Diss. ETH No. 16074

Bluebottle : A Thread-safe Multimedia and GUI Framework for Active Oberon

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Thomas Martin Frey
Dipl. Informatik-Ing. ETH
born February 01, 1975
citizen of Gontenschwil AG, Switzerland

accepted on the recommendation of
Prof. Dr. Jürg Gutknecht, examiner
Prof. Dr. Moira Norrie, co-examiner

2005

Acknowledgments

First of all, I would like to thank Prof. J. Gutknecht for the opportunity to work in his group and for his liberal supervision of the thesis. Prof. M. Norrie kindly accepted to be co-examiner and provided much-appreciated feedback.
I would like to thank P. Muller and P. Reali for writing the Aos kernel and Active Oberon compiler respectively and for their encouragement when I started the Bluebottle framework five years ago.
I would also like to thank the previous developers and contributors of Oberon. Especially the work and comments of E. Oswald and E. Zeller had great influence on the project.
I am also very grateful to B. Kirk for reading an early version of this thesis and providing valuable suggestions which helped to enhance its structure and reduce the number of mistakes.
Special thanks also go to B. Egger, P. Reali, P. Muller, P. Kramer and A. Fischer who read and commented on various chapters of this thesis and had otherwise great influence on the project.
Too many people of the Oberon/Bluebottle community contributed to the current system as to mention them all. D. Keller, R. Güntensperger, L. Blaeser, P. Lehmann, F. Nart and M. Szediwy of the *Programming Languages and Runtime Systems Research Group* did a lot of work with the Bluebottle system and provided good feedback and improvements.
I also wish to thank the past and present members of the institute for computer systems for creating an interesting work environment. E. Ruiz, H. Sommer and R. Hidalgo helped a lot in avoiding administrative problems and creating a good work environment.
About forty diploma and semester projects have been performed on the Bluebottle system and have provided feedback relevant to the system and this thesis. I would like to collectively thank the students who did these projects for their invested time and interest.
I would also like to thank my parents Andreas and Erika for their support and patience during the many years of study.

Table of Contents

Abstract

Thirty years after the first introduction of the desktop metaphor as a means of human computer interaction in general purpose computers, this thesis reconsiders and evaluates interaction methods on a technical and conceptual level in the light of the progress of hardware and software technology over the last decades. Notably the increased CPU speed and memory capacity, specialized processor instruction set extensions, and a clear recent trend to commodity multiprocessor systems and multi-threaded processors lead to new implementation requirements and decisions. Modern computer systems deliver the computational power for innovative extensions of the traditional desktop metaphor.

We describe the concepts and architecture of a new general purpose graphical user interface and multimedia framework and their thread-safe implementations in *Active Oberon*. The proposed user interface combines in a new way elements taken from the traditional desktop metaphor with interaction techniques known from zoomable and textual user interfaces. A concurrent display space manager with support for translucent free-form windows in a conceptually unlimited zoomable display space was developed and serves as proof of the feasibility of the proposed interaction concepts as well as for the evaluation and discussion of new implementation strategies designed for today's systems. The multimedia framework consists of abstract APIs for different multimedia formats and a plug-in structure for concrete implementations.

While the discussed framework can take advantage of one or more general purpose CPUs with possibly specialized instruction set extensions for multimedia or vector calculations, it is designed not to rely on any special purpose hardware for graphics acceleration. The design for modern general purpose processors offers several advantages over a design for special purpose hardware. One of the main advantages is the simple system architecture that matches or outperforms commercial hardware accelerated systems in common situations through structural advantages. It also allows the system to be easily ported to different hardware platforms, especially to small devices such as wearable computers. The portability of the framework has been demonstrated with a port to the QBIC wearable computer that has been developed at ETH Zürich.

The framework has been developed on top of the multiprocessor implementation of the *Active Object Runtime System* for Intel SMP systems and the single processor ports for the ARM and XSCALE processors.

Kurzfassung

Dreissig Jahre nach der Einführung der Desktop-Metapher als Benutzerschnittstelle für Personal Computer, überdenkt diese Thesis die Methoden des Zusammenspiels von Mensch und Maschine auf konzeptueller und technischer Ebene im Licht des Fortschritts in Hard- und Software. Der enorme Zuwachs der Prozessorgeschwindigkeit und der Speichergrösse sowie die Einführung spezialisierter Prozessorerweiterungen und ein deutlicher Trend hin zu Mehrprozessorsystemen und nebenläufigen Prozessoren führen zu neuen Anforderungen und Überlegungen bezüglich der Realisierung eines modernen Systems für grafische Benutzerschnittstellen. Moderne Computersysteme liefern die nötigen Ressourcen für innovative Erweiterungen der traditionellen Desktop-Metapher.

Wir beschreiben die Konzepte, Architektur und thread-sichere Realisierung eines generischen Frameworks für graphische Benutzerschnittstellen und Multimedia in *Active Oberon*. Die vorgeschlagene Benutzerschnittstelle verbindet in neuer Weise Elemente der traditionellen Desktop-Metapher mit Interaktionselementen von zoombaren und textuellen Benutzerschnittstellen. Als Machbarkeitsstudie sowie zur Diskussion und Evaluation der vorgeschlagenen Konzepte wurde ein nebenläufiges GUI system mit Unterstützung für durchscheinende Freiformobjekte auf einem konzeptuell unbeschränkten zoombaren Darstellungsbereich realisiert. Das Multimedia-Framework definiert eine Reihe abstrakter APIs für verschiedene Medienformate sowie eine Plug-in Schnittstelle für konkrete Implementationen.

Während das beschriebene Framework von mehreren Prozessoren mit eventuell vorhandenen Vektoreinheiten Gebrauch macht, wurde bei der Realisierung darauf geachtet, keine Grafikbeschleunigungshardware vorauszusetzen. Die ausschliessliche Verwendung allgemeiner Prozessoren bringt gegenüber der Verwendung von Spezialhardware einige Vorteile. Der Hauptvorteil liegt in der einfacheren Systemarchitektur, die durch strukturelle Vorteile in häufigen Situationen im Vergelich zu hardwarebeschleunigten kommerziellen Systemen konkurrenzfähige oder sogar überlegene Geschwindigkeit zeigt. Ausserdem kann das System dadurch einfach auf unterschiedliche Hardwareplattformen portiert werden, insbesondere auf kleine Geräte wie sie zum Beispiel in Kleidung integriert wird. Die Portierbarkeit des Frameworks wurde unter anderem mit einer Portierung auf den wearable Computer QBIC, der an der ETH Zürich entwickelt wurde, gezeigt.

Das Framework wurde auf der multiprozessor Version des *Active Object Runtime System* für Intel SMP Systeme entwickelt und läuft auch auf den einprozessor Variationen für ARM und XSCALE Prozessoren.

The real voyage of discovery consists
not in seeking new landscapes,
but in having new eyes.

— Marcel Proust (1871 - 1922)

1 Introduction

Vision is the art of seeing things invisible

— Jonathan Swift (1667 - 1745)

1.1 Motivation

Since the realisation of the first personal computer with a graphical user interface, the *Alto* [61] at Xerox PARC in 1973[1], the available computing power of personal computers has been significantly improved. Today, even the cheapest general purpose computers offer about 2000 times more RAM and one or more CPUs [111] that is more than 2000 times faster than the Alto. Even wearable computers integrated into clothes run about three orders of magnitude faster [2].

At the same time, the displays for general purpose graphical user interfaces evolved from black and white to true colour and offer about twice the number of pixels on the screen. The GUIs of most systems still use rectangular windows, icons, menus and pointers much like the PARC *Alto* system. Such systems are sometimes referred to as *PUI*s for PARC User Interface. In the HCI community the rather disdainful acronym *WIMP* for Windows, Icons, Menus and Pointers is often used to refer to PUI like interfaces when pointing out its weaknesses or when comparing it to suggested new systems .

While PUIs have evolved in their thirty year existence, for example with the addition of so called *tool-tip windows* that allow additional information to be provided where needed without taking away the scarce display space, no revolutionary changes comparable to the step from the command line user interface to graphical windowing systems have occurred. The noticeable improvements in graphical user interfaces can be considered small compared to the improvements in computer hardware. There is more than one reason for the lack of adoption of radically different user interface ideas:

[1] The Alto had 64k 16bit words of memory including the framebuffer, a 606x808 memory bit map. The processor ran at 400k instructions per second [68].

- The *desktop metaphor* [2] used in most current GUIs with objects lying on a desktop works well in combination with the human spatial memory [23] [14] [98].
- Windowing systems are in accordance with the *gestalt laws*[3] [19] and are well compatible with human perception.
- The current PUI style GUIs have become a standard. The positive aspect of this standardisation is that users can easily switch between different systems and versions without having to learn new concepts. On the other hand, standards can easily prevent the introduction and acceptance of new and better systems that offer measurable advantages [85].

User interfaces that strongly differ from the PARC user interface can be found in research labs and computer games. While many research groups all over the world focus on virtual and augmented reality, tactile or other non-graphical user interfaces for special purposes applications, the research on general purpose GUIs for workstations, personal and wearable computers has not been completely abandoned. The current general purpose GUI research is mainly focused on investigating zooming user interfaces and 3d extensions of the PUI. So far, none of the revolutionary new proposals was successful in main stream computing.

1.2 Contributions

In this thesis we suggest and implement *Bluebottle*, a new general purpose graphical user interface framework and multimedia controller on the basis of the Active Object System *Aos* [80]. The proposed user interface combines in a new synergetic way elements taken from traditional PUI systems with interaction techniques known from zoomable and textual user interfaces resulting in a flexible universal display space. The implementation of the proposed system is designed to run on systems from wearable devices up to multiprocessor workstations, efficiently using the available general purpose computing resources.
Apart from proposing and implementing a new user interaction metaphor, this thesis also revisits and implements under the aspects of thread-safety and internationalisation all important elements of current GUI systems such as the topics of *graphical primitives*, *GUI components*, *skinnability*, *input method editors*, *texts* and *fonts*. Extensibility of the primitives is provided with the consequent use of plug-in structures. For example, the support for TrueType and CCG fonts has been added by implementing the font object interface and simply registering the respective generator procedures in the system configuration, an XML file. The CCG font plug-in is especially interesting since it allows more than 80'000 Chinese, Japanese and Korean glyphs to be stored in a file of only 2MB compared to about 30MB in typical TrueType fonts. The

[2] *Desktop metaphor* is quite a far fetched name for current PUI user interfaces. First of all, the *desktop* space on the screen is much too small to represent a real desktop, it is rather a table in front of an airline seat in economy class. Second it is very uncommon to find trashcans, file-cabinets and menu-bars on a real desktop.

[3] The important "gestalt" laws are proximity, similarity, closure, good continuation, good form (Prägnanz), figure/ground and common fate.

small font file becomes possible by a recursive compositioned definition of the glyphs out of other glyphs, based on an analysis of the glyph structures. The small memory footprint allows complete CJK font support even for small wearable devices.
The system provides a general model for hierarchical data structures as well as a parser for internalising streams of XML documents and a externalizer that serialises an in-memory data structure to an XML stream. XML documents are used throughout the system to store configuration information, texts as well as composed component structures.
The multimedia framework defines a number of encoding and decoding interfaces for various media such as *audio*, *video*, *still images* and *texts* as well as a plug-in mechanism for actual implementations of encoders and decoders that can be used by application programs in a unified manner.

The main contributions of this work are:

1. Conceptual improvements to the traditional PARC user interface class in combining elements from zoomable and textual user interfaces with traditional PUI elements.

2. Proposal of implementation strategies to dissolve the borders of different application programs and achieving a more flexible and efficient user interface while reducing the overall system complexity.

3. A classification of deadlocks types in user interface processes and the design of a simple and practical locking scheme based on sequencing objects .

4. A thread-safe and system-wide text system supporting internationalised character sets, text styles and managed text-position for meta-information on the text.

5. Release of Bluebottle, a flexible and extensible thread-safe user interface and multimedia framework for the Active Object System (Aos) that is able to run sophisticated applications, not only on workstations and desktop systems, but also on wearable computers, measuring up to, and even outperforming, commercial hardware accelerated systems in common situations through structural advantages.

6. A number of case studies delivering proof of concept.

1.3 Overview

This thesis is structured as follows:

Chapter 2 discusses several different user interface concepts from the human computer interaction (*HCI*) point of view, with emphasis on the advantages and problems of zoomable and textual user interfaces from which the Bluebottle user interface concept and metaphor is derived. In section 2.6 the Bluebottle GUI is introduced.

Chapter 3 discusses the need for a strong system-wide integrated text system and presents the concepts of its implementation in the Bluebottle system.

Chapter 4 describes and evaluates the graphics framework that forms the basis for the Bluebottle display space manager and component framework.

Chapter 5 introduces the Bluebottle display space manager in concepts and implementation.

Chapter 6 presents the Bluebottle GUI component system and compares it to a number of related systems with a focus on synchronisation strategies.

Chapter 7 describes the Bluebottle sound system.

Chapter 8 introduces the Bluebottle codec framework that is responsible for internalising and externalising images, sounds, movies and texts.

Chapter 9 lists a number of case studies using aspects of the user interface and multimedia framework described in the previous chapters.

Chapter 10 draws conclusions and points to possible future directions.

Appendix A provides a number of commented programming examples relating to the different topics discussed in the previous chapters.

2

User Interface Concepts

In the struggle for survival, the fittest win
out at the expense of their rivals
because they succeed in adapting themselves
best to their environment.

— Charles Darwin (1809 - 1882)
The Origin of Species, 1859

This chapter discusses user interface concepts mainly from the human computer interaction *HCI* point of view. It starts with a general introduction to user interfaces, and then focuses on textual and graphical user interfaces. Four different user interface concepts are discussed in more detail: Command Line Interfaces (CLIs), PARC style user interfaces (PUIs), Textual User Interfaces (TUIs) and Zooming User Interfaces (ZUIs). This discussion leads to the proposition of a new user interface concept, combining elements of WIMPs, TUIs and ZUIs in a novel way. As a proof of concept, the proposed interface is implemented as the Bluebottle user interface. Implementation details can be found in the later chapters of the text.

Overview: Section 2.1 introduces the very basic concept of a user interface. Section 2.2 recalls properties of the *Command Line Interface*. Section 2.3 gives a short introduction to PARC user interfaces and compares them to the *CLI*. Section 2.4 introduces textual user interfaces and contrasts them with *CLI* and *PUI* interfaces. Because zooming user interfaces are not very well known, section 2.5 gives a more detailed introduction to the concepts and a short historical review. 2.5.1 describes recent and important work in the ZUI area. 2.5.2 describes a number of advantages of ZUIs over PUI systems. Section 2.5.3 lists and discusses a number of problems that are introduced by ZUIs. Section 2.5.4 presents conclusions drawn from the ZUI discussion. Finally section 2.6 presents a new conceptual model that synergetically integrates elements from TUIs, PUIs and ZUIs into a new general purpose computer user interface, alleviating many of the problems introduced in traditional zooming user interfaces.

2.1 Introduction

A *user interface* is the part of a system or tool that is exposed to the user. The first user interfaces were developed with the first tools invented by human ancestors. While the user interface of a hand-axe is, in terms of complexity, not really comparable to user interfaces seen in daily life of the twenty first century, it can be argued that it much better matches to its intended use than most of the user interfaces developed in the last few years.
The main reason for the increased complexity of current user interfaces is found in the number of tasks performed with a single tool. It is the task of a user interface designer to develop, with the available resources and limiting restrictions, a user interface that lets the user master the task-given complexity as adequately and efficiently as possible.
A system to play music may serve as an illustrating example: given no further restrictions, requirements and information, the designer might decide to put each piece of music, together with the player hardware, into a box in the form of a book. Opening the music-book would start playing the associated piece. This user interface is simple and may be appropriate in some contexts, maybe for pre-school education. If the number of pieces in a collection gets too large, the simple physical selection scheme becomes inefficient or infeasible by its own design that is limited by the physical access distance. All additional constraints and requirements lead to more versatile but also more complex systems, unifying the tasks of finding, selecting and playing of pieces of music into one tool.
When developing a user interface, the designer should be aware of all the requirements and restrictions of the given task. To get the best interface for a given task, it is normally necessary to integrate the designated users into the development process and to do several iterations on the fundamental concepts and implementation of the user interface. Detailed information about usability engineering and user interface development can be found for example in [83] and [92].
The main task performed with a general purpose user interface is managing task specific programs, called *application programs*. This includes starting of and navigation between different application programs. The general purpose user interface of a system strongly influences the possible interoperation between different application programs. Good interoperability between different application programs is desirable for efficient reuse of processed data. The worst possible integration is if the user has to manually type the output of application *A* as the input of application *B* possibly requiring data to be reformatted and converted. The best integration is if the user does not even notice the difference between the application programs.

The following sections introduce and discuss several different kinds of general purpose user interfaces.

2.2 Command Line Interface

Command Line Interfaces (*CLI*s) are used to send textual commands to the system.
The probably best known representatives of CLIs are the *UNIX* shells such as *Bsh*, *Csh*, *Ksh*

and about a dozen more. The main task of a shell is to offer a line editor where the user can type and modify the command line that is then interpreted according to the rules of the command line interpreter when the user presses a special *enter* key. In the simplest case, the content of the command line contains a command that is optionally followed by parameters. Many modern shells also offer chaining or piping of commands and input features such as automatic completion of file names.

In current *shells*, most commands operate on text input and output. Operating on text is very flexible but has two main weaknesses:

- The output of a command needs to be cut and parsed to get the desired pieces of information that then have to be formatted as the input for the next command. This transformation process is arduous, heavily version-dependent and error-prone.

- Text is not type-safe. The compatibility of one command's output to the next command's input cannot be checked by the command shell. It is sometimes, but not always, possible for a command to recognise bad input. An example of bad input that remained unrecognised by a follow-up process is the diagnostic message of the inertial reference system of the Ariane 5 rocket that caused an overflow exception. The diagnostic information was taken by the on board computer as input for flight control calculations resulting in a rapid change of attitude that caused the rocket to disintegrate and triggered a controlled destruction [26].

The Microsoft's *Monad shell* [73] from Microsoft avoids these two problems with the introduction of object streams that can be piped between commands. Sending typed object streams between commands becomes possible with the .Net framework and its reflection support.
From a usability point of view, the main drawbacks of the CLI are the *Invisibility of Commands* and the *limited interactivity*. While it is fast to type a well remembered short command, it is very difficult to find what commands are available or applicable in a given situation. The problem is especially hard for beginners as they have to learn a new 'shorthand' before they can do anything. It is normally aggravated by cryptic, inconsistent and inappropriate command names and parameters that are chosen to be short instead of meaningful. In CLIs, commands can be parameterised and started but normally not easily controlled at runtime. What is an advantage for scripting is a disadvantage for interactive applications. Especially in the graphical domains, such as CAD systems, interactive editing is useful and efficient.
Command line interfaces are best used for non-interactive automated tasks such as batch processing of data or for controlling and configuring non-interactive systems such as web servers.

2.3 PUIs - PARC User Interface

So called *PARC user interfaces* are the most widespread graphical user interfaces, named after the user interface of the *Alto* that has been developed at Xerox PARC in 1973 and influenced the GUI design of most modern systems. The *Alto* user interface introduced windows, icons, menus and pointers that are nowadays pervasive in general purpose computer systems. PUIs

are often referenced as *Desktop User Interfaces* because of the desktop metaphor that is implemented or as *WIMP*s as an acronym for windows, icons, menus and pointers. The *Macintosh* and *Windows* GUIs are the best known representatives of the PUI interface category. Because of their pervasiveness, the term *GUI* is often used as a synonym for *PUI* in daily usage and sometimes in the literature, too. In this discussion the term *GUI* is used to refer to the superset of all graphical user interfaces.
The main advantage of PUIs over the older command line interfaces is that commands and options are self-revealing because they are visible in the form of buttons, menus-items and icons. The user is always reminded of the operations that are possible on a selected object. In CLIs in contrast, all commands and parameters must be remembered or looked up before use. A drawback of menu commands is that they are less parameterisable and therefore less flexible than CLI commands. Scripting in PUIs is, compared to the scripting abilities of CLIs, almost non-existent.
Most PUI elements are based on a metaphor that maps the interactions with the computer to interactions with earlier known technologies. In fact it maps the interaction with the computer to the work in an office of the 1970s. Documents are stored in folders and deleted by putting them into a trash can. While the desktop metaphor was a great help for the first generation of computer users in offices, new features that are unknown in the domain of the metaphor are hard to integrate. For example, in typewriters, there was no feature like selecting a text and changing the font size or style. The typewriter metaphor is therefore not appropriate for modern text editors. About thirty years after the first introduction of the PUI the new generations of users expect personal computers on each office desk which renders the 1970s desktop metaphor, based on a pre-computer-area office, more and more inappropriate and limiting. In ”The Anti-Mac Interface” [37], D. Gentner and J. Nielsen point out a number of weaknesses of the traditional desktop metaphor such as for example the ”single-trash-can” metaphor. In a ”desktop system” documents are deleted by putting them into a virtual trash from where they can be retrieved again until the trash is emptied. If the user empties the trash can to get more space on one disk, the trashed documents from all other disks are unnecessarily deleted, too because of the inappropriate metaphor. Some of the improvements suggested in the paper are starting to find their way into the latest incarnations of Windows and MacOS.
A big problem of GUIs in general and PUIs in particular is the competition for screen space between visible interaction elements such as GUI widgets and the effective work data such as a texts or images. The competition for screen space is aggravated in smaller displays as used in PDAs or head mounted displays.

2.4 Textual User Interfaces

In *textual user interfaces*, so called *TUI*s, commands can be placed within any text and be invoked by pointing with the mouse cursor at it and pressing a special key or mouse button. This paradigm has several advantages over CLIs:

Visibility The commands and their parameters can be placed into *tool texts* prepared for a specific task or set of tasks. The commands are visible in the text and ready to be invoked by the user.

Readability There is no need for commands in *TUI*s to be short and cryptic since the commands are normally not typed very often. Normally, commands are typed only once for a specific task. If there is a chance to need the commands again for the same or a similar task, the respective text can simply be stored as a tool text for reuse.
In TUIs, scripting and command piping is normally not as sophisticatedly supported as in CLIs even though there are no conceptual obstacles. TUIs are more flexible than PUIs since the visible commands can easily be changed and parameterised simply by editing them. Usually TUIs are less interactive than PUIs.

An example for a TUI is the *Oberon System* as described in [117] and shortly discussed from an HCI point of view in [92]. While the original Oberon system uses a tiled display of texts it is also possible to integrate the essence of a textual user interface into a windowing system as shown in [120].

2.5 Zooming User Interfaces

A zooming user interface *ZUI* is a special kind of a *GUI* where panning and zooming are the only primitives to navigate between and within documents. *ZUI*s are sometimes referenced as *Multi-Scale User Interface*s since objects can display different levels of detail in different scales.
One of the basic ideas behind the zooming user interface is the *Spatial Data-Management* that was developed in the 1970s [23] [14]. In spatial data management systems (SDMS), large data-sets are presented in a graphical form on a screen. The user can pan and zoom on the data surface. When zooming in, the SDMS can present more and more details about the data items in view, when zooming out, the data can be abstracted.
Zooming as a concept was in use within specific application domains such as CAD and data visualisation long before it was discussed as a general user interface concept. In the 1990s the zooming subject was taken up by K. Perlin [86] and B. Bederson [9] and the idea of a more general zoomable user interface was discussed and implemented (see 2.5.1 for details).
In *PUI*s, documents are stored in virtual folders, often represented by icons. To view or edit the content of a document, the user clicks the icon that represents the document. The document is then loaded and displayed in an application program. After use, the document is closed again. In ZUIs on the other hand, all the documents are ”open” all the time, ready to be edited, lying on top of an unlimited virtual desktop. All the user has to do in order to view a document, is to find it on the desktop and zoom in until it becomes big enough for the intended use. If zoomed far out, a document can look like a single dot on the screen that becomes a thumbnail representation and finally the editable document, while zooming in and in. If the document is too big to fit on the screen, the user can use the same panning primitive that was used to

locate the document on the desktop. Hence no scrollbars are needed [92]. Figure 2.1 shows the conceptual difference of navigating in a text between a PUI and a zooming user interface. Combining inter- and intra-document navigation leads to a number of conceptual and usability problems that are described in detail in 2.5.3.
Most ZUIs offer an intuitive transition from a global overview to the smallest detail of an object or document. Zooming user interfaces in the wider sense have existed for a long time outside the HCI domain. In geographic atlases for example the earth is presented in maps of different zooming levels, starting with the entire planet over the continents to the countries and sometimes even provinces, that are distributed over the pages of the atlas.

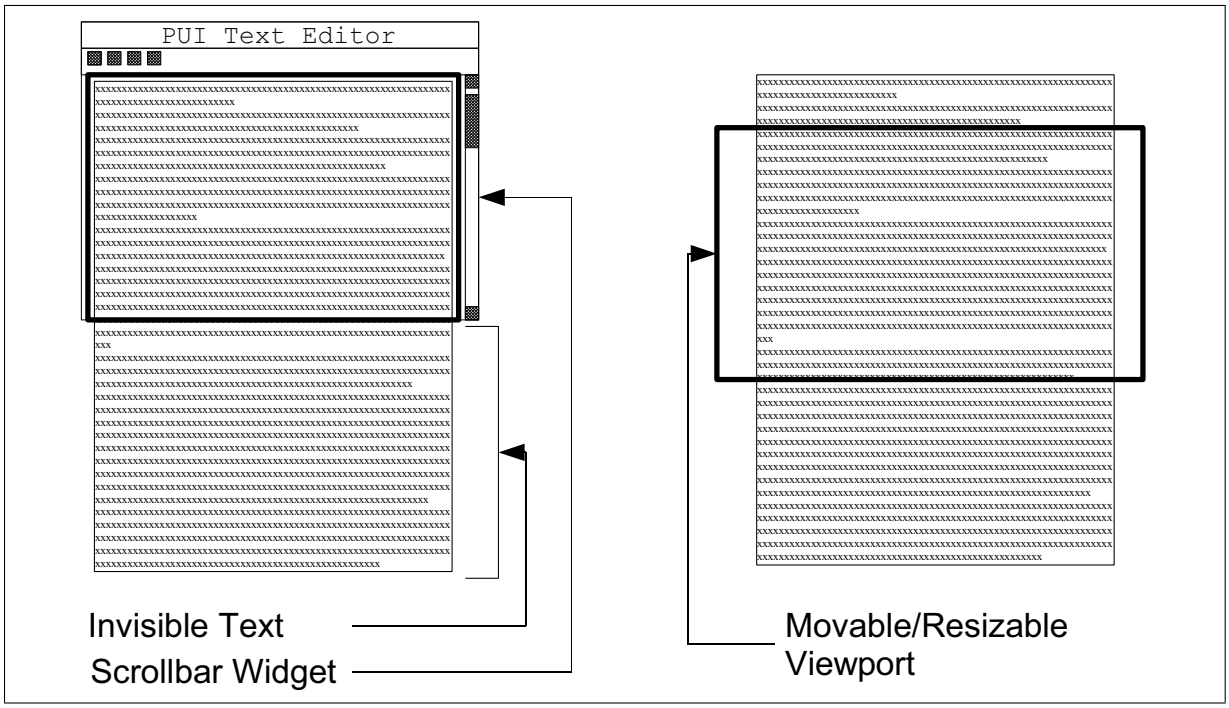

Figure 2.1: Navigation in a Text. Left: in a PUI, Right: in a ZUI

2.5.1 Related Work

There are several recent usability studies and example implementations of zoomable user interfaces. The following gives a short overview of selected implementations.

Pad K. Perlin and D. Fox from New York University defined and implemented a zoomable user interface concept called *Pad*. It was written in C++ and Scheme, running on top of SunOS, AIX, Linux and MS-DOS. ”Pad, An Alternative Approach to the Computer Interface” was published in a SIGGRAPH paper from 1993 [86]. Pad as the first workable implementation of a zoomable user interface running on commodity hardware had quite a big influence on a number of follow-up systems.
One of the main problems with *Pad* was the rough animation and jumpy zooming during navigation [9].

Pad++ B. Bederson and J. Hollan from Bellcore, and later the University of New Mexico, implemented a new version of *Pad* called *Pad++* in collaboration with J. Meyer and K. Perlin from New York University. There are many publications about *Pad++* [8] [9] [10] give a good overview of the implementation. *Pad++* compared to *Pad* had a significantly improved interaction speed. The smoother zooming improved the sense of relationship between the data in focus and the rest of the data space. The smooth zooming was achieved with an efficient implementation in C++ on Silicon Graphics computers. The main algorithmic optimisations compared to *Pad* can be found in the following:

- spatial indexing: the bounding boxes of objects are grouped into a hierarchy
- spatial level-of-detail (LOD) optimisation: only the visible details are rendered
- clipping: only the visible part of an object is rendered
- refinement: reduce the rendering resolution while navigating, refine when still
- adaptive render scheduling: the zooming rate is kept constant under varying frame rates

Pad++ supported a standard set of widgets to enter text, lines etc. and a set of operators to perform grouping, deleting, copying and more.

Jazz is a toolkit for developing ZUI applications in Java created by B. Bederson and his team from the University of Maryland [11]. The toolkit is built around a hierarchical 2d scene graph that is structurally similar to the better known 3d scene graphs. Analogously to 3d scene graphs, hierarchical groups of objects are inherently supported and affine transformations (translation, scale, rotation and shear) can be applied to nodes. The *Jazz* scene graph also supports layers, internal cameras, lenses and semantic zooming. The scene graph factors out transformations and rendering complexity from the ZUI toolkit and therefore makes it easier to write new widgets. It also facilitates improvements or changes in the rendering system while the rest of the toolkit remains unchanged. Some of the drawbacks of the scene graph approach are the memory footprint, the execution efficiency and the inherently imposed restrictions.
The *Jazz* framework can be integrated into Java Swing so that it can be used as a part of a conventional Swing application. It is also possible to integrate Java Swing components into the Jazz framework so that existing widgets can be reused.
Due to its general approach, it grew rather big was therefore superseded by *Piccolo*.

Piccolo is the latest incarnation of *Pad*/*Pad++*/*Jazz* developed by B Bederson and his team at the University of Maryland. It is a toolkit that supports the development of structured 2D graphics with support for zooming. It maintains a hierarchical structure of objects and cameras that makes it possible for the application programmer to operate on a higher level on the visual object without worrying too much about low level implementation details. *Piccolo* has been implemented in Java and then ported to the *.NET-Framework*. While *Piccolo* supports most of the features that have been found useful in *Jazz*, it no longer supports the integration of *Swing*

widgets.
The *Piccolo* framework is described and compared to the *Jazz* framework in "Toolkit Design for Interactive Structured Graphics" published in IEEE Transactions in Software Engineering 2004 [12].

Zomit is a generic zooming API written in Java that works together with a ZoomMap server written in C++. Zomit was developed as a framework for applications in bio-informatics. Biological genome databases have become so large that scientists can no longer interact directly on the available data with conventional methods. Zooming was introduced as an intuitive way to browse these large datasets [89]. Due to implementation limitations and slow hardware, zooming only worked in discrete steps. Portals, magic lenses [106] and semantic zooming are supported.

THE - Flash ZUI Demo J. Raskin specified a system, called *THE, The Human Environment* [92] that is supposed to be as easy or easier to learn than a traditional GUI system and as fast and flexible as a command line system. The main navigational concepts are zooming and panning. To promote the proposed concepts, an interactive demo has been developed in Macromedia *Flash*. Unfortunately it was not possible to implement the interactions as they are described in the *THE* specification [93] because of limitations of the *Flash* environment. The implementation lets the user experience navigation in a ZUI, including losing the context, the orientation and the focus as described in more detail in 2.5.3.

2.5.2 Advantages of ZUIs

This section mentions a number of advantages of ZUIs over more traditional graphical user interfaces:

Spatial navigation The spatial navigation in ZUIs matches well with the human spatial memory. Users can find documents again where they have placed them before, especially if they group related documents locally.

Natural hierarchy Hierarchical data-structures can be naturally mapped to objects containing other objects.

High information density ZUIs can have a very high information density.

Finding by pattern Documents and objects on the virtual desktop can be visually recognised and found by their form.

Overview when zoomed out When zoomed out, the user can see the context of a document or object on the virtual desktop. This requires a reasonable manual or semi-automatic alignment of the objects on the desktop.

Details when zoomed in When zoomed in, the documents or objects can present more information or be displayed bigger so that they become more readable.

Screen space usage The screen space can potentially be more efficiently used.

2.5.3 Problems of ZUIs

There are several major and minor problems with zooming user interfaces that are mentioned briefly in this section and discussed together with proposed solutions.

Lack of Context The *lack of context* problem describes the situation where a user has zoomed into the details of an object and then loses the context of the outer object or higher hierarchy levels.
When zoomed into details, the relationship between the visible details and the higher hierarchy levels is not apparent. Therefore the mental position within the information space gets lost and users have difficulties in locating the required information or objects. In such situations, a planar tree representation can be much better because tree nodes of higher hierarchy levels remain visible and give context information.
There are several proposed solutions to the problem. Furnas [35] proposed a fish eye lens which has a variable zoom factor, so that the context is not completely invisible. Pook [90] proposed a context layer, hierarchy layer or hierarchy tree that can be implemented as translucent overlay over the focus. A further possibility, implemented in an early version of the Bluebottle GUI, is a fast ”motor” zooming function where pressing a key zooms out so the context becomes visible. When the key is released, the zoom returns to the factor it was before. This functionality has been replaced with the possibilty to use the scroll-wheel of the mouse to zoom smoothly in and out, putting the navigation more direct into the hands of the user.
Another solution is an additional navigation window in a different zoom level. This solution can often be found in CAD systems and also in programs that are not inherently graphics oriented such as for example music notation systems [104].

Tunnel Vision The problem known as *tunnel vision* is very similar to the *lack of context* problem. It describes the conceptual impossibility of having more than one documents or information sources visible on the screen at the same time unless they are by chance placed next to each other and in the same zooming level. Proposed remedies for this problem include:

- sticky documents that virtually stick to the viewport during zooming and panning [11].
- additional viewports, sometimes referenced as *cameras* or *portals*, that can show a different region of the workspace in a different zooming level [86].
- links or a map overview that allow fast switching between two different views [90].

Loss of Orientation Due to the combined primitives for navigation between and within documents, users lose the spatial relationship between different documents while zooming and panning within a single task related document. Figure 2.2 gives a simple example of this problem. The user is working on the right text, using the documents to the left as information

sources or notes. While working and editing in the right text, the viewport has to follow the text cursor where the user is working. Due to these task oriented movements of the viewport within the document, the perceived spatial relationship between the work and the information source is broken.
This is a conceptual problem introduced with the combination of the navigation primitives for inter- and intra- document navigation. It can be reduced but not completely solved without breaking one of the fundamental concepts of ZUIs.

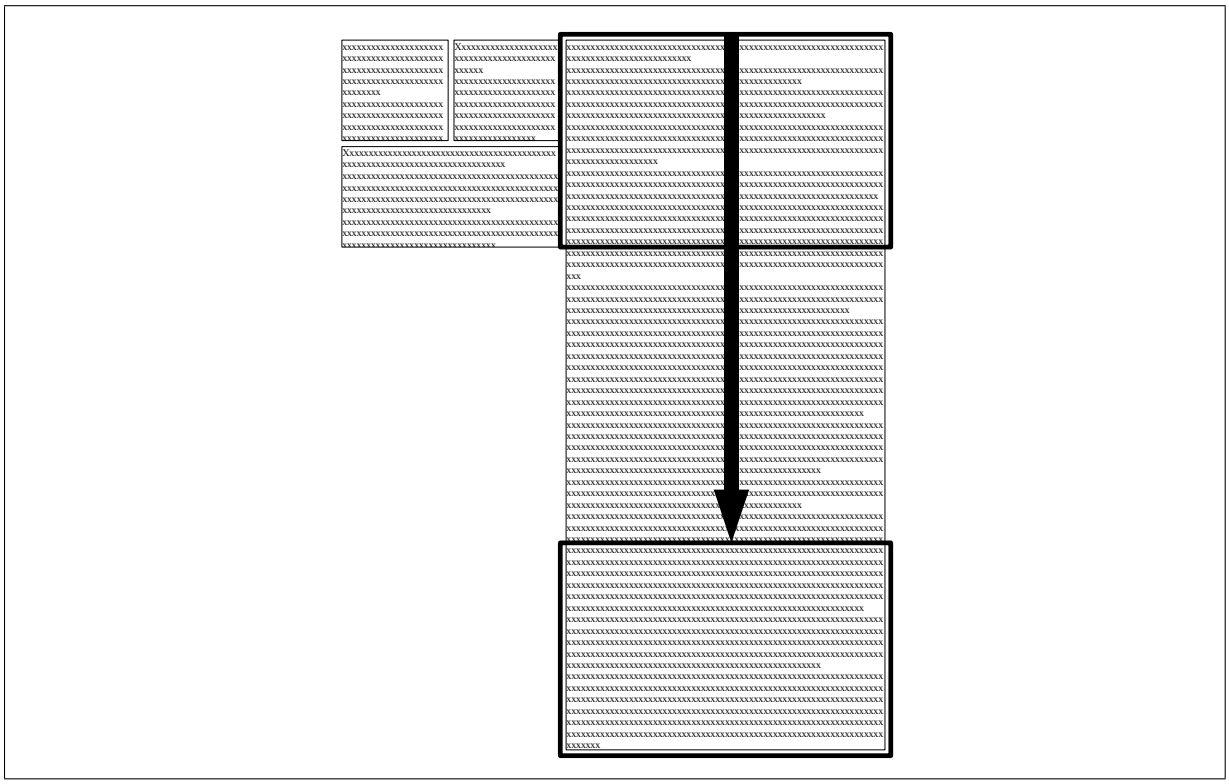

Figure 2.2: Example of the Loss of Orientation Problem

Desktop Consistency An important concept of ZUIs is that navigation *within* a document is the same as *between* documents. This requires that the entire document is visible within a range of the virtual desktop. This has a number of undesirable consequences when the content of a document is extended. Figure 2.3 *A* shows a section of a desktop of a ZUI. It contains three text documents, a pie-chart and a notice, possibly of a collaborator, to have a look at the pie-chart. When the text highlighted with the bold border is extended, a space conflict happens. Apart from not accepting the text extension or overlapping, the system has mainly three possibilities to resolve the conflict. In figure *B*, the system resolves the conflict, extending the text length on the desktop, moving conflicting documents down. This solution has the drawback that the document arrangement on desktop is changed by the system which is bad for the user's spatial memory that then conflicts with the changed reality of the system. In the sample picture, the rearrangement destroys an important spatial relationship between the note and the pie-chart. To avoid this new problem, unbreakable spatial links could be added as a feature of the system,

which of course adds to the overall user interface complexity. Multiple spatial links could also lead to pathological situations involving cycles of links. In figures *C* and *D* the font size of the document is adapted so that the text fits again into the same space. In figure *D*, the document width is also adjusted to the new font size so that lines do not have to be broken at new places due to the new font size. The resizing solutions are OK for the spatial memory and also avoid realignment conflicts. The drawback is that the document is moved into another level of magnification. Different magnification levels of related documents are not desirable since the navigation between them then not only requires panning but also zooming.

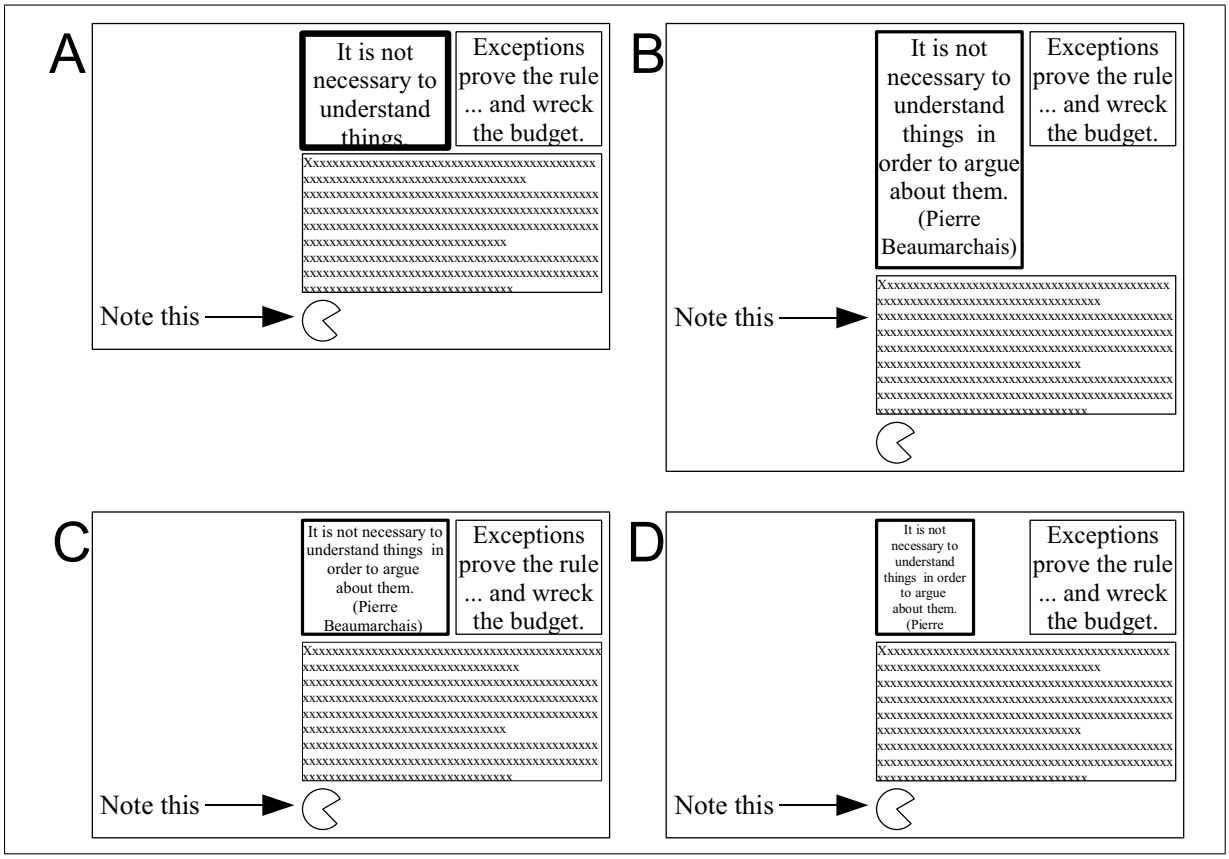

Figure 2.3: Conceptual Desktop Consistency Problem in ZUI

Lost in the Blue On January 1, 2003, the ETH Bluebottle system with the prototype of a zooming user interface was first released to a broader audience.[1] Three days after the publication, a member of the Oberon/Bluebottle community sent the following message to the Oberon mailing list : "*I love this new system. But I have a question. Is there a solution, now or later, to find windows again when, after some zoomings, we are lost in the blue.*" [69].
With this message he pointed out a typical but easily solvable problem with zoomable user interfaces, where the user navigates to a point where nothing but a homogeneous colour, the "blue space", is visible and it is not clear where to zoom or pan to. The problem can easily be

[1]Before this 2003 prototype, the Bluebottle GUI was implemented more like a regular *PUI* system that supported panning on a desktop twice as large as the screen.

fixed by providing a *home* key that moves the viewport either to a predefined position or zooms to an overview.

Implementation The implementation of a ZUI system that can handle thousands[2] of virtually open documents on the desktop requires complex level-of-detail (LOD) optimisations.

2.5.4 Conclusions About ZUIs

Zooming is a natural method to combine overviews with detail views. Using smooth visual transitions while zooming from an overview into the details helps users get a sense of relationship between the focus and its context.
It could for example be used in a area surveillance system where a guard needs to be informed about the location of a problem such as a fire alarm. The system can display the entire area with all the separate buildings using a mark on the building with the problem. The guard can then smoothly zoom in to get more details such as the floor and location without losing the context. Zooming can also be efficient in locating documents or objects in hierarchically ordered systems as long as the user knows the exact hierarchy. Many users could for example find their home on a zoomable world map because they know the spatial hierarchy of continent, country, district, city, quarter. Searching a place without this hierarchical knowledge becomes arduous.

Considering the problems of *tunnel vision*, *lack of context*, *loss of orientation* and the interaction overhead, zooming should not be applied as a magic bullet but only if it matches the problem space.

Especially when considering the use of zooming as the main navigation technique for a general purpose user interface, the *tunnel vision* and *lack of context* problems become critical since general purpose systems need to provide quick access to a number of different task specific contexts. Adding additional portals, overlays or hierarchy trees to the system as described in the literature, only complicates and dilutes the zooming concept.

2.6 The Bluebottle User Interface

This section introduces the Bluebottle user interface in the consequence of the discussion of general purpose user interface concepts in sections 2.2, 2.3, 2.4 and especially the conclusions about ZUIs in 2.5.4.
For the discussion the following terms are important:

Display Space The *display space* is the entire GUI coordinate space that is managed by the Bluebottle display space manager. The display space is populated by arbitrarily shaped display space objects. The display space is observed through viewports as defined below.

[2]Millions in case of library systems.

Figure 2.4 shows a display space with four rectangular objects and two specially shaped objects, as well as two viewports observing the display space. One of the observers is a local graphics adaptor, the other is a remote VNC display.

Display Space Objects a *display space object* is any object that is visible in the display space. While display space objects in general can be arbitrarily shaped and also translucent, an important subset is rectangular. In the following, the class of rectangular display space objects is called *window*s. *Window*s are mainly useful to present and manipulate texts, tables and graphics.

Viewports A *viewport* observes a certain region of the display space. It displays the display space objects in the observed region on an associated display such as a computer screen, a remote network frame buffer or a screen shot utility.

*Display space*s, *display space object*s and *viewport*s are discussed in technical detail in chapter 5.

In the following, section 2.6.1 introduces a task oriented zooming and navigation strategy that

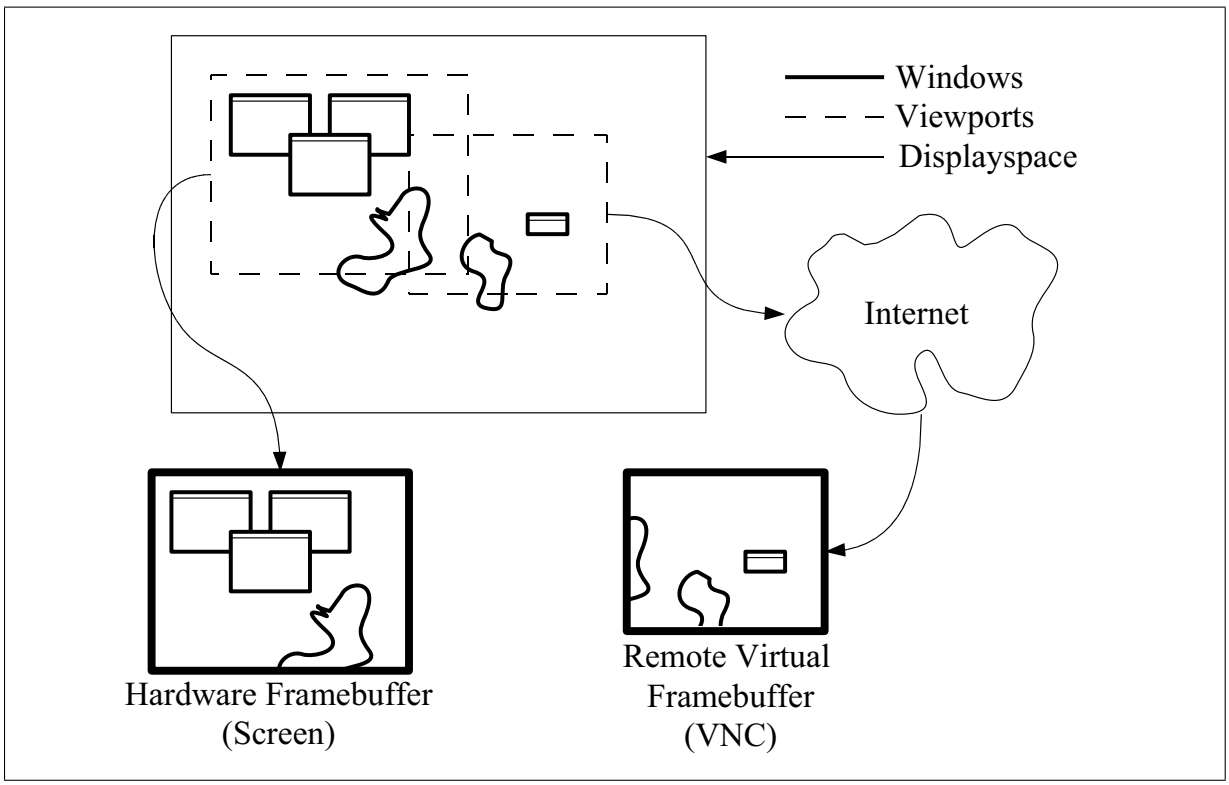

Figure 2.4: Display Space with Display Space Objects and Observing Viewports

avoids the problems of *tunnel vision* and *lack of context* on the system level. Section 2.6.2 packs the proposed zooming and navigation strategy into a metaphor.

Section 2.6.3 gives a short overview of the user interaction with the system.

Finally section 2.7 summarises the differences of the Bluebottle GUI compared with other systems.

2.6.1 Zooming Contexts as a Navigation Concept

We suggest using zooming, where appropriate, in a specialised task-oriented way, limited to a specific task-related display space object or task-window. Each task-window forms a natural *zooming context* that can be zoomed and panned independently of the rest of the virtual desktop. This means that zooming as well as panning within any *zooming context* does not affect any of the outer *zooming contexts*, whereas zooming and panning within an outer context affects all inner contexts. A *zooming context* acts like a microscope on a desktop. Whatever magnification the microscope is set to, it does not affect the distance of the observer to the desktop. The individual task-windows can be distributed in an unlimited display space forming the outer-most *zooming context*.
This approach has the main advantage that the spatial task-to-task relation on the desktop remains constant, unless intentionally changed by the user, and it is independent of the current task's context and task's magnification level.
Switching between different task-windows is an important feature of any general purpose graphical user interface. In traditional GUI systems, this is normally done with a task bar or hidden shortcut keys for task switching. In the proposed model, zooming and panning in the outer-most *zooming context* naturally models this operation.

2.6.2 The Instrumented Desktop Metaphor

This proposed model can be visualised with the following metaphor:
The display-space represents a large desktop. On the virtual desktop there are domain specific viewers. One viewer's domain can for example be texts. The text viewer can work like a book on the desktop or like an antique scroll which matches better with many page-less text viewers that use scrollbars to navigate within the text.
One artefact of real-world books or scrolls is that you can only zoom into it until it hits your nose[3] and it will not reveal much more information by doing so, and this is OK since all it needs to provide is readable characters. Another domain specific viewer could be a zoomable microscope. The ideal microscope allows the user to zoom into the observed object without limits, revealing more and more details. Zooming within the microscope neither changes the microscope's size nor does it change the size of the book lying next to it on the desktop. Also moving the object holder below the lens does not move the microscope or worse the entire desktop as it is the case in traditional ZUIs.

2.6.3 Principles of Interaction

This section describes the user interaction with the Bluebottle system, going into more details where it deviates from commonly known systems. The Bluebottle implementation on a standard personal computer uses the keyboard and mouse as the promary control device.
Following the proposed metaphor, the system offers a large virtual desktop for task specific display space objects. One of the most important activities of interaction is navigation.

[3]For hyperopic persons the useful zoom-in stops much earlier.

For moving or resizing of the viewport, the *meta* key on the keyboard is used.
When the *meta* key is pressed, the viewport follows the mouse cursor if it touches the edge of the screen (Fig. 2.5 *A*). To resize the visible area of the desktop, the user can simply turn the scroll wheel of the mouse (Fig. 2.5 *B*).

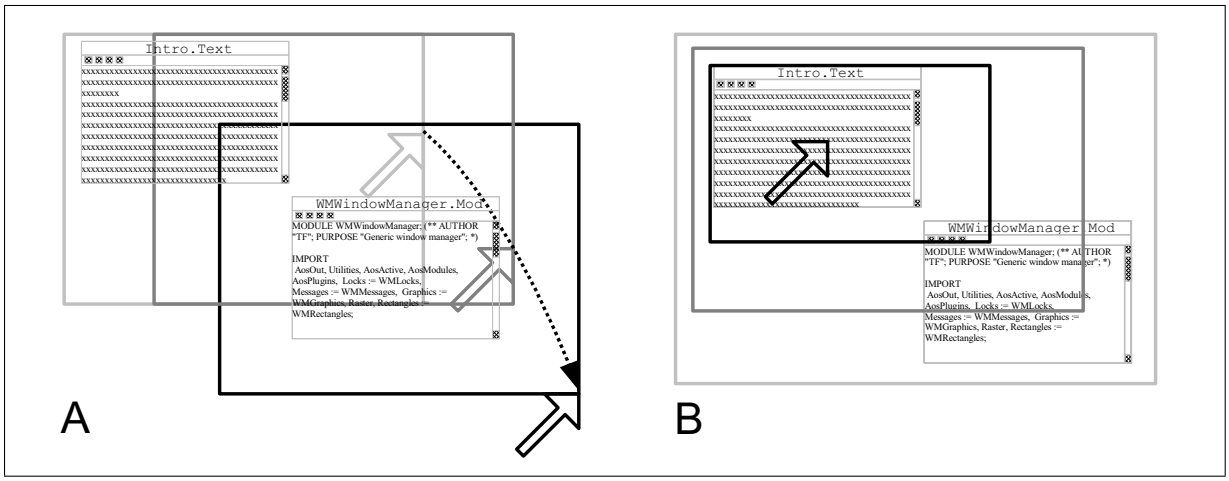

Figure 2.5: Panning and Zooming with the Mouse

Another possibility for this basic type of navigation is to use the keyboard. Pressing the direction keys on the keyboard while the *meta* key is held down moves the viewport by one screen size along the pressed direction. Figure 2.6 *A* shows the movement of the viewport when *meta-right* then *meta-down* keys are pressed. Pressing the *page-down* or *page-up* keys on the keyboard while the *meta* key is held down zooms in respectively out by factor of two, keeping the screen centre in position (Fig. 2.6 *B*).

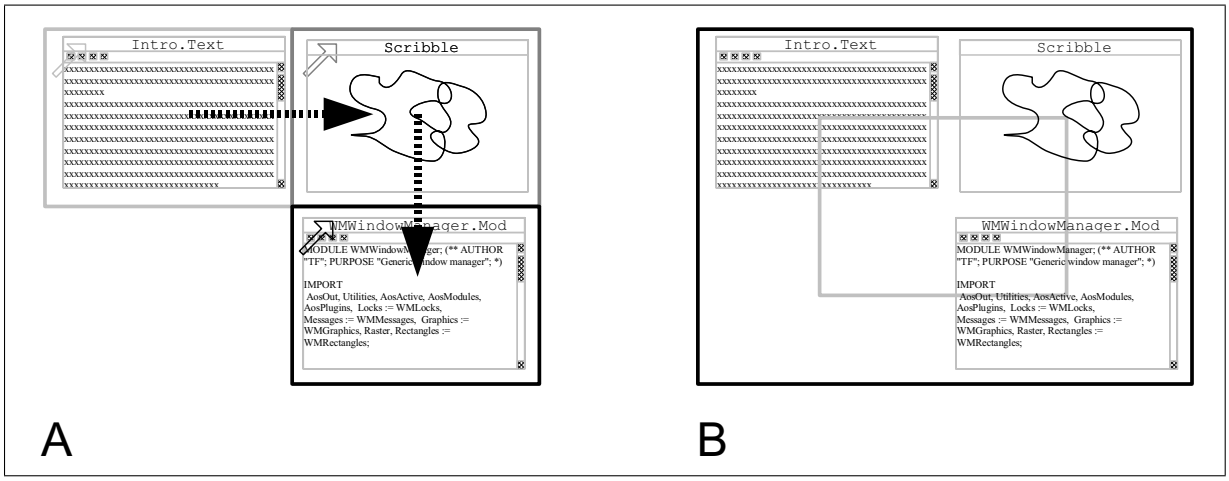

Figure 2.6: Panning and Zooming with the Keyboard

A more advanced navigation operation is to get an overview of the entire desktop. This can be done with the *meta-home* key combination. The viewport moves and zooms so that the used portion of the desktop becomes visible (Fig. 2.7). The user can now select the desktop object of interest by clicking it while holding the *meta* key pressed. The viewport moves so that the clicked object is visible top left. If the display object fits to the screen, the zoom factor is set

to 1 so that the selected display object is visible without scaling. Otherwise the zoom factor is adjusted so that the entire display object is visible. Pressing the *meta-end* key combination zooms to factor 1.

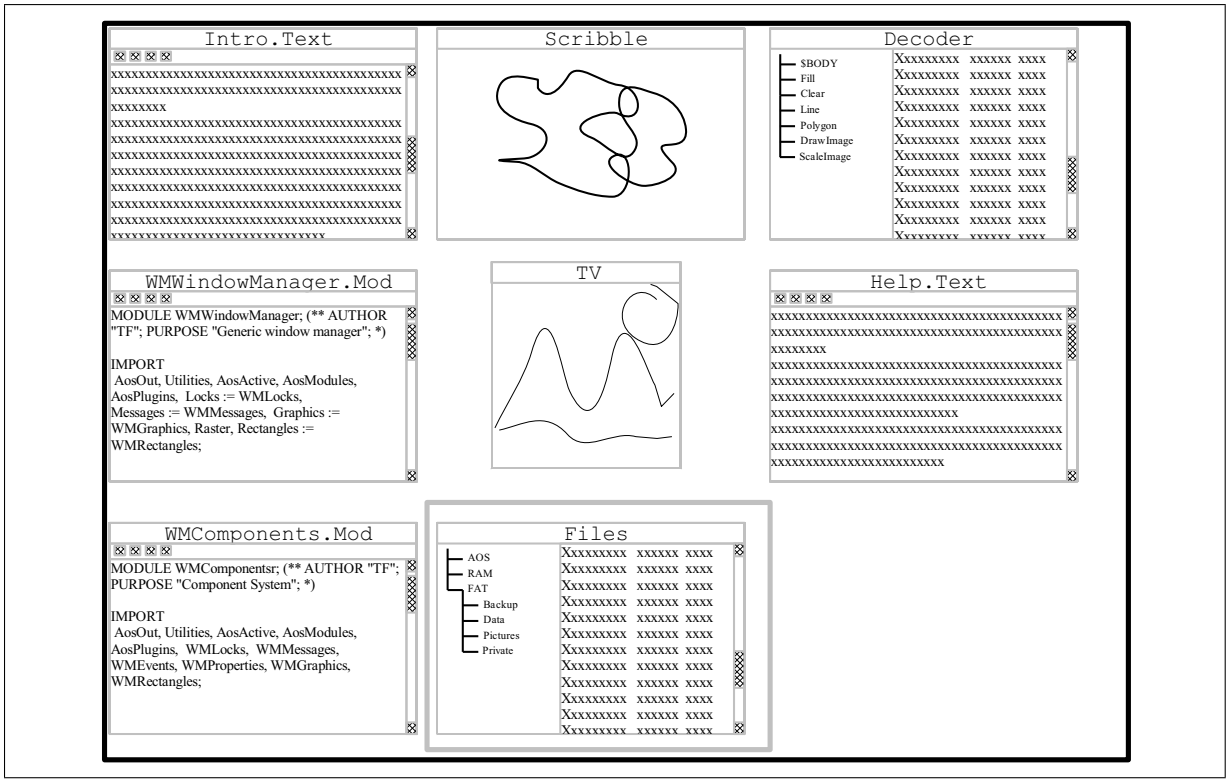

Figure 2.7: Zooming to an Overview

Windows as the rectangular subclass of the generic display space objects have a decorative frame that also serves as a handle for operations on the display object. The frame can be used to move and scale windows similar to well known PUIs. When being resized, windows either adjust their content to the new size or they are automatically scaled by the display space manager.

Even though the Bluebottle metaphor with its large desktop suggests to spread application windows instead of virtually stacking them on top of each other, it is still important for the work-flow to support overlapping. Overlapping is needed so that the user can move windows containing information to the place where the information is needed. The management of overlapping from the user perspective differs from other well known window managers. When clicking into the content of a window, it is activated but not moved to the front. This prevents hiding an information window that was possibly placed on top of the clicked window. To change the overlapping hierarchy the window's meta area, its border, needs to be used. When clicking into the border of an inactive window, the window is activated and moved to the front. To make it possible to move an active window without changing its overlapping, active windows require two clicks to be moved to the front.

At all positions on the virtual desktop it is possible for the user to summon the Bluebottle main menu that offers system-wide commands and tools. The main menu is called with the *meta-esc*

key combination.
The Bluebottle GUI offers many of the well known PUI widgets such as buttons and scroll-bars. Its text system allows commands and macros to be started from all text fields as in powerful TUI systems such as for example Oberon [117]. Commands written in any text can be started using a context sensitive menu that can be called with the right mouse button, or simply by clicking the command name in the text with the middle mouse button. All texts in the system are usable for copy and paste operations and are compatible with special text tools such as macro processors or style editors.
For better visual orientation and organisation it is possible to place different backdrops on the desktop that can serve to group tasks or projects. Apart from serving as decoration and visual clues, the backdrop images can be navigated to with a simple click, whenever they are visible.
The entire setup of the desktop with all the task-windows and backdrops can be saved persistently on demand, so that the user can continue work with the same desktop after restarting the computer. Saving the desktop differs from hibernation as known from other systems in that application programs are explicitly asked by the display space manager to store their relevant persistent data. This is much more flexible and robust than loading back an entire system memory image. It is for example possible to update the entire system or even change the hardware setup of the computer and still continue working with a desktop that was saved before the changes.

2.7 Summary

The Bluebottle user interface is not a zooming user interface in the classic sense as described in related literature (see 2.5.1). It extends the traditional PUI desktop with a microscope-like, smart zooming and panning functionality and a certain level of persistence. The use of task oriented *zooming contexts* avoids a number of problems of traditional ZUIs:

Lack of Context *Zooming context* containers such as task-windows in the outer-most *zooming context* naturally provide context information. The *lack of context* problem is thereby solved on the task level. For intra-task zooming, domain specific context providing containers can be introduced by the task specific application program.

Lost in the Blue Since the navigation within a *zooming context* does not affect the relative position of any outer or outside zooming contexts, the likelihood of getting lost because of navigation is greatly reduced. A special key combination can be used in the unlikely case of being lost, to smoothly ”motor” zoom into an overview display.

Tunnel vision The tunnel vision problem is tackled on the task level. Each task-window is on the same zooming level with the possibility of overlapping between the different task-windows. Callable translucent tool objects such as the main menu help avoiding unnecessary navigation operations.

The system-wide text system allows calling commands in context wherever they can be written. All texts are compatible and allow the use of macros and tools in a unified fashion.
Traditional PUI elements such as menus and buttons support situations where a pure textual user interface would be too limited.

3

Text System

Words are, of course,
the most powerful drug
used by mankind.

— Rudyard Kipling (1865 - 1936)

Despite all graphical symbols, animations and sound elements that can be found in modern general purpose user interfaces, texts play a fundamental role in most tasks users perform in such systems. There is no way or reason to change this; after all, texts are the most versatile way to represent stored information in a given human readable language. The most obvious examples for the use of texts are reading and writing emails, taking notes, reading web-pages, documentations, books and news. Writing computer programs and designing computer chips are other less obvious text oriented tasks, even if these texts represent highly structured artificial languages.
Therefore, Bluebottle takes particular care of the textsystem and implements an unusually generic text model as a system data structure that is potentially able to handle text in any language of the world. The tight integration of the textsystem as a system data structure has proven its worth in the Oberon [117] system and was hence absorbed into Bluebottle.
The following section 3.1 introduces and defines important terms for the discussion of the text system. Section 3.2 discusses the integration of text systems in general purpose computer systems. In later sections the design of the Bluebottle text system is described first in concepts then in technical details. Then a section introduces the concepts of the Bluebottle text editor components, followed by technical details and the conclusion.

3.1 Terms

When talking about texts, there are several terms that must be defined clearly and distinguished. This section gives concise definitions and short introductions to the most important terms used in the following discussion.

Character A character is the smallest unit of a written language. Examples of characters are letters, symbols, digits and ideographs. A visual representation of a character is called a *glyph*, it depends on the context consisting of font, style, size and surrounding characters in a text. The concepts of *character* and *glyph* should be strictly separated.

Glyph A glyph is a visual representation of a character. It is not part of a character set. For each character, there are different visual representations (glyphs) depending on font, size, style, text layout and surrounding characters. While font, style and size obviously influence the visual representation of a character, the influence of text layout and surrounding characters may be less clear. An example for the influence of the text layout to the character representation is the punctuation in Chinese, Japanese and Korean (CJK) that is different for horizontal or vertical layouts. Examples for the influence of surrounding characters are Arabic characters or ligations like the diphthong AE that is often written as Æ.

Font A font is a collection of glyphs that usually match in size and style. It is a mapping of characters to glyphs, optionally taking into account the surrounding characters and text layout. A font should not be confused with a character set, even though most fonts are tuned for one character set in implementing glyphs for all its characters and often internally storing glyphs in the order of the corresponding codepoints.

Character Set A character set is a mapping from numbers (integer values) to characters. Each integer value is called a *codepoint*. Codepoints can be unassigned, meaning the respective integer value does not map to a character. There are many different character sets that are more or less commonly used. *ASCII* is probably the best known character set. It is defined over the range of 7 bit unsigned integers. The integer value 65 as an example corresponds to the character *latin capital A*. Another character set widely used in Europe is the *ISO-8859-1* (Latin1)[53] character set, that is defined over the range of 8 bit unsigned integers and extends the ASCII character set with a number of accentuated characters used in west European languages[1]. The character set does *not* define visual representations of the contained characters. The visual representations of characters are defined in fonts.

Character Encoding Schemes A character encoding scheme is a mapping between characters and sequences of bytes. With a given character set, each character of a text is mapped to its codepoint. The number of the codepoint is then stored as a sequence of one or more bytes. A *fixed size encoding scheme* uses a fixed number of bytes to store each character. This has the advantage that indexed access to the single encoded characters is easily possible. The drawback is that all the characters take up the space needed to represent the codepoint with the highest number. ASCII, UCS-2 and UTF-32 are examples of fixed size encoding schemes. A *variable*

[1]The Latin1 extension of the ASCII set covers the following languages: Albanian, Basque, Catalan, Danish(missing IJ), Dutch, English, Faroese, Finnish, French(missing Œ), German(missing typographic quotation marks), Icelandic, Irish, Italian, Norwegian, Portuguese, Rhaeto-Romanic, Scottish, Spanish, Swedish.
By a happy coincidence, it also fits for Afrikaans and Swahili.

size encoding scheme stores the codepoint numbers in a variable number of bytes. This has the advantage of saving bytes if shorter encodings can be assigned to the more frequent characters. The obvious drawback is that no direct indexed access to the single encoded characters is possible. UTF-7, UTF-8 and UTF-16 are examples of variable sized encodings.
UTF-8 is used to encode characters of the character set specified by Unicode and ISO/IEC 10646. The UTF-8 variable size encoding has been cleverly engineered so that all ASCII codepoints are represented with the same single bytes as defined in the ASCII standard. All other characters of the significantly larger Unicode/ISO 10646 character set are encoded in more than one byte. A big advantage of UTF-8 over UCS-2 or UTF-32 is its byte order independence.

Text A text is an ordered list of characters. An abstract text is conceptually independent of character set and encoding. It only deals with characters. A text that does not contain additional information is called a *plain text*. If the text contains additional information about its visual structure, for example line-breaks, colours, styles or the fonts to be used for display, it is called a *rich text*.

Editor A *text editor* is a tool that allows the controlled manipulation of text. There are several different kinds of text-editors:

- *line editors*, where the text is manipulated on a per line basis. Line editors are simple programs that can be useful in special situations where either the device that is running the editor is limited in display or memory, or when the text to edit has a special line oriented structure, like some configuration files that are not based on XML.

- *full screen editors*, where the text manipulation is not restricted to a selected line. Full screen editors are useful for most applications that do not require rich text or complex layouts. The implementation of a full text editor is slightly more complicated than the implementation of a line editor.

- *rich text editors*, where the text manipulation supports colours, fonts, vertical offsets or similar attributes. Rich text editors are useful for applications where the document should be able to adapt to different screen layouts such as web pages. The implementation of a rich text editor is much more demanding than the implementation of a full text editor. The system requirements are also massively higher because of the more complex layouting process and the mapping between screen positions and text positions that needs to take formatting into account. The text model needs to support and store formatting informations, too.

- *WYSIWYG* (*W*hat *Y*ou *S*ee *I*s *W*hat *Y*ou *G*et) editors show the manipulated text and often graphical elements exactly like they will look in the end result, for example on a screen for an on-line presentation or a page of paper. WYSIWYG editors are mainly used to create documents for publishing and presentation where a specific layout is needed. In the view of on-line publications for a large number of different screen sizes and resolutions, strongly defined layouts are a disadvantage because they cannot be adapted to the

pecularities of a specific screen. For example a portrait layouted document does not fit well on a landscape orinted screen. For artistic and sometimes legal reasons, completely preformatted pages are sometimes preferred.

Input Methods *Input methods* are tools that map input, usually from a keyboard, to characters that are inserted into a text. *Input methods* are often referred to as *Input Method Editors* (*IME*s) or *front-end processors* (*FEP*s)[64]. They are mainly used for writing character in languages where the different characters cannot be mapped directly to the keys on an ordinary keyboard. IMEs usually combine multiple key-strokes, special key combinations and sometimes GUI widgets for the user to select the desired characters. IMEs can be used to input complex characters and symbols for example for writing in Chinese, Japanese or Korean (CJK). A large number of different input methods exist for inputing CJK characters[2]. The CJK IMEs can be grouped into several categories:

- *Input by structure*. Chinese characters (Chinese *Hanzi*, Japanese *Kanji* and Korean *Hanja*) are composed of *radical* or *radical-like* elements. In dictionaries, the characters are normally categorised by the initial-radical[3] and inside the categories ordered by the total stroke count. Other structural selection criteria are stroke shapes or three or four of the corners of a character. Some structural input methods allow the combination of more than one of these selection criteria.
  The input method editor normally presents the characters that match the given criteria and the user selects with the mouse, number or cursor keys.
  Relying on the structure rather than the pronunciations of a character makes it possible to input characters where the pronunciations is not known to the user. Another big advantage of structural input methods over methods based on transliteration is their independence of dialects.
  Wubi (short for Wubizixing) is a fast structural input method that is popular in mainland China. All characters can be written with five or less keystrokes. Expert writers can write up to 160 characters per minute.
  Cangjie is another structural input method, that is used mainly in Taiwan.

- *Input by encoding*. Input by encoding is only used in very rare occasions, if all other input methods fail. Modern encoding input methods present the entire list of characters in a huge table where the user can select the character. In older systems, the characters had to be searched in a printed table and the respective code typed.

- *Input by association*. Some input methods uniquely map (an important subset of) the characters to pairs of keystrokes. Two keystrokes result in a defined character. No can-

[2]Windows XP for example is shipped with 8 IMEs for simplified Chinese, 10 IMEs for traditional Chinese, an IME for Japanese with support for *Hiragana*, *Katakana* and *Kanji* a Korean IME with support for *Hangul* and *Hanja*. Many of the IMEs in Window XP support different input modes, so they could be counted as more than one IME.

[3]Depending on the dictionary there are about 186 categories for simplified respectively 214 categories for traditional Chinese

didate selection is needed. This kind of input requires a lot of training but it can be very fast.

- *Input by transliteration*. Transliteration based input methods are the easiest to learn since they work with the pronunciations of characters. No special tables need to be learned. The most popular input method in mainland China is the *pinyin input method* that relies on the standard romanisation of Mandarin Chinese.
  A big problem with transliteration based methods is the ambiguity. In Chinese, a large number of characters have the same pronunciations. *Tones* can be used for disambiguating but the remaining number of candidate characters can still be very large. Sorting the characters according to usage frequency helps finding the intended characters. While simple systems operate with single characters, modern IMEs look at character compounds or even phrases and use statistical and linguistic methods to present candidate characters in a better sortation [118] [20]. The best matching character is often automatically inserted into the text. If the user disagrees with the automatic choice and changes a character, some systems can update their data bases so that the new character usage is considered in similar situations.

IMEs are also used for the input of Russian or Greek characters. In these cases, the IME works similar to a keyboard driver, directly mapping keystrokes to characters. Enabling and disabling the IME makes it possible to quickly change between input methods for different character sets. For example, disabling the IME is used to type email addresses, URLs or CLI commands. Experienced writers often enable, disable and change IMEs in rapid succession to write words in different languages or scripts.

3.2 Integration

A unified system-wide text subsystem like it was implemented in Oberon [117] offers many advantages over specialised per application solutions:

Compatibility With a unified text system, all texts in the system are inherently compatible, meaning texts can be shared between different tools, commands and procedures without the danger of loss of information due to different internal representations of text and attributes. Text exchange between different character sets as it happens between applications with different text representations can in general not be done without loss of information. This information loss is obvious in the cases where the different character sets involved are targeted at different languages with different characters. It is less obvious in cases where both encodings are designed for the same language. In the case of Chinese for example, there are several different character-set encodings that do not contain the same set of characters. Because of the huge number of characters, not all the character set defining committees came to the same conclusions about which characters to include and which to leave out. In some cases there is also a dispute between linguists about what should be considered a character on its own and what just as a differently written form of another character.

Tools A system-wide text system allows each application program to take advantage of tools written for the text system. Importer and exporter procedures that are used for the text exchange between different computer systems can be written once and then be used throughout the system. This of course requires a system managed plug-in structure for importer and exporter procedures. Apart from the import and export, other system-wide text manipulation tools can be of great use. Such a text manipulation tool could be as simple as a procedure that changes all the letters in a selected text to lower case. More compelling ideas are translation aids, spell checkers, macros, thesauri and acoustic text readers that can help physically disabled persons.

Code Reuse A unified system-wide text system allows the reuse of the sometimes quite complex implementation of an efficient text system. The reuse of the code reduces the complexity of application programs and therefore helps reducing the total numbers of errors in the system.

In current operating systems, the integration of text gets better and better but it is still far from perfect. Problems occur for example with application programs that assume the system to use special character sets or code-pages. Microsoft Windows XP for example uses a system wide clipboard mechanism that allows text access in different encodings to deal with legacy applications [74] . One application may put an 8 bit text in the OEM character set into the clipboard, another may read the text as Unicode. This workaround usually works reasonably well if the legacy applications clearly specify the codepage they use.
Common problems in non-internationalised and legacy applications are fonts that are hard-coded and do not support all characters used in a text to be displayed or the wrongful assumption of a certain code-page.

3.3 The Bluebottle Text System

The core of the Bluebottle text system has a model-view-controller (MVC) structure with a thread-safe text model. An *EventSource* as introduced in 6.2.6 is used for the administration and invocation of observers.
The text model has the following properties:

- support for rich text with styles and attributes
- characters are stored in UTF-32 encoding
- support for embedded objects
- support for position markers that automatically float with the text

On a higher level of abstraction, the text system contains:

- a centralised text clipboard
- notions of a single system wide selected text and a single text selection
- a generic rich text viewer and editor with support for in-text command activations

- a powerful macro system
- support for *input mode editors*

On the system level it has the following properties:

- supports one or more registered observers that are efficiently informed about all changes in the text model
- thread-safe locking mechanism
- support for multiple simultaneous readers

The text utility module *AosTextUtilities* contains a loading and storing mechanism that uses *AosCodecs* for encoding and decoding a variety of text formats.

3.3.1 Programming Model

In the Bluebottle system many active objects (threads) can run at the same time and potentially simultaneously access a shared text model. The text model must therefore be protected by a common lock. Because most operations on the text model are non-atomic transactions, the lock has to be taken and released explicitly. An example of a non-atomic operation that needs to be carried out as a single transaction consists of reading the length of the existing text and adding the new characters at the last position.
The following object procedures implement the synchronisation mechanism:

- *AcquireWrite* and *ReleaseWrite* acquire respectively release the write lock
- *AcquireRead* and *ReleaseRead* acquire respectively release the read lock
- *GetTimestamp* returns a number that is increased whenever the text model is changed

Text model synchronisation is described in more detail in section 3.3.2. The text is modified using the following methods:

- *InsertPiece* inserts a piece into a text. The piece can be either a piece of characters or an embedded object.
- *InsertUCS32* inserts an array of 32 bit characters.
- *Delete* deletes a stretch of characters and objects.
- *SetAttributes* defines the attributes over a stretch of characters and objects.
- *UpdateAttributes* calls a delegate for each text-piece over a stretch of characters and objects. The delegate procedure can then, for each piece individually, change the attributes. This functionality is used, for example, when a single attribute such as the font size needs to be changed while leaving others such as the colour untouched.

- *CopyToText* Copies a stretch of characters into another text. This is for example used to copy selected text from an editor into the system wide text clipboard.

For all modification operations, the modifying thread needs to hold a write lock on the text. On top of the text modification interface of *AosTexts*, *AosTextUtilities* offers a *TextWriter* object that can be used to easily create rich texts. The *TextWriter* programming interface is described in more detail in section 3.3.6.

The object procedure *GetLength* returns the length of a text as the number of characters. Embedded objects count as one character each, so they can be addressed within the text for example for deletion. The method *RegisterPositionObject* registers text position objects within a text. Both procedures require the calling thread to hold at least a read lock on the text model.

3.3.2 Model Synchronisation

The abstract text model is protected by an optimised recursive reader/writer lock that is defined in the module *WMLocks*. Before any thread can change the text it must first acquire the write lock on the text model. Each access procedure that can lead to the modification of the text asserts the write lock is held by the calling thread. If the lock is not held, an exception is raised and the calling thread is aborted.
A thread can recursively take the reader/writer lock of a text. The read-lock is shared so that more than one observer can read on the text at one time. The following locking restriction is introduced and enforced with an inexpensive runtime check to prevent a notorious case of deadlock: If a thread needs to modify a text, it must acquire a write lock *before* taking any read locks[4]. This restriction orders the reader and writer part of the lock and hence prevents lock circularity and deadlock on a single text. If more than one text is shared between a number of threads and at certain times more than one text needs to be locked by a thread, an ordering of the texts must be enforced on a higher level to avoid deadlock.
While the model of a text is locked, it collects information about all modifications until the last write lock is released. It then informs all its observers about the place and kind of the modifications. Observers can be installed using the *onTextChanged* event source of the text model. Section 6.2.6 discusses the event mechanism in detail. Modification operations are insertion, deletion and attribute or style changes. If the modifications are not limited to a single continuous range of the text, the model informs observers about multiple changes, omitting the details. In most applications that involve a text with observers, for example an interactive text editor or a log viewer, the operations are insertions and deletions of single characters or words, not resulting in multiple changes notifications.
Before taking advantage of detailed change notifications, the observers must make sure that no other modification occurred between the time the mutator thread released its last write lock and the time where the observer acquired its (first) read lock. This is done by a comparison of a timestamp, set by the model into the change notification message, right before the lock was

[4]If a read-lock is still held on a text, after all write-locks are released, the thread may not take up a new write-lock before also releasing all the read-locks on the text.

released, with the timestamp read from the text, after taking the read-lock. The timestamp is incremented by the text model after each change of the text.

3.3.3 Text Positions

TextPosition objects are used for the text caret, selections, specially marked positions such as compilation errors and also for internal layout information such as the positions of line or paragraph starts.
They are a special kind of model observers that are, unlike normal observers, immediately informed about every single text modification, while the write lock is still held. They keep an internal position value that is adjusted on all text changes, so that they move with the surrounding text if it is changed. An insert operation in front of the internal position will increase the internal position by the number of inserted characters. Delete operations on the other hand decrease the internal position. If a delete operation on the text includes the internal position of a *TextPosition* object, the respective *TextPosition* is moved to the start of the deletion. A text position object that is set to the beginning of a word, for example, will always point to the beginning of this word, even if the text in front of it is changed.
*TextPosition*s can be registered with a text model using its *RegisterPositionObject* procedure. The *TextPosition* objects are automatically collected and unregistered by the system-wide garbage collector when they are no longer used. There is no manual un-register functionality.

3.3.4 Readers

A thread that wants to read from a text, must first acquire the read lock of the text. It can then open a *TextReader* on the text and set it to the desired start position. The reader object returns character by character and automatically advances its position on each call to the *ReadCh* method until the end of the text is reached. The style and attributes of the last read character are available in the *attributes* and *style* fields of the reader object. The concept of text readers has been taken from Oberon [117] and was extended to support styles, Unicode characters and thread-safety.
The reader can read forwards or backwards. Reading backwards is needed for operations such as finding previous line-breaks or word boundaries used to create the text layout and also for word-wise editing operations.
The *TextReader* is implemented as an extension of a *TextPosition* object as introduced in 3.3.3. If the reading thread gives up its last read-lock and the text is changed, the reader floats on the text.
The *TextReader* object takes an *AosTexts.Text* as a parameter in the constructor. Its position is changed with the *SetPosition* procedure that is inherited from the *TextPosition* object class. The direction of the reader can be set with the *SetDirection* procedure.

3.3.5 Attributes and Styles

For any range of characters, a style can be defined. If no explicit style is set, the editor uses a system wide default style. Styles can either be *named* or *ad-hoc*. Named styles can be defined per document or for classes of documents. If the user wants to set a style that is not defined for the document-class, an ad-hoc style is created that stores the relevant attributes.
The following text attributes can be defined in a style:

- vertical line offset
- font colour
- background colour
- font name
- font size
- font style
- explicit kerning

This set of text attributes is supported by the system-wide standard text editor (3.4) and is therefore available throughout the system.
Specialised layout or desktop publishing programs can add customised style extensions [63].
The style for a document class is stored as an XML file [112] containing a number of named *character-style* elements that specify the text attributes. Listing 3.1 gives an example of a text style file.

```
<styles>
  <character-style name="Normal" font-family="Oberon" font-style="0" font-size="10.0000"
    leading="12.0000" baseline-shift="0.0000" color="000000FF" bgcolor="00000000"
    tracking="0.0000" kerning="0.0000" h-scale="100.0000" v-scale="100.0000"/>
  <character-style name="Assertion" font-family="Oberon" font-style="1" font-size="10.0000"
    leading="12.0000" baseline-shift="0.0000" color="0000FFFF" bgcolor="00000000"
    tracking="0.0000" kerning="0.0000" h-scale="100.0000" v-scale="100.0000"/>
  <character-style name="Debug" font-family="Oberon" font-style="0" font-size="10.0000"
    leading="12.0000" baseline-shift="0.0000" color="0000FFFF" bgcolor="00000000"
    tracking="0.0000" kerning="0.0000" h-scale="100.0000" v-scale="100.0000"/>
  <character-style name="Lock" font-family="Oberon" font-style="0" font-size="10.0000"
    leading="12.0000" baseline-shift="0.0000" color="FF00FFFF" bgcolor="00000000"
    tracking="0.0000" kerning="0.0000" h-scale="100.0000" v-scale="100.0000"/>
</styles>
```

Listing 3.1: Excerpt of the Default Active Oberon Program Text-Style

The text style is normally generated by a text style definition tool that creates and manages the different styles for a document class.

3.3.6 Text Writer

The *TextWriter* is a wrapper on a text object that facilitates the creation of plain and rich texts. It is opened on a text and offers a streaming interface for text creation. The writer has an interface to select attributes such as font name, style, size, colour, background colour and vertical offset that are applied to all characters that are written via the *Add* procedure of the writer after setting the new attribute. The *Add* procedure implements the *AosIO.Sender* micro interface[5]. It interprets the content of the added buffers as UTF-8 encoded characters.
The *TextWriter.GetWriter* procedure returns an *AosIO.Writer* object instance *w* that is opened on the *Add* procedure of the *TextWriter*. Successive calls to the *GetWriter* procedure return the same writer instance *w*. The *TextWriter* automatically calls the *AosIO.Writer.Update* procedure of the *AosIO.Writer w* before the font attributes are changed so that buffered but not yet added characters are added with the correct attributes. If another instance of an *AosIO.Writer* is opened on the *Add* procedure, the application programmer needs to explicitly call the *Update* procedure on the writer before calling any attribute change procedures to make sure the stream is flushed before the new attributes are set.
A *TextWriter* object should not be shared between threads. If more than one thread needs to write on a single text, each one should bring its own *TextWriter* object. Note that opening more than one *TextWriter* on a text is safe.
The following lists and documents the procedures in the *TextWriter* object:

- *PROCEDURE &Init(text : AosTexts.Text)*; The constructor *Init* takes a text as a parameter to which the writer is bound to.

- *PROCEDURE Add(VAR buf: ARRAY OF CHAR; ofs, len: LONGINT; propagate: BOOLEAN; VAR res: LONGINT)*; The add procedure adds *len* UTF-8 encoded bytes from the buffer *buf* to the text, starting at offset *ofs*. The parameter *propagate* indicates whether the buffer should immediately be written to the text or if the *TextWriter* can delay the writing to accumulate more characters. The *propagate* parameter is a requirement of the *AosIO.Sender* micro interface that is used to connect *AosIO.Writer* objects. The current implementation of *TextWriter* ignores the parameter. The return parameter *res* is set to *AosIO.Ok* if the operation was successful.

- *PROCEDURE GetWriter() : AosIO.Writer*; returns an *AosIO.Writer* instance *w* that is opened on the *Add* procedure. The procedure always returns the same instance *w*. To flush the buffer of the *AosIO.Writer* instance *w*, its *Update* procedure is automatically called before applying any attribute or style changes to ensure the queued characters are written in the style or with the attributes that were set when the characters were added to the writer.

- *PROCEDURE SetPosition(pos : LONGINT)*; repositions the *TextWriter* within the text. If *SetPosition* is not called, the new text is appended at the end of the text.

[5] A delegate procedure variable comprising a method pointer and object reference

- *PROCEDURE SetFontName(name : ARRAY OF CHAR)*; changes the font.
- *PROCEDURE SetFontSize(size : LONGINT)*; changes the font size.
- *PROCEDURE SetFontStyle(style : SET)*; change the font style. The style set can be empty or contain *WMGraphics.FontBold* and/or *WMGraphics.FontItalic*.
- *PROCEDURE SetFontColor(color : LONGINT)*; changes the font colour.
- *PROCEDURE SetBgColor(bgColor : LONGINT)*; changes the background colour that is drawn behind the inserted text.
- *PROCEDURE SetVerticalOffset(voff : LONGINT)*; changes the vertical offset for the inserted text. This can for example be used to generate a superscripts or subscripts.
- *PROCEDURE SetStyle(style : ARRAY OF CHAR)*; applies a certain style for the inserted text. If a style is present, it overrides previously set attributes. It is recommended to use styles instead of setting individual attributes to achieve more flexibility and a unified look.

Appendix A.2 shows a commented example program using the *TextWriter* interface to create a rich text.

3.3.7 Internal representation

The internal representation of the text model is completely encapsulated by the text model interface and can be transparently replaced. The default implementation stores the text in a doubly linked list of piece-descriptors containing pointers to arrays of UTF-32 encoded characters. The fixed-size UTF-32 encoding was chosen for the internal representation because it offers direct access to each character while at the same time offering enough codepoints to store any character specified in Unicode 4. An internal UCS-16 [6] representation as used in some commercial systems cannot store all characters defined in Unicode 4.0 because the highest codepoint (U+10ffff) is far above the highest codepoint that can be encoded with 16 bits. Table 3.1 compares several important encoding formats with respect to the range of characters that can be stored, possible access methods and byte order dependence.
When inserting new characters into a text, the piece at the respective insertion position is split and a new piece is inserted. When deleting, the pieces at the start- and end-position of the deletion are split and the outer pieces linked together. To reduce the number of small pieces and internal fragmentation in a text, neighbouring piece-descriptors can be merged, if their styles and attributes are the same. The merging works by copying the referenced characters into a single character array and replacing the two piece-descriptors with a new one, referencing the copied characters. To avoid long copy operations when merging, the merge is only performed if the resulting combined size of both pieces is smaller than a threshold value.

[6] Systems like Windows NT that are based on an internal 16 bit character representation can, for some applications, be retrofitted by interpreting the 16 bits as UTF-16 variable sized character codes.

While the linked list is very efficient for linear reading of the text, directly accessing a character can result in a lengthy linear traversal. To speed up random positioning, the text maintains a lazy updated sorted array that contains references to the single pieces. Bisection search is used on the array to position on the text in $O(log(textsize))$. Figure 3.1 shows the internal structure of the default text model implementation.

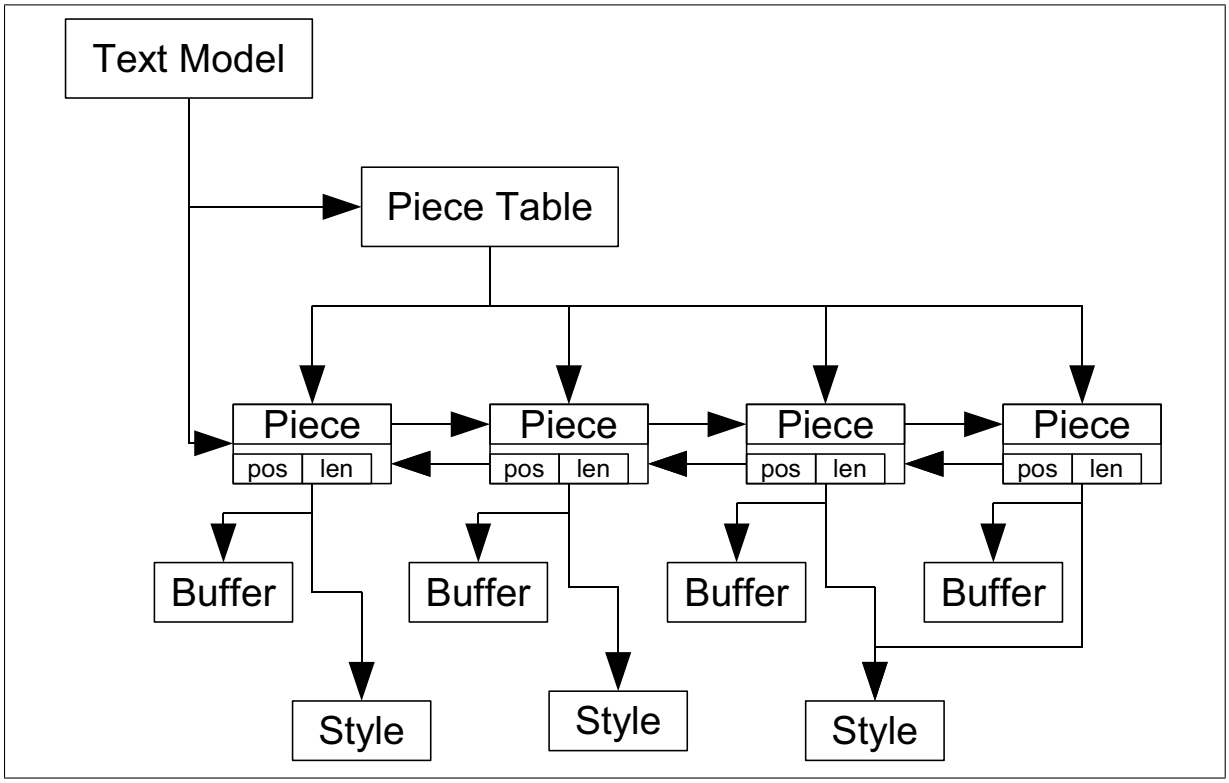

Figure 3.1: AosTexts Internal Structure

3.3.8 External Representation

In the Bluebottle text system, the loading and storing of texts is implemented as external operations on the text model. This has the advantage that the internal representation of the text model and the different external representations can be maintained independently. The loading and storing of texts is handled by text encoders and decoders as defined in the *AosCodecs framework* that is discussed in detail in chapter 8. In the 2004 version of Bluebottle, the system by default supports the following plain and rich-text formats:

- *UTF-8* (plain text)
- *ISO-8859-1* (plain text)
- *ASCII* (plain text)
- *UCS-16* (fixed size 16 bit, plain text in little endian format)
- *Bluebottle* (Rich text, XML-structure in UTF-8 encoding)

- *Oberon* (Rich text, 8-bit Oberon character set, without embedded gadgets)

The Bluebottle Text Format

The Bluebottle text file format is able to store the full character set, style and attribute information of the Bluebottle text model. Storing files in other formats can result in missing characters or loss of style information. The Bluebottle text file format is an XML document stored in UTF-8 encoding. The style information is stored per span as a reference to a style or as a directly specified ad-hoc style. The text data is stored in a `"<![CDATA["` [112] section. The *CDATA* sections have the advantage over normal tags that spaces, line-breaks and tabulators can be used without escaping. The text within a span can therefore be read and modified within any UTF-8 capable editor if there is no Bluebottle system available. Listing 3.2 gives an example of a text using predefined and ad-hoc styles.

```
<?xml version="1.0" encoding="UTF-8"?>
<?bluebottle format version="0.1" ?>
<Text>
<Span style="Normal"><![CDATA[This is in Normal style. ]]></Span>
<Span style="Bold"><![CDATA[This is Bold style. ]]></Span>
<Span style="AdHoc Oberon 20 1 0 222222FF FFFFFF00"><![CDATA[This is an AdHoc style.
It is using Oberon font 20, bold, no vertical offset
in a dark gray colour on white ground. ]]></Span>
</Text>
```

Listing 3.2: Example of the Bluebottle Text Format

	ASCII	UTF-8	UCS-16	UTF-16	UTF-32
Encode all Unicode characters	no	yes	no[a]	yes	yes
Indexed character access	yes	no	yes	no	yes
Byte order independent	yes	yes	no	no	no

***Table 3.1**: Encoding Characteristics*

[a] yes up to Unicode 2.0

3.4 Text Editor

The Bluebottle standard text editor follows a strict model-view-controller architecture. The layout mechanism, rendering, selection and navigation is implemented in the module *WMTextView*. The controller is implemented in the module *WMEditors*. An arbitrary number of text views or text editors can be opened on the same text model.
The text viewer is implemented in the component *TextView* as an extension of *VisualComponent* that is described in chapter 6. The editor component is itself an extension of a *VisualComponent*, containing two optional scrollbars and a *TextView* component.

3.4.1 Text View

Text Layout The *TextView* always keeps a rough text layout of the entire text up to date. The rough layout mainly consists of a list of the text positions of all line starts and the height, ascent, descent, active tabulator positions and alignment for each line. The tabulator positions are shared by reference over the range of lines that have the same settings. This rough layout information is used for an efficient mapping of view positions to text positions and vice versa. To reduce the computational overhead needed to create this rough layout, the *TextView* registers as a text model observer on the text and uses the detailed change messages to perform differential updates on the layout, where possible. Most text editing operations, are single insertions or deletions of a number of consecutive characters that can be handled in a differential layout update, if the timestamp of the text (see 3.3.2) and the timestamp of the modification notification are equal. In these cases, the detailed line layout algorithm only needs to operate on the lines that are either affected directly by the deletion or insertion or by possibly changed word-wrappings or character-wrappings. The first line that is not affected by wrapping effects, can easily be detected by comparing the newly calculated line-start position with the stored line-start position. If the difference equals the *character delta* from the change notification message, the line is not affected. With word-wrapping enabled, the detailed calculation of the layout can normally be limited to a single or a few lines.
In the rare case that multiple editing operations are combined or the timestamp of the text has changed, the layout of the entire text needs to be rebuilt.

Line Layout The detailed layout of a line is always created on demand to avoid the memory and computation overhead of keeping the detailed structure up to date in memory. It is used to create or update the layout of the text and to map horizontal screen positions to text positions and vice versa. The layout algorithm is in principle the same as the algorithm that draws a text line. It starts with the character or object in the text at the position stored in the respective line start information of the text layout and calculates the bounding boxes of all characters or objects in the line. If the algorithm is used for a position mapping, it can stop early if the position is found. If it is used to calculate the new line layout, it continues until the end of the line is reached. In the word wrapping case, the algorithm sometimes needs to step back several positions in the text so that a word can be moved to the next line. If the word is longer than the line width, character wrapping is performed. The word wrapping problems and special cases can be ignored in the position mapping cases because the mapping can rely on the text layout.

Markers Text view supports several kinds of special position markers. The most prominent markers are the caret and text selections. The text view supports two kinds of *point* position markers:

- Caret markers that adapt to the height of the line can be drawn in an arbitrary translucent colour. The standard text caret is using such a caret marker.
- Translucent images that can be positioned with an offset relative to the text position at

the base line. This can for example be used to mark compile errors in a program text.

The point position marker object class can be extended by a programmer to generate specialised markers. Snapshot 3.2 *A* shows an example of point position markers highlighting errors in a program text that were caught by the compiler. The caret is also visible. The text view also

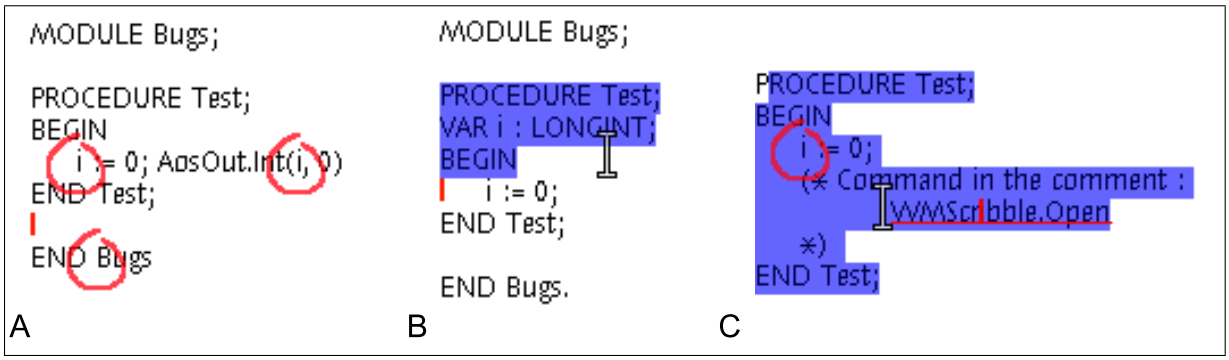

Figure 3.2: Snapshots of Different Text Markers

supports two kinds of text *range* markers:

- Overlay selections in arbitrary translucent colours can be used as highlights. The selected text is an example use for this kind of marker.
- Straight or undulated underlinings in arbitrary colours are used for example to highlight commands, hyperlinks or to mark errors or problems.

Snapshot 3.2 *B* shows a selection as a range marker. A caret and a mouse pointer are also visible.
All the markers can be used at any position and in any number in the text. Snapshot 3.2 *C* shows an example with two point markers and two range markers. The markers are implemented in the *TextView* and are not stored in the text. Several views that watch the same text can display different markers.
Point position markers are implemented with a single *AosTexts.TextPosition* object, that automatically floats with the text. The range markers use two separate *AosTexts.TextPosition* objects.
The text caret and a marker for the selected text are always present in each text view but can be switched to invisible.

Navigation and Command Invocation The navigation in the text as well as the selection and in-text command invocation is the responsibility of the text view. The rationale for this lies in the wish to give the user as much flexibility as possible to work with texts. It is therefore necessary that the user can select text even if the text is not editable. The selected text can then for example be copied or interpreted. The argumentation therefore also applies to the invocation of commands or the opening of hyperlinks.
For the navigation in the text, the *TextView* supports the cursor and other special keys, the mouse pointer and the mouse scroll wheel. Selection of text can be either performed with the mouse or

with key combinations. The support of both selection methods is dictated by the time overhead of moving the hands from the keyboard to the mouse and back.
When selecting with the mouse, first the mouse cursor is moved to the starting position of the desired selection and the left mouse button is pressed. Then the mouse cursor is moved to the end of the desired selection and the mouse button is released. The text view updates the selection marker during the operation for visual feedback. The scroll wheel can be used during the operation to navigate in the text. When the left mouse button is pressed twice at the same position, an entire word is selected. If the button is kept down the second time, selection continues on word level.
To select text using the keyboard, the text cursor is first positioned to the desired start of the selection. The *shift* key is then pressed and held down while the cursor is moved to the end of the desired selection, where the *shift* key is released.
Commands in the document can be invoked with the start command in the context menu that can be opened with the right mouse button. Alternatively, commands can be invoked by clicking them with the middle mouse button.
The exact functionality and a list of all navigation, selection and invocation options can be found in the *Bluebottle Tutorial* [33].

3.4.2 Editor

The text editor is implemented as a *VisualComponent* that contains a *TextView* component and two optionally visible scroll bar components. The text editor component intercepts all keyboard events that are directed to its *TextView* sub-component. Navigation key-events are directly forwarded to the text view. All other key-events that result in a modification of the text are handled by the editor component. If an input method editor is installed for the editor component, modification key events are forwarded to the respective IME. The IME is then responsible to modify the text model according to its rules. If no IME is installed, the editor itself inserts or deletes characters at the relative cursor position. Key-event messages that are sent to the text view component are intercepted by the editor which analyses it. If the key-event message stems from

- a navigation-key, it is passed on to the *TextView* and the editor returns from the key-event handler.
- a special edition key combination like copy, cut, paste or similar it is handled by the respective editor method.
- a modification key, it is either handled by the editor or forwarded to an installed IME. If an IME is installed, the editor returns from the key-event handler after forwarding the message. Otherwise, the key message is sent to all installed text macro processors. If none of the macro processors handles the event, the text editor first locks the text model, reads the caret and selection position of its view and performs the respective insertion or deletion operation. After releasing the writer lock on the text model, all views are automatically updated. Since the caret is implemented as an *AosTexts.TextPosition*, it

automatically floats to its new position after the modification operation. Hence the editor does not need to directly operate on the caret's position.

3.4.3 Input Method Editors

WMInputMethods provides input method support for the Bluebottle system. It defines an abstract input method editor and a plug-in mechanism for specific implementations. The text editor can install an instance of an input method editor as a layer between its keyboard message handler and its character insertion routine. Figure 3.3 shows the way travelled by a keyboard event in the text editor. First, the editor discerns between navigation-key-events and modification-key-events. In case of a modification-key-event, it checks if an IME is active on the editor. If yes, it forwards the key-message to the active IME. Otherwise the character represented by the key is directly inserted into the text model. If a key event is forwarded to an IME,

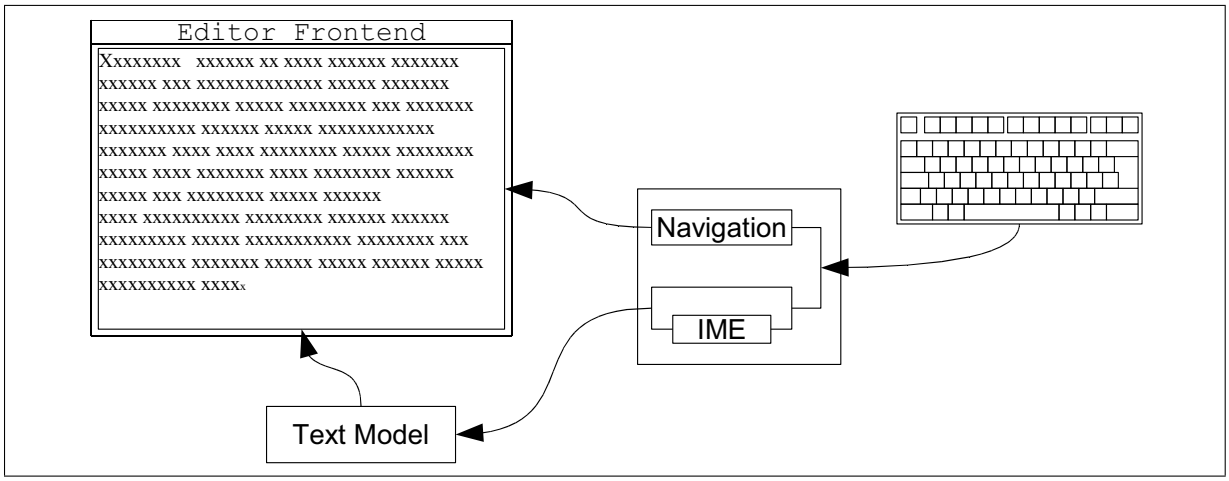

Figure 3.3: Data-Flow of a Keyboard-Event to the Text Model

it can be processed in different ways. Simple IMEs can just re-map the key to another Unicode character. In this simple one-to-one mapping case, the IME acts similar to a keyboard driver. Examples for this simple mapping are input method editors for Cyrillic or Greek character input.

In Bluebottle, *WMCyrillicIME* implements an input method for Cyrillic characters. It mainly consists of a mapping table. For more complicated input methods, the IME needs to take over the complete keyboard control from the main editor. *WMPinyinIME* implements a complex IME for Bluebottle. When the input method editor gets the first key-event from the editor, it opens a small window next to the cursor position in the main editor and tells the display space manager to forward keyboard events to the new window. Inside the IME window, key strokes are sent to an embedded editor[7]. While typing into the IME, it displays all possible characters or character compounds that match the pronunciations typed so far. The candidate list is sorted by their usage frequency as taken from the *Unihan* database. As soon as the typed text no longer matches any characters or character compounds from the IME's dictionary, it inserts the

[7]To avoid recursion, the editor inside the IME window may not enable another IME.

last recognised best match into the text and uses the new input as a new character or start of a character compound. The user can at each time select from the presented candidate list and force the insertion of the selected character. Selection and insertion is possible via keyboard or mouse. *WMPinyinIME* optionally supports *tones* to reduce the number of matches and a first character match strategy for inserting common character compounds or short phrases by the syllables' first characters. To support students learning Chinese, it displays the *tones* for the recognised characters and phrases and optionally a translation to English.
The simple selection strategies of *WMPinyinIME* and its limited dictionary of characters and phrases leaves room for improvement. To match the usability level of the best commercial pinyin IMEs, support for automatic learning of new character usages, a better prediction method as well as support for a certain pronunciations tolerance should be added to *WMPinyinIME*.

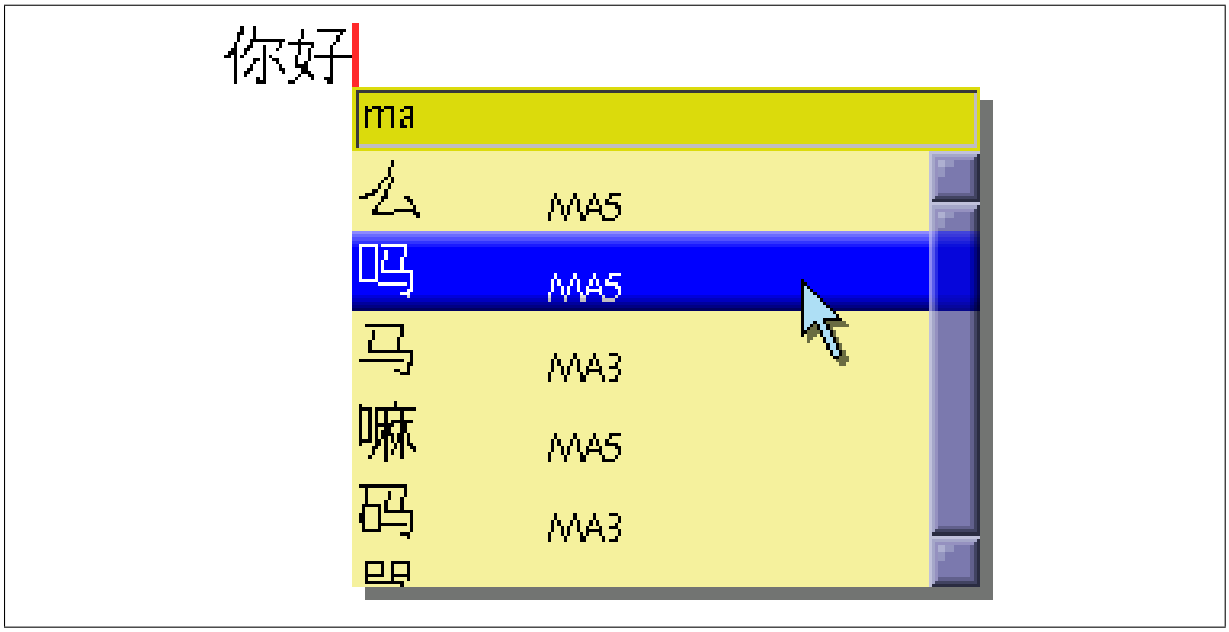

Figure 3.4: Bluebottle Pinyin IME in Action

3.4.4 Macros

The Bluebottle text editor supports macro plug-ins. Text macros plug-ins are procedures that operate on the text and cursor position of a text editor. The macro procedure is invoked by a special key or key combination. Although the system is open to support other key combinations, all the system defined macros are invoked by the *insert key*. In many systems, the insert key toggles between a character insertion and character overwriting mode. The system-wide text editor in Bluebottle does not offer these two modes, it is always in the insertion mode. The *insert key* is used more intuitively as a key that really inserts something, namely the result of a macro procedure evaluation.
When the default macro plug-in recognises the *insert key* it reads the text on the cursor position backwards to either a white-space character like a space or a line-break or a colon. The text between this white-space or colon character and the cursor position is interpreted as the name

of the macro procedure. Hence, the number of different macros that can be invoked by the *insert key* is not limited.
The macro names and macro functions are specified in *Macros.XML*. They replace the macro name in the text with a larger piece of text. The larger text can either be a fixed string or it can be a parameterised text. The macro parameters are searched in front of the macro name, each separated by a colon. The macro parameters can be inserted between fixed string elements in any order and repetition. Table 3.2 gives examples of some macros. The left column specifies the unprocessed macro string with optional parameters. The right column shows text that results when the macro in the left is invoked with the *insert key*.

Macro	Evaluation
tf	frey@inf.ethz.ch
Test:P	PROCEDURE Test; BEGIN END Test;
IME:WMInputMethods.IME:o	TYPE IME = OBJECT(WMInputMethods.IME) VAR END IME;

***Table 3.2**: Examples of Macros and their Evaluations*

3.5 Usage Example - Programming Environment

Since the second half of 2003, the Bluebottle system has been developed mainly with its own tools. Before, the system was cross-developed with tools of the traditional Oberon system running as an application on top of the Bluebottle system (see 5.7). The central element of program development on Bluebottle is a text editor that is specialized for programming and debugging in Active Oberon. The *Programmer's Editing Tool* consists of a number of standard GUI components and some glue code to integrate the compiler. Figure 3.5 shows a snapshot of the programmer's editing tool.

GUI widgets on top from left to right

Filename input field specifies the filename of the program text that is to be loaded or stored

Load button loads the text in the file specified in the filename input field

Store button stores the text in the file specified in the filename input field. The *Store* button is marked with an exclamation mark if the text has been changed since it was last stored.

Format menu selects the text format for loading or storing.

Search opens a search panel to search strings within the loaded text.

Compile compiles the currently loaded text using the compile options in the compile options input field.

FindPC searches for a selected program counter position in the source text. The program counter value is taken from the last selection, normally in a trap window. If no number is selected, the program opens a query input dialog.

Split opens an additional view on the text model that can display a different position of the source text.

Compiler options input field contains the compilation options like the target processor.

GUI widgets in the left panel from top to bottom

Program structure displays the structure of a loaded *Active Oberon* source code. Clicking into the structural overview positions the text cursor at the respective position in the source code. If an XML document is loaded instead of an *Active Oberon* program, the XML document's structure is represented and a tool offers support for checking the syntactical correctness of the XML text.

Scratch text is a tool area that is synchronized between all instances of the programmer's editing tool. It is used to note frequently used commands. It is a good example of the model view architecture of the text system. A single text model is displayed by a possibly large number of text views.

GUI widgets in the bottom In the bottom of the window a grid component displays the result of the last compilation, either the size of the resulting object file or a list of compilation errors. Clicking at an error message positions the text carret to the respective error position. The compilation errors are also highlighted in the program text.

3.6 Conclusion

An efficient and versatile system-wide text subsystem is an important part of any modern (graphical) user interface. In current systems, many application programs bring their own text system, that is only more or less compatible to the text-systems of other applications. The result of this inhomogeneity is a number of conversion problems that lead to loss of formatting, style and sometimes even characters when text is exchanged, even if the text remains in the memory of a single machine. The lack of a centralised text system hinders the innovation of system-wide text support tools such as dictionaries, macros, acoustic readers and more.
In the Bluebottle system, according to the Oberon tradition, a system-wide text system is one of the central features of the user interface. It significantly differs from and extends the standard Oberon text system [117] mainly in respect of its thread-safety, support internationalised character sets, styles and text positions.

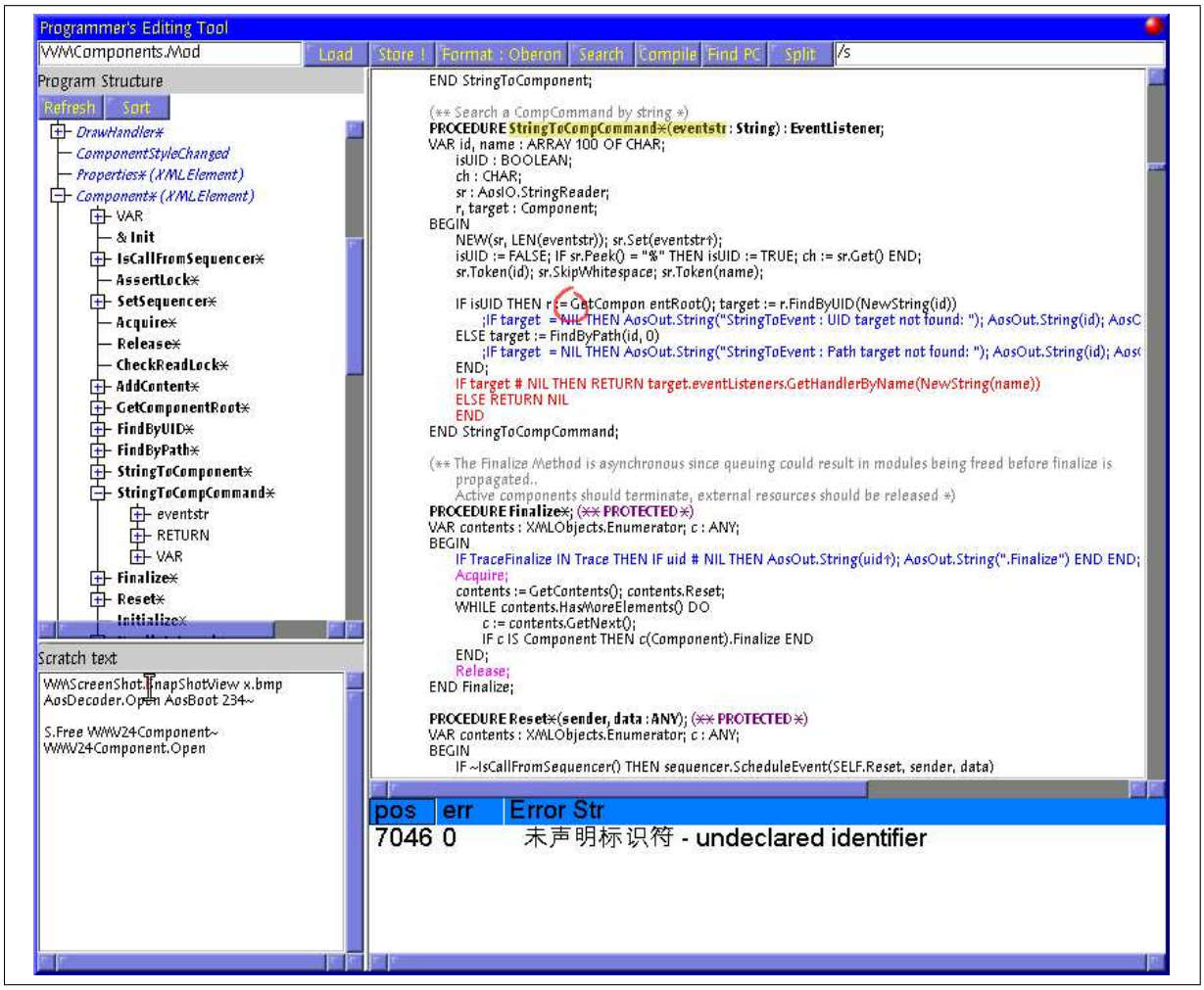

Figure 3.5: A Snapshot of the Programmer's Editing Tool

4

Graphics System

Idealism is what precedes experience;
cynicism is what follows.

— David T. Wolf (1943 -)

4.1 Introduction

This chapter describes the graphics system that has primarily been developed for the Bluebottle graphical user interface. Figure 4.1 shows a rough overview of the graphics system. It forms the basis for the display space manager and component system and comprises the *Hardware Framebuffer*, *Bitmap Abstraction* and *Canvas Abstraction*. The higher *Application* layers are the topic of chapters 5 and 6. Some *Applications* on an even higher level are described in chapter 9. The prefix *WM* in module names indicates that the module belongs to the window manager. It is also used to differentiate new modules from existing Oberon modules with the same name. The problem of name clashes is discussed in more detail in 10.2.

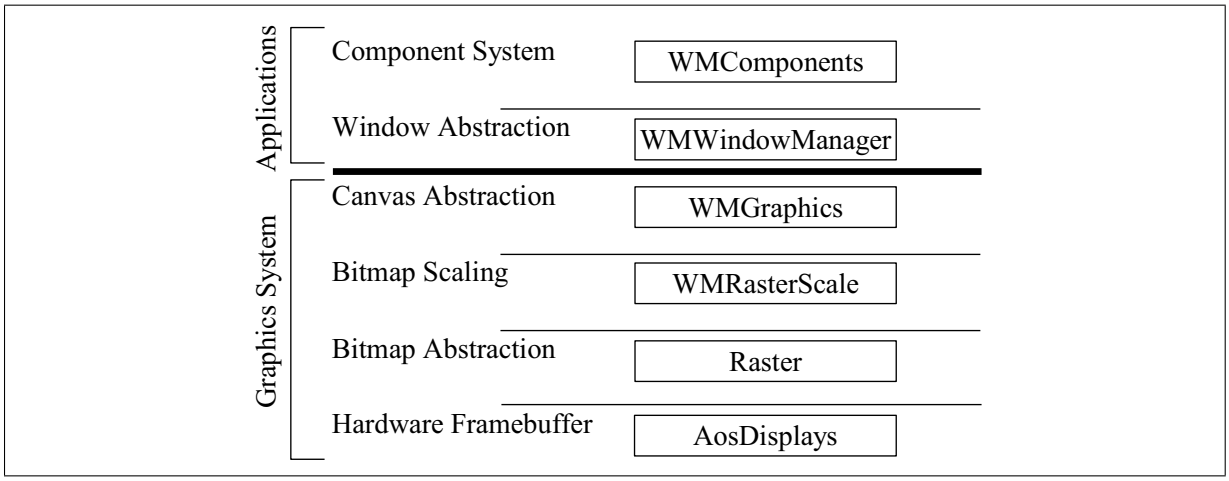

Figure 4.1: Graphics System Overview

4.2 Frame Buffer

The frame buffer driver, whose interface is defined in *AosDisplays*, is the lowest level of the Bluebottle graphics system. It mainly offers a transfer routine that moves a bitmap stored in main memory in the framebuffer's native colour format to the hardware framebuffer. Since no clipping, conversion or blending operations are performed on this level, the implementation is simple and can easily be ported to different hardware/software platforms. Until today the Bluebottle system including the graphics system has been ported to the following platforms :

- *DNARD* an ARM based network computer of DEC [25]
- *WinAos* an implementation of the Bluebottle runtime system on IA32 Windows [34]
- *QBIC* a wearable XSCALE based belt integrated computer built in the department for electrical engineering at ETH Zürich [2]

On the IA-32 implementation, the default framebuffer driver in *AosDisplayLinear* sets the processor's cache properties for the frame buffer memory area to *write combining* which significantly increases the memory throughput to the frame buffer. When writing to memory areas that have *write combining* enabled, the processor does not immediately forward the written data to the caches or main memory but stores it in one or more separate buffers until a number of bytes[1] have consecutively been written. It then transfers the whole buffer in an optimised way, possibly one burst, significantly reducing the bus transaction overhead [46]. Incompletely filled buffers are transferred by the processor on many software and hardware events, such as interrupts, uncached loads or stores, *INPUT*/*OUTPUT*/*IRET* instruction and many more. In the Bluebottle system, such events occur several times every millisecond. The frame buffer driver therefore does not need to flush the buffers explicitly.
More than one local frame buffer can be registered in *AosDisplays* as well as in the display space manager as described in 5.4. The support of multiple frame buffers allows the installation of more than one screen to a single computer.

4.3 Bitmaps

Bitmaps are two dimensional arrays of pixels, characterised by *width*, *height* and *colour format*. The colour format characterises the colour space and colour depth of the pixels in the bitmap. The Bluebottle bitmap module *Raster* supports a large number of different RGB colour formats and offers methods to convert between each format. Some of the colour formats also include an alpha channel that allows translucent colours. *Raster* is derived from module *Images* by Erich Oswald [84]. It differs from *Images* in being thread-safe, supporting additional specialised and optimised transfer modes and additional blending modes.
The thread safety of the module was achieved by removing all global module states. Images can be shared between activities as long as their characteristics (width, height, and colour format) are not changed while they are accessed by more than one thread. Instances of drawing

[1] 32 bytes in the Pentium II case

mode objects may not be shared between activities since they contain status information about drawing operations in progress.
Upon calling a bitmap operation such as a *Copy* or *Fill*, the respective pixel transfer method from the source to the destination image with respect to the selected blending operation is chosen in the method *Bind* and stored in a procedure variable in the respective drawing mode object. To improve the speed when operating on common bitmap formats, a number of specialised transfer routines have been added. One specialised transfer procedure is described in more detail in section 4.3.1.
An additional blending mode that inverts the pixels of the destination bitmap has been added for the efficient implementation of an ETH Native Oberon [79] compatible virtual display adaptor on top of a bitmap. The virtual display adaptor is used to integrate the ETH Native Oberon system as a subsystem on top of Bluebottle.
The bitmap access procedures do not support clipping. Providing parameters that operate outside a bitmap leads to an assertion violation and the calling activity is trapped. On top of the bitmap layer, higher level graphic interfaces can be implemented. *AosGfx* and *WMGraphics* are two examples. AosGfx is an extension of the *Gfx-Framework* [84] by Erich Oswald. It mainly extends Gfx by adding thread-safety and support for translucency. While it is possible to implement AosGfx windows on top of the display space manager as described in 5, the component system and display space manager itself use the high level canvas abstraction provided by WMGraphics which is described in detail in section 4.4.

4.3.1 Alpha Blending

Translucent colours are stored as *RGBA* values (red, green, blue and alpha components). The additional alpha channel value determines to what level a colour is solid or semi-transparent. In Bluebottle, an alpha channel value of 0 is interpreted as completely transparent. This means, whatever values the RGB components have, they do not affect the target bitmap, if the colour is painted to it. An alpha channel value of 255 is interpreted as completely solid. A solid colour completely covers existing colours on a bitmap where it is painted to. Alpha values $0 < \alpha < 255$ result in interpolated colours.
Alpha blending operations between translucent bitmaps are quite expensive in terms of processing time since they involve a per pixel reading and decoding of the source and destination colours, per component (red, green and blue) interpolation between the source and target colours, encoding of the resulting colour value into the target colour format and writing it back. The special cases of the alpha values 0 and 255 can be optimised so that no read-out of the target pixel is required.

To make translucency fast enough for interactive use, some transfer routines have been improved by taking advantage of the Intel *MMX*[2] vector operations that allow working on all the three colour components in parallel (fig. 4.2) [50] [47]. In these optimised cases, the MMX code has been measured to be about 50 times faster than the generic Oberon implementation, five times faster than a specialised Oberon implementation and about twice as fast as a hand op-

[2]Multimedia Extensions

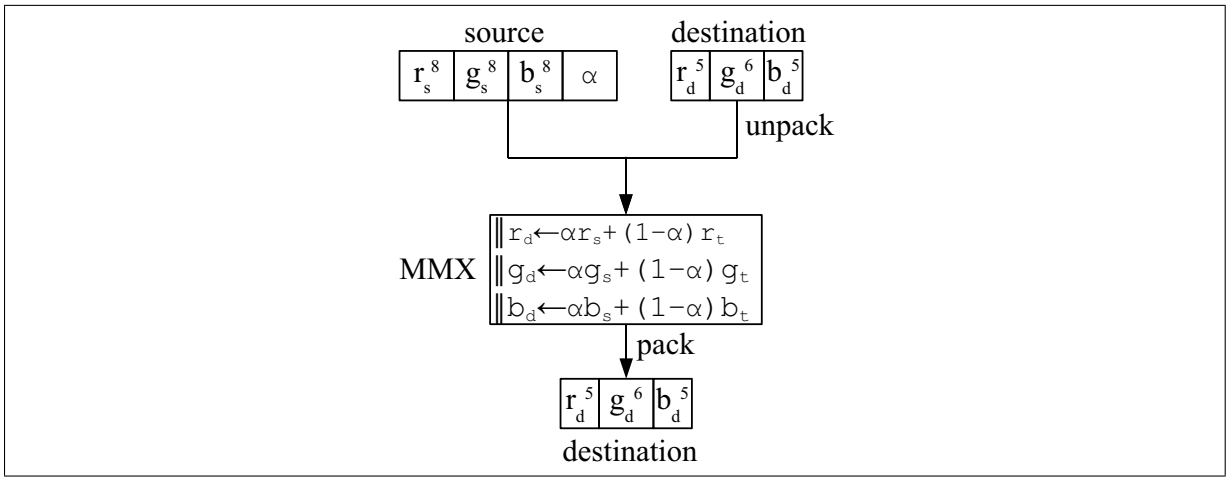

Figure 4.2: Parallelised Alpha Blending

timised assembler code without taking advantage of MMX instructions. If the processor does not support MMX, a fall-back routine is used.

4.3.2 Scaling

Apart from alpha blending, scaling of bitmaps is another important feature of the graphics system. The scaling routines are implemented in the module *WMRasterScale*. Similar to the format conversion routines, there is a generic scaling routine, that can handle all the formats with or without alpha blending. The zooming routines support a quality parameter that decides the filter function to be applied. In the current implementation an unfiltered mode and a bilinear interpolation mode are supported. The unfiltered mode is significantly faster than the filtered variation since it only reads as many pixels from the source bitmap as there are present in the destination area. The filtered variation in its current implementation considers four source pixels for each destination pixel for the bilinear interpolation. For efficiency reasons, the most frequently used source/destination combinations have been modestly optimised in assembler.

4.3.3 Loading and Storing

The graphic support module *WMGraphics* offers procedures for loading and storing images in various formats. The load and store procedures determine the format of an image file by the suffix of its name and use the appropriate image encoder or decoder from the *AosCodecs* framework that is described in detail in chapter 8.
The loader procedure *WMGraphics.LoadImage* takes the filename as a parameter and a boolean flag that specifies whether or not the loaded image can be shared:

```
loadedImage := WMGraphics.LoadImage("flash.png", TRUE);
```

The procedure returns NIL if the image could not be loaded for some reason. All shared images are stored in a list and are reused if they are loaded again in shareable mode. An image that is loaded as *shareable* may not be modified. All decoration images used in the display

space manager (5.6) and GUI component system (6) are loaded shareable, so that only one copy of each decoration image exists in the system memory. A shareable image that is no longer referenced is collected by the system wide garbage collector.
Images that are stored in streamable graphics formats such as *GIF*, *PNG*, *BMP* or *JPEG2000* can be directly loaded from archive files such as *zip* or *tar files* by specifying the archive file followed by "://" and the file name within the archive file. This makes it possible to compactly store all images that are used for a complete GUI style or an application program within a single archive file.

4.4 Canvas

The canvas abstraction offers a higher level of drawing primitives that support clipping and relative positioning of coordinates. In the *display space manager* (Chapter 5), the canvas abstraction is used for drawing display objects to viewports. The GUI *component system* (Chapter 6) uses it for drawing components into the buffers of display space objects.
In the display space manager and component system the *BufferCanvas* is the most commonly used implementation of the abstract *Canvas* object. It can be created as a wrapper on any *Raster.Image*.

The canvas abstraction can also be implemented on graphical primitives other than images.

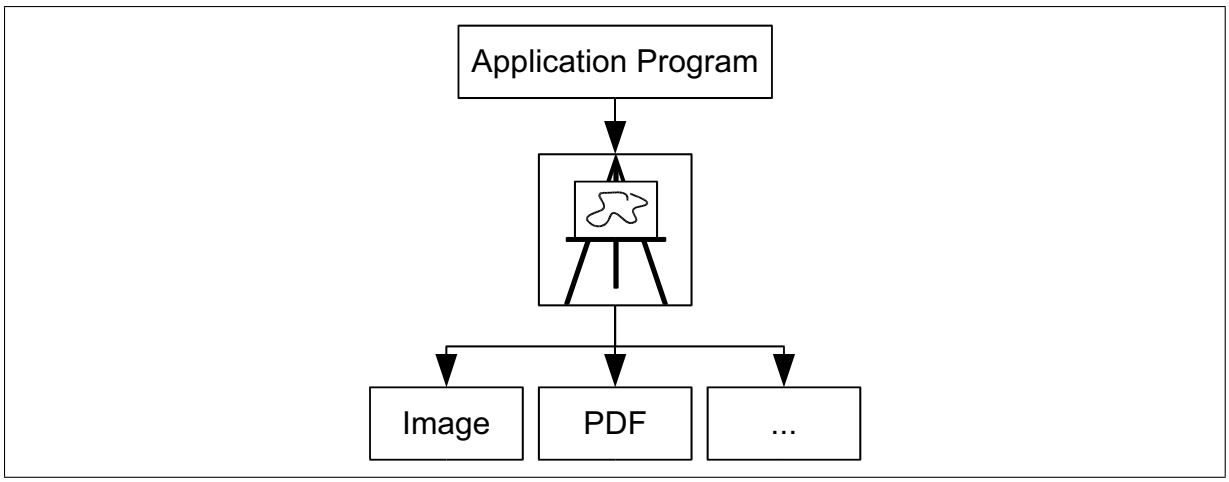

Figure 4.3: Canvas Abstraction

As a proof of concept, a special canvas has been implemented that allows applications to draw into a page of the popular PDF [39] [1] file-format. Figure 4.3 shows an abstract canvas as an interface to different targets for graphical content.

4.4.1 Clipping and Translation

The canvas abstraction supports rectangular clipping that is used in the display space manager and component system to ensure that no window or component can draw outside the assigned bounding box. The canvas supports two clipping rectangles that limit the drawing operations. One is a soft limit, the other is a hard limit. The soft clipping rectangle can be freely defined

with the *SetClipRect* method but is clipped at the the bounds of the hard limit. This ensures that the soft clipping rectangle always remains within the bounds of the hard clipping rectangle. All graphic operations are clipped at the soft clipping rectangle. The hard limit can only be narrowed but never extended. The distinction between hard limit and soft limit is used to allow sharing a canvas between different methods and objects. A method can set the hard clipping rectangle before calling another method and be sure the called method cannot draw outside the hard limit. The called method can still use the soft limit for special drawing effects like clipping bitmaps or glyphs. Both limits are stored with the *CanvasState* (4.4.2). By storing the canvas state before narrowing the hard clipping rectangle, it can be reset to the previous size by restoring the canvas state.
When a GUI component in Bluebottle needs to draw a sub-component, it first stores the canvas state, then sets the clipping rectangle to the bounds of the sub-component and changes it to a hard limit with a translation according to the top left position of the sub-component. Then the draw method of the sub-component is called. The translation is transparent to the sub-component and its drawing area starts at the *(0, 0)* coordinate. When the drawing procedure of the sub-component returns, the canvas state is restored. The soft clipping rectangle is changed to a hard limit with the *ClipRectAsNewLimits(dx, dy : LONGINT)* method. It takes two parameters that specify a coordinate translation to be applied to the drawing primitives. *dx* and *dy* are normally set to the coordinates of the top left corner of the clipping rectangle so that the accessible drawing area always starts at *(0, 0)*.

4.4.2 Canvas State

The canvas state contains all the state information of a canvas object. It can be stored with the *SaveState(VAR cs : CanvasState)* method that returns an instance of a *CanvasState* object. If the *CanvasState* variable *cs* is *NIL*, a new object instance is created, otherwise the existing instance is reused for efficiency reasons. Because concrete implementations of a *Canvas* object may need to store specialised state information, the returned *CanvasState* instance may be an extension of the abstract *CanvasState* object.
The canvas state is restored by the *RestoreState* method. A state can only be restored to the *Canvas* object that saved it. A saved state can be restored more than once so that it is possible to use the *RestoreState* method to repeatedly reset a *Canvas* object to a known state.
The canvas state contains the soft and hard clipping rectangles (see 4.4.1), the current coordinate offset, selected font and font colour.
The canvas state is normally stored before setting a hard clipping rectangle and calling a draw method that may under no circumstances draw outside the given clipping boundaries. When the draw method returns, the stored canvas state is restored. This not only restores the hard clipping rectangle but also ensures the caller procedure can continue drawing with unchanged clipping, colour and font settings.
Saving and restoring the canvas state are fast operations if the *CanvasState* object is reused.

4.4.3 Drawing Primitives

The interface of the canvas object offers a small set of primitive drawing operations that is introduced in the following sections. All colour parameters that are used in the drawing operations can optionally be translucent.
All drawing operations offer a drawing mode parameter *mode* that specifies if the operation copies the source colour, depending on the operation either taken from the *color* parameter or from an image, or if the source colour should be blended with the existing content of the image. Possible mode parameters are *WMGraphics.ModeCopy* or *WMGraphics.ModeSrcOverDst*. The alpha blending mode *WMGraphics.ModeSrcOverDst* is for all operations significantly slower than the copy mode because the target pixels need to be read and blended with the source pixels. Section 4.3.1 gives more details on alpha blending.
While the drawing mode has no effect for solid colours it is important for translucent colours. The copying mode for translucent colours only makes sense if the target drawing buffer supports translucency, too. If a translucent colour is painted with the *ModeSrcOverDst* drawing mode on top of a translucency supporting bitmap that is filled with a solid colour, the result is again a solid colour. If the *ModeCopy* drawing mode is used, the destination bitmap becomes translucent.
For bitmaps, the alpha blending drawing mode *WMGraphics.ModeSrcOverDst* only makes sense if the image itself also contains an alpha channel. On the other hand, the *WMGraphics.ModeCopy* drawing mode for images with alpha channel only makes sense if the canvas itself operates on a buffer with an alpha channel.

Rectangular Filling

Filling a rectangular area is a common operation in GUIs. It is for example used to clear translucent bitmaps before drawing arbitrary shapes into them, or to draw background colours in panels or text editors. The *Fill* method takes a rectangle, a colour and a drawing mode parameter.
Example:

```
(* blend a translucent red rectangle on top of the existing image *)
canvas.Fill(WMRectangles.MakeRect(10, 10, 20, 20), WMGraphics.RGBAToColor(255, 0, 0, 128),
            WMGraphics.ModeSrcOverDst);
(* replace a rectangle of the existing image with a translucent green colour*)
canvas.Fill(WMRectangles.MakeRect(15, 15, 25, 25), WMGraphics.RGBAToColor(0, 255, 0, 128),
            WMGraphics.ModeCopy);
```

Images

Bitmaps are a very important design element in current GUI systems. Almost all visible elements of modern graphical user interfaces such as window frames, buttons, icons, scrollbar and so on are normally realised with bitmaps. They allow the look of the user interface to be changed without the need to change program code.
The canvas abstraction offers two different drawing procedures for bitmaps.

DrawImage The *DrawImage* method takes the position, the source image and the drawing mode as parameters. Images cannot be scaled.
Example:

```
(* Draw an image without alpha channel at position (10, 10) *)
canvas.DrawImage(10, 10, myImage, WMGraphics.ModeCopy);
```

ScaleImage The *ScaleImage* procedure is much more flexible than the *DrawImage* procedure but is also more complicated to use. Apart from the image, it takes a rectangular source region in the image that is drawn into a rectangular destination region of the canvas. If the rectangles are of unequal size, a scale operation is performed. When scaling the colours of the destination pixels need to be interpolated from the pixel colours of the source image. *WMGraphics* offers a simple box filter that takes the nearest pixel in the source image and a bilinear interpolation filter. The filter to be used is specified by the *scaleMode* parameter that can be either *WMGraphics.ScaleBox* or *WMGraphics.ScaleBilinear*. Bilinear filtering is significantly slower than the box filter because of additional memory and CPU overhead but results in a better result image quality. *ScaleImage* also supports alpha blending like all drawing primitives. For speed reasons, the combination of alpha blending and bilinear interpolation should be avoided in interactive scenarios. As a compromise of speed and quality, it is often useful to apply alpha blending with a box filter during interaction and switching to alpha blending with bilinear interpolation for refinement when the interaction is finished. This approach is taken in the display space manager (Chapter 5) to achieve smooth interaction.
Example:

```
(* Draw myImage at position 50, 50 reduced by 50% with a bilinear filter *)
canvas.ScaleImage(myImage,
                  WMRectangles.MakeRect(0, 0, myImage.width, myImage.height),
                  WMRectangles.MakeRect(50, 50,
                        50 + myImage.width DIV 2, 50 + myImage.height DIV 2),
                  WMGraphics.ModeCopy, WMGraphics.ScaleBilinear);
```

Text

The *DrawString* method draws a UTF-8 string at a given position. The y position specifies the base line (see 4.5.2). The font to be used can be specified with the *SetFont* method. The font colour is defined with the *SetColor* method. If no font was explicitly set, the system's default font is used.
For drawing text and rich text in interactive applications, the use of a *TextView* (see 3.4.1 for details) (sub-)component should be considered. In contrast to a string drawn with *DrawString*, a *TextView* component allows the user to copy the text, to execute commands and to apply tools, making the interface more flexible (See 2.4 and 3.2 for details).
Example:

```
(* Draw a string at position (100, 100) *)
canvas.DrawString(100, 100, "Hello World");
```

Lines

Lines in general are relatively rare elements in current graphical user interfaces. The special cases of horizontal and vertical lines can be found in menus and panels as horizontal or vertical separator elements or as the borders of rectangles. Arbitrary lines are used in CAD systems or diagram displays.
The *Line* method draws a line between two points on the canvas in a given colour that can optionally be translucent.
Example:

```
(* Draw a red line from (100, 100) to (200, 200) *)
canvas.Line(100, 100, 200, 200, WMGraphics.Red, WMGraphics.ModeCopy);
```

Polygons

WMCanvas offers two methods for drawing polygons. Both procedures take an array of 2d points as the first parameter to specify the polygon. The second parameter defines the number of points in the array that are to be used. This allows arrays of points to be reused with different numbers of points actually belonging to the polygon.
Both methods use the same rasterisation procedure. To disburden the garbage collector, the implementation of the rasterisation algorithm uses a self managed heap of memory that is associated with each canvas for storing rendering information.

FillPolygonFlat The *FillPolygonFlat* method fills a polygon with a single colour that can optionally be translucent.
Example:

```
(* Draw a red triangle *)
VAR points : ARRAY 3 OF WMGraphics.Point2d;
BEGIN
  points[0].x := 200; points[0].y := 20;
  points[1].x := 20; points[1].y := 380;
  points[2].x := 380; points[2].y := 380;
  canvas.FillPolygonFlat(points, 3, WMGraphics.Red, WMGraphics.ModeCopy);
```

FillPolygonCB The *FillPolygonCB* method, uses a call-back procedure for each horizontal line segment that needs to be filled. The *FillLineCallBack* is a delegate procedure that takes a canvas as the first parameter, the y position and the start and end position of the horizontal line segment that needs to be filled.
The following example fills a triangle with the content of an image:

```
VAR pict : WMGraphics.Image;

(* The filler procedure fills a segment on a scan line with the
respective content of the image, using the ScaleImage method *)
PROCEDURE Filler(canvas : WMGraphics.Canvas; y, x0, x1 : LONGINT);
BEGIN
  canvas.ScaleImage(pict,
     WMRectangles.MakeRect(x0, y, x1 + 1, y + 1),
     WMRectangles.MakeRect(x0, y, x1 + 1, y + 1),
```

```
     WMGraphics.ModeCopy, WMGraphics.ScaleBox)
END Filler;

PROCEDURE Triangle;
VAR points : ARRAY 3 OF WMGraphics.Point2d;
BEGIN
  points[0].x := 200; points[0].y := 20;
  points[1].x := 20; points[1].y := 380;
  points[2].x := 380; points[2].y := 380;
  canvas.FillPolygonCB(points, 3, Filler)
END Triangle;
```

The *FillPolygonCB* method offers a simple way to fill complex regions with complex contents.

4.5 Fonts

Fonts are an important and amazingly complex topic for a GUI framework. The task of the font system is to provide and render glyphs and glyph-metrices [1] as well as kerning information for characters of a defined font, style and size. Finding the correct glyph and kerning for a character depends on neighbouring characters and text layout.
Rendering glyphs for vector fonts is a difficult topic, that involves mathematical functions, filling algorithms, pixel alignment, hinting and even physical features of the output device [88] [13]. Since the rendering of a given glyph for a specific size, style and output device is computationally quite expensive, the system needs to apply caching strategies on this level.
Supporting international fonts with more than 82'000 glyphs, such as the CCG fonts by *eForth Technology Taiwan* [18], requires on-demand loading and caching of partial fonts. The internal font structure must also efficiently support access to non-continuous codepoint-ranges of glyphs.
For international text support, the font system needs to be able to efficiently and reasonably handle missing characters in fonts. One strategy is to search the missing character of one font in preferably similar looking other fonts that are available in the system.
All the, sometimes complex, activities of the font system need to be performed in real-time at lightning speed so as not to become the GUI's bottleneck.
The following sections describe the Bluebottle font system in more detail.

4.5.1 Abstract Font Interface

The Bluebottle graphics system defines an abstract font object class that is implemented by a number of specialised font implementations. The abstract font interface offers methods to query font and glyph metrics and to render a glyph as a bitmap.

4.5.2 Font Metric

Figure 4.4 shows the different font and glyph metrics that are provided by the Bluebottle font system. Other font systems use different font metrics and terminologies.
Vertical measures:

Baseline The *baseline* is the vertical reference point of a character. In normal writing, the baseline of all glyphs in a line of text are aligned.

Ascent The *ascent* defines the maximal amount a glyph without accent marks will ascend over the baseline. The value is contained in the *GlyphSpacings.ascent* record field.

Descent The *descent* defines the maximal amount a glyph of the font will descend below the baseline. The value is contained in the *GlyphSpacings.descent* record field.

Height The maximal glyph height is calculated by $height = ascent + descent$. The value is contained in the *GlyphSpacings.height* record field.

Top-Side-Bearing The *top-side bearing (tsb)* defines the empty space above the maximal ascent of a glyph. In Figure 4.4 it is denoted with the character *e*. The value is contained in the *GlyphSpacings.bearing.t* record field.

Bottom-Side-Bearing The *bottom-side bearing (bsb)* defines the empty space below the maximal descent of a glyph. In Figure 4.4 it is denoted with the character *f*. The value is contained in the *GlyphSpacings.bearing.b* record field.

Vertical Advance The *vertical advance v* defines the top to top distance of text lines. It is calculated by $v = tsb + ascent + descent + bsb = tsb + height + bsb$.

Horizontal measures:

Left-Side-Bearing The *left-side bearing (a)* defines the empty space left of the glyph. The value is contained in the *GlyphSpacings.bearing.l* record field. The value can be negative in cases of overhang or underhang.

Right-Side-Bearing The *right-side bearing (c)* defines the empty space right of the glyph. The value is contained in the *GlyphSpacings.bearing.r* record field. The value can be negative in cases of overhang or underhang.

Glyph Width The *glyph width (b)* contains the width of the visible character without overhang or underhang.

Horizontal Advance The *horizontal advance (d)* defines the horizontal distance that should be added before drawing the next character. It can be calculated by $d = a + b + c$.

4.5.3 Oberon Fonts

The fonts used in the Oberon System are stored in a relatively simple font format. Because of the large number of documents in Native Oberon and Bluebottle that use these fonts, an Oberon font implementation of the abstract Bluebottle font interface has been realised. The Oberon font support can load existing Oberon font files and use them in the Bluebottle GUI system outside of the Oberon environment. For compatibility reasons, the Oberon font file

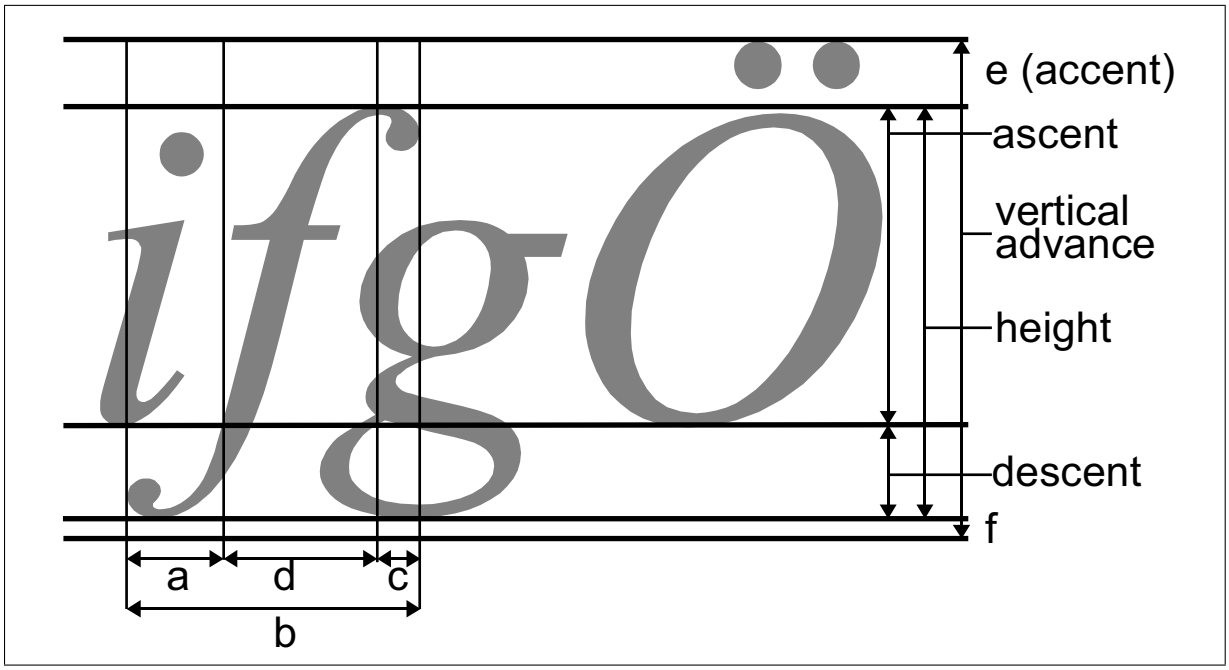

Figure 4.4: Bluebottle Font Metric

format was not changed. In contrast to the Oberon font system, the Bluebottle graphics system cannot use a one-to-one mapping of character codes to positions in the font, since the graphics system is based on Unicode and not on the Oberon character set.
While loading an Oberon font file, the contained *metric data* is converted to the Bluebottle glyph metric. To find an Oberon font, the font manager converts the given font name, size and style into the canonical font file name according the Oberon font naming convention.

4.5.4 OpenType Fonts

OpenType is a font format defined by Microsoft and Adobe that can contain glyph outlines in the *TrueType* or *Type 1* format. In contrast to TrueType or Postscript Type 1 fonts that require a font for every language, OpenType fonts support Unicode and can contain information for more than 65'000 characters. In 2004 many thousand different OpenType fonts are available which made the integration of the format into the Bluebottle system as an additional font plug-in attractive. The implementation is based on the offline TTF to Oberon font converter by E. Oswald [84]. It has been extended to support Unicode characters and dynamic loading of partial fonts. Partial on-demand loading of fonts is important in interactive applications because large Unicode fonts generally contain several thousands of glyphs and more than 10MB of font data that take several seconds to be loaded completely.

4.5.5 CCG Fonts

Traditional OpenType fonts are not very well suited to store glyphs of the group of Chinese, Japanese and Korean (CJK) languages in a space efficient way. The 60'000 most commonly used CJK glyphs require about 40MB of storage. While this size is acceptable for current desk-

top computers, it is by far too large for smaller devices such as PDAs or wearable computers. Making use of the highly structured composition of Chinese characters [60] [55], it is possible to store the same number of glyphs in a file as small as about one megabyte.
Each Chinese character is either a radical or contains one or more radical like elements. There are 214 unique radicals in traditional writing, 189 in the simplified form. Most parts of a glyph of a Chinese character can be drawn reusing radicals in different sizes and positions. Complex glyphs can be constructed by repeated re-use of radical elements or by the recursive use of other complex glyphs. To produce aesthetic complex glyphs, several different variations of the radical elements are needed.

Figure 4.5 shows the composition of the word *ming* (bright) out of the radicals *ri* (sun) and

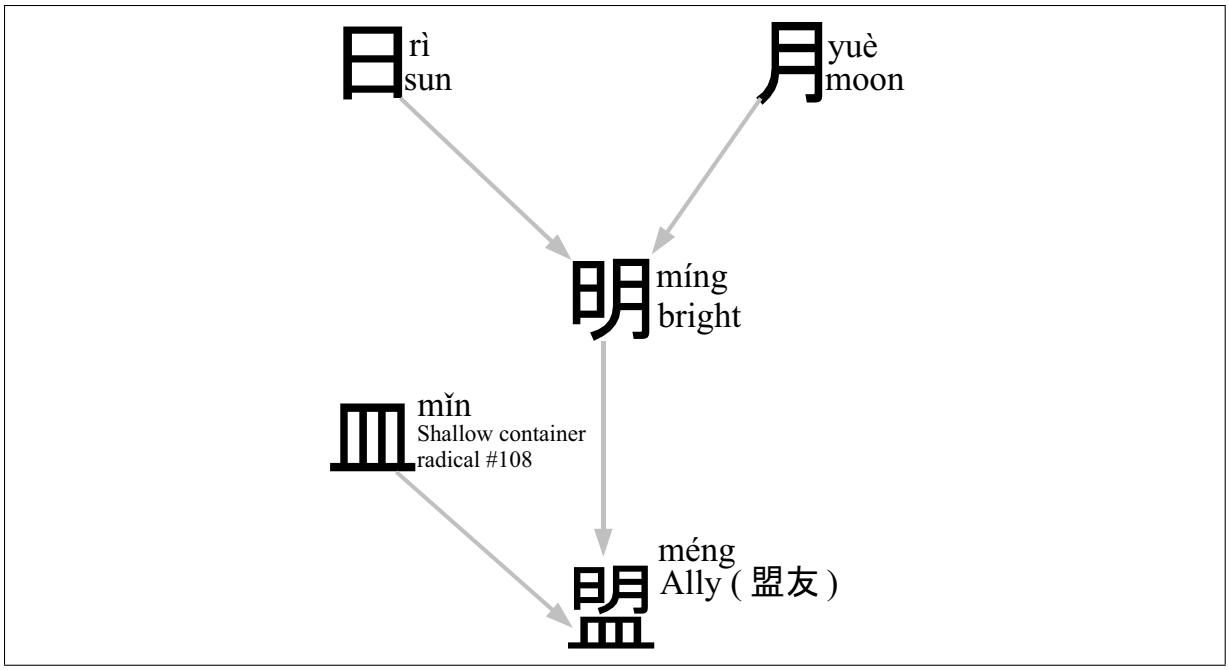

Figure 4.5: Character Composition

yue (moon). The word *meng* (ally) is stored as a combination of the word *ming* and the radical *min*.
The Taiwan based company *eForth* developed a font format and database, called *CCG font*, based on recursive composition of radicals and radical like elements. The font format can store the glyphs of more than 82'000 CJK characters in a file of about 2MB. This is a sufficiently small size for PDA like computer systems. *eForth* donated a single line stroke font with about 82'000 glyphs and an outline font with about 27'000 glyphs to the Bluebottle project based on a free license.

4.6 Evaluation

The Bluebottle graphics system does not take advantage of special graphic hardware acceleration for drawing. The main reason is the lack of documentation of current graphics hardware.

Given proper documentation it would still not be possible for a small computer systems research group to keep up with the fast pace of development in this area.
Having no hardware acceleration leads to the question of whether the GUI system can provide sufficiently fast graphics for the 2d desktop. In the following we measure and compare in Bluebottle and Windows XP the speed of a number of graphics operations that are important for current graphical user interfaces. Due to the large conceptual differences between the Bluebottle display space manager and graphics system and the Windows XP GDI [75] (Graphical Device Interface) (see 5), the measured numbers cannot be directly used to quantitatively compare the speed that is apparent to the user. The numbers should only serve for a rough qualitative comparison.
The main obstacle for a quantitative analysis lies in the buffering mechanism that is used in the Bluebottle display space manager that allows asynchronous reading and display of a window while it is being modified by an application process. This buffering mechanism allows the display space manager to skip frames in a consistent way that would have to be rendered to the display in a system without a similar mechanism. In Windows XP as well as in Bluebottle, the time is measured as the time the application process is waiting until the graphics system returns from the drawing operation. In both cases this is not necessarily the time it takes until the drawing finally appears on the screen.

For the measurement we use graphical operations that are typically used in today's desktop application software. The compared operations are:

- Rectangular filling as used in panels or for clearing the background in a text editor
- Drawing of bitmap images (non-resized, non-blended) as used for symbols, buttons and decorations
- Horizontal lines

Writing of text has not been compared. It is assumed that the font system uses glyph caches resulting in a speed proportional to drawing bitmaps.

The drawing operations are performed and measured with different parameters, varying the numbers of pixels that have to be changed. In one test run, the drawing operation is performed 10^6 times with the same parameters and measured in milliseconds by the system timer. Per test parameter, five runs are performed, dropping the highest value and averaging the remaining times. In the Windows XP system, colours, pens, brushes and images are created outside the measurement loop to avoid counting setup overhead that could be related to the more feature rich Windows GDI. Changing colours in the Bluebottle system does not influence the timing of the measured operations. In the Windows system, all other application programs are closed before performing the measurement and the network is disabled. In both systems, the measured application program runs with normal priority. In the following we measure and compare the speed of graphical operations for four different setups:

1. Bluebottle, 16 bit VESA 2.0 mode.
2. Microsoft Windows XP with full hardware acceleration enabled, 16 bit colour mode.
3. Microsoft Windows XP with hardware acceleration disabled, 16 bit colour mode.
4. Microsoft Windows XP with full hardware acceleration enabled, 32 bit colour mode.

In all setups write combining as described in 4.2 is enabled. All the measurements have been repeated in all four setups on different machines with similar results. The numbers used in the following are generated on a *NV34 GeForce FX 5200* AGP 8x graphics card in a *Shuttle SB61G2* board with 1GB 2xDDR RAM and a 3.0GHz Pentium 4 HT processor. Windows XP was running with *Service Pack 2*.

Diagram 4.6 shows the time needed to draw homogeneously filled rectangles of different size. The rectangle in this measurement is always 64 pixels in height and grows in the width. The x-axis of the diagram shows the total number of pixels in the image. In the filling test a 2d hardware accelerator can be most efficient since a large number of pixels have to be filled without the need to get additional data from the main memory. The relatively high constant overhead of 3.7ms in the Bluebottle filling algorithm results from the search for the appropriate filling strategy and the procedure call overhead per horizontal line.

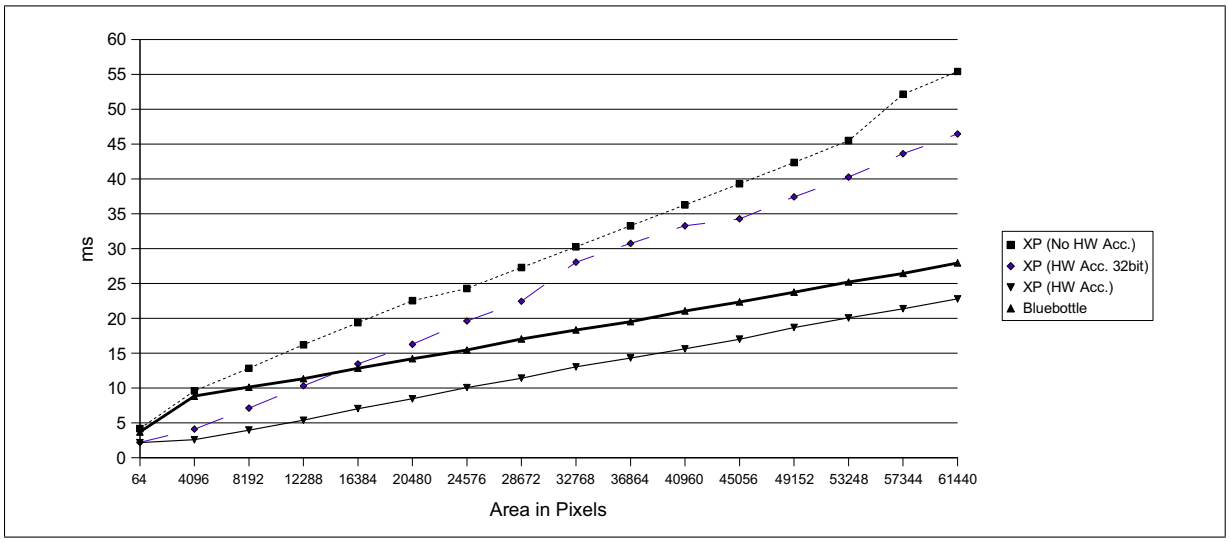

Figure 4.6: Diagram : Filling Rectangles

Diagram 4.7 shows the time needed to draw horizontal lines. While the measurements without hardware acceleration show a clear linear relationship between the line size and the drawing time, no such relationship is visible with the hardware accelerator. It seems to complete the task in constant time although with a high setup overhead. The measured times in Bluebottle stay below the setup time of the hardware accelerated system for the measured line sizes.

Diagram 4.8 shows the time needed to draw a bitmap image in the video card's colour mode, so that no colour conversion is needed. The image in this measurement is always 64 pixels in height and growing in the width. The x-axis of the diagram shows the total number of pixels in the image. It is interesting to note that in Bluebottle the time for drawing images in

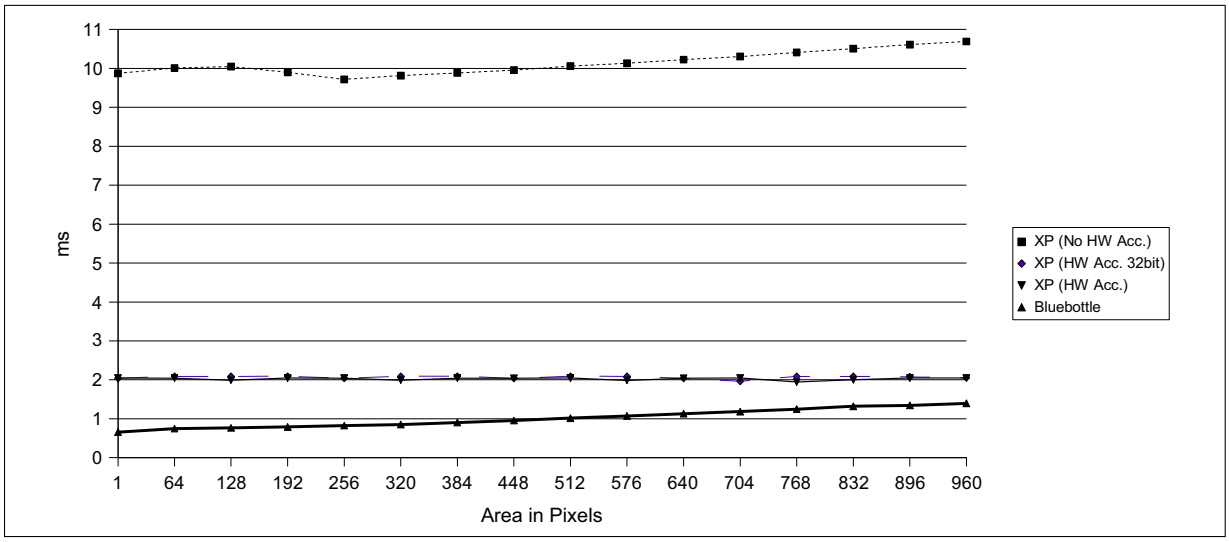

Figure 4.7: Diagram : Drawing Horizontal Lines

the range of 64 to 4096 pixels always stays well below the time it takes the graphics accelerator with Windows XP. Most icon images in current GUI systems are below 64x64 pixels. Diagram 4.9 shows the measurements for small images in more detail.

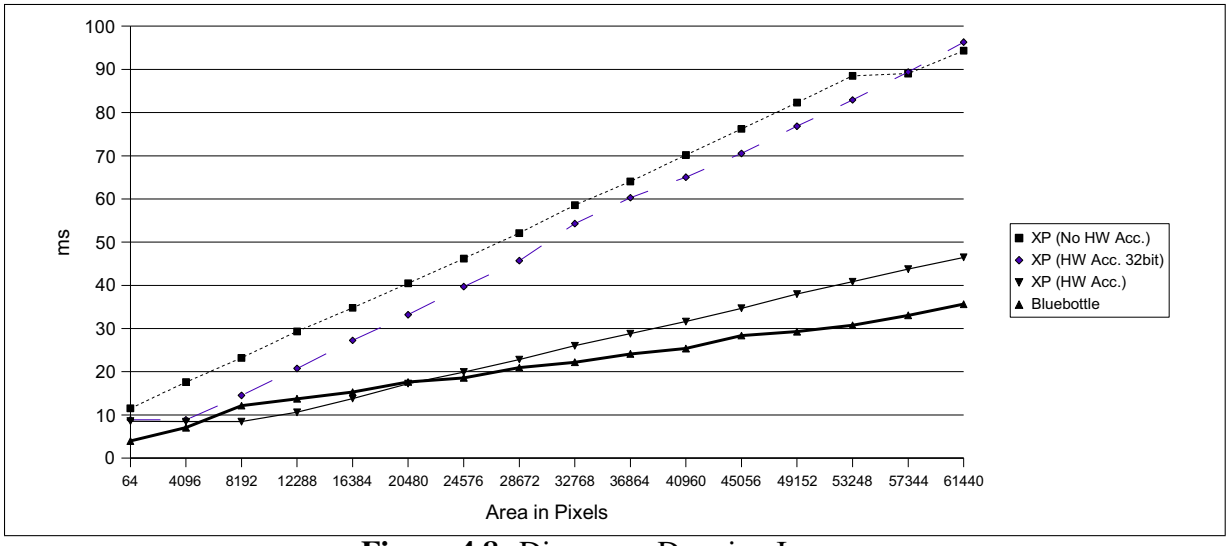

Figure 4.8: Diagram : Drawing Images

The diagram shows a constant overhead for initialisation of the graphics system i.e. the hardware graphics accelerator. Once initialised it can draw an image of up to about 16k bytes without taking more time. While the unaccelerated setup results in a linear graph from the beginning. To get a better understanding, the same experiment was repeated with a 32bit video mode, resulting in the same setup overhead but as expected only half the number of pixels that can be drawn without needing additional time. The per pixel time in the 32bit mode is twice as big as in the 16bit mode, again no surprise since the limiting factor for drawing the image is the memory bandwidth.

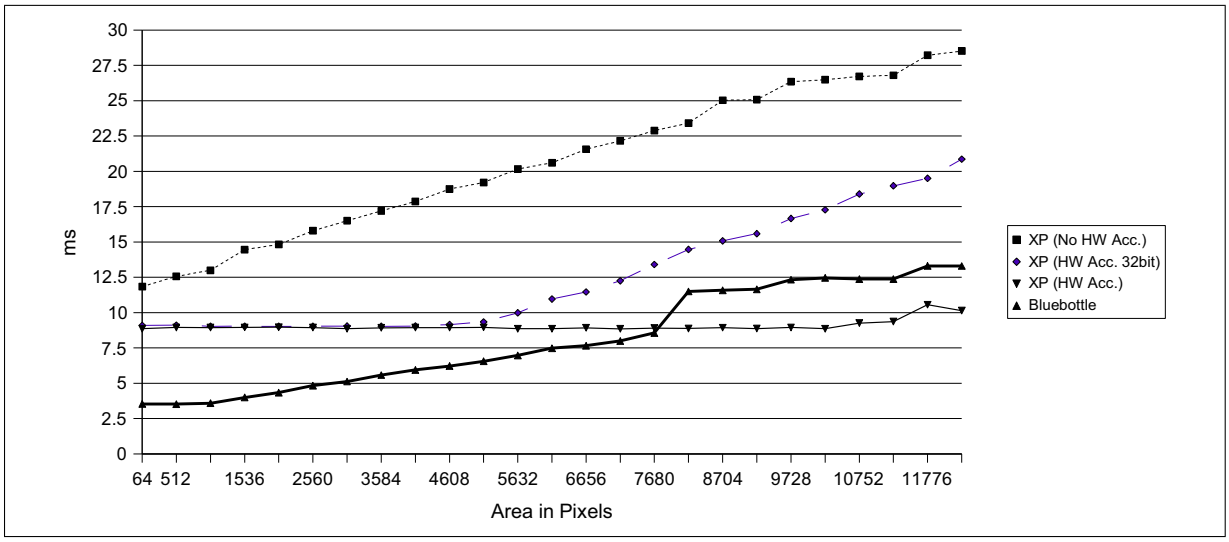

Figure 4.9: Diagram : Drawing Small Images

4.6.1 Conclusions

The measured timings for graphic operations that are relevant to the efficiency of PARC style graphical user interfaces show the competitiveness of the Bluebottle graphics system in this respect.
Even though the measured timings for Bluebottle represent the time of drawing into an in-memory buffer, we can still conclude the overall graphic system performance for GUI applications is sufficient, because :

- Drawing into a window buffer is as efficient or better than the compared system for small and medium areas that are typical for many GUI components.
- In application programs, the window buffer normally only needs to be updated to the hardware framebuffer after a series of several consecutive drawing operations such as clearing, drawing lines, glyphs and images so that the asynchronous time overhead that was not included in the measurement does not carry too much weight.
- The Bluebottle display model makes it possible for the display space manager to skip updates to the hardware framebuffer if there is not enough time available to render all states, without resulting in inconsistent displays. Sections 5.2.1 and 5.3.3 discuss the display consistency and in-memory buffer mechanism in detail.

5

Display Space Manager

Imagination is more important than knowledge

— Albert Einstein (1879 - 1955)

5.1 Introduction

The *display space manager* is responsible for managing *display space object*s and *viewport*s on a virtual desktop, called the *display space* as introduced in section 2.6. This includes establishing and maintaining consistency between the individual windows and their representations in the different viewports, dispatching messages to individual objects in the display space as well as managing insertion, removal, moves, resizes and changes in overlapping of these objects.
Display space objects can be arbitrarily shaped and optionally translucent. Rectangular display space objects are referred to as *windows*.
A *viewport* is a program that observes all or parts of the virtual desktop. The most obvious application for a viewport is to copy (parts of) the desktop to a physical screen. Other *viewports* are remote network framebuffers for example VNC [96] [97] or screen capture utilities.
The display space manager operates asynchronously to the drawing into individual display space objects. From the programmer's point of view, each regular display space object appears as a separate framebuffer with its own keyboard and pointing device.
The following sections explain the model and implementation of the display space manager in more detail.
Section 5.2 introduces the display space. Section 5.3 introduces display space objects. Section 5.4 discusses viewports.

5.2 Display Space

The bluebottle display space is a conceptually unlimited two dimensional coordinate space (Fig. 5.1 and also Fig. 2.4) in which an arbitrary number of windows and other desktop objects can be situated. In the practical implementation, the display space is limited to the range of 32

bit signed integers.
Orthogonal to the display space objects, a number of viewports can be installed that are observing ranges of the display space. The display space manager informs all the registered viewports about all the changes that occur in the regions they are subscribed to. Section 5.2.1 gives details on the mechanism that establishes display consistency.

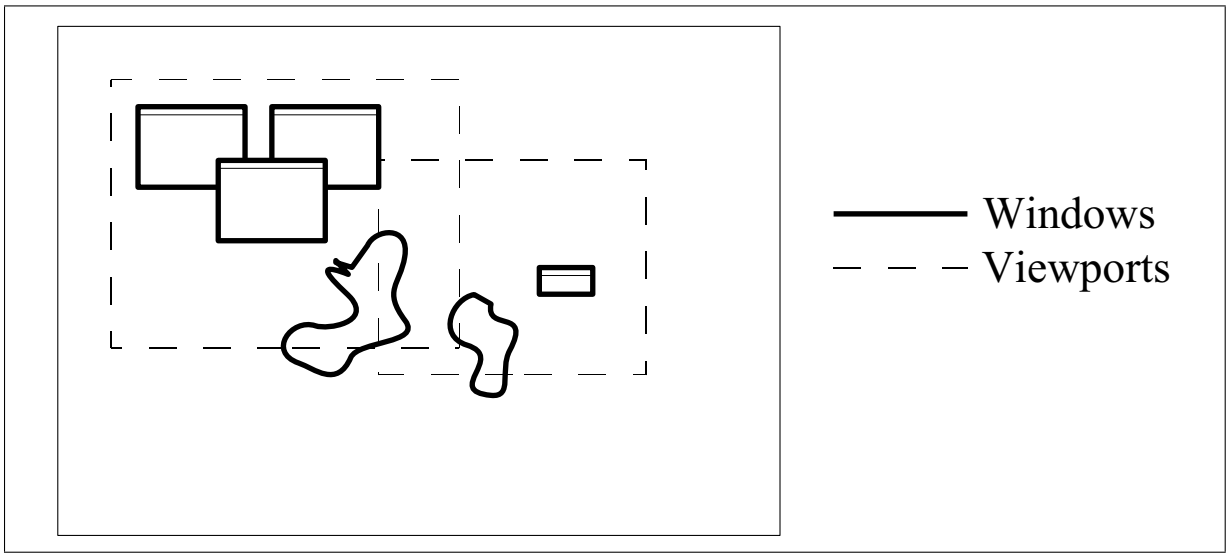

Figure 5.1: Display Space

5.2.1 Display Consistency

The display space manager offers the methods *AddDirty* and *AddVisibleDirty* for reporting invalidated rectangular display space areas. The reported areas are queued in a *dirty-queue* that buffers the invalidation messages. Arriving update requests are compared to all the previously queued invalidated areas that have not yet been refreshed. If possible, the new invalid region is merged with overlapping or adjacent regions in the queue, waiting to be updated. Repeated invalidation messages for the same region are dropped if the previous invalidation message has not yet been handled. Clipping against previous update requests in the queue reduces the impact of an application flooding the display space manager with redraw requests.
The *AddVisibleDirty* procedure checks if the reported area is visible in the reporting display space object. If the area is completely or partially covered by one or more solid display objects, the regions of the solid display objects are subtracted. This is achieved by recursively breaking up the area along the bounding boxes of the covering solid objects, into rectangular pieces, that are only added to the *dirty-queue* if they are visible.
Should the *dirty-queue* ever overflow, the entire display space is invalidated and the queue cleared.
The display space manager itself uses the *AddDirty* and *AddVisibleDirty* methods when objects or views are added, removed, resized or moved.
As long as the *dirty-queue* contains invalidated areas, the display is said to be inconsistent. A special display space manager activity is responsible for re-establishing consistency. To do

this it takes invalidated rectangular areas from the queue and asks the observing viewports to redraw the area in question.

5.3 Display Space Objects

Display space objects are arbitrarily shaped objects that can be inserted into the display space. The position and extents in the display space are managed by the display space manager. The display space manager uses the bounding box of an object to perform early clipping of position-dependent messages and drawing operations. The visible shape of the object is determined by its *Draw* procedure. The display space manager ensures that no object can draw outside its official bounding box. The shape of the clickable region of the object is dynamically determined by calling its *IsHit* procedure.
Each display space object has its own coordinate system relative to its bounding box's top left position (Fig. 5.2). The orientation of the coordinate system is chosen for compatibility with existing hardware framebuffers and protocols.

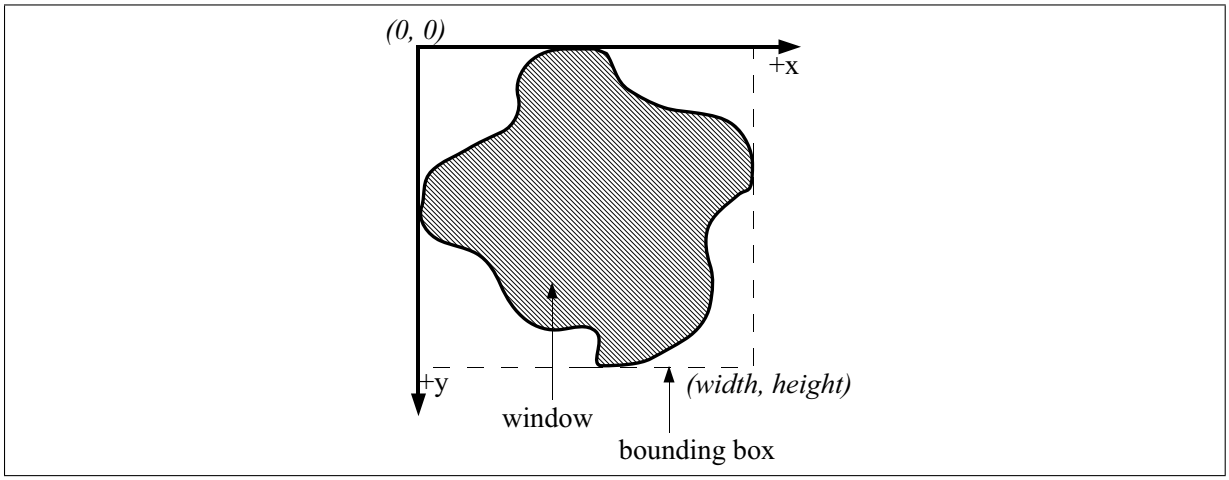

Figure 5.2: Local Display Space Object Coordinates

5.3.1 Basic Interface

This section describes the most important methods in the generic display space object interface that has to be implemented by all desktop objects. The abstract interface is defined in the *Window* object class. Predefined and specialised implementations of the *Window* interface exist that free the programmer from arduous details for most common applications. The predefined implementations *BufferWindow* (5.3.2), *DoubleBufferWindow* (5.3.3) and *FormWindow* (6.3) are described in later sections. The *raw* interface only needs to be implemented for very specialised applications[1]. In the standard Bluebottle system there are only three types of display space objects that are not either a *BufferWindow*, a *DoubleBufferWindow* or a *FormWindow*:

- The object that fills the entire display space with a uniform background colour.

[1]Programmers should avoid directly implementing *window* and rather use a predefined extension.

- The decorative frames that form the borders of rectangular standard windows.
- The backdrop images.

The reason for the special implementation of these common objects is to save resources. All other display space objects, including the mouse pointer, are implemented as regular *BufferWindow*s or extensions thereof.

IsHit The method *IsHit* is used for fine grained determination of the object's shape. It is called by the display space manager to determine the recipient of pointer messages. The implementation of the method returns *true* if it wants to own the position *(x, y)* relative to the object's local coordinate system. It then receives pointer messages when the mouse cursor is on top of this position given it is not hidden by another desk space object. The method is only called by the display space manager if the pointer lies in the rectangular bounding box of the object. See 5.5.2 for more details.
The predefined implementations of *BufferWindow*, *DoubleBufferWindow* and *FormWindow* automatically handle the *IsHit* request and reply with *true* if the respective position is visible. For translucent objects, the visibility is determined by comparing the α-channel value of the respective buffer position to an α-threshold stored in the *pointerThreshold* variable of the *BufferWindow*. In the solid case, the method always returns *true*.
In the current implementation, the *IsHit* method is directly called from the display space manager process to avoid slowing down user response. It must therefore be carefully implemented so as not to block or trap the display space manager. Section 10.2 describes the conceptual problem in more detail and gives a possible remedy.

Invalidate The *Invalidate* method invalidates a rectangular region *rect* given in local display space object coordinates. The method transforms the rectangle into global display space coordinates and calls the *AddVisibleDirty* method in the display space manager. The display space manager then clips the area into zero, one or more rectangular pieces that are potentially visible, and queues them for restoring the display consistency.
The default implementation of the *Invalidate* method generally does not need to be replaced.

Draw The method *Draw* must be implemented by the object and repaint itself upon asynchronous request with the new scaled width *w* and height *h*. The parameter *q* is a quality hint that is set to 0 if quality does not matter and speed is of most importance. It is mainly used to set lower filter qualities when zooming or moving large areas of the display to ensure interactive speed.
The region of the area that has to be redrawn is set as the clipping rectangle of the canvas. The drawing routine can possibly save computations by not drawing elements outside the clipping rectangle. Atomic canvas operations are efficiently clipped in the graphics systems (see 4.4.1) so that the application programmer does not need to adjust their sizes to the clipping region in order to save time.

The predefined implementations of *BufferWindow*, *DoubleBufferWindow* and *FormWindow* automatically handle the call to *Draw* and paint a buffered image of the requested area.

Handle The *Handle* method handles messages from its associated *sequencer* thread. *Handle* decodes the messages and forwards them to predefined local handler methods. This is done for the convenience of the programmer. For example the close message is automatically forwarded to the *Close* method that by default removes the respective object from the display space manager. It is always possible for implementations of *Window* to handle messages directly by overriding the *Handle* method.
The following methods are predefined and called from the message handler for the convenience of the programmer:

- *PointerDown(x, y : LONGINT; keys : SET) (x, y)* is the position in display space object coordinates, *keys* is the set of pointer keys that are pressed.
- *PointerMove(x, y : LONGINT; keys : SET) (x, y)* is the position in display space object coordinates, *keys* is the set of pointer keys that are pressed.
- *PointerUp(x, y : LONGINT; keys : SET) (x, y)* is the position in display space object coordinates, *keys* is the set of pointer keys that are pressed.
- *PointerLeave* is called if the pointer leaves the display space object without pressed pointer keys.
- *WheelMove(dz : LONGINT)* is called when a scroll wheel is moved while the pointer is on top of the display space object. The parameter *dz* contains a signed value that represents the number of ticks the wheel was moved up or down.
- *DragOver(x, y : LONGINT; dragInfo : DragInfo)* is called when the pointer of an explicit drag operation is over the object.
- *DragDropped(x, y : LONGINT; dragInfo : DragInfo)* is called when an explicit drag operation ends over the display space object.
- *KeyEvent(ucs : LONGINT; flags : SET; keysym : LONGINT)* is called if a key is pressed while the object has the keyboard focus.
- *FocusGot* is called when the display space object receives the keyboard focus.
- *FocusLost* is called when the object loses the keyboard focus.
- *StyleChanged* is called if the system wide style settings have been changed.
- *Resized(width, height : LONGINT)* is called after the display space manager assigned a new size to the display space object.
- *Close* is called when the standard close button of a rectangular window is pressed.

Application programs can extend the set of supported messages by using the *ext* extension field of the generic message record. The set of predefined messages does not use the extension mechanism so as not to allocate dynamic memory in common situations like mouse pointer movements.

Resizing The *Resizing* method is called directly from the display space manager before the object size is changed. Variable parameters *width* and *height* contain the suggested new extents of the bounding box. The object can change the parameters to values that are better suited for the application. The last word lies with the display space manager that will notify the display space object of its new size with the *Resized* message.

SetTitle The *SetTitle* method is used to set the title of the object. In the case of standard windows with a decorative border, the name is displayed in the title frame.

SetPointerInfo The *SetPointerInfo* method is used to set the cursor shape that should be used when the pointer is on top of the object. The display space manager defines a number of style dependent standard cursors that can be used by application programs. If needed, additional special cursor shapes can be defined by the application program.

5.3.2 Buffered Display Space Object Interface

The buffered display space object completes the abstraction of a virtual screen in offering a local frame buffer. The respective programming interface is called *BufferWindow*. The extents of the buffer are given in the constructor. The parameter *useAlpha* in the constructor, defines whether the local frame buffer contains an alpha channel or not. In the case of an alpha frame buffer, the bitmap format *RGBA8888* (red, green, blue and alpha channel consist of eight bits each) is chosen, otherwise the system chooses a non transparent RGB format. The preferred bitmap format equals the native format of the hardware framebuffer thus avoiding the need of colour format transformations when transferring images to the hardware.
Requests from the display space manager to resize a buffered display space object are handled by the *BufferWindow* implementation in scaling the buffer in the draw method. The program using the *BufferWindow* does not notice the change of the object's size, and its internal resolution remains unchanged. All position dependent pointer messages to and invalidation messages from the application are automatically transformed to the respective coordinate spaces.
The visible shape of the display space object is defined by the alpha channel of the local framebuffer. The clickable shape of an object is defined by a threshold value that is compared to the alpha channel value at the given position. The alpha channel values of 0 and 255 are specially optimised and act as a stencil buffer in the *Draw* procedure.

5.3.3 Double-Buffered Display Space Object Interface

A single frame buffer can successfully hide modification artifacts such as flickering in a situation where only solid display space objects are involved and the objects are not asynchronously

moved or resized since the display space manager only needs to read the buffer when it is declared as invalid by the application. The application has to wait until the updates have been performed by the display space manager before changing the content again.
Allowing translucent objects and more than one process in the display space requires asynchronous read-outs of buffers, even of solid display space objects, that have not been declared as invalid. The asynchronous read-outs occur in cases where an object is covered by a translucent object that has modified its content or that is moved or resized, or vice versa if an object below a translucent object is changed. Without strictly synchronising all the viewports and display space object updates, flickering occurs if a frame buffer is in a state of modification. Synchronising all viewports and objects is not a viable solution because it severely reduces the possible concurrency.
An improved solution uses two off-screen copies for each display space object, one to be used by the drawing client process and the other by the display space manager. The buffers are swapped when a consistent state is reached. Figure 5.3 shows an object with two buffers *A* and *B*. Each of the buffers is only used by either the display space manager or the application program. When a consistent state is reached, the references to the buffers are swapped atomically. This mechanism can successfully hide all updating artifacts. It can however not

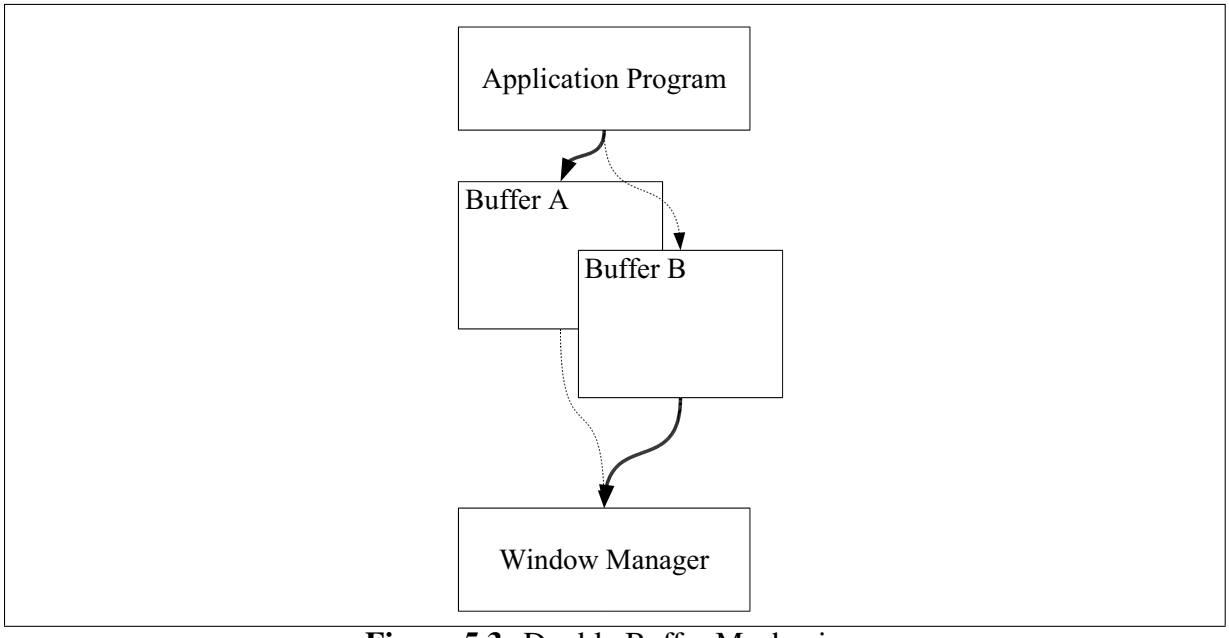

Figure 5.3: Double Buffer Mechanism

completely avoid artifacts of time, where one part of a buffer is used in the process of restoring the consistency of another object, before its own invalidation message could be handled. This could be avoided by tightly coupling the *dirty-queue*(5.2.1) of the display space manager to the buffer swapping mechanism in the display space object. In the current implementation, this is not done for mainly two reasons:

1. The time artifacts are very rare and barely noticeable unless the system is under too heavy load. The artifacts are normally corrected in less than 20 milliseconds until the next frame refresh.

2. To keep the *dirty-queue* implementation clean and understandable.

There are two significantly different ways as to how a display space object updates its contents and, correspondingly, two different buffer update strategies are required:

- The content is always completely replaced. This is for example the case in movie players, slide shows or in 3d applications where a changing camera position changes the perspective and invalidates the entire scene.
  Whenever a consistent drawing state is reached, the roles of the buffers are exchanged by simply switching pointers. The entire area of the object is then declared invalid in the display space manager.

- Only small parts of the content are updated. This is the normal behaviour of many GUI components.
  Whenever a consistent drawing state is reached, pointers are switched and the updated region is copied to the second bitmap, to make it consistent for the next update operation. Only the small updated area is invalidated.

Both double-buffering strategies are implemented in the *DoubleBufferWindow*. The first case is handled by the *Swap* method, the second by the *Update* method which takes the rectangle to be updated in display space object coordinates as a parameter.

5.4 Viewports

A viewport is an observer of an axis-aligned rectangular area in the global display space. It usually represents a physical display but can also represent a screen shot utility or a remote frame buffer such as VNC.
Viewports implement the methods *Update* and *Refresh* that are called by the display space manager when it detects an inconsistency in the observed area. The detection of inconsistencies is described in more detail in 5.2.1.
Calls to *Update* and *Refresh* are synchronised with changes in the display space structure, such as display space object positions, sizes and overlapping, so as to ensure fixed positions of all display space objects involved while updating. This synchronisation does not include the contents of the different objects. Viewports should handle updates efficiently so as not to lock the display space structure for too long. Viewports with inherently high update latencies should therefore use a buffering technique that allows updates to be processed asynchronously. The VNC viewport, for example, needs to do image processing, encoding and network transport and thus inherently has a high latency. Buffering of update requests allows uninterrupted local work on a machine that is serving a VNC viewport around the world with network delays of more than 350 milliseconds.
Whenever the viewport is notified about an inconsistency, it redraws the respective area. The most efficient way to display a region of the coordinate space in an observing view is an identical pixel to pixel mapping, including colour depth conversions where needed. However, the speed of current hardware allows more challenging mappings, including zooming and filtering.

The default implementation of a viewport for physical displays can observe an arbitrary axis-aligned rectangular region of the display space, resulting in a zoomed, possibly even distorted, representation of the display space.
Viewports can also act as message generators for the display space manager. When a mouse or keyboard that is associated with a viewport generates an event, the viewport forwards it to the display space manager's asynchronous message queue as described in 5.5. Messages originating from a viewport have the viewport noted in the message's *originator* variable.
Viewports are added to the display space by registering them to the display space manager with the *AddView* method.

5.4.1 Drawing Mechanism

When a region *r* of the display space needs to be redrawn, the *Update* method of the viewport is called by the display space manager process that is responsible for restoring display consistency.
Within the region *r* in the display space, there can be many display objects that can overlap each other and even be translucent. When drawing a translucent display object, all the visible underlying objects need to be drawn first and the result mixed with the image of the covering translucent object. If translucent objects are stacked on top of each other, the drawing must start with the lowest object. Solid display objects block the view further down in the z-order, and therefore anything below a solid object does not need to be drawn.
The simplest correct solution to restore the area is to set the clipping region of the viewport canvas to *r* and to draw all the objects from the bottom to the top. While obviously correct, this solution is not efficient because it does not take advantage of solid objects hiding objects below them. An efficient solution never draws a solid object over an area drawn before in the same update operation; each pixel is written only once with a solid colour.

To optimise the drawing, the *Update* method therefore first splits the region *r* into smaller sub regions that can be drawn in a uniform way in terms of overlapping translucent objects. This is done by recursively checking the region *r* for intersecting objects on the display space starting from top to bottom along the *z-order*. If an intersection is found, the region is split into two up to five smaller regions that are then recursively processed. The recursive process stops as soon as the considered subregion is completely blocked in the z-order by a solid object as, for example, the background. Before going back one level in the recursion, the respective object is drawn. This algorithm efficiently makes use of solid objects that hide other objects, to reduce the number of pixels that are drawn. The individual steps of the recursive break-down algorithm are visualised with an example situation in figure 5.4. In the first step, the solid top top is cut out of the update area and is directly drawn. The second object is cut out of the four remaining regions in the second step. Because the second object is translucent, all eleven remaining elements need to be clipped against the third object. When the algorithm terminates it has split the beginning region *r* into 23 segments and drawn them individually.

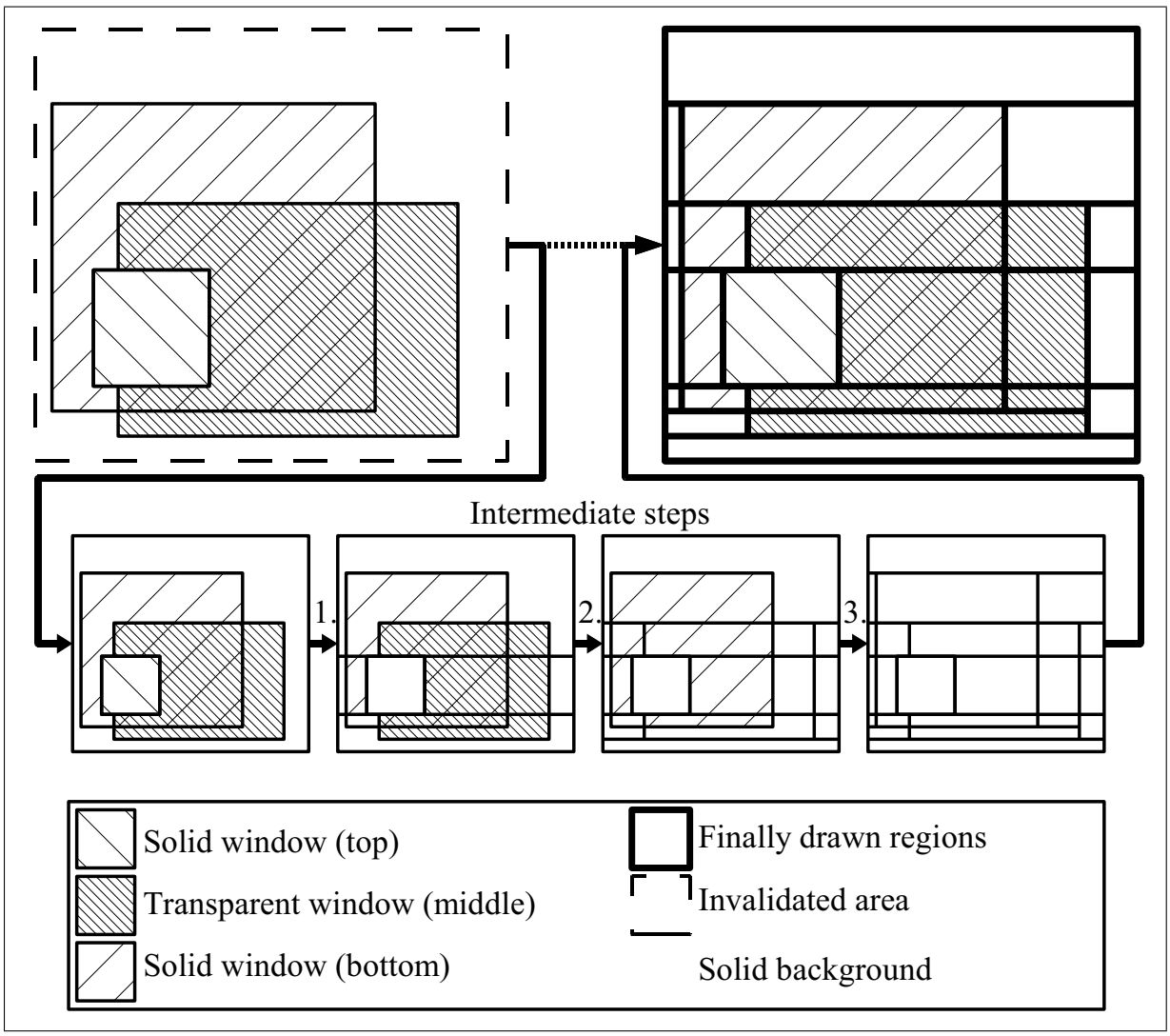

Figure 5.4: Recursive break-down of the redraw area

5.5 Message handling

Asynchronous external events such as mouse or keyboard events and messages that originate from other activities are put into the message queue of a *sequencer* object as described in 6.2.4 that is associated with the display space manager. The sequencer object provides synchronisation and thread-safety and delivers the messages sequentially to the *Handle* method of the display space manager.
The display space manager supports directed and broadcast messages. Directed messages are sent to a single dedicated recipient object in the display space. Broadcast messages are sent to all objects in the display space. Pointer or keyboard messages are directed whereas style change notifications or desktop store requests are examples for broadcast messages.
Figure 5.5 gives an overview of the message flow from the input devices to message consuming application programs.

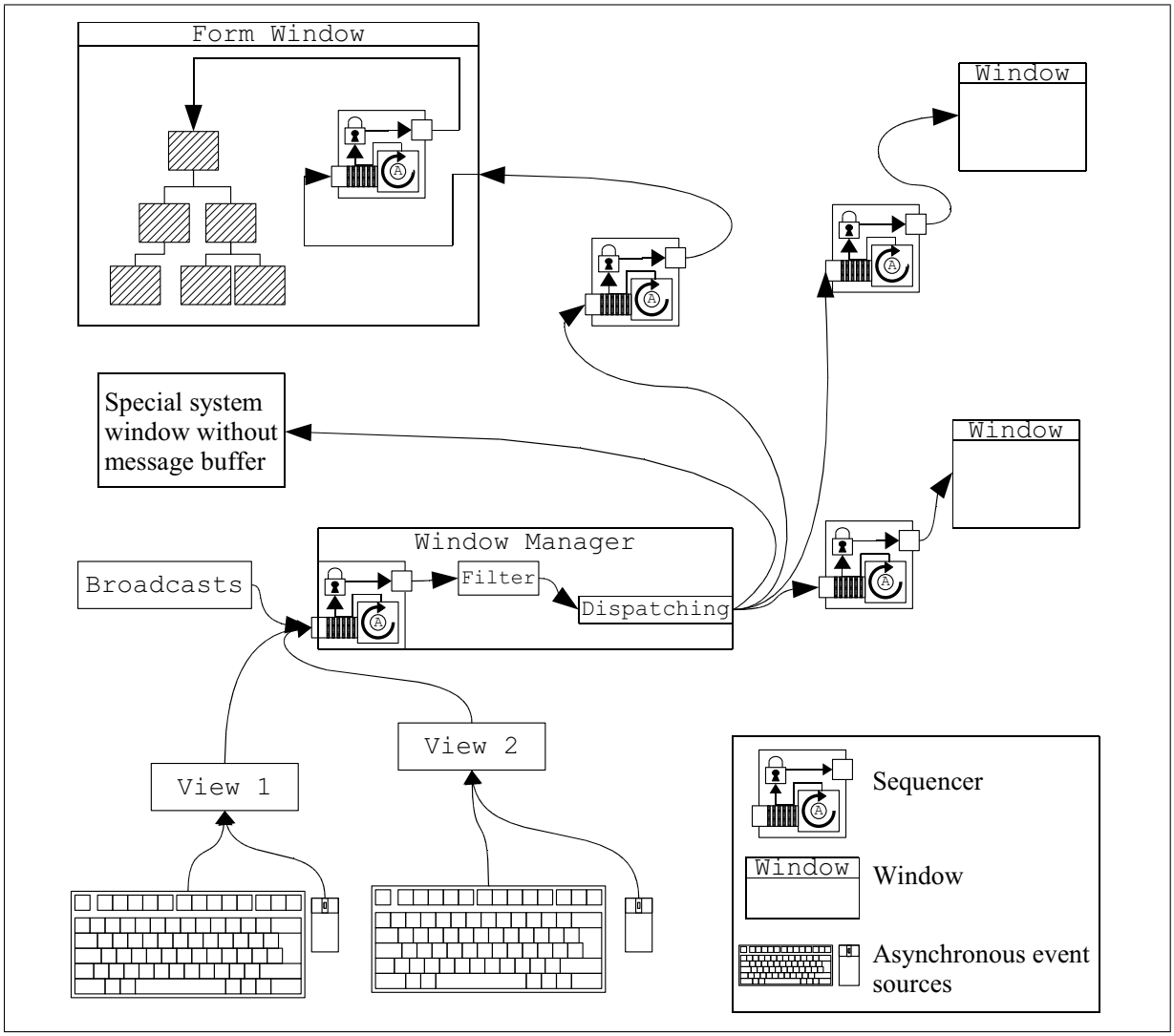

Figure 5.5: Message Flow Overview

Before forwarding any messages to one or all objects, the display space manager calls possibly installed message filter methods that can modify or discard the messages. The message

filters can be used for example to create system wide keyboard shortcuts. For example the system menu installs a message filter to check for the keyboard shortcut *meta-esc* on which it moves itself to the front of all display space objects and into the range of the display space that is observed by the viewport that generated the *meta-esc* keyboard message. Message filter methods are called synchronously by the display space manager and therefore should never raise an exception or delay the message for too long.
If no message filter discards the message, the display space manager adds it to the message queue(s) of the recipient object(s). The message queue of an object is again implemented with a *sequencer* object. With the exception of some special purpose system objects such as the mouse pointer, all display space objects have an associated *sequencer*. The sequencer acts as a message dispatcher process that decouples the display space objects from the display space manager through a message queue. The decoupling is needed to prevent an object from blocking the display space manager in its message handler method. Apart from preventing blocking, the decoupling also acts as a way of exception handling. A faulty message handler that raises an exception does not affect the display space manager but only terminates the object's sequencer.

5.5.1 Keyboard Events

Keyboard messages are directed to a single object that owns the keyboard focus. If no object has the keyboard focus, the message is discarded.
The keyboard focus is either transferred explicitly by a call to the display space manager, or implicitly by the rules of the display space manager. The default implementation of the display space manager implicitly transfers the keyboard focus to the object that was last clicked at with the mouse, so the object that owns the focus is not necessarily the same as the object that contains the mouse pointer.
The keyboard message contains:

- a key code identifying the key that generated the message
- the state of the grey keys such as *ctrl*, *shift*, *alt* and *meta*
- a flag that informs whether the key was pressed or released
- a translation of the key code to a Unicode codepoint, if applicable. Note: although simple input mode editors could in theory operate on the level of keyboard messages this is not possible for more sophisticated implementations because of the lack of context on this level. Section 3.4.3 describes the implementation of input mode editors at the appropriate level of the text system.
- a reference to the viewport that generated the message

5.5.2 Pointer Events

Pointer messages originating from a mouse or other pointing device are sent to the object that is responsible for the pointer handling. Determining the responsible object is a more complicated topic than finding the recipient of a keyboard message. It requires a set of state dependent rules and also locating objects in the display space.
The simple strategy of always taking the topmost object at a given display space position is insufficient because it loses the pointer ownership too early in drag operations between different objects. Examples of *drag* operations that need a longer pointer ownership are handwriting recognition, gesture detection and obviously drag and drop copy or move operations. These operations characteristically start at a certain position, continue over a period of time and end at a position that is unknown at the beginning of the operation. The end position in drag and drop operations is usually important and therefore exactly chosen by the user, it may or may not lie in the same display space object. In gesture input operations, the end point is less important and the operations are supposed to happen within the same object. Expecting the user to keep the pointer within the limits of the display space object when inputting a gesture is not practicable since it requires the user to carefully pay attention to the pointer, exactly the opposite of what gesture input wants to achieve. The problem is aggravated if the object's shape is non-regular.
Defining the *drag* operations more precisely, directly leads to a solution : "a drag operation is a pointer movement that is started with a certain button being pressed in the beginning and released in the end". The display space manager can easily recognise this general scheme and forward all messages of one *drag* operation to the object in which the operation was started.
To determine the object that contains the pointer, the display space manager traverses the list of display space objects from top to bottom in z-order and checks if the pointer position lies within the object's bounding box. If yes, it calls the object's *IsHit(x,y)* method that then reports at runtime whether it wants to be responsible for the position or not.
If the pointer leaves an object with no pressed buttons, the display space manager sends a *PointerLeave* message to the object that lost the pointer. A *PointerEnter* message is not needed since the entrance of the pointer into the object generates a *PointerMove* message.
The pointer positions reported to the objects are in display space object coordinates but may lie outside of the object's bounding box if the pointer left the object with a pressed button.
To implement efficient menus, where the user action of opening the menu can directly be extended to selecting a menu item, objects can give up the pointer ownership by calling the *TransferPointer* method of the display space manager that then explicitly transfers the pointer ownership to another display space object.

5.6 Styles

The display space manager defines the look of frames that decorate standard windows. The frames around windows serve several purposes:

- Most importantly frames offer a meta area that allows the user to manipulate the window in the display space. Moving, resizing and changing the z-order are examples of such

meta manipulations. See section 2.6.3 for details.

- The frame groups the content according to the gestalt law of organisation closure.

The display space manager decorates a standard window with 4 special frame objects, one for the title, two for the sides and one for the bottom. The decoration objects are docked to the master window. All moves of the master window are propagated to the frame objects. Figure 5.6 shows the frames docked to the master window. The look of the frame can be

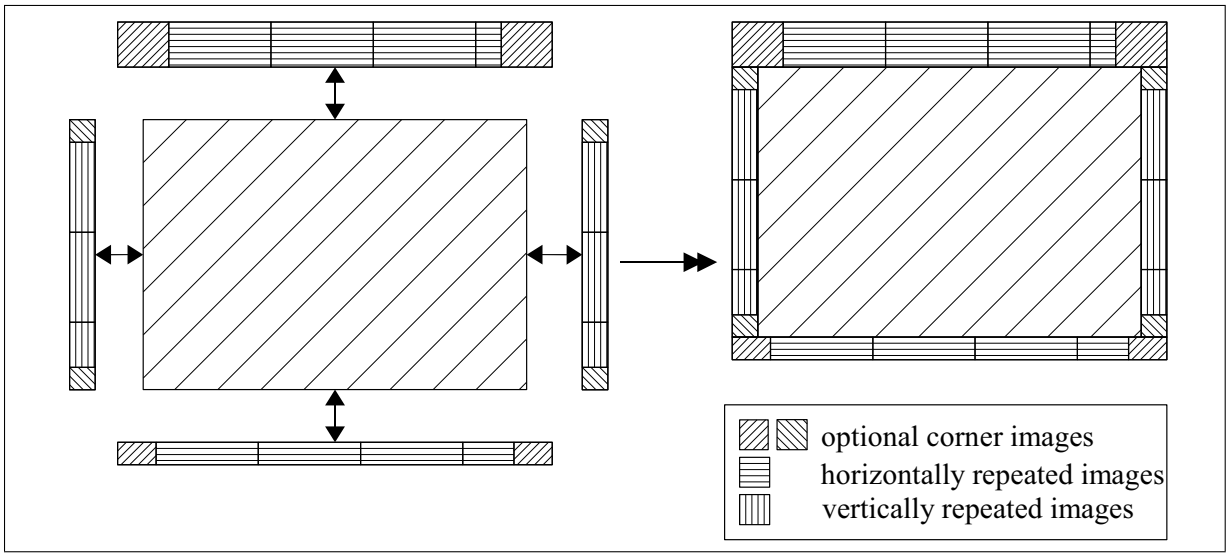

Figure 5.6: Window border

configured either by programming new decoration objects or with an XML description that can be loaded by the standard frame objects. The style description typically specifies several images, positions and colours that are used to draw the frame. While most of the images are optional, the horizontally or vertically repeated images are required. They are used to determine the width respectively the height of the frame objects. Two different sets of images can be specified: one to be used for the active focus window, the other for the windows that do not have the keyboard focus. Figure 5.7 shows a number of snapshots of different window border styles. The top left example is programmatically generated. All others are constructed from a set of images.
The style description also contains images and hot-spot data for a set of predefined mouse cursors that are used by the display space manager and by application windows.
Changing the style at runtime generates a *StyleChanged* message that is understood by the frame objects that then change their visual representation.
The window style is supplemented by a component style that can change the visual properties of GUI components. Section 6.2.5 gives details about the GUI component style support. In a diploma thesis by F. Nart an editor was developed that facilitates the creation of new window and component styles [82].

Figure 5.7: Window Style Examples

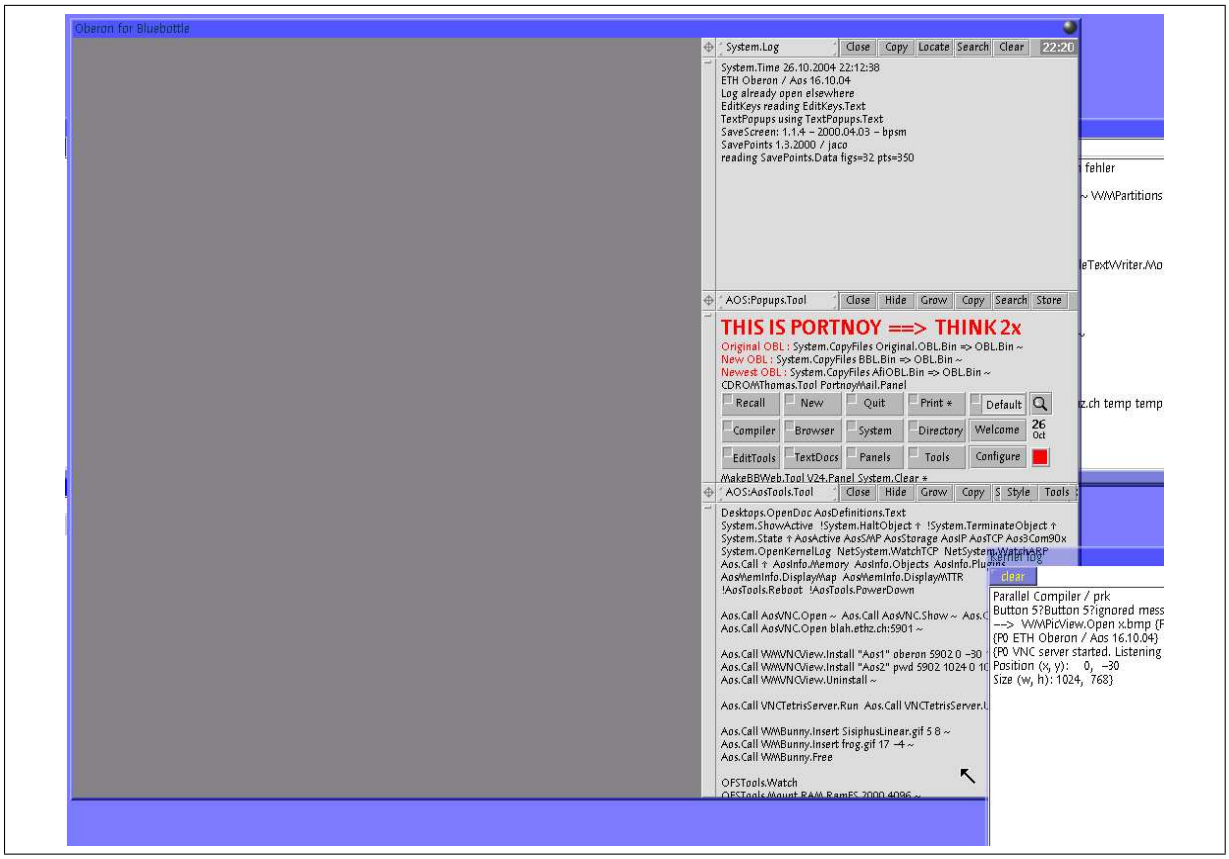

Figure 5.8: A Snapshot of Oberon as a Display Space Object

5.7 Oberon as a Display Space Object

The complete Oberon system can run as one active object within a window in the Bluebottle display space manager. A special window simulates an Oberon-compatible display driver and forwards mouse events and keyboard events to the Oberon loop. This allows continued use of a wealth of application programs that have been developed for *Native Oberon* and *Plugin Oberon*. In the early days of the Bluebottle system, Oberon was used as a cross-development environment for Bluebottle applications. Figure 5.8 shows a screenshot of Oberon on Bluebottle.

6

Component System

Nothing is particularly hard
if you divide it
into small jobs.

— Henry Ford (1863 - 1947)

Software components in general encapsulate a functionality that can be used in contexts different from the context they were developed. They are normally not tailored to fit into a specific application but to solve a problem more generally. A component system or component framework is a combination of interfaces, conventions and runtime support that defines how components are generated, combined, linked and accessed. Software components can be implemented with or without using techniques of object oriented programming. Most object oriented component frameworks are implemented for use with a single programming language because the object extension mechanism requires the same calling and storage conventions for all object definitions, object extensions and object uses. The well known programming language independent *Component Object Model* (COM) [71] from Microsoft, also known under the name *ActiveX*, is therefore implemented with a purely procedural interface. By adhering to a defined procedure calling convention, a compiler of any programming language can support creating or using COM components. Microsoft's new .NET framework on the other hand allows a tighter integration of object oriented components that can be written in many different programming languages. This becomes possible thanks to the *Common Language Runtime* (CLR) [40] that standardises and unifies common calling and storage conventions suitable for most programming languages[1]. The CLR also defines a common address space and memory management for different processes which improves the inter process usage of components.
The following introduction and discussion focuses on visual GUI components. Examples for visual components are panels, buttons, editors or scrollbars. Non-visual or *model* components can be substituted unless the topic is about alignment, drawing or direct user input. A timer is an example for an invisible component.

[1] Functional programming and multiple inheritance are not natively supported by the CLR.

The design goals for the Bluebottle component framework were as follows:

- Thread-safety
- Simplicity
- Flexibility
- Internalizable from XML data
- Support for translucency
- Support for styles
- Ease of use
- Introspection

In the following, section 6.1 gives a short overview of related GUI component systems focusing on the aspect of thread-safety. Section 6.2.1 introduces the principles chosen that lead to the implementation of the Bluebottle GUI component framework with the desired properties.

6.1 Related Work

This section introduces three related frameworks for GUI components. The three component systems are quickly introduced, followed by a description of their strategies for multi-threading.

6.1.1 Java Swing

The *Java Swing*[107] GUI component framework is the successor of the *Java AWT*[121] framework. It provides a rich set of GUI widgets such as buttons, panels, editors and tables, allowing Java programmers to create GUI application programs. The *Swing* API was introduced in the Java 2 platform and consists of 17 public packages from which normally only a small subset is used.
A Swing component can in general be accessed by only one thread at a time. Normally this thread is the associated *event-dispatching thread* that draws the components and dispatches messages. There are some exceptional methods that are thread-safe on their own: These are methods to invalidate and redraw a component and methods to register or unregister observers. All other manipulations on the state of GUI components need to be done in the *event-dispatching thread*. To run application program code in the *event-dispatching thread*, it has to be placed in the *run* method of a *Runnable* object. This can look like this:

```
Runnable guiManipulator = new Runnable() {
    public void run() { doSomething(); }
};
SwingUtilities.invokeLater(guiManipulator);
```

The *invokeLater* method schedules the invocation of the *run* method of the local object and immediately returns. The *run* method is subsequently called by the *event-dispatching thread* asynchronous to the thread that called the *invokeLater* method. Swing also offers a blocking variant of *invokeLater*, namely the *invokeAndWait* method. When using the blocking method, the calling thread may not hold any locks that might possibly be acquired by the invoked method otherwise the system may end in a state of deadlock.
An application program that wants, for example, to read the content of a text field from an external thread can use the following code pattern:

```
[...]
final String text;
[...]
    Runnable getTextFieldContent =
      new Runnable() {
        public void run() {
         // this is non-overhead code:
            text = textField.getText();
        }
    };
    SwingUtilities.invokeAndWait(getTextFieldContent);
// now the text can be used:
    System.out.println(text);
[...]
```

In this program the relevant information is the return value of *textField.getText()*. To get access to this information from an external thread, a temporary variable needs to be created, an object needs to be defined and instantiated and the object instance needs to be explicitly scheduled. In the end the value can be read from the temporary variable.
The Java Swing on-line reference [78] comments on the topic: *"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug. In general, they just aren't necessary for strictly GUI work, such as updating component properties."*

6.1.2 Windows Forms

Windows Forms is part of the .NET framework and provides widgets to create GUI applications. *Windows Forms* are often referenced as *WinForms*. The components are - like the Swing (see 6.1.1) components - not thread-safe on their own. Exceptions are the methods *BeginInvoke*, *EndInvoke*, *Invoke*, *InvokeRequired*, and *CreateGraphics*. Manipulations of GUI component's state needs to be done in the *event-dispatching thread*. The application program code is placed in a method that is then scheduled for execution on the *event-dispatching thread* with the *Invoke* method of the respective component. The *Invoke* method waits until the invoked method returns. It is hence similar to the *SwingUtilities.invokeAndWait* method of *Java Swing*. The *BeginInvoke* method is a non-blocking variant that works similar to the *SwingUtilities.invokeLater* method in *Swing*. In essence the model used in this case is that of an *asynchronous* method call. Invocation code in C# can look like this:

```
[...]
private void DoSomething() {
```

```
//operate on the component...
component.Hide();
}
[...]
component.BeginInvoke(new MethodInvoker(this.DoSomething));
[...]
```

Compared to Java, the delegate concept that is available in the .NET framework slightly simplifies the required invocation code since no explicit local wrapper class is needed.
A delegate[2] can be seen as a *micro interface* that can be implemented by any method of an object with the matching signature. It is implemented as a procedure variable that not only stores the address of the procedure but also the *this* pointer of the associated object.
Unfortunately, the delegate concept introduced in .NET is overly complicated due to its combination of *object-procedure variable* and a list there of. When more than one method is assigned to a .NET delegate, resulting in a so-called *multicast delegate* all assigned methods are called when the delegate is invoked. This is for many cases not only an unnecessary overhead but also results in confusion when the delegate's signature supports a return value. By definition only the last return value is kept.
Windows Forms components possess a property named *InvokeRequired* that indicates if the active thread can directly call the component's methods or if a special invocation is required. The *InvokeRequired* property uses a Win32 API call to get the process and thread IDs of the calling process and compares this with the thread ID associated with the window handle. If the thread is the same, *Invoke* is not required. The application programmer can take advantage of this property to avoid explicitly scheduling method calls in case it is not required.

```
private void DoSomething() {
//operate on the component...
component.Hide();
}
[...]
if ( component.InvokeRequired ) {
component.BeginInvoke(new MethodInvoker(this.DoSomething));
} else {
DoSomething();
};
[...]
```

6.1.3 Gadgets

Oberon *Gadgets* [67] is a powerful framework for component composition. It contains a rich set of predefined visual and non visual components ,so-called *gadgets*, that can be composed either interactively or descriptively. *Gadgets* can be used in all ETH Oberon Systems. Since the ETH Oberon System is a single-threaded co-operative multi-tasking system the *gadgets* framework does not support mechanisms for thread-safety. Together with the ETH Oberon System the *gadgets* framework has been ported among others to Windows [120] and Bluebottle [80] (see 5.7) that support multiple threads. To avoid concurrency problems on these systems,

[2]Delegates are called *procedure of object* variables in Object Pascal and Delphi.

threads other than the Oberon thread need to synchronise with Oberon before calling any procedures or modifying the state of the *gadgets* framework or the Oberon System[3]. In Bluebottle the synchronisation of external threads with the Oberon thread is done with a rendezvous in the Oberon loop where the Oberon thread stack is empty apart from the loop local variables ensuring no Oberon application program is running.

6.2 Bluebottle GUI Components

6.2.1 Concepts

This section introduces the principles and strategies that are used to achieve the different design goals for the user interface component system. For some of the topics, the introduction contains references to sections that go into more detail.

Thread-Safety In the component systems discussed in section 6.1, the thread-safety of the component framework has to be established by the application programmer using locks on a common object or by invoking methods on behalf of dedicated framework threads. This approach works quite well with conventional programming models, where separate threads have to be created and instantiated explicitly and the synchronisation primitives are rudimentary. In these systems, the usage of multiple threads is rather complicated so that the additional locking issues when using the GUI component framework are taken *as is* by the programmers.
In the Active Oberon [94] programming model in contrast, threads are first class citizens that are used throughout the system. The thread-safety aspects of the component framework therefore come to the fore. Instead of expecting the component-using application program explicitly acquiring locks or invoking methods on behalf of a dedicated thread, the Bluebottle components take over a great part of this task. The component author is hence responsible for the thread-safety of a component. Some methods might be non-critical and do not need protection, others may only be called if the entire composite of components that belong to the application program is locked. Instead of explicitly locking the composite, the component method can reschedule a method call on the message sequencer thread of the composite.
Of course the application programmer is not completely freed from thinking about multithreading issues. Although components guarantee that reading/writing of properties, installing and uninstalling observers as well as method calls are thread-safe on their own, higher level synchronisation problems including deadlock can occur and need to be handled by the application programmer. Section 6.2.4 gives more technical details about the synchronisation strategy.

Simplicity and Flexibility To keep the user interface components simple and flexible, object containment is preferred over object inheritance [36]. In other words, the *has-a* relation is preferred over the *is-a* relation. The *Editor* component for example is implemented as a direct extension of the basic *VisualComponent* containing a *TextView* and two *ScrollBar*s instead of

[3]Calling into the Oberon System - even into utility modules - from an external thread should be strictly avoided and, if unavoidable, needs synchronisation.

somehow extending the *TextView* component.
The main advantages of containment are:

- Reduction of the problem of the fragile base class
- Avoidance of complex inheritance graphs
- Contained objects can be created or replaced dynamically at runtime which increases the flexibility

Internalisable from XML An entire composite can be internalised from an XML document. To achieve a simple system design, the XML documents are directly transformed into composites, without the need for an intermediate structure. To make this possible, the basic component is a type extension of the generic *XML.Element*.
When internalising an XML document, the generic XML parser [113] calls an object generator procedure for each XML tag it encounters. The generator method returns an instance of a possibly specialised *XML.Element* object that is then appropriately linked into the in-memory document structure corresponding to a *DOM tree*. The application program can configure an XML parser instance with an element library that maps XML tags to application specific generator procedures. If the parser cannot find a specific generator procedure for a certain tag, it instantiates a generic *XML.Element* object that can store all attributes and sub-elements specified in the XML document for later use. The generic *XML.Element* also offers a basic query mechanism for *introspecting* the component, that is, retrieving the stored sub-elements and attributes.
When loading a composite from an XML document, the *WMComponents.Load* procedure instantiates a parser object and configures it with an element library containing the generator procedures of the installed GUI components. The parser therefore directly creates and links a hierarchical composition of components in the system memory.
The properties of the individual components are specified in specialised *Properties* sub-tags. The *Component* extension of the *XML.Element* overrides the *AddContent* method to recognise and handle the *Properties* element. The content of the *Properties* element is used to set the property values of the component. Property values that are not explicitly set in the XML document are set to the respective default values as defined in the component style. The property mechanism is described in more detail in section 6.2.5. Figure 6.1 shows the component loading process.
After an entire XML document is loaded, the observers are registered with event sources using a path mechanism that allows locating components by path or by UID.

Listing 6.1 gives an example of a composite defined in XML. The `<Properties>` tags contain the property values of the components. The composition consists of a left aligned panel with an embedded top aligned button with the title *Files*. Inside the button component there is an invisible *SystemCommand* component that is informed when the button is pressed and that will then call a specified command.

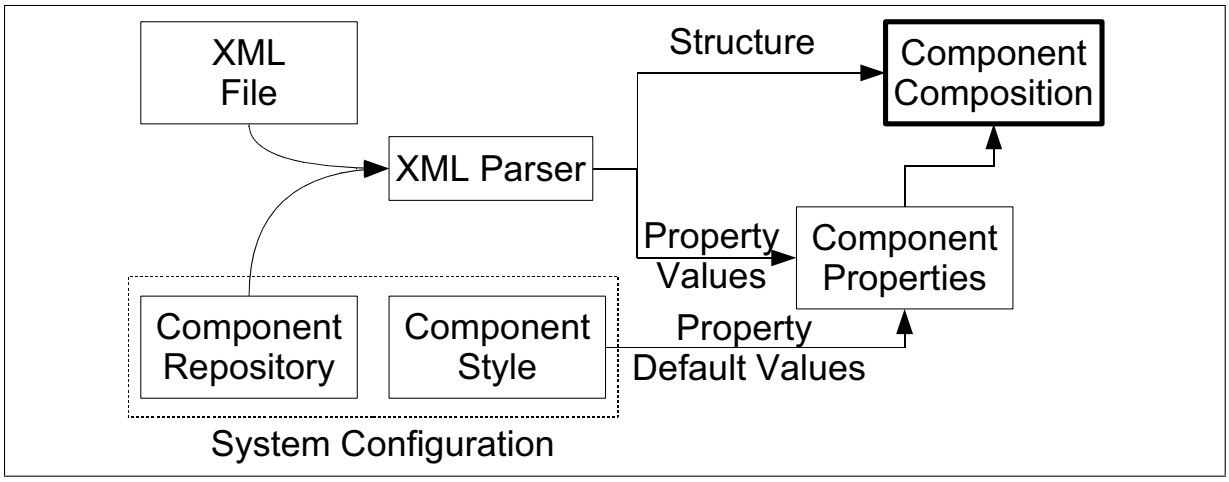

Figure 6.1: Loading Components from an XML Document

```
<Panel>
  <Properties>
    <Alignment>1</Alignment>
    <Bounds>
      <Width>92</Width>
    </Bounds>
  </Properties>

  <Button>
    <Properties>
      <Caption>Files</Caption>
      <Alignment>2</Alignment>
      <OnClickHandler>X Run</OnClickHandler>
    </Properties>

    <SystemCommand>
      <Properties>
        <ID>X</ID>
        <CommandString>WMSystemComponents.Open</CommandString>
      </Properties>
    </SystemCommand>
  </Button>
<\Panel>
```

Listing 6.1: Example of an XML Definition of a Component Composition

Supporting Translucency The support for translucency in the component system basically relies on the translucency support in the graphic system as described in detail in chapter 4. The translucency support requires drawing of underlying components strictly from bottom to top in the case of an update request.

Supporting Styles Component styles that make it possible to change the look of the system rely on default values of properties as described in detail in 6.2.5. Components are notified about system wide style changes by a *StyleChanged* message that is broadcast to all display space manager objects (see 5.5). *FormWindow*s that can contain composites (see 6.3) send a *RecacheProperties* message to all contained components upon receiving a *StyleChanged* message. The components then redraw themselves, using the new default values. Style changes only apply to properties that have not been explicitly set by the application program.

Introspection Since the AOS runtime reflection support is limited, the component framework explicitly implements a reflection mechanism. Each component contains lists containing references to all of its published features such as properties (see 6.2.5), event-sources (see 6.2.6) and observers (see 6.2.7). Applications such as component composer tools can query the lists to get access to the desired features.

6.2.2 Alignment

An important aspect of a GUI component system is how the layout of components can be organised. There are two conceptually different mechanisms to position visual components. Both are supported by the Bluebottle GUI component system:

Unmanaged Positioning Unmanaged positioning of visual components allows the designer to exactly specify where a component should be placed. The position is specified by the bounding box in coordinates relative to the bounding box of the direct parent component in the hierarchy. A component or parts of a component that are positioned outside the bounding box of the parent component are clipped away by the clipping mechanism of the graphics system as described in 4.4.1.
While the explicit specification of coordinates allows components to be arbitrarily positioned, it makes dynamic and also static changes difficult. Whenever a component needs to be inserted, deleted or resized, all other components must be explicitly adapted, too. For most applications, a less explicit positioning mechanism is better suited.
Visual components whose positioning is unmanaged have the *alignment* property set to *WMComponents.AlignNone*.

Managed Positioning Using an automatic alignment and positioning mechanism for visual components normally results in more adaptive user interfaces, both from the programmer's and user's perspective. Components with automatic alignment can dynamically adapt to changing screen, window or parent component sizes without the need for additional program code.

Adding, removing, resizing or moving components is simple since all the other components are automatically adapted.
There are many different strategies how components can be positioned. The Bluebottle GUI component system allows the installation of a layout manager per container component that implements a specific alignment strategy. This is similar to the Java AWT and Swing frameworks [28].
The default layout manager that is hard-coded in the standard visual component offers a very simple, but powerful, positioning mechanism[4]. It supports aligning a component inside the *available* space of its parent component. Each aligned component reduces the space that is *available* in its parent component. Components can either use the top, bottom, left, right or complete rest of the available space. A component that is left or right aligned only needs to specify its width; the height is automatically determined by the available space in the parent component. A component that is top or bottom aligned respectively only needs to specify its height; the width is automatically calculated. A component that takes the rest of the available space does not need to specify either extent. Only one visible component within a common parent component can reasonably take the rest of the available space.
The components take the space in the order they are inserted into the parent component. Figure 6.2 gives four simple examples of the alignment strategy. The following description of the four examples shows the alignments and insertion order of the sub-components:

A top, top, top, top, top

B left, left, left, left

C top, bottom, left, left

D top, bottom, left, left, bottom, bottom, right

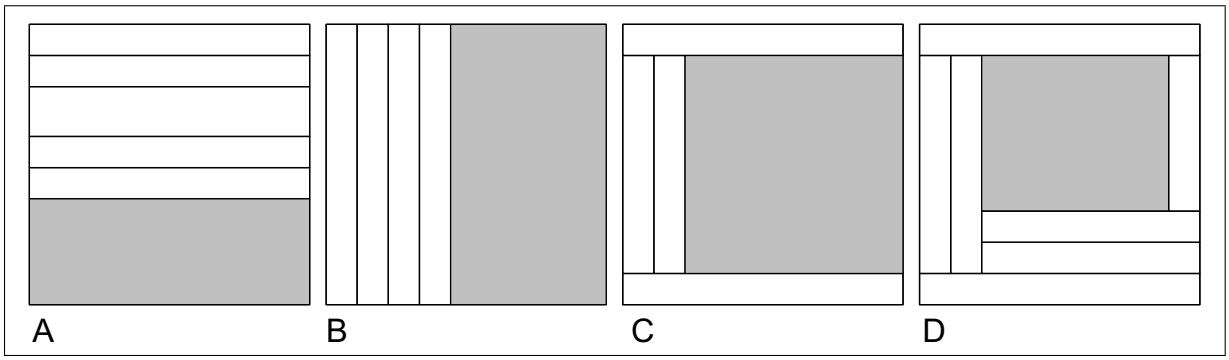

Figure 6.2: Simple Alignment Examples

Using the alignment mechanism recursively allows the creation of complex, but not overly specified, layouts that can within limits automatically adapt to size changes. Figure 6.3 *A*

[4]Up to the time of writing, the default layout mechanism was found flexible enough for all cases, no alternative layout mechanism has been necessary.

shows an example of a recursively composed layout. The thick lines indicate a component containing sub-components for recursive alignment. The grey bars are used as spacings. Figure 6.3 *B* shows the same layout in a resized window.
Panel components that have their colour set to a completely transparent black (the RGBA colour value 0) are optimised so that their use becomes inexpensive in terms of computing-time required for the drawing. This allows the extensive use of transparent panels for layout purposes.

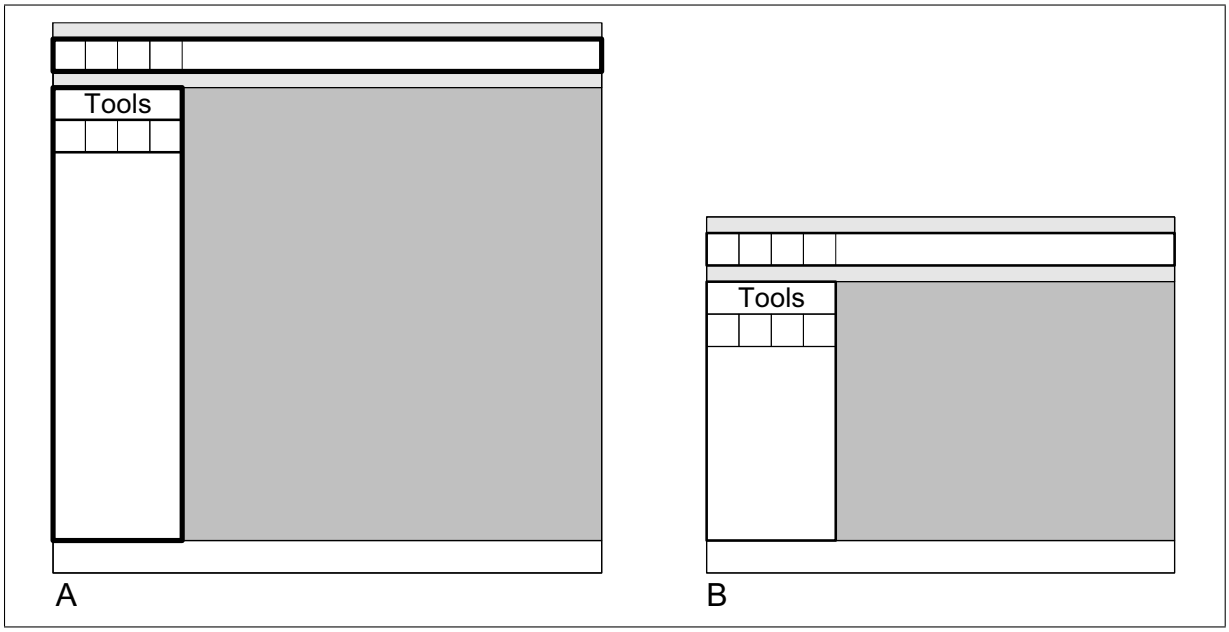

Figure 6.3: Complex Alignment of GUI Components

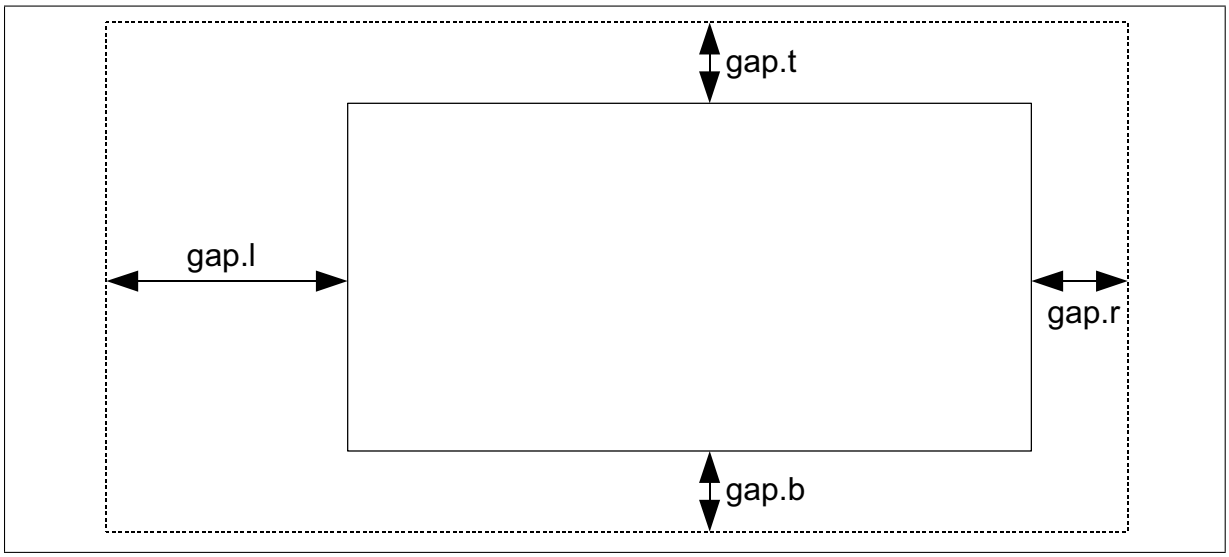

Figure 6.4: Gap between Components

To achieve visual grouping of related components, empty placeholder panels can be added between components. This method can require a great number of functionally useless panels. To avoid this programming or designing overhead, the visual components can define a gap of

empty space that is added by the layout mechanism (Fig. 6.4). The spacing can be inserted left, top, bottom and right of the component with the *SetLeft*, *SetTop*, *SetRight* and *SetBottom* methods of the *gap* property that is available for all *VisualComponents*. Negative gap values are possible, but they can lead to overlapping components.

6.2.3 Composition

Individual components can be composed to form larger interacting compounds that solve a certain problem. A typical example of such a composed component is a visual sound volume controller that can consist of a *panel* component containing a *scrollbar* and a *label* component as well as an invisible *volume control model* component. The composed component can be instantiated multiple times to form a mixer panel.
Whilst a component can have more than one contained sub-component, a component can only be contained in one super-component. Therefore, the containment relations between components form a tree and thus can be naturally mapped to an XML structure where sub-components map to sub-tags as described in 6.2.1.

Other inter-component relationships within a composite are less structured. One component may be interested in properties or events of components other than its super- or sub-components. To express these more complex interoperability in a textual form, paths or UIDs can be used.

6.2.4 Synchronisation

Single-threaded software components are generally substantially easier to write and use than multi-threaded implementations, because the latter requires locking mechanisms to achieve thread-safety. This section describes in more technical detail the synchronisation strategy used in the Bluebottle GUI component framework.
There are several different synchronisation problems that need to be solved by the synchronisation mechanism of the component framework:

Low-Level Data Races A software component can only be called thread-safe if all its accessible methods and properties are protected from reaching invalid or inconsistent states when they are accessed at the same time by more than one process or thread. For example it may happen that two threads access a shared variable and one of them changes the variable's value without preventing the other access from being simultaneous. Software components that export variables accessible to different threads are in general not thread-safe. Hence a component programmer should not directly export variables but rather offer synchronised accessor methods to non-exported variables. This simple measure solves the problem of low-level data races but introduces the possible problem of *self-deadlock*. A concise definition of low-level data races can be found in [101].

Self-Deadlock Self-deadlock is a special case of deadlock that happens on a single object instance if a synchronised method calls another synchronised method using the same non-recursive lock. Two common ways to avoid self-deadlock are the use of recursive

locks or a strict separation of synchronised interface methods from unprotected internal implementation methods [103].

High-Level Data Races There are several high levels of data races that can occur in multi-threaded situations even if a component is protected against low-level data races. High-level data races are based on the interdependence of separately meaningful properties of a component. A comparison of high-level data races and low-level data races can be found in [5]. A simple example of such a high-level data race is a rectangle that might be specified by the (left, top) and (right, bottom) coordinates. There might be separate synchronised accessor methods for both (left, top) and (right, bottom) because each of them is meaningful on its own. To calculate the width or height of the rectangle in a thread-safe way, these two synchronised methods are not sufficient; the rectangle might have been changed in the time between safely reading each of the two coordinate pairs. The higher level equation $width = right - left$ holds only in a fully synchronised context. There are mainly two approaches to prevent these high level data races:

- by offering synchronised methods for all meaningful access combinations to complex data
- by offering a lock instance that must be held before accessing any of the state variables (transaction model).

Detecting high-level data races is a current research topic of several research groups [4].

Deadlock Generic deadlock in contrast to *self-deadlock* can involve several different locks. Protecting each component's state with an individual lock substantially increases the danger of deadlock, in particular in combination with cyclic component structures.

Composite Data Races A composition of several different components is an even higher level of possible data races. For example, drawing or passing messages between components cannot be done in parallel with manipulations on the component hierarchy since this might lead to invalid clipping regions and misdirected or lost messages. To protect the entire composition from getting into aninconsistent state, a common hierarchy or composition lock must be held before operating on the composite. Without taking great care of the locking order, this can be a further source of deadlock.

In systems that feature extensible components, great care must be exercised that all extensions of a component adhere to the same locking scheme. Depending on the programming language used, the compiler can help with ensuring the locking discipline. Some research is currently being done in the area of extended static [91] or dynamic [101] checking of code with respect to correctness in terms of a given locking scheme.

Similar to most single-threaded graphical component systems, the Bluebottle GUI component framework uses a thread and a queue for the handling of asynchronous events and a

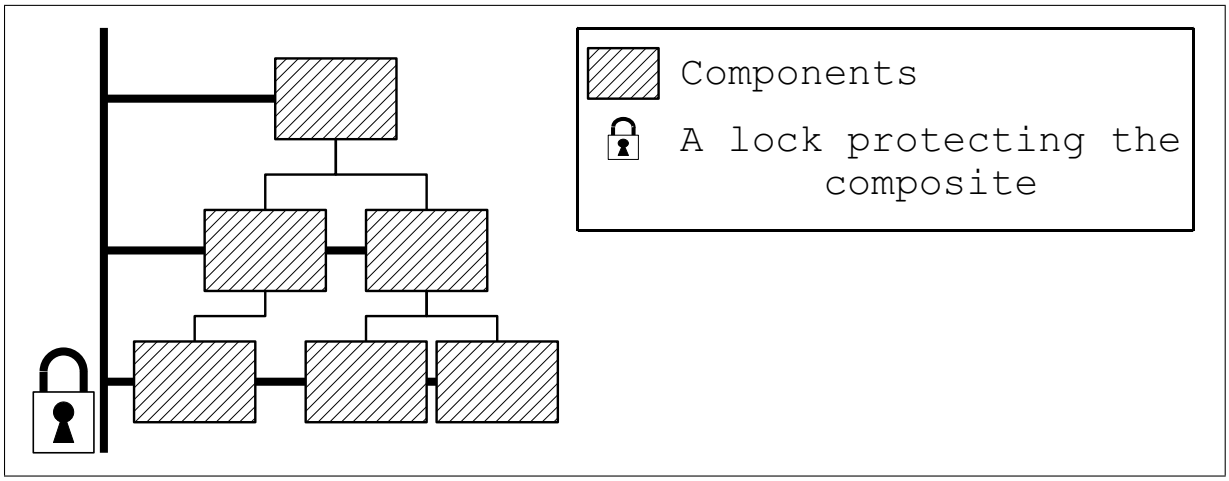

Figure 6.5: Hierarchy Lock

hierarchy lock that ensures a consistent view on inter-component relations. Figure 6.5 shows a composite that is protected with a lock.
The hierarchy lock, the message queue and the message handler thread are combined into an active *message sequencer* object that can be reused in synchronisation contexts within and outside of the component system. The sequencer is for example used in the display space manager (5.5). Its task is to provide

- serialisation of asynchronous messages and events (see 6.2.6 for details about events)
- decoupling of processes
- a recursive locking mechanism

The serialisation of messages and events is provided with the help of synchronised methods for adding messages and events to a single queue.
The decoupling of processes is achieved with an activity (thread) in the sequencer object that takes messages and events from the queue. Messages are forwarded to an installed handler method, and events are sent to the respective observers.
The locking mechanism is realised with a recursive lock that is implemented in *WMLocks*. The activity in the sequencer object synchronises with the recursive lock before calling the message handler or event handler methods. This ensures that the installed message handler method is only called with the recursive lock held by the sequencers activity.
Figure 6.6 shows the detailed composition and function of a sequencer object. The *A* in the image symbolises the activity of the object.
The lock that protects the component hierarchy (Fig. 6.5) is the same as the lock of the message sequencer object that is associated with the composite (Fig. 6.6).

In the Bluebottle GUI, the following strategy for synchronisation is applied:

1. Instead of exporting the state of a component with object variables, it is exported as a list of special property object instances that encapsulate the values and offer synchronised

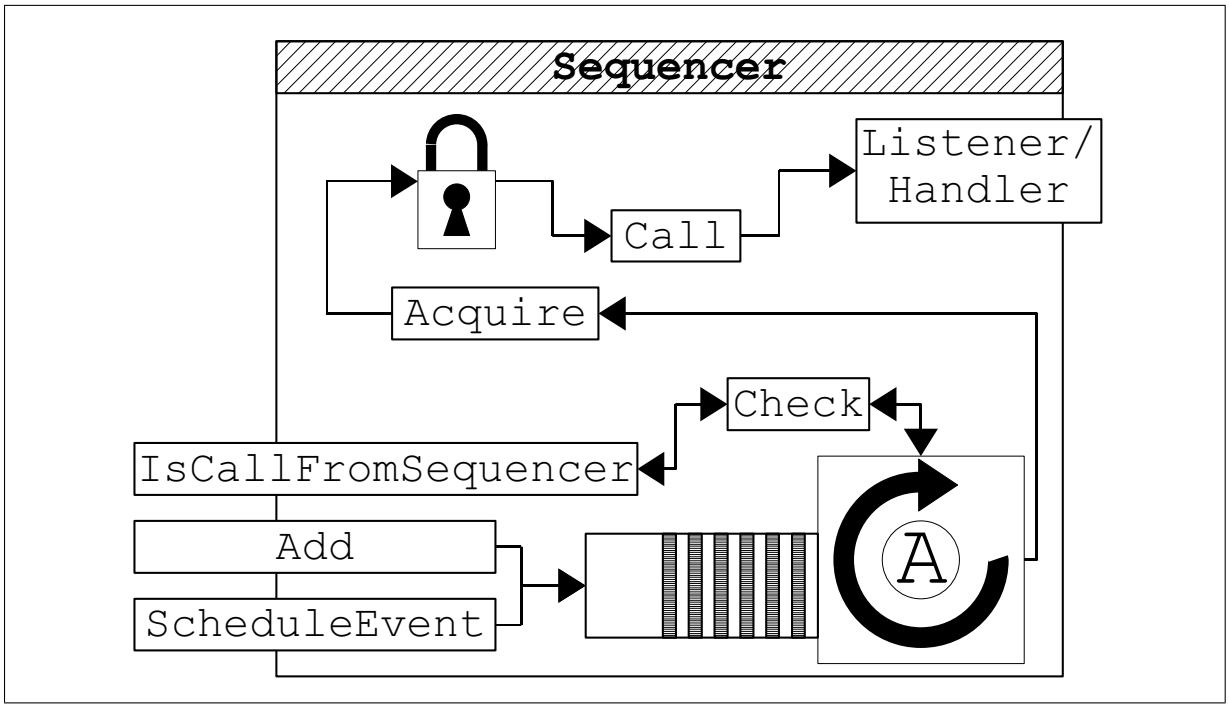

Figure 6.6: Detailed Description of a Sequencer

access methods. Properties can encapsulate basic types such as *LONGINT* or *BOOLEAN* but also complex types such as *Rectangle*s. The accesses to the properties of a single component can be grouped into *transactions*. In a transaction all accesses to the properties of a component happen without interference of other threads. When one or more properties are changed in a transaction, the component is informed with a *PropertyChanged* event, upon which the component internalises the changed values of the exported properties. The property mechanism is described in more detail in section 6.2.5.

2. Calls to event methods from activities other than the message sequencer are queued to be invoked on the message sequencer thread. The event re-scheduling mechanism is described in more detail in section 6.2.6.

3. Other special methods enforce their locking strategy by efficient runtime assertions.

As a fine point we note that checking if the current method call emanates from the associated sequencer object can be done very efficiently without involving locks. It is done by storing the sequencer's thread object reference in a private field when it is created and comparing this field to the current thread object. The current thread object can be accessed efficiently via the thread's stack [80].

6.2.5 Properties

For flexibility and versatility, most software components can be parameterised in one or another way. The parameterisation of a component instance can happen at construction time but usually also at runtime. To guarantee controlled access, especially in multi-process situations,

it is common practice to store the values in variables that are local and private to the respective objects and offer exported, sometimes synchronised, accessor methods. The concept of an exported state value of a component is often referred to as a *property*. Some programming languages such as *Delphi* or C# offer special support and notations for the definition and use of properties. In both, *Delphi* and C# , a named and typed property can be defined specifying the respective read and/or write access methods. If only either the getter or the setter method is given, a property is read- or write-only respectively. The properties can syntactically be accessed like variables. Behind the scene, the compiler generates a call to the respective accessor method.
Other programming languages such as Java or C do not have a special language construct to denote properties. This does not mean the concept of properties cannot be used in these languages. The Java-Beans framework as an example introduces the property concept based on the naming convention that all methods starting with get*P* and set*P* are read and write accessor methods for the *property P* respectively.

Before implementing a property support system for Bluebottle, we need to specify its requirements:

Access detection When a property value is changed it should be able to inform interested observers, that is, other objects that have subscribed to the property.

Serialisation It should be possible to load and store property values from and to data streams

Synchronisation The access to a single property should be atomic. It should be possible to group several accesses to one or more properties into a *transaction* to guarantee consistency across all of them.

Reflection It should be possible to get meta information about a property such as its name and type

Default values Properties of GUI components should support default values that can be used to implement *skins*. A skin defines the look of visual components, such as the colour of buttons or the shape of window frames. Since the look of components is defined by their properties, a skin defines default values for the properties of the different classes of visual components. Application programs need to be able to explicitly override the default look of any component. To allow changing skins at runtime the properties need to explicitly distinguish between *default* setting and *non-default* setting, even if both values are by coincidence the same, so that a skin change cannot override the settings of an application program.

Since Active Oberon - similarly to Java - does not offer a language construct for properties, the concept has to be expressed differently.
The main difficulty in adopting the Java-Beans approach lies in the less advanced reflection

mechanism of the Bluebottle system. However any centralized reflection support can be substituted by implementing the generic introspection functionality in the code of the basic component class. The reflection support that is integrated into the component code must at least offer methods to enumerate the properties of the component and to query meta information and offer ways to read and write the values of properties. This is essentially the mechanism used by the Gadgets framework or COM.
The following small code snippets show the implementation of such an explicit reflection mechanism. The mechanism is robust against changes of the number of properties in super classes because the property indices are normalised by subtracting the number of properties defined in the super class.

Procedure *GetNofProperties* returns the number of properties that are available in the component. This is the number of properties in the super class plus its newly defined additional properties.

```
PROCEDURE GetNofProperties() : LONGINT;
BEGIN
 RETURN GetNofProperties()^ + NofNewProperties
END GetNofProperties;
```

Procedure *GetPropertyName* returns the name of the property with index *propertyNr*. To make the index independent of changes in the properties of the super class, the number of properties in the super classes is subtracted from *propertyNr*. This example defines the two new properties *Color* and *Size*. If the property index *propertyNr* lies outside the defined range, the super class implementation of *GetPropertyName* is called.

```
PROCEDURE GetPropertyName(propertyNr : LONGINT) : String;
BEGIN
 CASE propertyNr - GetNofProperties^() OF
 | 0 : RETURN "Color"
 | 1 : RETURN "Size"
 ELSE RETURN GetPropertyName^(propertyNr)
 END
END GetPropertyName;
```

The method *GetPropertyAccessors* returns two method variables that point to methods implementing the access to the respective property values.

```
PROCEDURE GetPropertyAccessors(propertyNr : LONGINT;
      VAR set : PropertySetter; VAR get : PropertyGetter);
BEGIN
 CASE propertyNr - GetNofProperties^() OF
 | 0 : set := SetColor; get := GetColor
 | 1 : [...]
 ELSE RETURN GetPropertyAccessors^(propertyNr, set, get)
 END
END GetPropertyAccessors;
```

The following methods implement the effective access to the value of the colour property. The access methods could be synchronised.

```
PROCEDURE SetColor(color : String);
[...]

PROCEDURE GetColor() : String;
[...]
```

A problem that occurs with this approach is the loss of static type checking. In the programming example above, the *GetPropertyAccessors* method can only return delegates to methods that access a given type, in this case of the type *String*. To make more than one basic property type possible, the accessor methods must either operate on a generic type or there must be different methods to acquire the property accessors for each respective type.
Generic accessor methods can, for example, operate on objects that can in different subclasses encapsulate a field of the effective property type. This encapsulation that is sometimes called *boxing* requires dynamic memory and is very unnatural for the programmer if there is no support in the programming language. The encapsulation also prevents static type checking. Passing property values as strings as in this example is another possibility that offers no automatic type checking at all.
A solution that offers a different method to acquire the accessor methods for each different type requires a prohibitive amount of administrative code in the component.

Property-Objects Approach Even with the solution of a generic string accessor type, the amount of administration code required is large. An alternative solution models the exported properties as objects. These objects contain the respective property values as well as accessor methods. All properties of a component instance are stored in a property list. The representation of properties through object instances is a natural mapping to an object oriented language without specialised language support for properties. This mapping allows the property system to be equipped with all the desirable features.
The property object instances can be added as exported[5] field variables of a component, in addition to publishing them in the property list, so that the property access can be hard-coded into a program. This avoids searching at runtime, allows taking advantage of static type checking and gives natural access to the property values similar, but not equal, to the access of variables. Also, the additional overhead that might hide behind the property access is not completely invisible to the programmer.
An advantage of property object instances over properties that are deeply integrated into the language is the support for complex properties with several different access methods. A rectangle property can for example offer access to its width and height as well as to the rectangle as a whole.
One drawback of the solution is the additional memory overhead. Each property instance typically uses significantly more memory than the encapsulated value type alone. This is because

[5]The read only export of Active Oberon protects the property object instances from being replaced while still allowing read and write access to the encapsulated values

of heap management overhead resulting from type information and more importantly internal fragmentation due to heap block alignment.

Property Interface

This section introduces the conceptually important parts of the generic property interface:

- *PROCEDURE &New (prototype: Property; name, info: String)*; the constructor of the property object takes a *prototype* property that defines the default value and meta information. The *prototype* parameter can be set to *NIL* to omit the prototype, for example if the created object instance is itself acting as a prototype for other properties. The parameter *name* defines the identifying name of the property. The *info* parameter contains an info string that describes the property function. It is only used as an information for the programmer.
- *PROCEDURE GetName(): String*; returns the identifying name of the respective property. It is used for finding the property in the property list of a component.
- *PROCEDURE Reset*; resets the property to the default value.
- *PROCEDURE FromXML (xml: XML.Element)*; internalizes the property value from a given XML element. This is used when loading a component from an XML document.

The methods *Get* and *Set* are not defined in the generic interface since they are type dependent. Each concrete *property* implementation defines *Get* and *Set* methods of the respective type.

Default Values

As long as no *Set* method was called on a property instance, it returns the value of its prototype. The values of the prototype properties are used to specify all property values of a component that are not explicitly set by a program or XML specification of the component. This mechanism implements the skinning ability of the Bluebottle GUI component system. Whenever the property value is changed the first time, the component is no longer in the default state, even if the value written to it is the same as the default value. It will then no longer be influenced by changes of its prototype. The property can only be reset to represent the default value by a call to its *Reset* method.

Property List

The property list stores all the properties of one component. It is instantiated in the constructor of the basic *Component* object which also fills it with the generic properties that are defined for all components, for example the *id* property. Each constructor of an extension of the *Component* object adds its additional properties to the list. After the creation of the object, when all constructors have been called, the property list contains all the properties of the component.

Apart from storing the property object-instances of a component, the property list also maintains a recursive reader/writer lock that is shared by all properties of a component and informs interested observers about any changes. The notification about changes takes place whenever the last write-lock is removed. For the notification mechanism, an *EventSource* object as defined in 6.2.6 is used. In the case that no changes occurred between the acquisition of the first write-lock and the release of the last, no notification is sent. In the case of one single change, the notification data contains the reference to the changed property object-instance. If more than one property was changed, the notification data contains the reference to the property list itself. The change notification mechanism is conceptually similar to the mechanism used to notify text model changes as described in 3.3.2.
The recursive reader/writer lock can be used as another means of avoiding high-level data races. The property list can be used to enumerate all properties of a component, for example for reflection or serialisation.

6.2.6 Events and Observers

The *observer pattern* [36] that is often used in component frameworks defines a one-to-many dependency from one object that is being observed by many observers. Whenever the observed object changes its state, it has to notify all its registered observers. In other words, the model creates an event for which all registered observers are listening. The model/view/controller concept is probably the best known application of this pattern.
To simplify and standardise the implementation of the observer pattern in the Bluebottle sys-

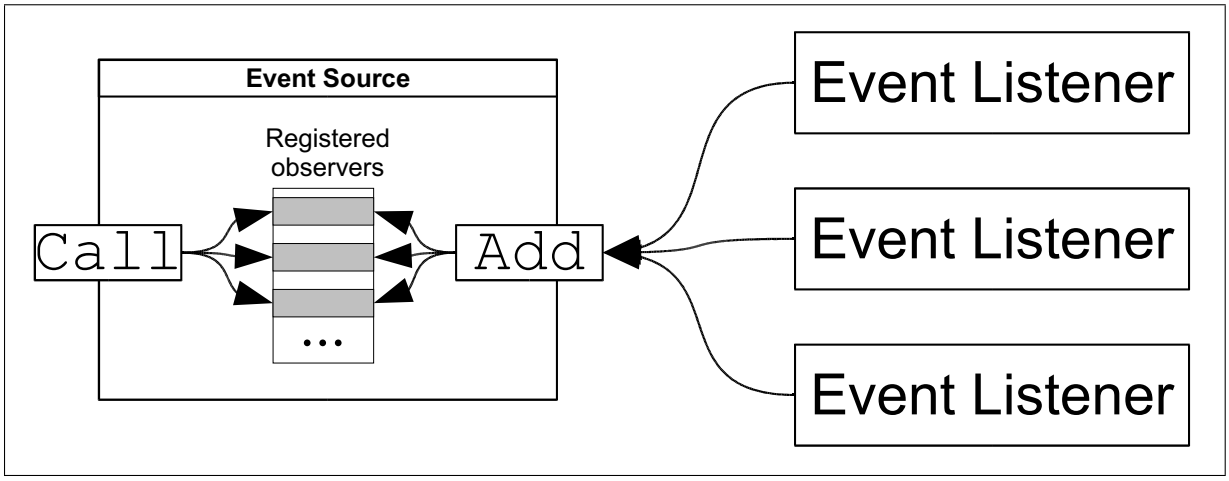

Figure 6.7: Observer : Event Source and Observers

tem, the module *WMEvents* specifies a reusable *EventSource* object and an *EventListener* micro interface in the form of a delegate procedure type. Figure 6.7 shows an *EventSource* object with three registered *EventListener*s. An object that wants to be observable, instantiates and exports an *EventSource* object variable. Whenever the observable object changes its state, it calls the *Call* method of the *EventSource* variable. The *EventSource* object then automatically invokes all registered observer methods. The registration and unregistration of *EventListener* methods is handled by the *EventSource* object so that the observed object does not need to implement

additional management code.

The *EventListener* signature takes a sender and a data object reference as parameters:

```
TYPE
  EventListener = PROCEDURE {DELEGATE} (sender, data : ANY);
```

Event Sources

The *EventSource* offers the following synchronised programming interface:

Constructor

```
PROCEDURE &New(owner : ANY; name, info : String;
               finder : CompCommandFinder);
```

- *owner*, contains the object-instance that offers itself to be observed. An object that creates an *EventSource* object usually sets this parameter to *SELF*. The *owner* parameter is stored in the *EventSource* object and used as the *sender* parameter when calling a registered observer.
- *name*, contains a reference to a string that is used as a name to identify the respective event source. The name is used to find a specific event source out of an *EventSourceList*.
- *info*, contains a reference to a descriptive string that is used for reflection. This is only used as meta-information for the programmer. It can be used in builder tools.
- *finder*, contains a delegate to a method that can find an *EventListener* by string. This is used for wiring component structures defined in an XML document.

All of the constructor's parameters can be set to *NIL* if no relevant information is available.

Add, Remove Add/Remove registers/removes an observer method that matches to the *EventListener* signature. Whenever the event is fired, each observer method is called as many times as it has been registered. The methods have the following signatures:

```
PROCEDURE Add(observer : EventListener);
PROCEDURE Remove(observer : EventListener);
```

Call The *Call* method invokes all the registered observers. It uses its *owner* field, set in the constructor as the *sender* parameter of the observer and takes one additional *data* parameter that can be any object reference for the observer's *data* parameter. The observer methods need to be thread-safe since they can be called at any time. To make the observer method thread safe, the efficient strategy described in 6.2.4 and [32] can be used. No invocation order for the observer method may be assumed, the invocation might even be in parallel[6].

[6]Although the current implementation sequentially calls the observers in inverse order of registration, any future implementation may change this

HasListeners The *HasListeners* method can be used by the observed object to check if at least one observer is registered with the event source. If no observerss are registered, the observed object can avoid the potentially expensive creation of the *data* object used in the *Call* method.

```
PROCEDURE HasListeners() : BOOLEAN;
```

The additional methods *AddByString* and *RemoveByString* use the *finder* delegate to find the observers to be added or removed. The name resolution is performed lazily on use of the *Call* method. If the delegate could be resolved, it is cached. If it is not found, the resolution is tried again on the next call.

6.2.7 Observers

An observer is implemented as a method with a signature that matches the *EventListener* micro interface. It is either registered with an *EventSource* object or can also be called directly from a program.
If an observer method of an object is called, it first checks if the call originates (directly or indirectly) from the sequencer thread. In this case the observer method simply does whatever it needs to do to handle the event. Otherwise, if the method call originates from any other thread, the observer simply re-schedules the event in its associated sequencer object and returns, knowing that it will be called again with the same parameters but this time from the sequencer thread. The sequencer thread always acquires the hierarchy lock before calling a component method. Thanks to this strategy, an event handler method can always safely operate on the state of its object, no matter from which thread it was originally called. The following code sequence shows a generic event handler:

```
PROCEDURE EventHandler*(sender, data : ANY);
BEGIN
  IF ~sequencer.IsCallFromSequencer() THEN (* sync needed ? *)
    sequencer.ScheduleEvent(SELF.EventHandler, sender, data)
  ELSE (* actual business logic *)
  END
END EventHandler;
```

Listing 6.2: Synchronizing Event Handler

The observer code shows some similarities with the synchronisation in the .NET GUI component framework (section 6.1.2). The *IsCallFromSequencer* method has the same function as the *InvokeRequired* property in .NET. *ScheduleEvent* matches the *BeginInvoke* method. The main difference between these solutions is the place where the decision happens whether or not to re-schedule the call. In .NET the decision is left to the application program that must know if it is running in the message handler thread. In Bluebottle the component is responsible for this decision.

6.2.8 Message Handling

Strict parental control is used as a strategy for the drawing and passing messages such as keyboard messages, mouse messages and broadcasts. Other inter-component event-handling can bypass parental control if event sources are directly wired to event handlers. The direct wiring is implemented by delegate procedures, a combination of a method pointer and object reference that allows direct component-to-component calls, without the need of sending a message up and down an entire hierarchy. The delegate procedures allow simple component wiring by registering observers with event sources without the need for an object to implement a special interface. In a graphical component system, an inter-component message can, for example, be sent by a button that is pressed or a string in an editor that is changed.

6.3 Display Space Manager Integration

Visual components expect their parent components to be a visual component again. To make use of the GUI components they need to be integrated into the display space of the display space manager. This integration is done with a special visual component called the *Form* and a special window called the *FormWindow*. The *FormWindow* is a specialisation of the *DoubleBufferWindow* that is described in section 5.3.3. When the *FormWindow* is created it internally creates a *Form* component and links the *Form* component's *FormWindow* reference to itself. The *FormWindow* forwards all messages from the display space manager to the *Form* component that then handles them according to the *VisualComponent* message handler. When the *FormWindow* is resized by the display space manager, it can either ignore the resize resulting in an automatic zooming by the display space manager as described in 5.3.2 or it can create new buffer images and send a *resized* message to its *Form* component. The behaviour is determined by the application program that creates the *FormWindow*.
The *Form* component differs from a *VisualComponent* in not expecting to be integrated into another *VisualComponent*. All messages that a normal *VisualComponent* would forward to its parent component are either handled by the *Form* component or forwarded to the *FormWindow*. The most obvious messages that are handled by parent visual components are requests to invalidate regions of a component. On an invalidation request of a child component, a normal *VisualComponent* transforms the request into its own coordinate system by adding the left top coordinates of the bounding box and forwards it to its own parent component. In contrast, the *Form* component has no parent component and therefore handles the request itself. It first prepares the *FormWindow*'s canvas by setting the requested invalidation rectangle as a hard clipping rectangle as discussed in 4.4.1. Then it issues a *draw* request to its child components that intersect with the clipping rectangle. When the *draw* request is handled, it swaps the respective area as described in 5.3.3 and invalidates it in the display space manager so that the update becomes visible.

6.3.1 Implementation

The Bluebottle component framework has an object oriented design. All components extend the basic component object defined in *WMComponents.Component* that is itself an extension of the generic *XML.Element*. The basic component offers the following interface[7]:

Object variables:

- *sequencer : MsgSequencer*; contains a reference to the message sequencer associated with the containing composite. The synchronisation is described in more detail in section 6.2.4.
- *properties : WMProperties.PropertyList*; contains all the properties that are exported by a component. It can be enumerated by a builder or reflection tool to locate a property or get a list of all the available properties of a component. See section 6.2.5 for more information about properties.
- *events : WMEvents.EventSourceList*; contains all the events that are generated by a component. It can be enumerated similarly to the property list for getting meta information. (See 6.2.6)
- *eventListeners : WMEvents.EventListenerList*; contains a list of public observers that can be added to event sources of other components. (See 6.2.7)
- *id, uid : WMProperties.StringProperty id* and *uid* are directly exported properties that contain strings that identify the property. *id* contains a string by which the component can be identified by its direct super component. The *id* value is used to identify a component by path, it must be unique within the containing component. The *uid* must be unique within an entire component hierarchy. It is normally used to link a program to a component that is loaded from a textual XML description. Both values can be ignored if the component does not need to be located via its name.
- *enabled : WMProperties.BooleanProperty* describes if a component is active or not. A button component might for example be not clickable if it is disabled. All components are enabled by default.

Methods:

- *PROCEDURE &Init*; the constructor of the component initialises the *properties*, *events* and *eventListener* lists. It also creates and adds the *id*, *uid* and *enabled* properties. The *Component* constructor also calls its inherited constructor. All extensions of *Component* must call their inherited constructor as the first thing in their own constructor to ensure the component is created correctly.

[7] Some internal bookkeeping methods and fields are omitted

- *PROCEDURE Initialize*; The method *Initialize* is called after an entire composite is completed or has been rearranged. Extending components can override the implementation of this method to internalise data that is depending on property values. An example for this is an image that needs to be loaded according to a property value. Property values are only reliable after *Initialize* was called the first time. If the method is overridden, the inherited *Initialize* method must be called to ensure the proper initialisation. *Initialize* is called by the *Reset* method. It should not be called directly.

- *PROCEDURE Finalize*; The method *Finalize* is called when a composite is being terminated. The method is not intended to be called by application programs. It is called by the component framework while holding the *hierarchy lock*. Extending components that override this method must call the inherited method. Extended components should stop possible activities that they are controlling, and close and release all their external resources. Examples for such resources are TCP connections, UDP ports or protocols that require a controlled finalisation.

- *PROCEDURE AddContent(c:XML.Content)*; The *AddContent* method is inherited from *XML.Element* and is responsible for adding content objects. The added objects can be XML contents or other components. The XML tag *Properties* is forwarded directly to the component's property list object *properties* which then fills the properties with the values extracted from the XML *Properties* element.
  AddContent is normally not overridden by extending components. Additional access control is a possible reason for overriding.

- *PROCEDURE GetComponentRoot(): Component*; returns the topmost element in the hierarchy as seen from the called component, that is still a component. This is not necessarily the root object of a hierarchy since a component hierarchy could be stored in an XML hierarchy. *GetComponentRoot* should be regarded as *final* and not be overridden by extending components.

- *PROCEDURE FindByUID(uid:String) :Component*; The method *FindByUID* recursively searches a component and its sub-components for a given uid. If found it returns the corresponding component, otherwise *NIL*. The method is *final* and should not be overridden. It is mainly used by programs to locate a specific component within a composite that was loaded from a file. The method is normally not called directly but is used indirectly via the *StringToComponent* method.

- *PROCEDURE FindByPath(VAR path:ARRAY OF CHAR;*
  pos:LONGINT):Component; The method *FindByPath* searches a component following a given path relative to the called component. If the path is valid and the component found, it returns the component, otherwise it returns *NIL*. The method is *final* and should not be overridden. It is mainly used to locate components in a loaded composite by relative paths. The method is normally not called directly but is used indirectly via the *StringToComponent* method.

- *PROCEDURE StringToComponent(str:String):Component*; Uses a uid or a id-path to localise a component within the composite.
- *PROCEDURE StringToCompCommand(eventstr:String):EventListener*; Uses a uid or a id-path to find an observer of a component within the composite.
- *PROCEDURE Reset(sender, data:ANY)*; recursively calls the *RecacheProperties* and *Initialize* methods of all components. It is used when components are rearranged or new components are added at runtime.
- *PROCEDURE HandleInternal(VAR msg:Message)*; The *HandleInternal* method is the synchronised message handler of the component. It must not be called directly. Clients that want to send a message to a component should always use the *Handle* method that ensures proper synchronisation. Extensions of *Component* can override this method to install a synchronised message handler. Calling the inherited *HandleInternal* method is optional and can be used as a message filter or to implement parental control.
- *PROCEDURE Handle(VAR msg:Message)*; The *Handle* method acts as a synchroniser for the *HandleInternal* method. A message that is sent to the *Handle* method from a process other than the *sequencer process* is re-scheduled and put into the message queue, that is, asynchronous calls to *Handle* return after the message is queued while synchronous calls return after the message is handled.
- *PROCEDURE PropertyChanged(sender, property:ANY)*; The *PropertyChanged* method is called by the component framework whenever a registered property of the component has been changed. The *property* parameter identifies the property that was changed. If more than one property were changed in a transaction, then the *property* parameter is set to the property list *properties* of the component. In the case of a transactional change of more than one property, the *RecacheProperties* method is called by the framework. See section 6.2.5 for more information about properties.
- *PROCEDURE RecacheProperties*; The *RecacheProperties* method is called by the internal property change handler via the sequencer, either if multiple properties have been changed or the *Reset* method has been called. In the case of a transactional change of more than one property, the *PropertyChanged* method is called too. Components that override the method should call the inherited *RecacheProperties* method. See section 6.2.5 for more information about properties.
- *PROCEDURE IsCallFromSequencer() : BOOLEAN*; The *IsCallFromSequencer* method checks if a call originates from the sequencer process that is associated with the composite. The method is used in the event re-scheduling pattern described in section 6.2.4.
- *PROCEDURE Acquire*; acquires the hierarchy lock that is associated with the composite.
- *PROCEDURE Release*; releases the hierarchy lock that is associated with the composite.

6.3.2 Available Components

The following list shows a selection of the available components:

Timer is an invisible component that generates periodic events.

Panel is a visual component that is often used as a container for groups of components.

Label draws a UTF-8 string into its bounding box.

Button generates an event whenever it is clicked with the mouse. It supports styles and images for decoration.

Scrollbar displays, in the form of a slider, an integer value between a given minimum and maximum value. On each change it generates a change event.

TextView displays a rich text. It offers selection and copying to the clipboard but not editing of the text.

Editor wraps a *TextView* component and acts as a controller on the text model.

GenericGrid displays and organizes a generic grid or table structure.

StringGrid is an extension of *GenericGrid* that can display the content of a string grid model.

Clock is an active component that displays the current time.

A large number of additional components are available in the standard Bluebottle system release.

6.4 Conclusions

Both the *Java Swing* and *WinForms* frameworks share a similar synchronisation strategy. Methods of a GUI component should only be called by a single message-dispatching thread. Whenever an external thread needs to modify the state of a GUI component, it has to encapsulate the mutator code into an object or method that is then scheduled for invocation by the message-dispatching thread. The encapsulation into a method is only possible in the .NET framework because of the support for delegates. Another difference of the *WinForms* to the *Java Swing* framework is the additional *InvokeNeeded* property that can be used to determine whether or not the code is running in the message-dispatching thread. In both cases the client application programmer is burdened with explicitly re-scheduling code-flow that leads to the modification of the state of GUI components. While the Bluebottle GUI component framework also uses a specialised thread for synchronisation that is associated with a composite, it differs from Swing/Winforms in the point where the synchronisation takes place. It centralises the thread-safety aspects and moves the decision whether or not a method call must be scheduled on the message-dispatching thread into the components. This disburdens the user of components but requires more thought of the component programmer. Figure 6.8 compares the synchronisation

strategies of *Java Swing* and *WinForms* to the *Bluebottle* GUI component system.
While comparing the thread ID in the *InvokeNeeded* property is rather expensive, even involving kernel calls, the equivalent method in the Bluebottle system is extremely efficient, so that checks on all interface methods of a component are possible without losing too much efficiency. The centralization of the synchronization code is especially beneficial in situations where many different threads are cooperating in a single system.
The proposed design pattern of an active sequencer object in combination with an event synchronisation strategy is a versatile and efficient means for managing synchronisation and locking concerns in a multi-threaded graphical component system. In Bluebottle it is not only used in the XML oriented graphical component framework mentioned in this section, but also in the display space manager as described in section 5. The use of delegate procedure variables combined with the protection of observer methods (see listing 6.2) considerably simplifies the wiring of components.

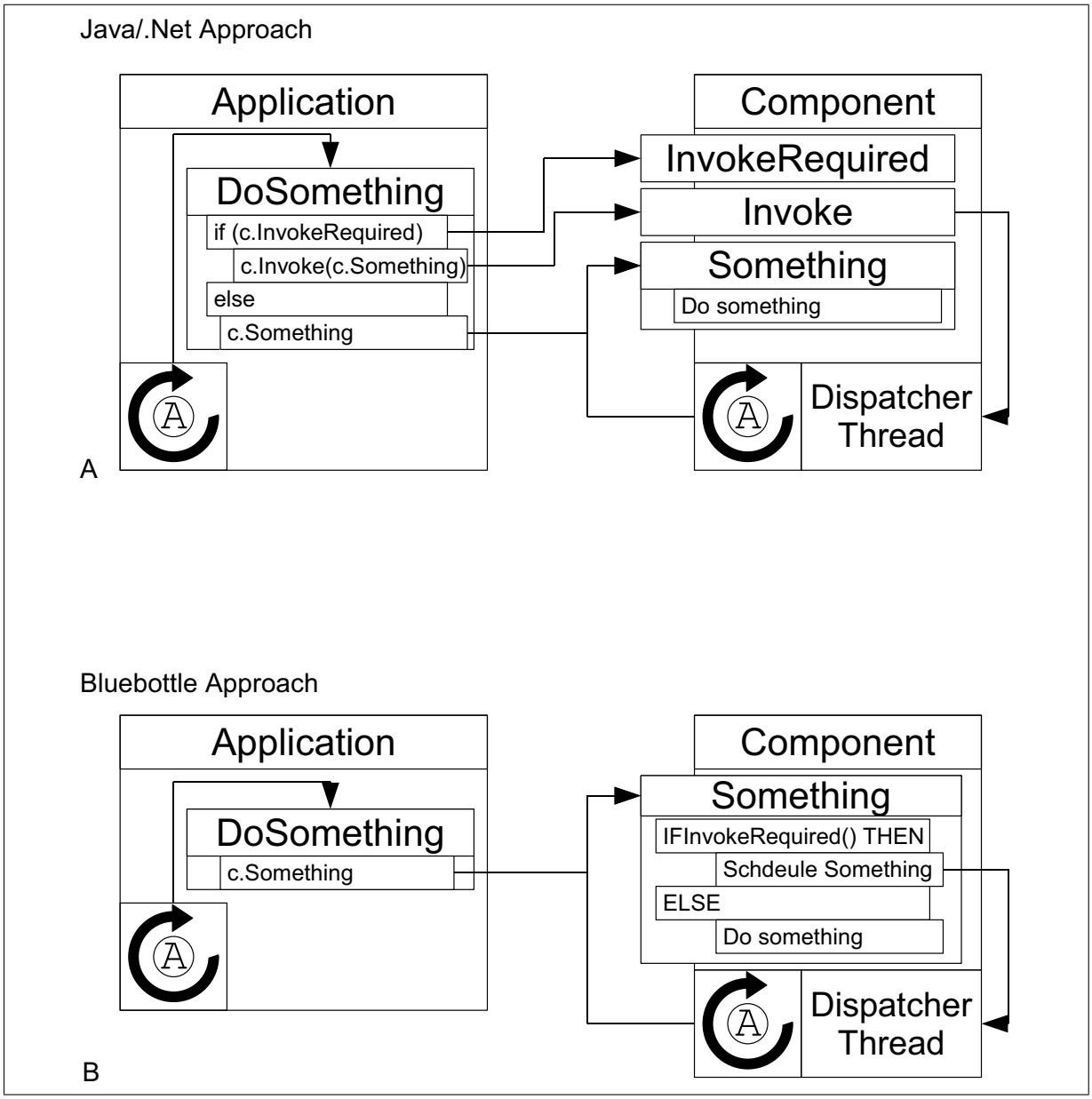

Figure 6.8: Comparison of Synchronisation Strategies

7

The Bluebottle Sound System

Let there be sound, there was sound
— AC/DC 1977

AosSound is the Bluebottle sound input and output system. It offers a plug-in mechanism for physical sound device drivers and specifies a generic software interface that is implemented by the actual sound drivers. The generic sound driver interface offers access to the hardware mixer settings and to input and output channels of sampled digital audio.
The following sections describe the *AosSound* interface as well as the intended mode of operation. Section 7.1 gives a conceptual description how an application program can play or record sound. The technical details are explained in the sections 7.2 which introduces and describes the driver interface - 7.2.1 explaining the content and use of sound buffers that are used in the input and output channels and 7.2.3 introducing the *Channel* interface. Section 7.2.4 gives details on the mixer channel interface that is used to control the hardware mixer settings. Finally, section 7.3 lists a number of sound drivers and applications that use the *AosSound* interface.

7.1 Mode of operation

This section introduces the intended mode of operation of an application program that uses *AosSound*. First, the program needs to get the reference to the sound driver of the hardware it wants to use. All drivers are registered in the *AosSound.devices* plug-in registry. The registry can be enumerated, or a device can be searched for by name. Usually, the application programs will use the default sound device, which can be found by calling the *AosSound.GetDefaultDevice* method. If no sound driver is installed, *AosSound.GetDefaultDevice* blocks until a driver is loaded.

Playing sound To play sound, the application program needs to open a player channel with the *OpenPlayChannel* method of the sound driver. The exact use of the *OpenPlayChannel* method is described in the explanation of the driver interface in section 7.2. Given the player channel, the application program installs a *BufferListener* delegate method with the *RegisterBufferListener* method of the player channel as described in detail in section 7.2.3. The

BufferListener method delegate is called by the driver activity whenever a buffer has been processed completely, so that the application can recycle the buffer. After setting up the *BufferListener*, the application can begin filling sound buffers with sound data. The sound buffer structure is explained in detail in section 7.2.1. The filled buffers are queued for playing in the play channel via the *QueueBuffer* method of the sound channel. The application program should at least queue two buffers of sound data so that the driver can swap between the buffers without delay. Whenever there is no buffer in the play channel, the channel stalls until there are new buffers available. After queueing some filled sound buffers, the application program can start the sound output via the *Start* method. The channel can be paused with the *Pause* method. If paused, sound output is stopped and all the queued buffers remain in the system until *Start* is called again. If the channel is stopped with the *Stop* method, all buffers are returned via the *BufferListener*. If the application program no longer needs the channel, it can be closed with the *Close* method giving the driver the opportunity to release its reserved resources before the channel is collected by the system garbage collection.

Buffering is best done using a bounded buffer pool. The application software uses a dedicated

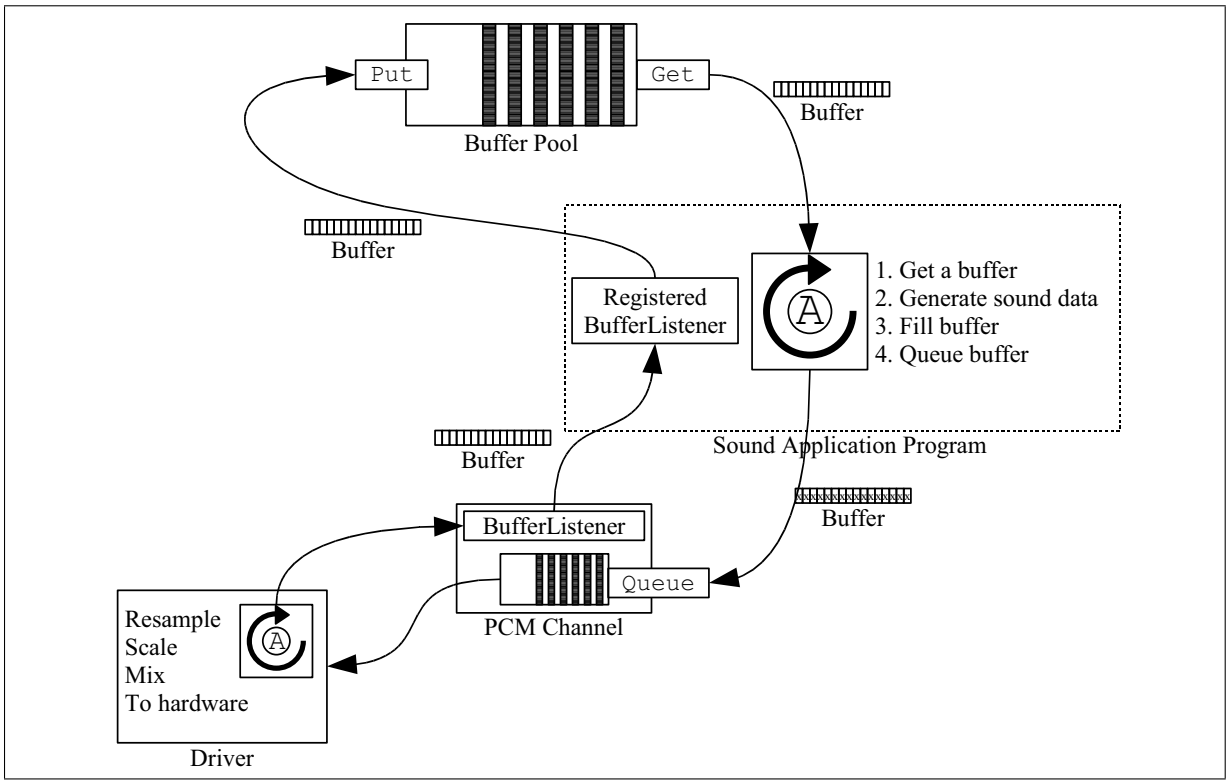

Figure 7.1: Buffer Life-cycle in a Player Application

activity to take buffers from the pool, fill them and queue them in the sound device. Whenever the pool is empty, the activity blocks. This strategy avoids unnecessary memory system overhead and implicitly adapts the application activity to the rate at which the data is needed. Support for pausing the player automatically comes from the buffer that blocks when the channel is paused. *AosSound* offers a bounded buffer pool that is described in section 7.2.2. Figure 7.1 shows the buffer life-cycle in a player application.

Appendix A.3 contains a sample program that implements a sound player using a sound device, play channel and buffer pool.

Recording sound To record sound, the application program needs to open a recorder channel with the *OpenRecordChannel* method of the sound driver. Buffering, starting, pausing and stopping is handled similarly as in the case of playing sound, with the difference that empty buffers are queued in the *QueueBuffer* method of the sound channel and filled buffers are returned to the *BufferListener*. Figure 7.2 shows the buffer life-cycle in a recorder application.

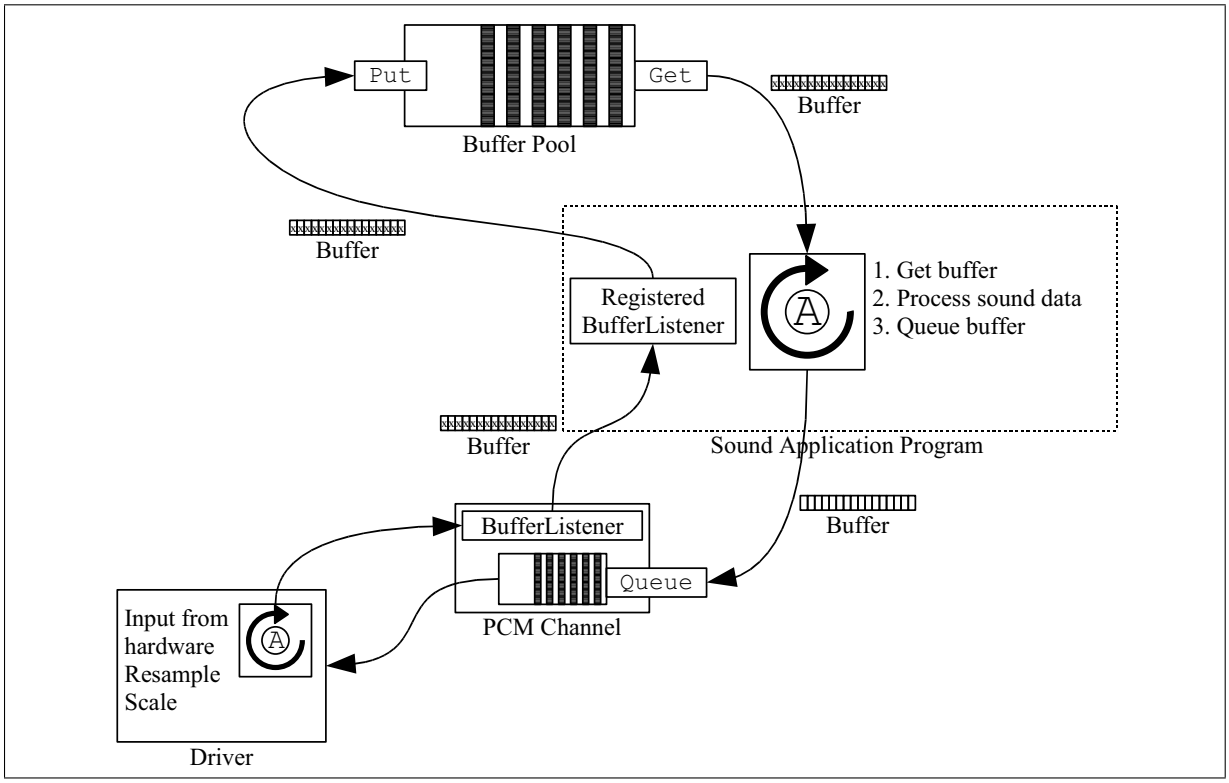

Figure 7.2: Buffer Life-cycle in a Recorder Application

7.2 Sound Driver Interface

The sound driver interface offers a number of methods that can be grouped into several categories:

Device Capability Detection The methods in this category are used to determine the capabilities of the hardware. Programs can use this functionality to adapt themselves to the available hardware. A sound player program could for example check if the selected driver supports 5.1 channel surround sound and, if so, to make use of surround sound information in the sound data or to convert the sound to stereo otherwise. Application programs that do not want to adapt to different sound hardware features can ignore the capability detection methods and directly try

to open the required channel. If the required channel cannot be provided by the driver, the application can quit or renounce from using sound. The capability detection interface is listed below:

- *PROCEDURE NofNativeFrequencies():LONGINT*; returns the number of different frequencies that can be played natively by the hardware D/A converter without up- or down-sampling either in hardware or in software. Today, most commodity hardware internally works with 48kHz D/A converters and use digital signal processing to re-sample the input data in any other sampling rate. In cheap audio hardware, this results in audible re-sampling artifacts. The effective frequencies can be queried with the *GetNativeFrequency* method.
- *PROCEDURE GetNativeFrequency(nr : LONGINT):LONGINT*; returns the natively supported frequency number *nr*.
- *PROCEDURE NofSamplingResolutions():LONGINT*; returns the number of supported sampling resolutions. The sampling resolution defines the number of bits per sample. In commodity hardware the sampling resolutions are usually 8 or 16. The effective sample resolutions can be queried with the *GetSamplingResolution* method.
- *PROCEDURE GetSamplingResolution(nr : LONGINT):LONGINT*; returns the number of bits per sample for the sampling resolution number *nr*.
- *PROCEDURE NofSubChannelSettings():LONGINT*; returns the number of supported sub-channel settings that are possible. A sub-channel setting specifies the number of physical sound output channels that can be individually addressed within one software sound channel. Applications can expect the hardware to always support mono and stereo sound or a software emulation thereof.
- *PROCEDURE GetSubChannelSetting(nr : LONGINT):LONGINT*; returns the sub-channel setting number *nr*. For example 1 represents monophonic sound, 2 stands for stereophonic sound.
- *PROCEDURE NofWaveFormats():LONGINT*; returns the number of supported sound formats. Currently only PCM is supported.
- *PROCEDURE GetWaveFormat(nr : LONGINT):LONGINT*; returns the supported wave format number *nr*. So far only *FormatPCM* is supported.

Opening Sound Channels The methods in this category are used to create either an input or an output sound channel.

- *PROCEDURE OpenPlayChannel(VAR channel : Channel; samplingRate, samplingResolution, nofSubChannels, format : LONGINT; VAR res : LONGINT)*; opens a new channel for sound output. More than one play channel may be opened on one sound device.

This is used for example to play an alarm sound while listening to internet radio or watching a movie. If more than one channel are open, the sound driver mixes the different audio channels in hardware or software. The sound driver also applies the corresponding volume settings of the channels involved by scaling the amplitudes in the sound data. A sound driver usually supports 8 or more playback channels. If the driver cannot open an additional channel, it returns *ResNoMoreChannels* in the *res* parameter. The audio driver also mixes channels with different sampling rates. If the hardware does not provide better support, current software implementations use simple linear interpolation on the wave forms to emulate the re-sampling. The driver returns *ResReducedQuality* in the *res* parameter in cases where the playback quality is reduced by re-sampling. If the playback quality is not too important for the application program, for example if it is just playing an alarm sound rather than high fidelity music, it can ignore the detailed *res* return value and simply rely on the *channel* return value, which is *NIL* if the desired play channel cannot be created.
The *samplingRate* parameter specifies the desired sampling rate of the PCM data. Most sound drivers can play any sampling rate with software re-sampling.
The *samplingResolution* defines the number of bits per sample per sub-channel in the PCM data buffer. The resolutions 8, 16, 24 and 32 are possible, with 8 and 16 bit resolution supported with all current drivers. The PCM data buffer is described in detail in section 7.2.1.
The *nofSubChannels* parameter defines the number of separate sound channels in the PCM data, 1 for mono, 2 for stereo, 4 for quadro and so on. All current drivers support at least mono and stereo.
The *format* parameter describes the encoding format. Currently, only PCM is supported.

- *PROCEDURE OpenRecordChannel(VAR channel : Channel; samplingRate, samplingResolution, nofSubChannels, format : LONGINT; VAR res : LONGINT)*; Opens a new channel for recording. If more than one channel is opened, the sound driver copies the recorded data to all the recording channels, applying the respective volume scaling and sampling rate conversions. Support for more than one recording channel is desirable to give several application programs access to the sound input data. This can be useful for example for running an acoustic environment analyzer, a speech recognition system and a generic sound recorder in parallel. If the driver cannot open an additional channel, it returns *ResNoMoreChannels* in the *res* parameter.
  The return value *channel* contains the recorder channel object. It is *NIL* if the desired channel could not be opened.
  The *samplingRate* defines the desired sampling rate of the PCM data.
  The *samplingResolution* defines the number of bits per sample per sub-channel in the PCM data buffer. The resolutions 8, 16, 24 and 32 are possible, with 8 and 16 bit supported with all current drivers.
  The *nofSubChannels* parameter defines the number of separate sound channels in the PCM data, 1 for mono, 2 for stereo, 4 for quadro and so on. All current drivers support

at least mono and stereo.
The *format* parameter describes the encoding format. Currently only PCM is supported.

Hardware Mixer Channels The hardware mixer takes various sound sources and combines them to a resulting output sound. Each of the sound sources can be scaled by a volume value or muted. Figure 7.3 gives a schematic overview of a hardware sound mixer.

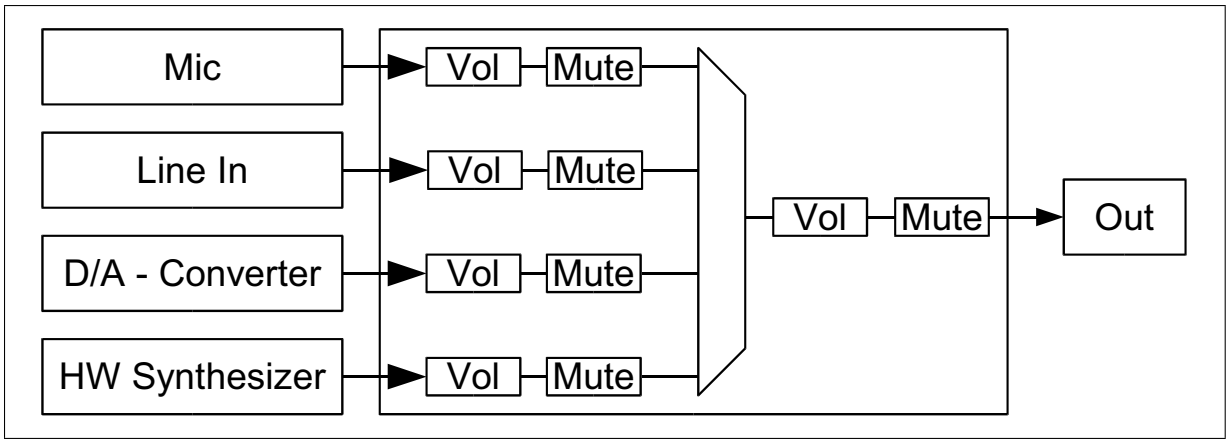

Figure 7.3: Hardware Sound Mixer

The following methods are used to control the hardware mixer:

- *PROCEDURE RegisterMixerChangeListener(mixChangedProc : MixerChangedProc)*; registers a mixer change listener that is called if a mixer channel setting is changed.
- *PROCEDURE UnregisterMixerChangeListener(mixChangedProc : MixerChangedProc)*; unregisters a previously installed mixer change listener.
- *PROCEDURE GetNofMixerChannels() : LONGINT*; returns the number of available mixer channels.
- *PROCEDURE GetMixerChannel(channelNr: LONGINT; VAR channel: MixerChannel)*; returns a channel instance corresponding to channel number *channelNr*. The mixer channel interface is defined in details in section 7.2.4. The channels number 0 and 1 are predefined. Channel 0 represents the master output and channel 1 is the master input volume. Numbers above 1 are hardware dependent mixer channels that can be identified by a channel name (see *GetName* in 7.2.4) assigned by the driver.

7.2.1 Sound Buffers

The sound *Buffer* object contains the fields *len* and *data*, where *data* is a pointer to a character array containing the PCM data to be played. The sample data for the channels is interleaved within the buffer data. The *len* field specifies how many bytes are valid data within the buffer. The buffer data must start with the first sample data byte of the first channel and end with the last sample data byte of the last channel. Fig. 7.4 shows the interleaving pattern in several cases

of 8 and 16 bit buffers with one, two and four sub-channels.

To guarantee seamless playback or recording, the sound driver should always have at least

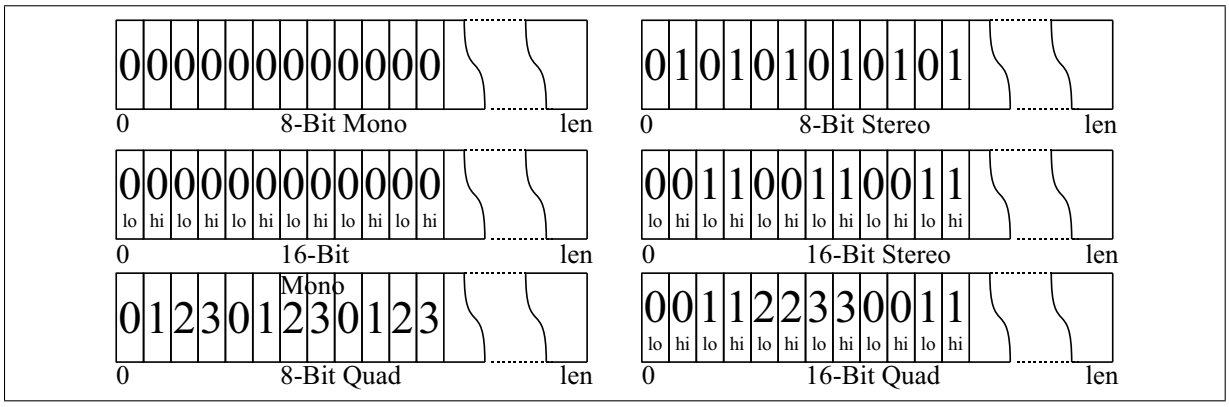

Figure 7.4: Examples of PCM Buffer Interleaving Patterns

one spare buffer available in its queue for switching. The buffer and queue sizes depend on the application. Interactive applications should use small buffers with a relatively small buffer queue to be able to quickly change the output as a reaction to interaction events. Less interactive applications such as radio or music players can use larger buffers and queues thus making buffer queueing less time-critical.

7.2.2 Buffer Pool

Because all sound playing or recording applications need sound buffer management, *AosSound* offers an implementation of a bounded buffer pool. The constructor of the buffer pool takes a parameter that specifies the capacity of the buffer pool, that is the number of sound buffers that can be queued. The buffer pool should be initialised with at least two buffers since this is the minimum number that allows smooth sound playback. After the *BufferPool* object is created, the playing or recording application creates a number of sound buffers and adds them into the buffer pool with the *Add* method. The number of buffers that is added to the buffer pool must be smaller or equal to the maximum capacity of the buffer pool, otherwise the buffer pool is blocked. The sound playing or recording process removes the sound buffers from the buffer pool with the *Remove* method that blocks when no buffer is left in the pool. The sound channel returns the buffers to the buffer pool either directly or indirectly via the sound application. In the recording case the buffer is normally added via the sound recording application since the returned sound buffer needs to be processed. The *BufferPool* interface looks like this:

- *PROCEDURE &Init(n: LONGINT)*; The constructor *Init* takes the capacity of the bounded buffer as a parameter.
- *PROCEDURE Add(x: Buffer)*; adds the buffer *x* to the buffer pool. The method blocks if the buffer pool is full.
- *PROCEDURE Remove(): Buffer*; removes a buffer from the buffer pool and blocks if the pool is empty.

Appendix A.3 contains a program example that uses a buffer pool to create a sound player.

7.2.3 PCM Channel Interface

The PCM channels offer the following interface for input and sound output:

- *PROCEDURE GetChannelKind() : LONGINT*; returns the kind of the channel. Possible return values are *ChannelPlay* or *ChannelRecord*.
- *PROCEDURE SetVolume(volume : LONGINT)*; sets the relative volume of a sound channel. The *volume* parameter is interpreted as an 8.8 fixed point value that is used as a scaling factor when mixing the content of different PCM buffers before output.
- *PROCEDURE GetVolume() : LONGINT*; returns the current volume of the sound channel.
- *PROCEDURE GetPosition() : LONGINT*; returns the position in samples that have been recorded or played through the channel.
- *PROCEDURE RegisterBufferListener(bufferListener : BufferListener)*; registers a buffer listener method that is called whenever a buffer has been processed. In the case of a playing channel, *BufferListener* can recycle the buffer to fill in more data to be played. For a recording channel, the data contained in the buffer needs to be stored or processed otherwise.
- *PROCEDURE Start* starts the channel.
- *PROCEDURE QueueBuffer(x : Buffer)*; queues a sound buffer for either playing or recording. More than one buffer should be queued for a seamless switch to the next buffer. Whenever a buffer is processed (either played completely or filled completely with recorded data) a registered *BufferListener* is called by the sound driver, returning the buffer to the application code.
- *PROCEDURE Pause*; pauses the channel. If paused, no buffers are returned and the playing or recording continues at the same buffer position when *Start* is called again.
- *PROCEDURE Stop*; stops the channel. All buffers are returned to the *BufferListener*.
- *PROCEDURE Close*; closes the channel and allows the driver to release all resources that have been reserved for the channel. A closed channel cannot be re-opened.

7.2.4 Mixer Channel Interface

Mixer channel objects can be acquired from the sound device driver for each mixer channel it implements. The mixer channel offers the following access methods:

- *PROCEDURE GetName(VAR name:ARRAY OF CHAR)*; returns the name of the mixer channel as an UTF-8 string. The name is used for identifying the channel.
- *PROCEDURE GetDesc(VAR desc:ARRAY OF CHAR)*; returns a description of the channel as a UTF-8 string.
- *PROCEDURE SetVolume(volume : LONGINT)*; sets the volume of the channel. 0 is silent 255 is the maximum value.
- *PROCEDURE GetVolume() : LONGINT*; returns the volume of the channel.
- *PROCEDURE SetMute(mute : BOOLEAN)*; sets the mute state of the channel. If *mute* is *true*, the channel is silent but retains the previous volume.
- *PROCEDURE GetIsMute() : BOOLEAN*; tests if a channel is muted.

7.3 AosSound Applications

A fair amount of work has been carried out as a proof of concepts on the basis of the *AosSound* framework:

Drivers The following sound drivers have been implemented as an extension of the *AosSound.Driver* object. In the text they are referred to as *current drivers*:

- *Yamaha YMF754* sound chip driver by M. v. Tessin [110].
- *ENSONIQ 137x* sound chip driver by C. Heinzer [45].
- *Intel AC'97* compatible sound driver by K. Jonsson [56]. AC'97 compatible sound cards are widely-used in on-board sound systems of current mainstream computers.

Sound Applications The following sound players and recorders have been implemented as student projects:

- An *MP3 player* has been written by C. Dornbierer [24]. MP3 is the de facto standard for lossy storage of digitised music.
- An *OGG Vorbis* player has been written by Ch. Wassmer [114]. OGG Vorbis is an open and patent-free standard for lossy storage of digitised music. Its compression efficiency and quality is competitive with MP3.
- A *wav*-file player and recorder has been written by M.v.Tessin [110]. Wave files are normally used for the lossless stored representation of digitised sounds and music.
- A *Speex* decoder and encoder has been written by F. Röthenbacher [99]. Speex is a compression format that is sometimes used in "voice over IP".
- Sound support has been added to the Bluebottle DivX player [109] by U. Müller [77].

8

Abstract Encoder and Decoder Framework

Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte.
I have made this longer, because I have not had the time to make it shorter.

— Blaise Pascal (1623 - 1662) (4.12.1656)

The abbreviation *Codec* stands for *co*der/*dec*oder. Codecs in general transform data streams mainly for compression, transmission or encryption. *AosCodecs* is the Bluebottle codec framework. It offers abstract interfaces and repositories for specialised classes of encoders and decoders in the areas of *video*, *audio*, *still images*, *texts* and *cryptography*.
AosCodecs, together with the graphics (chapter 4) and the sound system (chapter 7), forms the Bluebottle *multimedia framework*. It decodes multimedia data that is then visualised with the graphics system and made audible through the sound system. Figure 8.1 shows the position of the *AosCodecs* framework between the *AosIO* system and a multimedia application program that uses the graphics and sound frameworks.
Many multimedia data formats such as *AVI*[72] or *OGG*[119] are containers that specify how the encoded audio and video data is organised and stored but do not specify how the data has to be encoded. One and the same encoder or decoder can thus be used with different container formats. To ensure the interoperability of the specific encoders and decoders with different container formats, *AosCodecs* defines an abstract interface and repository for audio/video demultiplexer objects.
In the following, section 8.1 discusses how an application program can obtain an encoder or decoder, and how the system finds the respective objects. Later sections describe the programming interfaces of the different encoder and decoder classes that are currently supported in Bluebottle.
A complete programming example that implements an MP3 player using the *AosCodecs* framework can be found in appendix A.3.

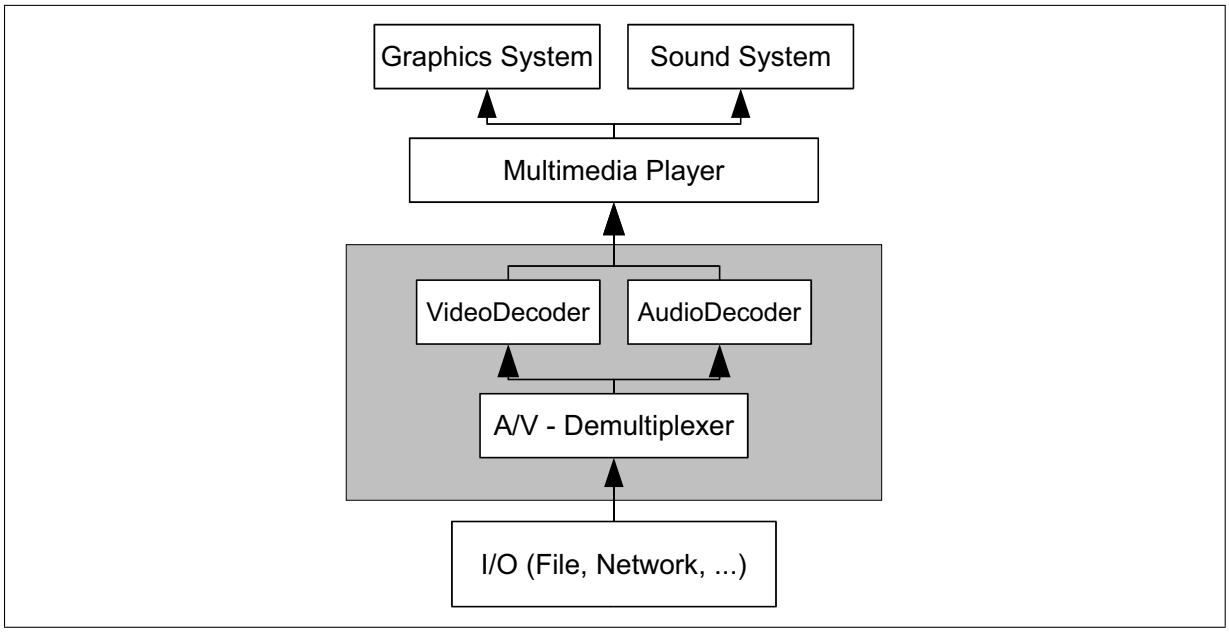

Figure 8.1: AosCodecs within the System

8.1 Obtaining an Encoder or Decoder

Application programs that need to encode or decode a data stream can obtain the required codec through a generic generator procedure for the respective codec class. Each different encoder or decoder class has its own generic generator procedure that takes the name of the respective codec as the only parameter and returns a codec object instance or *NIL* if the codec cannot be found. Each codec class has its own task specific programming interface that is described in later sections.
While codec object instances cannot be shared between threads it is possible that several threads use different instances of the same codec at the same time.
The following lists the names of the generic generator procedures for each codec class:

Audio GetAudioEncoder/GetAudioDecoder

Video (GetVideoEncoder)/GetVideoDecoder

Image GetImageEncoder/GetImageDecoder

Text GetTextEncoder/GetTextDecoder

Cryptography GetCryptoEncoder/GetCryptoDecoder

Multiplexer (GetAVMultiplexer)/GetAVDemultiplexer

Currently there is no video encoder or multiplexer implemented for Bluebottle. The generic generator procedures for each of the codec classes uses the *Codecs* section in the system configuration (*AosConfig.XML*) to translate the name of the requested codec into the name of the respective specific generator procedure. Listing 8.1 shows an excerpt of *AosConfig.XML*, giving one example of an available encoder or decoder per section. At the time of writing, there is

no video encoder available for Bluebottle. The respective programming interfaces is therefore not discussed in the following. If the name of the generator procedure is found, the containing module is dynamically loaded and the respective generator procedure is called. The specific generator procedure's return value is then passed via the generic generator procedure to the calling application program if it contains an object of the required codec class.

The application program can then open the returned object on an input or output stream and use it according to its programming interface to decode or encode data.

8.2 Audio/Video Demultiplexer

In multimedia container formats, the audio and video streams are normally multiplexed, interleaving video frames with audio samples. For efficiency reasons and also for streaming, the interleaving video and audio streams are often arranged in a way that time matching frames and samples are stored near each other within the container stream. Since there is no worldwide-agreed-on single movie container format, *AosCodecs* defines abstract interfaces for generic multiplexers and demultiplexers that are implemented by concrete movie container format implementations. This abstraction makes it possible to use one encoder or decoder for all container formats. Unfortunately some demultiplexers cannot be used on input streams but only work with direct access files because the respective formats are designed to require random access to data within gigabytes.
An application program that needs to demultiplex input data can obtain a suitable demultiplexer object by calling the *AosCodecs.GetAVDemultiplexer* procedure with the registered format name. If the format name is found in the *Codecs.AVDemultiplexer* section in the system configuration, the respective *AosCodecs.AVDemultiplexer* object instance is created and returned. Otherwise, the generator procedure returns NIL. If a demultiplexer object is returned, it can be opened on an *AosIO* stream or on a file. As mentioned above, some demultiplexers such as the *AVI* demultiplexer can only be opened on files since they require seeking in large amounts of data. After opening the demultiplexer, the application program can query the demultiplexer object for contained content-streams. The content-streams can be opened as streams and be decoded with the appropriate *AosCodecs.AudioDecoder* or *AosCodecs.VideoDecoder*.

8.2.1 Programming interface

This section describes the programming interface of the *AosCodecs.AVDemultiplexer* object class.

- *PROCEDURE Open(in : AosIO.Reader; VAR res : LONGINT)*; opens a demultiplexer on the input stream *in*. The return value *res* contains the error code of the operation, where *res* = *AosIO.Ok* indicates success. For many AVDemultiplexers, such as the AVI demultiplexer, the *in* reader needs to be an *AosCodecs.FileInputStream*.

```
<!-- Codecs -->
<Section name="Codecs">
  <!-- Multiplexer -->
  <Section name="Multiplexer">
  </Section>

  <!-- Demultiplexer -->
  <Section name="Demultiplexer">
    <Setting name="AVI" value="AVIDecoder.AVIDemuxFactory" />
  </Section>

  <!-- Encoders -->
  <Section name="Encoder">
    <Section name="Text">
      <Setting name="UTF-8" value="AosTextUtilities.UTF8EncoderFactory" />
    </Section>
    <Section name="Image">
      <Setting name="BMP" value="AosBMPCodec.EncoderFactory" />
    </Section>
    <Section name="Video">
    </Section>
    <Section name="Audio">
      <Setting name="WAV" value="AosWAVCodec.EncoderFactory" />
    </Section>
    <Section name="Crypto">
      <Setting name="AES" value="AosAES.EncoderFactory" />
    </Section>
  </Section>

  <!-- Decoders -->
  <Section name="Decoder">
    <Section name="Text">
      <Setting name="Oberon" value="AosTextUtilities.OberonDecoderFactory" />
    </Section>
    <Section name="Image">
      <Setting name="PNG" value="AosPNGDecoder.Factory" />
    </Section>
    <Section name="Video">
      <Setting name="DivX" value="DivXDecoder.DivXDecoderFactory" />
    </Section>
    <Section name="Audio">
      <Setting name="MP3" value="MP3Decoder.MP3DecoderFactory" />
    </Section>
    <Section name="Crypto">
      <Setting name="AES" value="AosAES.DecoderFactory" />
    </Section>
  </Section>
</Section>
```

Listing 8.1: Codec Configuration in XML

- *PROCEDURE GetNumberOfStreams() : LONGINT*; returns the number of streams that are contained in the container.

- *PROCEDURE GetStreamType(streamNr : LONGINT) : LONGINT*; returns the type of the content stream specified with the *streamNr* parameter. The result can be *AosCodecs.STError*, if the stream selected stream does not exist, *AosCodecs.STUnknown*, if the type of the content stream is not known, *AosCodecs.STAudio*, if the stream contains audio data or *AosCodecs.STVideo* if the stream contains video data.

- *PROCEDURE GetStreamInfo(streamNr : LONGINT) : StreamInfo*; returns an information object about the stream number *streamNr*. The *StreamInfo* record contains information about the content stream, such as for example, content type, seekabiliy and length.

- *PROCEDURE GetData(streamNr : LONGINT; VAR buf: ARRAY OF CHAR; ofs, size, min: LONGINT; VAR len, res: LONGINT)*; reads a maximum of *size* bytes from content stream number *streamNr* and stores it in the buffer *buf* starting from offset *ofs*. The procedure blocks until at least *min* bytes are read or an error occurs. It returns the number of bytes that have actually been read in the return parameter *len* and the result of the operation in *res*. A result value *res* of *AosIO.Ok* means the operation was successful. Otherwise the value contains an error code.
  The rather complicated looking interface is not intended for direct use by an application program but rather to open an *AosCodecs.DemuxStream* on top of it, that in turn offers a standard *AosIO.Reader* interface. The *GetStream* method returns such an *AosCodecs.DemuxStream*.

- *PROCEDURE GetStream(streamNr : LONGINT) : DemuxStream*; returns an *AosCodecs.DemuxStream* on stream number *streamNr*, that implements the *AosIO.Reader* interface.

- *PROCEDURE SetStreamPos(streamNr : LONGINT; seekType : LONGINT; pos : LONGINT; VAR itemSize : LONGINT; res : LONGINT)*; sets the position of the stream number *streamNr* to position *pos* with the seek granularity *seekType*. The semantics of *seekType* and *pos* is discussed in 8.3. The procedure returns the size in bytes of the item, the stream was positioned to, if available or -1 otherwise. The item size can for example contain the size of a frame. The result value *res* returns *AosIO.Ok* if the seek operation was successful.

8.3 Streams

The stream abstraction is used as an intermediate layer between *n* data-sources and *m* data-consumers. It allows $n*m$ different combinations while requiring only $n+m$ implementations. Figure 8.2 shows a selection of possible data-source to data-consumer connections. Figure 8.3 shows the use of streams in the context of two different player configurations .
Data streams in general offer access to sequential data. Unlike files it is normally not possible to randomly access data. Examples of streamable but non-seekable data sources are:

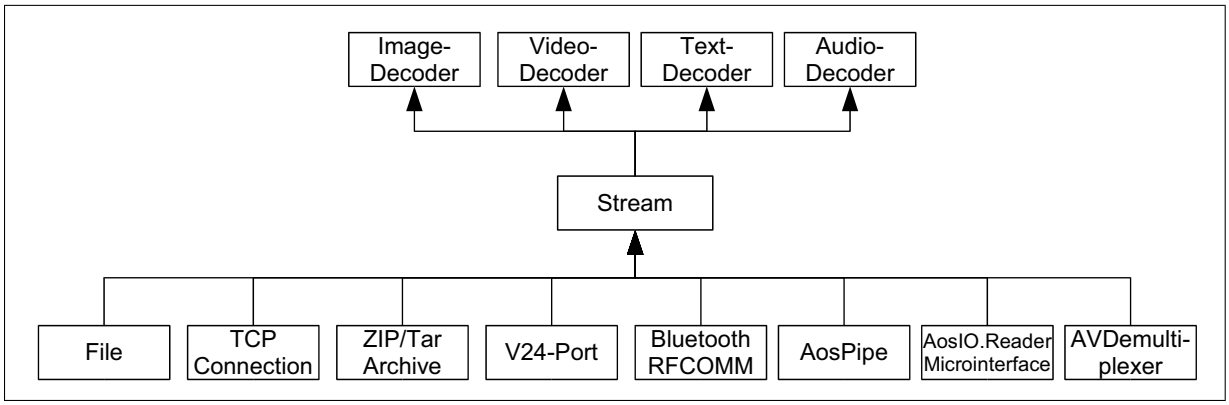

Figure 8.2: InputStream as Connection between Data-Sources and Data-Consumer

- Live data streams from physical sensors.
- Data streams without a separate control connection. For example a pure TCP connection.
- Demultiplexers if opened on non-seekable data sources.
- Archives if compressed and the compression does not allow seeking.

If random access to this kind of data is required, it has to be copied to a file beforehand. Examples of streamable and seekable data sources are:

- Files
- Specialised seekable multimedia streams
- Demultiplexers that are themselves opened on seekable data sources
- Archive files if they are uncompressed or the compression method allows for seeking

To unify seekable and non-seekable data sources the *AosIO* interface has been extended by the procedures *CanSetPos*, *SetPos* and *Reset*.

While positionable *AosIO.Reader*s can only be seeked on byte-level, the object extension *AosCodecs.InputStream* offers the possibility to be seeked on the level of higher structures like frames.

8.4 Audio

An audio decoder takes a stream of compressed digital audio and decodes it into a number of PCM encoded buffers of sampled digital audio. The audio encoder takes PCM encodec buffers and generates a stream of encodec audio.

8.4.1 Decoder

This section gives programming details about the *AosCodecs.AudioDecoder*. Section 8.9 and figure 8.3 show the use of *AosCodecs.AudioDecoder* in context.
An *AudioDecoder* is created with the module procedure *GetAudioDecoder* which takes the name of the decoder as input and returns an instance of the desired *AudioDecoder* object class. The mapping of an *AudioDecoder* name to its generator procedure is done by a lookup in the *Codecs.Decoder.Audio* section in *AosConfig.XML*.
AosCodecs.AudioDecoder offers the following interface (some administrative methods are omitted):

- *PROCEDURE Open(stream : InputStream; VAR res : LONGINT)*; opens the decoder on an input stream. Open must be called before any other procedure of the decoder is called. *res* contains the error code of the operation, with *res* = *AosIO.ResOk* to indicating success.

- *PROCEDURE GetAudioInfo(VAR nofChannels, samplesPerSec, bitsPerSample : LONGINT)*; queries information about the audio content. The return parameter *nofChannels* contains the number of audio channels, for example 1 for mono, 2 for stereo and so on. *samplesPerSec* returns the number of samples per second per channel that are stored in the sound data. The *bitsPerSample* return parameter contains the resolution. For compatibility with the sound system, values should be divisible by 8 with no remainder, i.e. modulo 8 = 0.

- *PROCEDURE CanSeek() : BOOLEAN*; specifies if the decoder can seek on the audio stream. If the decoder replies with true, it can seek by sample or by millisecond.

- *PROCEDURE GetCurrentSample() : LONGINT*; returns the ordinal number of the next sample to be filled into the buffer when FillBuffer is called.

- *PROCEDURE GetCurrentTime() : LONGINT*; returns the time in milliseconds of the next sample to be filled into the buffer when FillBuffer is called.

- *PROCEDURE SeekSample(sampleNr : LONGINT; goKeySample : BOOLEAN; VAR res : LONGINT)*; seeks the sample that is next to be filled into the sound-buffer.

- *PROCEDURE SeekMillisecond(millisecond : LONGINT; goKeySample : BOOLEAN; VAR res : LONGINT)*; seeks the time in milliseconds of the sample that is next to be filled into the sound-buffer.

- *PROCEDURE FillBuffer(buffer : AosSound.Buffer)*; fills the *AosSound.Buffer* with the next samples from the current position and increases the position. The number of sample bytes that are filled in the buffer is less than or equal to the size of the *AosSound.Buffer*.

- *PROCEDURE HasMoreDate() : BOOLEAN*; returns true until the end of the input stream has been reached.

8.4.2 Encoder

This section describes the *AosCodecs.AudioEncoder* interface. Because the current audio encoder interface only supports best quality encoding, its interface is relatively simple:

- *PROCEDURE Open(out : AosIO.Writer; sRate, sRes, nofCh: LONGINT; VAR res : LONGINT)*; opens an audio encoder. It takes the stream where the encoded audio is written to as the parameter *out*. The parameters *sRate*, *sRes* and *nofCh* define the sampling rate, sampling resolution and number of channels of the audio data.
- *PROCEDURE Write(buffer : AosSound.Buffer; VAR res : LONGINT)*; adds an *AosSound* buffer to the audio stream. *Note:* the *Write* procedure is time-independent, it can be called independent from the audio-time.
- *PROCEDURE Close(VAR res : LONGINT)*; defines the end of the audio data. The encoder can finish the encoding and flushes the out stream.

8.5 Video

A video decoder takes a stream of typically compressed video frames and offers an interface for extracting single video frames. Because there is currently no video encoder implementation for Bluebottle, the *VideoEncoder* class is not discussed here.

8.5.1 Decoder

This section gives programming details for *AosCodecs.VideoDecoder*. Section 8.9 and figure 8.3 show the use of *AosCodecs.VideoDecoder* in context.
The task of a video decoder is giving access to the individual frames that make up the video stream. Depending on the method of encoding, the decoding of a single frame can take a significant amount of computation time. In most video encoding formats the decoding of an individual frame depends on preceding and following frames, so that random access to images is extremely inefficient. Most encodings therefore regularly insert frames of a special class that can be decoded independently. These special frames are called *key-frames* and can be used to seek within the video stream.
A *VideoDecoder* is created with the module procedure *GetVideoDecoder* that takes the name of the decoder as an input parameter and returns an instance of the desired *VideoDecoder* object class.
An *AosCodecs.VideoDecoder* offers the following programming interface:

- *PROCEDURE Open(s : InputStream; VAR res : LONGINT)*; opens the decoder on an input stream. Open must be called before any other method in the decoder is called. *res* contains the error code of the operation, with *res* = *AosIO.ResOk* indicating success.
- *PROCEDURE GetVideoInfo(VAR microsecondsPerFrame, width, height : LONGINT)*; returns the number of microseconds per frame and the width and height of the full video

frames.

- *PROCEDURE CanSeek() : BOOLEAN*; specifies if the decoder can seek on the video stream. If the decoder replies affirmatively, it can be seeked by frame or by millisecond.
- *PROCEDURE GetCurrentFrame() : LONGINT*; returns the ordinal number of the next frame to be rendered in the *Render* method.
- *PROCEDURE GetCurrentTime() : LONGINT*; returns the point of time in milliseconds of the next frame to be rendered in the *Render* method.
- *PROCEDURE SeekFrame(frameNr : LONGINT; goKeyFrame : BOOLEAN; VAR res : LONGINT)*; seeks the frame to be rendered next. If *goKeyFrame* is set, it is acceptable for the decoder to jump to the key-frame that is nearest to *frameNr*. This is used in interactive settings where reaction time rather than frame accuracy is important. The frame finally chosen can be found with the *GetCurrentFrame* method.
- *PROCEDURE SeekMillisecond(millisecond : LONGINT; goKeyFrame : BOOLEAN; VAR res : LONGINT)*; seeks the frame to be rendered next. If *goKeyFrame* is set, it is acceptable for the decoder to jump to the key-frame that is nearest to *millisecond*. This is used in interactive settings where reaction time rather than millisecond accuracy is important. The time finally chosen can be found with the *GetCurrentTime* method.
- *PROCEDURE Next*; Progresses by one frame. The rendering is delayed until the *Render* method is called but the internal state is advanced. The *Next* and *Render* methods are separated so that a multimedia player can catch up on lost time by skipping one or more frames. *Next* is also implemented if the decoder or stream is not seek-able.
- *PROCEDURE Render(img : Raster.Image)*; Renders the current frame into the image *img*.
- *PROCEDURE HasMoreDate() : BOOLEAN*; returns true until the end of the input stream has been reached.

8.6 Still Images

The still image codecs encode and decode still images.

8.6.1 Decoder

*ImageDecoder*s are created with the module procedure *GetImageDecoder* which takes the name of the decoder as input and returns an instance of the desired *ImageDecoder* object class. The mapping of *ImageDecoder* names to the respective generator procedures is defined in the

Codecs.Decoder.Image section in *AosConfig.XML*.

AosCodecs.ImageDecoder implements the following interface:

- *PROCEDURE Open(s : InputStream; VAR res : LONGINT)*; opens the decoder on an input stream. Open must be called before any other method of the decoder is called. *res* contains the error code of the operation. A *res* value of *AosIO.Ok* indicates success.
- *PROCEDURE GetImageInfo(VAR width, height, format, maxProgressionLevel : LONGINT)*; returns information about the image that is being decoded. If the decoder is able to decode the image partially to improve the rendering speed, a *maxProgressionLevel* greater than zero is returned. It is then possible for the application program to select the desired decoding level with the *SetProgressionLevel* method. The fastest decoding mode is on progression level zero. The progression level *maxProgressionLevel* denotes the best available decoding quality.
- *PROCEDURE SetProgressionLevel(progressionLevel: LONGINT)*; specifies the maximum *progressionLevel* to be used by the decoder. If *SetProgressionLevel* is not called, the default *progressionLevel* is set to the maximal value of *progressionLevel* so that the image is decoded completely. If the *progressionLevel* is reset to a lower level than a previously rendered image, the new level will be ignored by the decoder.
- *PROCEDURE GetNativeImage(VAR img : Raster.Image)*; returns the image in the best fitting *Raster.Format* rendered at the given *progressionLevel*.
- *PROCEDURE Render(img : Raster.Image)*; renders the image into the given *Raster.Image* at the given *progressionLevel*.

8.6.2 Encoder

*ImageEncoder*s are created with the module procedure *GetImageEncoder* which takes the name of the encoder as an input and returns an instance of the desired *ImageEncoder* object class. The mapping of *ImageEncoder* names to the respective generator procedures is defined in the *Codecs.Encoder.Image* section in *AosConfig.XML*.

AosCodecs.ImageEncoder offers the following programming interface:

- *PROCEDURE Open(s : AosIO.Writer)*; opens the encoder on an output stream.
- *PROCEDURE SetQuality(q : LONGINT)*; sets the desired encoding quality as a compression ratio. Lossless encoders for still image formats ignore the quality setting.
- *PROCEDURE WriteImage(img : Raster.Image)*; encodes the image *img* and writes the result to the output stream.

8.7 Text

The text codecs are responsible for reading and writing texts from and to streams. The standard text codecs of the Bluebottle system are implemented in the module *AosTextUtilities*.

8.7.1 Decoder

The text decoder decodes a text from a stream. Its programming interface only consists of two procedures *Open* and *GetText*. *TextDecoder* instances are obtained by the module procedure *GetTextDecoder* which takes the name of the decoder as input and returns an instance of the desired *TextDecoder* object class. The mapping of *TextDecoder* names to the respective generator procedures is defined in the *Codecs.Decoders.Text* section in *AosConfig.XML*.

- *PROCEDURE Open(in : AosIO.Reader; VAR res : LONGINT)*; opens a text from a stream. A result value *res* of *AosIO.Ok* indicates success. If the text format was not recognised, the result value contains an error code.
- *PROCEDURE GetText() : AosTexts.Text*; returns a new *AosTexts.Text* instance containing the text that has been decoded from the input stream.

8.7.2 Encoder

The text encoder takes an *AosTexts.Texts* and encodes it to an *AosIO* stream. *TextEncoder* instances are obtained by the module procedure *GetTextEncoder* which takes the name of the encoder as input and returns an instance of the desired *TextEncoder* object class. The mapping of *TextEncoder* names to the respective generator procedures is defined in the *Codecs.Encoders.Text* section in *AosConfig.XML*.

- *PROCEDURE Open(out : AosIO.Writer)*; opens the encoder instance on the *AosIO.Writer out*.
- *PROCEDURE WriteText(text : AosTexts.Text; VAR res : LONGINT)*; does the actual work of encoding a text. A result value *res* of *AosIO.Ok* indicates success.

8.8 Cryptography

Cryptography codecs are responsible for encoding and decoding encrypted *AosIO* data-streams. Working as simple codecs on unstructured data streams, their interface is relatively simple.

8.8.1 Decoder

The *AosCodecs.CryptoDecoder* has the following programming interface:

- *PROCEDURE Open(in: AosIO.Reader; VAR res: LONGINT)*; opens a decoder on the *AosIO.Reader in*.

- *PROCEDURE SetKey(VAR src: ARRAY OF CHAR; pos, keybits: LONGINT)*; sets the decryption key that is stored in array *src*, starting from position *pos* and using *keybits* bits.
- *PROCEDURE GetReader(): AosIO.Reader*; returns an *AosIO.Reader* from which the decoded data is read.

8.8.2 Encoders

The *AosCodecs.CryptoEncoder* has the following programming interface:

- *PROCEDURE Open(out: AosIO.Writer)*; opens an encoder that writes on the *AosIO.Writer out*.
- *PROCEDURE SetKey(VAR src: ARRAY OF CHAR; pos, keybits: LONGINT)*; sets the encryption key that is stored in array *src*, starting from position *pos* and using *keybits* bits.
- *PROCEDURE GetWriter(): AosIO.Writer*; returns an *AosIO.Writer* to which the data to be encoded is writen to. If the writer is flushed with the *Update* method, the *out* stream will be fushed, too. This is especially important to keep in mind when using block ciphers that in this situation need to pad the data to reach a full block.

8.9 Principle of Operation

To ”play” multimedia streams or files, multimedia player programs need to know the type of the contents of the source data. In the case of simple streams, this can either be hard-coded or deduced from file extension or mime-type. If the source data is a container format, the content types of the contained streams can be queried from the appropriate demultiplexer object.
In the case of playing a container format, the player application opens the respective demultiplexer on the source data-stream and a new *AosCodecs.InputStream* on all of the multiplexed content-streams it is interested in.
The player then figures out what codec is needed to decode the data and generates the respective decoder instance that it then connects to the input stream.
If the source stream directly contains encoded audio or video data, the demultiplexer step is skipped, and the appropriate decoder is opened directly on the source stream.
If the underlying data source, the optional demultiplexer and the decoder all support seeking, the player application can offer seeking through its user interface.
Through the decoder(s), the multimedia player accesses the audio samples and/or video frames and presents them on the screen and/or audio hardware. The multimedia player is responsible for playing the video frames and audio samples synchronised and in the correct speed. It is the only part of the multimedia player setup that relies on the real-time clock. The decoders are real-time independent, their task is to deliver the required video frames or audio samples as fast as possible.

Figure 8.3 shows two sample setups of a multimedia player interacting with the *AosCodecs* framework. The first setup (left) shows a multimedia player playing a movie from a file. Since the movie file contains audio as well as video, a demultiplexer is used. The second setup (right) shows the simpler case of a multimedia player playing an audio stream from a TCP connection. A documented example of a simple media player can be found in appendix A.3.

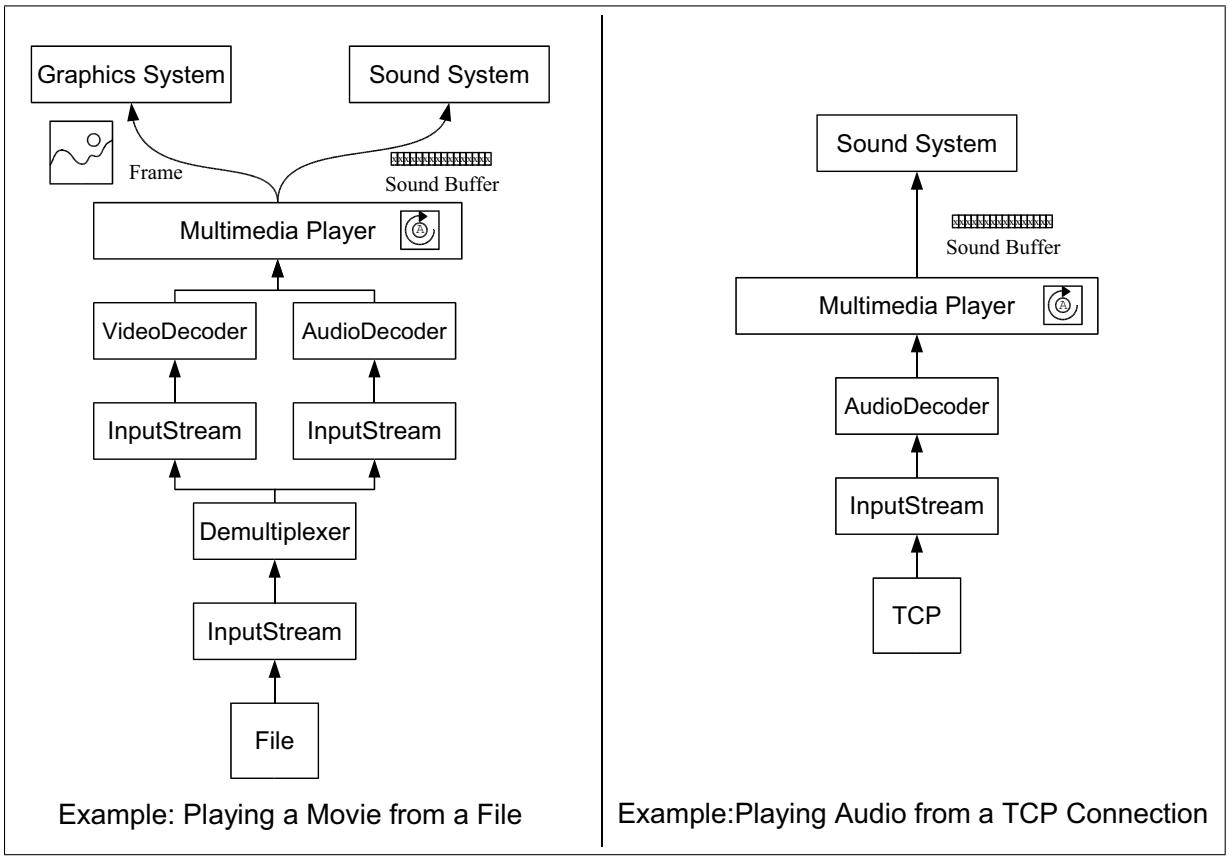

Figure 8.3: Interactions of Multimedia Player, Decoder, Demultiplexer and InputStreams

8.10 Available Codecs

The following list shows the codecs that are currently available for the Bluebottle system:

Video Codecs

DivX The DivX decoder for Bluebottle was originally written by T. Trachsel [109] and adapted to the AosCodecs framework by U. Müller [77].

MPEG-2 A decoder for MEPG-1 and MPEG-2 has been implemented by Y. Weber [115].

Audio Codecs

MP3 The MP3 decoder for Bluebottle was originally written by C. Dornbierer [24] and adapted to AosCodecs framework by U. Müller [77].

WAV An encoder and decoder for the WAV audio file format.

PCM A simple decoder for pure PCM streams.

Image Codecs

PNG a decoder for the Portable Network Graphics format [16].

JP2 a Jpeg2000 image decoder written by Z. Franjcic [31].

GIF a GIF image encoder and decoder.

BMP a BMP image encoder and decoder.

Text Codecs

BBT an encoder and decoder for the XML based Bluebottle text format that is described in section 3.3.8.

Oberon an encoder and decoder for the Oberon text format ignoring embedded *Gadgets*.

UTF-8 an encoder and decoder for UTF-8 encoded plain texts.

ISO-8859-1 an encoder and decoder for the ISO-8859-1 encoded plain texts.

UCS-16 an encoder and decoder for the UCS-16 encoded plain texts.

ASCII an encoder and decoder for ASCII encoded plain texts.

HEX a somewhat special decoder that converts an arbitrary stream into a text that displays the bytes of the stream in the hexadecimal system. The encoder encodes a text of hexadecimal characters into a byte stream, ignoring whitespace characters.

9

Case Studies

The proof of the pudding is in the eating.

— Saying

The Bluebottle user interface and multimedia framework is in daily use at various institutions around the world. A wide variety of applications have been developed using the framework in and outside ETH. This chapter introduces some of these applications, discussing them briefly to demonstrate the flexibility and generality of the framework.

9.1 GoingPublik

I have nothing to say
and I am saying it
and that is poetry
as I needed it

— John Cage (1912-1992)

GoingPublik is a *sound art* project for a distributed ensemble of trombones conceived by Art Clay [41] and realised at the ETH. *Sound art*, in contrast to traditional music does not rely on psychological relationship between sounds, but on their independence from one another. Pursuant to John Cage[1] an American composer, sounds are let to come into being for themselves. The *sound art* composition is not bound and limited by harmonic constraints but is more based on functions of time and rhythmic contrasts.
The core idea of the *GoingPublik* project is the mobility of the performers that are guided by an electronic scoring system, called *MatrixWindow*, running on a wearable computer system. The scoring system presents to the performer a compositional structure which permits impro-

[1] John Cage (1912 - 1992) had a great influence of the music of the 20th century. His most famous work 4'33" ("four minutes, thirty-three seconds") consists of only silence, several recordings are available.

visational elements. The electronic score is created in real time based on electronic monitoring of the performer's physical behaviour during the performance. The monitoring system uses an absolute reference system consisting of geographical position and orientation for all performers. The geographic position is determined by a GPS system and the orientation is determined with a 3d compass which relies on the earth's magnetic field. Since all performers share the same reference system, they are virtually linked together. Physical proximity and orientation of the performers result in a shared compositional palette.
The *Going Publik* project shows the portability and adaptability of the Bluebottle framework in a real application. *Going Publik* was performed in Monthey, Switzerland and Canada. It will also be performed at the ETH 150 year anniversary celebration in Zürich.

9.1.1 Hardware

For the performance, a number of musicians are equipped with wearable computers, GPS receivers, three dimensional digital compasses, *finger mouse* control input devices and head-mounted displays. The wearable computer is the central point of the setup. It analyses the input from the GPS receiver and compass sensor to calculate and render the realtime score that is presented in a head-mounted display for interpretation by the musician.
The wearable computer is a belt-integrated computer system named QBIC [2], that is being collaboratively developed at the *Wearable Computing Lab* at ETH Zürich and *Art-Of-Technology (AoT)*. It contains a 400Mhz XScale processor integrated into the belt's buckle, 256MB SDRAM, USB-Host controller, RS-232, VGA and Bluetooth. The interface connectors and the battery are integrated into the belt.
Figure 9.1 shows the hardware configuration. The GPS receiver is connected via RS-232, the 3d compass uses Bluetooth on the L2CAP protocol level. The head-mounted display is connected via a VGA analog video stream, (unfortunately) requiring an external analog to digital converter unit. The control input device that is used to control the performance parameters can either be a finger mouse connected via USB or a Smartphone connected via Bluetooth on the RFCOMM protocol layer.

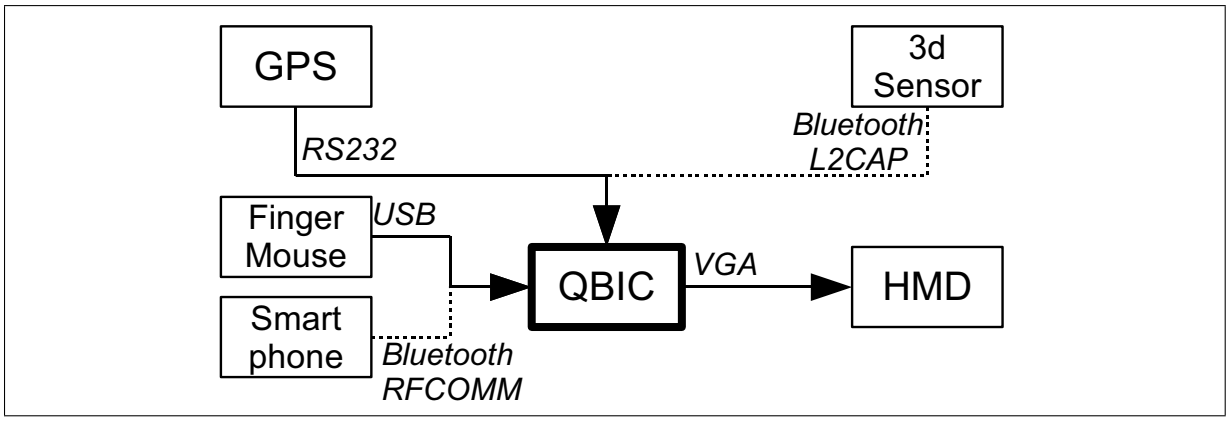

Figure 9.1: Hardware Setup of the GoingPublik Project

9.1.2 Software

The project uses the XScale-implementation [7] of Bluebottle as its runtime and graphics system. Since the user interface framework has exactly the same API on the XScale and on IA32, it was possible to develop and test the electronic scoring system on an available IA32 system before the wearable hardware became available.
The *GoingPublik* software collects and analyses the input from all sensor devices. The results are then used to control the *MatrixWindow*. Figure 9.2 shows an overview of the *GoingPublik* software.

GPS Data The GPS data is sent to a GPS analyser that returns normalised coordinates, an averaged speed-level and information about waiting times. The normalised coordinates are sent to the matrix transformer that calculates the geometric structure of the *GoingPublik* matrix that defines the rhythmic structure and the pitch range of the performance. The matrix structure only depends on the position of the performer. The normalised position coordinates are also visualised to the performer. The speed-level and waiting time information are sent to a state machine that controls a number of icons that suggest or demand actions of the performer and influence the speed of the conduction time line. The speed level is also sent to an image transformer that resizes the image depending on the speed level.

Compass Data The compass data is sent to a compass analyser that notes changes of heading over time. This information is sent as another parameter to the state engine that controls the icons. The heading directly influences the choice of the image out of a library that is interpreted by the performer. Pitch and roll data of the compass are sent to the image transformer as further parameters that stretch the selected image accordingly.
Figure 9.3 shows a snapshot of the *MatrixWindow* as it is seen by the musician during the performance, in the head-mounted display.

9.1.3 Interaction without a Desktop via Pie Menus

To control the *GoingPublik* software at the beginning of and during the performance, a special interaction technique was implemented that works outside a traditional office environment. The control-input device is either a finger-mouse or a ”Smartphone” device that is connected via Bluetooth. During a performance it is impossible to freehandedly control a pointer with a mouse on a desktop. Therefore a new kind of *pie-menu* [17] has been implemented that allows the desired actions to be selected with rough gestures via finger-mouse or joystick on the Smartphone.
To activate the menu, the performer presses a button on the finger-mouse or Smartphone. Moving the mouse-pointer or the joystick on the Smartphone into one of four directions selects one of the four options that can be presented by the menu. Menu options can then open new sub-menus for more options. A sequence of selections through a number of sub-menus results in a gesture-like movement. Snapshot 9.5 shows the *pie menu* controlled via Smartphone. When the menu is called with the finger-mouse, a mouse-pointer is visible in its center.

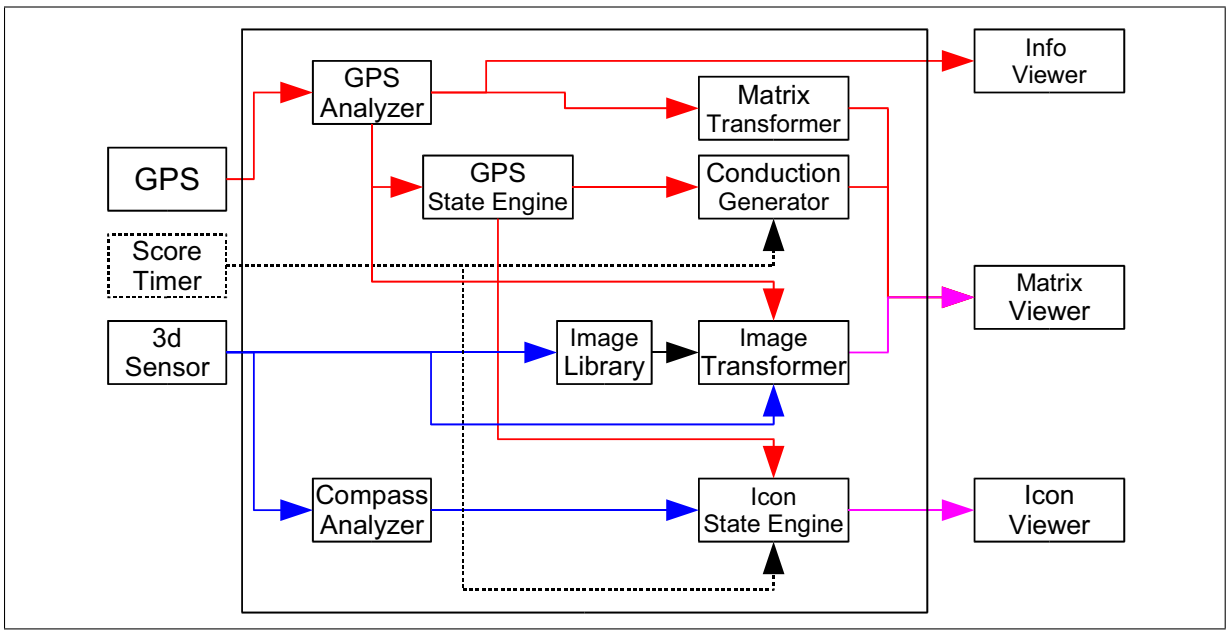

Figure 9.2: GoingPublik Software Schematic

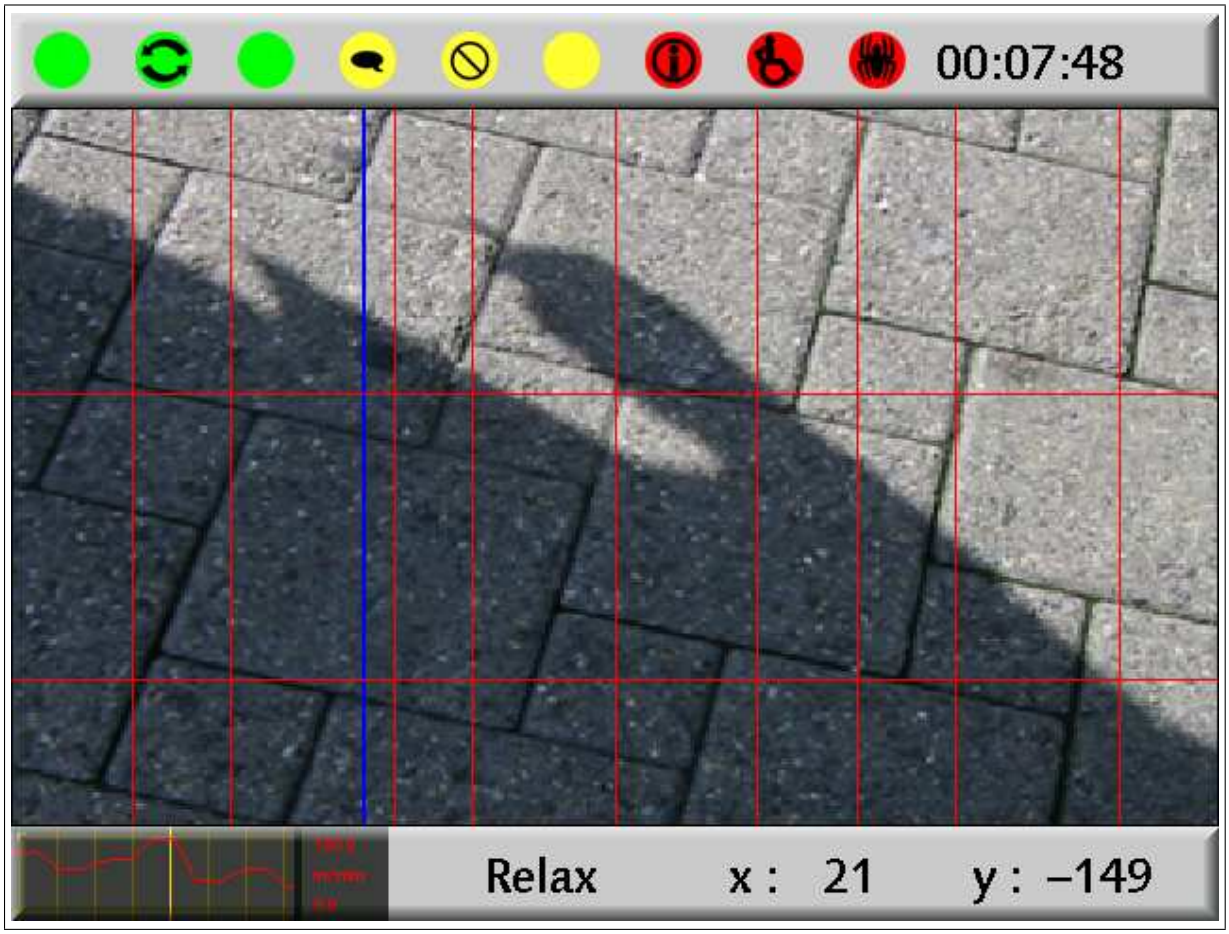

Figure 9.3: A Snapshot of the MatrixWindow as seen by the Performer in the HMD

The *pie-menu* opens at the position of the mouse cursor, or in the centre of the screen in the case of Smartphone joystick input. Figure 9.4 (A) shows a *pie-menu* being opened. Selecting an option from a pie-menu with four entries requires the user to move the finger mouse or joystick approximately into the direction of the entry with a tolerance of $\pm 45°$. Figure 9.4 (B) shows a number of possible movements that can chose the *target* entry. Figure 9.4 (C) shows a *sub-pie-menu* being opened when the target in (B) is chosen.

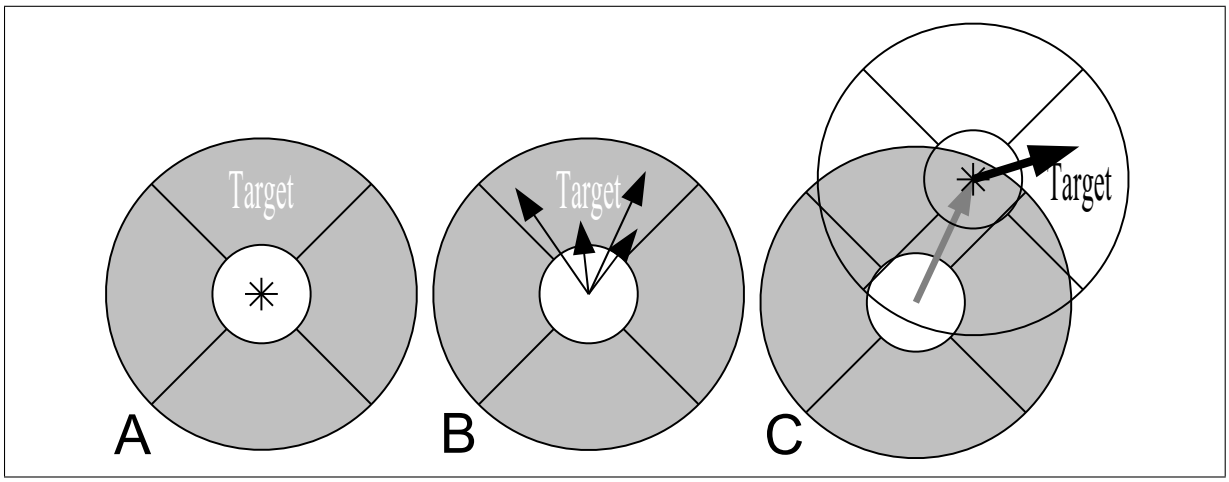

Figure 9.4: Selecting an Options with a Pie Menu

An advantage of pie menus is the same Fitt-distance for all menu items [17]. The Fitt-distance classifies the difficulty of exactly pressing (or in this case clicking at) an object with a fiven size and shape in a given distance [27] [65]. Selecting items in pie menus can be remembered as gestures. The readability of pie menus is normally sub-optimal if they contain a large number of items. They also use more screenspace than linear menus.
The pie menus were integrated into the GUI component framework as translucent *FormWindow*s (see 6.3). When a pie menu is opened by an application program, the pointer ownership is transferred to the menu object as decribed in section 5.5.2.

9.2 Instant Gain in Grace

The project ”Instant Gain in Grace”, a diploma thesis [57] at the Hyperwerk - Interaction Design Univ. of Applied Arts & Sciences - FHBB, visually enhances a Butoh dance performance with multimedia. Butoh is a contemporary avant-garde dance which originated in Japan. It combines dance, theatre and improvisation under the influence of traditional Japanese performing arts.
For the visual enhancement of the performance, the project uses a number of body-worn acceleration sensors whose data is collected by a wearable computer that sends the combined sensor data via a wireless *Bluetooth* connection to an analyzing computer. The result of the analysis is then used to control an animation software. The setup is shown in figure 9.6. 64 different *emotional* categories of motion patterns are recognized and the result of the classification is sent to the animation engine. The animation is on two levels influenced by the dancer’s movements,

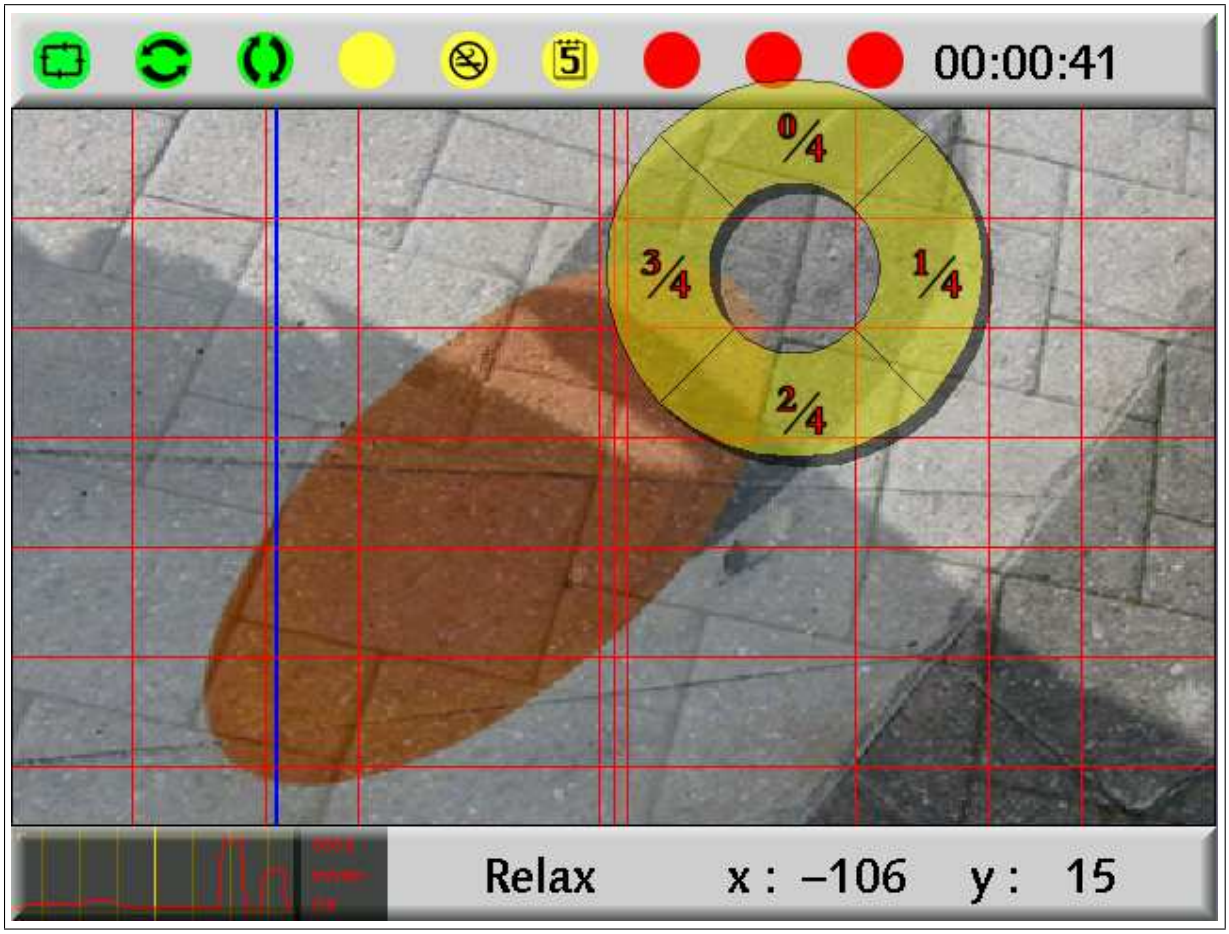

Figure 9.5: A Snapshot of an open Pie Menu

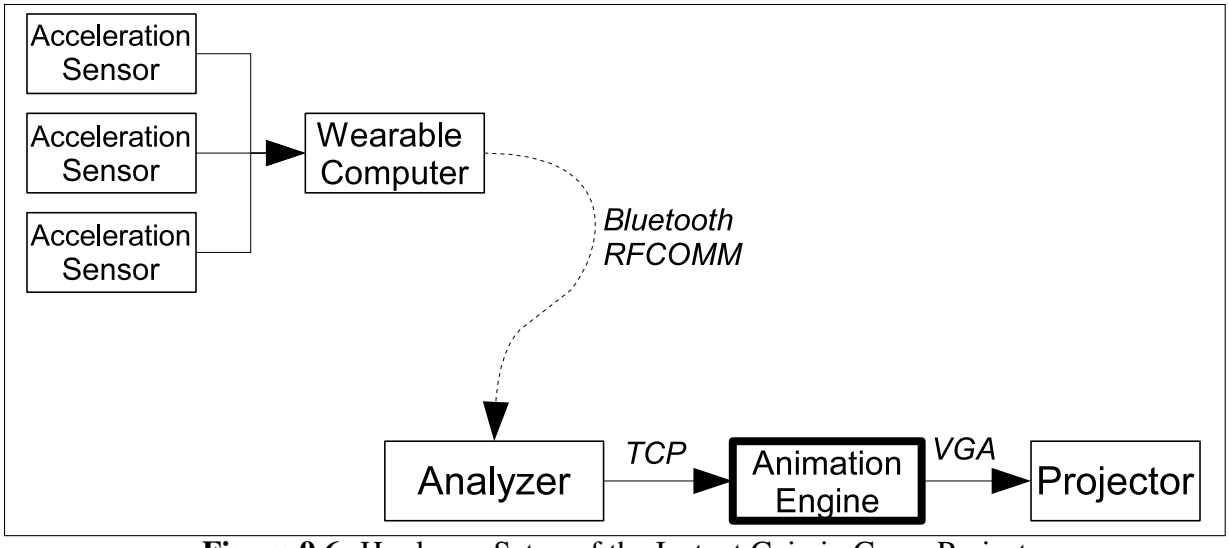

Figure 9.6: Hardware Setup of the Instant Gain in Grace Project

the scenery selection and its parametrisation. The scenery selection depends on the detected *emotional* category, while the parametrisation of the scene is based more directly on features like the root mean square of the acceleration components. For example, the scene parametrisation determines the size or colour of actor objects in a scene. The motion classification is described in detail in [6].
The visualisation system was developed as a semester project at ETH [38] and makes heavy use of several system components that have been discussed in previous chapters:

- The XML support is used for the declaration of the scenery and the scripting of animations
- The graphics framework is used to create full screen animations in realtime
- The CCG fonts are used as a source of vector graphics that appear as actor objects in the scenes. The outlines of Chinese glyphs are filled with solid or translucent colours or with images or even movies.

Live performances of the project were shown in autumn 2003 in Basel, Switzerland and in the "Disappearing Computer Jamboree" at the institute of media design in Ivrea, Italy.
The *Instant Gain in Grace* project makes use of the Bluebottle graphics framework for real-time full-screen animation, optionally including life-video feeds.

9.3 Was geschah am 6. Tag?

"Was geschah am 6. Tag" [62] (What happened on the 6th day) is an interactive story-telling video project. Different movie segments are projected onto four screens in the performance room. Based on the analysis of the audience's behaviour the selection of the movie segments changes continiously, where the transitions between movie segments are computed in realtime on four different computers that are controlled and synchronised via network broadcasts. Figure 9.7 gives an overview of the system setup. Each of the four multimedia computers is connected to a video projector, two loudspeakers and a network switch. A dedicated computer monitors and analyses the behaviour of the audience, especially its interest in individual screens. The analysis is based on the data provided by RFID sensors hidden in the performance room and tracking the position of RFID tags that are handed out to the audience before the performance.

The implementation of the project makes use of several system components that have been discussed in previous chapters:

- It uses the abstract decoder framework to get fine-grained access to the video frames and audio samples.
- It uses the graphics system to transform and display the video frames on the projectors.
- It uses the sound system to control the audio output on eight independent loudspeakers.

Up to the time of writing there was no public performance of "Was geschah am 6. Tag?". It will be demonstrated at the ETH 150 year anniversary celebration.

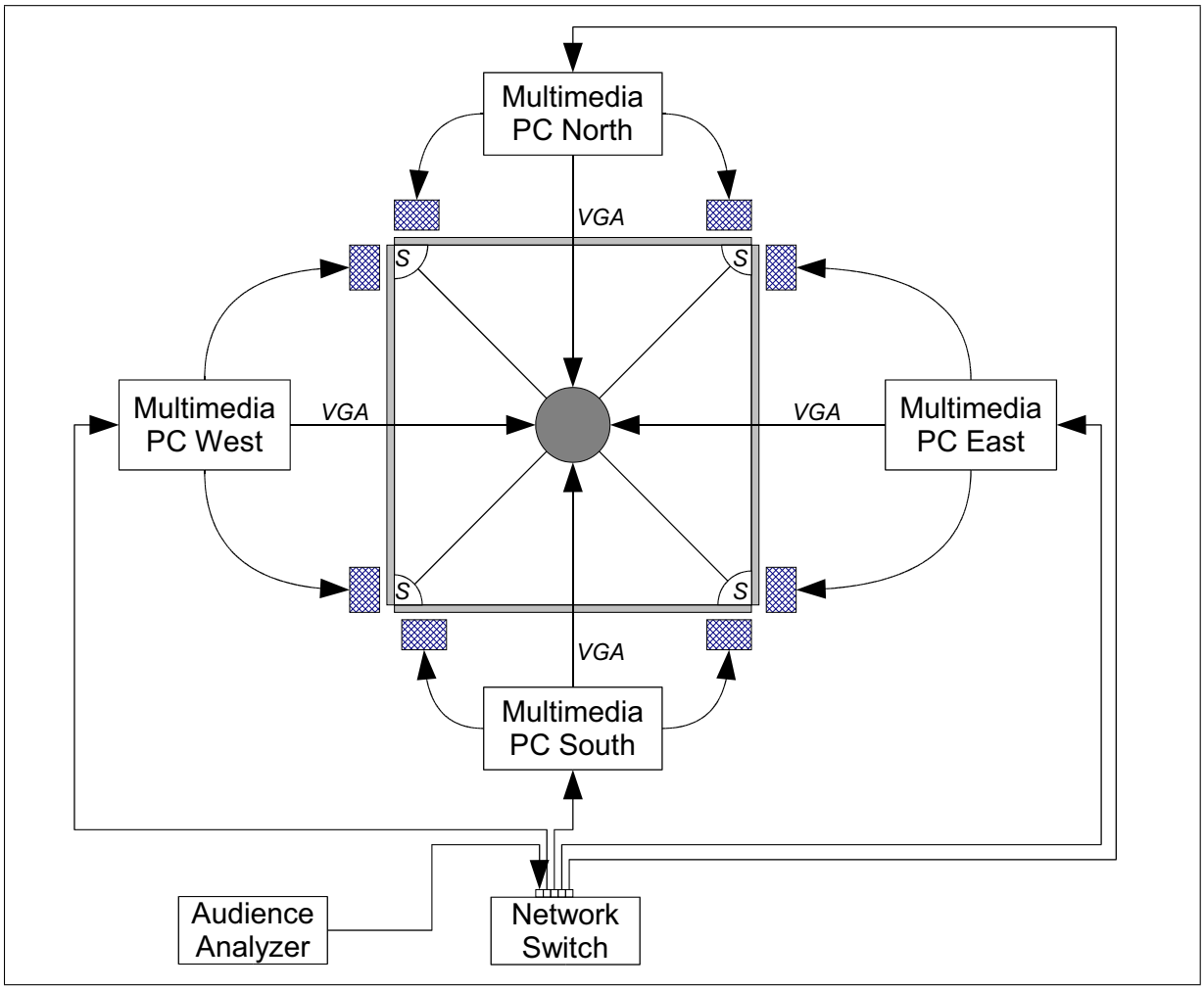

Figure 9.7: Setup of ”Was geschah am 6. Tag?”

9.4 Student Projects

A large number of student projects have made use of the user interface and multimedia framework. Some of the developed applications have become part of the Bluebottle system release. The following lists a small selection:

Flash Player A player for Macromedia Flash has been written by L. Häner [42]. It supports a subset of Macromedia Flash 4, excluding bitmaps, sound and interaction elements.

Filemanager A Filemanager application has been developed by B. Fluri [29].

Teletext A teletext viewer application using the *AosText* system and a standard text editor to display the highly attributed text has been implemented by O. Jeger [54]. The teletext content can also be accessed through a web interface running as a plugin in the Bluebottle dynamic HTTP server.

DivX Player T. Trachsel developed a DivX movie decoder and player for the Bluebottle system [109].

Composer Language An experimental composer language for the Bluebottle component system has been written by M. Sala [100].

Desktop Publishing System An interactive desktop publishing system was developed by P. Lehmann [63].

Web browser A web browser for Bluebottle is being developed as a masters thesis by S. Keel.

Realtime Video Effects A system for event based realtime video effects is being developed by R. Ghioldi.

Partition Visualiser A graphical tool for disk partition management has been developed by S. Stauber [105].

Skin Editor An editor for the development of system wide graphical skins has been written by F. Nart [82].

9.5 Typical Desktop

Figure 9.8 shows a cutout of a typical Bluebottle desktop.

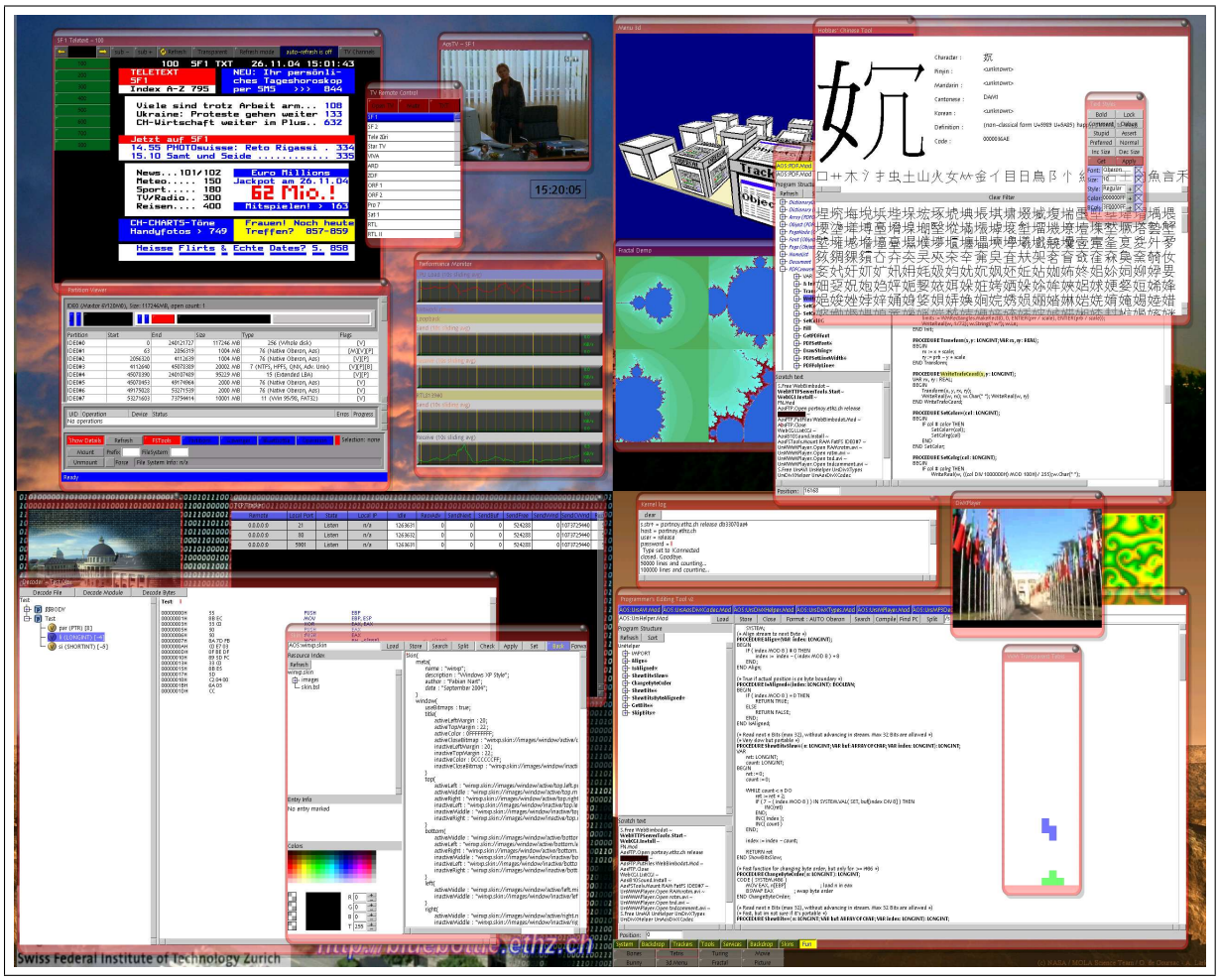

Figure 9.8: A Cutout of a Typical Bluebottle Desktop

10 Conclusions

Perfection is achieved,
not when there is nothing more to add,
but when there is nothing left to take away.

— Antoine de Saint-Exupery (1900 - 1944)

This chapter summarises what has been achieved and points out problems and possible future works.

10.1 Summary

Our pragmatic goal was to explore and develop a practical and versatile user interface model that is suitable for a wide range of applications and simple to program. One of the main concerns was the flexibility of the system and the efficient support for multiple application activities that access the GUI system simultaneously.
The efforts resulted in a new user interface concept that combines in an innovative way elements from traditional PARC user interfaces with elements from zoomable and textual user interfaces. The resulting user interface extends the traditional PUI desktop with a pervasive zooming and panning functionality and a certain level of persistence. The use of task oriented zooming contexts avoids a number of problems of traditional zooming user interfaces such as the *lack of context* or *tunnel vision* problems.
A flexible system-wide text system allows the activation of parameterised commands and macros wherever they can be written and allows the use of text tools in a unified fashion. It significantly differs from and extends the standard Oberon text system mainly in respect of its thread-safety, text positions, support for styles and internationalised character sets.
The new user interface concept leads to a number of strategies that result in a more homogeneous and efficient user interface while reducing the overall system complexity.
The consideration of thread-safety in the GUI framework leads to a classification of scenarios of programming deadlocks and the design of a scheme of locks and sequencing objects which is simple, practical and effective to avoid these deadlocks.
The work also includes the design and implementation of a flexible and extensible graphics and

multimedia framework for the Bluebottle system that is able to run real applications not only on workstation and desktop systems but on wearable computers as well. The graphics framework takes advantage of one or more general purpose CPUs with possibly specialized instruction set extensions for vector calculations but avoids the complexity of supporting special purpose hardware for graphics acceleration. This leads to a simple graphics system architecture that matches and even outperforms commercial hardware accelerated systems in common situations through structural advantages. It also allows the system to be easily ported to different hardware platforms, especially to small devices like wearable computers which has been demonstrated with a port of the system to the QBIC wearable computer that has been developed at ETH Zürich.
An extensive number of application programs that have been developed inside and outside ETH running on Bluebottle testify to its efficiency and flexibility.

10.2 Issues of the Base System and Suggested Improvements

During the development of the Bluebottle system a number of problems in various areas of the system have been detected and resolved. This section lists a number of remaining issues and suggests possible improvements.

10.2.1 Exception Handling

There is no fine grained support for exception handling in the Active Oberon language or the Bluebottle kernel. The only available mechanism to catch exceptions without losing the fallible activity is in the active body that can be declared as *SAFE* making it restart on a failure. This coarse exception handling makes it hard to cleanly release recursive locks on shared data structures in the case that an exception occurs.
To make the display space manager more resistant to faulty application programs it uses a flag in its *SAFE* active body to detect if it has been restarted. A restart means that an exception occurred, most probably in an up-call to an application program. To prevent the GUI from freezing, it breaks and resets all the display space manager locks. This sledgehammer method is suboptimal.
A simple fine grained exception handling strategy has been suggested where the system ensures that a *finalisation part* of a program block is executed even if an exception happened in the corresponding code block. A current diploma thesis is implementing and evaluating this strategy.

10.2.2 Thread Termination

In the current kernel a thread cannot be terminated externally by means other than removing its containing module. In certain situations with up-calls from a system-wide shared module into application programs, it is desirable to be able to limit the duration of an up-call to prevent an endless loop from blocking the entire system.
A mechanism that is able to raise an exception in a separate thread is being investigated in

a diploma thesis in connection with the implementation of a fine grained exception handling scheme.

10.2.3 Namespaces

During the development of Bluebottle, a number of naming conflicts between existing Oberon modules and new modules turned up. Since module names need to be unique, new Bluebottle modules had to be named differently from all existing Oberon modules. The generally good naming scheme of Oberon modules resulted in a lot of name clashes with new modules. These were resolved by adding a prefix to the names of Bluebottle modules. The system-wide text system of Oberon for example is called *Texts*. The prefixed Bluebottle version is calles *Aos-Texts*. The naming situation could be improved by the introduction of namespaces for different projects. Oberon modules could be contained in the *Oberon* namespace while the same concise module names could be used in the *Bluebottle* namespace.

10.2.4 Garbage Collection

The stop and go garbage collector of the Bluebottle system is a problem for multimedia applications and every other realtime application if the system is running background threads that use dynamic memory. Even if a multimedia thread itself does not dynamically allocate memory after its initialisation, it can still suffer from the system-wide disruption. A generational collector or a collector on partitioned memory that does not block all processes for too long could improve the multimedia abilities of the system.
A collector that does not change pointers during the heap traversal could allow system-managed and self-managed memory regions to coexist in the same address space, even with the possibility of a limited exchange of pointers between threads. A step in this direction has recently been made by L. Bläser who replaced the *Deutsch-Schorr-Waite* pointer rotating mark phase of the garbage collector with a fixed-size stack-based marking algorithm with overflow handling. Because pointers no longer need to be modified during the mark phase it is now possible to run a restricted set of programs during garbage collection.
Such programs

- may never hide pointers to objects in use i.e. they may only operate on objects and references to objects that are anchored in a list that is immutable during the garbage collection.

- may never directly or indirectly access the type tag of the object.

The second restriction is needed because the current Bluebottle heap management stores the mark flag in the type tag during the mark phase of the garbage collection. Only during the sweep phase, are the type tags restored. This restriction is very limiting because the compiler implicitly uses the type tags without knowledge of the programmer. The type tag can, for example, be used when working with arrays.
It is possible to remove the second restriction by adding an additional heap management data

word to each heap block. This change would require rewriting a significant part of the basic heap management code, which is an error-prone task and would also require changes in the compiler. On the bright side it could easily result in a significantly simpler heap structure.

10.2.5 Reflection

Although a lot of meta information about modules and objects is available in the system, there is no unified reflection API to access it. To get the desired information it is often necessary to parse cryptic and badly documented data structures that are directly loaded from the object file to the main memory.
To make the system extensible and more flexible, a reflection API could be added to *AosModules*. All programs that currently use meta information about the loaded modules should then be changed to use the new API. These changes would reduce the overall system complexity and pave the way for a new and simplified object file format with a richer set of meta information.

10.2.6 Sound System

HD Audio Standard Currently the Bluebottle system (Chapter 7) offers three sound drivers. In the near future, the AC'97 [51] sound standard will be replaced by the new HD Audio standard [52] that delivers significant improvements. HD Audio defines up to eight channels at 192 kHz with 32-bit quality, while the AC'97 specification only supports up to six channels at 48 kHz with 20-bit quality. Writing drivers for the new standard and, where necessary, extending the *AosSound* interface to better support the additional features of HD Audio should be considered in the future.

Mixing & Resampling Layer In the current *AosSound* system, the mixing of audio channels as well as the re-sampling of audio data is the responsibility of the sound driver. This works well for hardware that supports these features. If the hardware does not support mixing and re-sampling, the driver code becomes more complicated because the missing hardware support has to be emulated in software. Inserting an abstraction layer between the *AosSound* system and the sound driver that handles re-sampling and mixing could be considered to achieve an overall simplification.

Synchronisation Support The current sound drivers have no way of telling an application program what sample is currently being played. If available, this information could improve the synchronisation of audio and video output. The current sound drivers can only inform the application programs what sample is being mixed but not what sample is being played. For sound hardware that cannot provide this information, the driver should estimate the delay from mixing a sample until it finally becomes audible in the speaker.

10.2.7 Codec Framework

The number of implemented codecs in all codec classes is relatively small and could be increased with additional student projects to make the system more compatible with the outside world.
The codec framework (Chapter 8) is specified to be as simple as possible. With the implementation and integration of additional codecs it might become desirable to extend the basic codec interfaces to offer new features.
The Bluebottle compression libraries are not yet integrated into the *AosIO* and *AosCodecs* frameworks. The integration could unify the currently heterogeneous interfaces of the libraries and hence simplify the use of compression in application programs.

A

Programming Examples

Appendix A gives a number of programming examples about different topics covered in preceding chapters.

A.1 Display Space Manager Programming - Scribble Application

This section gives a programming example of how to use the display space manager and graphics framework. The example program opens a window and catches the events of the mouse pointer for drawing lines and keyboard events to store the artwork. Snapshot A.1 shows the *scribble* window.

Figure A.1: Snapshot of the Scribble Window

In the following, the entire program is discussed in detail, traversing the program text from top to bottom, explaining the functions and giving references to relevant chapters. The program execution starts with a call to its *Open* method that creates a new window object.
The new program is called *WMScribble*

```
    MODULE WMScribble;  (** AUTHOR "TF"; PURPOSE "Example program"; *)

A : IMPORT WMWindowManager, Utilities, WMGraphics, WMRectangles,
     WMDialogs, AosModules;

B : CONST
      LeftButton = 0;
      RightButton = 2;

C : TYPE
      ScribbleWindow = OBJECT (WMWindowManager.BufferWindow)
      VAR lx, ly : LONGINT;

D :     PROCEDURE &New();
        BEGIN
          Init(400, 400, FALSE);
E :       WMWindowManager.ExtAddWindow(SELF, 200, 200, {WMWindowManager.FlagFrame});
F :       SetTitle(Utilities.NewString("Scribble Example"));
G :       canvas.Fill(WMRectangles.MakeRect(0, 0, GetWidth(), GetHeight()),
                    WMGraphics.White, WMGraphics.ModeCopy);
H :       Invalidate(WMRectangles.MakeRect(0, 0, GetWidth(), GetHeight()));
I :       SetPointerInfo(manager.pointerCrosshair)
        END New;

J:      PROCEDURE PointerMove(x, y : LONGINT; keys : SET);
        BEGIN
K:        IF LeftButton IN keys THEN
            canvas.Line(lx, ly, x, y, WMGraphics.Blue, WMGraphics.ModeSrcOverDst);
L :         Invalidate(
              WMRectangles.MakeRect(
                Utilities.Min(lx, x), Utilities.Min(ly, y),
                Utilities.Max(lx, x) + 1, Utilities.Max(ly, y) + 1)
            )
           END;
M :        lx := x; ly := y
        END PointerMove;

N :     PROCEDURE PointerDown(x, y : LONGINT; keys : SET);
        BEGIN
O :       lx := x; ly := y;
P :       IF RightButton IN keys THEN
            canvas.Fill(WMRectangles.MakeRect(0, 0, GetWidth(), GetHeight()),
            WMGraphics.White, WMGraphics.ModeCopy);
            Invalidate(WMRectangles.MakeRect(0, 0, GetWidth(), GetHeight()))
          END
        END PointerDown;

Q :     PROCEDURE KeyEvent(ucs : LONGINT; flags : SET; keySym : LONGINT);
        VAR res: LONGINT; filename : ARRAY 128 OF CHAR;
        BEGIN
R :       IF ucs = ORD("s") THEN
S :         filename := "scribble.bmp";
T :         IF WMDialogs.QueryString("Save as :", filename) = WMDialogs.ResOk THEN
              WMGraphics.StoreImage(img, filename, res);
U :           IF res # 0 THEN
                res := WMDialogs.Message("Sorry",
                 "The image could not be stored. Try another file name.", {WMDialogs.ResOk})
              END
            END
          END
        END KeyEvent;

      END ScribbleWindow;
```

```
V : VAR sw : ScribbleWindow;

W : PROCEDURE Open*(par : ANY): ANY;
    BEGIN {EXCLUSIVE}
      IF sw # NIL THEN sw.Close END; NEW(sw);
      RETURN NIL
    END Open;

X : PROCEDURE Cleanup;
    BEGIN
      IF sw # NIL THEN sw.Close END
    END Cleanup;

Y : BEGIN
      AosModules.InstallTermHandler(Cleanup)
    END WMScribble.
```

A The program imports *WMWindowManager* (see chapter 5) to open a window. *WMGraphics* (see chapter 4) contains the graphics system. *WMRectangles* is used to create and operate on rectangles. *WMDialogs* is imported to open a query dialogue where the user can enter a filename to save the artwork. Finally *AosModules* is imported to install a handler that can close the *Scribble Window* if the *WMScribble* module is unloaded.

B With a typical mouse, the buttons are enumerated from left to right. The left button is number zero, the middle number one and the right is number two. If a mouse has no middle button it still returns number two for the right button. We define named constants for better readability in the program code.

C A new type *ScribbleWindow* is defined that extends a *WMWindowManager.BufferWindow* as introduced in 5.3.2. The new object has two fields *lx, ly* to store the position where the last line segment ended respectively where the left mouse button was pressed down the last time.

D The procedure *&New* is the constructor of the window instances. It first calls the inherited constructor procedure defining the intended window size to be 400 by 400 pixels. The boolean parameter *FALSE* defines the window to be non-transparent. While the inherited constructor procedure should be called first, most of the following operations in the constructor are order-independent[1].

E The *WMWindowManager.ExtAddWindow* procedure adds a window into the default display space of the system, at a position relative to the upper left corner of the default viewport that observes the display space (see section 5.2). The procedure takes the new window (*SELF*) as the first parameter, followed by the *x, y* position relative to the default viewport's position and a SET parameter specifying details of the behaviour and look of the window. In this case the window is just a regular window with a frame, so it can be moved around and closed.

[1]The only exception for avoiding artefacts is that the drawing to the buffer should happen before the window is invalidated

As soon as this procedure is called, the window is known to the display space manager, and it will appear on the screen.

F The *SetTitle* method is a basic window functionality available to all *Window*s. It gives the window a name that can be used by the display space manager when it needs to label the window. It takes a pointer to an *ARRAY OF CHAR* as a parameter. The dynamic array is created with the *Utilities.NewString* procedure. The new window is called *Scribble Example*.

G The *canvas* is an object that is available to all *BufferWindow*s. It allows access to the window's display buffer. The *canvas.Fill* procedure fills a rectangular area of the display buffer with a colour, in this case *white*. The filled rectangle has the size of the window, so the entire window will turn white. The parameter *WMGraphics.ModeCopy* tells the canvas not to apply any blending operations when painting the new colour but to simply copy it to the background. Un-blended drawing operations are significantly faster than blended painting. Section 4.3.1 gives details.

H The *Invalidate* procedure is a basic window functionality. It declares a rectangular region of a window as *dirty* or *invalid*. The display space manager will require all viewports that display the respective regions to re-establish the display consistency (see section 5.2.1). The detailed process is described in chapter 5. Here, the entire window is declared invalid. This tells the display space manager to make the result of the antecedent *Fill* operation visible.

I Now the *scribble* application is almost ready. As a last thing, the constructor sets a specially shaped mouse pointer for the window that is better suited for drawing than an ordinary arrow-like pointer.

J The *PointerMove* procedure is a basic *Window* procedure that is called via the window's message sequencer object whenever the mouse pointer belongs to the window and is being moved. The parameters *x, y* return the pointer position relative to the window's coordinate system (fig. 5.2). The parameter *keys* is the set of mouse buttons that are held down.

K If the left mouse button is pressed, a line from the last mouse position that is stored in *(lx, ly)* to the new position *(x, y)* will be drawn:

L Now the changed region must be invalidated, so that the display space manager will make the new line visible. The update area is calculated as the rectangle around the line segment.

M Store the current mouse pointer position in *(lx, ly)*.

N The *PointerDown* procedure is a basic *Window* procedure that is called via the window's message sequencer object whenever the mouse pointer belongs to the window and a mouse button is being pressed. The parameters *x, y* return the pointer position relative to

the window coordinate system (fig. 5.2). The parameter *keys* is the set of mouse buttons that are being held down.

O Store the current mouse pointer position in *(lx, ly)*.

P If the right mouse button is pressed, the entire window will be cleared with a white colour and its region will be invalidated so that the display space manager updates the display.

Q The *KeyEvent* procedure is a basic *Window* procedure that is called via the window's message sequencer object whenever a key event happens and the window has the keyboard focus. The *ucs* parameter contains the Unicode value of the key that is being pressed down, if available. *flags* contains the state of modifier keys such as *shift*, *control* or *meta*. *keySym* contains the X11 keyboard code [102] of the key that was pressed.

R To check if the pressed key was the *"s"*-key:

S If yes, now set a default file name into the *filename* variable and open a string input dialogue box with the title *"Save as :"* that allows the user to change the filename.

T If the string input dialogue box was closed by pressing the *Ok* button or the *enter* key, it returns *WMDialogs.ResOk*. Only in this case, the image will be stored with the user defined filename. The following line of code stores *img*, the buffer image associated with the *BufferWindow*, to the a file with the name in the local variable *filename*. *WMGraphics.StoreImage* automatically searches in the system codec library for an image encoder that matches the given filename extension. The result of the operation is stored in *res*.

U In the unlikely case of a problem, for example if no codec matching the file name extension was found, the user will be informed about the problem. The *WMDialogs.Message* opens an information box containing the title *"Sorry"* and the hint *"The image could not be stored. Try another file name."*. The last parameter contains a set of buttons that should be added to the information box. Here we add an *Ok* button only.

V The module variable *sw* stores the reference to the *ScribbleWindow*. It is used to close the window if the *WMScribble* module is unloaded.

W The exported procedure *Open* creates a new *ScribbleWindow*.

X The *Cleanup* procedure checks if the module variable *sw* is assigned to a *ScribbleWindow* instance. If yes, it closes the respective window.

Y The module body that is automatically started when the module is loaded, installs the termination handler procedure *Cleanup* that will close the potentially open *WMScribbleWindow* if the module is unloaded.

A.2 TextWriter Example

This section gives a programming example of using the *AosTextUtilities.TextWriter*. The example program opens a form window (see section 6.3) and adds a text editor component (see section 3.4). On the text model associated with the text editor it opens a *TextWriter* object and writes a number of example strings. Screenshot A.2 shows the output of the program.
In the following, the entire program is discussed in detail, traversing the program text from top to bottom, explaining its function and giving references to the relevant chapters. The program execution starts with a call to its *Open* method that creates a new window object.

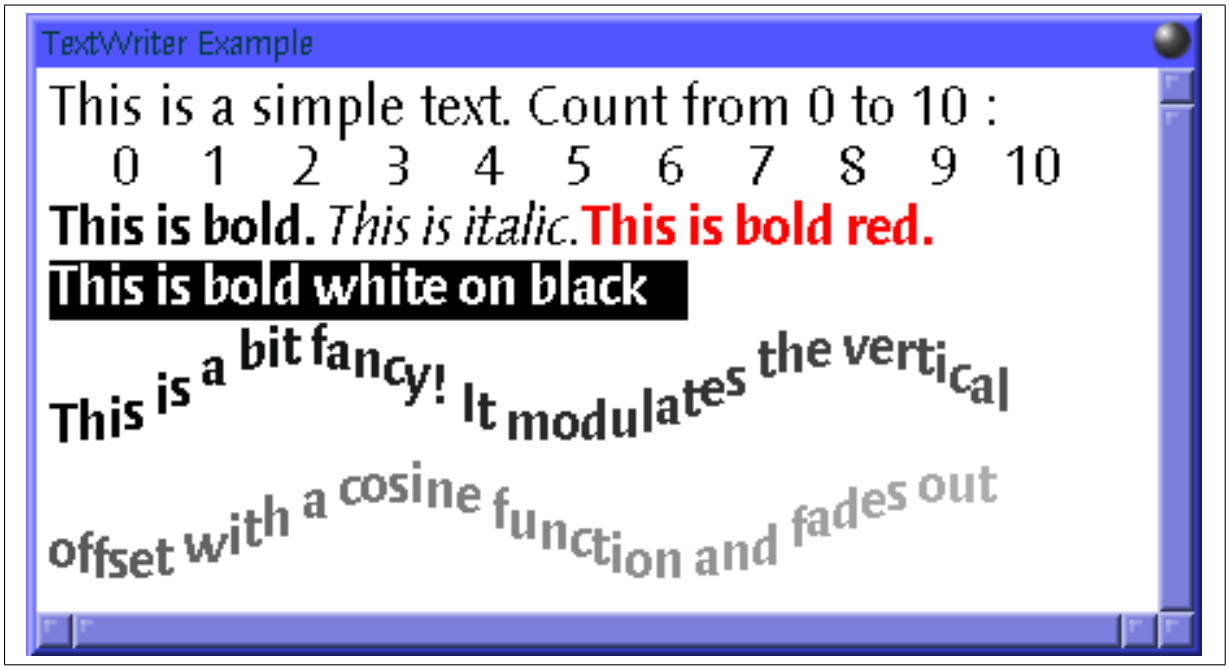

Figure A.2: Snapshot of the TextWriter Example Program

```
    MODULE TextWriterExample; (** AUTHOR "TF"; PURPOSE "TextWriter Example"; *)

A : IMPORT
      Utilities, WMGraphics, WMComponents, WMWindowManager,
      WMEditors, AosTextUtilities, AosIO, Math;

B : TYPE
      Window* = OBJECT (WMComponents.FormWindow)
      VAR editor : WMEditors.Editor;

C :     PROCEDURE &New();
        BEGIN
D :       NEW(editor);
          editor.bounds.SetExtents(400, 300);
          editor.fillColor.Set(WMGraphics.White);
E :       Init(editor.bounds.GetWidth(), editor.bounds.GetHeight(), FALSE);
F :       SetContent(editor);
G :       WMWindowManager.DefaultAddWindow(SELF);
          SetTitle(Utilities.NewString("TextWriter Example"));
H :       WriteToEditor;
        END New;
```

```
I :     PROCEDURE WriteToEditor;
        VAR
          tw : AosTextUtilities.TextWriter;
          w : AosIO.Writer;
          i : LONGINT;
          buffer : ARRAY 256 OF CHAR;
        BEGIN
J :       NEW(tw, editor.text);
K :       w := tw.GetWriter();
L :       tw.SetFontSize(20);
          w.String("This is a simple text. Count from 0 to 10 : "); w.Ln;
          FOR i := 0 TO 10 DO w.Int(i, 5) END; w.Ln;

M :       tw.SetFontStyle({WMGraphics.FontBold});
          w.String("This is bold. ");
N :       tw.SetFontStyle({WMGraphics.FontItalic});
          w.String("This is italic.");
O :       tw.SetFontStyle({WMGraphics.FontBold});
          tw.SetFontColor(WMGraphics.Red);
          w.String("This is bold red."); w.Ln;
P :       tw.SetBgColor(WMGraphics.Black);
          tw.SetFontColor(WMGraphics.White);
          w.String("This is bold white on black");
          w.Ln;
Q :       tw.SetBgColor(WMGraphics.White);
R :       buffer := "This is a bit fancy! It modulates the vertical offset with
          a cosine function and fades out.";
          i := 0;
          WHILE buffer[i] # 0X DO
            tw.SetFontColor(WMGraphics.RGBAToColor(i * 2, i * 2, i * 2, 0FFH));
            tw.SetVerticalOffset(ENTIER(15 * Math.cos(i/4)));
            w.Char(buffer[i]);
            INC(i)
          END
        END WriteToEditor;

      END Window;

S : PROCEDURE Open*(par : ANY) : ANY;
    VAR inst : Window;
    BEGIN
      NEW(inst);
      RETURN NIL
    END Open;

    END TextWriterExample.
```

A The program imports *WMGraphics* for colour and font-style constants, *WMComponents* for the form window and component support. *WMWindowManager* is used to add the form window to the display space. *WMEditors* contains the default editor component that is used to display the text. *AosTextUtilities* offers the *TextWriter* object that is described in section 3.3.6. *AosIO* contains the support for streams through which the text will be written. Finally the module *Math* is used for evaluating trigonometric functions.

B A new component container window that will contain the text editor is created.

C The constructor *New* initialises the window and adds it to the display space.

D The following creates a new editor component and sets its size to 400 by 300. It also sets the background colour of the editor to non-transparent white. The default colour for the text editor component is transparent.

E The following calls the inherited constructor *Init* with the size of the editor.

F Then the editor component is installed in the form window.

G The following lines add the window to the display space and set the window title to "Text-Writer Example".

H The constructor of the window then calls the *WriteToEditor* procedure that writes the rich text to be displayed.

I The following defines the *WriteToEditor* procedure with its local variables. The variable *tw* is the *TextWriter*.

J First the procedure opens a *TextWriter* on the text that is associated with the editor of the window.

K An *AosIO.Writer* is then opened on the *TextWriter* object so that the standard methods of writing to a stream can be used to write the text.

L The font size of the *TextWriter* is set to 20 and a string is written to the *AosIO.Writer* stream, followed by the integer numbers from 0 to 10.

M The font style is changed to *bold* and the string "This is bold." is written.

N The font style is changed to *italic* and the string "This is italic." is written.

O As an other example for creating rich text, the font is set to *bold* and its colour to *red*. The string "This is bold red." is written with these attributes.

P The text background is changed to *black* and the text colour is set to *white*. The string "This is bold white on black" is written.

Q The text background is reset to white.

R The following code prints a string character by character varying the font colour and vertical offset. Multi-byte UTF-8 characters are ignored in this example. They would be written with the attributes that are set at the time the last byte of the multi-byte character is added.

S The *Open* procedure opens the window and starts the program.

A.3 Sound and Codec Programming - Simple MP3 Player

The following programming example uses *AosCodecs* and *AosSound* to implement a simple MP3 player. For the sake of clarity, the program plays a hard-coded file and does not offer interaction or error handling.

```
    MODULE TestPlayer;

A : IMPORT
      AosSound, AosCodecs, AosIO;

B :  TYPE
      Player= OBJECT
      VAR
C :     decoder : AosCodecs.AudioDecoder;
D :     soundDevice : AosSound.Driver;
E :     playChannel : AosSound.Channel;
F :     bufferPool : AosSound.BufferPool;
G :     buffer : AosSound.Buffer;

H :     PROCEDURE &Init(fileName : ARRAY OF CHAR);
I :     VAR i : LONGINT;
          res : LONGINT;
          in : AosIO.Reader;
          channels, rate, bits : LONGINT;
        BEGIN
J :       decoder := AosCodecs.GetAudioDecoder("MP3");
K :       soundDevice := AosSound.GetDefaultDevice();
L :       in := AosCodecs.OpenInputStream(fileName);
M :       decoder.Open(in, res);
N :       decoder.GetAudioInfo(channels, rate, bits);
O :       soundDevice.OpenPlayChannel(playChannel, rate, bits, channels,
                                              AosSound.FormatPCM, res);
P :       NEW(bufferPool, 10);
Q :       FOR i := 0 TO 9 DO
R :       NEW(buffer); NEW(buffer.data, 4096);
            bufferPool.Add(buffer)
          END;
S :       playChannel.RegisterBufferListener(bufferPool.Add);
        END Init;

T :   BEGIN  {ACTIVE}
U :     playChannel.Start;
V :     WHILE decoder.HasMoreData() DO
W :         buffer := bufferPool.Remove();
X :         decoder.FillBuffer(buffer);
Y :         playChannel.QueueBuffer(buffer)
        END
      END Player;

Z : PROCEDURE Play*(par : ANY) : ANY;
    VAR player : Player;
    BEGIN
      NEW(player, "test.mp3");
      RETURN NIL
    END Play;

    END TestPlayer.
```

A The *TestPlayer* program imports *AosSound* for sound output, *AosCodecs* for the MP3 decoder and *AosIO* for stream support.

B The player is implemented as an active object. So more than one instance could run at the same time.

C The variable *decoder* is a generic *AosCodec.AudioDecoder* as described in section 8.4.1. It will later be assigned an MP3 decoder in the program but it is not specialised for the MP3 case.

D The *soundDevice* variable is a generic *AosSound.Driver* object as described in detail in section 7.2. It will later be assigned the system's default sound device driver.

E The *playChannel* variable is an *AosSound.Channel* as described in section 7.2.3.

F The *bufferPool* variable will be used to manage the *AosSound.Buffer*s as described in section 7.2.2.

G The *buffer* variable contains an *AosSound.Buffer* as described in section 7.2.1. It will be used first to fill the *bufferPool* and then to transport the sound data from the *decoder* to the *playChannel*.

H The constructor *Init* of the *Player* object takes as a parameter the *filename* of the MP3 file to play.

I The local variable *i* will be used as a counter, *res* takes the result value of different operations. The variable *in* is the *AosIO.Reader* that is opened on the MP3 file. The variables *channels*, *rate* and *bits* contain information that is needed to to create a suitable player channel.

J Now an instance of an *AudioDecoder* with the name *MP3* is requested. The *AosCodecs.GetAudioDecoder* searches in the system configuration for a matching codec factory and returns a respective decoder object instance if the factory was found, *NIL* otherwise.

K At this position the program should check if the *decoder* variable is *NIL* and exit with a adequate error message. To keep our sample program restricted to the absolute minimum, this check is ignored here.
Optimistically the program goes on and acquires a reference to the default sound driver. The *AosSound.GetDefaultDevice* procedure blocks if no driver is installed and waits for a driver to be loaded. Therefore, the return value always is a valid sound driver instance.

L After acquiring the *soundDevice* and the *decoder* the program needs to acquire the audio data. It does this using the *AosCodecs.OpenInputStream* procedure that returns an *AosIO.Reader* instance if the stream could be opened and *NIL* otherwise. The program optimistically assumes that the needed file exists and that the *in* return value is valid.

M With the valid *InputStream* the *decoder* can now be opened. The *Open* procedure returns *AosIO.ResOk* in *res* if the input stream header contains valid data.

N When the *decoder* is open, the *GetAudioInfo* can be used to obtain the essential audio data that is required to open a player channel.

O Now a player channel with the information obtained from *GetAudioInfo* must be opened. *OpenPlayChannel* returns a non-*NIL playChannel* if the operation was completed successfully. If the *res* value is different from *AosSound.ResOk* then the playback quality might be affected by artefacts of the internal sound format transformations of the sound driver that could not natively fulfil the audio requirements. The *TestPlayer* program ignores the *res* value and assumes the *playChannel* to be non-*NIL*.

P The next step initialises a buffer pool with 10 entries that can be used to synchronise the producer (the *decoder*) with the consumer (the *playChannel*).

Q Now the *bufferPool* needs to be filled with a maximum of 10 sound buffers objects.

R The following creates a new sound buffer *buffer* and in the buffer creates space for 4096 bytes of data.

S To make it possible for the playing channel to return played buffers, register the buffer pool's *Add* procedure as the buffer return handler.

T After the constructor terminates, the system starts the *ACTIVE* body of the active player object.

U Now the *playChannel* will be started. It was by default on pause since it has been created.

V As long as there is more data in the encoded audio stream...

W ... a *buffer* is taken from the buffer pool. This will block the player activity if the pool is empty. This can only happen if all buffers are queued in the *playChannel*.

X Now the *decoder* that fills the *buffer* with audio data will be called...

Y ... and the filled buffer will be enqueued in the *playChannel* for playing.

Z The *Play* procedure finally creates an instance of the active player object.

B

List of Relevant Modules

Appendix B lists, sorted by topic, the important modules that implement the Bluebottle GUI and multimedia system. The list is intended to serve as an overview for programmers who need to extend or use the system. Some of the modules fit into two or more categories. These are classified by concept rather than implementation. The text editor for example fits into the component system class by implementation but conceptually it fits into the *Text and Strings* topic since it manipulates texts. It can therefore be found in the *Text and Strings* section.

B.1 Display Space Manager

WMWindowManager.Mod defines the basic *WindowManager* and *ViewPort* interfaces.

WMDefaultWindows.Mod implements the default windows that are used to implement the window borders (title, left, right and bottom).

WindowManager.Mod implements the *WindowManager* interface and a *ViewPort* based on a display adaptor defined in *AosDisplays*.

WMVNCView.Mod implements a *ViewPort* for a VNC remote framebuffer.

WMScreenShot.Mod implements a *ViewPort* that stores the observed area of the display space to a file.

WMBackdrop.Mod implements backdrop images.

WMDropTarget.Mod implements drag and drop support on the display space manager level. It is also used indirectly in the component system.

WMRestorable.Mod implements the possibility to store and restore display space objects.

WMPopups.Mod implements pop-up elements such as menus or element pickers.

WMDialogs.Mod implements a set of system dialogue elements based on GUI components

B.2 Graphic System

Raster.Mod implements basic bitmap raster operations

WMRasterScale.Mod implements scaling on bitmaps

WMGraphics.Mod defines the abstract *Canvas* and *Font* objects and a font manager plug-in interface. Implements the *BufferCanvas* as a specialisation of *Canvas*.

WMGraphicUtilities.Mod offers a number of higher level drawing routines based on *WMGraphics*. For example, shaded rectangles, glass effects etc.

PDF.Mod implements *PDFCanvas* as specialisation of *WMGraphics.Canvas*.

B.3 Fonts

WMFontManager.Mod implements a simple font manager that tries to find the best matching font given a triplet of *font name*, *font size* and *font style*. When loaded, it registers itself in *WMGraphics* as the font manager.

WMOberonFonts.Mod implements the Oberon font file format.

WMCCGFonts.Mod implements support for CCG fonts.

WMBitmapFonts.Mod implements support for a proprietary Unicode bitmap font format.

WMDefaultFont.Mod contains a hardcoded version of the Oberon font. It is used as the last fall-back. Since the font is embedded in a module it can be linked into the system's boot image so that it is possible to write debug information even when there is no file system available to load a font file.

AFM.Mod implements minimal support for the Adobe Font Metrics. The module is needed to calculate the font widths when writing to PDF files.

B.4 Texts and Strings

Utilities.Mod defines the basic UTF-8 string type as a *POINTER TO ARRAY OF CHAR* and offers string manipulation routines.

UTF8Strings.Mod offers conversion and manipulation routines for UTF-8 encoded strings.

AosTexts.Mod Implements the text model, maintains a text clipboard and keeps track of the last selection in the system.

AosTextUtilities.Mod offers the *TextWriter* object and implements a number of text encoders and decoders that are compatible with the *AosCodecs* framework.

WMInputMethods.Mod defines the interface for input method editors.

WMPinyinIME.Mod implements a pinyin input method editor for Chinese characters.

WMCyrillicIME.Mod implements a Cyrillic input method editor.

WMMacros.Mod implements a text macro plug-in for the editor components

WMTextView.Mod implements a text viewer component that visualises a text model.

WMEditors.Mod extends a *VisualComponent* and aggregates a *TextView* and a text model. Key events are intercepted and interpreted as editing command on the text model.

B.5 Component System

WMComponents.Mod implements the basic *Component* and *VisualComponent* object types.

WMStandardComponents.Mod defines a number of frequently used GUI components such as *Panel*, *Button*, *Checkbox*, *Scrollbar*, *Resizer* and more. It also implements several non-visual components such as a *Timer*.

WMGrids.Mod defines a general visual grid component with support for variable-sized rows and columns, fixed rows and columns and also merged cells.

WMStringGrids.Mod implements a thread-safe model for a grid of strings and extends the generic grid component with a drawing routine that displays the model.

WMTrees.Mod defines a thread-safe model of a generic tree structure and a component that can visualize this model.

WMDiagramComponents.Mod implements a model and view for value-by-time diagrams.

WMTabComponents.Mod implements a bar with named tags from which one can be selected by pressing.

WMSystemComponents.Mod implements components that visualise the file system. For example a list of files or a tree of folders.

MixerComponents.Mod visualises and manipulates the sound mixer values

B.6 Sound System

AosSound.Mod defines the generic sound system and keeps a registry of sound devices.

EnsoniqSound.Mod driver for the Ensoniq sound chip. This chip is also emulated in VMWare.

AosYMF754.Mod driver for the Yamaha 754 sound chip.

Aosi810Sound.Mod driver for the common Intel 810 sound chip.

OGGVorbisPlayer.Mod a sound player that plays OGG Vorbis files. The decoder has not yet been adapted to the *AosCodecs* framework.

PlayRecWave.Mod a wave file player and recorder. The codec has not yet been adapted to the *AosCodecs* framework.

B.7 Codecs

AosCodecs.Mod defines interfaces and generic generator procedures for encoder and decoder objects for different types of encoded data. It includes codecs for still images, video, audio and text.

AosPNGDecoder.Mod implements an *AosCodecs.ImageDecoder* for PNG images.

AosBMPCodec.Mod implements an *AosCodecs.ImageEncoder* and *AosCodecs.ImageDecoder* for BMP images.

AosGIFCodec.Mod implements an *AosCodecs.ImageEncoder* and *AosCodecs.ImageDecoder* for the GIF format.

AosMP3Decoder.Mod implements an *AosCodecs.AudioDecoder* for the MP3 format.

AosDivXDecoder.Mod implements an *AosCodecs.VideoDecoder* for the DivX format.

B.8 Helper Modules

WMRectangles.Mod defines a rectangle and offers procedures to work with rectangles for example, moving, resizing, intersecting, combining rectangles.

WMLocks.Mod implements optimised recursive reader writer locks that are used in the display space manager, text- and component system.

WMMessages.Mod defines messages and the *message sequencer* object that is used in the display space manager and component system.

WMEvents.Mod offers event sources and event listeners.

WMProperties.Mod defines properties and property lists used in the component system.

List of Abbrevations

AGP Advanced Graphics Port

API Application Programming Interface

CHI Computer Human Interaction

CJK Chinese, Japanese , Korean

CLI Command Line Interface

CLR Common Language Runtime

Codec Coder Decoder

COM Component Object Model

FEP Front-End Processor, see IM and IME

GDI Graphical Device Interface

GPS Global Positioning System

GUI Graphical User Interface

HCI Human Computer Interaction

HMD Head Mounted Display

IM Input Method, see IME and FEP

IME Input Method Editor, see IM or FEP

LOD Level Of Detail

MVC Model View Controller

PCM Pulse Code Modulation

PDA Personal Digital Assistant

PDF Portable Document Format

PNG Portable Network Graphics

PUI PARC User Interface, a WIMP user interface following

RFID Radio Frequency Identification

RSI Repetitive Strain Injury

SDMS Spacial Data Management Systems

TCP Transmission Control Protocol

TUI Textual User Interface

UCS Universal Character Set (specified in ISO/IEC 10646)

UDP User Datagram Protocol

UTF UCS Transformation Format

VNC Virtual Network Computing

WIMP Windows Icons Menus Pointers, an acronym often used in the HCI community to reference traditional GUIs

WYSYWIG What You See Is What You Get

ZUI Zoomable User Interface, a special kind of a GUI

List of Figures

List of Tables

Bibliography

[1] Adobe. Font Metrics File Format Specification Version 4.1, 1998
http://partners.adobe.com/asn/developer/pdfs/tn/5004.AFM_Spec.pdf

[2] O. Amft, M. Lauffer, S. Ossevoort, F. Macaluso, P. Lukowicz, G. Tröster. Design of the QBIC wearable computing platform. Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures and Processors, 2004.

[3] U. Anliker, P. Lukowicz, G. Tröster, S. J. Schwartz, R. W. DeVaul The WearARM: Modular, High Performance, Low Power Computer Platform Designed for Integration into Everyday Clothing. ISWC 2001: Proceedings of the 5th International Symposium on Wearable Computers, 8.-9. October 2001, pages 167-168.

[4] C. Artho, K. Havelund, A. Biere. High-Level Data Races, VVEIS'03: The First International Workshop on Verification and Validation of Enterprise Information Systems, April 2003.

[5] C. Artho, K. Havelund and A. Biere. Using block-local atomicity to detect stale-value concurrency errors. Proceedings of the ATVA '04, Taipei, Taiwan, 2004.

[6] M. Barry, J. Gutknecht, I. Kulka, P. Lukowicz, T. Stricker. Multimedial Enhancement of a Butoh Dance Performance - Mapping Motion to Emotion with a Wearable Computer System. Second International Conference on Advances in Mobile Multimedia (MoMM2004), Bali, Indonesia, 2004.

[7] M. Baumgartner. Intel 80200 / XScale Aos. Semester project, Department of Computer Science, ETH Zürich, 2003.

[8] B. B. Bederson, J. D. Hollan Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. ACM UIST '94, 1994

[9] B. B. Bederson and J. D. Hollan. Pad++: A Zoomable Graphical Interface. CHI'94, short paper.

[10] B. B. Bederson, L. Stead and J. D. Hollan. Pad++: Advances in Multiscale Interfaces, Proceedings of 1994 ACM SIGCHI Conference.

[11] B. B. Bederson, J. Meyer, L. Good Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java In ACM UIST 2000, pp. 171-180.

[12] B. B. Bederson, J. Grosjean, J. Meyer. Toolkit Design for Interactive Structured Graphics.

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 8, AUGUST 2004.

[13] C. Betrisey, J. F. Blinn, B. Dresevic, B. Hill, G. Hitchcock, B. Keely, D. P. Mitchell, J. C. Platt, T. Whitted. Displaced Filtering for Patterned Displays, Proc. Society for Information Display Symposium, pp. 296-299, (2000)

[14] R. A. Bolt. Spatial Data-Management. Cambridge, Massachusetts, MIT Press, 1979.

[15] J. Borchers. A Pattern Approach to Interaction Design. John Wiley & Sons, LTD, 2001.

[16] T. Boutell, et. al. PNG (Portable Network Graphics) Specification Version 1.0, RFC 2083, 1997.

[17] J. Callahan, D. Hopkins, M. Weiser, B. Shneiderman. An Empirical Comparison of Pie versus Linear Menus. In Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems, 1988

[18] CCG-Font Specification. Personal communication with Cheah Shen Yap and online documentation. 2003
http://www.eforth.com.tw/efeditor/

[19] D. Chang, L. Dooley, J. E. Tuovinen. Gestalt Theory in Visual Screen Design - A New Look at an Old Subject. In Proc. WCCE2001 Australian Topics: Selected Papers from the Seventh World Conference on Computers in Education, Copenhagen, Denmark. Conferences in Research and Practice in Information Technology, 8. McDougall, A., Murnane, J. and Chambers, D., Eds., ACS. 5-12, 2002.

[20] Z. Chen, K. Lee. A New Statistical Approach To Chinese Pinyin Input. Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics, ACL 2000.

[21] D. A. Cox, J. S. Chugh, C. Gutwin and S. Greenberg. The Usability of Transparent Overview Layers. CHI'98 Summary Proceedings of the Conference on Human Factors in Computing Systems, p301-302, Late-breaking short paper. ACM Press, 1998.

[22] Digital Research. CP/M Operating System Manual. Digital Research, California, 1976.

[23] W. C. Donelson. Spatial Management of Information. Proceedings of 1978 ACM SIGGRAPH Conference, 203-209.

[24] C. Dornbierer. MP3 Player für Aos. Semester project, Department of Computer Science, ETH Zürich, 2002.

[25] B. Egger. Development of an Aos Operating System for the DNARD Network Coputer. Diploma thesis, Department of Computer Science, ETH Zürich, 2001.

[26] Ariane 501 Inquiry Board. Ariane 5 - Flight 501 Failure. Paris, 1996.
http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf

[27] P. M. Fitts, The information capacity of the human motor system in controlling the amplitude of movement. Journal of Experimental Psychology, 47, 381-391, 1954.

[28] D. Flanagan. Java in a Nutshell - A Desktop Quick Reference. O'Reilly, 1997.

[29] B. Fluri. Filemanager für Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2003.

[30] B. Francis, A. Fedorov, R. Harrison, A. Homer, S. Murphy, D. Sussman, R. Smith and S. Wood. Professional Active Server Pages 2.0, Brian Francis. Wrox Press Ltd, 1998.

[31] Z. Franjcic. JPEG2000 für Aos. Semester project, Department of Computer Science, ETH Zürich, 2004.

[32] T. Frey. Architectural Aspects of a Thread-Safe Graphical Component System Based on Aos, Lecture Notes in Computer Science 2789, Springer, 2003.

[33] T. Frey. Bluebottle Tutorial. Tutorial.Text in Bluebottle releases. ETH Zürich. http://www.bluebottle.ethz.ch

[34] F. Friedrich. ETH Win Aos Oberon. http://www.bluebottle.ethz.ch/WinAos, 2003.

[35] G. W. Furnas. Generalized fisheye views. In CHI 86, pages 16-23, Boston MA, USA, ACM Press, April 1986.

[36] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass., 1994.

[37] D. Gentner, J. Nielson. The Anti-Mac Interface. Communications of the ACM, Vol.39, No.8, August 1996.

[38] M. Gernss. Animations, Ein Animationssystem für Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2003.

[39] C. Geschke, J. Warnock.
PDF Specification, fourth edition, Adobe Systems Incorporated, May 2001.

[40] J. Gough. Compiling for the .NET Common Language Runtime (CLR). .NET Series, Prentice Hall, 2002

[41] J. Gutknecht, A. Clay, T. Frey, . GoingPublik: Testing System Design Outside of the Ivory Tower. Creativity & Cognition, London, 2005.

[42] L. Häner. Flash Player für Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2003.

[43] B. L. Harrison, H. Ishii, K. J. Vicente, W. A. S. Buxton. Transparent Layered User Interfaces: An Evaluation of a Display Design to Enhance Focused and Divided Attention. In Proceedings of CHI '95. (May 07-1), Denver. pp. 317-324.

[44] B. L. Harrison, G. Kurtenbach, K. J. Vicente. An Experimental Evaluation of Transparent User Interface Tools and Information Content. UIST 95, Pittsburgh PA USA, 1995.

[45] C. Heinzer. ENSONIQ 137x Audio Trieber für Aos. Semester project, Department of Computer Science, ETH Zürich, 2002.

[46] Intel Corp. Write Combining Memory Implementation Guidelines, 1998. Order Number 244422, http://developer.intel.com/.

[47] Intel Corporation. IA-32 Intel Architecture Optimization Reference Manual, 2004 Order Number 248966-011, http://developer.intel.com/.

[48] Intel Corporation. IA-32 Intel Architecture Software Developers Manual Volume 3: Basic Architecture, Order Number 253665, http://developer.intel.com/.

[49] Intel Corporation. IA-32 Intel Architecture Software Developers Manual Volume 3: System Programming Guide. Order Number 253668, http://developer.intel.com/.

[50] Intel Corporation. IA-32 Intel Architecture Software Developers Manual Volume 3: Instruction Set Reference A-M, Order Number 253666. & Intel Corporation. IA-32 Intel Architecture Software Developers Manual Volume 3: Instruction Set Reference N-Z, Order Number 253667.

[51] Intel Corporation. Audio Codec '97. Revision 2.3, 2002.
http://www.intel.com/labs/media/audio/

[52] Intel Corporation. I/O Controller Hub 6 (ICH6) High Definition Audio / AC 97 - Programmer's Reference Manual (PRM), Document Number: 302349-002, July 2004.
http://www.intel.com/design/chipsets/hdaudio.htm

[53] ISO/IEC JTC 1 ISO/IEC Joint Technical Commitee for Information Technology, Information technology - 8-bit single-byte coded graphic character sets - Part 1: Latin alphabet No. 1, ISO International Organization for Standardization, 1988.

[54] O. Jeger. Teletext für Aos. Semester project, Department of Computer Science, ETH Zürich, 2003.

[55] J. H. Jenkins. New Ideographs in Unicode 3.0 and Beyond. 15th International Unicode Conference, San Jose, CA, 1999.

[56] K. Jonsson. Intel AC'97 Sound driver. Semester project, Department of Computer Science, ETH Zürich, 2003.

[57] I. Kulka. Instant Gain in Grace. Diploma project, Hyperwerk Basel, 2003.
http://web.archive.org/web/20031206072930/http://www.hyperwerk.ch/curriculum
/exams/diplom03/index.htm

[58] G. Kurtenbach, W. Buxton. User Learning and Performance with Marking Menus. In Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems. 258-264, 1994.

[59] G. Kurtenbach, G. Fitzmaurice, T. Baudel, W. Buxton. The Design of a GUI Paradigm based on Tablets, Two-hands, and Transparency. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 35 - 42, 1997.

[60] P. K. Lai, D. Y. Yeung, M. C. Pong. A Heuristic Search Approach to Chinese Glyph Generation Using Hierarchical Character Composition. COMPUTER PROCESSING OF ORIENTAL LANGUAGES VOL.10, NO.3. 1996.

[61] B. W. Lampson, E. Taft.
Alto Users Handbook, Xerox Palo Alto Research Center, 1979.

[62] T. Lang. Was geschah am 6. Tag? - ein interaktives, multimodales Environment. Unpublished project documentation.

[63] P. Lehmann. Desktop Publishing System für Bluebottle. Diploma thesis, Department of Computer Science, ETH Zürich, 2004.

[64] K. Lunde. CJKV Information Processing, O'Reilly, 1999.

[65] I. S. MacKenzie, W. Buxton. Extending Fitts' law to two-dimensional tasks. Proceedings of the CHI '92 Conference on Human Factors in Computing Systems. ACM, 219-226, 1992.

[66] I. S. MacKenzie. Movement Time Prediction in Human-Computer Interface. Readings in human-computer interaction (2nd ed.) (pp. 483-493), Los Altos, CA: Kaufmann, 1995.

[67] J. L. Marais. Design and Implementation of a Component Architecture for Oberon. PhD thesis, Institut für Computersysteme, ETH Zürich, 1996.

[68] E. McCreight et.al.
Alto Hardware Manual, Xerox Palo Alto Research Center, 1978.

[69] G. Meunier.
Lost in the Blue, Message in the Oberon Mailinglist January 3, 2003
https://www.mail.inf.ethz.ch/archive/oberon/2003/000623.html.

[70] J. Meyer, K. Perlin B. Bederson, J. Hollan. Two Document Visualization Techniques for Zoomable Interfaces. Unpublished, 1995.

[71] Microsoft Corporation. COM: Component Object Model Technologies. 2004.
http://www.microsoft.com/com/default.mspx

[72] Microsoft Corporation. AVI File Format. Microsoft DirectX 9.0 SDK, Summer 2004
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directshow/htm/avifileformat.asp.

[73] Microsoft Corporation. Longhorn Developer FAQ. 2004.
http://msdn.microsoft.com/longhorn/support/lhdevfaq/default.aspx

[74] Microsoft Corporation. MSDN Library.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/dataexchange/clipboard/clipboardformats.asp, 2004.

[75] Microsoft Corporation. Platform SDK: Windows API - Graphics Device Interface.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/graphics_device_interface.asp, 2004.

[76] R. B. Miller. Response time in man-computer conversational transactions. Proc. AFIPS Fall Joint Computer Conference Vol. 33, (pp. 267-277), 1968.

[77] U. Müller. DivX Player Erweiterung für Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2004.

[78] H. Muller, K. Walrath. Threads and Swing. The Swing Connection, Reference, Technical Articles and Tips, September 2000
http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html

[79] P. J. Muller. Native Oberon Operating System.
http://www.oberon.ethz.ch/native/

[80] P. J. Muller. The Active Object System - Design and Multiprocessor Implementation. PhD thesis, Institut für Computersysteme, ETH Zürich, 2002.

[81] B. A. Myers. The importance of percent-done progress indicators for computer-human interfaces. Proc. ACM CHI'85 Conf. (San Francisco, CA, 14-18 April), 11-17, 1985.

[82] F. Nart. Style Layer für Bluebottle. Diploma thesis, Department of Computer Science, ETH Zürich, 2004.

[83] D. A. Norman. The Design of Everyday Things., Basic Books, New York, 1988.

[84] E. Oswald. A Generic 2D Graphics API with Objects Framework and Applications. PhD thesis, Institut für Computersysteme, ETH Zürich, 2000.

[85] R. Parkinson. The Dvorak Simplified Keyboard: Forty Years of Frustration, Computers and Automation magazine, November, 1972, pp. 18-25.

[86] K. Perlin, D. Fox. Pad: An Alternative Approach to the Computer Interface, Proceedings of 1993 ACM SIGGRAPH Conference, pp. 57-64.

[87] K. Perlin and J. Meyer. Nested user interface components. Proc. of 12th annual ACM symposium on Userinterface software and technology(pp 11-18), Asheville, North Carolina, USA, 1999.

[88] J. C. Platt. Optimal Filtering for Patterned Displays. IEEE Signal Processing Letters, 7, 7, pp. 179-180, 2000.

[89] S. Pook, G. Vaysseix and E. Barillot. Zomit: biological data visualization and browsing. Bioinformatics, 14(9):807-814, Nov. 1998.

[90] S. Pook, E. Lecolinet, G. Vaysseix, E. Barillot. Context and Interaction in Zoomable User Interfaces. AVI 2000 Conference Proceedings (ACM Press), 2000.

[91] C. v. Praun, T. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs Proc. Conf. Programming Language Design and Implementation (PLDI'03), 2003.

[92] J. Raskin. The Humane Interface: New Directions for Designing Interactive Systems. Reading Massachusetts: Addison-Wesley, 2000.

[93] J. Raskin. The Humane Environment (THE) External Technical Specification http://humane.sourceforge.net/the/spec.html, 2004.

[94] P. Reali. Using Oberon's Active Objects for Language Interoperability and Compilation. PhD thesis, Institut für Computersysteme, ETH Zürich, 2003.

[95] T. Richardson, Q. Stafford-Fraser, K. R. Wood and A. Hopper. Virtual Network Computing, IEEE Internet Computing, Vol.2 No.1, Jan/Feb 1998 pp33-38.

[96] T. Richardson, K. R. Wood.
The RFB Protocol
www.uk.research.att.com/vnc/rfbproto.pdf Olivetti Research Ltd, Cambridge, 1998.

[97] T. Richardson, K. R. Wood.
The RFB Protocol (Revised 11 July 2002)
http://www.realvnc.com/docs/rfbproto.pdf Formerly of Olivetti Research Ltd / AT&T Labs Cambridge, 2002.

[98] G. Robertson, M. Czerwinski, K. Larson, D. C. Robbins, D. Thiel, and M. van Dantzich. Data Mountain:Using Spatial Memory for Document Management. UIST 98. San Francisco, CA.

[99] F. Röthenbacher. IP-Phone for Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2003.

[100] M. Sala. Composition Language für Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2004.

[101] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs ACM Transactions on Computer Systems, vol. 15, pages 391–411, 1997.

[102] R. Scheifler. X Window System Protocol. X Consortium Standard, X Version 11, Release 6, March 1994.

[103] D. C. Schmidt. Strategized Locking, Thread-safe Interface, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components, C++ Report, vol. 11, Sept. 1999.

[104] Sibelius. The world of Sibelius - Complete range of music software. Sibelius Website, 2004.
http://www.sibelius.com/download/brochure/World_of_Sibelius(UK).pdf

[105] S. Stauber. Partition Viewer for Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2004.

[106] M. C. Stone, K. Fishkin, E. A. Bier. The movable filter as a user interface tool. In CHI94 Human factors in computing systems, pp. 306312, Boston MA, USA. ACM Press.

[107] Sun Corporation. Desktop Java - Java Foundation Classes (JFC/Swing). 2004.
http://java.sun.com/products/jfc/index.jsp

[108] C. Szyperski. Component Software : Beyond Object-Oriented Programming. Addison-Wesley / ACM Press, 1998.

[109] T. Trachsel. DivX Player für Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2003.

[110] M.von Tessin. Yamaha Audio Trieber für Aos. Semester project, Department of Computer Science, ETH Zürich, 2002.

[111] T. Ungerer and B. Robič; and J.Šilc. A survey of processors with explicit multithreading. ACM Computing Surveys, 35:29-63, 2003.

[112] W3C. Extensible Markup Language (XML) 1.1 Recommendation, 2004. http://www.w3.org/TR/xml11

[113] S. Walthert. Entwicklung eines Style Layers und Renderers für die XML-basierte GUI-Shell des Aos Systems. Diploma thesis, Department of Computer Science, ETH Zürich, 2001.

[114] C. Wassmer. OGG Radio for Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2004.

[115] Y. Weber. MPEG-1/2 Decoder für Bluebottle. Semester project, Department of Computer Science, ETH Zürich, 2005.

[116] M. Wille. Overview: Entwurf und Realisierung eines Fenstersystems für Arbeitsplatzrechner PhD thesis, Institut für Computersysteme, ETH Zürich, 1989.

[117] N. Wirth and J. Gutknecht. Project Oberon - The Design of an Operating System and Compiler. Addison-Wesley, 1992.

[118] R. Wu, A. Kenji, T. Koji A Method for Intelligent Association of Chinese Input Using Inductive Learning. Proceedings of the First International Conference on Information Technology and Applications (ICITA), 2002.

[119] Xiph.org Foundation. Ogg logical and physical bitstream overview. Xiph.org Foundation, July, 2002.
http://www.xiph.org/ogg/vorbis/doc/oggstream.html.

[120] E. J. Zeller. Fine-grained Integration of Oberon into Windows using Pluggable Objects. PhD thesis, Institut für Computersysteme, ETH Zürich, 2002.

[121] J. Zukowski. Java AWT Reference, O'Reilly. 1997.

Curriculum Vitae

Thomas Martin Frey

February 01, 1975	Born in Bern, Switzerland
	Son of Erika and Andreas Frey-Bigler
1982-1986	Primary school in Münchenbuchsee
1986-1991	Secondary school in Münchenbuchsee
1991-1995	Gymnasium Bern Neufeld
1995	Matura Typus C
1995-1999	Studies in Computer Science
	Swiss Federal Institute of Technology, Zürich
2000	Dipl. Informatik-Ing. ETH
2000-2005	Research and teaching assistant at the
	Institute for Computer Systems, ETH Zürich