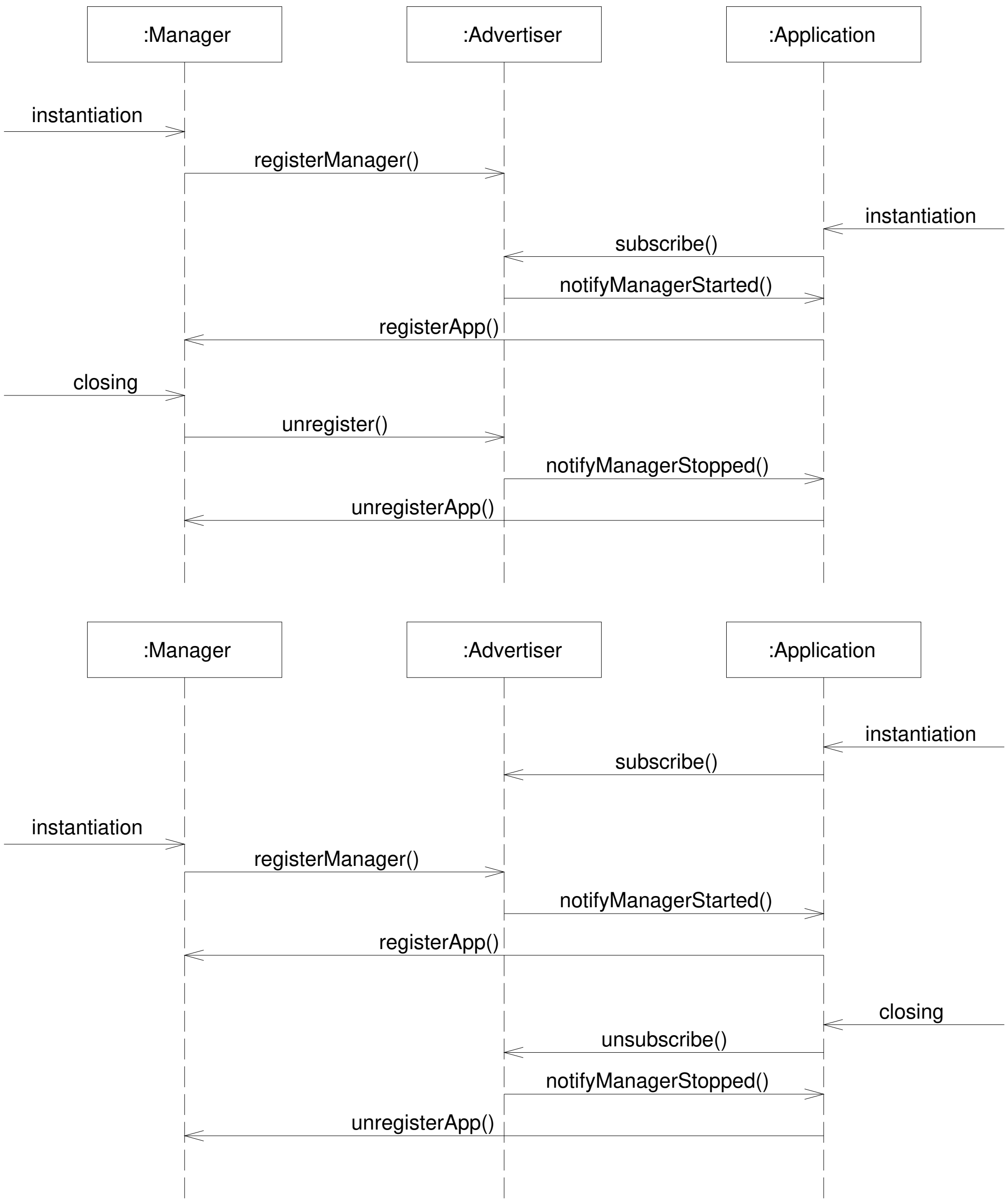

Figure 3.7: Examples sequencences of method calls between application and manager

Chapter 4

Implementation of the Modality Manager

This chapter discusses the sample implementation of the modality manager and the modality interfaces. First, the modality interfaces is discussed, which are independent from the actual manager and have a fairly simple design. Next, the design of the sample implementation of the modality manager and the features implemented in it are described. Finally, details of the implementation are discussed.

For notations and conventions used, see section 3.1. Our experiences and a preliminary evaluation are discussed in chapter 5.

4.1 Modality interfaces

As mentioned before, the different modality interface implementations are independent from the manager implementation. They only rely on the modality management library as described in chapter 3. This is very useful for changing the implementation of a modality interface or the manager implementation, since they do not rely on each other's code other than through the library.

Our current implementations of alternative modality interfaces (that is, all interfaces but the keyboard interface) have one thing in common: they communicate with the OS-level drivers through a TCP/IP connection. This makes the whole setup more flexible. In our case, it was also necessary to use different platforms at once since the gaze tracking and glove software run on UNIX machines, while the speech recognizer only runs on Microsoft Windows.

The disadvantage is that these network connections introduce some delay (*lag*). This is only an issue for modalities whose performance is sensitive to lag. For example, the gaze tracker is really lag sensitive (delay in the order of a twentieth of a second is already disturbing), while the speech recognizer output is not that sensitive (a delay of a quarter of a second is not disturbing). With the speech recognizer, the network delay is also not the only delay; the signal processing performed on the microphone signal also adds considerable delay, but a modern

PC such as we used is fast enough to perform this fast enough to be convenient.

Another issue with network connections is potential buffering problems if the input device generates more output than the receiving end can handle. Since Java performance is still lacking, we experienced this problem for the gaze tracker and the tactile glove (they both produce a sample at a rate in the order of 60 per second). The problem was resolved by adaptive undersampling of cursor display. This means that the modality interfaces always use the last sample, ignoring other samples produced between the last cursor update and the next one. If the machine would be fast enough to handle all the input, none of the samples would be dropped.

All our OS-level drivers are servers, accepting a connection from only one client at a time. Initially, the drivers were clients, and the multimodal demo was the server. This was cumbersome because you had to restart the driver every time the application was restarted (and only *after* the application had started). Turning the drivers into servers made it much easier to restart the application, which is especially useful for debugging. Also, it allowed run-time (re-)connection to drivers from within the application.

4.1.1 Gaze tracker

The gaze tracker interface is the simplest interface in our setup. It receives 2D coordinates $([0..512], [0..512])$ from the gaze tracker's driver over a TCP connection, and remaps them to the screen coordinate system (which can vary in size). It then redraws the gaze cursor, if necessary, and sends the updated coordinate to the manager.

The gaze tracker interface also provides a simple configuration dialog, which can be evoked from within the manager. This dialog contains simple options, such as the driver host and port, a button to connect, and checkboxes to disable the gaze tracker or hide the cursor.

The data format used over the TCP connection is simply least-significant-bit-first signed 32 bit integers (two for each update). Timestamps for the events are set to the current time with the Java method java.lang.System.currentTimeMillis().

4.1.2 Tactile glove

The tactile glove interface is a bit more complicated than the gaze tracker interface. Each sample from the glove contains more data; instead of a 2D coordinate the glove readings contain a 3D coordinate, directional information (azimuth, elevation and roll), and finger flexing data for the first four fingers (the pinky does not have a piston and therefore no readout).

The glove interface also deals with two simple conversions: conversion between 2D and 3D coordinates and between the four-finger readout and the five-finger event format. Mapping from 3D to 2D is done by simply ignoring the depth coordinate and scaling and translating from the glove XY plane and range to the screen plane and range. When the glove event is delivered to an application, both its 2D and 3D coordinates are translated with respect to the application's window origin. This is performed by another function that adjusts the 3D coordinates as a function of the old 2D mapping and the new 2D origin.

Conversion between four- and five-finger finger flexing data is performed by padding; the sample for the pinky is copied from the ring finger. This produces natural results for most situations.

4.1.3 Keyboard interface

The keyboard interface is a simple modality interface bean that uses the keyboard for natural language input (sentences) and the screen for feedback. It is *not* a replacement for event-based keystroke reception, since that is already supplied by AWT.

Any keyboard input that is not caught by an application is sent to the keyboard interface's text input box, until the enter key is pressed. When the enter key is pressed, the text is sent as a natural language event.

Since the event is supposed to have confidence score and individual word timestamp information, all scores are set to 100 (maximum) and all word timestamps are set to the moment when the sentence was sent. This might not be completely correct in all cases, especially when it comes to timestamps for fusion with other modalities, but the keyboard is so different from speech that this extra information would have to be treated quite differently. Besides, the current implementation of the fusion agent does not use individual word timestamps since the speech recognizer does not provide them.

Feedback text is displayed on the screen like closed captioning. If the text is sent out too fast, the next messages are stored in a queue and shown after a reasonable delay. This guarantees that all text can be read by the user.

4.1.4 Speech interface

The speech interface is the interface to the speech server we wrote for the Microsoft Whisper system (or any other speech recognizer and text-to-speech system that conforms Microsoft SpeechSDK 4.0). All data is sent over a TCP connection, like the glove and the gaze tracker. Again, this is done to achieve platform independence for the Java framework. Fortunately, the delay introduced by the network is hardly noticable here unless it is in the order of seconds. Communication is bidirectional and entirely in ASCII.

Other tasks of the speech interface are:

Maintain grammar list The interface maintains a copy of the list of grammars and the active grammar in case of (re-) initialization of a speech server connection. It ensures that the speech server has all application grammars.

Relay recognized sentences Sentences recognized from the audio input are sent to the speech interface. These messages contain the sentence, start and end timestamps for the whole sentence and each word, and confidence scores for the whole sentence and each word. Since all this information is transmitted in ASCII, the received message is parsed and a corresponding event is constructed. This event is passed to the manager.

Send feedback Feedback is sent to the speech server to be spoken by the text-to-speech system.

4.2 Internal design of the manager

This section gives an overview of the internal design of the modality management and fusion agent, called the *Octopus*. The name is a reference to an octopus' many arms symbolizing the modalities.

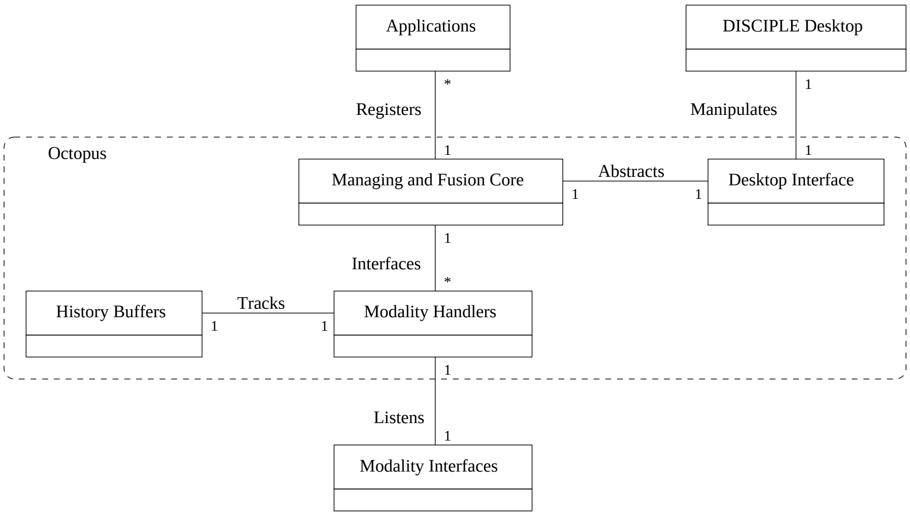

Figure 4.1: Components and relations in the Octopus

Figure 4.1 shows an overview of different components and relations in the Octopus. This picture is not very different from the description of the manager structures from the library (figures 3.4 and 3.5) since the Octopus was built on that library.

The several applications that make use of the manager are still there, the history buffers are still there, and they have the same position and function. The listeners are wrapped in the *handlers*, which also contain most of the modality-specific code within the Octopus.

New elements are the DISCIPLE desktop and desktop interface. Since the Octopus interacts with the desktop, this is an important element. The desktop cannot be treated like just another application, because the Octopus queries it (for available beans and focussed bean instance) and manipulates it (initiates dropping a new bean instance in the workspace, changes focussed bean). The desktop interface handles these tasks, abstracting the details for the manager.

4.3 Implemented behavior, grammar combination and other features

This section discusses the interactive behavior, mechanisms for combining the application and manager grammar, and other features.

4.3.1 Multimodal interactive behavior

We implemented several examples of multimodal interactive behavior, to prove the flexibility of the modality management library and to attempt to make a userfriendly multimodal environment.

Event delivery

To decide where incoming events go to, the Octopus uses two approaches for the two different event types, pointer events and natural language events. This approach is also taken by several X window managers.

Pointer events (both 2D and 3D) always have a point on the screen where they happen and can be related to a certain object on the screen. This is fundamentally different from natural language events, which do not have screen coordinates and can therefore not be related to an object on the screen.

Since it is clear for pointer events that they are to be delivered to the object corresponding to the screen coordinates of the pointer's cursor, they are delivered there. The only exception is when the user started a dragging motion in another application. Blindly delivering the new dragging events to the application that did not start the dragging motion is very confusing, especially if the user accidentally overshoots the target and ends outside the application's window. This approach is taken by several X window managers, such as fvwm2 [18].

This approach is not the only one, however; the default window manager for the Microsoft Windows line of operating systems delivers some pointer events only to the focussed application. Only mouse motion events are always sent to the window below the cursor, mainly to change cursor appearance. Mouse clicks outside the focussed window are only used to switch focus or to manipulate the desktop.

It is hard to say which approach is better; it is therefore a matter of preference. Window managers might even let the user decide and change it in runtime.

Natural language events are always delivered to the focussed window. How the focus changes depends largely on the Desktop's internal window manager (i.e. Swing and the Desktop itself). The next section describes additional, multimodal methods for switching between windows, provided by the Octopus.

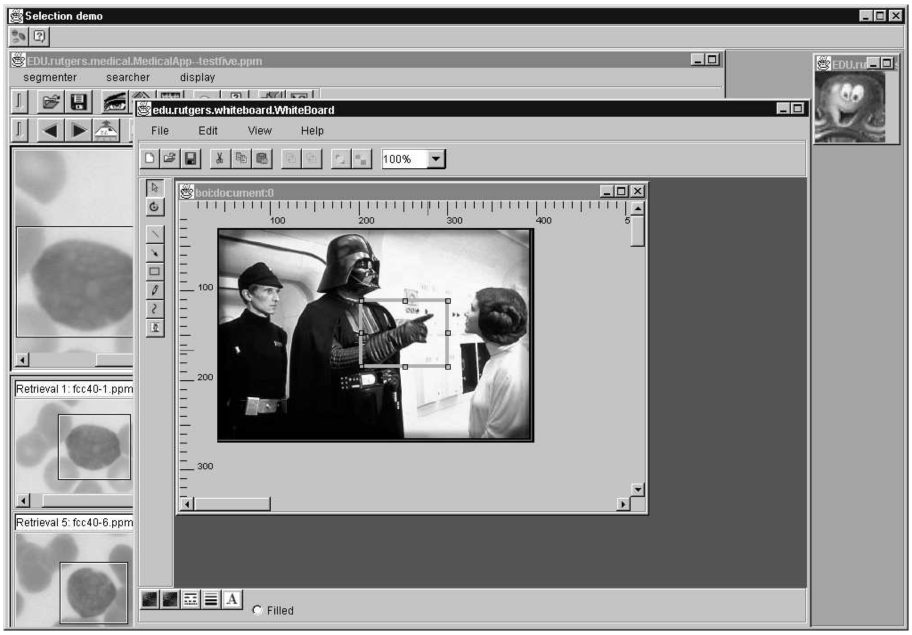

Figure 4.2: Example of window focus; the topmost window has the focus

Switching applications

Switching the focussed application can be done in several ways, involving any of the modalities. Normally, the user has to change focus by clicking the desired application's window. To achieve a more natural approach and exploit our multimodal capabilities, we implemented several other methods for switching focus.

voice command A voice command of the form "switch to <beanname>" switches focus to the named bean. Alternative forms are used for ease-of-use ("select" instead of "switch", and "to" is optional).

voice and gaze Looking at a window and saying "switch" or "select" also changes focus.

gaze Staring at a window for a set number of seconds will cause the focus to switch to that window.

glove Using mouse emulation (described below), the glove can be used to change focus by clicking (contracting and extending the thumb) on the window.

Are you talking to me?

One of the main problems with the use of speech recognition, which is usually overlooked, is to figure out when the user is talking to the computer and when the user is talking to someone else (or possibly *another* computer). Choosing the right microphone can alleviate this problem, filtering out other people's speech, but it will usually not filter out other "background" speech and certainly not the user's dialogs with other people.

In our case, we used a microphone array, which attenuates soundsources that are not in the focus volume. When the microphone array is placed on top of the monitor, this volume is around the head of the user's normal position. Attenuation of the out-of-focus soundsources is up to 20dB for the specific model we used, which is good enough to reduce a lot of the background noise and suppress recognition of most out-of-focus dialogs.

Of course, this still does not solve the problem of dialogs between the user and other people. The problem is not just a signal processing one; it is a problem of understanding the dialogs (across all modalities people have), or specifically, *who* is addressed *when*.

Since good natural language understanding is not available we cannot build a good semantic filter. This is a large research topic by itself. Therefore, we will have to settle for solutions that use cues from user input or explicit methods for telling the computer that it should listen or not.

We implemented both methods. We implemented a set of speech commands to ignore all other commands and gaze confirmation that uses cues from the gaze tracker. The user can also use the speech interface configuration dialog box to turn speech off (by mouse), but this is somewhat cumbersome.

The speech commands are simply "disable speech" and "computer enable speech" (and several aliases). They are understood by the Octopus and change flags that affect the way incoming speech is routed. When speech is disabled, the speech recognizer is still active, but only with a minimal grammar. All commands are ignored unless they are prefixed with escape word "computer" and even then they will not be sent to applications. This way, speech can be turned on again. It turns out that the speech recognizer almost never picks out "computer enable speech" or any of its aliases, but as a very fortunate coincidence, it seems to pick "play deaf" out of random conversations, which also disables speech.

Gaze confirmation uses the gaze tracker to find out when the user is looking away from the screen. When the user has not looked at the screen during the last few seconds, any speech input is ignored unless it is prefixed with "computer", as with the speech blocking commands. This behavior can be turned on and off with the commands "gaze confirmation on" and "gaze confirmation off". If gaze confirmation is active but the gaze tracker does not work properly, the user can turn it off by escaping the command with "computer".

The same problem, unintentional recognitions, can happen with other modalities. If a user stares longer than the switching treshold at a window, it does not mean the user wants to switch to that application. Likewise, curling up the hand with the glove on to let it rest, or pointing at something might be interpreted as a gesture for an application, if it is performed in the working volume of the glove.

However, we found that these problems does not occur very often. This is mostly because we set the stare-to-switch treshold relatively high (five seconds, which is easy to change) and most gestures that are not ment for the computer are usually performed outside the working volume. Also, the number of gestures is very small (point, click, grab) which will usually not result in recognized gestures for random finger motion (hand motion is not used for recognition in our gesture recognizers).

Mouse emulation

Mouse emulation is a technique where a modality or combination of modalities is used to simulate mouse events. This is useful since our applications were not originally written for the multimodal management interface. An application can still be used with the alternative modalities by predefined actions that send mouse events and it makes it easier to add the multimodal extensions since the extension can trigger false mouse events.

The glove can be used to send mouse events in three different ways. First, a simple mouseclick can be sent by "clicking" an imaginary button with the thumb. Second, normal mouse movement events can be sent by pointing with the indexfinger. And last, mouse dragging events (moving the mouse while holding a button) can be sent by making a grabbing gesture. All mouse event coordinates are set according to the glove cursor.

A combination of the gaze tracker and voice commands can also be used to generate mouse events. Looking at a point and saying "click" generates a click at that point, and a mouse dragging can be simulated with the voice commands "start drag" and "stop drag".

Finally, multimodal extensions can have the manager send false events by calling the `generateXXXEvent(), startDragging() and stopDragging()` methods (as described in section 3.5). However, this is not a specific feature of this manager implementation since the MIManager interface specification requires these methods.

4.3.2 Grammar for controlling the manager

Available voice commands

The available voice commands are listed in table 4.1.

It should be noted that some of the voice commands are not voice-only commands, but can be multimodal.

Grammar generation

Since the grammar definition does not only contain the syntactic structure but also the vocabulary, it has to include every single word that should be recognized by the speech recognizer. Some commands can have names of beans as arguments, so these names must also be included in the grammar. However,

Table 4.1: Voice commands for the Octopus

Voice command	Action description
hello computer	Are-you-there command, responds with “hello user”.
computer listen computer pay attention computer enable speech	Asks the manager to listen again when speech was turned off. If it was still turned on, it will confirm that is is listening (“yes?”).
switch to <bean> select <bean> switch [bean] <gaze> select [bean] <gaze>	Changes the focussed application, possibly by gaze.

at compile time, these names are not known, since the Jarfiles containing the beans can be loaded at runtime. Hence, the grammar has to be constructed dynamically at runtime.

A large portion of the grammar, such as the syntactic structure and the command verbs, are known at compile time. This is because the parser is specialized towards this set of commands, and cannot accept just anything. The only part that is actually dynamic, is the list of names of the beans. Therefore, the grammar can easily be divided into a static part (known at runtime) and a dynamic part (completed at runtime).

The static part is stored in the program as static strings (compile-time defined constants). When the grammar is needed, a set of rules for the list of bean names is generated on the fly by inspecting the bean browser and added to the static part. For an example of what this looks like, after merging the grammar with the application's grammar, see figure 4.4 (page 38).

Grammar merging

The manager has its own grammar for its voice commands, and applications have their own grammar to support their own voice commands. However, when the grammar is sent to the speech server, it should recognize both, since the user must be able to use both the application and the Octopus voice commands.

This is accomplished by merging the grammars. Merging is performed by constructing a grammar that is the union of both grammars, accepting anything that is in either or both grammars (the total area of both circles in figure 4.3). The grammar definition is in the Microsoft SpeechSDK format, which is a context-free grammar definition language and requires a specific starting rule.

The rules in the definition consist of a non-terminal symbol followed by an expansion sequence of terminal or non-terminal symbols. If there are multiple definitions for a non-terminal symbol, all of them apply. Since the definition language is such that it explicitly describes what is *included* in the language

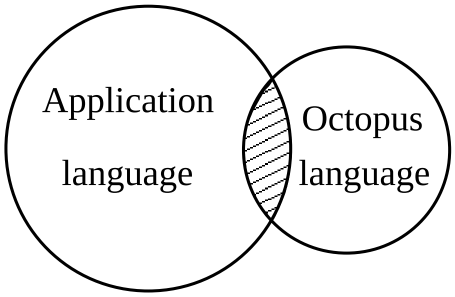

Figure 4.3: Venn diagram of grammar merging in language space

and not what it *excluded* in the language, rules can never conflict.

With this definition language, merging is quite easy. Adding the manager's grammar is simply appending the new rules at the appropriate point. The grammar file can have multiple sections, so it has to be parsed to insert the rules into the appropriate sections. An example of what this looks like is shown in figure 4.4 (page 38).

Ambiguity handling

Ambiguity can arise on two different levels: on the syntactic level (not able to decide which rule applies) or on the semantic level (choosing between different objects or meanings for the same phrase). It can have two origins: the application or the manager supplied an ambiguous grammar, or the Octopus added rules to the grammar that collide with the application's grammer (this corresponds to the marked area in figure 4.3).

In case of the application or the manager supplying an ambiguous grammar, the resolution is easy (from the managing point of view): since the recipient supplied the grammar, and has to parse the sentence, it is also the recipient's responsibility to resolve the ambiguity.

For the manager, the grammar was constructed such that, normally, there is no ambiguity. Only when someone loads new applications into the environment, their names may cause collisions. Examples of this would be a bean named "bean" ("select bean" could mean the bean "bean" or select a bean by gaze). Our implementation does not have explicit methods for ambiguity resolving, but does behave in a deterministic manner (there is an order in which constructs are recognized).

Some people also regard missing information as an ambiguous situation. In this implementation, it is an error condition. When information is missing (for example, the user says "select bean" and does not look at a window), the Octopus replies with an appropriate error message. In case of unsuccessful actions with the glove (e.g. grabbing empty space), they are just ignored.

When the ambiguity was caused by the manager merging its own grammar with the application's, the problem is resolved by the order in which the sentence is presented to the respective parsers: unless the sentence starts with an escape word ("computer" or "octopus"), the sentence is first offered to the application. The application then indicates whether or not it understood the sentence, and if not, the manager tries to parse it. If the sentence started with an escape word, the Octopus always parses it.

This trial-and-error parsing technique is similar to that found in some other environments with dynamically changing grammars, such as multi-user virtual worlds. These environments tend to be highly dynamic with respect to the verbs, nouns and grammatical constructs. They do not explicitly build a grammar (input is through a keyboard), but their parsing is dynamic, giving each object in the user's environment a try at parsing the user's input [1][24][23].

The only way this can cause trouble is if the application's parser is not robust enough to handle sentences that are not in the grammar (which is not wise since there is also a keyboard interface, where the user can type *anything*), or the application wants to have sentences beginning with one of the escape words. The first problem is a matter of good programming, the second problem will almost never be a problem and can usually be avoided by constructing a different grammar. Technically speaking, not being able to use one of the escape words in the beginning of the sentence is a limitation, but we need a fail-safe method for communicating with the manager at all times.

4.3.3 Additional features

The Octopus is visible in the workspace as a picture of an octopus. The picture is set in a window by the desktop, like other applications. Clicking in this picture will bring out a popup menu with a selection of dialogs, one for the Octopus itself, and one for each modal interface that is loaded.

The Octopus' dialog is just an "about" dialog, but it can be used in the future to add configuration options. The modal interface's dialogs are configuration dialogs. The window and the close button are provided by the Octopus, but the actual configuration panel is provided by the interface itself.

```
// Simple example grammar to set the background color.

// This part is mandatory and describes the type of
// grammar and (natural) language it is based on.
// This is US English and context-free grammar type.
[Grammar]
LangID=1033
Type=CFG
CompileTo=CFG

// Initial symbol is always <Start>
[<Start>]
<SetBG>= set background [to] <color>

// Additional runtime rules:
<Start>= [opt] <octoEscape> <octoCommon>
<octoCommon>= select <octoBeanName>
<octoCommon>= select [opt] bean
<octoCommon>= insert [opt] new <octoBeanName> bean

[color]
=red
=green
=blue

[octoEscape]
// Additional runtime rules:
=computer
=octopus

[octoBeanName]
// Additional runtime rules:
=GazeTracker
=SpeechInterface
=Octopus
=TactileGlove
=SetBackground

[Lists]
=color
// Additional runtime rules:
=octoBeanName
=octoEscape
```

Figure 4.4: Example of a complete generated and merged grammar. *(Normal font is the original application grammar, bold is added by the manager and italics is runtime generated and added by the manager. Note that this example is simplified for clarity.)*

Chapter 5

Tests and Human Factors

This chapter discusses initial test results and human factors issues with our system.

Since the design of multi-modal applications is not a well-defined discipline, it will be an iterative process. In order to develop more well-founded design processes, it is important to know what their human performance characteristics are. Therefore, part of the Stimulate project focuses on the performance characteristics of these new devices.

Because the design is an iterative process, the implementation should be evaluated to find room for improvement. This is when the information gathered by the human factors studies can be applied.

5.1 Input device performance characteristics

This section will briefly discuss the goals, the setup and the results of the experiments conducted at CAIP for estimating input device performance characteristics. For more in-depth information, see the CAIP's annual STIMULATE report [13].

The goal was to have performance models for these devices. For speech, there have been previous studies on its performance models. Therefore, a Fitts Law study was conducted on the gaze tracker and the tactile glove since these devices have not been studied extensively. Fitts Law establishes a relation between computer user movement time and the difficulty of the task, and is widely used in the field of human factors studies.

5.1.1 Performance data studies

To gather the performance data, several subjects had to perform simple tasks. The tasks were simply holding the cursor in a start circle until a target circle appeared in one of four orientations (up, down, left or right), then moving the cursor to the target circle. The target circle could have three different sizes and three distances from the start circle.

The data was compared against performance data from a study on a mouse, by Mackenzie, 1995 [25]). Since the devices we measured are protoypes, we should not expect them to perform optimally. However, we did not expect them to perform significantly worse. The results of the experiments for tasks of various difficulty were then used to fit a model based on Fitts Law.

The outcome for one of the experiments is described in table 5.1. Measured data that determines parameters in Fitts Law are cognitive starttime and bandwidth. Lower starttime corresponds to a lower delay between the start signal and the movement, and higher bandwidth indicates faster movement.

Table 5.1: Measurements for cognitive starttime and bandwidth

Device	**Starttime** (ms)	**Bandwidth** (bits/s)
Mouse	400..500	5..10
Gaze	250..350	25..60
Glove	420..500	2.8..4.7

The results show that the glove has a slower in performance than todays mouse. Gaze is significantly faster than the mouse, but the data did not seem to fit Fitts Law very well. Detailed studies about this phenomenon have not taken place yet. Speculations about the cause include the different nature of eye movement (much shorter distances than hand or arm movements, saccadic versus smooth movements) and a very flat learning curve, which resulted in change of a subject's performance during the trials.

Another study was performed on the effect of multimodal feedback on performance. In these experiments, the user had to perform a task, but was interrupted during the task. We analysed the recovery performance (how fast the user went back to complete the task after dealing with the interruption) with or without multimodal feedback.

Specifically, we used force feedback with the tactile glove. The task was simply moving an object from one point to another, and the interruption was a visual distraction, which had to be dealt with by pressing a key. Only after pressing the key, the task could be completed.

Unfortunately, there was no measurable effect in recovery performance. This does not mean that tactile feedback should be abandoned; it can still be used to convey information to the user such as shapes, and it feels reassuring.

5.1.2 Implications for designing the user interface

While there is more to take into account than sheer task performance models, these models indicate that there is no reason to ignore these new input devices. Other criteria include effectiveness, learning curve, fatigue and the user's abilities.

The effectiveness is very dependent on the task. For example, the glove can be about equally effective for most applications, compared to a mouse, but for 3D manipulations the extra dimension in the glove's input readings is really useful. Speech can be very effective to manipulate existing icons on a terrain map, but almost useless when creating new icons in places that have not been marked.

Another factor that determines the effectiveness of a modality is the environment in which it is used. For example, a keyboard is not very useful in conjunction with a wearable computer. However, it may be the only option when the wearable is used in a noisy environment such as a processing plant.

While some devices are just more "natural" to use than others, the learning curve can be very different for various tasks. It is also heavily dependent on the number of semantically different input sequences. This can be as simple as the usual one, two or three button mouse discussions, but it can also apply to the number of gestures that can be performed with the glove, or the number of commands that can be spoken to the speech recognition system.

No matter how well-designed an interface may be, the more options it has, the more users have to learn and remember. However, it is good to know that the learning curve can partially be chosen by the number of semantic constructs for your input devices.

Devices also differ in how much their use wears the user down. Moving your eyeballs or your hand is much less straining than your whole arm. However, you may be less likely to develop CTS with a glove than with a mouse. Unnatural fixation on objects on the screen (staring) can be very stressful too.

Finally, the user's abilities should be taken into account. One of the most important advantages of multimodal interfaces is the promise of enabling disabled people to use computers, which are more and more becoming *the* tool of the information age. This technology can significantly improve their position in society. From blind people to deaf people to people suffering from RSI (which is becoming a major problem), alternative human-computer devices require different methods of presenting and entering data.

5.2 Initial tests

To test our implementation, we modified some of the applications that were developed for our collaborative environment. These include a whiteboard application, a 2D mission planning application (similar to the previous multimodal system), a 3D mission planning application, and a leukemia diagnosis support system.

Implementing multimodal extensions in these applications was a good test of the quality of the modality management library. Missing functionality and awkward design artifacts (which seemed reasonable at the time) showed up and were fixed.

The multimodal manager and the new multimodal applications were a good opportunity to experiment with different behaviors, find out points of attention when designing applications and improving behavior.

Table 5.2: Rules for creating grammar rules based on available toolbars

Toolbar type	Example tools	Commands added
function	new, open	<function> document
figure	line, rectangle	create <figure>
unit	truck, camp	create <unit>

The next sections describes the multimodal applications in more detail. The description is focussed at the multimodal fusion and behavior, and not as much on the actual code behind it.

5.2.1 The whiteboard and the 2D mission planning applications

The Whiteboard is a simple yet effective drawing application. It features multiple document support, through-and-through collaboration support, several types of drawing objects (including lines, text, and images), and basic operations on objects (such as cut, paste, and resize). The whiteboard is shown in figure 5.1, together with the 3D application. The mission planning version of the whiteboard is the window called `edu.rutgers.military.AreaMap`.

It is built on top of a multiple document library, which supports the generic notion of multiple documents, tools, and properties and the application framework for placing toolbars and creating menus. The 2D mission planning application is in turn derived from the whiteboard, since they share many primitives.

The mission planning system, called the `AreaMap`, adds resource icons ("units") to the drawing board. These resources can have their own properties, such as availability of food, ammunition and fuel.

Both applications have no built-in multimodal support, but their design is very "open", making it possible to manipulate the applications from within other applications. We exploited this feature to write a separate module called the *Whiteboard enhancer*, which adds multimodal capabilities to both.

While both applications are different, the [Whiteboard] enhancer can handle both. Since the AreaMap inherits most of its functionality from the Whiteboard, this is an easy task. The additional functionality of the AreaMap, the resource icons, are the only tools that have to be added. This is implemented by a simple knowledge-based system: the enhancer inspects all available toolbars in the application, adding voice commands for the tools depending on the toolbar type (for an overview, see table 5.2). The enhancer builds its grammar dynamically. The available tools from the toolbar are also added at startup.

Not all of the grammar is built dynamically. Part of the grammar is static. This part is loaded from file, and is used for the preamble and includes commands for identification of objects ("identify") and help ("help" for general help, "help <tools>" for a list of tools in a toolbar). The static grammar can reference to some of the dynamically generated constructs of the grammar. The only restriction here is that the referenced construct must *always* be generated. This basically means that specific toolbars cannot be referenced, but the entire collection of toolbars can be used.

Multimodal fusion is mostly handled by the manager, using its default fusion behavior. This includes object dragging, rotating and creation by glove or the combination of speech and gaze. Specific multimodal fusion includes use of the gaze tracker for identification (you have to look at an object to identify it) and for creation of resource icons immediately on voice command.

Creation of resource icons without the gaze tracker is done by selecting the appropriate unit (either by selecting it in the toolbar or by using voice commands). Then, the user clicks at the position in the canvas where the unit should be placed (with mouse or glove).

If the gaze tracker is available, this sequence can be shortened by looking at the position and using the voice command ("create <unit>"). This is solved by simulating a mouseclick at the position where the user was looking during the utterance of the voice command. This may sound elaborate, but it is really easy to implement with the multimodal manager and the application does not know the difference.

We used a simple, very task specific parser using a bottom-up approach to parse the voice commands. Due to our small grammar, we saw no reason to use a more flexible but more complicated approach. When we would work towards a more complex grammar, this parser should be replaced but it would be easy to convert the current constructs. Note that the parser does not know the difference between the two types of applications; there is no need to. The grammatical constructs for the AreaMap will simply never be parsed for a Whiteboard because the dynamic grammar generation does not generate these constructs in this case.

5.2.2 The 3D mission planning application

To complement the 2D mission planning application, and to experiment with the use of the tactile glove as a dual-mode 2D/3D pointing device, we implemented a 3D mission planning application (shown in figure 5.1). This application features reading of standardized Department of Defence terrain elevation data (DTED) maps, generating accompanying colormaps, placement of multiple resource icons (VRML objects), manipulation of icons by glove, tactile feedback, and voice commands for viewpoint movement. The multimodal user interface is built-in.

The modalities used in this application are the glove for moving objects, and speech and keyboard for viewpoint movement. Currently, there is no support for the gazetracker or the mouse. The 3D map application is not as well-developed as the AreaMap, since it is a first prototype.

The input from the glove is used to modify and render a model of a hand in real-time. This includes finger flexing, and the visual feedback is very reassuring for the user. Its readings are always taken with respect to the viewpoint, not

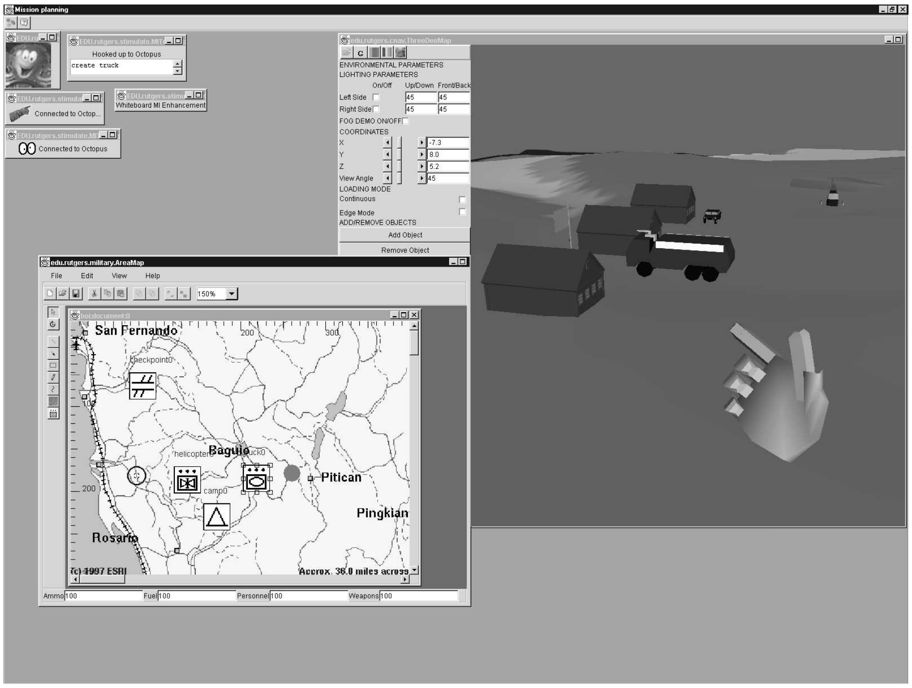

Figure 5.1: Snapshot of a multimodal workspace with the 2D and 3D mission planning applications

to with respect to the terrain; this would be confusing.

The application performs the gesture recognition; there is support in the manager to describe the set of gestures and have the manager recognize them. However, we define only one gesture, grabbing, so recognition is easy.

Gesture recognition is performed by a classifier and a state machine. The classifier recognizes the current flexing of the fingers based on the state of the state machine. The state machine uses the classification state to make transitions. The advantage of this approach is extensibility and support for tresholds (hysteresis) based on the current gesture. Tresholds increase the reliability by preventing flipping between states on the borders because of noise.

Voice commands include “move forward/backward”, “turn left/right”, “look up/down” and “show view of <object>”. The last command will move the viewpoint above the specified object, retaining the direction of the previous viewpoint. Of course, the hand and any object held by it will move with the viewpoint. This provides a quick way to “teleport” objects close to another object (e.g. moving a truck from camp one to camp two without the long travel).

As with the Whiteboard enhancer, the parser uses a simple, task-specific top-down approach. The grammar for the 3D mission planning application too simple and straightforward to justify a more complex approach.

5.2.3 The leukemia diagnosis support application

The leukemia diagnosis support application was originally developed by the Robust Image Understanding Laboratory at CAIP, in cooperation with the University of Medicine and Dentistry of New Jersey [11]. It is an application for analyzing bloodcell images on features (such as shape and texture), and using those features to search for similar images in a database.

The emphasis in the name for the application is on *support*. It is very important to understand that this application does not try to replace a specialist's judgement. What it does is narrow down the list of candidate cell types. With this data, the pathologist can perform her job more efficiently, saving time by not having to search through a book of hunderds of images to find a match.

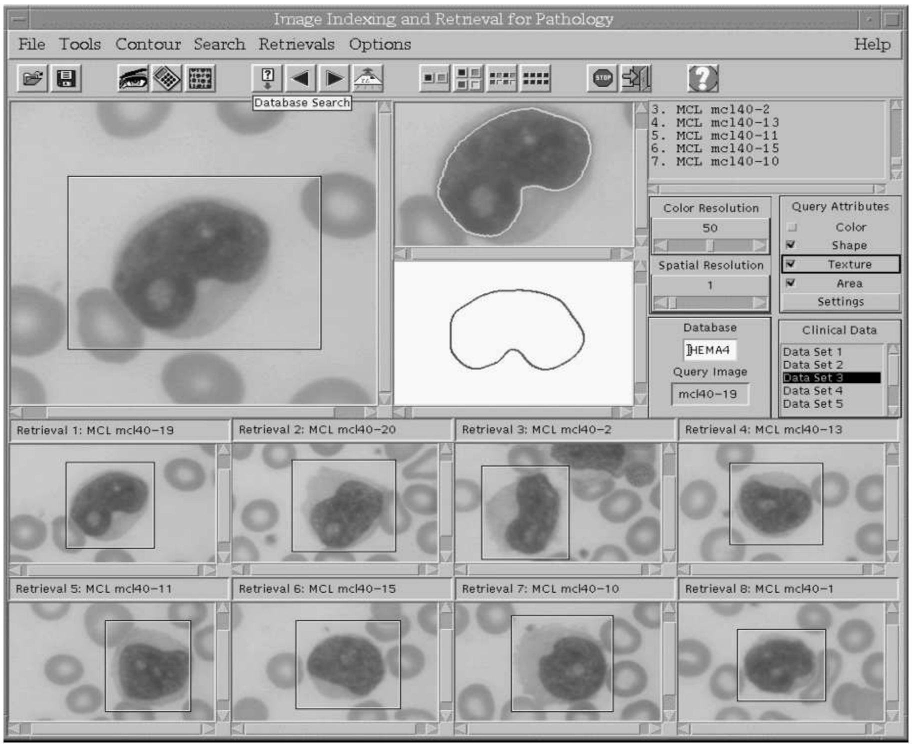

Figure 5.2: The original bimodal leukemia diagnosis support application

The original standalone application (shown in figure 5.2 was bimodal, using speech and mouse. The user interface was completely rewritten to use Java's new Swing graphics library instead of AWT, leaving the feature extraction and database search classes intact (shown in the background of figure 4.2).

Apart from rewriting the graphical user interface, we also added the necessary code for collaboration and a multimodal user interface, grammar-wise based mostly on the original bimodal system, but based on the modality management library and not directly connecting to the speech recognizer.

Using the program is pretty straightforward. First, an image of a bloodcell to diagnose is loaded. This can be done through the menu and the file dialog box, or for 10 sample images in the doctor's virtual workmap (currently implemented as files in a directory), by voice command.

Then, the region of interest is selected. This selection is performed by dragging the mouse over the desired region. Alternatively, mouse emulation by glove, or speech and gaze, can be used.

The region of interest is then segmented (quantized into a lower color resolution to extract outlines) by saying "segment" or clicking the appropriate button. The doctor selects a shape (by rightclicking in it), then searches the database by saying "search the database" or clicking the database search button.

Basically, the application is still bimodal, but the mouse emulation techniques of the manager enables the gaze tracker and tactile glove. Therefore, multimodal fusion is not performed by the application at all.

The parser is based on event translation. Speech commands come in as an event, and actions are performed by generating the appropriate internal events. Since the events in this application have a string as an identifier (this is unusual but not invalid Java practice, it was a design decision by the redesigner of the GUI component), we are able to perform text substitution to translate the commands.

The approach to this is having the substitution rules in a separate textfile, with descriptions like this:

```
class EDU.rutgers.medical.medicalBeans.SegmenterBean {
  "segment the image"  -> "Do-segmentation";
  "segment"            -> "Do-segmentation";
  "open image \([^ ]*\)"-> "Open-speech-image-$1";
}
```

This example construct specifies that the class `SegmenterBean` can receive two types of events, being "Do-segmentation" and "Open-speech-image-<name>", respectively linked to the speech commands "segment" or "segment image", and "open image <name>". The text to search for is specified as a regular expression; the replacement text is specified as a text with $n metacharacters. The functionality for this was provided by the `gnu.regexp` package, a GNU package implementing regular expressions for Java.

The internal event string parsing is handled by a parser for each class receiving these events. The parsers in each class are very simple, specialized and bottom-up.

5.3 Feedback and results

We had experience with the sample implementation of the multimodal mission planning system and the bimodal leukemia diagnosis support system before adding multimodal support to the application described above. We also had discussions within our group on how to improve the multimodal support.

After we did the initial implementations, one of our main sources of feedback were demonstrations of our system to visitors. The reason for this is that people attending the demonstrations are not "spoiled" with knowledge about what our software could do, and therefore did not keep themselves to the programmed commands. This is not a new observation; it is the reason that for some approaches for natural language understanding and discourse, people collect large amounts of sample discourse and use probabilistic methods trained on that data, even for simple tasks.

Another reason that the demonstrations were interesting, is that the conversations with the visitors introduced more noise on the speech recognition output. This is because the conversation's content is typically about the software, and is therefore very likely to contain phrases that are matched by the speech recognition system. This would then result in commands being executed that were never intended, disturbing the flow of the demonstration.

In other words, demonstrations are much closer to actual everyday use than our "clean room" tests. These observations were used to improve our implementation.

5.3.1 Experiences with the Tactile glove

While findings from demonstrations are no replacement for real user experiments or field trials, our project was not yet in the stage where we were going to conduct those. However, they are planned for the last year of the STIMULATE grant.

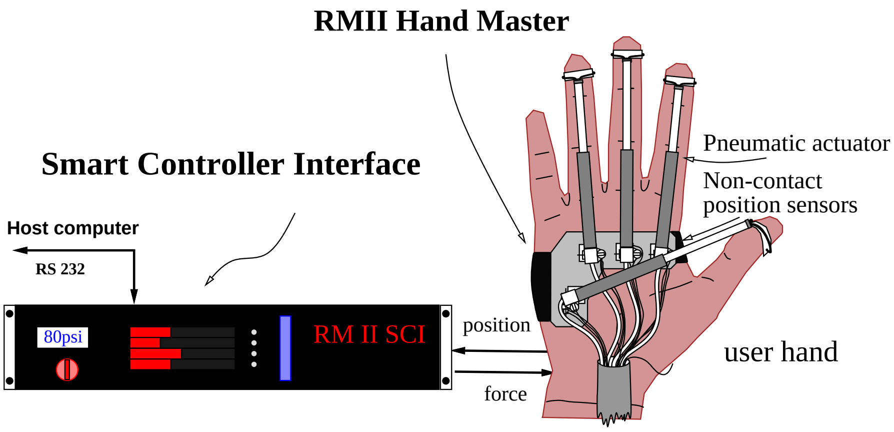

Figure 5.3: The force-feedback tactile glove

We calibrated scaling and translation parameters for the tactile glove for a beam-shaped volume (the `working volume`) next to the seat of the multimodal workstation, corresponding to the reach of the user's hand. This should make it easy and intuitive to find out where to hold the glove to get it to a certain position on the screen.

However, we found that it was sometimes pretty hard to find the working volume. There are a few possible explanations for this:

- Lack of indication where the glove is with respect to the screen when the glove cursor is off-screen (no feedback).
- System lag in displaying the cursor. This can make the cursor not show up at all even if the hand traverses the working volume, when it moves out again too fast. In general, this lag creates disorientation.
- 3D applications can conceal the glove if it is inside them depending on the clipping volume, position of the hand (depth plays a big role here), and objects in the scene. This causes disorientation.

The first problem can be solved by not clipping to the screen, but mapping regions outside the screen to the edge of the screen, with the usual cursor or possibly another icon.

We found that the problem of finding the working volume was mostly a problem for novice users. This first impression is very disappointing and can make the user choose other modalities that seem easier to use.

Another issue with the glove is that the gesture recognition is a bit sensitive. Gestures that are simular during the movement from one state to another can cause misrecognitions resulting in undesirable behavior. Again, for novice users, this is very confusing. An example of this is trying to grab an object, involving bending of all fingers. The gesture for clicking is just bending the thumb. If the user bends the thumb faster than the other fingers, the recognizer may only generate a click instead. More advanced users make up for this behavior by using different order of finger motion to avoid these borderline regions during the movement. This effectively increases the learning curve.

Force feedback is definitely a good thing. It enhances the feeling of immersion, making the glove seem more natural. The only disadvantage with this model is that the feedback has a dead spot when the fingers are completely contracted, moving the pistons all the way in the tube, shutting off the air supply without applying any pressure to the fingers. This is because the air supply is perpendicular to the motion axis of the piston. The result is that when you grab too fast, you get no force feedback until you open your hand a little bit.

Otherwise, using these simple gestures, the glove seemed quite easy to use. After an initial learning curve, getting a feeling for the mapping between the working volume and the screen, and performing the gestures the “right way”, the glove is almost a good replacement for the mouse. The only disadvantage is reduced accuracy. Hand motion is more precise than arm motion, and the Polhemus tracker used to track positions seemed to suffer from noise of all the equipment in the vicinity.

5.3.2 Experiences with the Gaze tracker

The gaze tracker is very promising but needs improvements in two areas: accuracy and the actual methods for applying eye-motion data.

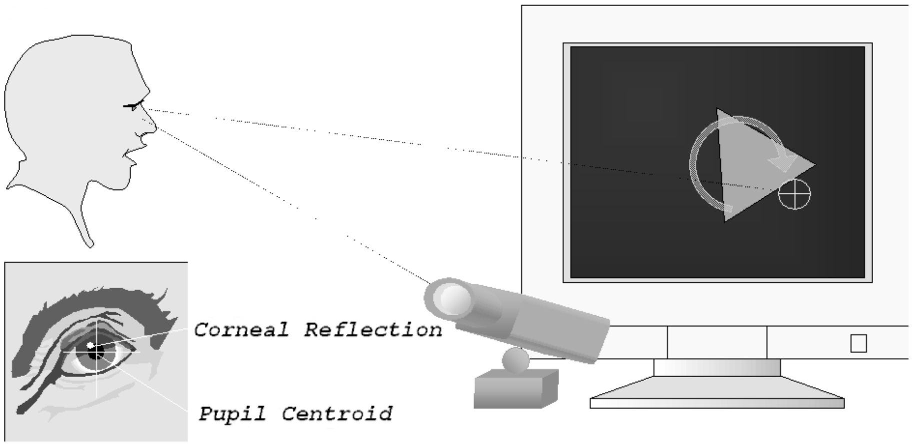

Figure 5.4: The gaze tracker. Gaze location is calculated by computing the angle between the centroid and the corneal reflection

During debugging, trial and demonstration sessions, we found that the gaze tracker that we use is very sensitive to calibration, and that it's performance degrades over time. Sometimes, it would work just great; on other occasions, it would either be systematically off (a fixed offset) after a while, or worse, be very unstable, resulting in a jumpy cursor.

There seemed to be no logic behind this. Both the correct and erratic behavior seemed to have no connection to environmental conditions, time of the day, placement of the tracker, or other setup changes. It was also not just a problem of our program; another program, used to gather the human factors data, resulted in the same behavior. We worked with the people from ISCAN, Inc., who built the eye tracker, but we were unable to resolve this problem.

To put these complaints into perspective, informal discussions with other research groups using gaze trackers (from the same and different manufacturers) showed that reliability is a common problem, indicating that eye tracking technology is still immature.

When the gazetracker performs well, there is only a small and flat learning curve. When there is a small offset, the learning curve gets much steeper, because the user has to make up for the offset, by looking at *another* point than the real point of interest. This can be very confusing and straining, especially because the cursor moving close to the real point of interest.

The human vision system has specialized subsystems for moving objects, which attract attention. In this case, it creates a feedback loop between the involuntary movement to look at the moving cursor, the conscious decision to look somewhere else to make up for the cursor offset, and the delay path between movements of the eye and updating the cursor position.

Even when the gazetracker performs well, this feedback loop can occur, but it is easier for the user to learn to suppress it. However, if consistently accurate equipment is available, it seems to be a better idea to do away with the cursor altogether. While the user has to get used to the computer "magically" knowing where he or she looks, this removes the visual distraction, the net result being stress relief.

A completely different issue with using eye-motion data as a pointing device is that it is not comparable to hand-based pointing devices. This is because when using a hand-based pointing device, hand-eye coordination is used to get the cursor to the desired spot. The eyes are used to find the spot at first, then the hand is moved to move the cursor in position, using visual feedback. When looking for the object to move the cursor to, the eyes move over several candidate locations.

Gaze tracking would eliminate the second step, moving the pointer to the desired location. However, the application will now have to figure out which of the candidate locations is the desired one. To solve this by gaze alone requires deep understanding of our vision system, especially how we scan a document, or search for information in a document. This would require a new intelligent agent, that would need knowledge of the application, specifically about placing, hierarchy and semantics of visual components. It may also not always be possible to decide on a single point. The gaze pattern can be ambiguous.

Alternatively, information form other modalities, most notably speech recognition, can be used to narrow down the possible points of attention and thereby speeding up the decision process. Speech can also be used to disambiguate the possible points of interest in the gaze pattern.

It has already been proven that multimodal fusion can help disambiguating input from the separate modalities. They usually discuss pointing device input to disambiguate speech, but when using a gaze tracker to replace the mouse, it is also possible to do it the other way around.

In our implementation, we included speech commands to simulate mouse events. There are three commands, “click”, “start drag” and “stop drag” to simulate a click, start a mouse drag (holding the button down while moving the mouse), and finishing the drag motion. Using these commands created an awkward feeling, but that may have been the novelty of it. On the other hand, for people paralyzed from the neck down, it can be a good alternative.

Yet another possibility is using other artificial commands to denote points of interest, not by speech, but by specific eye motion sequences. One simple sequence might be staring at the point of interest for a certain amount of time, or another might be blinking. Both have their own problems, and have to be learned just like the speech command.

All in all, we found that gaze tracking is a very promising technology, but we found it less mature than the tactile glove.

5.3.3 Experiences with the speech recognizer

During the project, we updated our speech recognizer. We started out with Microsoft Whisper that came with SpeechSDK 3.0. Soon after version 4.0 came out, we switched to it. All the findings presented here are based on Microsoft SpeechSDK 4.0 (beta 2) with the version of Whisper that comes with it.

We only used the speaker independent training, we never trained the recognizer for a specific user. In general, the performance was good enough to use it this way. We often had to set some of the tresholds to make it less picky (set it to fewer false recognitions). On slower systems, we also needed to lower the accuracy to improve response times.

While recognition accuracy was reasonable in most cases, for some specific sentences, depending on the speaker, the recognition was extremely poor. This made us look into alternative commands to achieve the same; these synonymous commands turned out to be really helpful. Depending on dialect and accent, people would find that it recognized one alternative better than the other.

We found that single word and especially monosyllabic single word commands ruined recognition performance. The recognition of those commands would not be that bad, but the accuracy for other commands would be severely impaired. This is because short utterances tend to fit a lot of starts of sentences, even if there seems to be no connection. Therefore, we tried to stay away from single word commands.

Similarly, some longer commands that are relatively close would at times cause misrecognitions. For example, the command "create <something>" is used for creation of both beans in the workspace (this is handled by the manager) and objects in the Whiteboard and AreaMap. Therefore, there is a long list of things that can be used as an argument to "create". While we had no ambiguity in this situation (all things had different names), the speech recognizer found "hand" and "camp" very similar, resulting in the insertion of a Hand bean (a test bean to test the glove's 3D input in our workspace) into the workspace instead of creating a new camp in the AreaMap. This was very disappointing.

We resolved this and similar problems by extending the sentence. In this case, we changed command for the creation of beans into "create <BeanName> bean". This is very effective.

Another issue that we found, especially during demonstrations, was that the speech recognizer would pick up parts of the conversations and interpret them as phrases in our grammar. This would cause random commands to be executed. Shorter commands have a higher chance of being picked up, but lengthier commands could also be mistakenly recognized because our conversations were usually about this system, resulting in actual partial phrase matches.

We found two reasonable methods for preventing this:

1. Turn off voice commands by speech, and turn it on again with a special command.

2. Use gaze tracking information to determine if the user is talking to the computer.

To turn off voice commands, we simply added a new command, "disable speech". This switches the current grammar over to a simple minimum grammar, which includes a command to turn speech back on. This command is "computer listen", which is long enough to avoid accidental false recognitions during a conversation. The minimum grammar actually has more commands, but unless they are prefixed with the escape sequence "computer", they are ignored.

The other option, using gaze tracking information to determine if the user is talking to the computer, is called *gaze confirmation*. The idea is similar to behavior in human-human communication, where people look at each other during a conversation to confirm the addressee.

It turns out that this information, the gaze cursor being in the screen or not, is a good estimator for determining whether the computer is being talked to or not. However, when the gaze tracker is not properly callibrated or shows erratic behavior, this option is very disturbing. Therefore, we had a command to turn gaze confirmation off ("gaze confirmation off").

We also decided that there should be more synonymous voice commands and that the applications should have had a little bit more intelligent commands. The current set of commands is fairly primitive, and in some cases, not even complete. For testing our framework, this was sufficient, but for field testing, we will need a better interface.

A related issue is that novice users will immediately assume that an application is intelligent and just say anything. For some reason, the fact that it can distinct several blurbs of sound and find out which predefined phrase belongs to it, seems to be enought to think that the computer is sentient. However, a few tries and a word from the demonstrator are usually good enough to guide the user in the right direction.

While the ultimate goal is to make the computer sentient yet our slave, we haven't even remotely reached signs of intelligence yet. The best we can do is make a program look smart, with generalization techniques and pre-programmed responses. However, making the computer *look* smart will invite the user to believe that it *is* smart. The user will be disappointed.

The question is: how smart should we try to make the application? How much effort do we want to put in it, and what can we achieve with it? As long as we have not found a solution to make computers behave smart enough to solve most of our problems through a natural language interface, designers will have to ask themselves this question.

Overall, the speech recognizer seemed to work reasonable, and we were able to work with or around its limitations.

5.3.4 Experiences with the wearable computer

To verify our argument of portability, adaptability and scalability, as well as to experiment with mobile computing, we purchased a Xybernaut MA IV wearable computer [42]. This is basically a belt-mounted scaled-down PC-notebook compatible machine without built-in display. Figure 5.5 shows the gear, and figure 5.6 shows how it is worn.

Our model came with a head-mount display (HMD). For installation, there is an extension adapter which provides support for keyboard, mouse, floppy

Figure 5.5: The Xybernaut wearable computer

drive, external monitor, and other devices. In our configuration, the only input devices available are a steerable button which acts like a mouse and a close-talk microphone on the head-mount display. There is a speaker in the headset, and the display has a resolution of 640x480 (full color). We use a wireless LAN to connect it to our network.

Most applications for this kind of computer is in situations where the user has to have both hands free. This can be in situations like on-site repairs with remote assistance or on-site quality control. This creates an entirely different environment than the multimodal workstation.

We found that the mouse replacement, the steerable button, needed some training to be used efficiently. It is located on the CPU unit on the belt, and is too sensitive for novice users. Besides being a bit hard to use, it also defeats the idea of hands-free computing.

Obviously, a gaze tracker built into the HMD would probably be a good thing to have. We could not find commercial implementations that would fit into an HMD and its processing equipment small enough to wear. ISCAN is currently working on a PCMCIA card for their HMD version of a gaze tracker.

Because of these limitations, people are more inclined to use speech recognition. This is one of the reasons that we need to improve our grammars and parsers; our current set of voice commands is simply not powerful enough to cover the complete range of interactions.

Another issue we found was that our workspace and application contain too much visual information, requiring more screen estate than is available on the tiny display. This is typical for applications that were developed on desktop systems, that tend to have higher resolutions (at least 1024x768 for modern desktop machines).

We found that the DISCIPLE desktop performed reasonably on this machine, which was not as powerful as our desktop systems. Despite the limited voice commands and overpopulated screen, it was still quite usable, albeit not hands-free.

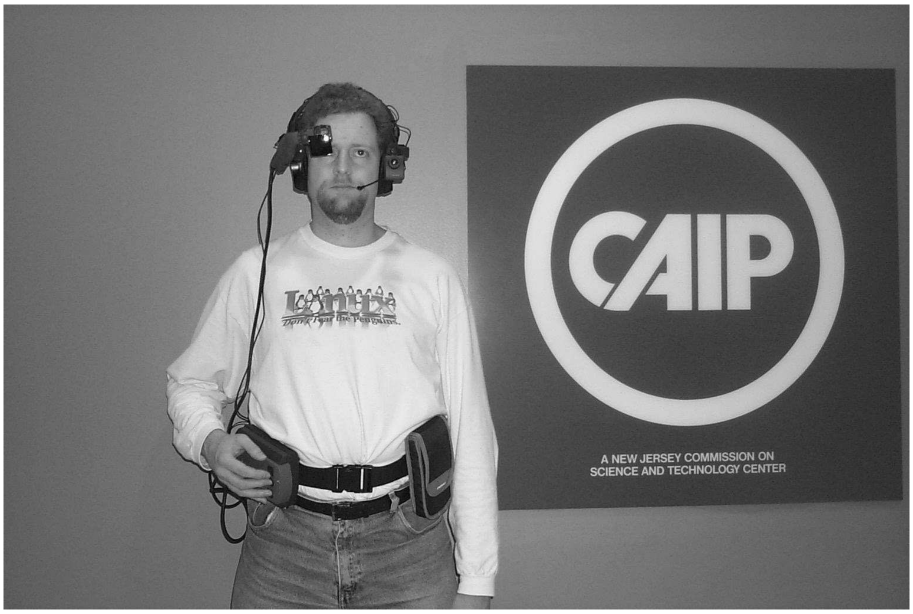

Figure 5.6: The wearable computer in action

5.3.5 Experiences with the software environment

We found the DISCIPLE desktop system quite stable. It is a bit slower than desired, probably due to its centralized design, but it is very well usable. The collaboration works fine, and depending on the network delay, remote updates can be reasonably fast.

The user interface is clear and consistent once the idea behind the collaboration space and workspace is understood. This is actually a minor problem for novice users; the organization-place-workspace hierarchy is not always directly understood.

The multimodal interface is also pretty clear and consistent, but the setup is somewhat crude. More automatic setup, workstation profiles (to have defaults for available equipment for specific workstations) and user preferences would be a good idea.

Overall, the whole prototype system performs very nicely.

5.4 Known issues and suggestions

Our prototype environment can be enhanced in various ways. Some of them are not really research topics, others are really hard problems that have not been solved yet. Here follows a list of known issues.

Multimodal fusion The method, complexity nor accuracy of multimodal fusion were specifically addressed in this work. The framework only provides methods for receiving live updates and retrieving older data. However, the fusion agents in the manager and the applications are very limited at the moment, both in features and extendibility. Research towards a framework or supporting library on top of the modality management library would improve this.

Design of the natural language interface Our current natural language interface is, like our multimodal fusion, not very advanced, since the emphasis for this work was on the modality management aspects for the multimodal user interface. Research on employing more flexible (but more complex) parsers (either existing methods from literature or a new setup) would be necessary to extend the current parsers beyond one more order of grammatical complexity. This might, again, be provided as a library, possibly integrated with a fusion library as described above.

Higher accuracy of input devices The accuracy of our current input devices leaves much to be desired. This is not as much with respect to the resolution of measurements, as much as with respect to the amount of noise. Both the gaze tracker and the glove show very jumpy cursors, indicating high noise levels. The speech recognizer performance is barely acceptable, and should improve an order of a magnitude for more convenient usage.

Cross-platform input/output devices Our current input/output devices are each bound to a single platform. The gaze tracker and the glove can only be connected to a UNIX (potentially only Sun) workstation, while the speech recognizer only works on 32 bit Microsoft operating systems.

It would be desirable to support different speech recognizers on different platforms. Unfortunately, there is no real cross-platform library for speech subsystems. Different OS-level drivers would have to be written for different systems, which can be connected to the environment with TCP connections, without any changes to the environment.

Similarly, it would be desirable to have the gaze tracker and glove run on our NT demo system. This is, again, a problem of the drivers, but also a hardware problem; either the NT system does not have enough serial ports (which can be solved by adding a serial port card) or it does not have enough ISA slots to hold all the cards, which would be necessary to do away with the gaze tracker CPU and the Polhemus Tracker CPU.

Easier startup procedures The current startup procedure is quite complicated for most people, and not very forgiving about the order in which some actions have to be carried out (for more information on the setup procedure, see appendix D). The DISCIPLE desktop is not the most friendly environment to set up, and the multimodal manager and devices drivers are not that easy to use either. While it is not really complicated, and the steps can be execute without knowing what it does, it is especially unfriendly towards novice users. Automatic setup scripts on the UNIX workstation would help a lot here, and adding scripting tools to the DISCIPLE environment might be a good idea.

Performance enhancement While the multimodal workstation is usable, it would not hurt to improve the performance. Cursor updates seem to be slower than necessary, and drawing the 3D terrain seems to be slow, despite the state-of-the-art PC with OpenGL accellerator.

Our current ideas are that the Swing component library is not used properly by some applications, causing unnecessary redraws of the entire screen. While our scene does not seem to be exceptionally large, the Java3D library is probably not yet really optimised and has trouble with the number of polygons in our terrain map (in the order of 10,000 per scene). The current texture map is not too large, only 256 by 256 pixels, resulting in at most 256KB of memory usage.

Another source for delays seems to be the communication layer in DISCIPLE. All in all, it was really hard to find the sources of the performance decay, so the probably causes mentioned here are not more than educated guesses.

Chapter 6

Results and Conclusions

6.1 Summary

The introduction in Chapter 1 describes the motivation for this research topic, the status of the project at the beginning of this thesis work, and directions for further research. The conclusion was that in order for the project to become a useful testing and evaluation platform for multimodal human-machine communication, we needed an environment to easily write or convert applications for, as well as run any or all of those on demand.

Chapter 2 discusses the requirements from different perspectives (the user, the project and the application programmer). A conceptual model, designed to meet these requirements, is then proposed, to form the base for the programming environment. The model proposes to separate the code for interfacing the I/O devices from the applications, abstracting properties of devices and using an arbitration layer to regulate application access to the devices. To achieve indepence of code and interoperability, a common API contained in a platform independent Java library would be used.

Chapter 3 explains the design of the programming environment, a library containing a set of Java classes and interfaces. The discussion is mostly focussed on explaining the structure of the library and the purpose of classes and methods.

Chapter 4 describes the prototype implementation of the multimodal managing agent, called the Octopus. Each of the modality drivers is described briefly, the internal design of the manager and features and how specific issues, which have no specific solution required by the library, are solved.

Chapter 5 discusses the implementation of multimodal extensions in example applications briefly, as well as user test results. These test results include both our own impressions and feedback from users and observers during demonstrations. The results are split with respect to focus on each of the modalities and applications.

6.2 Results

With respect to the goals described in section 1.1, this thesis presents the following contributions:

1. We established a set of reasonable user interface requirements and software requirements (with respect to our environment and project goals).
2. We designed a programming environment, a library of Java classes and interfaces, as a framework to develop the managing agent and multimodal applications (or extensions) in. The initial design of this library was good enough that only minor additions had to be made during the development of the manager and application extensions. It is also flexible enough to allow at least the entire range of alternative modal input and output devices available at CAIP, being the tactile glove, gaze tracker and the speech subsystem, to be used successfully in different applications.
3. We designed and implemented a modality manager, using the modality management libraries, at first in a stand-alone environment, and later, when it became available, in the DISCIPLE desktop. We wrote a multimodal extension bean for two applications (the WhiteBoard and the AreaMap), and added multimodal extensions to two more applications (a 3D terrain map for manipulating objects, and a leukemia diagnosis support tool).
4. We have successfully demonstrated our multimodal environment, both on our multimodal workstation as well as on a first-generation commercial wearable computer. Preliminary test results were described, but scripted, controlled and comparable tests have not been made yet.

6.3 Towards future multimodal environments

In the design and implementation of our multimodal interface, we used an incremental approach. Based on the experience of our prototypes, we could envision next-generation multimodal interfaces. One of the main restrictions we encountered is that we had to adapt *existing* applications.

To achieve a much more natural user interface for (by definition artificial) computing environments, we need radical changes in our views on user interfaces. We may have to review everything we think we understand about it. One of the abstractions that is widely used in user interfaces is the *application*.

Our vision is that applications are bulky, complex artifacts that do not bear as much resemblance to tools in the real world as they should. Since they tend to be unfriendly to users, they apparently do not connect very well to our train of thought. Maybe we should abandon applications, and move to an object-oriented approach where actions are defined on objects.

Recent developments in office suites allow the user to use more diverse objects in documents, across applications, but it is only a half-hearted attempt and therefore very limited. JavaBeans promises something along these lines, which is sometimes called “end user programming”, but fails to deliver because it would still be too hard to program for the typical end user.

An object based user interface could not only be more intuitive, but it can also help to solve some of the problems we have now to incorporate knowledge into our computing environment. Applications separate data from meaning, forcing the programmer to build a separate user interface. Semantically annotated objects could deliver the knowledge for rendering on different output devices, what relations they have with respect to other objects, and what kinds of actions they are suited for in which role.

A multimodal interface built on this concept is outlined in figure 6.1. In this concept, a context tracker keeps track of different contexts, being the different objects that can be directly or indirectly addressed at any given time. The context tracker notifies the dialog manager about changes, and activates different action agents when appropriate.

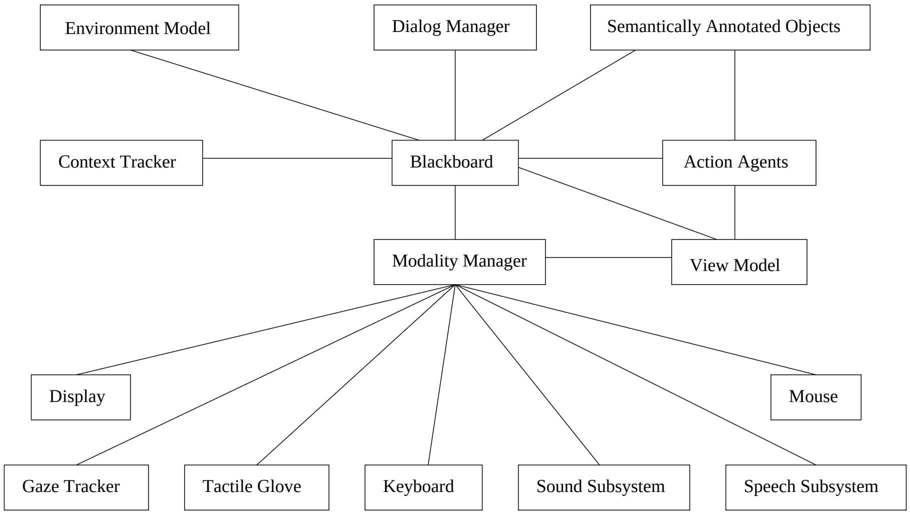

Figure 6.1: Sketchy model of components in an applicationless multimodal workstation

The grammar is changed continuously, as the context changes. The modality manager is reduced to a input-output handler and preprocessor, translating between devices language and the language used on the blackboard. The dialog manager performs multimodal fusion and decides which action agent to invoke. The dialog manager can consult the objects, agents and the environment model to reach a decision. Disambiguation should also be resolved on this level.

Objects are rendered into device primitive by the modality manager. How objects are presented is determined by the view models. Manipulations on the objects and view models is performed by the action agents.

The role of the environment model is to keep track of knowledge that is not stored by any of the objects in the system. This might include data such as the user’s name, privileges and preferences, the system name, semantic models for determining who is addressed in conversations, and “common sense”.

While the experience we gathered by building two prototype multimodal environments is very useful, we cannot build this system without many more years of research. People in the field of AI are only beginning to understand issues such as building common sense databases, storing and processing semantic knowledge, dialog management, and robust parsing.

It is clear that we have a long way to go before we have a really friendly and natural interface for our computers. Great advances in understanding the different kinds of human languages will be necessary, as well as changes in user interface concepts.

Boi Sletterink,
Delft,
August 1999

Bibliography

[1] Efun: add_action. Available at: http://www.darkrealms.com/~dwind/-efuns/functions/add_action.html.

[2] Maurits Andre, George Viorel Popescu, Arshad Shaikh, Attila Medl, Ivan Marsic, Casimir A. Kulikowski, and James L. Flanagan. Integration of Speech and Gesture for Multimodal Human-Computer Interaction. In *Proceedings of the 2nd International Conference on Cooperative Multimodal Communication*, pages 20–27, Tilburg, The Netherlands, January 1998.

[3] Maurits J.A. André. Multimodal human-computer interaction. Technical report, Delft University of Technology, Knowledge Based Systems group, Zuidplantsoen 4, 2628 BZ Delft, the Netherlands, August 1998. Masters thesis.

[4] R. A. Bolt. Put–That–There: Voice and Gesture at the Graphic Interface. *Computer Graphics*, 14(3):262–270, August 1980.

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[6] Grigore Burdea. *Force and Touch Feedback for Virtual Reality*. John Wiley & Sons, New York, 1996.

[7] Adam Cheyer and Luc Julia. Speech: A Privileged Modality. In *Proceedings of the EUROSPEECH'97*, Rhodes, Greece, September 1997.

[8] Adam Cheyer, Luc Julia, and J. C. Martin. A Theoretical Framework for Multimodal User Studies. In *Proceedings of the 2nd International Conference on Cooperative Multimodal Communication*, Tilburg, The Netherlands, January 1998.

[9] CMU Interactive Systems Labs. Gesture and speech integration. Carnegie-Mellon University, Pittsburgh, PA, 1998. Available at http://www.is.cs.cmu.edu/ISL.multimodal.gesture.html.

[10] Phil R. Cohen, M. Johnston, D. McGee, Sharon Oviatt, and J. Pittman. QuickSet: Multimodal Interaction for Simulation Set-up and Control. In *Proceedings of the 5th Applied Natural Language Processing Meeting*, Washington, DC, 1997.

[11] Dorin Comaniciu, Peter Meer, David Foran, and Attila Medl. Bimodal System for Interactive Indexing and Retrieval of Pathology Images. In *Submitted to the 4th IEEE Workshop on Applications of Computer Vision (WACV'98)*, Princeton, NJ, October 1998. Available at `http://www.caip.rutgers.edu/~comanici/jretrieval.html`.

[12] B. Dorohonceanu and I. Marsic. A desktop design for synchronous collaboration. In *Proceedings of Graphics Interface '99 (GI'99)*, Kingston, Ontario, Canada, June 1999.

[13] J. Flanagan, C. Kulikowski, I. Marsic, G. Burdea, J. Wilder, and P. Meer. Nsf – stimulate: Synergistic multimodal communication in collaborative multiuser environments (annual progress report). Technical report, Rutgers University, Center for Computer Aids for Industrial Productivity (CAIP), 96 Frelinghuysen Road, Piscataway, NJ 08854-8088, USA, 1999. Contract Number: IRI - 9618854.

[14] James L. Flanagan. Technologies for Multimedia Communications. *Proceedings of the IEEE*, 84(4):590–603, April 1994.

[15] James L. Flanagan. Synergistic Modalities for Human/Machine Communication. In *Proceedings of the IEEE Workshop on Speech Recognition and Understanding*, Santa Barbara, CA, December 1997. Plenary talk.

[16] James L. Flanagan. Multimodality. In R. Cole and J. Mariani, editors, *Survey of the State of the Art of Human Language Technology*, chapter 9, pages 277–300. National Science Foundation and Directorate General XIII of the European Commission, Cambridge University Press, (in press). Available at `http://www.cse.ogi.edu/CSLU/HLTsurvey`.

[17] James L. Flanagan and Ivan Marsic. Issues in Measuring the Benefits of Multimodal Interfaces. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'96)*, pages 163–166, Munich, Germany, April 1997. Invited article.

[18] The official fvwm homepage. Available at: http://www.fvwm.org/.

[19] X. Huang, A. Acero, F. Alleva, D. Beeferman, M. Hwang, and M. Mahajan. From cmu sphinx-ii to microsoft whisper: Making speech recognition usable. Technical report, Microsoft Research, 1994. Report number MSR–TR–94–20.

[20] Human-Computer Interaction Laboratory. Research project descriptions. University of Maryland, College Park, MD, 1998. Available at http://www.cs.umd.edu/projects/hcil/Research/projects.html.

[21] Iscan, inc.: Eye & target tracking instrumentation. Available at: http://-www.iscaninc.com/.

[22] W. D. Keidel. Information Processing by Sensory Modalities in Man. In *Cybernetic Problems in Bionics*, pages 277–300. Gordon and Breach, 1968.

[23] Intermediate lpc. Available at: http://www.nmt.edu/~mattg/mysite/LPC/-intermediate/chapter2.html.

[24] Introduction into lpc. Available at: http://mud.stack.nl/INTRO.html.

[25] I.S. Mackenzie. Movement time prediction in human-computer interfaces. In R. M. Baecker, W.A.S. Buxton, J. Grudin, and S. Greenberg, editors, *Readings in Human-Computer Interaction (2nd Ed.)*, pages 483–493. Morgan Kaufmann, Los Altos, CA, 1995.

[26] Ivan Marsic and Bogdan Dorohonceanu. An Application Framework for Synchronous Collaboration using Java Beans. In *Proceedings of Hawai'i International Conference on System Sciences (HICSS-32)*, Maui, Hawai'i, January 1999. Software Technology Track.

[27] P. McKevitt, editor. *Integration of Natural Language and Vision Processing: Computational Models and Systems.* Kluwer Academic Publ., Dordrecht, The Netherlands, 1995.

[28] Attila Medl, Ivan Marsic, Maurits Andre, Casimir A. Kulikowski, and James L. Flanagan. Multimodal Man-Machine Interface for Mission Planning. In *Proceedings of the AAAI Spring Symposium on Intelligent Environments*, pages 41–47, Stanford, CA, March 1998.

[29] Navy Research Laboratory. Virtual Reality Laboratory Web Page, 1997. Available at `http://www.ait.nrl.navy.mil/vrlab`.

[30] Sharon Oviatt, A. DeAngeli, and K. Kuhn. Integration and Synchronization of Input Modes During Multimodal Human-Computer Interaction. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'97)*, pages 415–422, Atlanta, GA, 1997.

[31] L. Rosenblum, J. Durbin, R. Doyle, and D. Tate. Situational Awareness Using the VR Responsive Workbench. *IEEE Computer Graphics and Applications*, 17(4):13–15, April 1997.

[32] Rajeev Sharma, Thomas S. Huang, and Vladimir I. Pavlović. Multimodal Framework for Interacting With Virtual Environments. In Celestine E. Ntuen and Eui H. Park, editors, *Human Interaction With Complex Systems: Conceptual Principles and Design Practice*, pages 53–71. Kluwer Academic Publ., Dordrecht, The Netherlands, 1996.

[33] Rajeev Sharma, Vladimir I. Pavlović, and Thomas S. Huang. Toward Multimodal Human–Computer Interface. *Proceedings of the IEEE*, 86(5):853–869, May 1998.

[34] Ben Shneiderman. *Designing the User Interface: Strategies for Effective uman-Computer Interaction.* Addison–Wesley Publ. Co., Reading, MA, 2nd edition, 1992.

[35] Ben Shneiderman. Beyond Intelligent Machines: Just Do It! *IEEE Software*, 10(1):100–103, January 1993.

[36] Ben Shneiderman. Dynamic Queries for Visual Information Seeking. *IEEE Software*, 11(6):70–77, June 1994.

[37] M. T. Vo, R. Houghton, J. Yang, U. Bub, U. Meier, Alex Waibel, and P. Duchnowski. Multimodal Learning Interfaces. In *Proceedings of the ARPA Spoken Language Technology Workshop*, Austin, TX, January 1995.

[38] M. T. Vo and C. Wood. Building an Application Framework for Speech and Pen Input Integration in Multimodal Learning Interfaces. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'96)*, pages 3545–3548, Atlanta, GA, May 1996.

[39] Alex Waibel. Multimodal human–computer interaction. In *Proceedings of the 3rd European Conference on Speech Communication and Technology (Eurospeech'93)*, page 39, European Speech Communication Association, Berlin, September 1993.

[40] Alex Waibel, M. T. Vo, P. Duchnowski, and S. Manke. Multimodal interfaces. *Artificial Intelligence Review*, 10(3–4), 1995.

[41] M. Wingfield. Applied Visualization Techniques for Virtual Environments. MITRE Co., Bedford MA, 1997.

[42] Xybernaut product information. Available at: http://www.xybernaut.com/-F00005.htm.

Appendix A

HCI International'99 Paper

The following paper is a copy of the extended abstract for the student poster presentation accepted for the HCI International'99 conference held in Munich, August 22–27 1999.

This page intentionally left blank.

Appendix B

Article Specom

The following text is an abstract of an article for Specom'99, to be held in Moscow, Russia.

This page intentionally left blank.

Appendix C

Considerations for Writing Applications

The following article is a an excerpt of a tutorial on guidelines for writing multimodal applications. While originally intended for internal use, it shows my ideas about implementing multimodal applications. The original document was much larger and included a discussion of the modality management classes (which is discussed in more detail in this thesis) and a small and a much larger example, which are too large to be printed here.

This page intentionally left blank.

Appendix D

Multimodal Workstation Manual

The following paper is a user's manual for the multimodal workstation.

This page intentionally left blank.