ANDRÉ MIEDE

REALIZING RESPONSIVE INTERACTION FOR TABLETOP INTERACTION METAPHORS

REALIZING RESPONSIVE INTERACTION FOR TABLETOP INTERACTION METAPHORS

ANDRÉ MIEDE

Master of Science

School of Computer Science
Otto-von-Guericke-University of Magdeburg

April 2006

Miede, André: *Realizing Responsive Interaction for Tabletop Interaction Metaphors*
Master's Thesis, Otto-von-Guericke-University of Magdeburg, © 2006

SUPERVISORS:
Prof. Dr.-Ing. Maic Masuch
Prof. Dr. Sheelagh Carpendale
Dr.-Ing. Tobias Isenberg

LOCATION:
Interactions Laboratory
Department of Computer Science
University of Calgary
Calgary, Alberta (Canada)

TIME FRAME:
November 24, 2005 – April 24, 2006

Ohana means family.
Family means nobody gets left behind or forgotten.

— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.
1939 – 2005

ABSTRACT

Large, high resolution displays and especially tabletop displays can be used as powerful tools to support co-located collaboration of a group of people. The differences of interaction on a large display compared to on a common desktop computer are a fruitful field of work for researchers and developers nowadays. The complexity of both the development and the performance of applications for large displays poses serious limitations for the required user interfaces.

This thesis shows the development and implementation of solutions to both regain responsive interaction on high resolution displays and to support developers build applications for such displays. Successful ideas from computer graphics and selected aspects from swarm intelligence form the basis for a different approach for designing interaction. Centralized, complex user interface control is replaced by a concept for an indirect and localized way to organize interfaces via buffers. This novel and powerful concept is integrated into the *buffer framework*, a modular software framework architecture that is built on best practices from software engineering, such as object-oriented design patterns. This provides developers with access to a framework that both increases the performance of applications for high resolution displays and supports their overall development process.

Finally, the evaluation of these concepts and their implementation shows their superior performance in comparison to an existing prototype application. This success emphasizes that the framework architecture presented in this thesis will be a good foundation upon which to build future software.

ZUSAMMENFASSUNG

Große, hochauflösende Anzeigegeräte – insbesondere horizontal ausgerichtete in Form von Tischen – bieten wertvolle Möglichkeiten zur gemeinsamen Zusammenarbeit von mehreren Personen. Die starken Unterschiede zwischen der Interaktion auf großen Anzeigegeräten und der auf herkömmlichen Computern stellen Forscher und Entwickler vor große Herausforderungen. Dadurch treten jedoch sowohl bei der Leistung als auch bei der Entwicklung von Anwendungen für solche Systeme Komplexitäten auf, die die Möglichkeiten der zu entwickelnden Benutzeroberflächen stark einschränken.

Diese Arbeit stellt die Entwicklung und Implementierung von Hilfsmitteln vor, mit deren Hilfe leistungsfähige Interaktion möglich ist und die außerdem den Entwicklungsprozess von Anwendungen für große Anzeigegeräte unterstützt. Erfolgreiche Ideen aus der Computergraphik und ausgewählte Konzepte der Schwarmintelligenz bilden die Grundlage für ein neues Design von Benutzeroberflächen. Die zentrale und komplexe Handhabung von Benutzeroberflächen wird durch einen indirekten und lokalen Ansatz ersetzt, der Oberflächen mit Hilfe von Puffern organisiert. Dieses mächtige und neuartige Konzept wird innerhalb des *Buffer Frameworks* zusammengefasst. Es handelt sich hierbei um eine modulare Software-Architektur, die von erfolgreichen Konzepten aus der Softwaretechnik, wie objektorientierten Gestaltungsmustern, unterstützt wird. Dadurch erhalten Entwickler Zugang zu einem Rahmenwerk, das sowohl die Leistung von Anwendungen für hochauflösende Anzeigegeräte steigert, als auch ihren Entwicklungsprozess insgesamt unterstützt.

Die Evaluierung dieser Konzepte und der Vergleich ihrer Implementierung mit einer bestehenden prototypischen Anwendung zeigt deutlich ihre überlegene Leistung. Dieser Erfolg betont noch einmal explizit, dass die in dieser Arbeit vorgestellte Software-Architektur eine gute Grundlage für die Entwicklung zukünftiger Anwendungen bietet.

PUBLICATION

Some ideas and figures have appeared previously in the following publication:

Tobias Isenberg, André Miede, and Sheelagh Carpendale. *A Buffer Framework for Supporting Responsive Interaction in Information Visualization Interfaces*. In *Proceedings of the Fourth International Conference on Creating, Connecting and Collaborating through Computing (C^5) 2006*, New York, NY, USA, 2006. IEEE Computer Society Press.

The software presented in this thesis is available via the developer cookbook of the Interactions Laboratory at the University of Calgary:

`http://grouplab.cpsc.ucalgary.ca/cookbook/`

Please note that license and copyright restrictions may apply.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

—Donald E. Knuth [24]

ACKNOWLEDGMENTS

Many people have contributed to this thesis, be it by discussions, inspiration, or different kinds of support.

I would like to thank Sheelagh Carpendale and Tobias Isenberg for getting me interested in the topic of tabletop interaction, for many fruitful meetings, for encouragement, and for giving me access to the Interactions Lab's terrific technology. Many thanks go to Maic Masuch for supervising me despite the great distance and his full schedule.

Stefan Habelski, Henry Sonnet, and Niklas Röber deserve another big "thank you" for helping me to come to Canada and for answering many of my questions.

I am greatly indebted to AIESEC Calgary, namely Layial El-Hadi, Messalina Tiro, and Harshitha Gunawardena for helping me wherever they could (which was quite a lot).

John Jackson and Fabrício Anastácio helped me to get started in Calgary, shared their house with me, and together we realized many cultural differences we were not aware of before.

Together with David Ladouceur, Sabin Schuler, Johnny Aase, Márton Naszódi, and Lothar Schlesier I explored many beautiful places in the Rocky Mountains and managed to come back alive from the more dangerous ones.

To work in the Interactions Lab was really a wonderful experience I will never forget. Many nice and highly skilled people gather there to do world-class research and I want to thank them for sharing much of their time with me: Mark Hancock, Jeroen Keijser, Nicolai Marquardt, Petra Neumann, Mike Nunes, Edward Tse, Lothar Schlesier, Ehud Sharlin, Stephanie Smale, Charlotte Tang, Annie Tat, Kim Tee, Min Xin, Jim Young, and Torre Zuk.

Daniel F. Abawi deserves an extra "thank you" for preparing me so well for this difficult task and for showing me what I am capable of.

Regarding the typography, many thanks go to Marco Kuhlmann, Philipp Lehman, and the whole LaTeX-community for support, ideas and some great software.

Last but not least, I want to thank my family and my girlfriend Friedi for all the support they have given me and most of all: for letting me go.

Magdeburg, April 2006 A. M.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	Application Programming Interface
CSCW	Computer Supported Cooperative Work
DLL	Dynamic Link Library
DRY	Don't Repeat Yourself
HCI	Human-Computer Interaction
RNT	Rotate'N Translate
SDG	Single Display Groupware
SDK	Software Development Kit
STL	Standard Template Library
UML	Unified Modeling Language

1 INTRODUCTION

Social interaction is an important factor in both our private and work life. Especially in many work scenarios, co-workers interact with each other to complete their tasks [37]. However, although collaboration makes work more satisfying and more productive [50], modern desktop computers do not support co-located collaboration sufficiently. They are usually designed as they were thirty years ago for a single person sitting in front of one or more displays, using one keyboard and one mouse. Thus, both research and commercial interest in large displays is increasing as these displays offer a variety of different possibilities due to their characteristics, which include enabling a collective experience of concurrent multi-user interaction and an unconstrained movement in the perimeter of the display [18].

Horizontally oriented large displays—tabletop displays—are motivated by the fact that traditional tables are a powerful and common tool for co-located collaboration [39]. Digital counterparts could finally bridge the gap between the electronic and the physical world on our desktops. Due to the size and the alignment of large displays, interaction becomes very different compared to that on desktop computers. Issues such as orientation arise through the unique point of view of each user, the size makes it sometimes hard or uncomfortable to reach objects, and user territories as on traditional tables have to be considered as well. This requires dedicated software featuring new interaction metaphors and makes it inefficient just to run desktop software on these displays.

Figure 1. Co-located collaboration of several people on a tabletop display [12, 57].

1.1 MOTIVATION

Applications for large displays need to provide responsive interaction because the response time of an application has to be appropriate for the users' tasks [50, page 367]. In this case, the response time needs to resemble that of a traditional table, thus, it has to be immediate to fully utilize the table concept and to support co-located collaboration best.

As the response time of interaction depends on the overall performance of the application, it is important to increase the performance of a system in general. Several factors affect performance and, therefore, have to be addressed.

A serious issue is the complexity of the user interface, especially how it is organized. At the moment, the components of the interface control the visualized content of an application in a global and active way. Due to the organizational complexity, this affects the application seriously, e. g., if the number of visualized objects is increased.

Figure 2. Dedicated interaction metaphors for a tabletop display [21].

An additional decrease in performance results from the much higher resolution on large displays. Many more pixels than on desktop computers have to be controlled and this decrease cannot be addressed entirely by waiting for newer and better hardware support.

Co-located collaboration can also have an effect on the performance as usually multiple concurrent inputs have to be processed.

Another big issue in this research community is the lack of support for developing applications for large displays. Researchers and developers usually build applications from scratch, e. g., each time they design and evaluate new interaction metaphors. This binds valuable resources for tedious programming tasks and makes it often hard to reuse successful solutions.

1.2 RESULTS

This thesis shows the development and implementation of solutions to both regain responsive interaction on high resolution displays and to support developers build applications for such displays.

Successful ideas from computer graphics and selected aspects from swarm intelligence form the basis for a different approach for designing interaction. Centralized, complex user interface control is replaced by a concept for an indirect and localized way to organize interfaces via buffers.

Figure 3. Example application of the buffer framework.

This novel and powerful concept is integrated into a modular software framework architecture that is built on best practices from software engineering, such as object-oriented design patterns.

This provides developers with access to a *buffer framework* that both increases the performance of applications for high resolution displays and supports their overall development process.

Finally, the evaluation of these concepts and their implementation shows their superior performance in comparison to an existing prototype application. This success emphasizes that the framework architecture presented in this thesis will be a good foundation upon which to build future software.

1.3 ORGANIZATION

The thesis is structured as follows:

CHAPTER 2 introduces the reader to large displays in general and especially tabletop displays. It shows how large displays are different to desktop computers and what kinds of interaction metaphors have been developed so far. This and the presentation of important concepts from other research areas provide necessary background information for the following chapters. By pointing out and analyzing particular problems with software for large displays, this chapter also lays the foundation for the developed concepts of this thesis.

CHAPTER 3 describes the developed solutions to the problems explained in the previous chapter. Starting with a solution outline, a buffer concept is introduced and refined to show how interaction can be realized efficiently and in new ways using ideas from computer graphics and swarm intelligence. Utilizing this new concept, a software architecture is developed using important concepts from software engineering, mainly design patterns. The underlying ideas and modules are presented and discussed to reveal advantages and disadvantages and to justify certain design decisions.

CHAPTER 4 shows that the solutions presented in the preceding chapters in a rather abstract way can be realized in program code. Important details about optimizing the rendering process and under which circumstances hardware acceleration has to be employed for buffers is discussed.

CHAPTER 5 has a critical look at the results of the thesis and how they were developed. For this, the work process is investigated and test series compare a standard prototype for interaction metaphors with an application built on the buffer framework concerning their performance. In addition, the quality of the developed software architecture is discussed.

CHAPTER 6 draws conclusions from the developed results and their evaluation. A selection of open and worthwhile questions is provided to give ideas and suggestions for further research based on the findings of this thesis. Personal remarks by the author conclude the chapter and the thesis.

2 RELATED WORK AND ANALYSIS

This chapter introduces the reader to large displays, especially tabletop displays, and points out what different kinds of technology are being researched. Furthermore, it shows why the interaction possibilities on tables are so fundamentally different from the ones used for common desktop computers. The motivation for this thesis' work is developed by an analysis of two major problems that arise with computer applications for large displays and their development. The chapter concludes with an overview of related ideas that are important for the solutions presented in the next chapter.

2.1 INTRODUCTION TO LARGE DISPLAYS

In this section, an introduction to large displays is given. It is explained why large displays are a worthwhile area for research and where current technological issues are. After a general overview, tabletop displays in particular are discussed and one of their promising application areas is presented.

2.1.1 *Motivation*

Today's world is full of computers in many different forms being present in many different areas of our lives. Many—even most—work places have become unimaginable without personal computers providing access to information and communication, e. g., via the world wide web, or without many different kinds of domain specific programs.

Since the *Alto*, the world's first personal computer developed at Xerox PARC in the 1970s, not much has changed: usually, there is still a single person sitting in front of the computer, using a single mouse and keyboard to interact with the machine [54].

Figure 4. The Alto personal computer [65].

However, social interaction plays an important role in our daily life and many kinds of work involve a group of several people interacting with each other, for example, by the means of natural and man-made objects [37, page 20]. The goals and results of cooperation can best be described as Shneiderman does: collaboration makes work more satisfying and more productive [50, page 479]. With decreasing costs for hardware but increasing costs for integrating new technology and the procedures to use it into work places [37, page 21], enhancing the interaction possibilities of groups of people gets more and more important.

These important insights about human working habits and the growing interest in understanding and improving these by means of computers created the research area of Computer Supported Cooperative Work (CSCW). While one major approach of CSCW sees many opportunities in remote collaboration, e. g., by networked computers, there are other ways that were neglected in the beginning but proved to be quite fruitful later on. One of the other approaches investigates the rather natural way of *co-located collaboration*. It introduces a new computer paradigm called Single Display Groupware (SDG) and aims at exploring ways and related technologies that improve group work and interaction [54]. One fast growing area of research to achieve this goal are *large interactive displays*. Large displays support collaboration of groups by enabling more fluent interaction on a large interaction space [39].

An advantage over remote collaboration is that such computer-mediated communication over distances is considered to be impoverished compared to personal and face-to-face communication [37, pages 178–182]. On the other hand, also without technology-mediation, group communication is different to paired communication, as issues such as turn-taking etc. arise. This has to be reflected by the developed technology and new social protocols for compensation are necessary. The field of Human-Computer Interaction (HCI) researches ways to solve these problems. It is therefore tightly linked to large displays as it supports and enables further applications of these kinds of displays.

This section showed that collaboration is an essential part of our lives, which is not sufficiently supported by commonly available and used technology. In this context, large displays are a promising area of research for they offer ways for co-located collaboration. To use them to their full potential, other research areas such as CSCW and HCI offer existing and develop new methods.

2.1.2 *Technology*

Although there are commercial products in the field of large displays, there is still a lot of research necessary to advance the technology [28]. Research on large displays aims at improving the necessary hardware and making it affordable in the long run. When talking about costs, also the construction and the maintenance of the technology have to be considered. And because potential applications of such displays are seen in information, art, and entertainment, affordable solutions and alternatives are big research issues before having large displays ready for the mass-market.

Despite the on-going research, there are common attributes of the different systems. Hachet and Guitton [18] state the characteristics of large display systems and their effects as the following:

- *Large size of the display* which results in an individual point of view for each user and which is usually different from the camera point of view.
- *Collective experiment* as a group, in which multi-user and concurrent interaction takes place.
- *Unconstrained movements* in the perimeter of the display which is not hindered by devices or cables.

- *Visualization of the real environment*, where co-workers, equipment, and input devices (such as the own hands) are still visible. Other approaches, like, virtual reality can make the user feel uncomfortable when disconnecting her from a well-known environment.

At the moment, a major technological limitation of large displays is still the need for high quality imagery [6]. This is especially an issue if users get very close to the display. A current solution to this is generating wall-sized displays by tiling multiple projectors to form one large, single virtual image (cf. Figure 5), as opposed to using only a single projector with a rather limited resolution quality. Using this insight, another characteristic can be added to be above list:

- *High number of pixels* from having high resolutions or from tiling several displays of medium resolution. The number of pixels can be significantly higher than on regular desktop computers and eventually leads to technical issues, such as performance problems due to the higher workload.

But there are not only common attributes: large displays vary in several dimensions such as size, orientation, shape, etc. The *orientation dimension* can be used to categorize them into two different types, as shown in Figure 7 on page 10. Namely, these types are:

- Wall displays.
- Tabletop displays.

Nearly all commercially available displays are designed to be used in a vertical orientation as wall displays. This usually implies a whiteboard-kind of interaction, where a person is using the display for presentation purposes in front of a group. Typical interaction consists of the presenter touching the screen and by this simulating a mouse click, e. g., to proceed to the next slide. But as outlined above, a major goal and advantage of large displays is their capability to facilitate co-located group collaboration. Using such a display mainly for presentation purposes in a vertically oriented way can be unnatural and

Figure 5. Interaction on a multi-projector large-scale wall display [6].

uncomfortable, because it neglects more natural and traditional collaborative possibilities, e. g., provided by tables [39].

Tabletop displays provide users with a similar appearance like that of a traditional table while adding digital display capabilities. Because of the importance of tabletop displays to support collaborative work and because of their powerful possibilities for new types of interaction, they are discussed in more detail in the next section.

Having looked at the technological aspects of large displays, it became clear that the hardware is still too expensive to make the technology available for the mass-market. Although large displays share a set of common characteristics, such as offering a collective experience to users and offering a very high number of pixels. Large displays are usually divided into two classes, depending on their orientation. From these two classes, tabletop displays are considered to facilitate and to improve co-located collaborative work.

2.1.3 *Tabletop Displays*

As already pointed out, a conventional way for a co-located group to collaborate is to sit around a physical table. People communicate and interact with each other by and with the help of objects on such a table [39]. The usual way of organizing work in a contemporary workplace is to divide it into interaction with the computer and into work with actual physical objects. People are therefore moving between two different worlds:

- The *electronic world*, the computer workstation.
- The *physical world*, the "real" desk.

This can pictorially be described as "paper pushing" versus "pixel pushing" [64]. Each world has its own special set of advantages and disadvantages. For example, physical interactions are deeply embedded into humans and have a natural and familiar feeling to them, which led to the introduction of the classical desktop metaphor. However, this metaphor and the classical desktop computer do not effectively support co-located, multi-user collaboration [54].

These insights led Wellner [64] as one of the first to describe the concept of tabletop displays in 1993. His work consisted of building the *Digital Desk*, a combination of a real physical desk with projected electronic images, as shown in Figure 6.

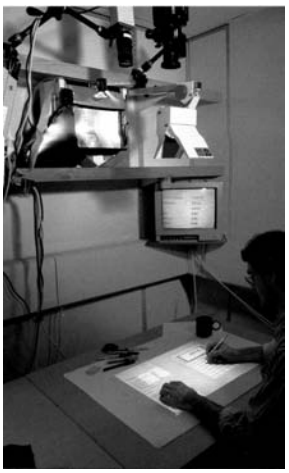


Figure 6. The Digital Desk [64].

There is still a lot of research to be done in this area, as several basic things are still unknown. For example, it is still a research question

what the most appropriate tabletop system would be like [46]. Thus, due to the lack of a standardized system, most researchers design and build their own custom-made tables to investigate the HCI issues of tables.

For this, hardware configuration ideas from tiled-projector, high resolution walls are borrowed and transferred into horizontal orientation. At the University of Calgary a SMART Technologies DViT-board [58] was transformed into a table, providing a five by four feet (about 1.52 by 1.22 meters) table with a resolution of 2,048 by 1,280 pixels (a total of about 2.6 million pixels) [44]. More recent and specially engineered table-prototypes are even capable of resolutions up to 2,800 by 2,100 pixels (a total of about 5.9 million pixels) on the same table size as the one mentioned above, which is a great technological advance. Those tables, which are used for this thesis, are shown in Figure 8.

Due to the open question of table configuration, tables can be categorized according to different criteria [44]:

- *Technology/media used*:
  - Top-projected computer displays on traditional tables [48] using projectors and mirrors.
  - Rear-projected tabletop displays [11] also use projectors and mirrors, but have this equipment installed beneath the table surface.
  - Self-illuminating displays [38, 53] display the screen by themselves.
- *Input devices*[1] *used*: mice, pens, styli and direct touch, or tracked physical devices. For tables in general, it would be a tremendous step forward to enable the possibility of richer, multi-handed, and multi-user input on large displays/ tabletop displays, but this is yet another challenging research area [28].

According to Scott et al. [44], a more general classification scheme is to divide tables into four *application classes*:

- *Digital desks* [64], which are designed to replace traditional workplace desks by a combination of paper-based and digital media, as shown in Figure 6.
- *Workbenches* [11] provide interaction with digital media via a semi-immersive, virtual reality environment which is projected onto a table surface.
- *Drafting tables* [8] are individually used digital displays to replace drawing and drafting tables of, e. g., architects and graphic designers.
- *Collaboration tables* [47, 53] are designed for supporting collaboration of co-located people. Examples for such collaboration activities are planning, design etc.

Examples for these classes are shown in Figure 9.

1 Input device: a device that together with appropriate software transforms user data into computer-processable data [37, page 235].

(a) i-land Dynawall [55].

(b) MERL's DiamondTouch table [12, 57].

Figure 7. Walls and tables.

(a) Top-projected custom-made table.

(b) Rear-projected table prototype by SMART Technologies [58].

Figure 8. Tabletop displays at the University of Calgary.

(a) Responsive workbench [11].

(b) Drafting table [8].

(c) Personal Digital Historian collaborative table [47].

(d) The Pond collaborative table [53].

Figure 9. Examples for the application classes of tabletop displays.

The application area of this thesis' research is in the above-mentioned and strongly related fields of Human-Computer Interaction (HCI), Single Display Groupware (SDG), and Computer Supported Cooperative Work (CSCW). The focus is on *collaboration tables*, because they have a great potential in enabling and supporting collaborative work. Therefore, a variety of commercial scenarios in corporate and other contexts can be thought of, promising this field of research many useful practical applications in the future. One of these and its possible effects is described in the following section. However, the general, underlying concepts and ideas developed in this thesis can also be transferred to other table types and large displays in general.

2.1.4 *Computer-Augmented Tabletop Games*

Games have been used by mankind for thousands of years both for entertainment and education. The computer and video games[2] industry as well as its popularity as a whole have been growing steadily during the last years. By this, computer games have become a very big global business [66]. They have been a major drive behind technological advancements such as hardware acceleration for graphics; they also have been applying and motivating new ideas in HCI, or making personal computers and consoles in general more affordable and more popular; this also led to severe cultural impacts as described by Kushner [29].

Despite the advent of the digital age and digital games, traditional board games are still very popular and successful. This is due to a number of advantages over computer games; however, there are also disadvantages as mentioned by Magerkurth et al. [30, 31] and shown in Table 1. Computer games seem just to be at the other end of the spectrum as they offer dynamical game worlds with complex sets of rules and multi-sensual stimulation. But usually they are lacking social interaction—at least co-located interaction—as human-to-human interaction is mediated by a computer screen in most of the numerous multiplayer games available.

ADVANTAGES	DISADVANTAGES
⊕ Strong social situations and events by direct interaction and both verbal and non-verbal communication between players.	⊖ Purely physical game nature limits the domain of possible games.
⊕ Physical playing pieces which feel good, could be collectibles etc.	⊖ Establishes sometimes awkward interaction patterns.

Table 1. Advantages and disadvantages of traditional board games.

2 In the following, the terms "computer game" and "video game" are used synonymously.

This leads to the desire for some kind of computer game experience that is interwoven with the real world. There are different approaches to this being investigated [34, 35], and one of these are *computer-augmented tabletop games*. These build on the success of old-fashioned board games and modern computer games, combine emotionally involving strong social situations with the capabilities of large displays, and offer a hybrid form of entertainment and/or education. This is achieved by using the computer for the virtual game logic, to track the game states, to provide atmospheric visual and audio assets, or to take an active role as a participant. The human players still preserve their social interactions by gathering around a table and by having the opportunity of engaging physical acts such as the rolling of dice and the touching of game pieces [4].

An experimental system which implements these ideas is the STARS platform [30, 31]. It uses an interactive table with a touch-sensitive display for the game board. Physical playing pieces serve as a tangible interface similar to a traditional board game. An overhead camera tracks these pieces and the players' positions (their hands). Wireless technology is used to detect other game items such as walls. The setup avoids any classical computer interfaces like mice, keyboards etc. at all. Figures 10 and 11 show the experimental game *KnightMage* during game play and its setup.

Figure 10. STARS game setup [31].

If we look at the history of personal computers and how its evolution is still tightly coupled with gaming requirements, the wish for a similar development in the area of large displays and tables sounds reasonable. This might contribute to motivating further research, building business models based on tabletops or at least increase the public's awareness and acceptance of this new technology.

Figure 11. STARS' made *KnightMage* during game play [30].

2.2 TABLETOP INTERACTION

As we have seen above, one research challenge for large displays and especially tables is *hardware*. The goal is to provide affordable, high-resolution displays that allow multiple concurrent input.

What is at least as important, is the need for appropriate *interaction metaphors*. Many traditional desktop user interfaces and known techniques are not applicable to large displays. The different types of display and input media change how users interact with and relate to the digital information provided, as Kurtenbach and Fitzmaurice [28] pointed out. The challenge in this area is therefore one of design, namely, how to design the interactions with the tabletop display [46]. Several of the interaction challenges have been identified and investigated by a variety of researchers [6, 26, 44]:

- *Orientation*, with users having different views on table objects.
- *Territoriality*, as traditional tables have different spaces with special uses associated with them.
- Other challenges include topics such as
  - *Remote reaching*, because the interaction space might be huge and drag-and-drop could mean to walk several meters, as in Figure 5 on page 7 for example.
  - *Sharing of objects*, because users need to exchange data during collaboration.
  - *Space and layout management*, as data may be outside of the user's focus and needs different layout techniques than overlapping windows on a regular computer desktop.
  - *Command input*, to overcome, e. g., the lack of a keyboard.

The following sections show a selection of these challenges in more detail and describe researched solutions to them. They have been chosen due to their importance and the quality of the developed solution. This information is used later on in this thesis to show what different kinds of interaction metaphors have to be supported by tabletop software.

2.2.1 *Orientation*

When people gather around a physical table or tabletop display for collaboration, it is very likely most of the collaborators have a different view on the presented objects. One of the tasks of interaction is to handle these different views, namely, to solve the issue of the objects' orientation.

There are different approaches to this, as shown by Kruger et al. [26, 27]:

- *Fixed orientation*: For this it is assumed, that all users sit side-by-side at the table and, thus, only one fixed orientation is necessary.
- *Multiple copies*: Each user has her own copy of tabletop objects. These may then be oriented in any way the individual user likes.

- *Person-based automatic orientation*: This idea orients objects towards the user who has most recently interacted with the object.
- *Environment-based automatic orientation*: Because user positions may not be trackable at all times, this approach orients objects based on their position on the table or within the tabletop environment.
- *Manual orientation*: User have to and can rotate each item manually and decide how it is oriented.

The problem of orientation is very difficult and so far there is no real basis on how to inform a decision about orienting a particular object on the tabletop display.

An important step forward was the discovery by Kruger et al. that orientation has three distinct roles in collaboration:

1. *Comprehension*, to ease reading or to help doing tasks.
2. *Communication* independent of speech, to communicate an intention.
3. *Coordination*, where orientation is used as a cue for personal spaces or the ownership of an object.

Kruger et al. developed a technique named Rotate'N Translate (RNT) to enable a fluent change of orientation. RNT allows the user to simultaneously rotate and translate an object by a single fluent motion. At the same time making it platform and technology independent (e. g., from special input devices). For this, a single 2D contact point is used: together with a movement vector from there and pseudo-physics that simulate moving the object against friction, the object is rotated and translated at the same time. Figure 12 illustrates this further and shows some examples of moving an object with this method.

From this can be concluded, that it is a crucial part of an interactive tabletop display to have an interaction metaphor addressing orientation issues.

2.2.2 *Territoriality*

It is a natural habit for humans to do space partitioning and establishing territories when working on traditional tables or tabletop displays [45]. These territories have different purposes and may be for personal items, for sharing objects with the whole group, or for storing something until it is needed again later. Thus, social interactions between people working together are mediated through laying claim to a certain space. This enables a transition between personal and group work and concurrent interaction. In addition, social interactions and tasks can be coordinated by providing shared access to objects and making them transferable between special spaces [36]. Such spaces have spatial properties, e. g., size, shape, and location as well as functionality, such as reserving an area or task resources.

These findings led to the idea of *Storage Bins*, developed by Scott et al. [44], which was also motivated by the so-called "pile-metaphor" [32] found on physical desktops. The underlying concept of the pile-metaphor is the insight that users like to group items spatially to organize them. Such an organization through creating a pile of objects is

not reflected by the common desktop metaphor of a hierarchical tree structure.

A Storage Bin providing this functionality and reflecting the territoriality aspect of tables is shown in Figure 13 on and has the following attributes:

- *Container*, with capabilities of adding and removing items by users.
- *Resizable*, to adapt to different amounts of stored objects.
- *Mobility*, to move all stored objects easily around on the table.
- *Shrinking* of contained objects to conserve screen space.

In order to have tabletop displays with rich user interfaces that facilitate collaborative work, the above-mentioned different types of territories have to be supported. Borrowing ideas from traditional tables and enhancing these with possibilities such as the shrinking of contained objects make metaphors like the Storage Bin intuitive and natural ways to organize work items.

2.2.3 *More Challenges*

Apart from the problems and solutions described in the previous sections we conclude this chain of thought with two more.

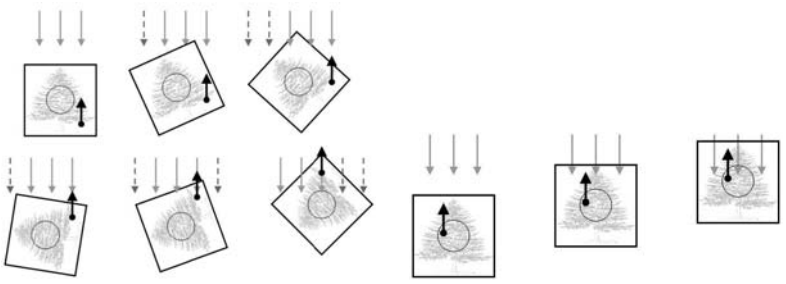
SHARING AND REACHING: Due to the size of the display, passing objects to other people or reaching remote objects can be hard. The table interface should support an easy handling of these issues.

Hinrichs et al. [20] came up with the idea of *Interface Currents*, inspired by the concept of conveyor belts known from Sushi restaurants or airports. An Interface Current consists of a continuous onward movement of objects at a certain location with a certain speed and is within boundaries, as shown in Figure 14.

A similar approach was developed by Shen et al. [47] with a circular display metaphor for the Personal Digital Historian to share and display items, shown in Figure 9(c).

Such interaction metaphors become even more important as the size of tabletop displays increases with advancing technology. Although the main focus of these metaphors is on sharing and reaching, they are strongly related to territoriality as they can also provide different kinds of storage areas for the user or the group.

COMMAND INPUT: Command menus known from desktop computers can also be useful on large displays. Especially context menus are of a high value due to the great distance a fixed menu can be away. Guimbretière and Winograd [17] developed the idea of *FlowMenu*, a special kind of context menu system suitable for large displays. FlowMenu offers a radial selection of menu items, with the input process resembling shorthand writing. This enables command, text, and data entry on a display using nothing more than a pen-based device. Figure 15 shows an example of using FlowMenu.



(a) Movement that causes an upward translation and counterclockwise rotation.

(b) Upward translation without rotation by touching the translation-only area (indicated by the circle).

Figure 12. RNT examples [27]. The black dot and the black arrow depict the user's touchpoint and movement vector.

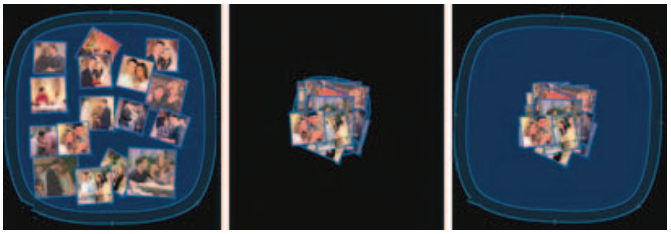
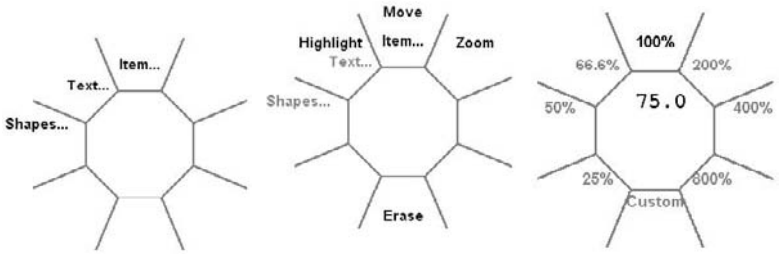


Figure 13. Storing images in a Storage Bin [44].

Figure 14. Interface Currents on a tabletop display [21].



(a) First-level menu, selecting "Item".

(b) Second-level menu, selecting "Zoom".

(c) Bottom-level menu to select the zoom value. The current value is shown in center.

Figure 15. Steps of a FlowMenu context menu for zooming [17].

2.3 TECHNICAL COMPLEXITY

Hardware research for large displays advances, producing displays with higher resolutions and, thus, more pixels to display. In the research branch of interaction metaphors, researchers come up with new interaction techniques for tabletop displays. What is affected by and affects both of these areas is the development of actual software for large displays. The technical problems that arise when implementing applications can be divided into two areas that are described in the following two sections.

2.3.1 *Performance*

Large displays and therefore also tabletop displays visualize information to support collaboration. Thus, they have to be considered being extensions of interactive media from the desktop domain to other input and output media.

As were have seen in examples of Section 2.2 and as Isenberg et al. [22, 23] pointed out, there are two types of entities in interactive systems:

1. Actual *visualization objects* that carry the information to be communicated.
2. So-called *control structures* or *interface components* that guide the behavior of the visualization objects and that are manipulated by user interactions.

The Storage Bin solution presented in Section 2.2.2 is a good example for this: Visualization objects could be images or textual elements. An interface component to control such objects could then be the Storage Bin [44], which stores, resizes, and relocates the stored objects.

Realizing this concept with regular methods raises the following computational limits:

- The number of visualization objects is limited due to the steering and monitoring by the rather complex and application-specific interface components.
- The complexity of the interaction between the visualization objects and the interface components can become very high and time-consuming.
- Even higher might even be the complexity between several interface components when interacting.

These limitations slow down the whole system which hinders interactivity. It is quite a severe problem if timely user interaction is required, as users prefer shorter response times or at least the response time should be appropriate for the users' task [50, page 367].

Although this might be just "problematic" on regular desktop computers, it is definitely a serious issue on large displays due to the huge amount of pixels. A four by five feet tabletop display with a resolution of 2,800 by 2,100 pixel has approximately *5.9 million pixels* in total.

Compared to a relatively low amount of 1.3 million pixels on a desktop machine (at a resolution of 1,280 by 1,024 pixels).

For example, using only one interface component and between 50 and 100 visualization objects (images) the rather intuitive implementation of the Interface Currents described in Section 2.2.3 [20] yielded the following measurements:

- 25–30 frames per second on a 1,280 by 2,048 tabletop display.
- 5–10 frames per second on a 2,560 by 2,048 tabletop display.

The tests were conducted on tabletop displays of the Interactions Lab at the University of Calgary using the free version of FRAPS [56].

While the framerates on the lower resolution table are still suitable for interaction, the rates on the other one are beyond responsive interaction. With improving technology offering higher resolutions, the current implementations will probably become obsolete. A potential solution would be to wait for computer hardware to improve, providing faster CPUs and graphics cards. However, sophisticated algorithms and concepts in software might yield even higher gains and also in less development time. In addition, these solutions would also profit from the hardware accelerations later on. Thus, improving software concepts behind the applications is a promising research area for near-future performance improvements.

2.3.2 *Application Development*

The second technical difficulty is the actual process of developing applications for large displays. Software development by itself is a hard and expensive task, costing time and money, and constitutes still a rising figure while hardware costs are dropping [52, page 2]. However, software does not only mean computer programs but includes also their documentation: the documentation is usually mandatory to successfully install, use, develop, and maintain the software and must not be undervalued.

Software development is even harder if the domain itself of the application is not fully understood as it is usually the case with research prototypes. This is especially true for tabletop displays where applications are built, e. g., to evaluate new interaction metaphors (cf. Section 2.2).

In addition, there are virtually no commercial or industrial high-end applications for tabletop displays yet and, thus, there is little to no support in developing such programs. Any kind of support usually consists of toolkits for highly specialized certain aspects like multiple concurrent user input, for example, Tse and Greenberg's SDG Toolkit [61], or for orientable widgets [49].

Therefore, applications are usually developed from scratch, leading to a waste of valuable resources in the "mere" process of software development. A big issue with this are the high costs and wasted efforts from the continuous rediscovering and reinvention of core concepts [42]. These resources are drawn from the main task—which is usually not software engineering in this area of research—and finally make the

resulting custom-made applications hard to maintain, hard to reuse, and difficult to reconstruct for other researchers [61].

At the moment, the important technique of *software reuse* is poorly addressed in the field of tabletop displays research. Software reuse is important for cost reduction but also for the improvement of productivity and does not simply emerge as a by-product of regular software development as Sommerville [52, pages 312–318 and 327] pointed out.

Concluding from this, the lack of support for application development for tabletop displays might have hindered past research and is also likely to negatively affect its future.

2.4 MISCELLANEOUS CONCEPTS

This section gives an introduction to concepts of computer graphics and software engineering that are related to this thesis. It presents an overview of important background information, such as buffers and design patterns. This information is used to lay the reader's foundation to understand the following chapters better.

2.4.1 *Buffers*

The personal computer dictionary [33] defines a "*buffer*" as a temporary area for storage that is usually located in RAM. Purpose of this is to provide fast non-harddisk access by the CPU.

Buffers are heavily utilized in computer graphics, such as using buffering for raster image display technology. There, a buffer stores what the display controller is going to put on the screen. For this, the necessary data is stored in a 2D array which is mapped pixel by pixel to the screen. This array is called a *bitmap* (one bit per pixel) or *pixmap*, if many bits per pixel are stored. To store multiple bits at a pixel position can have different reasons:

- To provide color information. 24 bits can store about 16 million different colors.
- To have control information, e. g., for input devices.
- To enable double buffering of two images and, thus, having two (or multiple) buffers. One image is refreshed on the screen while the other one is being updated.

The term "*framebuffer*" denotes the actual buffer memory where the data is stored and is well-known in computer graphics [14, pages 9–15 and 166]. Advantages of this concept are fast access to the stored data and easy application of the information to the display, for example, by raster scan. Disadvantages include the discrete representation of the buffer data which results in aliasing effects and complexity due to converting graphical objects to the raster.

Probably one of the most well-known buffers is the *z-buffer* invented by Catmull [9] in 1975. This buffer records depth information of an image in a bitmap. Purpose of this approach is to simplify hidden

surface removal and thus accelerating image rendering. For this, a z-buffer value is associated with each pixel of the image to be rendered, containing the smallest value for the depth z recorded so far. During rendering, a point is only written to the framebuffer if its depth is less than its value in the z-buffer.

Due to this rather simple buffer concept, it is independent of the object's representation: be it polygonal, constructive solid geometry, etc.; which is one of its greatest advantages [63, pages 189–198]. Because of the simple array data structure used, it is rather simple to implement, which can be considered a great advantage for application development. However, the main disadvantage of z-buffering is the possibly high memory consumption which depends on how much depth information has to be recorded, respectively, how accurate this information has to be for good rendering results.

Yet another innovative use of the buffer concept was developed by Saito and Takahashi [40] by introducing geometric buffers (G-buffers). These are used to speed up the rendering of 3D shapes. For this, the rendering process is divided into distinct steps:

1. *Geometric* processes: perspective projection, hidden surface removal, etc.
2. *Physical* processes: shading, texture mapping, etc.
3. *Artificial* processes to enhance the image: edge enhancement, line drawing illustrations, contour lines, hatching, etc.

Rendering acceleration is motivated by the desire to explore different types of image enhancements (Step 3) without long rendering times for the different images. To achieve this, G-buffers are used to store intermediate results of the geometric rendering process (Step 1) per pixel. Such intermediate results consist of geometric properties such as depth or normal vector of the visible object. From this result set stored in 2D, a variety of 3D images featuring different enhancement techniques can be rendered by using only 2D image processing operations, thus gaining rendering speed.

As the examples show, buffers can be applied in various contexts to improve performance and to simplify complex systems. They can be used to store a variety of properties and enable a focus on local data awareness as there is a mapping, e. g., between a screen position and a certain associated buffer position. Furthermore, their simplicity and 2D data structure support fast access to and modification of stored information.

However, the general problems of memory consumption and discretization issues have to be considered and weighed in the application domain.

2.4.2 *Design Patterns*

A good tool to design software architectures are *design patterns*. They have been part of the programming folklore for many years, in terms that they make it easier to reuse successful designs and architectures.

However, design patterns are by no means limited to programming but were used even earlier in many other fields such as architecture or languages. The most basic and general definition is [43]:

> *A design pattern is a recurring successful solution to a common or standard problem.*

Apart from offering a solution to a problem, their greatest advantage is providing a vocabulary for talking about particular problems.

Here, design patterns are used as *object-oriented* design patterns[3] as formalized and introduced to computer science by Gamma et al. [16] in 1995. This step was heavily influenced by Alexander et al.'s [2] ground-breaking work on pattern languages for architecture, building, and planning. Actually, Alexander gave a much more precise definition of "design patterns"; Gabriel and Coplien [15] suggest the following, more Alexander-conform, definition for design patterns in software development:

> *Each pattern is a three-part rule, which expresses a relation between a certain* context, *a certain* system of forces *which occurs repeatedly in that context, and a certain* software configuration *which allows these forces to resolve themselves.*

An important effect of design patterns is the bridging of abstractions in object-oriented analysis and design with the specific realizations of these, which is an advantage for implementation and maintenance of software. Therefore, they are not mere theoretical constructs but have practical value and impact. Schmidt et al. [43] attribute the following to design patterns:

- The success of a solution is more important than its novelty.
- The information communicated by a pattern is clear and understandable.
- Good design patterns arise from practical experience and have succeeded many times in the past.

To describe a design pattern formally, Gamma et al. [16, page 3] define its following essential elements:

- *Name*, to describe a problem, its solution etc. in a word or two. It is the basis to build a common design vocabulary, thus enhancing communication.
- *Problem*, explains in what scenario to apply the pattern.
- *Solution*, explains the design that solves the problem. This includes relationships, responsibilities, and collaborations of used components.
- *Consequences*, describe both the positive and the negative effects of the pattern. Especially the trade-offs are very valuable for evaluating the usage of a pattern in a certain situation. This is important in case an alternative might be more suitable in a scenario.

3 In the following for simplification just called "design patterns".

The connection to this thesis is the very strong relationship between design patterns and frameworks. Usually, object-oriented frameworks embody many patterns, but this is not true vice versa [16, 43]. The formalized and clear structure of describing design patterns can be used to document the form and the contents of frameworks.

These benefits are used in this thesis to make the software architecture better accessible and understandable by users. The following paragraphs give a brief and informal overview of two used patterns [16], as these are heavily utilized later on. Understanding these patterns is crucial for understanding the solutions presented in Chapters 3 and 4.

COMPOSITE PATTERN: Applications that use simple objects and more complex objects built from the simpler ones face the problem of a rather complicated implementation. At least, this is the case if the objects have to be distinguished from each other by the program while the user usually treats them identically. An example would be a diagram which consists of text, geometric forms, etc. Composition resembles a tree structure as shown in Figure 16.

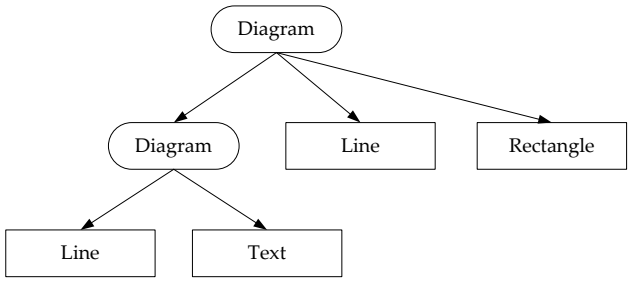


Figure 16. Composite tree (after [16, page 164]).

The idea of the Composite pattern is to provide one abstract class for both simple and complex objects—for both primitives and containers. By this, application code is able to ignore the difference between these two types.

The resulting classes and their collaborations look as depicted in Figure 17 and can be described as follows:

- *Component*, provides an unifying interface to handle composed objects.
- *Application*, works with composed objects by the Component interface.
- *Leaf*, implements the behavior for the primitive objects, which do not have children.
- *Composite*, implements the behavior for components with children and handles child-related operations.

The consequence of this idea is a class hierarchy of primitive and complex objects, where primitives can be composed into more com-

plex objects and these into even more complex ones. From a framework point of view, it makes the integration of new components very simple as they work automatically with existing application code and structures due to subclassing from the abstract interface. Last but not least, application code is easier to develop because it does not have to care about the object type; all objects are treated the same by the application and special cases are handled by the classes of the Composite pattern.

To create composed objects, the Builder pattern is often used.

BUILDER PATTERN: This pattern might seem rather complicated and confusing when encountered the first time, but the power and advantages it provides after understanding it are well worth the effort.

The basic idea is to create a variety of different objects (such as a composite object) from one single object, which could be a control object within the main application. Before we come to an example clarifying how this pattern works, we have to look at the participating classes and their structure (cf. Figure 18):

- *Product*, a complex object that is being created.
- *Builder*, an abstract interface with methods for creating separate parts of the Product.
- *Concrete Builder*, a specific implementation of the abstract interface; it returns the built Product upon request.
- *Director*, calls the Builder interface to construct special Products.

The UML sequence diagram shown in Figure 19 shows the general steps of creating a Product. A little example illustrates this further:

- The *Product* we are interested in is a meal from a fast-food restaurant.
- Usually, the components of those meals are quite similar, regardless of where you go. This will therefore be our abstract *Builder* interface:
  - `buildBurger()`
  - `buildFries()`
  - `buildSoftDrink()`
  - `buildSundae()`
- Now, a Concrete Builder is a specific restaurant and how they make these parts of your desired meal:
  - `McHaroldsBuilder`
  - `BoingBoingBurgerBuilder`
  - `MandysBuilder`
- Your mind respectively your stomach is the *Director*, knowing what different kinds of meals you are interested in, depending on how hungry you are:

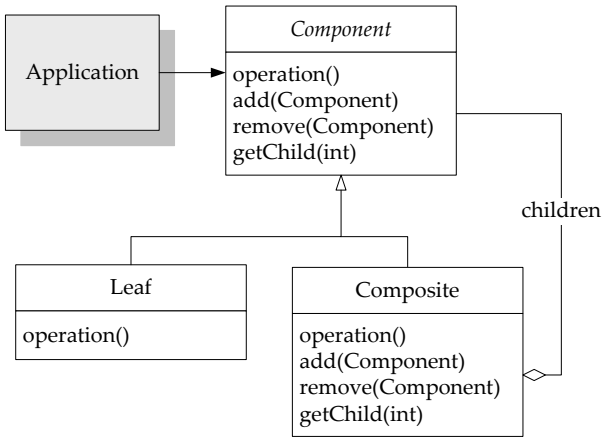


Figure 17. Composite pattern structure (after [16, page 164]).

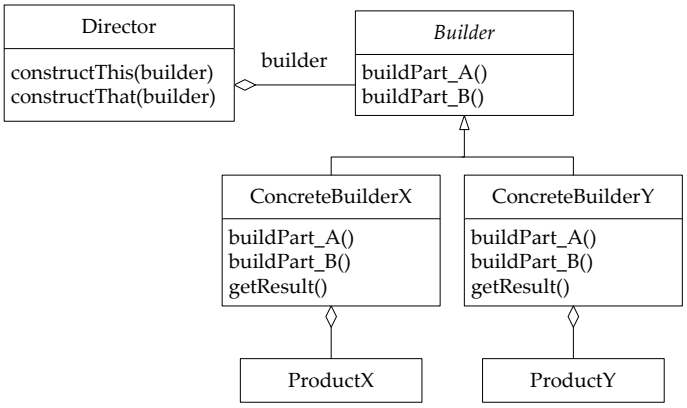


Figure 18. Builder pattern structure (after [16, page 98]).

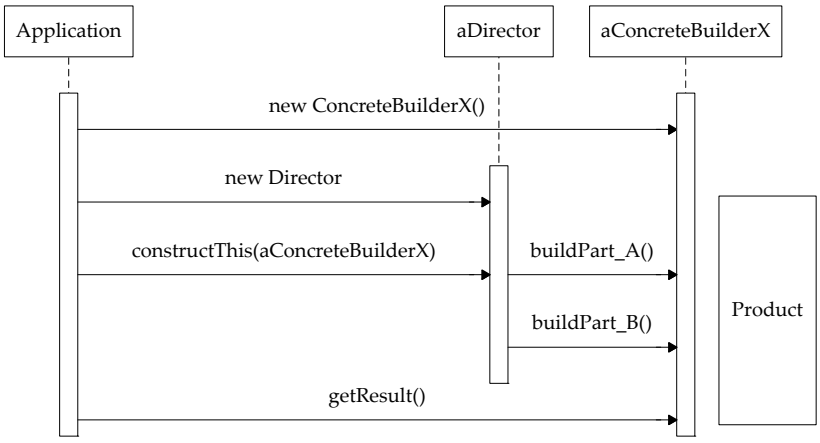


Figure 19. UML sequence diagram for the Builder pattern (after [16, page 99]).

- `hungryMeal(builder)`
  - `builder.buildBurger()`
  - `builder.buildBurger()`
  - `builder.buildFries()`
  - `builder.buildSoftDrink()`
  - `builder.buildSundae()`
- `snackMeal(builder)`
  - `builder.buildFries()`
  - `builder.buildSoftDrink()`

Using this structure, you would first decide where to go for a meal (choose the Concrete Builder). Then you would order there one of your meal-combinations you have in mind (create the Director and call its desired method with the Concrete Builder to build the Product). From the meal components on your wish-list the restaurant will make you a hopefully tasty meal (the Director calls interface methods of the Concrete Builder to build the Product). The restaurant gives the meal to you after it is prepared (the Concrete Builder returns the Director and/or the calling application).

Once the pattern is implemented and setup, very few lines of code can encapsulate powerful and complex processes to create objects. As for our fast food example above, pseudo-code to get lunch from McHarold's yourself looks like this:

```
myLunch = me.hungryMeal(new McHaroldsBuilder);
// lunch is two burgers, fries, a soft drink, and a sundae
```

Or if you send a nice friend to get you a snack from Mandy's:

```
mySnack = aNiceFriend.snackMeal(new MandysBuilder);
// snack is fries and a soft drink
```

Consequences of using the Builder pattern are that the internal representation of a complex object may vary, as it can be created by different Concrete Builders. Users also have very fine control how an object is created, as different Director methods are possible. Thus, different fine-tuned Products can be produced using the Builder's interface. Finally, this results in the fact, that the code for creation is isolated from the code that represents the actual object.

Object-oriented design patterns provide both solutions to common problems and a vocabulary to talk about problems and designs. Their focus lies on the success of a solution and they come from practical experience. Good patterns have succeeded many times before. Patterns are heavily linked to software frameworks, as these usually consist of several patterns and use patterns for their documentation. For this thesis, two powerful patterns are particularly important.

- The Composite pattern because it provides ways to simplify application code and allows for extensions without affecting existing code.
- The Builder pattern because it provides high flexibility and simple interfaces for creating complex objects.

In the following chapters these patterns are used to solve difficult design problems and to document them.

2.5 CHAPTER SUMMARY

In this chapter we have seen that large displays enable and support co-located, collaborative work of groups. Despite the currently predominant vertically oriented displays, horizontal displays relate much more to traditional and common working devices and have led to the concept of tabletop displays. Interaction on large displays and especially on tables is quite different from that on desktop computers and requires other metaphors than those already available. These metaphors are implemented in soft- and hardware to be evaluated.

Large displays introduce severe *performance problems* to the applications running on them. This is due to the much higher number of pixels they offer, to the nature of the visualized objects and how they are controlled by applications at the moment. An effect of this lack in performance is the limitation of user interaction caused by the low frame rate and busy system. The rather intuitive approaches used so far do not address this performance problem satisfactory and make more elaborate ideas necessary to regain responsive user interaction.

The second problem discovered is *application development for tabletop displays*. The absence of adequate tools makes nearly all application designers build their programs from scratch, thus wasting valuable resources which could be used in a better way in other areas of their projects.

Computer graphics and software engineering offer interesting and important concepts such as buffers and design patterns that contribute to solving the problems stated above.

3 METHODOLOGY

This chapter first gives a brief overview of this thesis' solution to the problems described in the previous chapter. This is followed by detailed descriptions of a *buffer concept* and the design of a *framework architecture* to build tabletop applications.

3.1 SOLUTION OUTLINE

It was shown in the previous chapter that there are severe limitations on the research for interaction metaphors and for the development of tabletop display applications in general. Therefore, there is a need for a concept that speeds up the performance of tabletop applications and, thus, enables responsive interaction. There is also the need for software that can be used as a foundation to build tabletop applications, as this simplifies the development process and fosters further research. This call has also been made in the literature, e. g., by Bezerianos and Balakrishnan [6].

According to Sommerville, the characteristics of well-engineered software are *maintainability*, *efficiency*, and an *appropriate user interface* [52, pages 4/5]. Thomas and Hunt [59] reduce the criteria for good code even more to one single underlying quality: *flexibility*.

> *The solution proposed by this thesis is the design and the implementation of a framework to build responsive tabletop applications.*

The term "*framework*" is used here according to Schmidt and Fayad [42], meaning an object-oriented, reusable, and semi-complete application that can be specialized to produce custom applications. A more concrete explanation of the term can be found in Gamma et al. [16, page 26], where they define it as a set of cooperating classes that make up a reusable design for a specific class of software.

As mentioned, such a framework is usually targeted at a particular application domain, which puts it in contrast to libraries, especially those that provide common functionality. For this thesis, the specific class and application domain of the proposed framework is the development of interaction metaphors for tabletop displays.

Heavily influenced by Sommerville, his above mentioned characteristics have to be addressed:

To address the serious performance issue, buffer ideas as described in Section 2.4.1 are transferred to interaction issues, developed in more detail, and incorporated into the framework to use and evaluate their benefits.

Choosing the framework approach enables the use of the benefits frameworks are well-known for [42]:

- modularity,
- extensibility, and
- reusability.

Gamma et al. [16, page 27] go even further and see a framework's benefits in its focus on *design reuse over code reuse*. Users should be programming to an interface, not an actual implementation. This is also reflected by the usage suggestion: "reuse the main body of a framework and write the code it calls". This can be seen to address the issue of maintainability and flexibility, as maintainable software must offer the possibility to be extended and reused. In this context, design patterns as introduced in Section 2.4.2 play an important role to achieve this goal.

To address the issue of the user interface, which can be seen in this context as the Application Programming Interface (API), we return to Schmidt and Fayad [42] and the fundamentals of frameworks: Volatile implementations are encapsulated behind stable interfaces. Components are defined in a generic way, so they can be reapplied for new applications. Furthermore, it is possible and allowed to extend stable interfaces. This results in frameworks usually providing appropriate user interfaces.

Of course, a framework also has its drawbacks, which are mainly connected to its development: developing complex software is known to be a hard process and developing frameworks is even harder. It requires a wide range of skills in object-oriented analysis and design, implementing, and application programming [42]. For example, reusable software does not simply emerge as a by-product of regular software development; it is a special task that needs extra effort to be successful [52, page 327]. Testing the components of a framework is also hard as testing is not possible in isolation. Therefore, framework errors are hard to distinguish from errors in their application. Another aspect is the integration of multiple frameworks and libraries which has to be considered during the design phase and is crucial for reusability and extensibility. Concerning the usage of a framework, it is also a big issue that it requires effort to learn and understand it. Therefore, information and documentation about how it works and how to use it is mandatory [52, page 327], because a framework is just as good as the people who built it and those who use it [42]. For this, we make use of the strong relationship between framework documentation and design patterns as described in Section 2.4.2.

To illustrate how the framework is integrated into a tabletop system, Figure 20 depicts how such a system works. At the top, different kinds of input are fed via the table or other hardware into to the system, ranging from multiple touches to colored laser beams etc. At the bottom, different interaction metaphors are shown, which are implemented by the framework and its API. Both meet at the center in a specific application that is built using the framework and that runs on the necessary hardware.

Having these requirements and benefits of the proposed solution in mind, the undertaking of this thesis must not be considered as just routine software development. Difficult problems of large displays are identified and addressed by transferring successful ideas from other fields. Details on how to solve these problems are presented in a general way and are eventually implemented. This implementation enables researchers to benefit from the derived solutions and to reuse

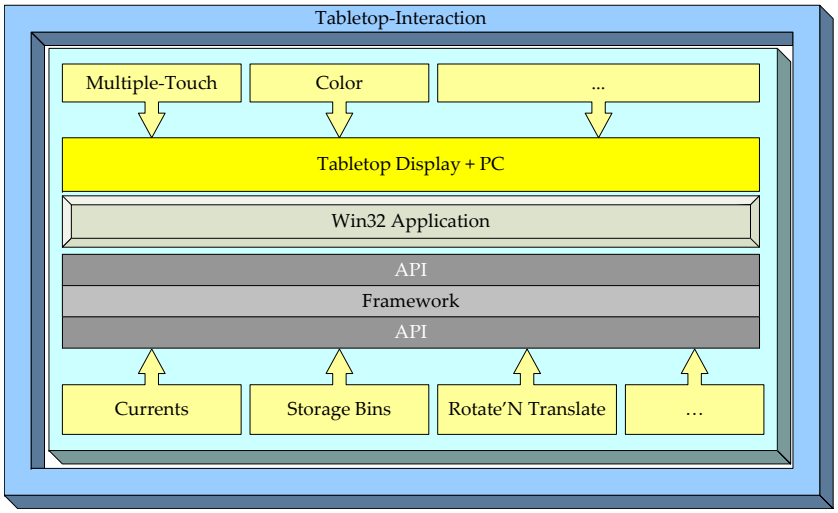


Figure 20. Tabletop application scheme.

and extend them for their further needs. We can, therefore, conclude that this work can be considered as a worthwhile scientific contribution to the tabletop research community.

3.2 BUFFER CONCEPT

As shown in Section 2.4.1, buffers allow for the possibility to increase performance and to simplify systems. Motivated by these findings, this section describes a buffer concept to improve responsive interaction on tabletop displays, addressing the performance issues described in Section 2.3.1.

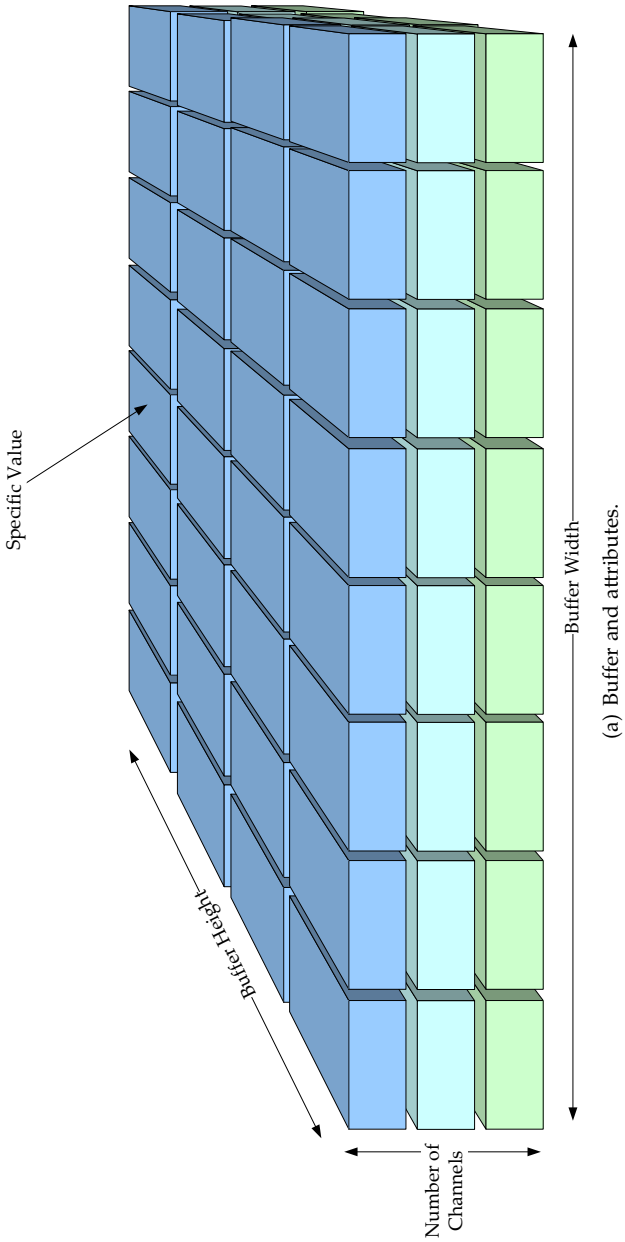
First, the general idea is presented to give an introduction to this novel approach. After that, more details are added to the overall concept to enhance it and to make it work better.

3.2.1 *Basic Idea*

The described performance problems of visualization systems are directly related to how the visualization is organized [20, 23], because the steering and monitoring of visualization objects by interface components[1] can be very costly tasks. In this context, especially complex interactions or operations such as access to objects or steering objects by analytical calculations are very expensive. Eventually, this results in bottlenecks if these approaches are scaled to a higher number of visualization objects and interface components.

It was outlined before that the main advantage of buffers are their fast access to data and the local awareness by mapping a position to a certain buffer value. As seen in the example of the G-buffers on

1 See Section 2.3.1 for a definition of the terms "visualization object" and "interface component".



(a) Buffer and attributes.

(b) Buffer stack.

Figure 21. Schematic buffer concepts.

page 20, buffers can store different kinds of properties. Transferring this to interaction on large displays, the basic idea is as follows: the complexity of the interactive system is reduced by modeling the steering of the visualization objects by layered buffers. Buffers are used in this concept to store properties of interface components and to make these properties available to visualization objects. By this, a buffer provides a value (or more than one) at an object's position, which can be of any type such as an integer, a floating point number etc. Figure 21(a) shows the different attributes of a single buffer. To store more than one property, more than one buffer is necessary and a *buffer stack* is introduced, as depicted in Figure 21(b).

For example, to model motion in a certain direction, a buffer can also store vector data, stored in so-called channels: this means, at a certain buffer position more than one value is available, e. g., an x, y, and z direction. This addresses the issue with the expensive analytical calculations mentioned above: The buffer just stores previously-computed, discretized values which represent, e. g., motion of objects. Then, expensive calculations are only necessary when creating or modifying the whole buffer with the specific values and not at every movement step. This is even true for frequent and dynamic changes, as most interactions not necessarily affect a whole buffer but only parts of it that have to be modified.

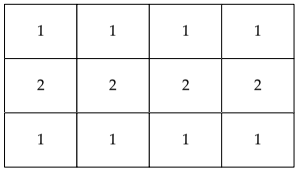
By this, the visualization objects are no longer directly controlled by the interface components but interact with the saved values of each buffer in the stack. To achieve the complex behavior of the visualization objects that before was achieved by their complex interaction with the interface components, ideas borrowed from *swarm intelligence* [7] are used. Now, each visualization object acts as a separate entity with a local awareness, namely, of the values throughout the buffer stack at its current position; these are also used by the visualization object for making local decisions (cf. Figure 22). As in swarm intelligence concepts, a visualization object is limited to its own domain and it is not aware of the other objects that are also influenced by the underlying buffers (the swarm as a whole).

Similar ideas to use self-organizing systems in connection with data stored on 2D grids were previously used by other researchers as well:

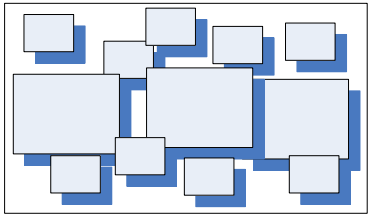
- Fall and Fall [13] developed the SELES system to simulate landscape dynamics. Landscape structures are represented in this system by layers of grids which consist of fixed-size square cells. These cells store values about vegetation coverage, topography etc. By applying probabilistic disturbances to them, changes to the landscape over time are modeled.

- Baker et al. [3] designed their system *GeneVis* for simulating and visualizing genetic regulation networks in real-time. The simulation environment is a 2D grid representing a symbolic view on a biological cell. These grids are used to control the resolution of the simulation, to track the positions of the proteins and genes involved, and to determine reactions between them.

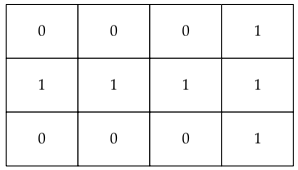
- Schlechtweg et al. [41] developed a system to facilitate the rendering process of stroke-based non-photorealistic images. Behavior-based, autonomous agents—called *RenderBots*—are simulated on the source image and on G-Buffers (cf. Section 2.4.1) to render individual strokes of the result image.



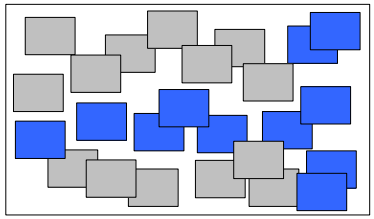
(a) Contents of a size buffer.



(b) Objects using the size information.

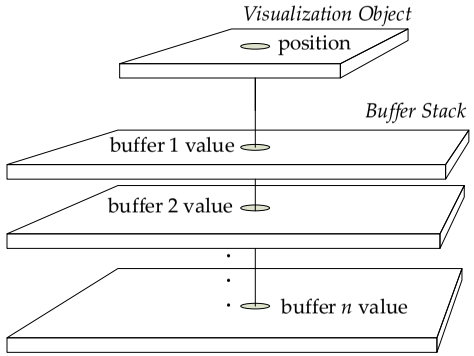


(c) Contents of a color buffer.

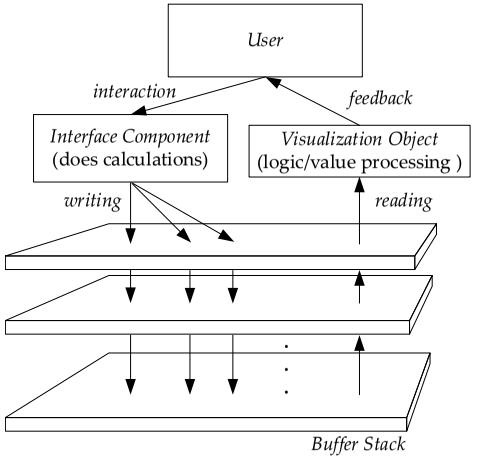


(d) Objects using the color information.

Figure 23. Buffer examples: how buffer contents are able to affect object properties.



(a) The buffer stack and the local awareness of a visualization object.



(b) Relationship of an interface component, a buffer stack, and a visualization object during user interaction.

Figure 22. Schematic view of how buffers are integrated into interactive systems.

These approaches aim at improving the simulation and visualization of biological processes or at rendering images in certain styles. Thus, they differ fundamentally from the approach taken in this thesis, which is to realize responsive interaction of a user interface.

It is important to understand that the buffers in this concept do not contain any logic or algorithmic information but numerical or Boolean values. By this approach, logic is transferred from the interface components to the visualization objects (cf. Figure 22(b)). Thus, the interface components are relieved of the complex administration and steering tasks and the swarm intelligence concept is utilized. The objects read their locally aware values from the respective buffer location, process them, and decide how to react. Therefore, a certain buffer usually serves a special purpose within the stack, e. g., one buffer could be used for object size, another for orientation, yet another for color, and so on; Figure 23 shows examples for these different uses. However, as mentioned before, the object itself decides what to do with this information, because the information processing is done locally, and it is possible to think of an object that does not use buffer data at all or that uses a size buffer for other processing than size as well. The role of the interface components is to modify the buffers. While the visualization objects read and react, the interface components are manipulated by the user and, in consequence, write to the affected buffers in the stack. By this they eventually affect the objects. See Figure 22(b) for a schematic visualization of this concept.

The assumption of this buffer-centered interaction approach is that it decreases the overhead caused by the interface components and increases the number of visualization objects while maintaining responsive interaction. An early prototype implementing a rough buffer concept achieved slightly higher frame rates than those noted for the Interface Current application in Section 2.3.1 but *displayed 1,000 objects*, which is a performance gain by one order of magnitude. Figure 24 shows the visual difference between about 100 and 1,000 visualization objects.

(a) "Original" Interface Current [21]. (b) Buffer Interface Current Prototype [23].

Figure 24. Interface Currents with different numbers of visualization objects.

From this follows that there are essentially three factors why the described buffer concept improves the performance of a visualization.

First, the elements of the visualization only have *local awareness*; therefore they have fast access to crucial data and are not burdened with unnecessary information. Second, information is provided in a *discretized* way, thus, recurrent time consuming calculations for analytical representations are avoided. Third, *information is processed locally* in the visualization objects, providing local decision-making on how and when to use retrieved information.

Nevertheless, these are just the basics of applying buffers to interaction problems in tabletop applications. Further details are addressed in the next section.

3.2.2 *Details and Issues*

To fully benefit from the buffer approach, several important details have to be developed. These details affect performance and the simple application of buffers for interaction purposes.

As mentioned above, buffers are used to represent steering data and by that control objects. A difficult question in this context is *where and how to store these buffers*. Basically, there are two alternatives that are discussed in the following:

1. Using a big central buffer stack that is relative to the screen or window size.

2. Using several smaller buffer stacks—one for each interface component, relative to its bounding box.

Each option offers different advantages and disadvantages which have to be weighed carefully. The biggest advantage of the central stack is its easy connection scheme for all interface components and visualization objects. Everything is permanently connected to the whole buffer stack and decides what to do with it (as described in the previous section). However, this also results in one of the biggest disadvantages of this idea: it is extremely difficult, if not impossible, to have multiple interface components working independently of each other. Possible solutions to this problem would require high computational efforts for numerical combination schemes for the buffer data or extremely memory consuming extensions of the buffer stack. A similar problem is that the stack size would grow for each new attribute that is needed by an interface component or visualization object. In addition, if certain attributes (and, therefore, their corresponding buffers in the stack) would not be used often, huge areas of memory would be wasted. The memory consumption should not be underestimated, considering the high resolutions for which this approach is developed for. A single 4 byte buffer at the table resolution of 2,800 by 2,100 pixels uses more than 20 megabytes of space. An important aspect to consider in this context is that buffer access is fast, as long as we have the buffers in the fast memory. As soon as we exceed the capacity of this type of memory, paging will severely affect any buffer related operations.

In contrast to this, the second approach requires less memory because each interface component or visualization object has its own buffer stack just in its needed size. This also allows interface compo-

nents to be completely independent of each other, while leaving open the option of combining different buffers later on.

Another advantage is that the buffer stacks are independent from the window or screen size. This allows for accessing these buffers even from outside of the screen; it is very useful when interface components are moved partially or completely off screen but the contained objects are still moving. A local buffer stack also enables other uses of buffers such as having a button buffer where certain buffer areas represent buttons and are linked to functionality.[2] However, this makes a rather sophisticated connection scheme necessary. This scheme is required to get the connections right between steering and steered components, as a wide variety of different buffer types might be involved. On the other hand, this allows for a high flexibility and extensibility as new buffer types and new interface components using these can be added on top of the existing ones.

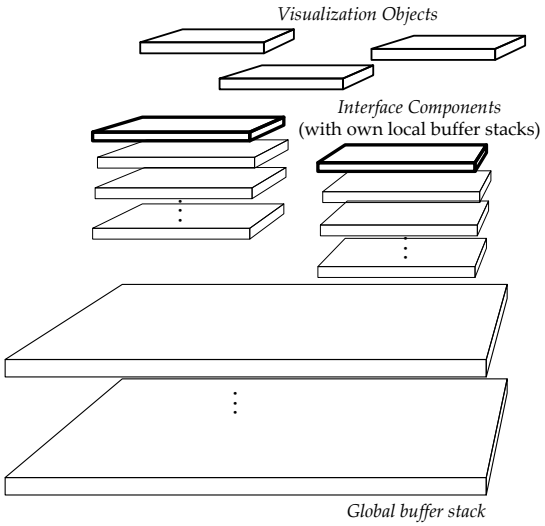


Figure 25. Global buffers, local buffer stacks, and several visualization objects on top.

Being aware of these characteristics of the different approaches, a combination of both is chosen for this thesis: In general, each interface component has its own buffer stack and profits from the advantages of this. These buffers are called *local buffers*. In addition, *global buffers* relative to the screen size support the communication between different interface components and also visualization objects. They might be useful for tasks such as picking and dropping which is discussed later on.

The following paragraphs explain important details that round up the concept.

GENERIC CONNECTION SCHEME: As mentioned above, supporting any number and any kind of buffers requires a generic way to make these buffers available for visualization components that want to be steered by them. To achieve this, every interface component and visualization object divides its buffers into two different categories:

1. *Active Buffers*: From the point of view, e. g., of an interface component, these are the buffers it creates, initializes, and modifies to control others. These buffers constitute the local buffer stack.

2 This concept is described in more detail later on.

2. *Passive Buffers*: From the point of view, e. g., of a visualization object, these are the buffers it looks into for information on how to behave. Usually, these are references to active buffers of the interface component it is connected to.

These two categories are completely independent of each other *within* one component or object. To allow a simple connection, every used buffer is labeled with a specific user-defined type, for example, `size` or `orientation`. Applying the procedure depicted in Figure 27, this together with the two categories is the key to generic connection: an interface component features a set of buffer types for which it provides buffer data.

Upon connection, a visualization object is only connected to buffers that it needs and which are provided by the steering interface component. The internal processing is then only done with the available information.

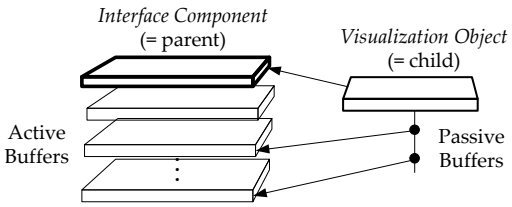
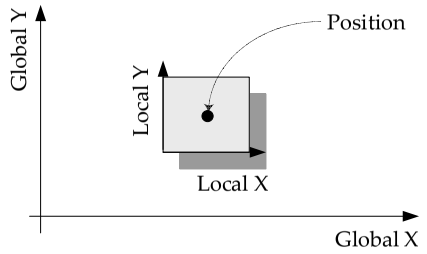


Figure 26. Active and passive buffers.

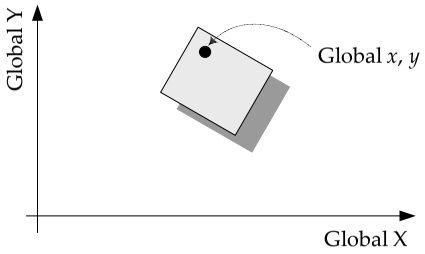
The greatest advantage of this approach is that new buffer types can easily be added to the existing ones without affecting existing components or objects as connection and disconnection is completely automatized.

BUFFER TRANSFORMATIONS: Due to the local aspect of local buffer stacks, these stacks are tightly coupled to the geometric representation of the interface components. For initialization or modification purposes, components write data into their buffer stack and in many cases this data depends on the actual shape of the component. An example for this would be the previously explained Interface Currents: their current-like shape would be reflected, e. g., by the underlying orientation buffer that always orients contained objects to the outside of the Interface Current. In this context, we might also call this a special *render method*, as information is rendered into the local buffer stack. However, interface components as well as visualization objects are subject to many interactions by the user. Many of these interactions influence the geometric representation and, therefore, also the local buffer stack. In particular, *rotation* and *resizing* are the most critical operations which have to be discussed in greater detail.

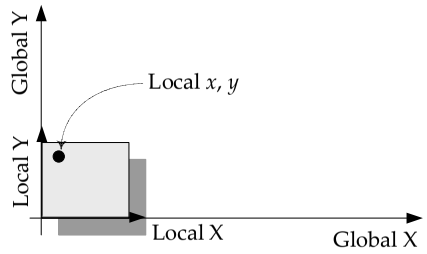
Rotation can either be dealt with by rotating the whole buffer stack or by transforming the coordinates on the geometry into local (not-rotated) coordinates within the object whenever necessary, e. g., for access. While a rotation of the whole buffer stack is not feasible in an efficient way, the coordinate transformation provides a fast and easy way to rotate the geometry and to leave the buffers untouched. Figure 28 provides a schematic overview of the different coordinate systems that are necessary to handle the transformations.



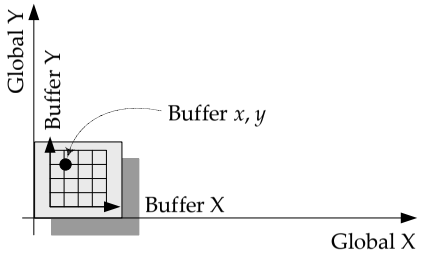
(a) Local coordinate system of an interface component or visualization object. The position is specified by the center of the object in global coordinates.



(b) Rotated object and access location with global coordinates x, y.



(c) Transformation into local coordinates.



(d) Mapping onto buffer coordinates.

Figure 28. Global, local, and buffer coordinates.

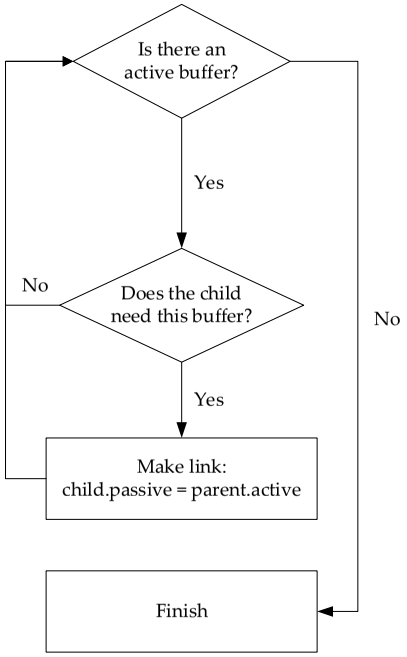


Figure 27. Generic connection of active and passive buffers.

To transform global screen coordinates into local component coordinates, the following matrix product is applied:

$$\begin{pmatrix} \text{local}_x \\ \text{local}_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \frac{\textit{baseWidth}}{2} \\ 0 & 1 & \frac{\textit{baseHeight}}{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos -\theta & -\sin -\theta & 0 \\ \sin -\theta & \cos -\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -\text{center}_x \\ 0 & 1 & -\text{center}_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \text{global}_x \\ \text{global}_y \\ 1 \end{pmatrix}.$$

Input for this transformation are the global coordinates global_x and global_y. Output are the respective local coordinates relative to the bottom left corner of the component's or object's bounding box. The parameters *baseWidth* and *baseHeight* provide the base dimensions of this bounding box and the current rotation is provided by the angle θ, where the center of the rotation lies at the object's center.

The second critical operation mentioned above is resizing. Basically, there are two different ways to address this problem:

1. Adjust the size of the buffer stack.
2. Provide mapped access to the buffer and keep the original size.

The first approach offers two alternatives on how to actually adjust the size. Either all old buffers are deleted, new buffers according to the updated size parameters are created, and the geometry is rendered into these buffers accordingly. A different approach would be to enlarge or shrink the existing buffers and to interpolate their contents according to special interpolating functions. For this thesis, the first approach is chosen for simplicity and reusability reasons. All functionality that is required to delete, create, and to render the local buffer stack is already in place due to the general buffer concept as described above. Thus, all this functionality can be reused to perform this task. The only reason to provide the extra functionality of the interpolating approach could be a gain in performance, which is not given: interpolating between all the buffer cells is very likely to cause a high computational effort and might also result in numerical imprecision.

The second approach of mapping the actual geometric size onto the buffer size is the most simple and fastest way to address the resize problem. Using the following equation to transform local coordinates into coordinates on the buffer grid

$$\begin{pmatrix} \text{buffer}_x \\ \text{buffer}_y \end{pmatrix} = \begin{pmatrix} \text{local}_x \times \dfrac{\textit{bufferWidth}}{\textit{baseWidth}} \\ \text{local}_y \times \dfrac{\textit{bufferHeight}}{\textit{baseHeight}} \end{pmatrix},$$

it becomes obvious that buffer size and geometrical size are independent from each other and even raises the general question of how big

buffers have to be. As this thesis deals with buffers for interaction, this question cannot be answered definitely. Various factors influence the answer: the general resolution of the display, the user's subjective impressions and expectations on how the interface will react, and probably most important, the input resolution of the used input device. Therefore, this thesis tries not answer this question but the presented concept offers flexible and simple ways how to adjust to the different needs. For this, the two size-related procedures mentioned above can be used: If necessary for highest precision and if computational costs are not too important, the buffer size can be adjusted to equal the geometric size of an object. It could even be used the opposite way as any desired ratio between geometry and buffer stack can be established. No matter how this ratio is, the simple procedure of coordinate mapping can always be employed to translate a coordinate from the local geometry system to a buffer coordinate. Thus, either procedure or a combination of both can be used, depending on the context and requirements.

BUFFER ACCESS: Retrieving information from the buffers can be done in different ways depending on the context.

The usual and most simple access is to access each discrete buffer cell directly by specifying its unique position. This is important and useful to avoid ambiguities, e. g., if retrieving a certain id from an id buffer. However, if a buffer is accessed from arbitrary positions on and off the buffer grid, a different technique is more appropriate. For example, if a smooth motion has to be performed by a visualization object that reads buffer data from its current position, this could not be achieved well by just the direct cell access. Instead, four cells are considered: the one the current position is mapped to and the three being closest to this one. The contribution of each cell is calculated by placing a square with the area of 1 around the current position and calculating the coverage of each cell. The retrieved value can then be calculated by weighing the cell values with their coverage and adding these up. This approach makes smooth transitions between cells possible and should be used, wherever continuous access to buffer data is needed, such as for the motion example above. The disadvantages for this approach are the higher access costs due to the interpolation computations and numerical errors that arise with the used data types. Section 6.2 on future work outlines a possibility how to give developers more flexibility for the used interpolation method which both determines the computational cost and the quality of the results.

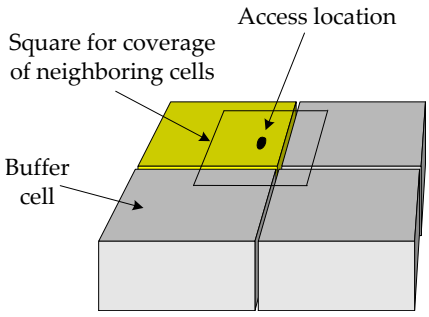


Figure 29. Weighed access to a buffer.

As was shown, several issues arise when looking at the buffer concept in more detail. Although the concept can be partially considered a space-for-time acceleration technique [5, Appendix 4], space must not be wasted. Therefore, the ideas of *global* and *local buffer stacks* were introduced. To realize this advantage and to allow for the uncomplicated extension of the types of buffers, a *generic connection scheme* was devised. This scheme is based on the concept of *active* and *passive buffers* and clearly divides between buffers that are used to steer other objects and buffers that are used to gather data for an object's own behavior.

Transformations on the geometric representations of interface components and visualization objects raised the question how this affects the underlying buffer structures. Transforming relevant coordinates into the required space was shown as the most efficient solution in the context of responsive interaction. This also led to the important question which size the buffer stack should have relative to the geometry. This question was deliberately not answered, as the answer is depending on many different factors. Instead, two different concepts—coordinate mapping and a generic buffer resize process—were presented. Depending on the factors and the context, a suitable solution can be selected. The solutions are also not mutually exclusive but a combination or an easy transition from one to other is possible.

3.3 FRAMEWORK ARCHITECTURE

As pointed out before, developing applications for tabletop displays is hard due to several different factors. Support for this task is in most cases non-existent. This section introduces an object-oriented framework—the "*buffer framework*"—that employs the previously devised buffer concept and that is specially designed to aid the building of tabletop applications. These two benefits combined enable developers to produce applications that feature responsive interaction with less effort on the software engineering side. This leaves more resources for their actual tasks such as designing new interaction metaphors.

First, the main setup of the framework and the motivation for several design decisions are explained. In addition, high-level interdependencies between the framework's modules are revealed. Second, the modules and their interfaces are discussed to show their capabilities to handle general as well as special problems. These sections do not describe implementation-specific details but the overall design of the framework architecture. The way to present this information is top-down to provide the reader first with the general ideas which are fleshed out more and more as the text proceeds.

3.3.1 *Main Layout*

The buffer framework is especially designed to support the building of tabletop applications, addressing the issues of performance and software reusability as outlined before.

It is not an application by itself but a major part of one. This allows the framework to focus on what it is strong at: enabling responsive

interaction. At the same time, this approach allows for including other frameworks and toolkits in the application as well. Thus, the developer is not limited concerning which foundations to build the application on or how to get input into the buffer framework, as these parts of the application are completely exchangeable.

By this, the buffer framework avoids competition with top-notch software in these areas and leaves room for them to be integrated; developers can pick their favorite or what is best suited for a task.

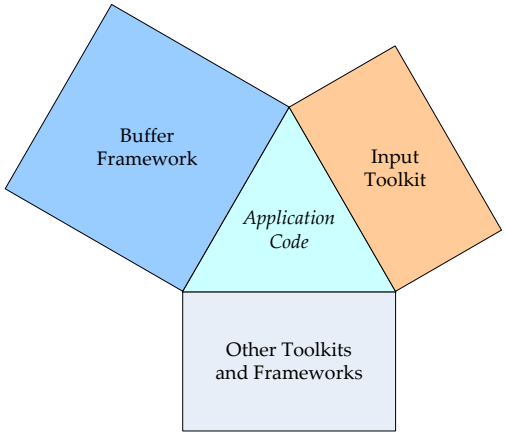


Figure 30. Typical setup of a tabletop application.

A typical setup of this kind consists of the following: At the center is the application code, that includes the specific logic of the application. The application is typically built on top of a framework or toolkit that handles all window and operating system-related things (shown at the bottom). For tabletop displays, multi-user interaction and therefore, multi-user input is of great importance. For this, usually special toolkits are provided, e. g., by the table manufacturer. This kind of toolkit has also to be integrated into the application code to get access to the multiple user data. Finally, the buffer framework is hooked up to the application, providing a dedicated interface that gathers all relevant data to represent the respective interactions and visualizations.

To achieve this setup, the framework has to provide a flexible and yet easy-to-use interface towards the application to grant access to all relevant data and functionality. The approach to tackle this is heavily influenced by Thomas and Hunt's object-oriented design technique "DRY, shy, and tell the other guy" [59, 60]. While this is a rather uncommon name for a paradigm, it contains important insights on how to successfully design object-oriented architectures. In the following, these three aspects and their effect on the buffer framework are explained.

- *Don't Repeat Yourself (DRY)*: This idea deals with the way information is represented within an architecture. To quote Hunt and Thomas:

  > *"Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system."*

  The presented framework architecture reflects this idea by having a framework core that takes care of the creation, destruction, and general organization of all relevant visualization and buffer data. It also takes care of the interrelationships between all par-

ticipating entities. There are different ways to access this information but eventually all requests end up in the framework core, where the single, unambiguous, and authoritative representation of knowledge lies. Figure 31 shows the three main modules of the buffer framework.

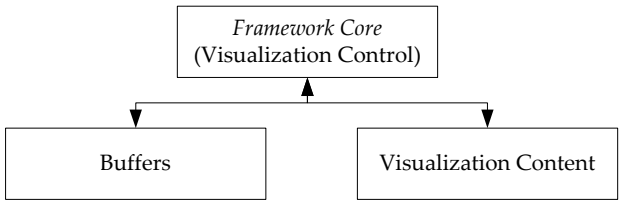


Figure 31. The main modules of the buffer framework (schematic view).

- *Shy*: This aspect refers to the different kinds of coupling modules of an architecture. Unnecessary coupling must be avoided to make the architecture more robust, better maintainable, and easier to extend. In a "shy" design, internals are not revealed to "strangers" and modules are not to nosy about other modules' internals. A shy architecture also does not heavily rely on internal hierarchies of other modules. Daisy-chaining of commands must be avoided, as such a design easily breaks when things change. A very fragile example would be this:

  ```
  getOrder().getCustomer().getAddress().getState()
  ```

  This thesis' solution addresses these issues by providing simple data and input methods to access only the required information and functionality. The framework architecture makes no assumptions about the application data but relies on and processes only the data it is being provided.

  To further emphasize the decoupling of the framework's modules, a concept named *orthogonality* [60, pages 34–43] is used. The basic idea is: if one module in an architecture changes, it does not affect others. Thus, it aims at eliminating effects between unrelated things and tries to avoid designs that rely on the properties of things which might be subject to change and which are therefore not under control, such as the command chain above. Having separate, unifying interfaces for each of the three framework modules, the following benefits result:

  - A standardized interface for the core, the visualization control, offers the possibility to have different kinds of functionality or contents behind these interfaces without affecting the application design. Practical advantages of this approach lie in changing the whole visualization at run time and still using the same application code. Other advantages of this were discussed above.

  - A unifying interface for all visualization content (which will eventually be interface components and visualization objects) allows for automated processing of these by the visualization control. Furthermore, new content can be added later on without affecting existing code of the core, the application, or other content.
  - Having the buffers also separate and detached from the rest of the framework, buffer internals behind the interfaces can be changed and optimized without affecting modules that use the buffers. However, such changes might affect the visual appearance, e. g., if internal calculations for interpolating buffer positions are changed.

- *Tell the other guy*: The idea behind this concept is a service-oriented, operation-centric viewpoint towards code that is calling a method. Most of the framework's interfaces are designed to take orders from calling code, e. g., from the application and to do what is requested. This allows the application to send a message to the interface to do something and not to care about how this request is actually executed. It is very different to the approach that the application code asks for data and does the processing itself. The great benefit for an architecture is that it is less vulnerable to changes, as the way of execution behind the stable interfaces might change, without the calling code noticing or even caring about this change. (This is strongly connected to the "shy" concept described above.)

To bring these ideas together in the buffer framework, the design is mainly built on having a framework core, the visualization control, that offers methods for common tasks. Application code calls these methods to access functionality without caring how they are actually defined. The core itself handles parts of these calls but, in general, functionality is divided between the different modules of the framework. This allows for decoupling of different tasks and exchanging or extending existing methods without the need to change the calling code.

Concluding from this, the usage of the buffer framework is, therefore, in general twofold (cf. Figure 32):

1. On the *application side* the developer has to decide how to use the provided interfaces best for her needs and how to build an application from this.

2. On the *content side* the developer has the possibility to design new interface components and visualization objects by implementing the required interfaces. Thus, the new content can be handled automatically by the existing framework. The content side offers the additional possibility of introducing new types of buffers or to use buffers in new and different ways. (It is also possible to introduce content that completely ignores all kinds of buffers as mentioned before.)

Figure 32. Two ends to use and to extend the buffer framework. The core usually serves as some kind of black box.

Strictly speaking, there is also a third way, namely to extend the framework's interfaces that are provided by the different modules. This option should be used carefully and extensions must always consider how existing code and the overall design of the framework is affected.

To realize the discussed characteristics of the buffer framework, design patterns are employed: the Builder pattern and the Composite pattern, as introduced in Section 2.4.2. The schematic architecture using these patterns looks as depicted in Figure 33. The next section gives detailed information on the internal design of the modules and how the design patterns are adapted to the requirements of the buffer framework.

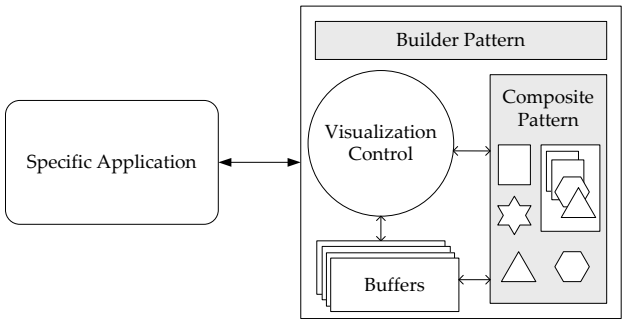


Figure 33. Modules and design patterns of the buffer framework.

3.3.2 *Modules and their Interfaces*

In the previous section, a rough description of the modules was given and the main design ideas were discussed. This section gives more details about the modules' capabilities and how important aspects of the interfaces look like. The intention behind this is to illustrate the internal structure of the framework, where special benefits and drawbacks are, and what parts are renderer-specific. This leads to an important aspect of the framework: the integration of rendering APIs. As the topic of this thesis deals in many ways with visualizations, support from the available high-class rendering APIs is required. The buffer framework deals with this issue by providing different layers within its architecture. Figure 34 shows an overview of the layers and what classes they contain.[3]

3 Abstract classes are indicated UML-like by *italicized* names and renderer-specific classes are labeled with an "R" at the end of their name.

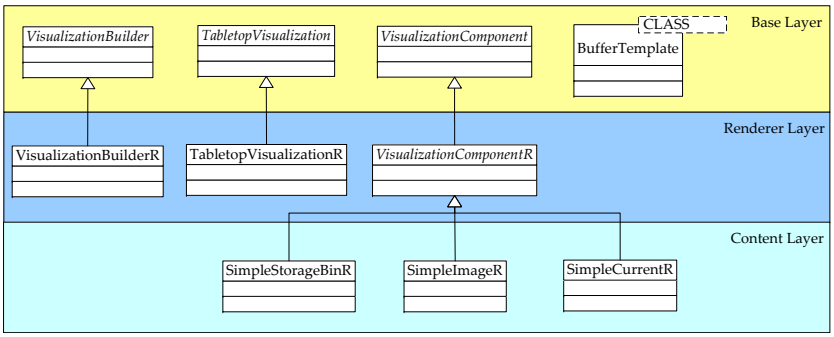


Figure 34. Layers of the buffer framework.

The top layer of these provides abstract base classes for all modules that are directly involved in the visualization part, namely the core and the content. The buffers are renderer-independent as they are decoupled from the visualization and provide the math and functionality behind it.

Functionality is implemented as high as possible in the class hierarchy to allow for general handling of problems and to avoid code duplication. All renderer-specific elements have clearly defined interfaces defined at the abstract base. These interfaces have to be implemented for the used rendering API. If other, more specialized code is required deeper down the class hierarchy, the object-oriented concept of *dynamic binding* still enables the developer to override certain methods and to have the desired functionality for her objects.

The following describes each module, its interface, and utilized design patterns.

The Visualization Content Module with the Composite Pattern

As explained in greater detail before, a tabletop application consists of interface components and visualization objects. Interface components contain and control visualization objects, which usually are the main carriers of information. However, user interaction with both is commonly quite similar. For example, on traditional tables users pile and group items, and treat the resulting piles similar to regular items by moving and grouping for example [32]. Some interaction metaphors developed so far reflect this best practice successfully [44]. As shown in Section 2.4.2, the Composite pattern supports these ideas by simplifying the underlying software architecture. For this, one abstract base for both interface components and visualization objects is designed, called

`VisualizationComponent`.

This allows for general treatment of both types by other code. While the code does no longer care about the differences between these two, the developer still does. The Composite pattern, therefore, offers a special terminology to differentiate (cf. Section 2.4.2):

- Everything is now a *Component*, and this expression is used whenever it does not matter what kind of Component is being dealt with.
- A Component becomes a *Composite* if it can contain other Components.
- A Component that cannot contain anything else is called a *Leaf*.

Figure 35 shows an example for Composites and Leaves for tabletop interaction metaphors by presenting them in a tree structure.

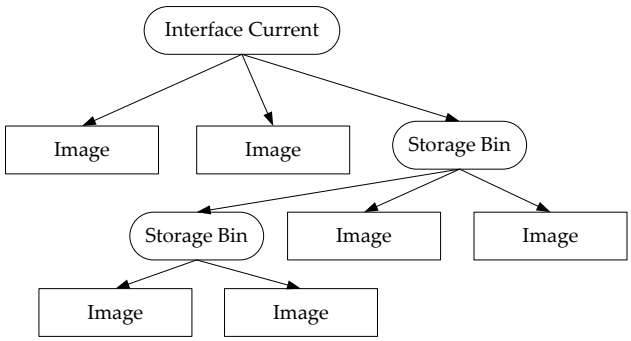


Figure 35. Composite tree of interaction metaphors.

Having defined this communication base for the developer, it is fairly easy to decide of what kind a new Component has to be and which parts of the interface have to be considered. However, all visualization content has to be derived from the abstract base `VisualizationComponent` or from one of its children. This enables the compatibility with the framework and allows the developer to make use of the automatic processing capabilities.

From a software engineering point of view, having the same base for both Composites and Leaves raises the question where to define the functionality for the Composites as this is not needed by the Leaves. According to Gamma et al. [16, pages 167–169] there are two possibilities:

- *Compressed*: The base contains only definitions for those methods that are used by *all* Components. Other methods will be implemented by the respective subclasses.
- *Transparency*: All possible functionality is included in the base, although it might not be used by all Components.

For the buffer framework the transparency approach is chosen. Reasons for this decision are the following:

- The clarity of the interface is valued much higher than its space consumption. This makes the framework easier to understand and to document as fewer special cases have to be considered by developers.

- Reuse of the design and the code is much easier because default definitions of all functionality are provided right at the base of the class hierarchy and do not have to be reimplemented at child level. This is especially useful for the big and sophisticated Composite methods which constitute important foundations of the framework.

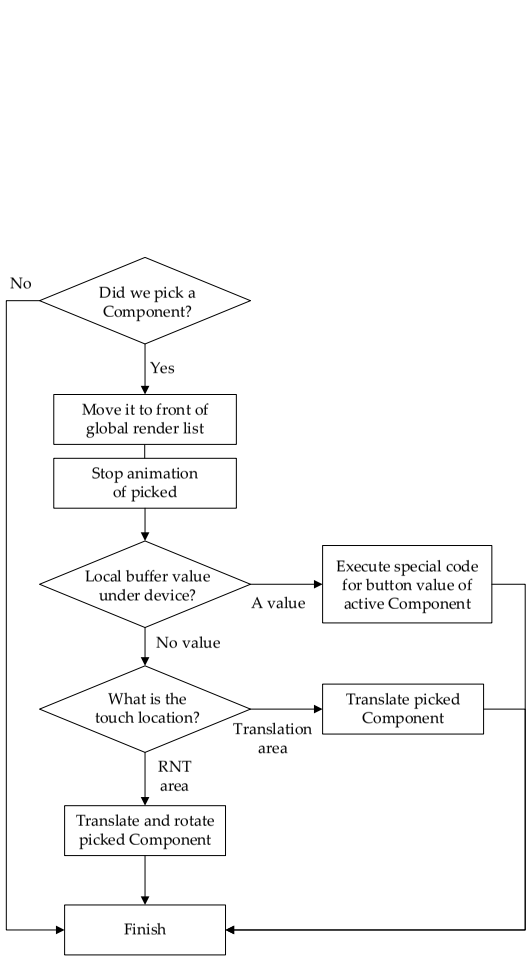
After clarifying these important design decisions, a concise overview of the interface is given. This overview is structured into different parts which describe the general attributes, the default functionality, the abstract part, and the renderer-specific part of a base class.

GENERAL: A Component has several important attributes that are crucial for its behavior.

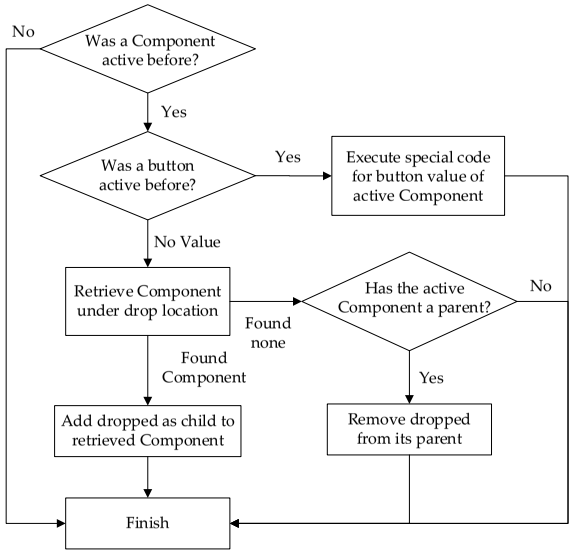
- *Reference to the visualization control*: To access buffer creation functionality and other important concepts that are only available through the core, each Component has to know of which visualization it is part.
- *Unique id*: For identification purposes that enable fast and unambiguous access, each Component is assigned an unique id by the visualization control.
- *Base width and base height*: This is the size of the Component in pixels. It will not be changed directly but scaled with a special radius for each axis. This enables resizing via transformation methods but still allows access to pixel values without recalculation. Working on specific and constant base values also provides some kind of protection against numerical errors that might after numerous changes.
- *Position*: The position determines where the center of the object is (with regard to the base sizes mentioned above).
- *Rotate'N Translate (RNT) capability*: General powerful interaction concepts such as RNT—which was introduced in Section 2.2.1—are not implemented within single Components but are triggered via an attribute and handled by one central implementation in the framework core.
- *Children/parent information*: If a Component is a Composite, it has knowledge about all its children by storing their ids. Regardless of the type of Component, a reference to the parent is stored, if existing. Thus, a Component can only be child of one parent at the moment.

DEFAULT INTERFACE: This constitutes a selection of important functionality defined at the abstract base of the class hierarchy. Thus, it is available by default to all derived Components.

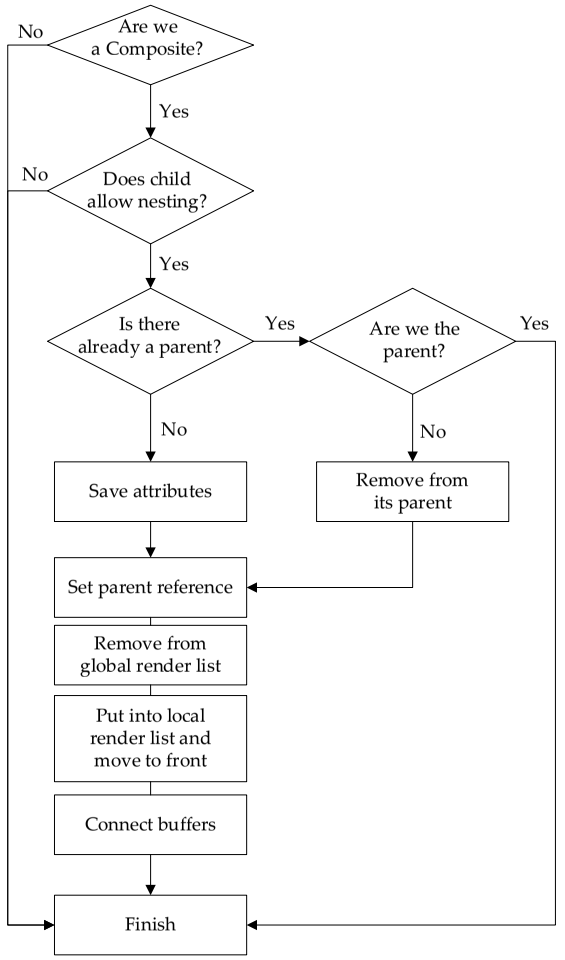
- *Buffer stacks*: Access to active and passive buffers—as introduced in Section 3.2.2—for manipulation and retrieval is provided. For identification purposes of the different buffers, tags are assigned. These are also the foundation of the generic connection scheme



(a) Procedure how to handle interaction with a single component.



(b) Procedure how to handle a dropped component.



(c) Procedure how to add a child to a Composite.

Figure 36. Flowcharts of important recurring actions within the framework.

that was described earlier. Although the developer is free to choose whatever buffer tag she likes, it is highly recommended to use a consistent set of tags that denotes both function and type of the buffer.

- *Coordinate transformations*: To achieve the switches between the different types of coordinates that where introduced in Figure 28, respective functionality is provided.
- *Children administration*: The rather complex task of adding a child to or removing it from a Composite is completely automatized. Thus, the developer has not to take care of this herself and can use the available methods. Figure 36(c) shows how this process works in detail and what steps are taken.

ABSTRACT INTERFACE: The abstract part of the interface enables the developer to create her own customized Components. Depending on what kind of Component is developed (Composite or Leaf), different sets of methods can be defined. The following gives an outline of the possibilities.

- The most important part of the abstract interface is the one that essentially puts the smarts into a Component and which is, therefore, used to animate it. This method does the buffer lookup in the linked passive buffers and processes the retrieved values to guide its behavior. By defining it, the developer shapes the user interface.
- To differentiate easily between Composites and Leaves at runtime, a special method is used. By default, this method returns nothing, making everything a Leaf to the outside. Composites redefine this method to return themselves and to indicate that they are a Composite. This is particularly useful to make safety checks, e. g., before adding a child, so a Leaf is not able to call this method. It is also useful to include additional functionality into methods that is only executed if the calling object is a Composite.
- *Active buffer setup*: The active buffers are a little more complex to setup than the passive ones because they are not only links but contain important data. Actually, these methods require a different way of thinking and programming because steering information is provided in a localized and discretized way. This is rather unusual compared to the "common", mostly analytical and global approach. Handling a Component's active buffers is twofold:
  - *Creation*: Developers have to build the needed buffers via the core and assign tags to them for lookup and connection.
  - *Content*: The most sophisticated part of a Component with active buffers is their content, the steering information. This means to fill the active buffers with actual values that can be used by connected Components.

As mentioned in the description of the buffer concept, this functionality is used for more than initializations. It provides also the functionality to resize and recreate the local buffer stack dynamically at run-time.

- *Button buffer interaction*: A very powerful concept is the possibility of associating functionality with a local buffer. The idea behind this is to have a special local buffer: a "button buffer". This buffer serves as an active buffer that stores button values at certain positions of a Component. Upon interaction, the value of the touched location can be retrieved and functionality associated with this value (and, therefore, also this location) is executed. Examples for this would the possibility to destroy the Component, to dynamically resize it, or to trigger the creation of new Components. For a generic handling of this interaction capability of the framework, two abstract methods are declared that can be defined by developers to give their Components the desired functionality.
  - To achieve this, a special method is called each time a Component is touched. Redefining it might evaluate the buffer value of the local touch position, execute certain code, or to store the value for processing in the following method. If the method returns nothing, interaction proceeds as usual, otherwise it stops after executing the button code.
  - Another special method is called after the user releases a Component. This is useful in combination with the previous method to execute special code just after the user finished interaction. About the return value the same as above applies.

  These methods are directly integrated into the two main interaction methods of the core, which are described in the following section on the framework core module. Figures 36(a) and (b) show where and how the methods above influence interaction with a Component.

RENDERER-SPECIFIC INTERFACE: The preceding functionality is independent from the Component's shape and geometry. To actually render its visual appearance, a specific rendering API has to be used. This part of the interface provides the developer with certain methods that can be defined using the special rendering API. Examples for this would be the geometry setup or the local render functionality of a Component.

The Buffer Module

This module is completely independent from any rendering API and can be used right away.

GENERAL: Important attributes are the following ones.

- *Data type*: This determines of what kind the buffer contents are. At the moment, this is the same for all cells of the buffer.

- *Number of channels*: This specifies how many layers of cells there are (cf. Figure 21 on page 30).
- *Width and height*: The number of rows and columns the buffer has, which has the greatest influence on how much memory will be needed for the buffer.

DEFAULT INTERFACE: The buffer interface is quite simple as there usually are not many different operations on a buffer.

- *Direct access* to a discrete buffer cell is provided, returning the value stored at that position. Direct access can also be used to set a value at a certain position.
- *Interpolated access* to buffer data can be very useful for certain applications. For this, values are weighed as described on page 39.

These methods exist in different variants, as there might be multiple channels defined which a developer then also wants to access. Buffer creation for special types is not done via the buffer interface but the framework core is used for simplicity reasons.

The Framework Core Module

The framework core provides a number of important methods as it is the main communication interface with the outside, such as the application and the Builder pattern.

GENERAL: The following attributes are administered by the core.

- *Global buffer stack*: The global buffer stack is created and stored in the core. All access to it can only be done via special methods that can vary for each buffer in the stack.
- *Ids of all existing Components*: As the core creates and destroys every Component of the visualization, it stores all their ids for fast access.
- *Multiple-user data during interaction*: While users interact with the application, the core stores certain information for each user, such as the currently active Component, device positions etc.

DEFAULT INTERFACE: Most important methods of the core are independent from the rendering API and can, therefore, be provided right at the base.

- *Methods to create every available buffer type* are provided. Via the core reference of each Component they can access these methods to create their own local buffer stack.
- *Device input*: To feed input data into the framework from the application, so far three methods can be used: pressing, releasing, and moving an input device. By providing the id of the interaction user, this data can be associated with the internal attributes and further processed.

- *Methods for the actual interaction processing*: This can be done in two steps if needed by the application. First, if the input device is pressed or the table surface touched, the Component under the touch position is processed. For this, a method that handles interaction with a single Component is provided, that does all necessary processing, if a Component was picked. This includes RNT capabilities as well as checking if a local buffer value requires any special code to be executed (see above in the Component section). Figure 36(a) gives an overview of what happens during this process.

  Second, certain interactions may require special actions if a Component is released. For this purpose, a method that handles a released Component is provided which processes the location *where* a Component is dropped. This might be to place a Component in a Storage Bin, for example, or to notice that a release happened over a certain local buffer value as mentioned before. Figure 36(b) shows what the steps of this process are.

RENDERER-SPECIFIC INTERFACE: For a specific rendering API, a set of methods has to be defined to provide the required functionality.

- *Renderer setup*: For an optimized and specialized setup of the renderer, a special initialization method can be defined. To allow for more flexibility at the creation time of the visualization, several flags are provided which can be evaluated in this method. Examples for such flags would be settings for texture compression, mipmapping, linear filtering etc.

- To load textures that will be used for decoration or to provide information, the loading capabilities of the renderer have to be made available to the framework. This might also make the definition of other methods necessary, methods to determine a certain texture format and texture parameters of the underlying hardware.

- *Component creation* is also renderer-dependent, as was explained before. Thus, all methods to actually create specific Components have to be provided, e. g., for the Builder pattern.

Visualization Creation via the Builder Pattern

Setting up the visualization with all needed content is a rather complex compilation of many different objects. To simplify and to encapsulate this process for better reusability, the Builder design pattern is employed. As described in Section 2.4.2, the Builder pattern usually consists of three to four parts. The desired product is the visualization itself, represented by the framework core. It is built and assembled by the *Builder* or its specific implementation, respectively.

Basically, the Builder provides an interface to build separate parts of the visualization. In the simplest case, this would be just an image Component or a single Interface Current. The true power of this design pattern lies beyond these simple tasks. To easily create more complex visualization content, the Builder provides functionality to build

combinations of Components, e. g., an Interface Current containing a certain number of images. Other complex tasks include the automatized loading of a number of textures from a specified location and other Component combinations.

So far, the buffer framework provides useful methods for the creation of common content, such as building image Components, Storage Bins, Interface Currents, and various combinations of these. Having easy creation methods for complex content is only the first step, as it is only one part of a complete visualization. The *Build Director* takes this idea one step further and provides an interface to different sets of calls to the Builder interface. Thus, it allows for creating predefined visualization content with a single method call. This is especially useful if there are recurrent setups of content or if a change between different contents is needed at run-time. The Build Director is usually part of the application or inherited into it. There, it allows to change the whole visualization and all of its content with a single method whenever required. It also supports the easy setup of visualization content at compile time by just changing the sequence of calls to the Builder interface.

To easily use new Components with the framework, new Components should be accessible via the Builder interface and added to it. Figure 37 shows a typical process how the visualization setup is chained through the different layers of the Builder pattern and ends up in the framework core.

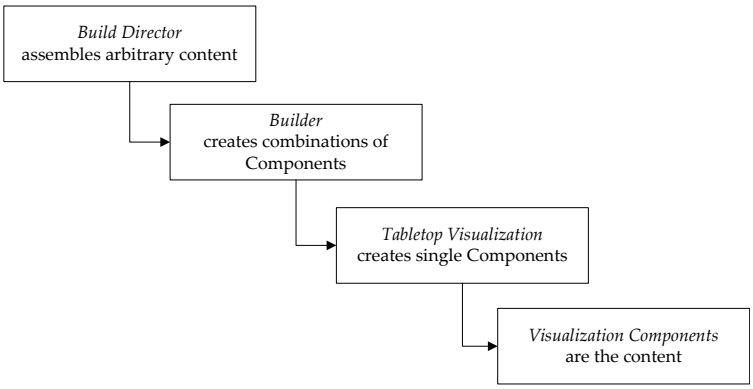


Figure 37. Chain of command in the Builder design pattern.

This section showed that the buffer framework is based on a hierarchical class structure with different layers. Abstract base classes provide general solutions to common problems within the framework. They also provide developers with clearly defined interfaces to realize new Components. As support from rendering APIs is necessary for visualization purposes, well defined parts of the base classes have to be implemented for a specific rendering API.

To simplify the code for handling visualization content, the Composite pattern is utilized. This makes it possible to build any content on the same base class but at the same time to easily differentiate between simple and complex content.

For the setup of the visualization the Builder pattern is used to pro-

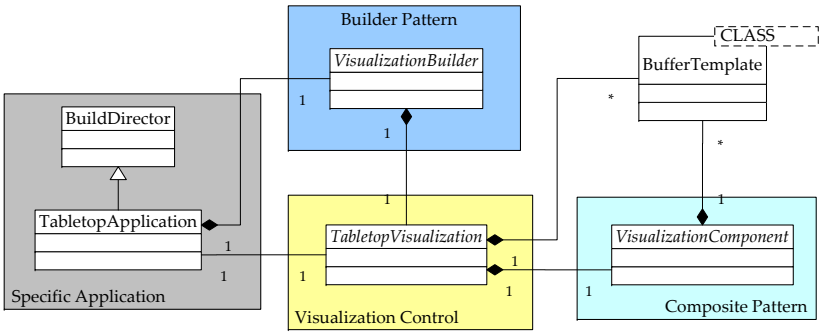


Figure 38. Schematic UML diagram of the framework setup.

vide methods for content creation in different types of granularity.

The final layout of the framework and how it links to an application is shown in Figure 38. For simplicity reasons, this figure neglects the representation of the different layers and subclasses.

After this high-level overview, Chapter 4 shows selected details of the implementation and how these affect the performance of applications built with the buffer framework.

3.4 CHAPTER SUMMARY

In this chapter it was shown that the design and implementation of a buffer framework addresses two important issues in the research of large displays. The rest of the chapter was divided into the two parts of the solution to these issues.

First, a buffer concept for interaction was developed which transfers successful ideas from computer graphics and selected aspects from swarm intelligence to the area of interaction. The main ideas of this concept were local awareness, pre-computed and discretized availability of information, and local processing of this information. Further development of this concept made the introduction of global and local buffers stacks necessary and lead to the idea of active and passive buffers to enable a generic connection scheme for buffers.

Second, a framework architecture was developed that incorporated the buffer concept for significant performance gains and which addressed the general software issues modularity, extensibility, and reusability. To achieve these challenging goals, techniques from object-oriented analysis and design such as decoupling, centralized knowledge, and operation-centric view were used. Another important foundation of the framework that contributed to these goals were design patterns, namely, the Composite and the Builder pattern.

Combined, these two parts provide a powerful key to an easier development process of responsive applications for high resolution displays.

4 IMPLEMENTATION

After the abstract and general descriptions of the buffer concept and the framework architecture in the preceding chapter, this chapter shows and discusses selected details of the implementation. Used technology as well as utilized techniques and important code subtleties are explained to close the gap between the abstract and the specific.

4.1 REALIZATION OF THE FRAMEWORK ARCHITECTURE

This section discusses the technology and procedures that were used to build the buffer framework.

4.1.1 *Technology*

The goal for the implementation of the architecture devised in Chapter 3 and its underlying buffer concept was to go beyond a mere proof-of-concept. Specifically, it is designed to provide researchers with a reusable and extensible software architecture to explore new ways of interaction on large displays easier. Therefore, the technology to implement and to realize this goal had to be chosen carefully. The high performance requirements led to choosing C++ as the programming language, but alternatives such as C# or Java were considered as well. Although the latter provide an automatic memory management and are, therefore, generally easier to use, current versions still lack the high performance capabilities and the flexibility of well-used C++.[1]

For the internal organization of the framework data structures from the Standard Template Library (STL) are used. These offer advantages as being widely available and being subject to strict complexity specifications.

The abstract base layer of the buffer framework (cf. Figure 34) can be and was, therefore, implemented in a general, operating system-independent way. For the implementation of renderer-specific parts the *OpenGL* API was used. It is freely available, widely accepted, well understood and offers high performance.

Due to the used display hardware, the available drivers, and toolkits such as the SMART SDK, Microsoft Windows with Visual Studio .NET 2003 was used as the development environment. To achieve the setup explained in Section 3.3.1, the whole framework is encapsulated into a Dynamic Link Library (DLL) and this DLL can be used to access the framework functionality. Section 4.2.3 shows a simple and specific example in C++ that makes use of the provided functionality.

Figure 39 shows an overview of the different technologies used to build the buffer framework and how it fits together with other technology used to build tabletop applications. On the left are the different

1 A detailed comparison of modern programming languages is far beyond the scope of this thesis. The rough discussion given here is used to give reasons for the choice made and to point out advantages of this choice.

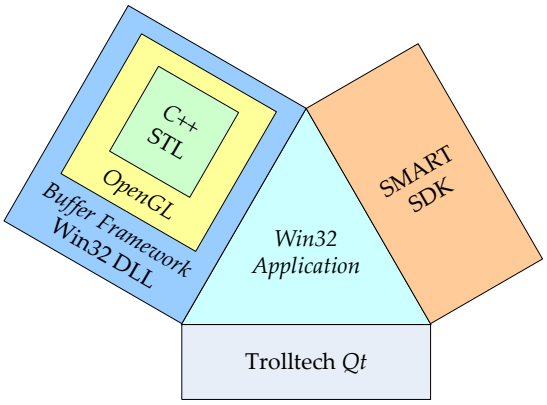


Figure 39. Technology overview for a tabletop application.

technology layers of the buffer framework, ranging from standard C++ to a Microsoft Windows DLL. The right side and the bottom show other toolkits that provide special functionality for an application: there is the SMART SDK for the two-touch input and Trolltech's Qt library to build the application on. All together works seamlessly as an application for tabletop displays.

4.1.2 *Development Approach*

Designing and implementing reusable and extensible software in a very limited time frame is a difficult task that requires sophisticated development techniques.

The estimating and planning process was done according to Thomas and Hunt [60, pages 64–70] and is also reflected by the thesis' organization:

1. Understand what is asked (Chapter 2).
2. Build the model and break the model into pieces (Chapter 3).

→ From this followed the work estimation for the implementation. The schedule was iterated with the progress of the code.

The implementation followed an incremental development scheme according to Agile Development paradigms [1], or as this is called by Thomas and Hunt [60, pages 48–52]: *Tracer Bullets*. The underlying idea is that a project is never finished, functionality might have to be added, and requirements change often during the implementation. This approach fits for this thesis, as the work on the buffer framework is not finished with the thesis but just started.

To achieve this incremental approach, a basic structure—the main layout of the architecture—was designed and built to work in. This code skeleton was then fleshed out more and more as functionality was added. Essentially, this yields two major advantages:

1. There is always an integration platform (the main architecture) to integrate new ideas, to experiment with new features, and to test the effects of new code.
2. There is running code at all stages of the project.

In addition, this gives the developer a better feeling for the progress of the project.

It is very important to distinguish the Tracer Bullets approach from developing a prototype. While a Tracer Bullet is designed to grow to a full product during the development process, prototypes are intended to be thrown away. Therefore, prototypes usually ignore important aspects such as

- correctness,
- completeness,
- robustness, and
- style

which are essential for good software [60, pages 53–56].

The utilized techniques enabled a development process that was flexible enough to adapt to new requirements and to solve unforeseen problems. Having running software at all stages also for demonstration purposes proved very useful to discuss the state of the project and to adapt the schedule. Therefore, Tracer Bullets and Agile methods can be seen as a major contribution to the implementation of the buffer framework.

4.2 RENDERING SPECIALTIES

As pointed out before, applications for large displays deal mainly with visualizing data and interfaces. Thus, rendering takes a major role in the implementation of systems for large displays.

This section discusses a selection of special problems that arose during the implementation process and their solutions. First, the interrelation between rendering and responsive interaction is shown and a solution to resulting problems is presented. Second, optimizations to the render loop[2] of an application using the buffer framework show the specific impact of well-written code on the performance of an application. The section concludes with the effect of certain data structures on the internal organization of the framework which is an important factor for the framework's performance.

4.2.1 *The Picking Problem*

The buffers of the buffer framework are stored in RAM and calculations and operations on them are done on the CPU. This is due to the possibly very high memory requirements and the rather simple operations needed for interaction tasks with the buffer framework, namely, retrieval and writing of data from or to buffer cells.

So far, this works fine for the needs of a Component's local buffer stack. For other requirements, such as the picking[3] of Components by the user, the following would be necessary:

2 A "render loop" is to be understood as that method of a program which is called for each frame. It is typically used to draw the visual content on the screen.

3 In this thesis, the term "picking" refers to the process of finding out which Component is under a screen location touched by a user.

1. A global buffer to store the ids of Components in their covered areas is required. User interaction with a certain location would then be forwarded to this id buffer, the stored id (or none, if no Component is present there) would be retrieved, and interaction with the associated Component would be processed.

2. Every Component would need a special render method to transform its geometric representation into a rasterized id buffer representation, as users usually expect to interact with what they see, at least for the picking process. Figure 40 shows a schematic view of the different stages of this process and how the different representations could look like.

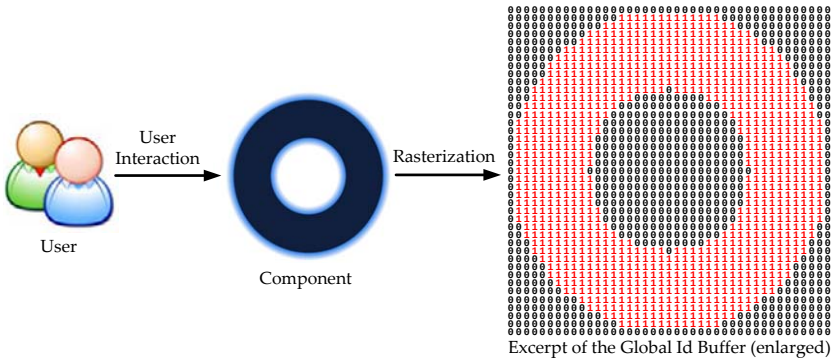


Figure 40. The different steps of the picking process done via a global id buffer.

The most critical factors of this process are

- the size of the geometry,
- the complexity of the geometry, and
- the size of the global id buffer.

Thus, it is even for a rather simple Component, e. g., like the simple Interface Current depicted in Figure 40, very costly to calculate and write the required values. Definitely, it cannot be done every frame as the computational costs are too high. There are even additional complexities such as removing the previous footprint of a Component from the id buffer and synchronizing this removal with other users' interactions. Test implementations revealed that the costs are even too high if the rendering into the id buffer is only done when user interaction with a Component is finished, i. e., when a Component is dropped.

This shows that the implementation of the buffer concept using RAM and doing operations on the CPU is suitable for its core tasks for which it was designed. This technology is the right choice if only small portions of the buffers are changed at a time, if pre-computed data can be used, or if only simple operations are required. Responsive interaction cannot be maintained by this approach for intense and complex calculations which affect large areas of one or even more buffers, e. g., as needed for the picking of Components.

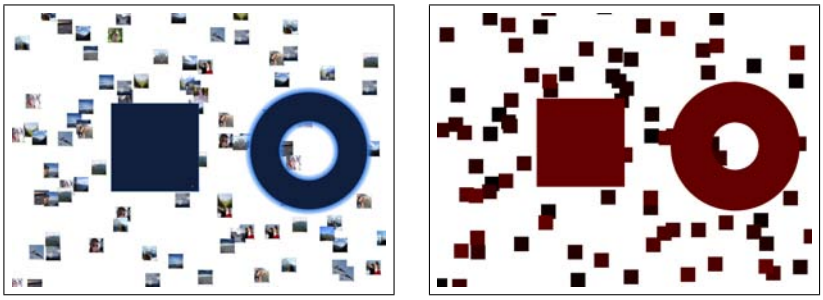
The next section shows a solution to this special problem with the help of hardware.

4.2.2 *A Framebuffer Solution*

As was shown in the preceding section, special cases of the buffer concept cannot be realized efficiently in software. To solve this problem and to implement an efficient picking mechanism for Components, picking is done by grabbing colors directly from the framebuffer. The details of this approach are as follows:

1. Every Component has a unique color attribute which is calculated from its unique id.
2. This color is not visible to the user as it is hidden under textures or decoration colors.
3. Rendering for picking is done separately in a first render pass and it puts the colors into the backbuffer of the framebuffer at their respective positions. During interaction, the color under the touched location is retrieved from the backbuffer, the Component's id is calculated from the color, and interaction with this Component can be processed. After this, the rendering for the visualization is done, the buffer-flip of the double buffering happens and, therefore, the user does not see the extra rendering step for the color picking.

Figure 41 shows an example for the different results of the two rendering types.



(a) Regular rendering with textures and decorative colors.

(b) Rendering for picking with an unique color for each Component. (Neighboring colors are hard to distinguish by the human eye.)

Figure 41. Examples for the two different rendering types.

The biggest disadvantage of requiring an unique color is the limitation to the number of available different colors. These are for the RGB space

$$256^3 = 2^{24} = 16,777,216 \text{ different colors.}$$

While this is a theoretical limit, it is not a limitation that might be encountered in practical applications soon (cf. Section 5.2.1).

However, there are other picking schemes[4] available and possible, e. g., the name stack in *OpenGL* [51, Chapter 13]. But color picking is a good choice to keep the interfaces of the buffer framework renderer-independent.

To sum this up, the color picking technique overcomes the performance problem of frequently rendering into global buffers by grabbing relevant information directly from the framebuffer. It couples id information with color information via simple calculations and, thus, provides a fast and easy way to identify a Component, which is an important feature for realizing responsive interaction.

4.2.3 *The Render Loop*

The buffer framework is no application by itself but it provides an interface to render and to process visualization and interaction content.

To achieve the color picking scheme presented in the preceding section, the rendering process of an application using the buffer framework has to be adapted. This is done by calling special framework methods at the right time, as rendering too much and too often will reduce the application's performance drastically. Unfortunately, this cannot be done automatically by the framework, as the developer needs to maintain full control over her render loop for a maximum of flexibility. However, the framework provides all necessary information and simple methods to implement a highly optimized render loop for an application.

The most important and framework-relevant parts of an example render loop are schematically shown in Figure 42. In addition, Listing 1 shows C++ code from the render loop of the example application that was developed to test and demonstrate the buffer framework.

The information the framework provides is essential to reduce the number of times very costly methods are called. As both picking and dropping a Component require an additional rendering step to get the needed color information from the framebuffer, this additional rendering should only be done if there is actually picking or dropping in progress. Therefore, important information whether picking is occurring or not is provided by the method `getIsPicking(userId)`, which is called in line 20 of Listing 1 for example. A Boolean `true` is only returned if the user touches something. Any following interaction such as dragging the already picked Component does not trigger the extra rendering and other expensive operations such as `glReadPixels()`.

It is important to note that a user is either picking *or* dropping a Component in the same iteration of the render loop, she cannot do both at the same time. Thus, the extra rendering for the color picking is done once per iteration at most. However, input might become a critical factor if many users are picking concurrently with multiple input devices per user, as this requires many iterations of the `for` loop (cf. line 7) or even nested `for` loops. At the moment, this is not a practical limitation due to the available hard- and software. How this affects both performance and the general interaction setup around a tabletop display might be an interesting research question for the near as well as the far future.

4 It has to be noted that this color picking scheme was not invented in this thesis but it is a commonly used technique in graphics programming [51, page 602].

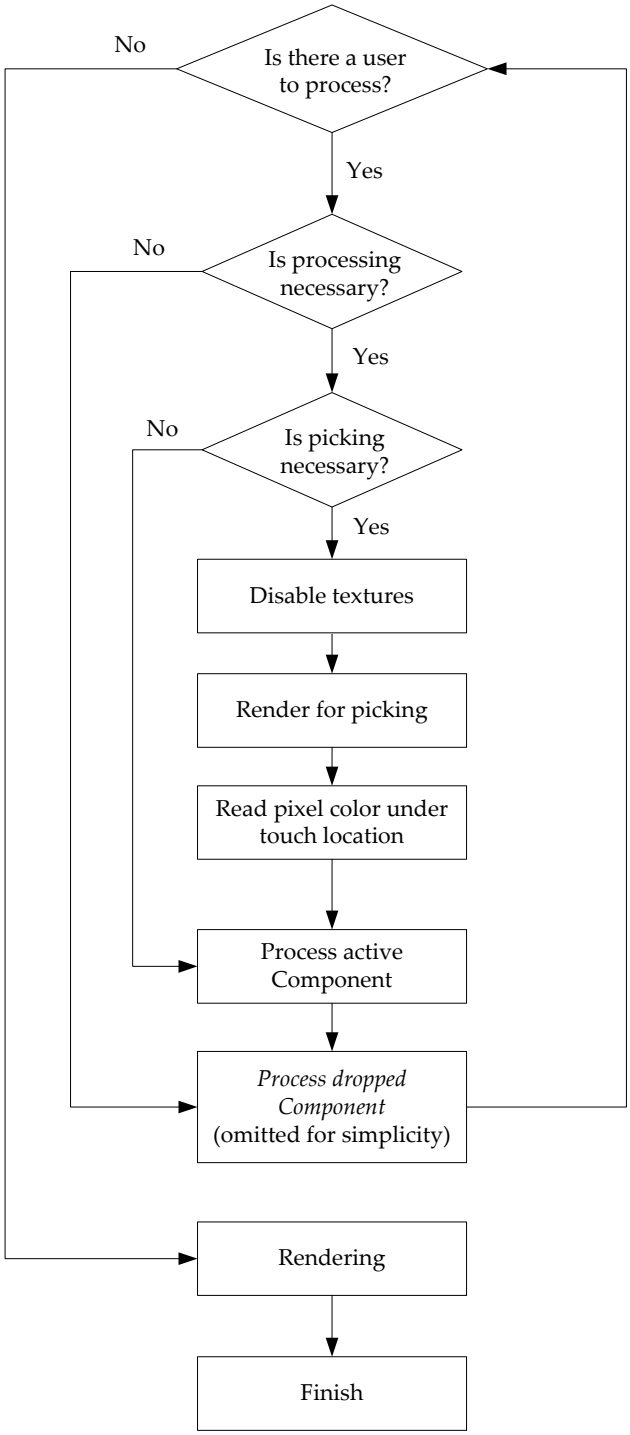


Figure 42. Flowchart of important parts of the render loop.

Listing 1. Important parts of the render loop

```
void QTTabletopWidgetGL::paintGL()
{

  // ...

  // iterate over all users
  for (unsigned int i = 0; i < NUMBER_OF_USERS; ++i)
  {

    // ...

    GLubyte pixelColor[3] = {0,0,0};

    // Picking
    // do we really need to draw?
    if (getVis()->checkDoDrawingForUser(i))
    {
      // this is important to do the expensive
      // operations not too often
      if (getVis()->getIsPicking(i))
      {
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        // render without the textures and only where the input is
        glDisable(GL_TEXTURE_2D);
        // important so we won't see the other Components
        // moving without textures
        glScissor(getVis()->getPosXToPaint(i),
                  getVis()->getPosYToPaint(i), 1, 1);
        glEnable(GL_SCISSOR_TEST);
        // render objects for picking
        getVis()->renderAllComponentsForPicking();
        // finishing the rendering first is important on quite fast
        // machines, otherwise we read wrong data from the input
        glFinish();
        // find pixel under mouse pointer
        glReadPixels(getVis()->getPosXToPaint(i),
                     getVis()->getPosYToPaint(i),
                     1, 1, GL_RGB, GL_UNSIGNED_BYTE, pixelColor);
        glPopAttrib();
      }
      // cf. Figure 36(a) on page 48
      getVis()->processSingleComponent(i, pixelColor);
    } // end of picking

    // Dropping part here (omitted for simplicity)

  } // end of user iteration

  // do the real rendering (not for picking etc.)  here

  // ...

} // end of paintGL()
```

4.2.4 *Internal Organization*

To achieve responsive interaction and efficient rendering as discussed before, the data structures to organize the information within the framework have to be chosen carefully. This section briefly discusses the tasks information has to be organized for and why certain data structures are used in the framework implementation.

COMPONENT STORAGE: The framework core has the single, unambiguous, and authoritative representation of all visualization content, the Components. Every Component is identified by its unique id, which suggests an organization of all Components in an array or vector due to the cheap $O(1)$ costs.[5] However, as Components can be deleted by user interaction and the ids of deleted Components can be reused by new Components, the difficult and expensive remove operations on arrays or vectors make these data structures not suitable for this task. This leads to using a STL map, where values can be accessed by unique keys, here the unique Component id. Due to the internal tree representation of the data, the crucial operations insertion, lookup, and removal can all be done in $O(\log n)$ time.

RENDERING ORDER: Although the order of the Components does not matter for the internal representation, it is very important for the rendering process. Components have to be displayed according to user interaction, e. g., the currently active Component should be rendered over the other ones for intuitive interaction. The STL map used for the Component storage orders the stored data according to the keys. Thus, because of the decoupling of the storage from the rendering order, a separate data structure is used. The required data structure for this should provide easy access to its front or end and removing a Component anywhere should be possible as well. This leads to using a linked list with expensive Component removal in $O(n)$ time but very fast insertions in $O(1)$ due to the use of a special case: we insert always at the front of the render list. Rendering works then by iterating through the list from back to front and rendering each Component. Whenever interaction with a Component occurs, it is removed from the render list and put at its front to be rendered on top of all the other Components. This enables a local optimization for the lookup as recently touched Components are rather close to the head of the list, not requiring a full iteration through the whole list. Thus, the $O(n)$ worst case is only likely for seldom touched Components or if the number of users is close to the number of Components.

We have seen that the internal storage of all Components is a different concept than their order for rendering. Thus, different data structures have to be utilized to take advantage of the respective requirements. In this context, optimizations for special cases help to reduce the native disadvantages of certain data structures such as lists and, therefore, contribute to improving the overall performance of the buffer framework and applications built with it.

5 O-notation provides information about an *asymptotic upper bound* and is used in this thesis according to Knuth as well as Cormen et al. [10, 25]: $O(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants c and n_0 with $0 \leqslant |g(n)| \leqslant cf(n)$ for all $n \geqslant n_0$. For example, an algorithm with a running time of $O(n)$ on a number of inputs n has a worst case running time of order n. However, no information about the *lower* bound is implied.

4.3 CHAPTER SUMMARY

In this chapter, technology and techniques to implement the concept devised in Chapter 3 were presented. It was shown how these help to achieve framework design goals such as reusability and extensibility. The utilization of an incremental development process supported and emphasized the underlying framework concept, namely, that the buffer framework is not finished with this thesis but just started.

Complex calculations that affect large areas of one or even more buffers were identified as a bottleneck for responsive interaction, in particular for the picking of Components. A solution to this particular problem with the help of graphics hardware was explained in detail and implemented.

The chapter concluded with the detailed description of the framework's capabilities to optimize an application's render loop and with a critical view at the data structures used for the internal organization of the framework core.

5 EVALUATION

This chapter contains a critical evaluation of what has been achieved in this thesis. First, the process that was used to reach the goals of the thesis is examined and evaluated. Second, the devised results are presented and compared to other significant work in this area of research. This includes mainly performance benchmarking to measure the responsive interaction capabilities of the implemented concept but also the quality of the software engineering regarding reusability and extensibility.

5.1 WORK PROCESS

The greatest challenge of this thesis was that there are not many publications on the topic of responsive interaction on large displays. This is mostly due to technological reasons, as the size of displays has been increasing faster than their resolution.

The start of this thesis' research was, therefore, a close look at the available prototypes for interaction metaphors on different tables and resolutions. Software that is running smoothly on lower resolution tables severely lacks responsive interaction when it is run on the table prototype at the University of Calgary which resolution is one of the highest in the world at the moment. The investigation of the different parts of these applications led to knowledge about their typical bottlenecks. Intensive studies of the current tabletop research as well as of other fields with similar problems led to the discovery of successful and already well-understood techniques. These techniques were adapted and further elaborated to suit the needs of interaction metaphors.

On the other hand, the study of existing prototypes and publications about them surfaced yet another seldom researched area within the tabletop community. This led to the idea of designing modular support for building applications, a foundation which is taken for granted in many other fields. Due to their completely different interaction requirements—which were shown by researchers over years—the software for tabletop displays has to be different from that for regular desktop environments, featuring different interaction metaphors. To achieve this goal, best practices from a variety of fields, especially software engineering, were gathered and adapted to the requirements of of high resolution displays.

The next logical conclusion was to combine these two rather separate concepts. This can be considered the major research contribution of this thesis, as it makes higher performance easier accessible by integrating it into an optimized development process. Agile methods were used to realize this combination—this served the purpose of demonstrating the power and versatility of the devised concepts on existing large displays and to make it publicly available to the re-

search community later on. As frameworks cannot be presented and tested by themselves, an example application was built on top of the framework. This example transferred several important table interaction metaphors from their prototypes into the buffer framework and showed how they can be realized efficiently. The results of these transfers and comparisons between the framework and an important prototype are presented and discussed in the next section.

To sum up the general process of this thesis, research and ideas were always oriented towards the urgent needs of the research area and how these could be satisfied using best practices and successful solutions from other fields.

5.2 RESULTS

As the concept was divided into two distinct parts—the buffer concept for performance gain and the framework architecture to support the application development process—the evaluation of the results has to reflect this division as well. Thus, the performance of the buffer concept is compared to that of a previous approach and the extension and reusability capabilities of the framework are critically evaluated.

5.2.1 *Performance*

To measure the performance of a typical application built with the buffer framework, test series were conducted. As responsive interaction is one of the main goals of the framework, *frames per second* (fps) were chosen as the metric to measure performance. This is due to the fact that the quality of responsive interaction depends on how fast visualization content can be displayed on the screen.

For orientation and quality purposes the measurements are compared to those that can be achieved with a common, but complex tabletop application prototype. The demo application for Interface Currents [20, 21] was chosen for this comparison, as this application uses comparable types of interface components and visualization objects.

The factors that influence the performance of the applications are the following:

- *Resolution of the display*: The tests are conducted on two different high resolution tabletop displays with 2,048 by 1,280 and 2,800 by 2,100 pixels, respectively. Due to hardware issues the latter table could not be used at its full resolution with hardware acceleration. Therefore, the next available lower resolution was used for these tests, which is 2,560 by 2,048.

- *Used texture memory*: The visualization objects are textured with images. For the test series the used texture memory is about equal for both applications and depends on the number of objects. The total amount of used memory varies between 10 and about 100 megabytes of uncompressed textures. For further comparison, the size of the used visualization objects is about the same with a side length between 80 and 120 pixels.

- *Number of visualized objects*: The most critical factor for responsive interaction is how many objects have to be displayed and animated on the display. This factor is varied to see its influence on the performance.

Concerning the sampling procedure of the framerate, the benchmarking function of the freely available tool FRAPS [56] is used. For a given time frame—here 5 minutes—the tool records the measured frame rates per second and calculates the average value. As we deal here with constant outputs and not random variables, techniques known from simulation such as confidence intervals are not required.

The hardware setup for the machines that run the tables is as follows:

- *2,048 by 1,280 table*: Running Windows XP with 2 gigabytes RAM and an Intel Xeon 2.8 gigahertz CPU. NVIDIA Quadro4 700GL graphics hardware with VSYNC[1] turned off.
- *2,560 by 2,048 table*: Running Windows XP with 512 megabytes RAM and an Intel Xeon 1.4 gigahertz CPU. Matrox QID Pro graphics hardware with VSYNC enabled, as it could not be disabled due to driver issues.[2]

The test itself is split into two parts. One varies the number of visualized objects and compares how these changes influence the framerate on the different display resolutions. The second part aims at keeping the framerate constant and comparing the different numbers of visualized objects the two applications can handle on the given resolutions.

VARIABLE FRAMERATE: This test series aims at comparing the performance of the two applications when the number of visualized objects increases. About 20 frames per second can be seen as the lower limit for responsive interaction in this context. A benchmark is the number of objects that makes each application lose its capability of responsive interaction. It is also important to see how the different resolutions affect the performance. Table 2 and Figure 43 show the results for this series.

FIXED FRAMERATE: For a different kind of comparison, this test series uses a base framerate, which is the framerate the common application with the Interface Currents can achieve with 100 or 25 objects, respectively. For the buffer framework application, the number of objects is increased until its framerate reaches the base framerate. Table 3 shows the results for this series.

1 VSYNC is a setting for the graphics hardware that limits framebuffer swaps to the display's refresh rate. This results in an artificial upper limit to the framerate and is usually used to avoid visual artifacts.

2 The latest certified driver from Matrox with support for quad stretched (1.10.01.002 SE) did not work for both applications. An older, unified driver version (2.01.00.081 SE) enabled the setup that is described above.

RESOLUTION	OBJECTS	CURRENTS	FRAMEWORK
2,048 by 1,280	50	38.470 fps	158.933 fps
2,048 by 1,280	100	23.913 fps	134.436 fps
2,048 by 1,280	200	12.433 fps	104.996 fps
2,048 by 1,280	300	8.480 fps	85.676 fps
2,048 by 1,280	400	6.673 fps	71.610 fps
2,048 by 1,280	500	5.460 fps	60.603 fps
2,048 by 1,280	1,000	—	31.763 fps
2,560 by 2,048	50	13.273 fps	31.822 fps
2,560 by 2,048	100	7.489 fps	29.866 fps
2,560 by 2,048	200	4.104 fps	19.960 fps
2,560 by 2,048	300	2.856 fps	19.933 fps
2,560 by 2,048	400	—	14.990 fps
2,560 by 2,048	500	—	14.803 fps
2,560 by 2,048	1,000	—	8.540 fps

Table 2. Results for the test series with a variable framerate and an increasing number of objects.

RESOLUTION	FRAMERATE	CURRENTS	FRAMEWORK
2,048 by 1,280	≈24 fps	100 objects	1,400 objects
2,560 by 2,048	≈20 fps	25 objects	300 objects

Table 3. Results for the test series with a fixed base framerate and a variable number of objects.

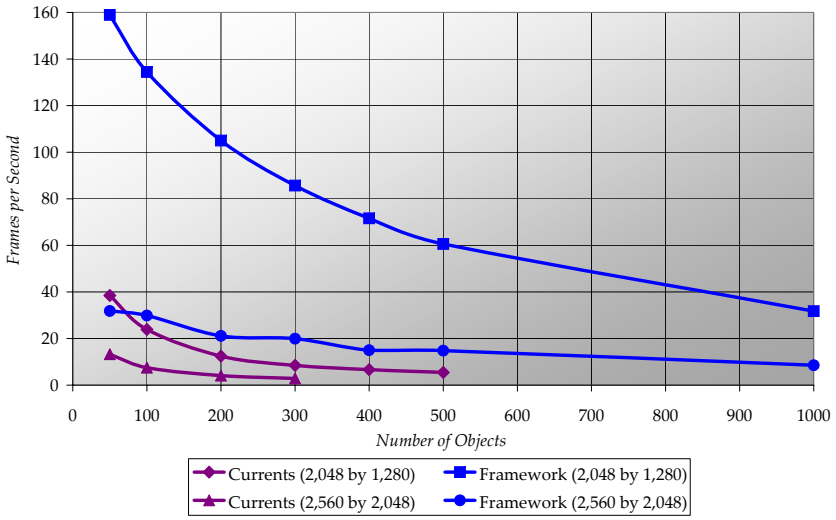


Figure 43. Graph for the results of the test series with a variable framerate and an increasing number of objects.

DISCUSSION: As the comparison shows, both display resolutions and the underlying hardware heavily affect the performance. It also shows that scaling an tabletop application means increasing the number of displayed objects, which in return decreases performance significantly.

But what the comparisons most clearly show is that the traditional application with the Interface Currents does not scale very well and adding objects is reflected immediately by the framerate. On the other hand, the application built with the buffer framework is rather resistant against additions of Components, starting at very high framerates which are declining slowly as the number of Components increases. As VSYNC could not be disabled for the test series on the higher resolution table, the artificially limited results might differ from the real values. This is indicated by the resulting stair shape of the •-plot in Figure 43. It is possible that at these values get somehow normalized to fractions of 60 Hz due to VSYNC. Therefore, at least the first value is very likely to be higher than the measured framerate. However, this does not affect the overall results of the tests.

> *Both test series show that the increase in objects while maintaining a responsive framerate is* about one order of magnitude higher *for the buffer framework.*

More and different tests showed that the rendering hardware is also a factor in these benchmarks. Adding little geometrical details to the scaled number of objects, such as an additional borderline around them, had a severe impact on the framerate of about the factor 2 to 3. But to clearly verify these highly complex interdependencies is beyond the scope of this thesis. However, it follows from these findings that keeping the hardware for high resolution displays up-to-date is important to maintain performance or to even get an additional increase.

To conclude this section it can be stated that the benchmarking tests conducted with the buffer framework revealed two important insights. First, the new and elaborated concept of using buffers for interaction together with a sophisticated software architecture resulted in a significant performance increase of about one order of magnitude. Second, using high performance graphics hardware addresses rendering-related issues that arise with a high number of objects and helps, therefore, to stabilize performance.

5.2.2 *Software Architecture*

There are many different factors to evaluate the quality of the developed software architecture. For this evaluation the benefits of object-oriented frameworks as outlined in Section 3.1 are used as a guideline and it is checked to what extend they were achieved.

MODULARITY: The buffer framework features a variety of different modules. These modules are decoupled from each other and provide solutions for different aspects of the visualization such as creation, organization, content, and access to buffered information. In addition, these modules' base functionality is independent from any rendering API and, thus, not limited to a particular one.

REUSABILITY: Having different specific layers within the architecture enables both code and design reuse on different levels. Core functionality can be reused to support other rendering APIs, whereas specific implementations can be reused to realize new interaction metaphors. Both types were shown in this thesis by providing implementations for *OpenGL* and various interaction metaphors. To fully access these possibilities, a thorough understanding of the buffer framework, its API, and the underlying design patterns is necessary. However, this can be seen as a minor drawback as these require time and effort to be learned.

EXTENSIBILITY: Apart from the already realized extensions, there are several other in development at the moment. Researchers are already extending the available interaction metaphors of the framework to make them more powerful and easier to configure (cf. Figure 44). Furthermore, there is a buffer-based Air Hockey game for tabletop displays in preparation as well as work towards using the buffer framework for non-photorealistic rendering applications.

DRAWBACKS: There are also drawbacks that arise with the architecture and with frameworks in general.

By definition, a framework provides a generic solution to a common set of problems via design reuse. This makes a framework solution always less versatile and configurable than a dedicated and specialized application that is targeted at one particular problem.

Another drawback lies in the buffers themselves at the moment. The idea to provide buffered information for local processing is fairly new in the field of interaction. Experiences with other developers so far

Figure 44. Interface Currents realized with the buffer framework: a simple one on the right and a more complex one on the left, featuring control points for splines.

have shown that the buffer concept requires some time to be grasped and to understand its power. Therefore, it is usually difficult for the buffer novice to come up with the right algorithms to fill the active buffers of a Component. These difficulties arise especially, if highly mathematical concepts, e. g., such as vector fields for the buffers of an Interface Current are required. Ideas to approach this problem are outlined in Section 6.2 as future work.

In total, the software architecture meets the set goals and provides the benefits frameworks are generally known for. Concerning the drawbacks, they are for the one part initial problems that are common for novel techniques and they will be eventually resolved with future refinements of the software. For the other part, the problems are known issues of generic approaches and constitute the price that has to be paid for design and code reuse.

6 CONCLUSION AND FUTURE WORK

This chapter sums up the results of the thesis and discusses different aspects of the work. Furthermore, it provides a concise statement about the meaning of both the results and the process that was used to achieve them. An outlook at future work provides several points to continue and to improve the work that was started with this thesis. The chapter and the thesis conclude with personal remarks by its author.

6.1 CONCLUSION

The goal of this thesis was to find a solution to the performance problems that arise on high resolution displays, especially with complex interfaces featuring many visual objects. To make this solution available to application developers and, furthermore, to support the process of building applications for large displays, a software framework had to be designed.

These goals were motivated by the growing interest in and the potential of large displays, especially tabletop displays. Such displays support co-located group collaboration much better than regular desktop computers. However, because of their characteristics, special interaction metaphors that address orientation, reach, territories etc. have to be provided for using such displays to their full potential. As the analysis part of the thesis showed, the way such interfaces are organized as well the huge amount of pixels seriously affect the performance of applications on large displays, thus, hindering responsive interaction. In addition, it was discovered that there is limited to no software support available to help developers build such specialized applications. This can be seen as a major obstacle to research as researchers have to spend much time on from-the-scratch programming which might later on be missing for their main work.

These findings led to the formal and rather abstract definition of the buffer concept. This concept is built on ideas borrowed from computer graphics and on selected aspects of swarm intelligence. It features three important advantages:

1. Information is pre-computed and discretized, thus, expensive calculations at run-time are avoided.
2. Local awareness of the objects on the screen gives them fast access to necessary information.
3. Local processing of the gathered information avoids expensive communication schemes and offers new ways of flexibility.

These features reduce the complexity of the interface's underlying structure. Looking at this concept closer made some refinements necessary. These refinements introduced important concepts such as local

buffer stacks, a generic connection scheme, and coordinate transformations which added to the advantages of the basic idea.

This complexity-reducing and, thus, performance-increasing concept was then integrated into the design of a framework architecture which would simplify the development process of tabletop applications: the *buffer framework*. The architecture is built on a modular design which enables the authoritative representation of knowledge at a single point within the system. Furthermore, the modules are decoupled from each other to clearly separate unrelated things and to minimize side-effects. To achieve important goals such as reusability and extensibility, object-oriented design patterns were employed. In addition, the whole architecture was divided into different abstraction layers to create a base for different rendering APIs while not being limited to a particular one.

An implementation of the abstract base and building on that, for the *OpenGL* rendering API, realized the two parts of the concept. This led to the discovery of important insights about using software or hardware for buffer operations. The implementation part was concluded with different optimization possibilities for the buffer framework and applications built with it.

The evaluation chapter had a critical look at three important aspects of this thesis:

1. *Work Process*: Despite the novelty of the researched issues, the whole process that led to the devised concepts and its results was at all stages aimed at using and adapting successful ideas from other fields to fulfill the requirements of tabletop research. The process in general can, therefore, be seen as very well suited to solve the discovered problems.

2. *Performance*: Important results about the actual performance of the implementation were gathered by carefully designed and conducted test series. Comparing these results to a current state-of-the-art prototype showed that applications built with the buffer framework can handle an increase of objects of about one order of magnitude and still maintain responsive interaction. Furthermore, important insights how graphics hardware influences this metric were extracted from the tests. In total, this makes the buffer framework a powerful tool to realize responsive interaction on high resolution displays, handling large amounts of objects.

3. *Software Architecture*: An investigation of the buffer framework regarding the usual framework benefits revealed that modularity, extensibility, and reusability were well considered in the design of the architecture and are based on the requirements of tabletop applications. Drawbacks related to frameworks in general and to the novelty of the buffer concept were discussed as well. However, in a rapidly evolving and highly dynamic field such as tabletop interaction, requirements are likely to change and may not be met by the buffer framework in the future. It is a serious start to improve performance and application development on high resolution displays, but future requirements could only be roughly estimated and considered in the design process.

To sum up the evaluation, the results meet and even exceed the goals that were identified during the analysis, despite some minor drawbacks.

It is also important to point out that the implementation of the buffer framework is not just a proof-of-concept or a prototype, it is more. It seriously aims at providing generic support for developing applications for high resolution displays. At all times, the whole design and development process was oriented towards one attribute of the result: being a well-designed foundation for future work. As the results and the extensions in progress show, the work is not finished with this thesis. It has just begun. The next section presents various ideas and possibilities to reuse the buffer framework's advantages and to take its concept further. This kind of solution—which lays the foundation for such a variety of future research—is the main contribution of this thesis.

6.2 FUTURE WORK

The buffer framework was designed and implemented with reusability and extensibility in mind, and Section 5.2.2 showed that this goal was achieved. As there are already extensions and reuses of the framework in development, this is the main area of future work for this thesis. A good idea is to extend the library of interaction metaphors and to make the existing ones richer and more comfortable to use. In this context, extensions to new research questions such as 2.5D or 3D for tabletop interaction metaphors like Interface Currents or RNT are likely to be investigated using the buffer framework in the near future. These would carry on research done by Hancock et al. [19].

To achieve a more comfortable use of the framework also for non-expert programmers, higher level programming languages such as C# might be considered. It should be investigated whether the interfaces and certain layers of the buffer framework can be made available via a *wrapper class* for C# [16, pages 139/175]. Potential issues of this approach are the effort needed to realize the wrapper, how easily the wrapper can be adapted afterwards to changes of the underlying code, and most of all: how the use of both a wrapper and a higher level language affects the performance of resulting applications.

As briefly indicated in Section 3.3.1, the buffer framework allows to be used together with different kinds of *input toolkits*, e. g., the SMART Board Software Development Kit (SDK) for the DViT. Recent research by Tse et al. on richer input possibilities for multi-user tabletop displays showed some impressive improvements and many useful ideas [62]. Combining the features of such a toolkit with the performance capabilities of the buffer framework could be a tremendous step forward towards powerful applications for tabletop displays. Issues to consider are the compatibility between such toolkits and the buffer framework as well as the impact on the overall performance due to intensive input pre-processing.

Concerning the buffer implementation, the *Strategy pattern* [16, page 315–323] might be useful. By this, different interpolation techniques could be provided for accessing a buffer off its regular grid. These

would allow the developer to choose between different types of quality and speed.

Another interesting area of work would be a more detailed investigation of the impact of graphics hardware on tabletop applications. Especially research on the advantages and disadvantages of using the graphics hardware for all buffer operations—or just a subset—could shed some light on further optimization possibilities of the buffer concept implementation.

Section 5.2.2 mentioned difficulties concerning the filling of buffers with the correct content. An idea to address this issue would be to generate buffer content from graphical input by providing special bitmaps that can be converted into buffer values. An example could be height maps that are loaded during initialization and which are then converted to fit into the required buffers. A possible next step of this idea could be a graphical authoring tool to create such bitmaps easily. Although such approaches might be of a limited use, they would help developers lacking a deeper mathematical understanding to get a better idea about how the buffer concept works and how it can be applied to realize interaction.

As this section shows, there are many different ways of reusing, extending, and improving the concept and implementation developed in this thesis. Thus, the design goals were clearly achieved or even exceeded. This emphasizes once more that the contributions presented in this thesis will be a good foundation upon which to build future software for tabletop display research. This is also reflected by the fact that the buffer framework is now maintained and extended by other researchers at the Interactions Laboratory of the University of Calgary. The latest version and updated information is publicly available via the lab's cookbook:

`http://grouplab.cpsc.ucalgary.ca/cookbook/`

Please note that license and copyright restrictions may apply.

6.3 PERSONAL REMARKS

Writing this thesis about a—for me—completely new topic and in an unknown environment was a big challenge. But due to the very friendly atmosphere in the Interactions Laboratory, I could adjust quickly to my tasks. Many group meetings and personal talks to the other researchers there helped me to understand the bigger picture of the ongoing research. One of the most important results of this is to have the buffer framework cooperate with input toolkits—a very crucial feature I might have excluded if I had not had many discussions with Edward Tse about the interaction between visualization and input software.

Having frequently external visitors from industry and the government in the laboratory for demonstrations was a new and exciting experience. It provided me with additional feedback and motivation as I could see to which applications my work might grow in the future.

After finishing most of the work in Canada, my supervisors helped me with a nearly seamless transition back to Magdeburg, where I could complete the thesis.

APPENDIX

CONTENTS OF THE ACCOMPANYING DVD

The DVD attached to this thesis includes all relevant information and software. The contents are structured as follows:

- `bin/` features an executable example application of the buffer framework for the Windows operating system.
- `code/` includes the source code and project files to compile the framework and the example application.
- `latex/` provides the LaTeX sources and graphics for this thesis.
- `movies/` has demos of the example application in MPEG format.
- `slides/` includes the slides and videos of the thesis' defense.
- `software/` contains a selection of useful and necessary software for the creation and evaluation of the framework.

COMPILING THE SOURCE CODE

The buffer framework was developed and tested using Microsoft Windows XP and Visual Studio .NET 2003. It is likely to compile, work, and run on other configurations, but the following setup can only be guaranteed for the development configuration:

- Install Visual Studio .NET 2003
- Install Qt 3.2.0 Educational. It is needed to build the example application. At the moment, it is also needed within the file `TabletopVisGL.cpp` to provide texture loading via `qimage.h`. This could easily be changed later on, if required.
  - Make sure to enable the integrating options "Microsoft Visual C++ .NET" and "Install .NET AddIn".
  - Also enable the option "Set `QTDIR`".
- *Optional*: Install SMART Board Software.
- Open the solution `UofCTabletopFramework.sln`.
- Build and run the solution. *Note*: If you have SMART Board Software installed and a firewall running it might notify you that the example application is trying to access the SMART Board. Unless you are actually using a SMART board and want to have two touches available, it does not matter whether you grant access or not.
- For fine tuning of the example without recompilation the file `config.xml` is available in the respective folder (`Debug` or `Release`).

BIBLIOGRAPHY

[1] Agile Alliance. *Manifesto of the Agile Alliance*, 2001. http://www.agilemanifesto.org. Website last visited on April 20, 2006. (Cited on page 56.)

[2] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, New York, NY, USA, 1977. (Cited on page 21.)

[3] Charles Baker, Sheelagh Carpendale, Przemyslaw Prusinkiewicz, and Michael Surette. *GeneVis: Simulation and Visualization of Genetic Networks*. *Journal of Information Visualization, Special Issue on Coordinated Multiple Views*, 2(4):201–217, December 2003. (Cited on page 31.)

[4] Steve Benford, Carsten Magerkurth, and Peter Ljungstrand. *Bridging the Physical and Digital in Pervasive Gaming*. *Communications of the ACM*, 48(3):54–57, March 2005. (Cited on page 12.)

[5] Jon Bentley. *Programming Pearls*. Addison–Wesley, Boston, MA, USA, 2nd edition, 1999. (Cited on page 40.)

[6] Anastasia Bezerianos and Ravin Balakrishnan. *View and Space Management on Large Displays*. *IEEE Computer Graphics and Applications: Special Issue on Large Displays*, 25(4):34–43, July/August 2005. (Cited on pages 7, 13, and 27.)

[7] Eric Bonabeau, Marco Dorigo, and Guy Théraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York, NY, USA, 1999. (Cited on page 31.)

[8] William Buxton, George Fitzmaurice, Ravin Balakrishnan, and Gordon Kurtenbach. *Large Displays in Automotive Design*. *IEEE Computer Graphics and Applications*, 20(4):68–75, July 2000. (Cited on pages 9 and 10.)

[9] Edwin Catmull. *Computer Display of Curved Surfaces*. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures*, pages 11–17, 1975. (Cited on page 19.)

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, 2nd edition, 2001. (Cited on page 63.)

[11] Lawrence D. Cutler, Bernd Fröhlich, and Pat Hanrahan. *Two-Handed Direct Manipulation on the Responsive Workbench*. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics (SI3D)*, pages 107–ff., New York, NY, USA, 1997. ACM Press. (Cited on pages 9 and 10.)

[12] Paul Dietz and Darren Leigh. *DiamondTouch: A Multi-User Touch Technology*. In *Proceedings of UIST 2001*, pages 219–226, New York, NY, USA, 2001. ACM Press. (Cited on pages 1 and 10.)

[13] Joseph Fall and Andrew Fall. *SELES: A Spatially Explicit Landscape Event Simulator*. In *Proceedings of the NCGIA Third International Conference on GIS and Environmental Modeling*, pages 104–112, Santa Barbara, CA, USA, 1996. National Center for Geographic Information and Analysis. (Cited on page 31.)

[14] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C*. Addison–Wesley, Boston, MA, USA, 2nd edition, 1995. (Cited on page 19.)

[15] Dick Gabriel and James O. Coplien. *A Pattern Definition*. `http://hillside.net/patterns/definition.html`, 2005. Website last visited on April 20, 2006. (Cited on page 21.)

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, Boston, MA, USA, 1995. (Cited on pages 21, 22, 24, 27, 28, 46, and 75.)

[17] François Guimbretière and Terry Winograd. *FlowMenu: Combining Command, Text, and Data Entry*. In *Proceedings of UIST 2000*, pages 213–216, New York, NY, USA, 2000. ACM Press. (Cited on pages 15 and 16.)

[18] Martin Hachet and Pascal Guitton. *The Interaction Table: A New Input Device Designed for Interaction in Immersive Large Display Environments*. In *Proceedings of the Workshop on Virtual Environments (EGVE) 2002*, pages 189–196, Aire-la-Ville, Switzerland, 2002. Eurographics Association. (Cited on pages 1 and 6.)

[19] Mark S. Hancock, Frédéric D. Vernier, Daniel Wigdor, Sheelagh Carpendale, and Chia Shen. *Rotation and Translation Mechanisms for Tabletop Interaction*. In *Proceedings of the First IEEE International Workshop on Horizontal Interactive Human-Computer Systems 2006 (TableTop 2006)*, pages 79–88, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press. (Cited on page 75.)

[20] Uta Hinrichs, M. Sheelagh T. Carpendale, and Stacey D. Scott. *Interface Currents: Supporting Fluent Face-to-Face Collaboration*. In *ACM SIGGRAPH 2005 Conference Abstracts and Applications*, New York, NY, USA, 2005. ACM Press. (Cited on pages 15, 18, 29, and 66.)

[21] Uta Hinrichs, M. Sheelagh T. Carpendale, Stacey D. Scott, and Eric Pattison. *Interface Currents: Supporting Fluent Collaboration on Tabletop Displays*. In *Proceedings of Smart Graphics 2005*, volume 3638 of *Lecture Notes in Computer Science*, pages 185–197, Berlin, Germany, 2005. Springer-Verlag. (Cited on pages 2, 16, 33, and 66.)

[22] Tobias Isenberg and Sheelagh Carpendale. *Framework for Supporting Responsive Interaction in Information Visualization Interfaces*, February 2005. Research Proposal, Department of Computer Science, University of Calgary. (Cited on page 17.)

[23] Tobias Isenberg, André Miede, and Sheelagh Carpendale. *A Buffer Framework for Supporting Responsive Interaction in Information Visualization Interfaces*. In *Proceedings of the Fourth International Conference on Creating, Connecting and Collaborating through Computing (C5) 2006*, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press. (Cited on pages 17, 29, and 33.)

[24] Donald E. Knuth. *Computer Programming as an Art*. *Communications of the ACM*, 17(12):667–673, December 1974. (Cited on page xi.)

[25] Donald E. Knuth. *Big Omicron and Big Omega and Big Theta*. *SIGACT News*, 8(2):18–24, April/June 1976. (Cited on page 63.)

[26] Russell Kruger, M. Sheelagh T. Carpendale, Stacey D. Scott, and Saul Greenberg. *Roles of Orientation in Tabletop Collaboration: Comprehension, Coordination and Communication*. *Journal of Computer Supported Cooperative Work*, 13(5–6):501–537, December 2004. (Cited on pages 13 and 14.)

[27] Russell Kruger, M. Sheelagh T. Carpendale, Stacey D. Scott, and Anthony Tang. *Fluid Integration of Rotation and Translation*. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)'05*, New York, NY, USA, 2005. ACM Press. (Cited on pages 13 and 16.)

[28] Gordon Kurtenbach and George Fitzmaurice. *Applications of Large Displays*. *IEEE Computer Graphics and Applications: Special Issue on Large Displays*, 25(4):22–23, July/August 2005. (Cited on pages 6, 9, and 13.)

[29] David Kushner. *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture*. Random House Trade Paperbacks, New York, NY, USA, 2004. (Cited on page 11.)

[30] Carsten Magerkurth, Adrian David Cheok, Regan L. Mandryk, and Trond Nilsen. *Pervasive Games: Bringing Computer Entertainment Back to the Real World*. *Computers in Entertainment*, 3(3):1–19, July 2005. (Cited on pages 11 and 12.)

[31] Carsten Magerkurth, Maral Memisoglu, Timo Engelke, and Norbert Streitz. *Towards the Next Generation of Tabletop Gaming Experiences*. In *Proceedings of the Conference on Graphics Interface (GI) 2004*, pages 73–80, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society. (Cited on pages 11 and 12.)

[32] Richard Mander, Gitta Salomon, and Yin Yin Wong. *A "Pile" Metaphor for Supporting Casual Organization of Information*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI) 1992*, pages 627–634, New York, NY, USA, 1992. ACM Press. (Cited on pages 14 and 45.)

[33] Philip E. Margolis. *Personal Computer Dictionary*. Random House, New York, NY, USA, 1991. (Cited on page 19.)

[34] Keith Mitchell, Duncan McCaffery, George Metaxas, Joe Finney, Stefan Schmid, and Andrew Scott. *Six in the City: Introducing Real Tournament – A Mobile IPv6 Based Context-Aware Multiplayer Game*. In *NETGAMES '03: Proceedings of the 2nd Workshop on Network and System Support for Games*, pages 91–100, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-734-6. (Cited on page 12.)

[35] Wayne Piekarski and Bruce Thomas. *ARQuake: The Outdoor Augmented Reality Gaming System*. *Communications of the ACM*, 45(1): 36–38, January 2002. (Cited on page 12.)

[36] David Pinelle, Carl Gutwin, and Saul Greenberg. *Task Analysis for Groupware Usability Evaluation: Modeling Shared-Workspace Tasks with the Mechanics of Collaboration*. *ACM Transactions on Computer-Human Interaction*, 10(4):281–311, December 2003. (Cited on page 14.)

[37] Jennifer Preece, Yvonne Rogers, Helen Sharp, David Brenyon, Simon Holland, and Tom Carey. *Human-Computer Interaction*. Addison–Wesley, Boston, MA, USA, 1994. (Cited on pages 1, 5, 6, and 9.)

[38] Jun Rekimoto and Masanori Saitoh. *Augmented Surfaces: A Spatially Continuous Work Space for Hybrid Computing Environments*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI) 1999*, pages 378–385, New York, NY, USA, 1999. ACM Press. (Cited on page 9.)

[39] Yvonne Rogers and Siân Lindley. *Collaborating Around Large Interactive Displays: Which Way is Best to Meet? Interacting with Computers*, 16(6):1133–1152, February 2004. (Cited on pages 1, 6, and 8.)

[40] Takafumi Saito and Tokiichiro Takahashi. *Comprehensible Rendering of 3-D Shapes*. In *Proceedings of ACM SIGGRAPH 90*, pages 197–206, New York, NY, USA, 1990. ACM Press. (Cited on page 20.)

[41] Stefan Schlechtweg, Tobias Germer, and Thomas Strothotte. *RenderBots—Multi Agent Systems for Direct Image Generation*. *Computer Graphics Forum*, 24(2):137–148, June 2005. (Cited on page 31.)

[42] Douglas C. Schmidt and Mohamed Fayad. *Object-Oriented Application Frameworks*. *Communications of the ACM*, 40(10):32–38, October 1997. (Cited on pages 18, 27, and 28.)

[43] Douglas C. Schmidt, Mohamed Fayad, and Ralph E. Johnson. *Software Patterns*. *Communications of the ACM*, 39(10):37–39, October 1996. (Cited on pages 21 and 22.)

[44] Stacey D. Scott, M. Sheelagh T. Carpendale, and Stefan Habelski. *Storage Bins: Mobile Storage for Collaborative Tabletop Displays*. *IEEE Computer Graphics and Applications: Special Issue on Large Displays*, 25(4):58–65, July/August 2005. (Cited on pages 9, 13, 14, 16, 17, and 45.)

[45] Stacey D. Scott, M. Sheelagh T. Carpendale, and Kori M. Inkpen. *Territoriality in Collaborative Tabletop Workspaces*. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW)'04, CHI Letters*, pages 294–303, New York, NY, USA, 2004. ACM Press. (Cited on page 14.)

[46] Stacey D. Scott, Karen D. Grant, and Regan L. Mandryk. *System Guidelines for Co-located, Collaborative Work on a Tabletop Display*. In *Proceedings of the European Conference Computer-Supported Cooperative Work (ECSCW)'03*, pages 159–178, Dordrecht, The Netherlands, 2003. Kluwer Academic Press. (Cited on pages 9 and 13.)

[47] Chia Shen, Neal Lesh, and Frédéric Vernier. *Personal Digital Historian: Story Sharing Around the Table*. *ACM Interactions*, 10(2):15–22, March/April 2003. (Cited on pages 9, 10, and 15.)

[48] Chia Shen, Neal B. Lesh, Frédéric Vernier, Clifton Forlines, and Jeana Frost. *Sharing and Building Digital Group Histories*. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work (CSCW) 2002*, pages 324–333, New York, NY, USA, 2002. ACM Press. (Cited on page 9.)

[49] Chia Shen, Frédéric D. Vernier, Clifton Forlines, and Meredith Ringel. *DiamondSpin: An Extensible Toolkit for Around-the-Table Interaction*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI) 2004*, pages 167–174, New York, NY, USA, 2004. ACM Press. (Cited on page 18.)

[50] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison–Wesley, Boston, MA, USA, 3rd edition, 1998. (Cited on pages 1, 5, and 17.)

[51] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison–Wesley, Boston, MA, USA, 5th edition, 2005. (Cited on page 60.)

[52] Ian Sommerville. *Software Engineering*. Addison-Wesley, Boston, MA, USA, 4th edition, 1992. (Cited on pages 18, 19, 27, and 28.)

[53] Olov Ståhl, Anders Wallberg, Jonas Söderberg, Jan Humble, Lennart E. Fahlén, Adrian Bullock, and Jenny Lundberg. *Information Exploration Using The Pond*. In *Proceedings of the 4th International Conference on Collaborative Virtual Environments (CVE) 2002*, pages 72–79, New York, NY, USA, 2002. ACM Press. (Cited on pages 9 and 10.)

[54] Jason Stewart, Benjamin B. Bederson, and Allison Druin. *Single Display Groupware: A Model for Co-Present Collaboration*. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI) 1999*, pages 286–293, New York, NY, USA, 1999. ACM Press. (Cited on pages 5, 6, and 8.)

[55] Norbert A. Streitz, Jörg Geißler, Torsten Holmer, Shin'ichi Konomi, Christian Müller-Tomfelde, Wolfgang Reischl, Petra

Rexroth, Peter Seitz, and Ralf Steinmetz. *i-LAND: An Interactive Landscape for Creativity and Innovation*. In *Proceedings of ACM SIGCHI 99*, pages 120–127, New York, NY, USA, 1999. ACM Press. (Cited on page 10.)

[56] FRAPS. *Official Website*, 2006. http://www.fraps.com. Website last visited on April 20, 2006. (Cited on pages 18 and 67.)

[57] MERL. *Mitsubishi Electric Research Laboratories, Official Website*, 2006. http://www.merl.com. Website last visited on April 20, 2006. (Cited on pages 1 and 10.)

[58] SMART Technologies, Inc. *Official Website*, 2006. http://www.smarttech.com. Website last visited on April 20, 2006. (Cited on pages 9 and 10.)

[59] Dave Thomas and Andy Hunt. *OO in One Sentence: Keep It DRY, Shy, and Tell the Other Guy*. *IEEE Software*, 21(3):101–103, May/June 2004. (Cited on pages 27 and 41.)

[60] David Thomas and Andrew Hunt. *The Pragmatic Programmer: From Journeyman to Master*. Addison–Wesley, Boston, MA, USA, 1999. (Cited on pages 41, 42, 56, and 57.)

[61] Edward Tse and Saul Greenberg. *Rapidly Prototyping Single Display Groupware Through the SDGToolkit*. In *Proceedings of the Fifth Conference on Australasian User Interface (CRPIT) 2004*, pages 101–110, Darlinghurst, Australia, 2004. Australian Computer Society, Inc. (Cited on pages 18 and 19.)

[62] Edward Tse, Chia Shen, Saul Greenberg, and Clifton Forlines. *Enabling Interaction with Single User Applications through Speech and Gestures on a Multi-User Tabletop*. In *Proceedings of the International Conference on Advanced Visual Interfaces (AVI) 2006*, River Edge, NJ, USA, 2006. World Scientific Publishing Co., Inc. (Cited on page 75.)

[63] Alan H. Watt. *3D Computer Graphics*. Addison–Wesley, Boston, MA, USA, 3rd edition, 1999. (Cited on page 20.)

[64] Pierre Wellner. *Interacting with Paper on the DigitalDesk*. *Communications of the ACM*, 36(7):87–96, July 1993. (Cited on pages 8 and 9.)

[65] Wikipedia (Alto). *Entry about the Alto*, 2006. http://en.wikipedia.org/wiki/Alto_(computer). Website last visited on April 20, 2006. (Cited on page 5.)

[66] Wikipedia (Computer Games). *Entry about computer games*, 2006. http://en.wikipedia.org/wiki/Computer_game#Popularity. Website last visited on April 20, 2006. (Cited on page 11.)

DECLARATION

I declare that this thesis has been composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Furthermore, I declare that this thesis may be used for publication by my supervisors, Prof. Dr. Sheelagh Carpendale, Prof. Dr.-Ing. Maic Masuch, and Dr.-Ing. Tobias Isenberg.

Magdeburg, April 2006

André Miede

COLOPHON

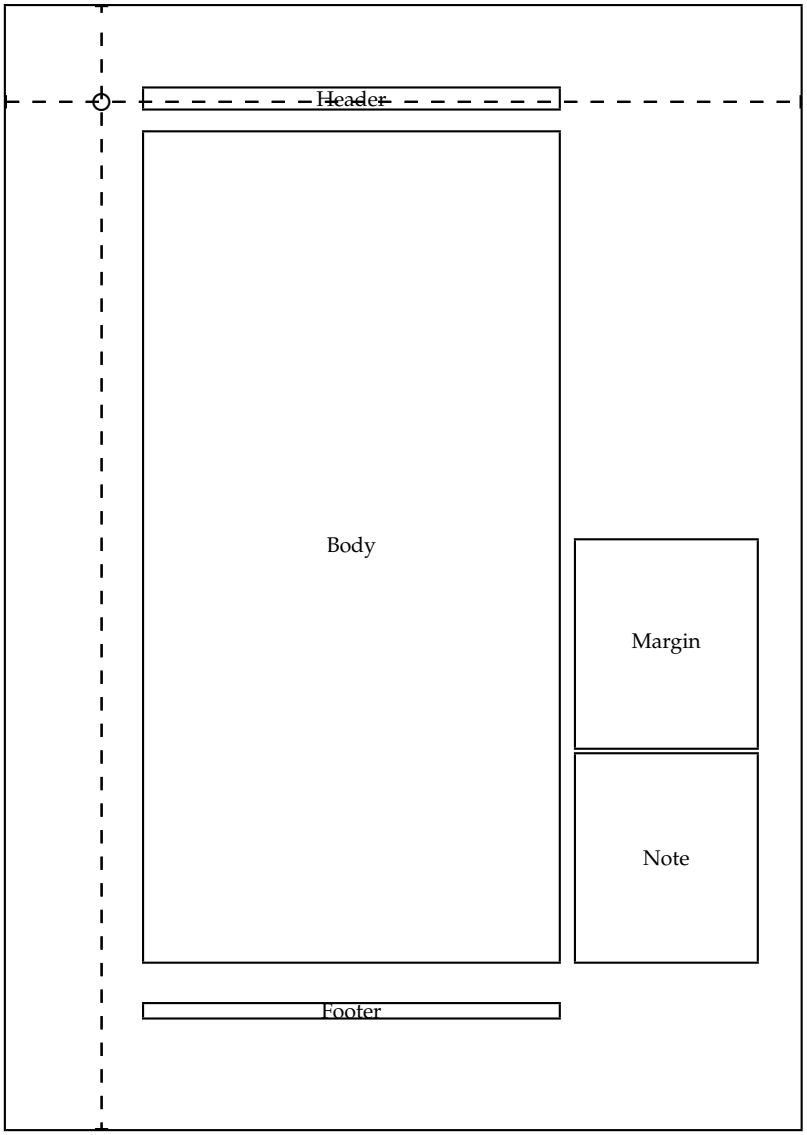
This thesis was typeset with LaTeX 2ε using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

The typographic style was inspired by Robert Bringhurst's genius as presented in *The Elements of Typographic Style* (Version 2.5, Hartley & Marks, 2002) and is available for LaTeX via CTAN as "`classicthesis`".

Final version as of April 20, 2006.

NOTE: The custom size of the textblock was calculated using the directions given by Bringhurst (pages 26–29 and 175/176).

10 pt Palatino needs 133.21 pt for the string "abcdefghijklmnopqrstuvwxyz". This yields a good line length between 24–26 pc (288–312 pt). Using a "*double square textblock*" with a 1:2 ratio this results in a textblock of 312:624 pt (which includes the headline in this design). A good alternative would be the "*golden section textblock*" with a ratio of 1:1.62, here 312:505.44 pt. For comparison, `DIV9` of the `typearea` package results in a line length of 389 pt (32.4 pc), which is by far too long. However, this information will only be of interest for hardcore pseudo-typographers like me.



`\paperheight` = 845.04694pt	`\paperwidth` = 597.50793pt
`\hoffset` = 0.0pt	`\voffset` = 0.0pt
`\evensidemargin` = 108.58438pt	`\oddsidemargin` = 32.38356pt
`\topmargin` = -9.58768pt	`\headheight` = 15.0pt
`\headsep` = 18.0pt	`\textheight` = 624.0pt
`\textwidth` = 312.0pt	`\footskip` = 42.0pt
`\marginparsep` = 12.8401pt	`\marginparpush` = 5.39996pt
`\columnsep` = 10.0pt	`\columnseprule` = 0.0pt
1em = 10.0pt	1ex = 4.68999pt