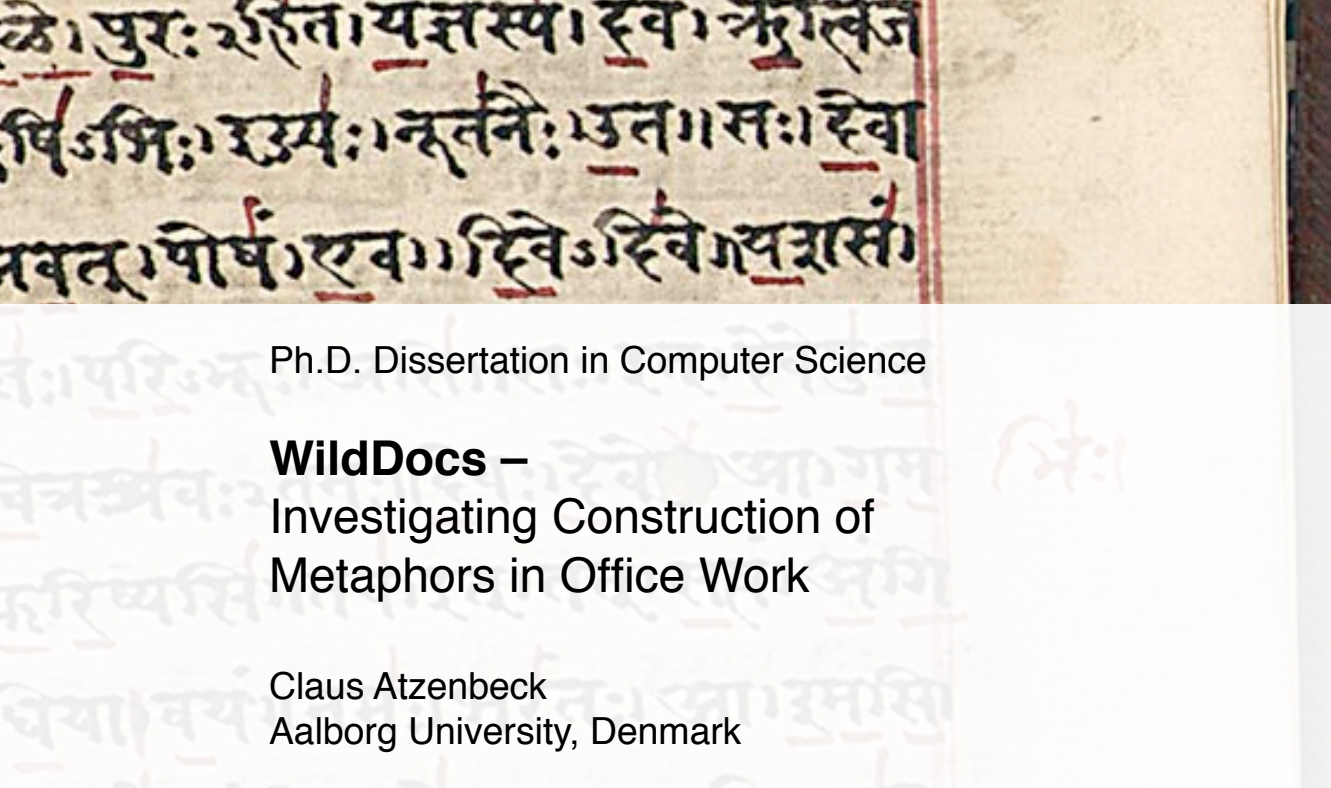
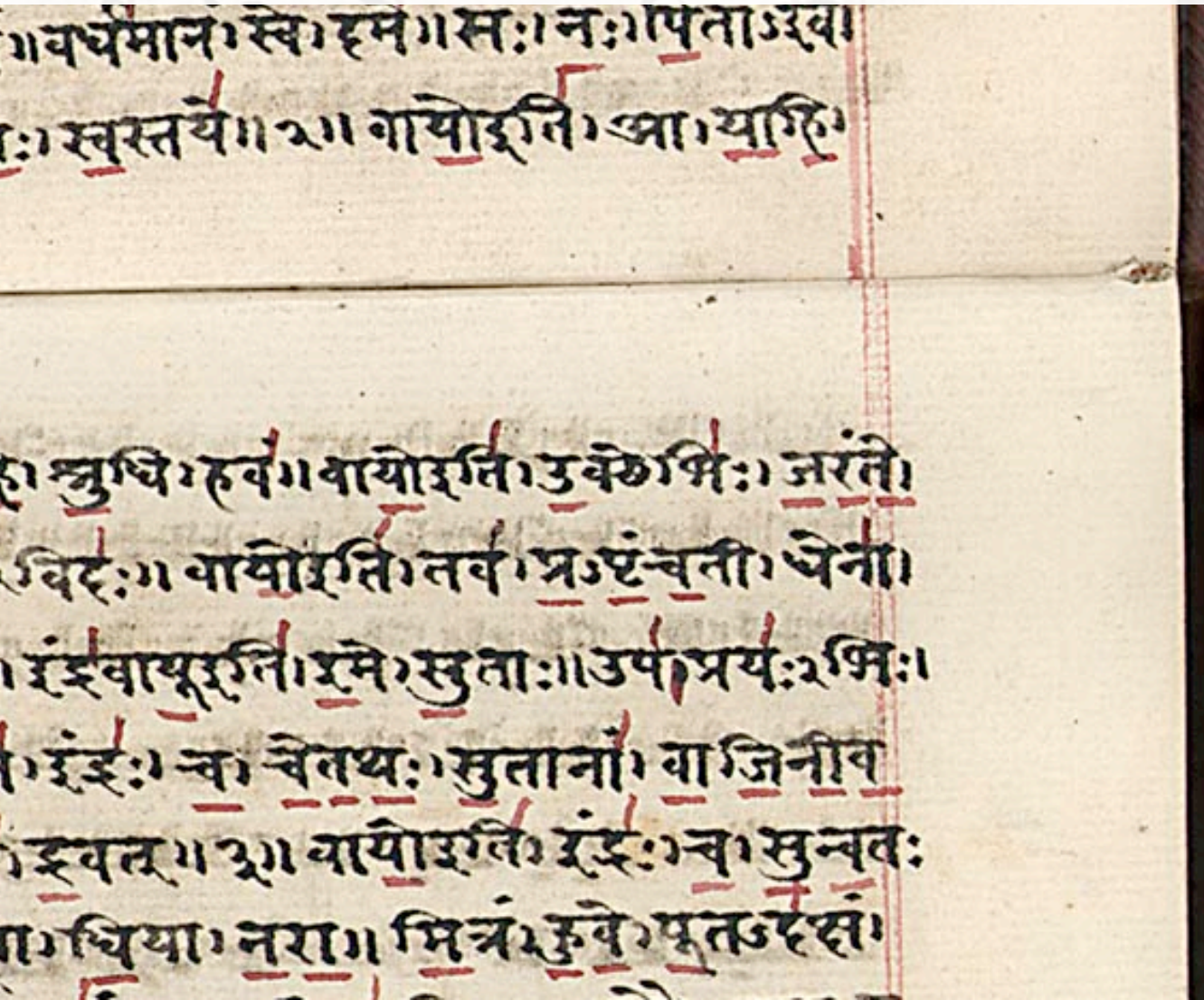


Ph.D. Dissertation in Computer Science

WildDocs – Investigating Construction of Metaphors in Office Work

Claus Atzenbeck
Aalborg University, Denmark



ISBN: 87-7606-014-4

Document version: July 4, 2006 – 10:58

Cover: Rigveda MS in Sanskrit on paper, India, early 19th c., 4 vols., 795 ff., 10 cm × 20 cm (magnified), single column (7 cm × 17 cm), 10 lines in Devanagari script with deletions in yellow, Vedic accents, corrections, etc., in red. Binding: India, 19th c., blind-stamped brown leather, gilt spine, sewn on 5 cords, marbled endleaves. Picture taken from http://commons.wikimedia.org/wiki/Image:Rigveda_MS2097.jpg

Ph. D. Dissertation in Computer Science

WildDocs – Investigating Construction of Metaphors in Office Work

Claus Atzenbeck
http://www.atzenbeck.de
Aalborg University, Denmark

July 4, 2006

Advisor
Peter J. Nürnberg, Ph. D.

To my Family.

Contents

List of Figures

List of Tables

Abstract (English)

Knowledge is an important resource in an information society. People use it to develop new products, find new medical treatment to fight diseases, adapt or change international relationships, or gain new knowledge. The amount of knowledge constantly grows and therefore produces the need for tools that help people to structure, store, or retrieve information.

One example of a tool is paper, an old medium that still is used to store, exchange, or retrieve information. People have been refining this carrier to support knowledge worker in managing their work load. Paper needs to be organized. Therefore, libraries were invented even in ancient times that allow, for example, civil servants to reach the information they need efficiently. This is still true today.

In the late 1970s, another tool that aims to solve the problems of structuring, storing, and retrieving of information was introduced: personal computers. They have become normal devices in today's offices, used by millions of office workers.

Computers can run databases that store large amounts of data or offer information retrieval systems that support the user in finding information. Recently, semantic technologies became popular in computer science. They allow applications (so-called "agents") to use attached semantics for improving retrieval related functionality.

Another branch of research and applications focus on structure domains. There is a variety of structures, each built with well-defined tasks in mind. For example, taxonomic structures are appropriate for classification, such as those used in biology. The classification must exist before biologists can start classifying plants or animals.

These techniques may not be appropriate when the final structure is not known. For example, associations come up during brainstorming sessions that do not follow a predefined structure. One way to represent the associations is to use small pieces of paper, write or draw the associated term or a picture on it, and place it on a workspace. During the session, participants can move these nodes around to express relationships among the information snippets. The structure changes constantly over time. Most of the structure is implicit, such as spatial arrangement based on completely freely movable snippets.

Brainstorming sessions usually take only a short period of time. Other paper-based structures, for example, structures of printed articles or books on a desk, evolve over weeks or months. Devices, such as binders, folders, or shelves, help to put them in place. In many cases, offices have to be restructured due to lack of space or growing pieces of information. That leads to emerging spatial arrangements that exist beside predefined ones (e. g., computer science books are located at the top part of the shelve). Also here, spatial structures that were created or modified over time carry implicit metainformation to a large extent. This metainformation is best interpreted by the person who created it.

Most of this implicit information happened to be created without any explicit intention. For example, there may be a sloppily arranged pile of articles at the right side of the desk, because there was no place on the left side. Furthermore, it is sloppy, because the office worker did not have time to align it properly. The person knows that this is a preliminary pile, because of its position and its shape.

There are computer applications that support the informal creation of spatial structure. Examples include spatial hypertext application. They are based on a cards-on-table metaphor. However, physical cards on a table look and behave differently to shapes in spatial hypertext applications. As many metaphor-based applications, spatial hypertext applications' metaphor implementations are highly abstract, compared to the original real world referents. Many influences that makes spatial structure emerge over time, as described above, are ignored. This causes the reduction of implicit metainformation.

For example, physical forces (e. g., gravity or friction) or incidental rotation during moving an object are ignored. The size of the space is practicably unlimited in applications, whereas the real world has limited space on a desk. A closer look shows that physical paper structures are of higher complexity than equivalent metaphor-based implementations.

In this thesis, we describe different structures created with paper and propose a classification scheme. Then, we compare these structures to selected computer applications. We argue that the discovered aspects may improve finding and organizing of information. However, most are not implemented in applications due to a high abstraction level.

For this thesis we focus on rapid zooming, rotation, and fixed size documents. We claim that those interactions or attributes will decrease the time for finding information significantly, because they support natural interaction and/or emerging implicit metainformation. In order to test this, we built a prototype, WildDocs, a 2D-based spatial application that supports the requested features. We found our expectations about more effective information retrieval partly supported.

This thesis includes a detailed discussion of WildDocs. One basic concept of this application is that all documents are considered structured and structuring at the same time. The implementation is based on the classification for paper structures we defined and provides a high degree of freedom to system developers to add new document types.

Furthermore, we point to related areas that are of future interest. Those include extending WildDocs to support paper-like movement (e. g., through simulation of gravity or friction), or act on physical/digital mixed environments or as window manager. We also describe an input device that is designed to match WildDocs's navigation and zooming features. Finally, we discuss WildDocs and its integration in structural computing environments.

Resumé (Dansk)

Viden er en vigtig ressource i informationssamfundet. Den anvendes til at udvikle nye produkter, nye behandlinger for sygdomme, tilpasse eller ændre internationale relationer, eller finde ny viden. Mængden af viden vokser konstant, hvilket skaber et behov for værktøjer der hjælper mennesker med at strukturere, lagre, og genfinde information.

Et eksempel på et værktøj er papir, der stadig anvendes til at gemme, udveksle, og hente information. Man har forædlet værktøjet til at understøtte vidensarbejderen i at håndtere sin arbejdsbyrde. Papir skal organiseres. Derfor opfandt man biblioteker der, selv i gammel tid, tillod, for eksempel embedsmænd, at få fat i den information de havde behov for, på en effektiv måde. Dette gælder stadig i dag.

Sidst i 1970'erne introduceredes et nyt værktøj, som også er målrettet til at løse problemer med strukturering, lagring, og genfinding: den personlige computer. I dag er de blevet almindelige redskaber, som anvendes på millioner af kontorer.

Computere kan køre databaser der gemmer store mængder af data eller tilbyder information retrieval systemer der understøtter brugeren i at genfinde information. I nyere tid er semantiske teknologier blevet populære indenfor datalogien. De tillader applikationer (såkaldte »agenter«) at udnytte semantisk information til at forbedre genfindingsfunktionalitet.

En anden gren af forskning og applikationer fokuserer på strukturdomæner. Der er en mangfoldighed af strukturer, hver bygget med henblik på en specifik opgave. Taksonomiske strukturer, for eksempel, er egnet til klassificering og anvendt til dette i biologi. Man er nødt til at have en klassifikation før man kan begynde at klassificere planter og dyr.

Disse teknikker er ikke nødvendigvis hensigtsmæssige når den endelige struktur ikke er kendt. Eksempelvis associationer der kommer frem i en brainstorm, der ikke følger nogen prædefineret struktur. En måde at repræsentere disse associationer på, er at benytte små stykker papir, skrive eller tegne det associerede term eller et billede på dem, og placere dem på en arbejdsflade. I løbet af sessionen kan deltagerne så flytte rundt på noderne, for at udtrykke sammenhænge imellem informationsbidderne. Strukturen ændrer sig konstant over tid. Det meste af strukturen er implicit, så som den spatiale arrangering af de frit flytbare informationsbidder.

Brainstorming sessions finder som oftest sted i et begrænset tidsrum. Andre papirbaserede strukturer, for eksempel strukturer af printede artikler eller bøger på et skrivebord, udvikler sig over uger eller måneder. Anordninger, så som bind, mapper, og hylder hjælper med at holde dem på plads. I mange tilfælde bliver kontorer omorganiseret, eksempelvis på grund af manglende plads eller voksende informationsenheder. Dette fører til udviklingen af nye spatiale arrangementer, der eksisterer side om side med de prædefinerede (eksempelvis datalogibøger står på den øverste sektion af reolen). Også her indeholder de spatiale strukturer, der er blevet skabt eller modificeret over tid, i vid udstrækning implicit metainformation. Denne metainformation tolkes bedst af den person der har skabt den.

Det meste af denne implicitte metainformation er blevet skabt uden at det var skaberens eksplicitte hensigt. For eksempel, kan der stå en sjusket arrangeret stak af artikler på højre side af skrivebordet, fordi der ikke var plads på den venstre. Ydermere er den sjusket fordi

kontorarbejderen ikke havde tid til at arrangere den ordentligt. Personen ved at det er en midlertidig stak, på grund af dens position og form.

Der findes computer applikationer der understøtter den uformelle skabelse af implicitte strukturer. Eksempler på dette inkluderer spatiale hypertekst applikationer. De er baseret på en kort-på-bordet metafor. Fysiske kort på et bord opfører sig imidlertid anderledes og ser anderledes ud end former i spatiale hypertekst applikationer. Som mange metaforbaserede applikationer er spatial hypertekst applikationers metafor implementeret på et meget abstrakt niveau, sammenlignet med de originale »virkelige« koncepter. Mange af de indflydelser der får spatiale strukturer til at fremkomme over tid, som beskrevet ovenfor, bliver ignoreret. Dette medfører en reduceret mængde af implicit metainformation.

For eksempel bliver de fysiske kræfter (eksempelvis friktion og tyngdekraft) og tilfældig rotation imens man flytter et objekt ignoreret. De fysiske rammer er af, praktisk talt, uendelig størrelse, hvorimod borde i den virkelige verden har begrænset plads. Nærmere undersøgelse viser at de fysiske papirstrukturer er af højere kompleksitet end deres ækvivalente metaforbaserede implementationer. I denne afhandling beskriver vi forskellige strukturer der er skabt med papir og foreslår et klassificeringsskema. Herefter sammenligner vi disse strukturer med udvalgte computer applikationer. Vi hævder at de fundene aspekter kan forbedre genfinding og organisering af information. De fleste er imidlertid ikke implementeret i applikationer, grundet et højt niveau af abstraktion.

I denne afhandling fokuserer vi på hurtig zoomning, rotation, og dokumenter af fast størrelse. Vi påstår at disse interaktioner og attributter vil reducere tiden det tager at genfinde information betydeligt, fordi de understøtter en naturlig interaktion og udviklingen af implicit metainformation. For at teste dette, har vi konstrueret en prototype, WildDocs, en 2D-baseret spatial applikation der har de fornødne egenskaber. Vi fandt at vores forventninger om en mere effektiv genfinding af information var delvist underbygget.

Denne afhandling slutter med en detaljeret diskussion af WildDocs. Et grundlæggende koncept i denne applikation er at alle dokumenter bliver betragtet som både strukturerede og strukturerende på samme tid. Implementationen er baseret på den klassifikation af papirstrukturer vi definerede og giver en høj grad af frihed for systemudviklere til at udvikle nye typer af dokumenter.

Ydermere peger vi på relaterede områder der kan være af interesse i fremtiden. Blandt disse er at udvide WildDocs til at understøtte papiragtig bevægelse (eksempelvis igennem simulering af tyngdekraft og friktion), og at agere i fysisk/digitalt blandede miljø eller som window manager. Vi beskriver også en inputanordning der er designet specifikt til at matche WildDocs' navigation og zoomning faciliteter. Endelig diskuterer vi WildDocs og dens integration med structural computing miljø.

Part I.

Theory

Chapter 1.

Introduction

„Es ist die Arbeit der Interpretation im Kopf, die aus den Zeichen, die Computer anzeigen, eine Information macht. Wir kriegen auch meistens nicht die Zeichen, die wichtig sind für eine Entscheidung.“

(Joseph Weizenbaum, May 2005)

1.1. Physical and Digital Documents

Paper is a very important medium in our culture. For centuries it has been used to carry information. Equally important are structures that were built with paper. For example, single sheets were bound as books. Those were stored in libraries, placed in areas assigned to specific topics, or ordered by authors. Structuring supports fast retrieval and easy traversal. Structured information are easier to understand and therefore support knowledge exchange more efficiently.

Today's offices have a variety of tools that support structuring paper. Perforators let office workers punch holes into sheets to be put into binders afterwards. Trays hold loose sheets in place. Staples are used to clip individual sheets. Copy shops offer binding of collections of sheets as books. Shelves hold various objects, such as binders, books, or even piles of paper. Such facilities have been developed over time and are mostly well-known through long term use.

When computers were brought to offices, they were an additional tool to be used beside paper. They are capable of storing, transporting, and displaying information independently of paper. Computers have even been seen as a potential replacement for paper. A discussion about the “paperless office” began. However, the opposite happened. More paper has been used.

It seems to be a paradox that new technology that offered to reduce the amount of paper in an until then unexpected manner increased the amount of paper used (Frohlich & Perry, 1994). Even more, Whittaker & Hirschberg (2001, 157) have shown that “[o]nly 49 % of people's original archive was unique: 15 % was unread, and 36 % consisted of copies of publicly available documents.” Easy printing of digital documents may be one reason for extensive paper use. Another and more important reason may be advantages of physical paper:

> “The use of paper in the modem world persists *because* of its physical properties, not only despite them. Paper is convenient, requiring no batteries. It is light, readable, and durable. It has a tangible, persistent existence that is valued in

legal and business transactions." (Johnson et al., 1993, 512)

The gap between digital and physical documents has led to two research directions: "augmenting real paper with computational properties, and simulating the properties of paper in an electronic document" (Ashdown, 2004, 19). For example, apparatuses have been developed that project digital documents onto a real desk (e. g., Ashdown, 2004). On the other side, new software and hardware technology have raised potentials of simulations to photorealistic quality, for example, realistic animation of turning a virtual book's pages (e. g., Chu et al., 2003).

1.2. Structures

1.2.1. General

Merging physical and digital paper is not the only attempt to close the gap between both media types. Other research directions explicitly state the additional features of digital media. One example is hypertext. The term was originally introduced as "a body of written or pictorial material interconnected in such a complex way that it could not conveniently be presented or represented on paper" (Nelson, 1965, 96). This definition underlines the difference between paper and computer, especially the ability of machines to do something that is not possible on physical paper.

Interestingly, some components in hypertext systems were named after paper related items. For instance, a "node" is called "card" in NoteCards and HyperCard, "document" in AUGMENT and Intermedia, or "article" in Hyperties, as summarized in Halasz & Schwartz (1994, 32).[1] Other systems use the term "pages" for nodes (e. g., the WWW). It is also interesting to note that the system to which the hypertext pioneers refer to was based on microfilm, a physical medium: Memex has been described by Vannevar Bush (Bush, 1945) in the middle of last century, but never implemented. The name is an abbreviation for "Memory Extender". This machine was designed to "build a trail of [...] items" which can be followed again at any time later.

Based on Bush's idea, hypertext was originally based on a *node–link* concept. Nodes are connected via links. One of the most prominent models is the Dexter Hypertext Reference Model (Halasz & Schwartz, 1994; Grønbæk & Trigg, 1994). It was designed to provide a generic model for system developer as well as to unify the diverse terminology.

In the late 1990s, the hypertext research community adapted different other structure types and introduced them under the umbrella "hypertext". Those included *spatial hypertext*, which was originally introduced as "graphical knowledge structure" (Marshall et al., 1991, 262). This model refers to the real world by implementing a card-on-table metaphor. Objects can be related by spatial positioning or visual cues, similar to what people can do with small cards on a table. Another structure type that was introduced to the community under the term "hypertext" was *taxonomic hypertext* (Parunak, 1991).

Many other structure or application domains have been discusses within or related to hypertext. For example, *argumentation structures* (Conklin & Begeman, 1987, 1988), *hypertext*

[1] Examples for the use of those terms can be found in Halasz et al. (1987) for NoteCards, Smith & Bernhardt (1988) for HyperCard, Engelbart (1984) for AUGMENT, Garrett et al. (1986) for Intermedia, and Shneiderman (1987) for Hyperties.

fiction and *narrative structures* (Bernstein, 1998; Weal et al., 2001b), *ontology support* (Weal et al., 2001a), *musical structures* (De Roure et al., 2002), or *multi-structure interfaces* (Wang & Fernández, 2002) and *structure interoperability* (Atzenbeck & Nürnberg, 2004). A recent branch discusses the linking of digital and physical objects, known as *physical hypermedia* (Grønbæk et al., 2003). Currently, a new structure domain brought to the hypertext community deals with social structures.[2]

Even though we have mentioned many different structure types that were discussed with respect to the medium computer, they do not have to be necessarily represented digitally. For example, Mark Lombardi (1951–2000) was an artist who created aesthetic drawings that represent relationships among people and organizations. He calls them *narrative structures*.[3]

Lombardi used a large number index cards with handwritten notes as database for his drawings. They relate to current incidents at that time and include people of public interest. For example, one drawing is titled "Banca Nazionale del Lavoro, Reagan, Bush, Thatcher, and the Arming of Iraq, 1979–90" (Hobbs, 2003, 87–91).[4] Lombardi extracted implicit information that he found in public sources (i. e., newspapers), noted them on index cards, and finally drew their interrelations.

1.2.2. Levels and Multi-Layers

Our reflection of this world is very data centric. This can be seen in the way computer application are programmed and used. However, by taking a closer look it turns out that structure is an equivalent view. For example, a tree has a certain structure. Observers easily can recognize two parts: one trunk and many branches. By "zooming" closer, they will see that the branches have their own structure. One part of it is comprised of leaves. Also these have a specific structure. The observers can continue doing this until the level of atoms or beyond.

This leads to a model that proposes structure as being built upon structure recursively. However, if everything is structure, where would be "data"? It can be argued that data appears at the level where structure is no longer perceived. Therefore, this model provides that elements are both structuring and structured at the same time.[5] It is a matter of detail that a person perceives.

We call the level of details in our research *structure level*. It is also named "scale" in the literature: Ware (2004, 338–339) distinguishes between three different scales: scaling with respect to specific spatial locations (e. g., Europe on a world map); conceptual levels of detail (e. g., level of bookshelf → book → text, etc.); and timing of events (e. g., following a package from being sent to its delivery).

Humans can "jump" from one scale or structure level to another. For example, this can be performed by rapid zooming in graphical computer applications or by getting closer to a document in the real world. In both cases, more details will be revealed (assuming that they exist in the computer application).

[2] The 17th ACM Conference on Hypertext and Hypermedia, Odense, Denmark, will be the first conference in this series with a special focus on "Tools for Supporting Social Structures" (see http://www.ht06.org; visited on 2006-04-06).

[3] Lombardi writes in an artist statement about his drawings: "I call them 'narrative structures' because each consists of a network of lines and notations which are meant to convey a story, typically about a recent event of interest to me, like the collapse of a large international bank, trading company, or investment house. One of my goals is to explore the interaction of political, social and economic forces in contemporary affairs." (Lombardi, 1997)

[4] This and other drawings can be seen on http://www.pierogi2000.com/flatfile/lombardi.html (visited on 2006-04-06).

[5] This is further discussed in Sect. 6.2.5 with respect to a research direction called *structural computing*.

A single structure may consist of several structure dimensions that exist in parallel. For example, a spatial structure may use spatial alignment of objects to express relationships among them. Another relationship may be color. Red colored nodes, even though not necessarily placed closely to each other, may represent nodes of the same kind. Objects with such visually represented data variables are called *glyphs* (Ware, 2004, 176–185). Other possible visual variables include shape, orientation, surface texture, motion coding, and blink coding (Ware, 2004, 183). They can be used in parallel.

There are other structures that live in parallel. They are discussed originally in linguistics as textuality criteria. They are used to express the relation among different parts of a text. The most relevant criteria in this work is *cohesion*. It describes grammatical dependencies on texts (de Beaugrande & Dressler, 1981, Chap. 4). The remaining textuality criteria are coherence (meaning), intentionality, acceptability, informativity, situationality, and intertextuality.

A text can be described according to its intrinsic grammatical connections among its parts (i. e., cohesion); it also can be seen with respect to the coherence among its parts; etc. There has been research done in evaluating hypertext with respect to these criteria (Hammwöhner, 1997; Brügger, 2001).

1.2.3. Importance of (Structure) Details

Detailed representations may be disturbing, but they also may be necessary to understand information correctly. Consider the following story:

> A Mafia boss was found dead inside his car. He has been shot. The car is not damaged. All doors are locked and the windows up. The only key is still in the ignition lock. However, no weapon was found in the car. How can the Mafia boss possibly be shot?[6]

It seems mysterious how the Mafia boss can possibly be shot, since the car is locked from the inside and no pistol or gun was found inside. The text is an abstraction of what happened in reality. The author leaves away irrelevant information, for example, that swallows passed along when the police arrived. However, since this is a riddle, some relevant information are missing as well. The circumstances become clear when it is revealed to the listener that the car was a cabriolet.

The information about the type of car was hidden on purpose; otherwise, the riddle would have been spoiled. This was an intended violation of Grice's conversational maxims of quantity and manner (Grice, 1975).[7]

What is obvious with this riddle is not always obvious for other cases. We have referred many times in the previous sections to structure types or applications that borrow ideas or concepts from the real world: they implement metaphors. In order to implement a metaphor, the relevant part of the real world needs to be analyzed. A description is more abstract than the described item or situation. Grice's second part of the maxim of quantity – "Do not make

[6] Storyline taken from Springfeld (2004).

[7] We see the first part of the maxim of quantity – "Make your contribution as informative as is required" (Grice, 1975, 45) – violated, because in order to understand the circumstances fully, the information about the car being a cabriolet would have been needed. Furthermore, we also see one of the maxims of manner – "Avoid ambiguity" (Grice, 1975, 46) – is violated, because the term "car" can be seen as ambiguous in this case: either a vehicle with a closed car body or a cabriolet. The author intends to make the recipient think of a closed car. The maxim of quantity and the maxim of relation were met, however.

your contribution more informative than is required." (Grice, 1975, 45) – becomes a problem. Leaving away information may lead to models that miss helpful parts or behavior.

This means for the descriptions of structures that one must pay attention and argue carefully *why* one stops unfolding or perceiving structures at that particular level, but does not go further.

1.3. Use Case Scenarios

This section shows the use of rich structure details in office work by slightly exaggerated use case scenarios. They additionally provide an impression of the main focus of this work.

The central person of the use cases is Otto, a knowledge worker. He works in a company dealing with lots of information daily. His workflow includes paper as well as digital documents. Otto needs to find certain documents for his projects. In the following, we describe the questions Otto raises and his answers with respect to the used medium.

Otto's question: "Where did I put the red document that I had two days ago?" – *Otto using computer:* "I don't remember the folder I put it to, certainly not into my file archive. I cannot search for keywords, because I don't remember them either. My directory folders contain thousands of documents." – *Otto using paper:* "I had the document on my desk. I remember that I did not file it yet. It is somewhere in my piles, probably in the messy one, because I did not have time to sort those either. There it is! I can see parts of the red paper peeking out of this pile."

Otto's question: "I had something in mind, but do not remember what." – *Otto using computer:* "No idea. This is similar to when I was looking for the red document, only worse." – *Otto using paper:* "Good that I saw this document on my desk. It reminded me of what I need to do today."

Otto's question: "I lose the overview of my documents, because I simply have too many of them here." – *Otto using computer:* "There are many documents in my folders, but I will not spend the time now to reorganise. It would take too long." – *Otto using paper:* "I cleanup my desk daily, because otherwise I would run out of space. I get rid of what is obsolete, and become aware of important documents. This does not take long, but it helps to sort my documents and thoughts constantly and prioritise them."

Otto's question: "There were three versions of this report. Which one is the final one?" – *Otto using computer:* "Well, I have to check their creation date; however, this may not be true, because an earlier version was sent to me later by e-mail. It will take some time, but I will find out." – *Otto using paper:* "I remember that the final version has been bound as book, because it will not change anymore. An earlier version that I printed is this pile over there, because I only used it temporarily for browsing through. The pre-final version is this stapled one. I remember that a colleague stapled it before giving it to me in order not to mess up the sequence of pages. It was easy to remember the different versions, based on how the pages were put together."

Otto's question: "Where is the catalogue that has about 600 pages?" – *Otto using computer:* "I remember that the catalogue is a PDF document, but in my file system browser I cannot see how many pages the PDF files have. It is not practical to open all of them."

– *Otto using paper:* "Finding this 600 page book was not a problem. I found it right away because it is big enough that it caught my eye immediately."

These cases show situations where Otto finds information more easily on his desk, especially when information is missing that could be used for searching conveniently on computers. Otto is reminded by the existence of documents on his desk. Various bindings, such as stapled piles or books, attach information *about* the document that Otto helped to find it. A high level of structure details is provided. Because his desk is limited in size, he needs to clean it up regularly, which he might not do otherwise. However, restructuring makes him aware of all documents on his desk that are part of his current workflow. This raises the question of how Otto's computer application could be modified in a way that would allow him to gain similar positive effects in such use cases using computers.

In the beginning of this research project, observations similar to the aforementioned use cases made us consider that real paper might show advantages in certain situations compared with computer applications, including those that are metaphor driven. In order to develop our inspiration further, we worked on finding differences between physical paper and digital equivalents. Beside physical restrictions, such as capacity, weight, and size, we also considered differences between moving physical versus digital objects spatially and changing focus and context.

Furthermore, we considered paper structures, such as piles. Many of these look more or less sloppy, whereas graphically presented objects on a screen are often neatly aligned. Based on our first impression and some thought experiments, such as the aforementioned use cases, we expected that properties of physical objects would help users organizing and finding information more efficiently compared to paper metaphor-based applications without those properties. This includes fixed size documents (simulating paper), emerging sloppiness (simulating movement of physical objects), and efficient focusing (simulating eye focusing).

In this thesis we walk along the line of physical paper and computer by taking a closer look at the background and origin of our work: paper metaphor-based applications used for knowledge work in offices. How can we improve such applications for knowledge workers to gain some of the positive effects that we have with real world paper.

1.4. Summary and Agenda

In summary, we note that there is a variety of different structures used for knowledge representation. Many of them are designed to be used on computers; however, some can be also experienced in other areas, such as art. Many of those structures have a direct or indirect connection to real world objects in use (i. e., metaphor) or terminology; for example, when referring to nodes as cards on a (virtual) desk.

Structure elements are both structuring and structured. What is known as "data" can be perceived as "structure" as well. From a human perspective, "data" appears when structure is not relevant for the task at hand. A person can "jump" between different levels of structure. This is true for physical (e. g., paper) as well as for computer related structures. There may be different layers of structure found in parallel as well.

The example with Grice's maxims has shown that structure analysts should be aware of the abstract nature of their description. People need to question the chosen level of detail in order to reveal potential deeper structure levels that may help to improve an application or structure

technique. The use case scenarios round off the introduction with examples that show the relevance of structure details and metainformation in office work.

These underlying observations will be the ground for the following chapters. They are grouped in three parts: Part I discusses the theoretical background of our work. Part II describes the application implementation, experimental design and evaluation, concludes this work, and refers to open future work. Finally, Part III contains the appendix.

In Chap. 2 we investigate in describing physical paper structures and compare them to computer applications that are based on paper metaphors. We aim to find out differences between structure or behavior of physical paper and the analyzed applications. We pick some of the raised questions to formulate them as hypotheses in Chap. 3. This concludes Part I.

The hypotheses are of such kind to be tested by usability evaluations on an application prototype. Chapter 4 presents the implementation of the prototype. We will use the application for a usability test that we describe and evaluate statistically in Chap. 5. Finally, we conclude our research in Chap. 6 and discuss open questions and potential future investigations.

Chapter 2.

Analysis

„Wenn nun plötzlich all die kleinen Pausen wegfallen, zum Eintauchen der Feder, zum Nachfüllen des Halters und zum Ablöschen der Tinte, wann um Himmelswillen soll man dann noch Ideen entwickeln?“

(Friedrich Nietzsche criticizing ballpoint pens)

2.1. Overview

This chapter discusses our analysis of real offices as well as of relevant applications. It provides the foundation of our hypotheses. The idea is to analyze paper and its appearance in collections in great detail and compare it to paper metaphor-based applications to point out the differences. We investigate finding out positive or negative effects these properties have for organizing and finding information. In this chapter, we raise many questions. Some of them are taken as our hypothesis (Chap. 3) and used for further investigation (Chap. 4 and 5).

The discussion on physical paper structures is divided into three subsections. Firstly, we put our work in context of previous research about how people organize paper (Sect. 2.2.1). Secondly, we show our observations (Sect. 2.2.2) and raise them onto abstract level, developing a taxonomy for bindings (Sect. 2.2.3). We also point out influences of constraints and emerging metainformation on paper-based structures (Sect. 2.2.4).

The second part of this chapter describes applications that aim completely or partly to implement paper behavior (Sect. 2.3.1). For a closer analysis, we take spatial hypertext applications, a subdivision of paper metaphor-based applications, and compare them to our real world observations (Sect. 2.3.2). This comparison delivers relevant open questions. We will answer some selected ones in Chap. 5.

2.2. Structures Built with Real Documents

2.2.1. Related Research

This section gives an overview of related work on paper-based structures in the real world (see also Atzenbeck & Nürnberg, 2005b, 52–54). It is an interesting observation that there is only little related research published. Whittaker & Hirschberg (2001, 150) mention the fact

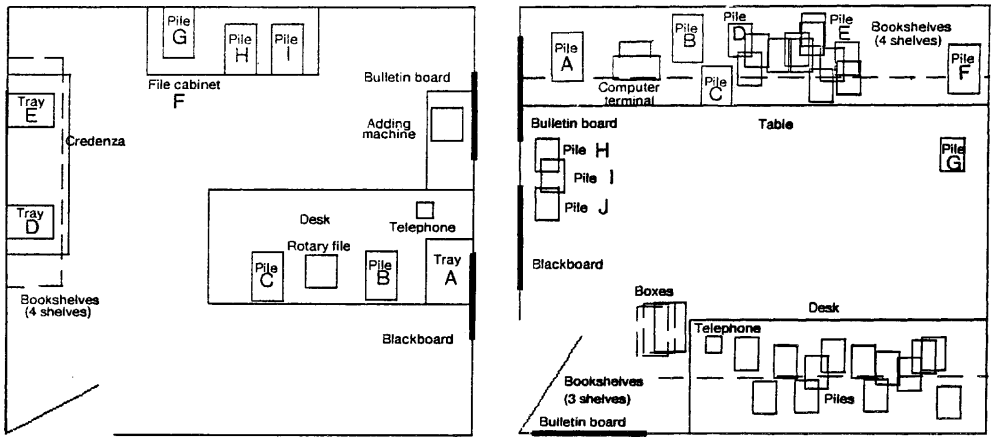


Figure 2.1.: Maps of two offices by Malone (1983, Fig. 1 and 2) – © 1983 ACM, Inc., used with permission

that "[d]espite the past importance of paper archives in office work, there are still relatively few studies of their nature and function"[1].

One article to which Whittaker & Hirschberg (2001) refer is written by Malone (1983) and cited in various other publications. Malone describes how people organize their desks. At first glance, it is similar to what we did; however, the classifications are different. One aspect is the abstraction level. Malone describes paper structures in less detail. Figure 2.1 depicts drawings of two offices. They are taken from Malone's article. All document piles are right-angled, even the right drawing, which shows a room "filled with loosely stacked piles of mixed content" (Malone, 1983, 103). It can be assumed that piles or documents had various rotation angles, even though this is not represented in the drawings. In comparison, Fig. 2.2 to 2.4 on pages 35–37, which we analyze in detail later, show pictures of the real world, including all those nuances.

Malone does not argue *why* he used this specific level of abstraction. His choice seems to be arbitrary. This is not necessarily bad. It depends on what the describing person wants to show. However, the recipient needs to understand that the description hides details that existed in the real office. In contrast to Malone (1983), we focus on a lower abstract level and therefore richer details in our descriptions. However, this is also an arbitrarily chosen abstraction level.

Beside the level of details, we also differ in the usage of terms; partly, because we distinguish between a larger number of different types. Borrowing terms from Tsichritzis (1982), Malone classifies structures as *file* and *pile*:

> "*[F]iles* are units where the elements (e. g., individual folders) are explicitly titled and arranged in some systematic order (e. g., alphabetical or chronological). In some cases, the groups themselves (e. g., entire file drawers) are also explicitly titled and systematically arranged; in other cases, they are not. [...] In *piles*, on

[1] In this context, Whittaker & Hirschberg (2001, 150–151) refer to Cole (1982), Kidd (1994), Lansdale (1988), and Malone (1983).

> the other hand, the individual elements (papers, folders, etc.) are not necessarily titled, and they are not, in general, arranged in any particular order." (Malone, 1983, 106)

As argued in Sect. 1.2.2, we focus mostly on cohesion, not coherence (meaning). Therefore, titled elements or ordered sequences are not relevant for us. For example, Malone's descriptions (Malone, 1983, 106) of an unordered bookshelf, which he calls a pile, or sorted shelves in a library, which he calls files, would be the same for us. A similar difference can be experienced when he writes about his interviews with those who worked in the observed offices: "The interviewees were asked to give the interviewer 'a tour of their office,' explaining what information was where and why it was there. [...] The interviewer also attempted to focus the discussion on the *information* present in the office, not on physical artifacts such as staplers, blank paper, and so forth" (Malone, 1983, 100). Except for some few cases, we describe observed structures without asking what information they contain or the reason they are structured in a certain way.

Malone (1983, 110–111) mentions four different reasons for creating piles. Firstly, it may be too difficult for the person to create folders or binders according to the desired structure, especially when multiple structure levels are intended. Secondly, the user may have difficulties in creating categories that would be needed for easy retrieval later. Thirdly, a person may want to be reminded by the document and therefore keep it visible on the desk instead of filing it. Finally, a person may want direct and easy access to a document, for example, because it is frequently used.

Lansdale (1988, 56) agrees with Malone about the problems that may occur when classifying documents. However, even though piles do not classify their containing documents, they still may help a person in finding information. For instance, knowledge workers may find documents by remembering the location where they placed them previously, or can possibly reduce the area where to look for them. Also, browsing through piles may let them find information by recognizing documents' visual appearance. Some implicit metainformation was attached automatically during creation, for example, more recent document appear usually more on top of a pile than older ones. We investigated *emerging metainformation* and will discuss it more detailed in Sect. 2.2.4.

Malone also uses the term *finding* versus *reminding*. He points out that filing information in order to support better finding may not always be worth the time (Malone, 1983, 104–105). In fact, more recent research has shown that pilers access more documents than filers, mainly caused by "large overheads for constructing, maintaining, and rationalizing complex systems" (Whittaker & Hirschberg, 2001, 161) when filing. The same survey also reports statistical significance in filers preserving larger original archives and having more preserved information left after cleaning. The time aspect of organizing information is also important for our research. For example, it relates to our observation about paper structures affected by a person's lack of time, as we discuss in Sect 2.2.2.

In addition to finding, reminding is "an equally important function of most desk organizations"[2] (Malone, 1983, 106). We will describe several types of paper-based structures in Sect. 2.2.3, which support a high level of reminding, such as covering or open placed individual pages.

[2] Malone (1983, 107) made "a rough analysis of the piles visible in the photographs[, which] suggests that of those for which uses could be determined from the interviews about 67 percent were piles of things to do, presumably placed there to serve as reminders."

Beside statistical tests on experiments that are based on filing and piling, Whittaker & Hirschberg (2001, 151) discuss also "[t]he *uniqueness* hypothesis[, which] concerns the reasons why people retain paper information". As mentioned earlier, we do not consider *why* office workers create certain structures, except for marginal parts in our discussion. Therefore, this topic is of less interest for our work.

Cole (1982) classifies filed information as "action information", "personal work files", or "archive storage". The first term refers to information that is used frequently by users. The cognitive model is "predominantly spatial" (Cole, 1982, 61). This compares best to Malone's piles. Interactions with personal work files are less frequent. The user is aware of location and classification of information. Finally, archive storage holds information that is exclusively classified. Spatial information does not play a dominant role anymore. Personal work files and especially archive storage are mostly files in Malone's terminology. Cole (1982) does not describe the appearance of paper structures. Therefore, it is not of prime importance for our research to discuss this further.

Kidd (1994) points to the need of office space for knowledge workers for placing paper even on the floor. In Malone's terminology, Kidd describes piling behavior. He further discusses types of knowledge workers and their behavior. In our work, however, we will focus on paper and paper structures rather than on their creator.

Some other research directions discuss note taking (Khan, 1994) or forms (Tsichritzis, 1982). Even though it is interesting and important for understanding how people work with paper, it goes beyond our focus. This is similar to research that is done in the field of annotation and hypertext (Marshall, 1997, 1998). Even though we mention annotating structures later in our analysis, we still view the physical structure itself, not the written content.

2.2.2. Our Observations

Our observations of real world structures are based on several use cases. In the following we discuss Britta's office, which is representative for most other use cases.[3] Britta is a secretary at a university. Daily, she receives much information to be processed.

We describe the discovered structures that involve paper. As argued above, we focus on the *physical* structure. Discussions about *why* they were created may help to understand more complex matters, but is not of prime importance to our work.

Structures

Britta's main workflow relays essentially on paper. Since paper is part of the real world, it also is subject to real world forces, such as placement within certain dimensions (spatial aspect), gravitation, friction, changing appearance related to light, distance, position, or size, among others. Structure depends on the possibilities of the chosen media. In any case, there are limitations.

Britta uses specific locations, which are assigned to specific topics. For instance, she uses different areas on a shelf for placing student papers. Additionally, she puts student papers on the floor in front of the designated location on the shelf, as shown in Fig. 2.2. This happens when she does not have the time to put them to the right place immediately. In the terminology of Malone (1983), those would be called "piles". However, since we focus on

[3] We would like to thank Britta an all other involved people for answering patiently the many questions we had about their paper work and workflows.

Figure 2.2.: Unstructured heaps of student hand-ins on the floor

their appearance, we call them heaps. Even their internal structure depends on time: The lowest paper came in first. This observation is identical to the one given in Lansdale (1988, 56). Nevertheless, Britta calls the internal structure of a heap "unstructured", mainly because the structure was initially not intended. She does not consider emerged metainformation as "structuring". The documents' locations are a necessary prerequisite for creating heaps and to be recognized as belonging together. Additionally, their place in front of the shelf indicates to which class or category they belong to.

The "unsorted" student hand-ins that were placed on the floor can be recognized as groupings. Figure 2.2 shows one big heap on the left hand side. Originally there were at least two well shaped stacks, but they became covered by randomly placed folders and turned into a single heap.

The middle part of the picture shows also some stacks. However, even those seem to morph from well placed ones to amoebic looking shapes. This happens usually when new documents are not positioned precisely onto a stack. The stack becomes more sloppy.

This shows that time plays an important role in the way paper structures evolve. If there is no time left to put paper to the place where it actually belongs to, it has to be placed somewhere else. Apparently, the less time there is for structuring the more "chaotic" a structure becomes.

Beside the time factor, there are also other constraints in paper structures. For example, Britta divided information that would belong together and put the two parts to different places. We noticed this with paper work for students' grants. Originally, there was one binder that contained all information. After some time, Britta had to create a second binder; however, there was no space left next to the first binder on the shelf. A larger portion of the shelf would have to be restructured in order to receive vacant space to put both binders side by side.

This example shows constraints at two levels: Firstly, the possible amount of paper within a binder is limited. A second binder had to be created after the first one was full. This first step already requires some restructuring. Secondly, the bookshelf is also limited in space. Because the part where the first binder was located was occupied, the new binder had to be placed somewhere else. In the above described example, Britta had no time to restructure

Figure 2.3.: Britta's "center of work"

the book shelf immediately. This draws a dependency of paper structure constraints to the available time for structuring.

Figure 2.3 shows Britta's "center of work". It includes computer keyboard and monitor. Britta pointed out specific locations where she puts information:

- Urgent tasks are put between keyboard and monitor. They are not processed at that time. However, the location makes them visible for Britta at any time she sits in front of the monitor. The space is small and can take only a small number of documents. Britta removes them once they are not longer used or not urgent anymore.

- Occasionally, Britta also puts very urgent and important documents on top of the keyboard. She uses the keyboard very often. Paper that covers the keyboard prevents her from typing. Britta uses this to be reminded of the covering documents. She is forced to take them off before she can use the computer. There are two main differences compared to the place in between keyboard and monitor: Firstly, the documents on top of the keyboard require an action in order to use the computer. Secondly, the keyboard cannot carry as many documents as the space behind, because its keys would be pressed down and may lead to unintended results on the computer.

- Frequently used lists, for example, student group listings, are placed next to the monitor. Britta can reach them easily. However, in order to select the desired list quickly, this space needs to be kept free from other documents.

- Britta uses sticky notes to note important and short information. She puts them onto the monitor's front side. Because of their vertical adjustment, they usually are not covered by other documents. Additionally, they are placed in a way that helps reminding Britta of their content. They also can be found on other spots, mostly put within the main working area. The chosen places let them be visible to Britta most of her working time. They usually are completely visible, for example side by side, as shown in Fig. 2.3.

Figure 2.4.: Britta's workspace

We could see at Britta's desk that documents that have to be processed urgently or are important are placed at areas where work is frequently done. Very urgent matters may be even placed in a way that forces Britta to remove it within a short period of time, for example, on top of the keyboard. Britta puts paper, which she wants to write on to right of the keyboard, where conveniently writing is possible.[4]

Britta puts documents that are not urgent, but still needs to be processed, on both sides of the "central work space", that is, where the keyboard is located. On the left hand side are documents that she uses occasionally. They are spread to provide a better overview of their content. The area to the right of the "center" is depicted in Fig. 2.4. At the right lower corner of the picture is a stack mostly of printed e-mails that Britta needs to work on. There is no specific ordering among them. Britta browses through them from time to time, pulls-out one, and continues working on it after putting it to the work space area. The stack is sloppy. This helps to provide some information about what documents are located there without touching the stack.

We experienced at Britta's office that documents that are unimportant or less urgent are put further away from the central work space. Archived documents can be found primarily on shelves and the window ceiling. This includes student papers, regulations, forms, etc.

Some spots on the desk or on the floor are used to store documents temporarily that will be archived later. For example, Fig. 2.2 shows student hand-ins on the floor to be filed later. It is not their "final" archive. Constant restructuring is necessary; however, Britta did not have time to file them yet. This shows the temporal aspect of spatial structures in active work environments.

Time influences even documents in archives. They may be there only for some period. After that they might be restructured, become parts of other structures, be thrown away, or become decomposed by aging.

[4]The side may vary among right-handed and left-handed persons. Britta writes with her right hand, therefore the right side for the current work is most convenient for her. At the same time she is able to use the mouse with her left hand.

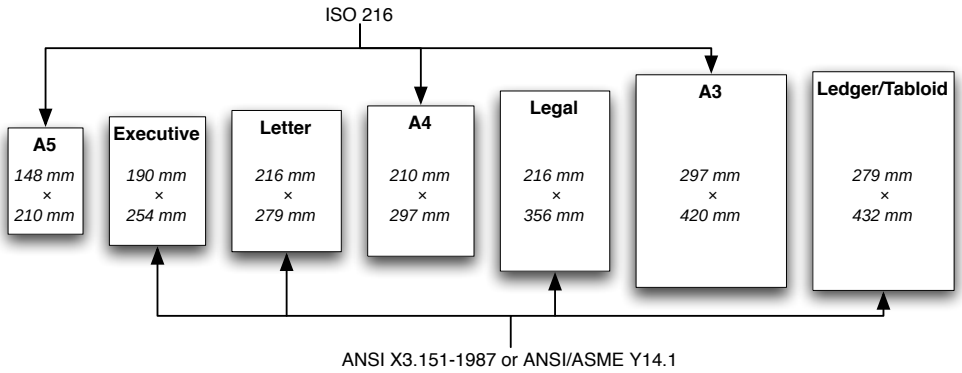


Figure 2.5.: Selected ISO and ANSI paper sizes (see Kuhn, 1996; Bergman & Hastings, 2002; EDS Inc., 1997)

Main Paper Types (Sizes)

This section describes what paper types Britta uses for her work. We will focus exclusively on their size (length and height), not on other attributes, such as paper quality or color nuances. We also do not discuss related devices such as pen, stapler, or printer.

ISO A4 Paper The ISO 216 standard defines paper sizes (Kuhn, 1996). It is used by most countries worldwide. It describes ISO A, B, and C series of paper formats. This thesis is typeset for ISO B5 paper size. However, we focus on ISO A-sized paper, which we assume to be the most often used paper size among the ISO 216 defined series. The basic format is ISO A0, which covers 1 m^2. The ratio of width to height is $1 : \sqrt{2}$. This is the only ratio which stays the same if a paper is cut along the middle line of the $\sqrt{2}$ ratio side. This ratio reaches back to the German scientist Georg Christoph Lichtenberg (1742–1799), who probably was the first who noted the advantages of this relation. Later, Porstmann (1918) discusses also aesthetic aspects of this format, that other formats do not have:

> „Es kann keiner geometrischen Form eine besondere Schönheit untergeschoben werden. Es findet jedes Format seinen ästhetischen Vertreter. Es ist einzig Gewohnheit, wenn wir dies oder jenes Format in einer gewissen Grenze auswählen, falls wir auf sogenannte Schönheitsgründe aufbauen. Nur wenn eine Form gestreckte Zwecke gut erreichbar macht, ziehen wir sie – unbewußt vielleicht ästhetische Momente unterschiebend – als ausgezeichnet vor. In diesem relativen Sinne der Schönheit der einzig anwendbar ist, ist unsere gefundene Form ebenfalls ‚schön', was man sofort empfindet, wenn man nicht *ein* einzelnes Format, sondern die durch forgesetzte Halbierung daraus gewonnene Reihe betrachtet. Alle anderen Ausgangsformate liefern dabei zweierlei Formen, nur das mit dem Seitenverhältnis $1 : \sqrt{2}$ liefert eine Reihe ähnlicher Formate, eine einzige Form. Diese Harmonie empfindet man als schön." (Porstmann, 1918, 201)

Paper formats with this ratio became a DIN standard in Germany. Later it was adopted as an international standard (ISO) and recommended by the United Nations (Working Party

on Facilitation on International Trade Procedures, 1981). However, it is still not adopted by some countries, for example, the USA, which used to have even two official standards for ISO A4 equivalent paper size: 8.5 in × 11 in (Letter format) for the U. S. industry and 8 in × 10.5 in for the U. S. Federal Government (Dunn, 1972, 6–7). The paper standard for the U. S. government changed in 1980 to 8.5 in × 11 in (Kuhn, 1996). Some common paper sizes in the USA in comparison to selected ISO formats are depicted in Fig. 2.5.

Beside the fact that using non-standard paper sizes may capture the reader's interest, it also has some drawbacks, which are discussed in Dunn (1972, 10–33). Most of them come back to economical disadvantages, based on the need for special storing facilities or machines. ISO A paper sizes can also be cut into two parts, which creates two other ISO A paper sizes without any waste.

This discussion leads to the question about which paper sizes would support workers' structuring tasks most conveniently? Whereas the ISO standard with its $1 : \sqrt{2}$ ratio has some aesthetic and practical advantages, we could not find research results with similar conclusions for comparable paper formats, such as Letter, Legal, or Executive format used in the USA.

We assume ISO A4 the most common paper format in Europe, therefore, we were not surprised that most of the paper in Britta's office is A4 format. All documents printed on the departmental printer have this format.

Sheets may be part of structures at various levels, for example, loose or bound. For the latter one, they can be structured inside binders, folders, or transparent sheets, or be clipped. These techniques shift loose A4 paper to the next structure level (see also discussion on page 47). Most A4 paper in Britta's office are organized as stacks or inside binders.

The use of specific media types depend on their availability. This is related to introduced standards and available machines, such as printers.

Sticky Notes Britta uses sticky notes for handwritten information. She uses two different sizes: 50 mm × 40 mm or 75 mm × 75 mm. Neither of them follow the ISO 216 ratio of $1 : \sqrt{2}$.

There is a supply tray for the squared sticky notes, which can be found on a central place within the main working area, visible in Fig. 2.3. Even though they look similar to annotation aids, Britta mostly does not use them as such. For example, some of them are attached to the monitor. They do not hold information about the monitor or related to it. The information even does not necessarily have to do anything with Britta's work on the computer. As discussed on page 36, she uses this place to be reminded of important matters that she wrote on the sticky notes. There is also a minor risk that they will be accidentally covered.

2.2.3. Structure Descriptions

In the following sections, we analyze our observations. Parts of our discussion will influence the implementation of our prototypic application (see Sect. 4.2.3). Additionally, we introduce informally the used terminology.

Binding Types

We applied three scales on our observed structures that express their level of overview, division, or annotation. Table 2.1 shows their impact for overview, division, or annotation. The scale has four possible values ($++$, $+$, $-$, $--$). More pluses mean a higher support, minuses less support. "N/A" is used in cases where an attribute can not be applied uniquely.

Structure Type	Overview	Division	Annotation
open placed objects	++	++	−−
cluster with overlays	++	++	−−
stretched stack	+	+	−−
heap	+	−	−−
stack	−−	−−	−−
folder	−	−	−−
drawer	−−	−	−−
binder	−	−−	−−
stapled	−	−−	−−
book	−	−−	−−
blotting pad	N/A	++	−−
zigzag stack	−	++	−−
pull-out	−	+	++
folded document	−−	N/A	++
sticky note	N/A	N/A	++
code (on objects)	N/A	++	N/A

Table 2.1.: Tendencies of overview, division, or annotation of different paper bindings

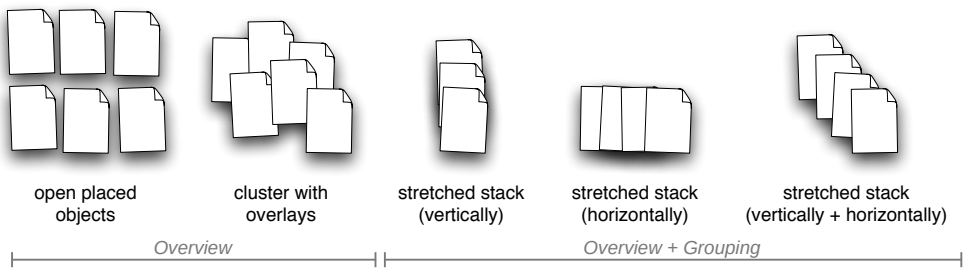


Figure 2.6.: Overview structures

The described binding types are not nearly complete. They are a selection based on our observations. There are many more binding types. For instance, scripture roles are not used commonly in the western culture nowadays, whereas they were common in classical Egypt or Judaism.

Overview Support This section describes structure types that have the primary goal to provide an overview of the content, even though there is also a grouping aspect. Figure 2.6 depicts different kinds of this category. Table 2.1 points out their good overview capabilities.

Open placed individual objects show all or most of their viewable content. This is, for example, for single sheets their surface, or for stacks their viewable part, such as in most cases their first page. Objects are facing the recipient for easy reading. These structures are very space consuming, depending also on the size of the objects. If the available space becomes too small, the objects may be moved closer to each other until they overlap. This structure can be recognized as cluster with overlays. Its members have a larger viewable area

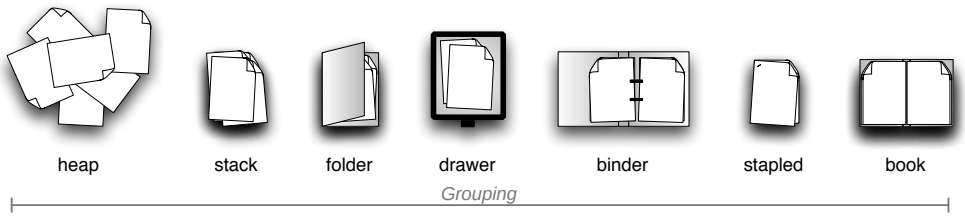


Figure 2.7.: Grouping structures

than a heap (which we will discuss later) and align toward the user with only little rotation. All documents are at least partly visible.

Open placed objects or cluster with overlays can be used to cover objects, such as a computer keyboard, in order to be reminded of processing them. We discussed an example on page 36.

Variations of the discussed overview structures are stretched stacks. Three variants are depicted in Fig. 2.6: Vertically or horizontally stretched stacks as well as stacks that are both, vertically *and* horizontally stretched. In all cases, they increase the overview of their content in addition to the primary task of dividing its parts and grouping the whole. Their orientation towards the recipient supports its grouping force, because documents are equally aligned.

Grouping and Division Support Table 2.1 notices the level of division at its second column. In our terminology, "division" is the opposite of "grouping". Bindings with weak division, for example, binder or book, have strong grouping forces. Figure 2.7 depicts group forcing structure techniques for paper. A heap[5] is a sloppy pile of loose documents. A heap may have two non-overlapping documents at the same horizontal layer. A stack is similar to a heap; however, its contained documents follow tighter bounds. A stack appears sharper and better defined in its occupied area. Normally, each layer is taken by exactly one document. However, sloppy arrangements can be experienced also on stacks, as indicated in Fig. 2.7. Documents can easily be pulled-out. This is easier to manage for stacks, because heaps have sloppier bounds, which would force the pulled-out paper to move far outside. In cases where the heap has very sloppy bounds, a pull-out may be even not recognized as part of the heap.

Folders and transparent sheets support paper collections. They define an exact upper and lower bound of a stack. Those two boundaries are connected and recognized as one single folder or transparent sheet. They are very limited in capacity. Most of them cannot host stacks higher than few centimeters. Pulled-out papers are still possible; however, in a more limited way, because one side is closed by connecting front and back cover.

Drawers and trays have sides. Basically, a drawer or tray is a box with an open side. The document inside has to follow these boundaries. There are different types of drawers or trays. Pull-outs are possible with any kind of them, but depending on the their location, they may have to follow partly the tray's or drawer's side up, as depicted in Fig. 2.8.

Binders add a new quality of grouping documents: The grouping forces are much stronger than at the previous discussed types, for example, open placed objects or heap. In order to

[5] It has been suggested that this structure could also be named "disoriented pile". However, this would not represent the larger area which a heap occupies compared to a stack. We asked English native speakers about their understanding of the used terms. They were satisfied with our terminology in this context.

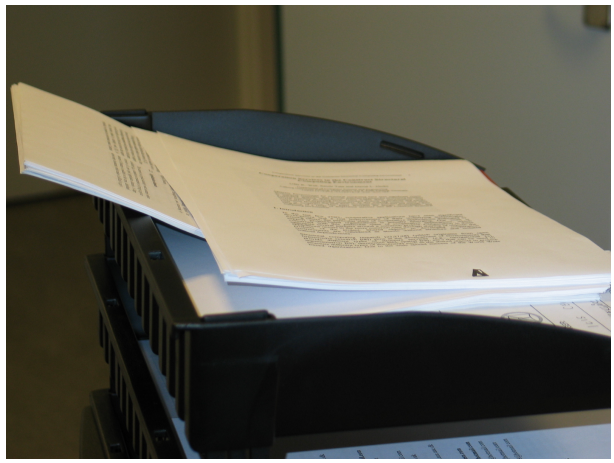


Figure 2.8.: Tray with paper stack pulled-out

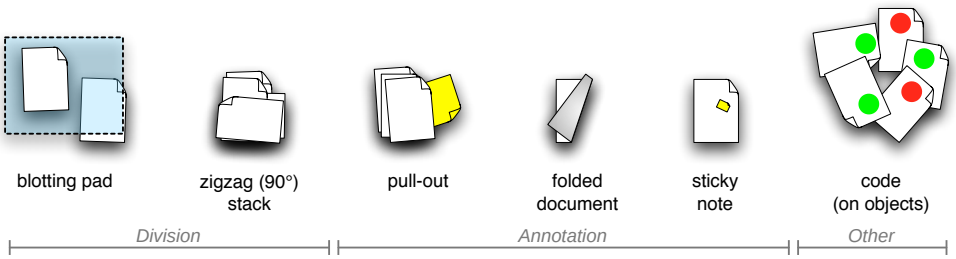


Figure 2.9.: Division, annotation, and other structure types

change the paper sequence, the binding mechanism has to be opened. Punched sheets usually cannot be moved while they are held by a closed binder mechanism. They may be moved a little, depending on the free play of the punched holes and the binding mechanism part that goes through them.

Whereas binders can be restructured easily after opening their binding mechanism, it is more difficult for stapled documents. It is possible to remove the staple, either with a tool or by hand. However, the location where the staple was, will show some damage in any case. Because of this, a stapled stack is usually a more final grouping than a binder, even though the paper of a stapled stack can be moved further to the outside (rotated), than it is possible with paper held by binder mechanisms. There is the restriction that the paper are fixed at the spot of the staple. Also, the staple allows sheets to rotate only to some degrees without damaging them.

The most final structure among those depicted in Fig. 2.7 holds books. They are bound. It is not possible to reorder its pages without destroying the book, or to move a page outside, not even slightly. Books bind shaped stacks of sheets very strictly and do not allow variations without damage, once bound.

The first two iconographic structure representations in Fig. 2.9 demonstrate techniques, that

Figure 2.10.: Colored binders

primarily are used to divide sets of documents. The first example depicts a blotting pad, that covers a document and has another document on top. We have experienced that sometimes frequently used documents are put underneath, for example, phone lists. They can be accessed easily without getting mixed with the documents on top, where usually current work is placed.

The second example for division supporting structures are zigzag stacks. They are often built of other stacks, such that every second stack is rotated approximately 90 degrees. This allows to pick up any sub-stack easily.

Annotation Support Annotation supporting structures can be found frequently in paper-based office work. An annotation can be some text written on a paper as well as the paper itself, marking a specific position. As already mentioned on page 34, we do not focus on written annotations. The middle part of Fig. 2.9 displays three types of paper annotations. The first example shows a pulled-out paper marking a position within a stack. Pull-outs can be applied for most of the other mentioned structure types as well. They mark a specific position.

As mentioned on page 41, some bindings make pull-outs difficult or impossible to create, for example, books. A book page cannot be pulled-out without damaging the book. However, for most of them it is possible to fold a page in a way that its height increases, as depicted as the fourth iconography in Fig. 2.9. The increased height will mark its position within the grouping.

A pull-out can also be simulated with an additional document, for example, an individual sheet of paper. This makes it possible to apply this kind of structure conveniently to books or binders as well. However, the additional document may not be part of the original binding and possible not recognized as such by the recipient, for example, a white sheet of paper that is used as a bookmark. A binding can also mark itself, for example by being folded, as depicted in the previously mentioned figure.

The next example in Fig. 2.9 depicts a small paper sticked on another document. The small paper annotates the larger object. The other way around would not be recognized by the user correctly. We described some examples for sticky notes on page 36.

Other Structures Beside those structure types explained above, people also use color or other signs to mark documents that belong together, as depicted at the last example of Fig. 2.9. Figure 2.10 shows a real world example of a secretary's shelf. She uses white binders to indicate matters of university staff members and red ones for other target groups, for example,

Structure Type	Grouping Aid	Binding Dimension	Thickness Growth
open placed objects	abstract	area	add
cluster with overlays	abstract	area	add
stretched stack	abstract	area	add
heap	abstract	area	add
stack	abstract	area	add
folder	2-sided	area	add
drawer	1-sided	area	none, add if $> x$
binder	2-sided	line	none
stapled	2-sided	spot	add
book	2-sided	line	add
blotting pad	1-sided	area	add
zigzag stack	abstract	area	add
pull-out	abstract	area	add
folded document	abstract	intrinsic	double
sticky note	abstract	area	add
code (on objects)	abstract	intrinsic	add

Table 2.2.: Binding characteristics

students. Whether a person recognizes the binder colors as a grouping indicator depends on various aspects. For instance, a person would need to know the semantics of the used colors, such as that the binder's color relate to different target groups. The location of the binder plays an important role. A recipient may not recognize them as being related if they are put too far apart.

Bindings Characteristics

In this section, we will categorize bindings according to their appearance or behavior. Table 2.2 gives an overview of our observations. The core of this section was presented at the ACM Conference on Hypertext and Hypermedia 2005 and published in Atzenbeck & Nürnberg (2005a, 63–64).

Grouping Aids Some of the observed structure types have add-ons that support grouping. Everything that is not one of a structure's grouped objects is classified as grouping aid. As shown in Tab. 2.2, grouping aids can be one-sided, two-sided, or abstract. The last one appears whenever a group of objects is recognized as a group, but does not have explicit bounds. Examples of abstract grouping mechanisms are stack or open placed objects. Drawer or blotting pad consist of one single layer on or underneath which the related substructure can be found. Most well-known are two-sided grouping aids, such as folder, binder, or book. They have a cover on two sides: on top and on bottom. Also, the staple of a stapled document stack can be considered as two-sided, since it is visible from two sides.

Binding Dimensions Binding dimensions classify the behavior of elements that are applied to the binding. Those can be area, line, spot, or intrinsic. For most aforementioned bindings, there is an *area* in which the object may be moved around. For example, a single page that is part of a heap may be moved within a designated area, without leaving the heap. Once it is moved too far, it will not be recognized as an object of the heap anymore. Also a drawer and a folder have binding areas, because they allow paper to move within an area freely, even if parts of the paper hang over a drawer or are pulled outside a folder.

There are bindings that support a *line* as binding dimension, along which objects are bound and along which a user may possibly turn documents. Examples include book or binder. In these cases, pages have to follow a specific path when they are turned. A *spot* appears only on stapled objects. It allows more freedom than a line binding dimension. The angle when turning a page may vary. However, the binding dimension of a stapled stack may also be interpreted as line.

A line can be seen as a spot with zero rotation. The same binding dimension may have different ranges of possible angle ranges. For example, a pinned paper can be rotated 360 degrees, whereas a stapled stack (assuming it is interpreted as spot) only has a few degrees free play.

Finally, *intrinsic* binding dimensions appear on folded or coded objects. The binding is intrinsic and implicit. Coded objects are bound together, because they are grouped by having the same color or symbol, not because of a certain location. However, space plays some role. For instance, coded objects that are too far apart will not be recognized as members of the same "color space".

Thickness Growth Behavior An important behavior of paper structures is how they change in thickness when new objects are added. Many of them just add the height of the new object to the already existing one. For example, if the height of a stack is 10 cm and 500 single pages of a total height of 5.1 cm are added, the new height of the stack will be 15.1 cm.

This is different for drawers. A drawer has a fixed height, even when it is empty. However, if the height of the objects inside a drawer extends the drawer's original height, the total height just adds as with stacks.

Folding an object doubles its height.[6] The height of binders does not grow. Binders have a specific width, regardless of whether they are empty or full. Whenever the amount of paper that can be put into a binder exceeds the capacity of the binder, a new binder has to be created. An additional structure level gets pushed into the existing structure (see discussion on page 47 or Atzenbeck & Nürnberg, 2005b).

Bindings, including those that grow when new objects are added, have a maximum height that is sometimes not precisely defined or dependent on various attributes. For example, a stack will fall over after it has reached a certain height, or a book that is very thick, for example, 50 cm, would not be useful anymore. Most office workers have the experience of seeing what is useful or possible and what is not. A secretary who has already a quite high stack on her desk will start a second one or even divide the one into two parts to prevent it from falling over.

Slot Types, Structure Dissolution, and Automatic Conversion Table 2.3 shows possible combinations of structure types that are discussed in this dissertation. The table has draft

[6] Not all objects can be folded, for example, it is possible with single pages, but not with binders.

X (↓) may bind Y (→)	single page	open placed obj.	cluster w/overlay	stretched stack	heap	stack	folder	drawer	binder	stapled	book	blotting pad	zigzag stack	pull-out	folded document	sticky note	code (on obj.)
open placed objects	•		•	•	•	•	•	•	•	•	•		•	•	•	•	•
cluster with overlays	•					•	•	•	•	•	•		•		•	•	•
stretched stack	•			•		•	•	•	•	•	•			•	•	•	•
heap	•					•	•	•	•	•	•				•	•	•
stack	•			•			•	•	•	•	•			•	•	•	•
folder	•			•			•			•	•		•	•	•	•	•
drawer	•			•			•		•	•	•		•	•	•	•	•
binder	•						•			•	•			•	•	•	•
stapled	•									•				•	•	•	•
book	•													•	•	•	•
blotting pad	•	•	•	•	•	•	•	•	•	•	•		•		•	•	•
zigzag stack	•			•		•	•	•	•	•	•			•	•	•	•
pull-out	•					•	•	•	•	•	•				•	•	•
folded document	•					•				•				•		•	•
sticky note	•																•
code (on objects)																	•

Table 2.3.: Structure types and potential related structure types

status; other interpretations are possible and there are exceptions that can be found in the real world. We call potential sub-structures *slot types*, in relation to slots that can be filled with designated structure types only.

Bindings may bind objects other than single pages. For example, a zigzag stack can be built of books as well as of several individual stacks of stapled paper. Some other combinations are not possible, for example, binders cannot be bound as a book, even though their content may be used for that.

Combinations may also depend on other attributes, for example, the thickness of objects that are to be bound. For instance, a common office does not have the facility to punch holes in a book that is 8 cm thick. Therefore, this book cannot be added to a binder's mechanism. On the other hand, it is easy to punch holes in a 2 mm thin booklet and add it to a binder.

We made the important observation that some combinations change the type of one target structure to the type of the other. We call this effect *structure dissolution*. For example, if open placed individual papers are added to a heap, they will not be recognized as parts of the previous structure anymore. They become direct members of the heap. The original binding force is completely gone so that the objects become individual objects when given to the target structure.

Another aspect is that some types are restricted to what objects they can take because of their own size. For example, a folder usually cannot contain a sloppy shaped heap, because its size is made for aligned objects, such as stacks. A similar example would be a drawer.

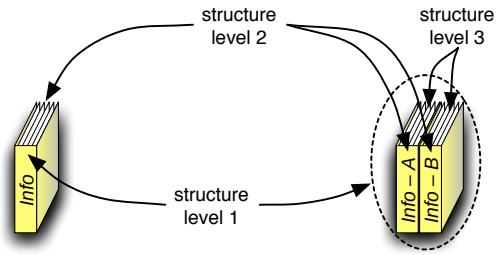


Figure 2.11.: Structure pushed to the next deeper level

A heap turns automatically into a stack when it is put into a drawer. This is an *automatic conversion* of a structure type. Its structure does not dissolve. The binding still exists, but is now of another type.

Some behavior may also cause automatic conversion of structure. As already mentioned, a stack may fall over when it becomes too tall. This is a conversion from one structure type into another, initiated by adding new objects to it.

Cases of structure dissolution or conversion are not explicitly marked in Tab. 2.3 yet. Both are special cases of structures that may be brought together, but cannot exist together in a contents relation.

Pushing Structure Levels Structure instances that are combined and did not dissolve create an additional level of structure. For instance, consider a binder that cannot hold more documents. A new binder is taken, which receives the remaining documents. We described a real situation where this became necessary on page 35. The two binders are marked as grouping, for example, by color or label. Figure 2.11 depicts an example: Originally, there was one yellow binder with the label “Info”. It held various single pages. The binder is part of structure level 1, but at the same time it structures its content (level 2).

The content then is split up, and cohesion of both binders becomes the the first structure level, for example, equal size, color, or label. In our example, the label partly varies. In addition to the previous “Info”, the letter “A” or “B” were added to indicate the appropriate relation. The second structure level can now be found at the level of the individual binder. It still structures and binds its content. The single pages within each folder are at the third structure level.

The original structure levels were pushed to the next higher levels. A new structure layer became inserted. This may have effects for finding information. For instance, a person who has two binders similar to the ones depicted in Fig. 2.11 can cut down the amount of pages to browse (structure level 3) in half by selecting the appropriate binder (level 2) of both binders (level 1). This is in comparison to one single large binder that would hold all documents.

It must be noted that the balance between the size of a sub-structure and the number of structuring entities is important. The used structure types and the purpose also play an important role. For example, it is possible to hold 5,000 sheets of paper within a special binder. However, this is not convenient to browse. We could split up the content of this binder into 10 binders, each containing 500 sheets. By doing this, we created a new structure level, as described above. However, we also could use 100 binders that can hold 50 sheets each.

People have developed binding devices such as binders for different sizes. They support various tasks. However, there is an upper bound above, which a binding would become inconvenient for a particular situation. We assume that experience plays an important role in deciding which binder to use.

2.2.4. Constraints and Emerging Metainformation

In the previous discussion, we mentioned aspects of limitations, and emerging behavior of paper and real world bindings. We argue that those carry implicit metainformation that may be helpful for finding information. The most obvious aspect of emerging behavior is sloppiness.

Figure 2.4 on page 37 as well as many other pictures of paper structures show sloppily aligned stacks of papers. Perfectly shaped stacks would look unnatural in an actively used work environment. Any document that is moved changes its angle, even if only slightly. We argue on page 35 that time plays an important factor in evolving paper structures. Neatly aligned stacks take longer to produce than sloppy ones. Office workers may be able to get an idea of which stack was created in a hurry and which one was not. People perceive hastily constructed piles differently from carefully constructed ones. This motivation has been also discussed for differently looking icons in file system browsers (Lewis et al., 2004). Furthermore, caused by individual orientations or offsets, a sloppy pile exposes parts of its containing documents to the recipient without being touched.

Other examples of emerging behavior are changing colors of documents or ink over time. For instance, most white paper receives a yellow or brown tinge when reached by sunlight, or some ink gets lighter. It can be assumed that most office workers have an understanding how an old document looks. This is also demonstrated by "aging" book simulations on computers (Chu et al., 2004, 85).

Similar to the aging process through sunlight, also other emerging changes may occur through frequent usage. Documents that are often used accumulate soiling from the user's hands. This can be experienced even with document parts, for example, an often used page within a book. Additionally, most books may be more likely open at the position of frequently used pages. If a document is neither used nor cleaned frequently, dust may cover parts of it. This also expresses that it has not been used recently.

As we describe in Sect. 2.2.2, we made the observation that location is an important implicit indicator for frequently used or urgent documents. Britta puts those closer to her workspace center, whereas documents that are to be archived are placed on the window ceiling or on bookshelves. This indicates implicit metainformation about how frequently a document is used.

This kind of implicit metainformation becomes automatically attached while a person uses documents. They are based on behavior of the *document*, for example, the fact that paper changes its color is based on chemical processes of the paper, or on behavior of its *environment*, for instance, dust falls onto documents over time.

Implicit metainformation is also supported by limitation. For instance, a binder has a certain capacity. A human would not expect much more than 600 or 700 pages within a binder that has a spine of 8 cm. Humans also have world knowledge about binding combinations. For example, office workers that are aware of books and binders know that a book with a spine of 5 cm will not be put into a binder, even though it would fit according to its size. Most offices do not have tools to punch holes into books of that size. Additionally, the binding mechanism of most binders would not allow its addition.

Figure 2.12.: Screenshots of the Open The Book application – taken from http://www.nzdl.org/html/open_the_book/demonstration.html (visited on 2006-03-18), used with permission

Emerging metainformation and limitations play a role in understanding possible connections between documents. This may help office workers to exclude certain documents or bindings, because their location or appearance does not fit the expected documents that they are looking for. We discuss constraints and metainformation with the aspect of computer applications further in Atzenbeck & Nürnberg (2005c).

2.3. Applications Based on Paper Metaphors

2.3.1. Paper Simulation

Overview

There are various research projects and spatial structure based applications that aim to simulate paper. They do this on different levels. Some projects aim to implement real objects with very high detail in appearance and behavior. This requires a detailed observation and description of the real world. Others take one ore more attributes they discovered in real life and apply them to their application.

All of those approaches are metaphor-based. They implement appearance or behavior of real objects, such as books or paper. Even though this is pushed to different levels of detail, they all are still abstractions of what there exists in reality, and therefore there will be details that are not implemented. This refers to our discussion in Sect. 2.2.1, where we argued similarly about Malone's description of paper structures in offices. We also will show that most projects do not support emerging metainformation, as described in Sect. 2.2.4.

Highly Realistic Implementations

There are some applications that represent books with 3D look and feel highly realistically. One of them is the 3Book (Card et al., 2004a,b), a prototype that aims to provide a realistic look of books as well as realistic behavior. The process of turning a page is simulated. The prototype offers additional features that are of less interest for our research effort, such as multi-page comparison, bookmarks, and annotation.

Similar to 3Book, the Open The Book project (Chu et al., 2003, 2004) attempts also to create realistic experience for reading books on a screen. Figure 2.12 shows screenshots. Chu et al. (2003) state clearly the aesthetic aspect of real books:

> "And beauty is functional: these books give their readers an experience that is richer, more enlightening, more memorable, than the prosaic, utilitarian – often plain ugly – web pages offered by today's digital libraries." (Chu et al., 2003, 186)

Various real world behavior are implemented: The user can click on a certain position of the book's edge and move the cursor. The book will open and the upper part will follow the cursor. If the book is slightly more then half open, the book will fall down after releasing the mouse button; otherwise, it will close. The same behavior is applied for turning single pages or ranges of pages. Even though "[t]he model is over-simplified" (Chu et al., 2004, 80), it is a first step of implementing gravity. There is a detailed visualization of how pages bend into the spine. The rich implementation details are also demonstrated by "the spine[, that] even bows slightly during the turning process to adjust to the pressure that the two covers exert on it, just as a physical book does" (Chu et al., 2004, 80).

The most interesting aspect for us is the implementation of an "aging option". It adds "dirt" to pages that are accessed. The second image in Fig. 2.12 shows "dirty" pages. The more often a page is accessed, the more dirt is represented. This gives the user an impression of how often a book was accessed. We call this "emerging behavior", as discussed in Sect. 2.2.4.

Paper Metaphor-Based Documents

Other projects do not aim to implement paper metaphors in great detail, but rather implement those on high abstraction levels. For example, applications that display PDF documents (Adobe Systems, 2004) come in mind. As opposed to HTML (Raggett et al., 1999), "PDF introduced the possibility of fast interactive reading and browsing because the formatting and layout were pre-computed" (King, 2004, 95). Therefore, PDF supports fixed size "pages", that give the impression of digital "paper".

This is useful in many areas, for example, projects in the field of augmented reality (AR). "An AR system supplements the real world with virtual (computer-generated) objects that appear to coexist in the same space as the real world" (Azuma et al., 2001). Examples of paper metaphor-based augmented reality systems are DigitalDesk (Wellner, 1993) or the Escritoire (Ashdown, 2004). They feature real desks on which digital documents are projected.

The left image in Fig. 2.13 depicts the Escritoire. It uses two projectors, one that targets the complete desk, a second one that provides a higher resolution area right in front of the user for better readability. The high resolution area can be experienced as having brighter documents. The user has two pens used as input devices, used for different tasks, such as moving or browsing stacks, or annotating documents.

The Escritoire supports also VNC (Virtual Network Computer), which allows to display the screen output of any other computer running VNC as single "document" on the desk. This includes window support with scrollbars inside the screen display. However, except for VNC's contents, all documents are of fixed size. There is support for bitmap images and PDF files. The latter is rasterized before being displayed.

There is a generic implementation of piles. The right part in Fig. 2.13 depicts the representations of two piles on the Escritoire. The left one has a north-west direction, that is when

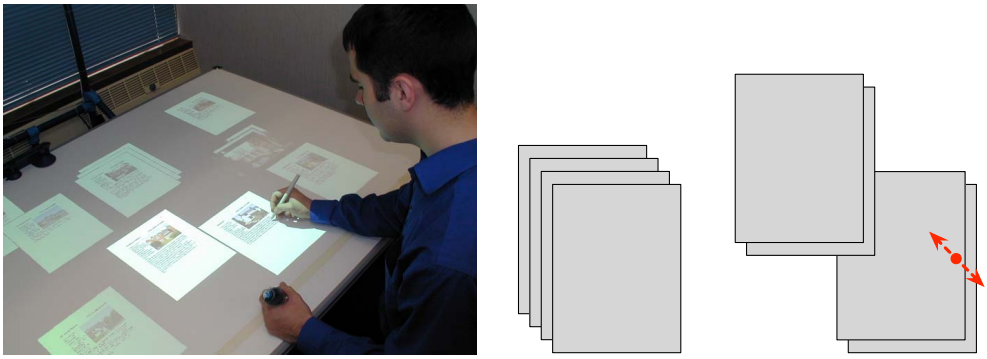


Figure 2.13.: The Escritoire (left), a pile, and browsing a pile with marked pen position (right) – picture taken from (Ashdown, 2004, Fig. 1.7), used with permission; pile representations based on Ashdown (2004, Fig. 7.1)

there are increasing negative offsets of the documents behind others. Other possible directions are north-east, south-west, and south-east. All piles are well ordered and neatly aligned. The offset is identical for all documents. Even the offset that is applied when browsing a pile (as indicated at the second pile in Fig. 2.13) is identical for all moved documents.

A document gets added to a pile when there is at least an intersection of "60 per cent of the area of the smallest item to be covered" (Ashdown, 2004, 107). The hard cut when a document gets added to a pile is an abstraction of the real world. In reality, it depends on various factors as to whether a document is seen to belong to a certain pile or not. This include closeness of surrounding piles, sloppiness, or number of documents. Also textuality criteria at the document level are considered, such as coherence (content belonging together?) or cohesion (similar layout, color, size, or fonts?). This enables people to interpret that a paper belongs "somehow" to a pile. This is impossible with the Escritoire's pile implementation.

When piles are moved, the containing documents do not change their relative position or offset. However, real piles would change when moved. Possibly it would just affect its overall angle, but possibly also its shape, for example, when a messy pile is straightened in order to be lifted more conveniently.

The Escritoire's use of a paper metaphor is obvious. However, due to the high level of abstraction, many of the discovered behavior of real paper (see Sect. 2.2.2 and 2.2.3) is not supported. This results in less emerging metainformation than there would be otherwise.

Figure 2.14 shows two screenshots of an application that uses a camera to track documents located on a desk (Kim et al., 2004a,b). As depicted at the left screenshot, the application can then be used to search for documents by title or author, or to browse documents on the document list, which appears on the left side of the application window. The position of a selected document is marked on the virtual desk on the right side of the window.

This area depicts the documents as they are captured by the camera from the top. The red surrounded document in the document list as well as at the desk area indicates that this is the one that the user was looking for. In this case it is the result of an author search. The green surrounded documents are those that have to be removed on the real desk in order to see the requested document below. In the application they are moved slightly to the side in order to

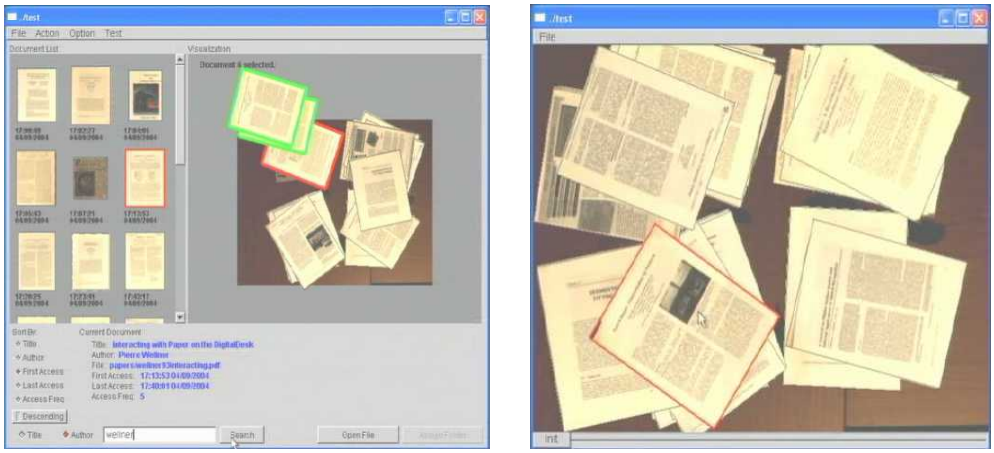


Figure 2.14.: Video-Based Document Tracking; document query (left) and document dragging (right) – taken from the demonstration movie Kim et al. (2004c)

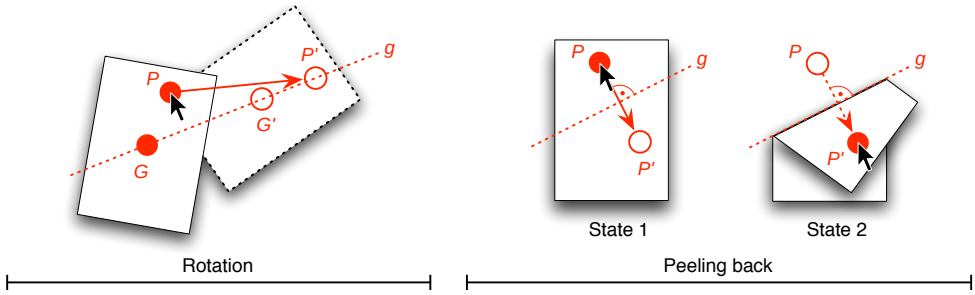


Figure 2.15.: Rotation and peeling back in Beaudouin-Lafon (2001, based on Fig. 4)

provide a better view onto the found document.

The right window depicts a screenshot of a realistic desk view. It shows the same representation as the one at the application window on the left. Positions and orientation of the virtual documents are equivalent to those on the real desk. The user can browse piles by dragging individual documents. The still image looks realistically; however, moving documents does not imply real world behavior, such as incidental rotation while being moved.

Improvements Toward Realistic Look and Feel

Other projects do not aim for completely realistic implementation of real world objects, such as books, but rather try to improve existing elements of applications by adding some real world behavior. For example, Beaudouin-Lafon (2001) aims to apply different novel interaction techniques for overlapping windows, such as rotating or peeling back windows.

Rotation is applied to a window when it is dragged. The left part of Fig. 2.15 depicts how it works: A window is dragged at position P to P'. The relative location of the mouse pointer

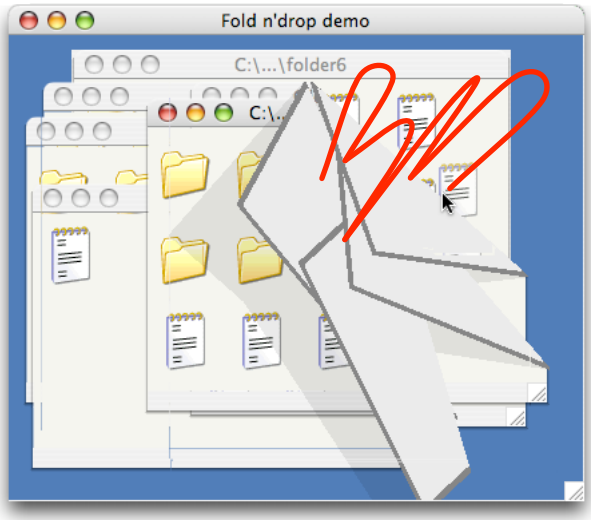


Figure 2.16.: Leaving through windows dragging an icon (approximated mouse path indicated)

on the window stays the same during the complete action. The "center of gravity", G, moves to G' with G, G', and P' aligned on an imaginary line g and a persistent distance between drag point and "center of gravity": $\overline{PG} = \overline{P'G'}$. Additionally, some constraints are applied, such as the maximum rotation to avoid sideways or upside down rotations, or rotating back to an upright position when a window's rotation is less than 10 degrees, in order to provide better readability.

A related idea in Beaudouin-Lafon (2001) is peeling back windows in order to see the window behind. The right part of Fig. 2.15 shows how it works: The virtual line g goes perpendicularly through the middle of segment $[PP']$. The window gets split along g into two quadrilaterals, in one special case possibly into two triangles. The part on which the mouse button was pressed originally is reflected along the virtual line, $S(g)$, as depicted. The window can even be turned completely with this function. After releasing the mouse button, the window moves back to its original position in an one-second animation.

Another project with focus on overlapping windows proposes a "fold-and-drop technique" that allows leafing through windows while dragging an object (Dragicevic, 2004). Figure 2.16 depicts a screenshot with indicated mouse path. The mouse is dragging an icon while it leaves through multiple windows. This project directly borrows from the previously described peeling back function by Beaudouin-Lafon (2001).

We described on page 43 folding of physical documents for the purpose of structuring. This is different to the above proposed technique, which does not consider folding as structuring, but as the visualization of a temporary interaction.

Other projects aim to improve the desktop metaphor. For example, Mander et al. (1992) were investigating and implementing (Rose et al., 1993) a pile metaphor for computers, which became a U. S. patent by Apple Computer, Inc. (Mander et al., 1994) in 2001. Their work was based on real world observations:

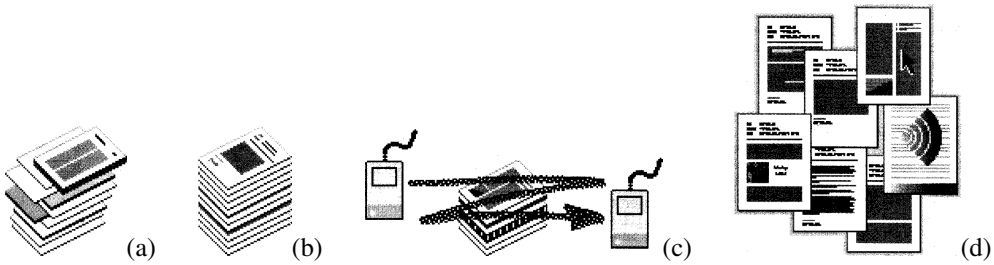


Figure 2.17.: Pile metaphor for user created pile (a), pile with script attached (b), gesturing a pile (c), and result of gesturing (d) by Mander et al. (1992, Fig. 1 and 4) – © 1992 ACM, Inc., used with permission

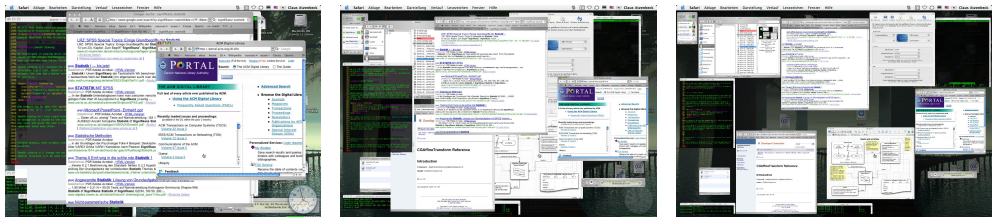


Figure 2.18.: Sequence of resizing and repositioning of windows with Exposé on Mac OS X (duration approximately 0.23 s)

> "Like Malone (Malone, 1983), we found that users like to group items spatially and often prefer to deal with information by creating physical piles of paper, rather than immediately categorizing it into specific folders. Computer users are confronted with large amounts of information, but currently are only provided with a hierarchical filing system for managing it." (Mander et al., 1992, 627)

Figure 2.17 (a) depicts a user created pile. It appears disheveled. The neatness of the pile at (b) indicates that there is a script or a set of rules behind it. A document can be put on on a pile via drag and drop. It appears as the first document on the pile. However, the user cannot rotate or straighten a pile. Every document representation has the same orientation. Mander et al. (1992) uses well-defined semantics for neat and disheveled piles. The appearance as well as its semantics are binary – there are no gradation or shades between them.

Further, they propose gestures to invoke certain actions on the pile, such as spreading out a pile or browsing a pile's documents individually. The first mentioned is depicted in Fig. 2.17 (c) and (d), which shows the gesture and its result. The spread out thusly documents can be manipulated individually. The way the documents are laid out reminds one of clusters with overlays, as depicted in Fig. 2.6 on page 40. There are three major differences to the real world, though. Firstly, there is no sloppiness. The document representations are perpendicular. Secondly, the depicted images are icons and therefore *representations* of the document, but not the documents themselves. Thirdly, a document is represented as one single icon, independently of how many "pages" it has.

We are not aware of any pile implementation in today's major file system browsers or

desktop metaphors. However, Apple Computer, Inc. implemented Exposé[7], a built-in feature on Mac OS X since version 10.3, which acts similar to spreading out a pile of windows. When activated, most[8] visible windows on the screen start to move to a position where they do not overlap with others. If necessary, they shrink proportionally, which gives the impression of zooming out. Figure 2.18 depicts three screenshots that were taken during the transformation, which lasts approximately 0.23 seconds in total. A click on a window will initiate the reverse animation such that the windows move back to their original position, but the clicked window will appear at the very front. It is also possible to apply this function exclusively to windows of the frontmost application. A third feature pushes the windows of all applications outside the desktop area. This gives the user direct access, for example, to icons on the desktop.

These actions can be invoked by keyboard, assigned mouse button, or an active corner on the screen to which the mouse moves. Whereas the gesture in Mander et al. (1992) was to move the mouse over a pile, imitating the movement of a hand over a real pile, as depicted in Fig. 2.17 (c), the activation of Exposé features became more abstract and located "outside" the area of the "windows pile" (keyboard, mouse buttons, or active corners).

Summary

We have shown that only some applications focus on the implementation of highly realistical appearance or behavior. It is natural for projects that deal with real paper or digital documents within a real environment (e. g., augmented reality) to used fixed sizes. However, even though the appearance (size) is similar to real paper, its behavior is often reduced.

Many other developer of metaphor-based applications learned from real world observations that they implemented in their applications. Even though the application uses a high abstract level (e. g., windows or icons that symbolize documents), they support behavior that is known from paper, such as folding or leaving through a pile of windows. However, the behavior is restricted to interactions, but does not emerge during time or act unexpected, such as yellowing paper or incidental rotation when moving.

We conclude that there are applications that have novel techniques implemented that is reminiscent of real paper. However, only in rare cases is behavior implemented that supports advanced emerging metainformation, as described in Sect. 2.2.4.

The next section takes a specific group of metaphor-based applications and analyzes them in relation to the real world in more detail.

2.3.2. Spatial Hypertext Applications with Respect to the Real World

General

The following section analyzes a group of applications that follow a card-on-table metaphor and are used for spatial knowledge structures: *spatial hypertext applications*. We will analyze selected ones with respect to differences to the real world, as discussed in Sect. 2.2. We point out questions that we discover during our comparison. Some of them will be raised directly to become our hypotheses (Chap. 3). Parts of this section are published in Atzenbeck & Nürnberg (2005b, 54–63).

[7] Information about Exposé can be found at http://www.apple.com/macosx/features/expose/ (visited on 2006-03-08).

[8] Some minor windows fade out, for example, the analog clock at the bottom right corner at the screenshots in Fig. 2.18.

Our Work in Context of Spatial Structure Supporting Applications

With this section, we aim to place our work in context of spatial hypertext or related areas. We also point out those applications we use further for our analysis.

Aquanet (Marshall et al., 1991), the first so-called *spatial hypertext application* (Shipman et al., 1995), contains a browser that lets users add or relate rectangular text objects spatially or by using visual cues. The goal is to support rich structures and collaborative knowledge work. Aquanet's background can be found in NoteCards (Halasz, 1987), an associative hypertext model based on a card metaphor, and gIBIS (Conklin & Begeman, 1987, 1988), an argumentation support application. Both applications use spatial maps.

After Aquanet, VIKI was developed. It introduced "collections, a system-supported hierarchy of navigable information spaces, and composites, higher-level structures composed of regular spatial patterns of objects and collections" (Marshall et al., 1994, 13). Later, a fisheye view was implemented (Shipman et al., 1999).

VIKI's successor is VKB (Visual Knowledge Builder)[9] (Gupton & Shipman, 2000; Shipman et al., 2001a). It enhances VIKI by extended visual attributes, easier interaction with other applications, and improved structure recognition (Shipman et al., 2001b, 114), but does not support fisheye views. We used VKB version 1.50, later version 2.00, as one of the main applications for the following analysis.[10] The implementation is written in Java.

A commercial spatial structure application is Tinderbox (Eastgate Systems, 2004; Bernstein, 2003).[11] It supports different structure types on the same information synchronously (Atzenbeck & Nürnberg, 2004; Atzenbeck et al., 2004). Basically, the types can be categorized as spatial structure (map view), hierarchy (chart, outline, tree map, explorer view), navigational structure (HTML view), and linear sequence (Nakakoji view)[12]. Tinderbox supports agents that help to group information automatically. This feature as well as different structure types (except the map view) are of minor interest for our investigations. Beside VKB, Tinderbox version 2.2 is another application we used for the following analysis.[13] The application runs natively on Mac OS X.

2D views have been extended in three-dimensional spatial hypertext applications. Examples are Manufaktur (Mogensen & Grønbæk, 2000) and its successor Topos (Grønbæk et al., 2002). This research branch developed toward "physical hypermedia" (Grønbæk et al., 2003), which aims to interconnect physical and digital objects (e. g., by using RFID tags[14] that allow a machine to track their locations via a tag-reader).

Spatial hypertext applications have the advantage, that they appeal to the user's visual recognition and can be processed in parallel due to their visual representations. They reduce the communication overhead and support quick problem solving (Shipman & Marshall, 1999, 2). However, a disadvantage of spatial structures is that the consistency of visual attributes

[9] See http://www.csdl.tamu.edu/VKB/ for the VKB project site, including free download (visited on 2006-03-10).

[10] Version 2.5 as of March, 2006 is the current version. Main improvements since version 2.00 were a metadata applicator, MP3 audio preview, and spatial parser related changes. Those do not affect our analysis.

[11] See http://eastgate.com/Tinderbox/ for the project site, including a demo version download (visited on 2006-03-10).

[12] Interoperability of spatial structure and linear sequences seems to be inspired by the work of Kumiyo Nakakoji (Yamamoto et al., 2002a,b).

[13] The current version is 3.0.5. Most changes introduced since version 2.2 do not effect our analysis; otherwise, they are explicitly mentioned. Main changes include new rules and actions for machine behavior, or performance improvements.

[14] RFID stands for "radio frequency identifier". Those tags can be detected by tag-readers (see Grønbæk et al., 2003).

may not be given. Additionally, problems may occur if attributes cannot be interpreted correctly, for example, caused by different interpretations of author and recipient. For instance, this may happen due to different connotations of colors or symbols among members of different cultural backgrounds (see also Russo & Boor, 1993).

Some spatial hypertext applications can parse spatial structures and recognize implicit associations between objects. A spatial parser evaluates visual attributes and puts them in relation. Francisco-Revilla & Shipman (2005) give an overview of different parser types used in spatial hypermedia.

Examples for the use of a spatial parser are some of VKB's agents, namely those that use spatial analysis, for example, placement or relationship suggestions (Shipman et al., 2002, 30–31). The application presents suggestions to the users in order to help them in creating structures. A spatial parser-based feature since VKB version 2.0 is the spatial structure shader. It represents hierarchical structures that are expressed by alignment or color as transparent gray rectangles in the background.

Another use case is the conversion of spatial structures into other structure types, as described in Atzenbeck & Nürnberg (2004). Since we do not focus on computational structure awareness, spatial parsers are not fundamental to our research. In fact, only a few spatial hypertext applications offer a spatial parser; most of them do not (e. g., Tinderbox).

Beside spatial hypertext applications, there are also various applications available that may be used for creating spatial structures, even though their main purpose may be different. We will analyze OmniGraffle in the following sections and compare it to VKB or Tinderbox. OmniGraffle is a commercial chart application for Mac OS X "to create everyday documents like photo albums, yard sale flyers, CD covers, garden layouts, newsletters, and almost anything else you can think of" (Omni Group, 2005, 2). It is obvious that the application domain is desktop publishing rather than the creation of spatial knowledge structures. Additionally, as opposed to spatial hypertext applications that follow a card-on-table metaphor, OmniGraffle uses a canvas metaphor: the user can draw or modify graphical objects on canvases. Nevertheless, in the following we compare objects on canvases to card equivalents in spatial hypertext applications and a canvas to the "table". The "misuse" of OmniGraffle as "spatial hypertext application" does not come out of the wild. A first test with OmniGraffle for this application domain was successful.[15] We use OmniGraffle Pro[16] version 4.1.1, and will refer to it in the following as "OmniGraffle".

In the following sections, we analyze real world paper structures to comparable computer applications, mainly to those presented above. The central question is: What can we learn from how people use paper in order to improve spatial knowledge supporting applications? Because our comparison focuses heavily on paper, we exclude parts that cannot be found in the real world, such as search engines, agents, spatial parsers, or similar application-based helpers from our analysis.

Figure 2.19 gives a first impression of differences between real world structures and spatial hypertext applications. It depicts the result of a survey on how students use VKB for magnetic poetry (Shipman et al., 2001a). In the following, we will refer to this figure when pointing out differences in structure or behavior.

[15] We plan to extend our test and report our analysis and experiences in a future publication.
[16] There is a regular and a professional ("Pro") version.

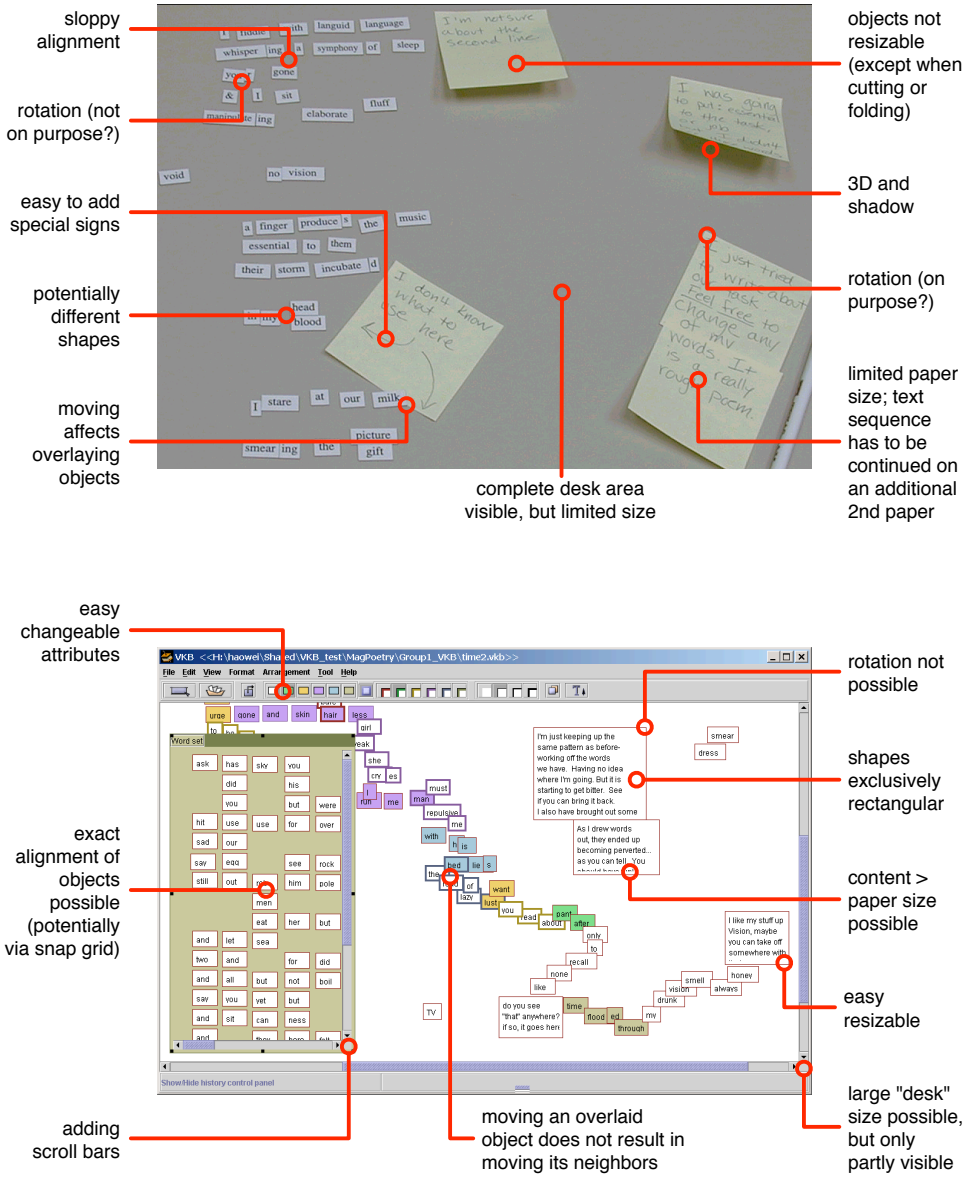


Figure 2.19.: Comparison of real world and spatial hypertext application – pictures taken from Shipman et al. (2001a, Fig. 7 and 8); © 2001 ACM, Inc., used with permission

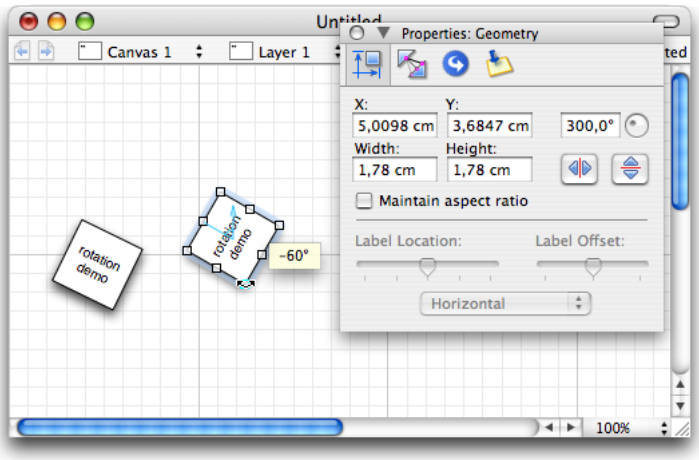


Figure 2.20.: Rotation in OmniGraffle

Rotation and Sloppiness

As Fig. 2.19 depicts, none of the objects of the real world magnetic poetry are exactly aligned horizontally. Some of the yellow sticky notes seem to be rotated on purpose; at least the rotation angle is obviously larger than others. VKB or Tinderbox, however, have no support for rotating objects. Nodes are positioned exclusively horizontally.

OmniGraffle allows rotation of any object, as depicted in Fig. 2.20. An object can be rotated with the mouse while the command key is pressed or via the geometry inspector window, which is also shown in the figure. The current angle is shown in a pop up window during the rotation process.

Even though OmniGraffle supports rotation, the user is required to perform this explicitly. In the real world, rotation angles mostly emerge while building a structure, as the magnetic poetry picture demonstrates. On the other side, even intended rotation, as we assume for the yellow sticky notes shown in Fig. 2.19, cannot be applied by VKB or Tinderbox.

The following questions regarding rotation arise:

- Would a person use rotation explicitly for spatial structures?
- Would incidental rotation in spatial structure applications, as it happens in the real world, support the user in orientation or finding?
- Would rotation create more natural or aesthetic looking spatial structures?
- Would rotation seriously conflict with the bad resolution on screens (compared to print media) and the fact that pixels are aligned in horizontal lines and vertical columns? Would anti-aliasing solve this problem?

Rotation is partly responsible for sloppy-looking spatial structures. Beside sloppiness, there are also x or y offsets of objects that represent the delta to the exact position, that is, mostly the intended position. Many computer applications support the user in neat alignment

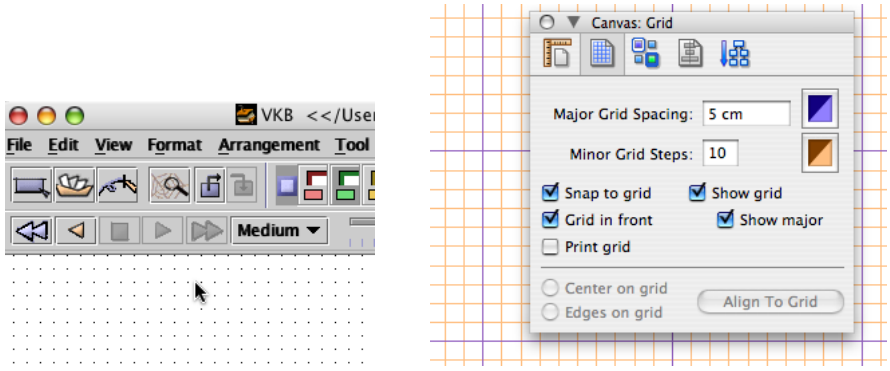


Figure 2.21.: Grid in VKB (left), and grid and grid inspector window in OmniGraffle (right)

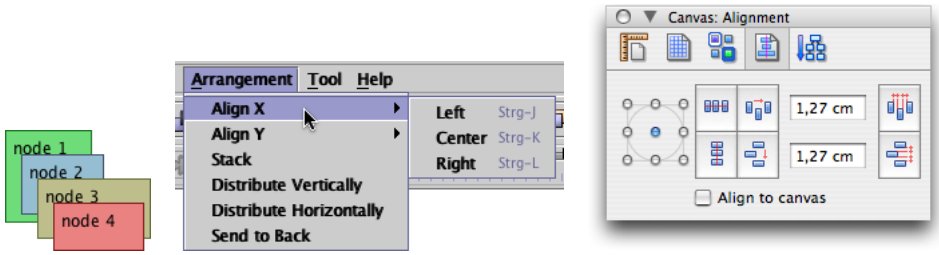


Figure 2.22.: As stack aligned nodes in VKB (left), and alignment tools in VKB (center) and OmniGraffle (right)

or positioning, for example, with a grid to which object snap to. However, most spatial real world structures do not follow an exact grid. This causes, for example, piles that are not aligned exactly, as depicted in Fig. 2.4 on page 37.

Most spatial structure supporting applications of the type we analyze have a grid feature implemented. For instance, Tinderbox has an invisible grid of factor 0.125 of the default height of a note,[17] which cannot be switched off. Nodes snap to it on mouse release.

VKB has a fixed grid which can be switched on or off. It is represented on the background, as depicted in Fig. 2.21. The screenshot in Fig. 2.19 does not have the grid switched on, neither was it on during creation. This can be seen on slightly sloppily aligned nodes.

OmniGraffle supports an optional grid of arbitrary size. As shown in Fig. 2.21, it has several additional visual features. Those include independent color selection for major and minor grid,[18] whether the grid should be painted in the front, or whether the grid should be used for printer.

Many applications provide specialized tools for alignment support. Figure 2.22 depicts those tools for VKB and OmniGraffle. VKB provides alignment functions for horizontal or vertical alignment or distribution, but also for creating stacks. The left screenshot in Fig. 2.22 shows an example of a stack arrangement.

[17]This information was given by Mark Bernstein in a personal e-mail of 2004-05-06.
[18]The screenshot shows translucent blue for major and translucent orange for minor grid.

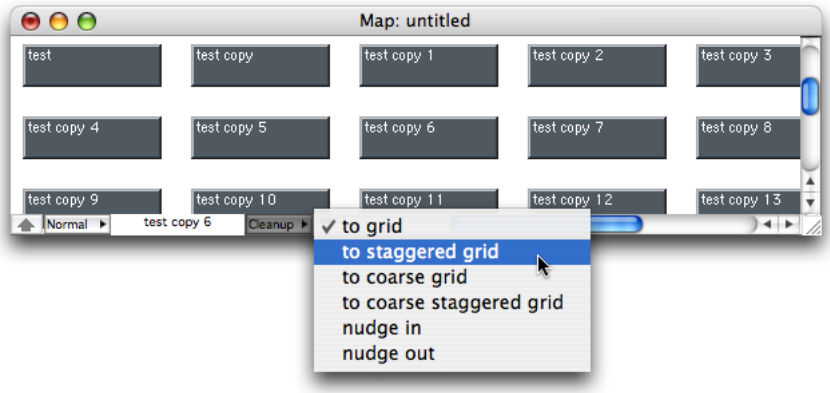


Figure 2.23.: Grid options in Tinderbox

The right screenshot depicts the OmniGraffle's alignment inspector window. It provides similar functionality than VKB. The alignment position (left, middle, right, and top, center, bottom) can be selected by radio buttons. Additionally, the alignment can be relative to the canvas. Horizontal or vertical distributions are possible. There is no "stack" function. However, there are two buttons on the right of the inspector window that place selected objects horizontally or vertically according to the given distance. As the grid inspector window in Fig. 2.21 depicts, OmniGraffle supports also aligning objects to the center or edges of the grid.

Beside a grid, Tinderbox also supports grid related commands. They are provided via a pop up list named "Cleanup", next to the horizontal scrollbar. Figure 2.23 shows an example. The first four entries place the nodes on grids of different types. The screenshot is an example for "Cleanup to grid". The remaining two entries nudge nodes in or out.

In the real world, alignment support exists partly. For example, loose paper on a desk can be transformed into a stack easily by moving the hand over the table and collecting the paper at once. However, the result will not look as perfectly aligned as it does in any of the computer applications.

An interesting aspect can be raised by looking at the level of sloppiness from an aesthetic point of view. (We want to raise this question without trying to give an answer.) On the one hand, it seems that people tend to like aligned and symmetric spatial structures. The right picture in Fig. 2.24 shows an example. Schloss Schönbrunn is a palace in Vienna, Austria. It was built between 1692 and 1713 and altered in the 18th and 19th centuries (Wikipedia, 2006b). The picture shows the palace's regular looking frontage. Groups of windows are aligned neatly and share the same size or type. The architecture style seems "well organized".

On the other hand, people like the Austrian architect and artist Friedensreich Hundertwasser follow a different idea. The left picture in Fig. 2.24 shows the "Grüne Zitadelle" in Magdeburg, Germany. It was finished in 2005, five years after the architect's death. It was build mostly according to his plans. There are only some horizontally or vertically aligned lines. The "sloppy look" of this building is also supported by differently shaped or sized objects. For instance, there is a variety of windows with different shapes or sizes.

Both architectures, Schloss Schönbrunn as well as Hundertwasser buildings, have propo-

Figure 2.24.: Schloss Schönbrunn, Vienna, Austria (left),[a] and Grüne Zitadelle by Hundertwasser, Magdeburg, Germany (right)[b]

[a] This picture has been taken by Alexander Umbricht on 2002-10-01. It is available at http://en.wikipedia.org/wiki/Image:Wien_Schoenbrunn_Rueckseite.jpg (visited on 2006-03-06) and has been cropped for the purpose of this thesis. The original has been released under the public domain.

[b] This picture has been taken by Doris Antony on 2005-09-14. It is available at http://en.wikipedia.org/wiki/Image:Magdeburg_Hundertwasserhaus.jpg (visited on 2006-03-31). Copyright © 2005 Doris Antony, released under the GFDL.

nents and are well recognized all over the world. Both are different in how "sloppy" they look. This raises the question about whether people feel more comfortable with sloppy or straight looking buildings and whether this can be addressed for paper structures as well. This may be also a matter of different times or cultures.

In a personal e-mail, Mark Bernstein, chief scientist of Eastgate Systems, told us that Tinderbox's grid was implemented, because "people like to have things line up neatly"[19]. This would be a contradiction to what Hundertwasser was doing, if architecture and knowledge representation domains are comparable in this matter.

The following questions about sloppiness can be risen:

- Would grids change the development of structures?
- What kind of implicit metainformation would structures contain that were built with grid or alignment support in comparison to structures that were built without?
- Would finding or recognition be supported by spatial structures that were built without grid or alignment support?
- What would be the subjective satisfaction of people working with sloppy spatial structures versus those working with neat looking ones?

3D and Physical Forces

None of the analyzed applications support true 3D. However, Tinderbox and OmniGraffle offer visual cues to give the impression of depths. As depicted in Fig. 2.25, Tinderbox draws nodes with a three-dimensional look. However, this is just a fixed visual cue on all nodes without additional semantics.

[19] E-mail of 2004-05-06.

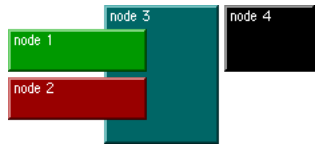


Figure 2.25.: Node representations in Tinderbox

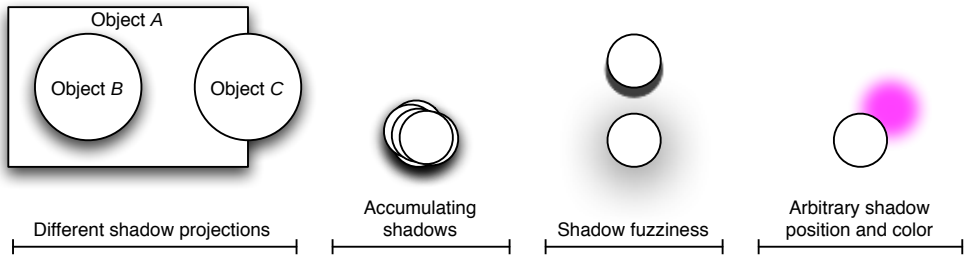


Figure 2.26.: Shadow attributes in OmniGraffle

OmniGraffle supports optional shadows to indicate a third dimension. There are different attributes available for shadows. Figure 2.26 depicts some examples. The left example shows different shadow projections. Object *B* is set to display its shadow immediately beneath itself, whereas object *C* displays it behind *all* object of the same layer. Therefore, *C*'s shadow is not visible on top of object *A*. This can be used to indicate a distance between two objects, for example, object *C* seems to touch *A*, whereas *B* does not. The second example shows that overlaying shadows accumulate in intensity. The area where many shadows are projected appears darker. This gives an impression of how many objects there are. The remaining two examples depict different attributes in shadow fuzziness, color, or position. This may be useful if additional visual attributes are required.

Several yellow sticky notes shown at the magnetic poetry example in Fig. 2.19 do not stick flatly on the board. They are curved toward the recipient. This may be not intended in this case; however, we have experienced that some people use this as a visual cue to indicate an important spot. One example are dog eared pages. They help the user to find a page easier through its different shape. None of the observed applications allow something similar. Also none of the 3D spatial hypertext applications mentioned on page 56 support this kind of marking. However, work is done in the past on simulating dog ears in computer applications (Hoeben & Stappers, 2000).

Physical forces, such as gravity, friction, or inertia play an active role in structure creation or modification, for example, overlapping nodes: The magnetic poetry shown in Fig. 2.19 has the word "milk" at the lower part of the picture partly on top of a yellow sticky note. Removing the sticky note results in some movement of the "milk" object. If an object, for example, an ISO A6 paper is completely on top of an ISO A4 paper, slow movement of the A4 paper will also move the A6 paper.

Tinderbox and VKB do not offer similar behavior. OmniGraffle has a grouping function, which lets the user group objects on the same layer. Movement or alignment functions will be applied to the group as a whole. However, this only reflects some aspects of the described real

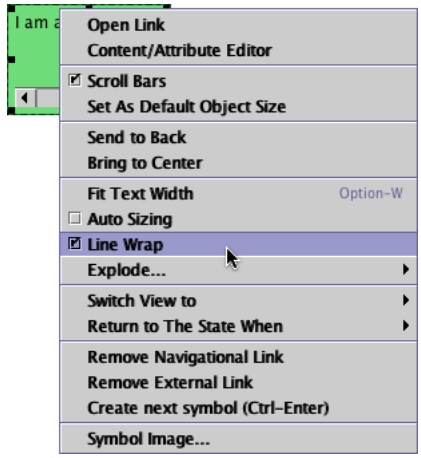


Figure 2.27.: VKB's context menu for nodes

world behavior. The user needs to group items explicitly, whereas it happens automatically in the real world. We are not aware of any knowledge management application capable of simulating gravity, friction, or inertia.

This raises the following questions:

- How would structures evolve when gravity, friction, or inertia are simulated?
- How would users interact with spatial structure environments that would simulate forces known from the real world?

Shape and Size

Office workers use mostly rectangular shaped paper. Similarly, VKB as well as Tinderbox provide only rectangular nodes. However, external graphics may be included that show other shapes. OmniGraffle is the only application among the analyzed ones that allows to create arbitrary shapes.

Size is another important difference. As argued in Sect. 2.2.2, the used paper size mostly relates to standards and remains unchanged, for example, through cutting the paper. Once a paper is cut, it is difficult to extend it again without any remaining damage. All three observed computer applications, however, allow to resize objects without "damage".

Figure 2.19 shows two yellow sticky notes at the lower right corner of the magnet poetry picture that are put together. Apparently, the content did not fit one single paper and had to be extended on a second one, which was placed below the first one. The VKB screenshot at the same figure depicts some equivalent nodes. Also here, the text exceeds the visible size of the node. However, a closer look shows that the node acts similar to a window, which displays the text only partly. The user is able to scroll the text using the text cursor. Alternatively, resizing the node will cause more text to be visible. VKB also supports scrollbars, automatic sizing, or line wrap. These functions can be switched on or off individually for each node at any time through the context menu, as depicted in Fig. 2.27.

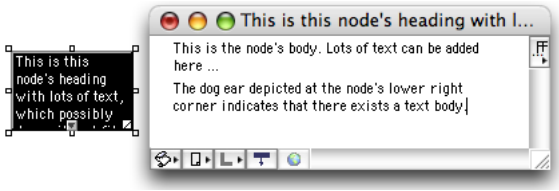


Figure 2.28.: Node with heading in map view and its content in Tinderbox

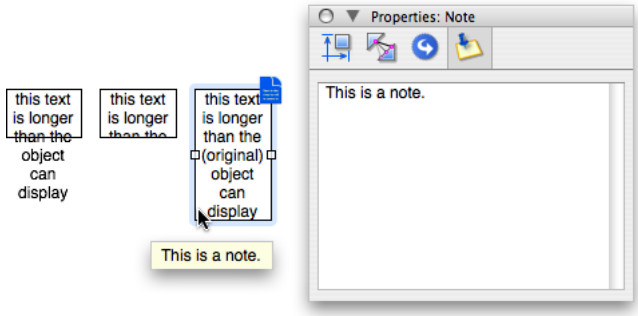


Figure 2.29.: Nodes with different attributes and note inspector in OmniGraffle

Tinderbox's map view, represents only the node's heading. Text as much as possible is displayed on a node, the rest is hidden. Since version 2.5.0, there are menu commands that let a node expand horizontally or vertically until the complete text is visible. Users cannot scroll headings as they can do with VKB. A heading can be edited via a dialog window. This differs to VKB or OmniGraffle, which edit the text directly at the node. Tinderbox supports a text body beside the heading. As mentioned above, it will not be displayed at the map view; however, it can be used either in a separate window or be exported in another format, for example, HTML. The spatial representation of a node with long heading and its content window is shown in Fig. 2.28. The small dog ear symbol at the right bottom corner indicates the existence of a text body.

OmniGraffle supports three modes for objects to handle long texts. Figure 2.29 gives an example for each mode. The rectangles represent the nodes' bounds. The left example continues drawing the text over the node's bound. The second node cuts the text. The last one adjusts the node's height automatically according to how much space is needed to display the text completely. Alternatively, the user can set the side or top/bottom margin to adjust the space between text and node bounds.

Similar to Tinderbox, OmniGraffle allows the attachment of a note to any object by using the note inspector. Figure 2.29 depicts the inspector. Additionally, the note appears as tooltip next to the cursor when moved over an object. A blue icon on the node's top right corner indicates the existence of a note.

The previous discussion leads to the following questions:

- What would be users' subjective satisfaction, if only some fixed node sizes or shapes would be available?

- What would be users' subjective satisfaction, if the size or shape of an object could not be changed after creation?
- What problems or advantages would be experienced, if the amount of content of a node would depend strongly on its size, similar to real paper?

Desk Size

There is a significant difference between the real world and the observed computer applications with respect to the desk size. A real desk is very limited in space compared to simulated space in computer applications. It can be argued that the desk space may be extended, for example, by putting paper on the floor or in shelves, but this is only a small extension. Tinderbox and OmniGraffle exceed this size significantly.[20] According to a personal conversation[21] with Mark Bernstein, Tinderbox uses 16 bit (possibly even 32 bit)[22] in each direction to store the offset of an object. 16 bit would be 65,536 pixels, which is equivalent to over 23 m × 23 m on a screen with 72 dpi resolution. 32 bit would result in over 1,515 km × 1,515 km. According to our tests, OmniGraffle is limited to a maximum canvas size of 3,527.8 km × 3,527.8 km, independent of scaling. This is an area larger than the USA or Canada.

These examples show that the possible virtual desk size is much larger than a real desk possibly can be. All analyzed applications have horizontal and vertical scrollbars to navigate. In addition to that, they support panning the background via mouse.[23] However, large spaces do not *force* an office worker to clean up the desk before placing other documents. This is different to the real world, where someone has to clean up from time to time in order to gain vacant space. Frequent reorganizing or restructuring is necessary, as discussed on page 37.

Some questions come in mind:

- How would users structure spatially, if space would be limited?
- How would spatial structuring change, if output or input devices would be more natural, for example, large touchscreens or projection on a desk? – As discussed in Sect. 2.3.1, there is research done on using digital documents on a physical desk (e. g., Wellner, 1993; Ashdown, 2004). However, we are not aware of any analysis of the thusly created spatial structures in real work environments.

Collection Objects

A collection object follows the metaphor of a drawer or box in which the user may put objects. The "box" may be located on the same space than other nodes. Collection objects also can be seen as hierarchically ordered sub-spaces. OmniGraffle does not provide this kind of sub-spaces. However, it supports linear ordered canvases, that are, spaces on which objects can be placed.

[20] Currently, we do not know the maximum size of a VKB space.
[21] Conversation of 2004-05-07.
[22] Mark Bernstein was uncertain about the exact number.
[23] In VKB, the command key has to be pressed. In Tinderbox, panning does not require any additional key stroke. OmniGraffle has a tool for panning, which can be activated, for example, by pressing the space bar.

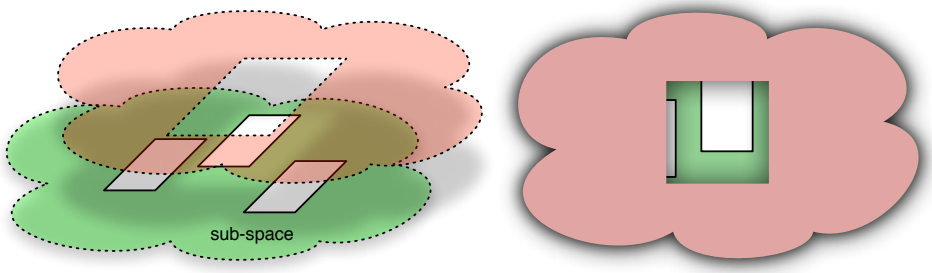


Figure 2.30.: Representation of a collection object as hole; bird view with transparent spaces (left) and top view (right)

A more appropriate metaphorical interpretation would see collection objects on 2D spaces as “holes” that lead to sub-spaces, as depicted in Fig. 2.30. Sub-spaces of all analyzed spatial hypertext applications do not show specific constraints. Therefore, they can be seen as as parallel spaces underneath the current one. However, a metaphor break occurs when collection objects are moved. In the real world, a hole in a solid material cannot be moved. Furthermore, moving the “hole” results also in moving the complete space underneath.

VKB supports collections. An example is shown in Fig. 2.19. The collection object looks as an additional window. They are different to text nodes. The depicted one has a different background than the main window and a title. The scrollbars indicate that the sub-space is larger than the visible area. However, they also can be switched off. There are functions to scale the content of collection objects in VKB.

In Tinderbox, any node may serve as collection object. The first image of Fig. 2.32 on page 70 depicts a node which contains other nodes. Similar to VKB, it has a single line with the node’s title and shows a part of its sub-space. However, this part is downscaled. The size of this “preview” area can be changed at any time by resizing the collecting node. The last image of Fig. 2.32 depicts the inside of the collection object. The size of the containing node is indicated as a frame on the background.

In the real world, people use drawers, boxes, etc., to collect objects. Such collection facilities have bounds. They have roughly the same dimensions, both inside and outside. In comparison, a VKB or Tinderbox collection may contain a larger space inside than is visible at the outside. The limitation in the real world forces reorganization whenever a container becomes too small for its content. A potential advantage would be that a person has an understanding of a drawer’s possible contents by number or size. Spatial hypertext applications do not provide information about the size of a collection. Even collections inside collections are possible.

Collection related questions are:

- Would containers with limited space support the user? If yes, how?
- How would structuring change, if only non-resizable collection objects would be available?

Focus–Context

Changing focus and context *affects* structure creation, but is not part of the structure itself. We experienced the importance of related interactions while comparing human interactions with paper and spatial hypertext applications.

This discussion refers to the levels of structure, or "scales" (Ware, 2004, 339), as we have explained in Sect. 1.2.2. Ware (2004, 340–344) distinguishes between four different techniques that solve the *focus–context problem*, that is, how to jump between different scales or levels of structures: *Distortion techniques* distort the map to provide more room at where the focus is. One example are hyperbolic trees (Lamping et al., 1995). *Elision techniques* hide structure parts until they are needed. For example, the graphical representation of a large structure collapses into a single object when not needed. The object still would be placed on (or in) the space; however, it would not reveal its internal structure. Examples include generalized fisheye views (Furnas, 1986), that are used to hide details the further the focus moves away. A third possibility to support focus–context changes in applications are *multiple windows*. They can be used to display different views on the same space. Finally, there are *rapid zooming* techniques:

> "In rapid zooming techniques, a large information landscape is provided, although only a part of it is visible in the viewing window at any instant. The user is given the ability to zoom rapidly into and out of points of interest, which means that although focus and context are not simultaneously available, the user can move rapidly and smoothly from focus to context and back." (Ware, 2004, 342)

"It is worth noting that the focus–context problem has already been solved by the human visual system" Ware (2004, 339). Human sight depends on the visual field. Humans do not perceive anything visual outside. Additionally, "[t]he acuity of the eye falls off rapidly with distance from the fovea" (Ware, 2004, 51). This means in analogy to computer monitors, that there is a higher resolution and more details in the center of the focus; Ware (2004, 52) mentions "brain pixels" in analogy to pixels on the screen. It becomes blurry toward the outside of the visual field. This reminds one strongly on elision techniques, that hide structure parts, or distortion techniques, that have a higher resolution and greater details at the center of focus.

Compared to the capabilities of humans in real world environments, the focus–context support in most of today's computer applications is poor. We will discuss related features of the analyzed applications in the following.

All of them support basic zooming; however, no rapid zooming is provided. VKB has three different zoom relevant entries in its menu, as shown in Fig. 2.31: 125 %, 80 %, and 100 %. The first two mentioned are relative to the current scale, the last one resets to 100 % absolute. Alternatively, the scale factor may be set arbitrarily via a dialog window.

Tinderbox has pop up lists at map view windows with zoom levels from -4 to 4. The middle position is called "Normal". Similar to Tinderbox, OmniGraffle has a zoom pop up menu on any window available. It shows basic zoom scales in percentage, which can be selected by one click. The menu also allows to enter an arbitrary zoom factor manually, up to 800 %. Another menu entry selects a zoom level such that the complete content of the canvas is visible. OmniGraffle maps the mouse wheel to zoom in or out. However, a large number of

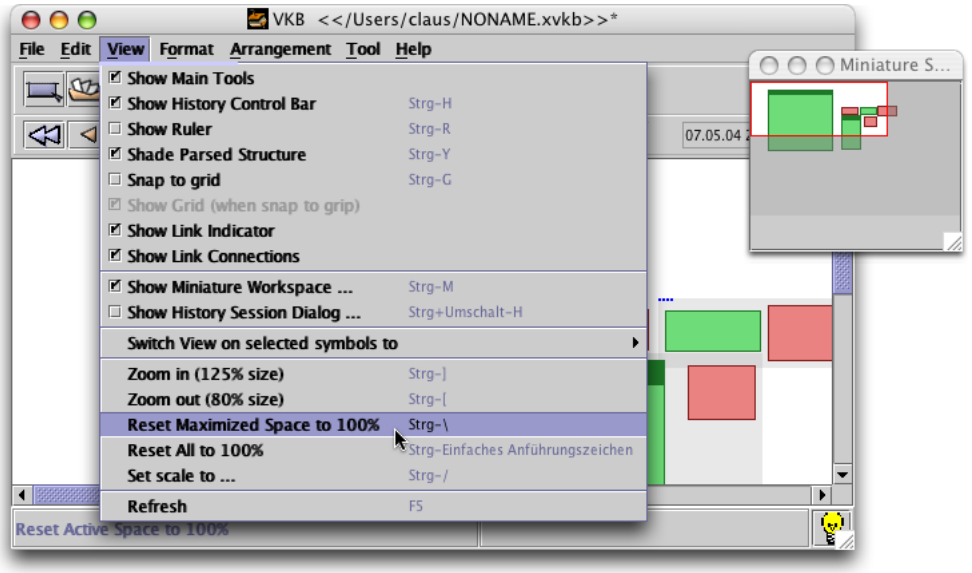


Figure 2.31.: Zoom functions and floating miniature workspace window in VKB

objects heavily affects zooming performance. Depending on the used hardware, efficient and smooth zooming may not work smooth in those cases.

Figure 2.31 depicts VKB's floating miniature workspace window, which gives an overview of the complete workspace. The user can imitate a quick "zoom out" by changing the eye focus from the normal window to the overview map. The red border on the mini map indicates the viewport at the main window. It can be used for navigation by dragging it with the mouse.

Tinderbox allows users to have a maximum of two map windows of the same parent (i. e., space) open at the same time. Both views represent the same set of data, but are independent in scale or location from each other.

OmniGraffle supports unlimited views on the same document, represented in individual windows. Similar to Tinderbox, they may vary in size, viewport, or scale. For example, this can be used to have three windows open, one scaled to 100 %, another one to 50 %, and the third one set to automatically keeping the complete content of the canvas visible. Users can change their focus from one window to another. Unlike the mini map in VKB, OmniGraffle's alternate windows are identical in functionality. Objects can be modified at any of them. Changes are updated immediately on all windows. On the other hand, a window cannot be used to navigate the view of another one. Similar to VKB and Tinderbox, OmniGraffle's feature allows switching quickly between different levels of detail.

Overview maps may be beneficial if efficient zooming interactions are missing. However, there is evidence that for zooming-enabled applications an additional overview map may cause slower performances for finding (Hornbæk et al., 2002).

We are only aware of one publication within the spatial hypertext research field that explicitly deals with focus–context: Shipman et al. (1999) describe the implementation of a multiple fisheye view for VIKI. VIKI's successor VKB does not have fisheye views implemented. The lack of usability tests, however, still leaves the motivation for and advantages of

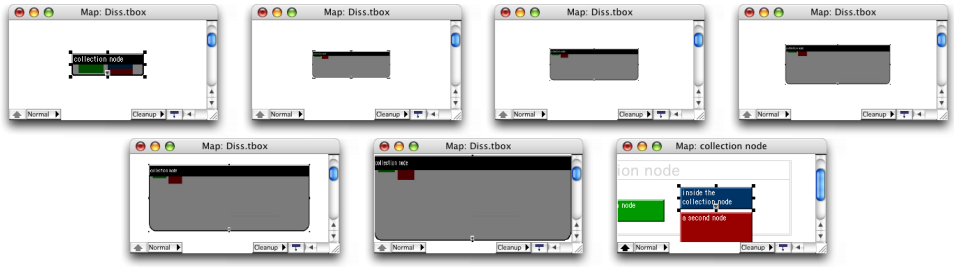


Figure 2.32.: Sequence of zooming into an object in Tinderbox (five different zoom steps visible in total, duration approximately 0.3 s)

multiple fisheye views for spatial hypertext in the dark.

We believe that rapid zooming is the most natural focus–context supporting technique in combination with human perception among the above discussed. The reason is obvious: The human eye scans constantly different parts of the monitor. It does not stay at a specific location on the screen. As discussed above, the human eye has its own focus and views the part outside blurry. Distortions or elision views on the screen could not be updated as the eye moves. Therefore, both the rendered and the eye focus could be in contradiction.

Zooming, however, provides a plain view with no distortion. Users may focus any point of a displayed scale at any time. Their eyes focus on the desired spot, whereas the context becomes blurry. This is based on the eye's anatomy, not on machine rendering. The zooming action itself simulates a change of the area that intersects with the visual field, without having the person to move. It could be mentioned that a screen is limited in size and therefore does not provide the same amount of context that would exist in the real world. In this respect, huge screens in high resolution that are close to the user would support this idea better.

There are research projects on efficient zooming. Examples include zooming for image browsing (Combs & Bederson, 1999), more recently also image browsing on PDA devices (Khella & Bederson, 2004), a zooming browser for hierarchically clustered documents (Toyoda & Shibayama, 2000), or a presentation application with zooming support (Good & Bederson, 2002). However, the zooming functions in many applications that are primarily used for spatial knowledge structures, such as spatial hypertext, are not satisfying compared to the real world. As described above, mostly their commands activate explicit scale factors. For instance, the user has to state that the application should zoom to 150 %. More realistic behavior would be to zoom in or out "some more", without thinking about the scale factor. Of the analyzed applications, the one closest to the real world in zooming is OmniGraffle, when using the mouse wheel for zooming.

Some applications provide the visual impression of rapid zooming. For instance, Tinderbox shows an animated "moving into a node". Figure 2.32 depicts the sequence as pictures. The duration and the number of visible zoom steps depend on the window size, the processor speed, and the number of items at the destination space. The depicted example an animation with five different zoom steps visible. The duration of the whole process was approximately 0.3 seconds.

Even though the visualization appears smooth, it cannot be stopped. It shows a node as start state and the node's sub-space as end state. It is not possible to stop in between or

have a node's content side by side to objects at other structure levels, as it can be the case in the real world. Therefore, the user looses the context when entering a sub-space. However, the animation may help to provide a better understanding of the spatial change or the spatial relation of collection node and its space.

This discussion leads to the following questions:

- How would knowledge workers make use of rapid zooming in spatial hypertext applications?
- Would rapid zooming change the way spatial structures are created?
- How would knowledge workers make use of rapid zooming for finding information in spatial knowledge structures?

Summary

We have shown that the analyzed applications reduce limitations beyond what is known from the real world. This includes emerging sloppiness, physical forces, or maximum available space size (including sub-spaces, i. e., collection objects). Reducing limitations result in higher freedom. However, it also leads toward more complexity within structure levels. We raised questions that asked about the benefits of limitation reducing behavior that is based on what users know from the real world.

Whereas in most observed cases, applications aim to avoid limitations by breaking their metaphors, focus–context changes are different. Human perception is based on a highly developed system. Focus and context can be changed rapidly. This is different in most of today's applications. The analyzed ones provide only basic support for that.

In the next chapter, we formulate our hypotheses. They are based on this analysis.

Chapter 3.

Hypotheses

„Das Ganze ist ein riesiger Misthaufen, der Perlen enthält. Aber um Perlen zu finden, muss man die richtigen Fragen stellen. Gerade das können die meisten Menschen nicht.“

(Joseph Weizenbaum about the Internet, May 2005)

3.1. Remarks

We analyzed real world paper structures in Sect. 2.2 and compared some aspects to selected spatial structure supporting applications in Sect. 2.3. In Sect. 2.3.2, we discussed especially constraints as well as focus–context support, and pointed out specific questions after each section.

We argued that even though the analyzed applications follow paper-related metaphors, they have fewer constraints than the real world. We pointed especially to emerging sloppiness, physical forces, and maximum available space. On the other side, we argued that changes in focus or context are more efficient for the humans than current computer implementation. We further mentioned that we believe zooming (in combination with human perception) is the most natural among several discussed ones.

In the following, we define our hypotheses. They are based on selected questions that have been raised in the previous chapter, and summarized in Tab. 3.1. The table head shows abbreviations that stands for applications with one extended feature each.[1] Those are:

[1] The abbreviation is also used in Chap. 4 and 5 to refer to versions of our prototype, including *v4*, which supports

Test	*v1*	*v2*	*v3*	Focus
Time for organizing	▼	▼	□	time used
Time for finding	▲	▼	▼	time used
Occupied area	▼	□	□	occupied area
Zoom usage	▼	–	□	activation count
Subjective satisfaction		▲		users' rating

▼ = significantly less; ▲ = significantly more;
□ = no obvious tendency; – = not part of comparison

Table 3.1.: Summary of our assumptions

1. Variable document sizes (*v1*)
2. Extended zooming (*v2*)
3. Rotation and sloppiness (*v3*)

We describe the application which we coded for our usability test in Chap. 4 and report the test design and statistical evaluation in Chap. 5. A summary overview of the statistical results can be found in Tab. 5.4 on page 208.

3.2. Hypotheses Phrasing

3.2.1. Variable Document Sizes (*v1*)

Computer applications have little constraints in object sizes, whereas text on paper is limited to the paper's size. Nodes in spatial hypertext applications can be resized; paper can not. We argue that it may be a disadvantage for users to have object sizes independent of the amount of text they carry. Opposed to real paper, users cannot judge the amount of text, if no other representation of this information is given. Scrolling text or resizing nodes becomes essential.

We expect users of variable size documents to be significantly slower in finding information than those with fixed size documents of ISO A4 size. However, because of the initial small document size we expect variable size documents to lead to shorter organization times for organizing a stack of documents compared to fixed size version.

We further assume that users would occupy less space with variable size than with fixed size documents. Because of a smaller occupied area, we assume that users will not or significantly less often use zooming than with a fixed size version.

3.2.2. Extended Zooming (*v2*)

Zooming is provided by all applications we analyzed in Chap. 2. However, none of those are close to what we experienced with the human capability of changing focus. We investigate in two kinds of rapid zooming: smooth zooming and quickzoom.

Smooth zooming is a feature that enables users to zoom in or out seamlessly without predefined zoom steps. This simulates the human eye in respect of getting closer or further away, or stop at any level.

Quickzoom simulates the behavior of users who sit on a desk reading. When they want to look for another document, they leans back to get an overview of the complete desk. After the desired document is found, they leans forward again and focus on its content. Quickzoom is a fast way of zooming out completely and zooming back to where the mouse cursor points to. Both, zooming in and out, are animated and give an understanding of where the focus is brought to.

We claim that extended zooming enables users to organize or find information more quickly, compared to a version without smooth zooming or quickzoom. We do not expect larger occupied areas.

Additionally, we expect that the user's subjective satisfaction will be higher with smooth zooming or quickzoom enabled.

fixed size documents.

3.2.3. Rotation (*v3*)

Rotation is an obvious property of paper structures, except for the content of certain bindings, such as books. However, our analysis has shown that many applications do not support rotation, but instead often support tools for eliminating sloppiness in spatial structures. As described in Sect. 2.2.2, sloppily aligned documents may provide information about documents located behind them. They also give some impression about how many documents there are.

Additionally, individually rotated or pulled-out documents may be used to mark specific locations within a structure, for example, a document that is pulled-out of a pile.

We argue that rotation or sloppiness, both purposeful or automatically applied by the system during structuring, help the user to remember or recognize locations and therefore support quick finding. We expect significantly shorter times for finding information, compared to a version without.

We do not expect any significant difference in time for organizing a pile of documents, because we assume that most users will organize document by document from top to bottom, which would not be affected by sloppiness in most cases. Even though users may receive a better understanding in how many documents the pile has, we do not assume that to be beneficial for the organization time.

We assume that the size of the occupied area is in relation to the frequency of zooming in and out. Sloppy piles occupy a larger area than neat ones, but we do not expect significance for the total space occupied. Therefore, we expect a similar number of zooming in or out compared to versions without rotation.

Part II.

Implementation and Evaluation

Chapter 4.

Application Design and Implementation

> "I liked zooming in and zooming out. It was almost like a real desk where you can look around."
>
> *(Participant s31 about WildDocs)*

4.1. General

4.1.1. Overview

In this chapter we discuss the implementation of WildDocs, our prototype that we use for testing our hypotheses. The name WildDocs is an abbreviation of "wild documents" and comes from the fact that it supports emerging sloppiness among other features; this causes sloppy looking spatial structures that look rather "wild".

Due to the nature of our analysis we focus on a high level of details. This leads to a detailed discussion of our prototype. The goal is to create an application that is capable of testing the effects of simulating real world properties for organizing and finding information on a 2D space.

This section discusses the general WildDocs base implementation. Section 4.2 continues with presenting the implementation of documents. Our goal was to provide support for WildDocs behavior in separate classes, which are member of the package machines. We discuss those in Sect. 4.3. Finally, Sect. 4.4 focuses on handling user interactions and Sect. 4.5 concludes this chapter with miscellaneous code.

The discussed implementation is based on WildDocs v20050919, as of September 19, 2005. We developed the prototype on Mac OS X. Mainly because Java 5 was not available for this system at that time, we started with Java 1.4.2 and stayed with it. However, the application compiles and runs also on Java 5 environments. We used Eclipse 3. The code base contains 13,917 lines of code in 68 Java classes. A list of classes grouped by packages can be found in Tab. 4.1.

We use Piccolo 1.1 for Java as the main underlying framework for WildDocs.[1] Piccolo provides support for 2D spaces, including zooming (Bederson et al., 2004). The core classes in Piccolo are specializations of PNode, generally called "nodes". Figure 4.1 depicts their relation. PRoot is the root node of the Piccolo tree. PLayer and PCamera are classes that are necessary to build a scene graph. A camera looks at a layer. Multiple cameras, also on the same layer, are possible. Nodes that should appear on the screen are added directly or indirectly to a layer that is looked at by a camera.

[1] Piccolo's current version for Java is 1.2 as of 2005-12-06.

de.atzenbeck.wilddocs		
WDBoundsHandle	WDDeskInputEventHandler	WDNodeInputEventHandler
WDCanvas	WDLayer	WDZoomEventHandler
WDDeskImitation	WDMainMenu	WildDocs
de.atzenbeck.wilddocs.comparators		
WDIndexComparator		
de.atzenbeck.wilddocs.documents		
WDDocument		
de.atzenbeck.wilddocs.documents.adornments		
WDAdornment	WDShadow	
WDLowLevelDocBorder	WDShadowSurrounding	
de.atzenbeck.wilddocs.documents.bindings		
WDBinding	WDDesk	WDPrimitiveBinding
WDBook	WDPage	WDSheet
de.atzenbeck.wilddocs.documents.bindings.mechanisms		
WDBindingAreaMechanism	WDBindingPointMechanism	WDPageMechanism
WDBindingLineMechanism	WDBookMechanism	WDSheetMechanism
WDBindingMechanism	WDDeskMechanism	
de.atzenbeck.wilddocs.documents.lowLevel		
WDBindingCover	WDLowLevelDoc	WDStyledText
WDImage	WDShape	WDText
de.atzenbeck.wilddocs.filters		
AdornmentFilter	IntersectionFilter	OpenBindingMechanismFilter
BindingMechanismFilter	LargerNodeIndexFilter	PrimitiveBindingFilter
ChildrenFilter	LowLevelDocFilter	ShadowFilter
ClusterOnTopFilter	NodeInBetween	SmallerNodeIndexFilter
DescendentFilter	NodesOnLayer	WDFilter
DocumentFilter		
de.atzenbeck.wilddocs.machines		
WDBindingClipCalculator	WDNodeDragger	WDNodeRotator
WDClusterRecognizer	WDNodeFactory	WDUnitConverter
WDDocTurner	WDNodeIndexPusher	
de.atzenbeck.wilddocs.storages		
ObjectStore	WDObjectStore	WDTempNodeStorage
de.atzenbeck.wilddocs.util		
FileChooser	WDRotationPoint	WDTextLoader
WDBox	WDRubberBand	WDTextSaver

Table 4.1.: WildDocs packages and classes

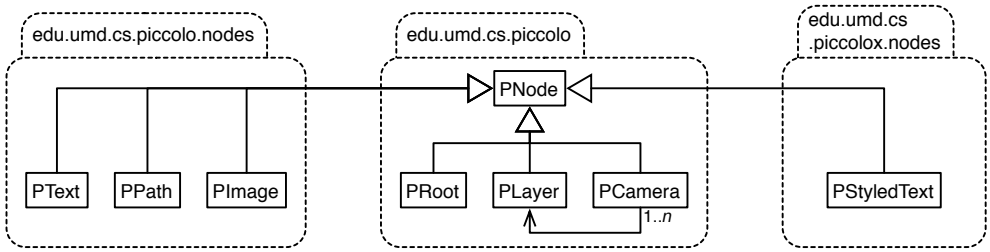


Figure 4.1.: Overview of Piccolo core classes (based on Bederson et al., 2004, Fig. 9) and class for styled text support

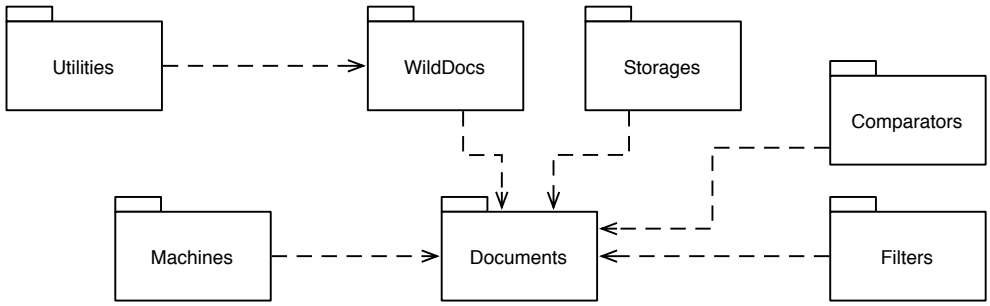


Figure 4.2.: WildDocs packages overview

Piccolo also supports WildDocs with basic nodes for text (PText and PStyledText), images (PImage), and generic lines (PPath). The term "node" is mostly *not* used in WildDocs, for which we built a terminology that is based on names of real items, such as *book*, *page*, or *document*. However, in this chapter we will use "node" for all instances where the focus is rather on the Piccolo based coding than on the WildDocs concept.

Figure 4.2 shows the logical WildDocs packages and how they relate to each other. The package "WildDocs" (de.atzenbeck.wilddocs) contains classes for managing the space and interaction related tasks. We discuss those in various sections, mainly in Sect. 4.1.2–4.1.4 and 4.4. The WildDocs space supports documents to be placed. They are part of the package "Documents" (documents), presented in Sect. 4.2. The "Machines" package (machines) takes care of behavior that is applied to documents, such as incidental rotation, offset, or unit conversion. We explain those in detail in Sect. 4.3. "Comparators" (comparators) and "Filters" (filters), discussed in Sect. 4.5, are built for node comparisons and filtering. The package "Storages" (storages) holds classes for saving and loading documents persistently or temporarily, as shown in Sect. 4.5.3. Finally, "Utilities" (util) provides additional tools for WildDocs, which we present in Sect. 4.4.3 and 4.5.

4.1.2. Main Class

WildDocs's main class is WildDocs. With its 1,910 lines of code it is the largest class among all other WildDocs classes. It extends Piccolo's class PFrame. In order to provide a better

overview, we divided the code into several parts, which we will describe in the following sections.

Constants, Preference Switches, and Preference Values

Line 96–307 contain preference switches and values, as well as other constants. At the current version, preferences can be set via static variables. They allow configuration of new WildDocs instances. For a later version, those preferences should be moved to a preference window that allows the user to change and activate them during runtime.

We distinguish between preference switches (which can hold either ON or OFF) and preference values (which can hold any value or object). Table 4.2 lists all preference switches. They are divided into five groups, which we will describe in the following. Preference values hold mainly visual attributes, such as shadow transparency, offset, or color, but also menu zoom factors, grid spacing, or desk size. They are of less interest, because they do not add main features, but rather modify the appearance. We will not focus on them.

Zooming and Navigation Preferences Switches for zooming include smooth zooming, menu zoom, and quickzoom. Additionally, keyboard shortcuts for menu zoom can be switched on or off as well as whether there should be a mark for the departure area when quickzoom fully zooms out. Also scrollbars can be enabled or disabled.

Rotation and Sloppiness Preferences Rotation and sloppiness preferences allow to switch on or off purposeful rotation, incidental rotation (named RANDOM_ROTATION), and random offset. The latter is used for positioning newly loaded or automatically dragged documents.

Because we experienced problems with incidental rotation for dragged nodes on systems other than Mac OS X, we had to introduce a working, but simpler incidental rotation behavior, as explained in Sect. 4.3.1. The preference switch RANDOM_ROTATION_ONLY_AT_| ↩ ↪ |MOUSE_RELEASE toggles between the more complex, animated, and better simulated incidental rotation and the simple rotation implementation. This switch has only effect when RANDOM_ROTATION is set to ON.

Additional sloppiness related preferences exist for grid support or straighten stacks, for example, via `CTRL-S`. If grid is on, another switch is checked to see if it should be visually represented with lines or whether it should be invisible and apply only its behavior. Grids are currently in experimental state. They work most of the time correctly; however, they may show strange behavior sometimes.

Fixed Size Documents Preferences One switch among the fixed size supporting features is to *force* fixed sizes at document creation time. The sizes of loaded documents follow then defaults, for example, based on ISO standards. This does not necessarily mean that a document can be resized, which would be another preference switch.

We implemented a switch to *force* nodes to appear small when loaded. Currently, this is used exclusively for WildDocs *v1* to ensure nodes about the size as with other spatial structure applications at load time. There is also a switch for a desk imitation, which we will discuss in Sect. 4.1.4.

Preference Switch	Default
Zooming and Navigation	
– SMOOTHZOOMING	ON
– MENUZOOM	ON
– MENUZOOMSHORTCUT	ON
– QUICKZOOM	ON
– DEPARTUREAREAATQUICKZOOM	ON
– SCROLLBARS	ON
Rotation and Sloppiness	
– PURPOSEFUL_ROTATION	ON
– RANDOM_OFFSET	ON
– RANDOM_ROTATION	ON
– RANDOM_ROTATION_ONLY_AT_MOUSE_RELEASE	ON
– STRAIGHTENSTACK	ON
– GRID	OFF
– GRIDVISIBLE	OFF
Fixed Size Documents	
– FIXEDSIZEDOCS	OFF
– STYLEDTEXTINSETS	OFF
– RESIZEABLEDOCS	ON
– SMALLSIZEONLOAD	OFF
– DESKIMITATION	ON
General Settings	
– FULLSCREENATSTART	OFF
– FULLSCREENMENU	OFF
– INDEXPUSHER	ON
– AUTOCLICKONMOUSEOVER	OFF
– LOWLEVELDOCBORDER	OFF
– MANUELINDEXPUSH	ON
– MANUELSIDEPUSH	ON
– DELETEDOCUMENTS	ON
– KEYRUBBERBANDSELECTION	ON
Experimental Settings	
– SHADOW	OFF
– MOUSERUBBERBANDSELECTION	OFF
– PRIMITIVEBINDINGS	OFF
– COMPLEXBINDINGS	OFF
– DESK	OFF
– CANVAS_OR_DESK_PICKABLE	ON

Table 4.2.: Preference switches and default settings

General Settings WildDocs's general settings include switches for full screen mode at startup, adding "Toggle Fullscreen Mode" to the "Window" menu, and for panning the background. The index pusher, discussed in Sect. 4.3.4, can be completely switched off. It also can be decided to activate the feature of automatically changing the active node when the mouse hits a node while dragging another one, as described in Sect. 4.4.2. LOWLEVELDO| ⤸ ⤷ |CBORDER sets whether low level documents are surrounded by an additional frame. Low level documents are discussed in Sect. 4.2.2.

There is a range of settings for structure manipulation.[2] There are switches for pushing the node to the back or to the front (MANUELINDEXPUSH), usually performed by pressing `CTRL-D` or `CTRL-U`, or to push the node below the cursor to the left or right (MANUELS| ⤸ ⤷ |IDEPUSH), usually performed by pressing `CTRL-L` or `CTRL-R`.

The possibility of deleting documents, available at the "Document" menu, can be enabled by setting DELETEDOCUMENTS to ON. Finally, the selection of a range of nodes via rubber band, created by menu or keyboard, can be enabled through a preference switch. Its functionality is discussed in Sect. 4.4.3.

Experimental Settings Several preference switches are available in conjunction with functionality that is not implemented completely or working in an experimental state. This includes support for shadows or mouse rubber band selection.

As we will discuss in Sect. 4.2.3, WildDocs has partly implemented primitive and complex bindings. There are preference switches for both types to enable or disable them independently from each other. They are in experimental state.

There is a switch for extended desk support, named DESK. This is a real WildDocs binding, not an adornment as the above mentioned desk imitation (DESKIMITATION). It is experimental and currently not completely functional. We will discuss it in Sect. 4.2.3.

CANVAS_OR_DESK_PICKABLE is indirectly responsible of setting the pickable flag at the WDLayer instance to enable panning of the canvas if set to ON. It is planned, that dragging events on the desk binding are passed through to result in panning the background instead.[3] Currently, this switch is not implemented for the desk binding.

Initialization and Reset

The constructor takes an instance of PCanvas as mandatory attribute. It calls the superclass constructor with the given canvas instance and requests to disable full screen mode. The associated canvas, which is the one that was passed to the superclass, is then casted to WDCanvas and the current WildDocs instance is associated at the canvas's instance. The constructor enables or disables full screen mode according to the preference switch FULLSCREENAT| ⤸ ⤷ |START, and creates and associates an instance of WDObjectStore.

WildDocs is started via its main method, which contains one line:

[2] The class WildDocs also declares the static variable BROWSESTACKS (line 224), which we ignore, because it is currently not used anywhere else. It also does not appear in Tab. 4.2. This constant was replaced by MANU| ⤸ ⤷ |ELSIDEPUSH. – There is a typographic error for MANUELSIDEPUSH and MANUELINDEXPUSH. The fifth letter of both should be an "A" instead of an "E".

[3] It would not be practicable to drag the desk. The user would have to find a spot outside the desk that allows him/her to drag the background. However, because desk bindings may be large, it is likely that most of the time there is no such a spot at the visible area. It also would be a metaphor break, since a real desk usually is not moved for navigation. Instead, the person's view point changes, which is equivalent to panning the canvas in WildDocs.

Preference Switch	*v0*	*v1*	*v2*	*v3*	*v4*
SMOOTHZOOMING	ON	OFF	ON	OFF	OFF
MENUZOOMSHORTCUT	ON	OFF	ON	OFF	OFF
QUICKZOOM	ON	OFF	ON	OFF	OFF
SCROLLBARS	ON	ON	OFF	ON	ON
PURPOSEFUL_ROTATION	ON	OFF	OFF	ON	OFF
RANDOM_OFFSET	ON	OFF	OFF	ON	OFF
RANDOM_ROTATION	ON	OFF	OFF	ON	OFF
STRAIGHTENSTACK	ON	OFF	ON	OFF	ON
RESIZEABLEDOCS	ON	ON	OFF	OFF	OFF
SMALLSIZEONLOAD	OFF	ON	OFF	OFF	OFF
DESKIMITATION	ON	OFF	ON	ON	ON
INDEXPUSHER	ON	OFF	ON	ON	ON
LOWLEVELDOCBORDER	OFF	ON	OFF	OFF	OFF
MANUELSIDEPUSH	ON	OFF	ON	ON	ON

Table 4.3.: Individual preference switches for WildDocs versions

[WildDocs.java | main(String[])]

```
1907    public static void main(String[] args) {
1908        changeToCompleteSettings();
1909    }
```

changeToCompleteSettings() is one of five methods that enables predefined and version specific sets of features. In this case it sets its preferences to WildDocs *v0*. The other methods are called when WildDocs *v1*, *v2*, *v3*, or *v4* are instantiated. Table 4.3 depicts all settings that may differ from the default, represented in Tab. 4.2. Each of those five methods calls resetWildDocs(String) with the WildDocs version name as parameter. The name appears, for example, in the window title.

After setting the title and creating a new WildDocs instance, the statistics of the previous instance are compiled and written in a file. Its window is then closed and the native screen resources released:

[WildDocs.java | resetWildDocs(String)]

```
1833    protected static WildDocs resetWildDocs(String aTitle) {
1834        WildDocs lastWD = CURRENT_WILDDOCS_INSTANCE;
1835        WildDocs wilddocs = new WildDocs(new WDCanvas());
1836        wilddocs.setTitle(aTitle);
1837
1838        if (lastWD != null) {
1839            lastWD.showStatistics();
1840            // wilddocs.setSize(lastWD.getSize());
1841            // wilddocs.setLocation(lastWD.getLocation());
1842            lastWD.setVisible(false);
1843            lastWD.dispose();
1844        }
1845
1846        CURRENT_WILDDOCS_INSTANCE = wilddocs;
```

The following part sets some attributes, which are then put in place:[4] the menu bar; a node factory that is used to create nodes; and, the window background, which is set to a given default color:

[WildDocs.java | resetWildDocs(String)]

```
1853        PCanvas canvas = wilddocs.getCanvas();
1854        wilddocs.setMainMenuBar(new WDMainMenu(wilddocs));
1855        wilddocs.setNodeFactory(new WDNodeFactory(wilddocs));
1856        // wilddocs.setShadowLayer(new PLayer());
1857        canvas.setBackground(WILDDOCSDEFAULTCOLOR);
```

Statistics are very important for our usability test. In order to avoid losing the statistics accidentally by exiting the application, we added a shutdown hook. This causes WildDocs to create the statistics and save them to a file before quitting:

[WildDocs.java | resetWildDocs(String)]

```
1874        Runtime.getRuntime().addShutdownHook(new Thread() {
1875            public void run() {
1876                CURRENT_WILDDOCS_INSTANCE.showStatistics();
1877            }
1878        });
```

The last part of resetWildDocs(String) deals with preferences and enables features according to them:

[WildDocs.java | resetWildDocs(String)]

```
1880        if (MOUSERUBBERBANDSELECTION)
1881            canvas.addInputEventListener(new WDDeskInputEventHandler(wilddocs));
1882
1883        if (GRIDVISIBLE)
1884            wilddocs.paintGrid();
1885
1886        if (SCROLLBARS)
1887            wilddocs.activateScrollbars();
1888
1889        if (DESK)
1890            wilddocs.createDesk();
1891        if (DESKIMITATION)
1892            wilddocs.setDeskImitation(new WDDeskImitation(wilddocs, DESKWIDTH,
1893                    DESKHEIGHT));
1894
1895        if (!PANNING)
1896            canvas.setPanEventHandler(null);
1897
1898        return wilddocs;
1899    }
```

[4] Usually, this would be placed inside the method initialize(), which overrides the method in PFrame and is called during initialization. In this case, however, we experienced problems with GUI parts, such as click behavior for scrollbars.

Document Loading and Creation

Line 840–1200 include mainly methods that are related to document creation, including showing the license as WildDocs document on the screen or creating statistics. Statistics are discussed in its own section, starting on page 90.

Adding Documents to the Space There are methods for saving or loading the complete object store. Both are triggered from the WildDocs instance. It delegates the task to the associated instance of WDObjectStore. This is still experimental and currently not fully functional. We will describe the object store more detailed in Sect. 4.5.3.

There is a central method for adding documents to the WildDocs space:

[WildDocs.java | addNodeToDesk(WDDocument)]

```
1134    public void addNodeToDesk(WDDocument aDoc) {
1135        addNodeToDesk(aDoc, WITHRANDOMROTATION, WITHRANDOMOFFSET);
1136    }
```

This method delegates the creation, assuming that incidental rotation and offset are intended:

[WildDocs.java | addNodeToDesk(WDDocument,boolean,boolean)]

```
1146    public void addNodeToDesk(WDDocument aDoc, boolean ifRandomlyRotated,
1147            boolean ifRandomOffset) {

1151        if (getDesk() == NO_DESK) {
1152            getLayer().addChild(aDoc.toPNode());
1153        } else {
1154            getDesk().getBindingMechanism().addDocument(aDoc);
1155        }
1156
1157        // Rotate and apply offset to the node.
1158        if (ifRandomlyRotated) {
1159            new WDNodeRotator(aDoc.toPNode()).rotateRandomly();
1160        }
1161
1162        if (ifRandomOffset) {
1163            new WDNodeDragger(this).dragRandomly(aDoc,
1164                    WDNodeDragger.HUGE_MAX_RANDOM_OFFSET);
1165        }
1166
1167        if (aDoc instanceof WDLowLevelDoc && RESIZEABLEDOCS) {
1168            WDBoundsHandle.addBoundsHandlesTo(aDoc);
1169        }
1170
1171        if (aDoc instanceof WDLowLevelDoc && LOWLEVELDOCBORDER) {
1172            ((WDLowLevelDoc) aDoc).createBorder();
1173        }
1174
1175        updateShadows();
1176    }
```

Adding a document depends on the given settings. The if-clause at line 1151 checks if there is a desk binding associated.[5] If there is one, the document is added to the desk's binding mechanism. Otherwise, it is added to the associated WDLayer instance.

Parameters about whether incidental rotation or offset are enabled were passed to this method. If set to true, they trigger an instance of WDNodeRotator or WDNodeDragger to perform the appropriate action.

If the given document is an instance of WDLowLevelDoc (see Sect. 4.2.2) and the preference setting allows to resize documents, bounds handles are added to it. Similar to this, a border may be painted around the document's bounds on WDLowLevelDoc instances. Finally, shadows are updated.

Importing Files Files of the type GIF, JPEG, PNG, plain text, HTML, or RTF can be loaded directly into WildDocs. They are added as documents on the WildDocs space. The menu entry "Import Documents..." calls the method loadNode() in WildDocs:

[WildDocs.java | loadNode()]

```
1102    public void loadNode() throws IOException {
1103        boolean switchBackToFullscreen = false;
1104        if (getCurrentlyFullscreen() == ON) {
1105            switchBackToFullscreen = true;
1106            setFullScreenMode(OFF);
1107        }
1108
1109        File [] fileSelection = FileChooser.getFiles(this);
1110        if (fileSelection.length != 0) {
1111            for (int i = 0; i < fileSelection.length; ++i) {
1112                URI fileURI = fileSelection[i].toURI();
1113                WDDocument doc = (WDDocument) getNodeFactory().newNode(fileURI);
1114
1115                if (SMALLSIZEONLOAD) {
1116                    doc.toPNode().setWidth(SMALLLOADSIZE.getWidth());
1117                    doc.toPNode().setHeight(SMALLLOADSIZE.getHeight());
1118                }
1119
1120                addNodeToDesk(doc);
1121            }
1122        }
1123
1124        if (switchBackToFullscreen == true) {
1125            setFullScreenMode(ON);
1126        }
1127    }
```

Line 1109 opens a file dialog window that allows to select one or multiple files. The file URIs are individually passed to the associated instance of WDNodeFactory, which produces a WDDocument instance for each. If the preferences are set to force small sizes for newly loaded documents, the appropriate width and height are applied. Finally, the node is added to the desk or layer.

[5] This is a desk binding (see Sect. 4.2.3), not a desk imitation (see Sect. 4.1.4).

The if-clauses at the method's beginning (line 1104) and ending (line 1124) are necessary because of a problem we discovered with opening a file dialog window while WildDocs is in full screen mode. We tested this on Mac OS X with an active keyboard shortcut for importing documents (CTRL−O).[6] When activated while the full screen mode was on, the complete display went black until ESC was pressed. Therefore, we introduced a simple mechanism that checks if full screen is currently activated. If on, it switches full screen off and changes back after loading the documents.

It can be argued that this method should be moved to WDNodeFactory, since it belongs strongly to node creation.

Creating Bindings and "Internal" Text Documents Currently, WildDocs has partly implemented bindings. Their instantiation is activated by selecting the appropriate entry in the "Bindings" menu. This calls the desired action in WildDocs. New bindings are easy to add to the system. We partly implemented book, sheet, and a generic primitive binding. The trigger for creating a new book binding demonstrates exemplary the instantiation of a binding:

[WildDocs.java | createBook()]

```
898         WDBook book = new WDBook(this, WDBook.STANDARDMAXTHICKNESS, Color.blue,
899                 WDDocument.A4);
900         addNodeToDesk(book);
```

The new WDBook instance is instantiated by passing additional information, such as the maximum thickness, color, or size. Then, the newly created book is added to the space.

Internal texts, such as license and copyright information, or statistics, can be displayed as WildDocs documents. An existing string is simply passed to WDNodeFactory, which returns an instance of WDText. This may be modified before it is added to the WildDocs space. The following example creates a document with the software license, which existed originally as string:

[WildDocs.java | showLicense()]

```
954         WDLowLevelDoc licenseDoc = (WDLowLevelDoc) getNodeFactory().newNode(
955                 license.replaceAll("@year@", year));
956         licenseDoc.toPNode().setPaint(Color.YELLOW);
957
958         if (SMALLSIZEONLOAD) {
959             licenseDoc.toPNode().setWidth(SMALLLOADSIZE.getWidth());
960             licenseDoc.toPNode().setHeight(SMALLLOADSIZE.getHeight());
961         }
962
963         addNodeToDesk(licenseDoc);
964         return licenseDoc;
```

Changing the Desk Maximum one desk binding (WDDesk), as described in Sect. 4.2.3, is associated to the running WildDocs instance. If a new desk is set, all children are moved to the new one and the association to the old becomes overwritten:

[WildDocs.java | setDesk(WDDesk)]

```
588     public void setDesk(WDDesk aDesk) {
```

[6] This keyboard shortcut was disabled later, because it was not needed for the usability test.

```
589         passDeskChildren(getDesk(), aDesk);
590         if (getDesk() != NO_DESK) {
591             getDesk().getParent().removeChild(getDesk());
592         }
593         if (aDesk != NO_DESK) {
594             addNodeToDesk(aDesk, WITHOUTRANDOMROTATION, WITHOUTRANDOMOFFSET);
595         }
596         desk = aDesk;
597     }
```

Passing the children from the old desk to the new one is triggered at line 589 and executed by passDeskChildren(WDDesk,WDDesk):

[WildDocs.java | passDeskChildren(WDDesk,WDDesk)]

```
1192    private void passDeskChildren(WDDesk aOldDesk, WDDesk aNewDesk) {
1193        if (aOldDesk != NO_DESK) {
1194            if (aNewDesk == NO_DESK) {
1195                getLayer().addChildren(aOldDesk.getChildrenReference());
1196            } else {
1197                aNewDesk.addChildren(aOldDesk.getChildrenReference());
1198            }
1199        }
1200    }
```

The positions of the documents will not change. If the new desk is smaller or has a different position than the old one, documents may be placed outside its bounds. The semantics would be that the documents were put onto the floor. It would be possible to write a method that uses WDNodeDragger to drag the objects onto the new space; however, this may cause confusion by the user.

It is not clear under what circumstances a desk could be changed and what benefits a user would gain. On the other side, it also would be an option to support an arbitrary amount of desks, possibly not allowing to put them on top of each other. This would cause an obvious metaphor break.

Statistics Support

There are variables and methods for counting specific WildDocs features, such as how often a user activated the shortcut for straightening a stack:

[WildDocs.java | increaseStatStraightenStack(int)]

```
784    public void increaseStatStraightenStack(int aNumber) {
785        statStraightenStack += aNumber;
786    }
```

A complete list of logged actions can be seen at the log file example in Sect. B.2. Methods for increasing those counts are mostly called at the part of code that initiates the counted action.

The menu entry “Show Statistics” creates a string that contains the desired data. A new string is only created one time for each WildDocs instance. New statistics become appended to old ones. They include the current time in milliseconds and the hash code of the WildDocs instance. If the check box at the menu entry “Save Statistics automatically” is on, the statistics

are saved as text file by the following code. The current milliseconds and the instance hash are also part of the file name:

[WildDocs.java | showStatistics()]

```
1078        if (getMainMenuBar().isSaveStatistics()) {
1079            String fileName = "WildDocs−" + millisNow + "msec_id" + id + ".log";
1080            try {
1081                WDTextSaver.save(fileName, stats);
1082            } catch (IOException e) {
1083                e.printStackTrace();
1084            }
1085        }
```

Finally, a document with the statistics is created and added to the WildDocs space by calling addNodeToDesk(WDDocument). The passed document was created by WDNodeFactory and marked with a cyan colored background:

[WildDocs.java | showStatistics()]

```
1066        WDLowLevelDoc statDoc = (WDLowLevelDoc) getNodeFactory().newNode(stats);
1067        statDoc.toPNode().setPaint(Color.CYAN);
```

The menu entry “Save Statistics in File Only” calls showStatistics(), but trashes the added document from the WildDocs space afterwards.[7] It appears to the user that the statistics were only saved as a file.

[WildDocs.java | stampStatistics()]

```
971     public void stampStatistics() {
972         WDDocument stat = showStatistics();
973         stat.trash();
974     }
```

Full Screen and Scrollbars

Full Screen In the used version, we used Piccolo’s PFrame has no method that allows other classes to check whether it is currently in full screen mode. The easiest solution was to add a variable in WildDocs that stores the current mode. The method setFullScreenMode(boolean) overrides a method in PFrame. It delegates the actual action to its superclass and sets the current status locally via setCurrentlyFullscreen(boolean):

[WildDocs.java | setFullScreenMode(boolean)]

```
1209    public void setFullScreenMode(boolean ifFullscreen) {
1210        super.setFullScreenMode(ifFullscreen);
1211        setCurrentlyFullscreen( ifFullscreen );
1212    }
```

For example, this is used by toggleFullScreen(), which changes from full screen mode to window mode and vice versa. It is called by the menu entry “Toggle Fullscreen Mode” or via shortcut `CTRL−F`. The code sets the inverse of the current mode:

[7] A better way would be to move code for statistics creation and saving to stampStatistics(), returning the created statistics string. showStatistics() should call stampStatistics() and add the returned string as document on the desk.

[WildDocs.java | toggleFullscreen()]

```
1217    public void toggleFullscreen() {
1218        setFullScreenMode(!getCurrentlyFullscreen());
1219    }
```

Scrollbars Scrollbars can be activated by activateScrollbars(), which is called from reset| ↩ ↪ |WildDocs(String) if the static preference constant is set to ON. We use Piccolo's support for scrollbars. The code for activateScrollbars() is based on the method initialize() of Piccolo's example class ScrollingExample:

[WildDocs.java | activateScrollbars()]

```
1226    public void activateScrollbars() {
1227        final PCanvas canvas = getCanvas();
1228
1229        final PScrollPane scrollPane = new PScrollPane(canvas);
1230        getContentPane().add(scrollPane);
1231
1232        final PViewport viewport = (PViewport) scrollPane.getViewport();
1233        final PScrollDirector windowSD = viewport.getScrollDirector();
1234
1235        viewport.fireStateChanged();
1236        scrollPane.revalidate();
1237        getContentPane().validate();
1238    }
```

After instantiating a new PScrollPane for the associated canvas (line 1229), it is added to the content pane. PViewport (line 1232) and PScrollDirector (line 1233) handle position and size, and give control over the scroll pane.

Grid

The functionality that lets documents snap to a grid when dragged is handled by WDNode| ↩ ↪ |InputEventHandler. It is independent of the grid's visibility. We will discuss this further in Sect. 4.4.3. However, painting the grid is part of the WildDocs instance and performed by paintGrid(). If on WildDocs reset the preference constant GRIDVISIBLE is checked to be ON, the painting is initiated.

The grid is painted on the graphical representation of a WDLayer. The code for paintGrid() is mostly taken from Piccolo's class GridExample that demonstrates the use of a painted grid. Currently, WildDocs uses the default values for grid spacing, which is given in millimeters and converted by an instance of WDUnitConverter (line 1251).

[WildDocs.java | paintGrid()]

```
1249        final PLayer gridLayer = new WDLayer(this) {
1250            // BTW, this is WDLayer.getWildDocs().
1251            double gridspacing = new WDUnitConverter(getWildDocs())
1252                    .wdToJava(GRIDSPACING);
1253
1254            protected void paint(PPaintContext paintContext) {
```

```
1257                double bx = (getX() - (getX() % gridspacing)) - gridspacing;
1258                double by = (getY() - (getY() % gridspacing)) - gridspacing;
```

```
1259                double rightBorder = getX() + getWidth() + gridspacing;
1260                double bottomBorder = getY() + getHeight() + gridspacing;
1261
1262                Graphics2D g2 = paintContext.getGraphics();
1263                Rectangle2D clip = paintContext.getLocalClip();
1264
1265                g2.setStroke(GRIDSTROKE);
1266                g2.setPaint(GRIDPAINT);
1267
1268                for (double x = bx; x < rightBorder; x += gridspacing) {
1269                    GRIDLINE.setLine(x, by, x, bottomBorder);
1270                    if (clip.intersectsLine(GRIDLINE)) {
1271                        g2.draw(GRIDLINE);
1272                    }
1273                }
1274
1275                for (double y = by; y < bottomBorder; y += gridspacing) {
1276                    GRIDLINE.setLine(bx, y, rightBorder, y);
1277                    if (clip.intersectsLine(GRIDLINE)) {
1278                        g2.draw(GRIDLINE);
1279                    }
1280                }
1281            }
1282        };
```

The method paint(PPaintContext) at line 1254 overrides the method in WDLayer. It calculates the grid according to the given spacing, grid stroke, and grid color. After that, the existing layer is replaced by the new grid layer:

[WildDocs.java | paintGrid()]

```
1285        root.removeChild(camera.getLayer(0));
1286        camera.removeLayer(0);
1287        root.addChild(gridLayer);
1288        camera.addLayer(gridLayer);
```

The following code matches the grid layer bounds with the camera view bounds. The grid appears to be infinite:

[WildDocs.java | paintGrid()]

```
1293        camera.addPropertyChangeListener(PNode.PROPERTY_BOUNDS,
1294                new PropertyChangeListener() {
1295                    public void propertyChange(PropertyChangeEvent evt) {
1296                        gridLayer.setBounds(camera.getViewBounds());
1297                    }
1298                });
1299
1300        camera.addPropertyChangeListener(PCamera.PROPERTY_VIEW_TRANSFORM,
1301                new PropertyChangeListener() {
1302                    public void propertyChange(PropertyChangeEvent evt) {
1303                        gridLayer.setBounds(camera.getViewBounds());
1304                    }
1305                });
1306
```

```
1307        gridLayer.setBounds(camera.getViewBounds());
1308    }
```

Zooming

Absolute and Relative Zoom Several methods in WildDocs are directly related to zooming. Zooming is applied to Piccolo cameras. The zoom method for relative zoom calculates the absolute zoom scale and delegates it to zoomAbsolute(double):

[WildDocs.java | zoomRelative(double)]

```
1315    public void zoomRelative(double aRelativeFactor) {
1316        PCamera camera = getCanvas().getCamera();
1317        zoomAbsolute(camera.getViewScale() * aRelativeFactor);
```

zoomAbsolute(double) sets the camera's view scale to the given absolute zoom scale:

[WildDocs.java | zoomAbsolute(double)]

```
1331    public void zoomAbsolute(double aAbsoluteFactor) {
1332        PCamera camera = getCanvas().getCamera();
1333        camera.setViewScale(aAbsoluteFactor);
```

The following line causes the next quickzoom command to zoom completely out at next call. This happens every time this method is called, for example, by activating a menu zoom action:

[WildDocs.java | zoomAbsolute(double)]

```
1342        setZoomFrameAtQuickZoomOut(WildDocs.FULLZOOMOUT);
1343    }
```

Quickzoom Quickzoom can be triggered by the menu item "Toggle Quickzoom", but is usually activated by pressing `CTRL-Z`. This calls toggleQuickZoom(). Quickzoom zooms out completely until all documents including the desk imitation can be seen on the screen. The next quickzoom activation zooms back to the original zoom scale to where the mouse pointer is located at that time.

Quickzoom – Calculation of Full Zoom Bounds Because quickzoom uses complete zoom out, we wrote the method calcFullZoomBound() which calculates the destination bounds. This only works if there are documents or a desk imitation on the WildDocs space. There is an if-clause at the very beginning that tests this case:

[WildDocs.java | calcFullZoomBounds()]

```
1420    protected PBounds calcFullZoomBounds() {
1421        if (getLayer().getChildrenCount() > 0) {
```

If there are documents or a desk imitation on the WildDocs space, we create temporarily a rectangle that represents the current camera view area on the space:

[WildDocs.java | calcFullZoomBounds()]

```
1429            double scale = getCanvas().getCamera().getViewScale();
1430
1431            PBounds bounds = getCanvas().getCamera().getFullBounds();
1432            Point2D centerLocal = bounds.getCenter2D();
```

```
1433            Point2D center = getCanvas().getCamera().localToView(centerLocal);
1434            double width = bounds.getWidth() / scale;
1435            double height = bounds.getHeight() / scale;
1436
1437            PPath scaleBoundsGraphics = PPath.createRectangle(0, 0,
1438                    (float) width, (float) height);
1439            scaleBoundsGraphics.centerBoundsOnPoint(center.getX(), center
1440                    .getY());
```

Firstly, the camera's scale and bounds are calculated. The center of the auxiliary rectangle is the center of the camera. We need to transform its coordinates, because the camera's bounds do not represent the current scale. The correct width and height of the rectangle on the space are calculated by dividing the camera's width and height by the current scale (line 1434–1435). Finally, the rectangle is created and centered on the calculated center point (line 1439).

The bounds of the new rectangle are calculated and associated for later use. They represent the departure bounds from where the quickzoom command will zoom away:

[WildDocs.java | calcFullZoomBounds()]

```
1446            PBounds scaleBounds = scaleBoundsGraphics.getFullBounds();
1447
1448            setZoomFrameAtQuickZoomOut(scaleBounds);
```

The next part of the method deals with calculating the arrival bounds to where quickzoom is supposed to zoom to. These bounds surround all existing documents, including desk imitation, if existing:

[WildDocs.java | calcFullZoomBounds()]

```
1453            WDTempNodeStorage allNodes = new WDTempNodeStorage();
1454            allNodes.addAll(getLayer().getAllNodes());
1455            allNodes.keepFiltered(new DocumentFilter());
1456
1457            // Add the desk imitation if available.
1458            if (getDeskImitation() != null) {
1459                allNodes.add(getDeskImitation());
1460            }
```

Firstly, all WDDocument instances are filtered (line 1454). This includes an existing desk binding, but *not* a desk imitation. This would lead to behavior that would be unexpected and not obvious for the user, who may interpret a desk binding and desk imitation as being from the same source. Therefore, an existing desk imitation is also added to the list of documents at line 1459.

An iterator iterates through the collection, adding the bounds of each object to the bounds of a PBounds instance that was created only for that purpose. After finishing, these bounds surround all given documents as well as the desk imitation:

[WildDocs.java | calcFullZoomBounds()]

```
1462            PBounds span = new PBounds();
1463
1464            Iterator iterator = allNodes.iterator();
1465            while (iterator.hasNext()) {
1466                PNode node = (PNode) iterator.next();
1467                span.add(node.getFullBounds());
```

```
1468            }
```

The bounds object is returned on line 1481. If there are no objects on the WildDocs space, then the if-clause at line 1421 returns false, and the previously discussed calculations are not performed. Instead, null is returned on line 1483:

[WildDocs.java | calcFullZoomBounds()]

```
1481            return span;
1482        } else {
1483            return null;
1484        }
```

Quickzoom – Zoom Out The method toggleQuickZoom() supports context-based zooming out or zooming in. The first part handles zooming out.

[WildDocs.java | toggleQuickZoom()]

```
1354    public void toggleQuickZoom() {
```

```
1360        if (getZoomFrameAtQuickZoomOut() == FULLZOOMOUT
1361                || getZoomFrameAtQuickZoomOut() == null) {
1362            PBounds fullZoomOut = calcFullZoomBounds();
1363
1364            if (fullZoomOut == null) {
1365                System.err.println("WARNING: No fullzoom performed, "
1366                        + "because there are currently no nodes.");
```

The if-clause at line 1360 uses getZoomFrameAtQuickZoomOut(). It returns the bounds of the previous viewport at the time quickzoom was triggered to zoom out. A returned value null indicates that there is no previous viewport stored. In this case, WildDocs continues with the full zoom out sequence. The same happens when the recent zoom scale is set to FULLZOOMOUT.[8]

Quickzoom's full zoom out works only if there are objects on the desk. The destination view will show all of them on the screen. Line 1362 requests the bounds that go around all documents and desk imitation on the space. The following if-clause returns an error message if the bounds are null. This would mean that there are no objects that can be surrounded. Otherwise, an animation toward the calculated bounds is triggered:

[WildDocs.java | toggleQuickZoom()]

```
1367            } else {
1368                animateZoom(fullZoomOut);
```

The method animateZoom(PBounds) creates an instance of PActivity which contains information to animate the current camera view to the center of the given bounds. It is then scheduled with the root and executed:

[WildDocs.java | animateZoom(PBounds)]

```
1493    protected void animateZoom(PBounds aBounds) {
1494        // Change the duration according to your needs.
1495    long zoomDuration = 300;
1496
```

[8] Currently, the constant FULLZOOMOUT is set to null. The reason why the if-clause at line 1360 checks both is that there is no guarantee that the constant will continue to have this value in the future.

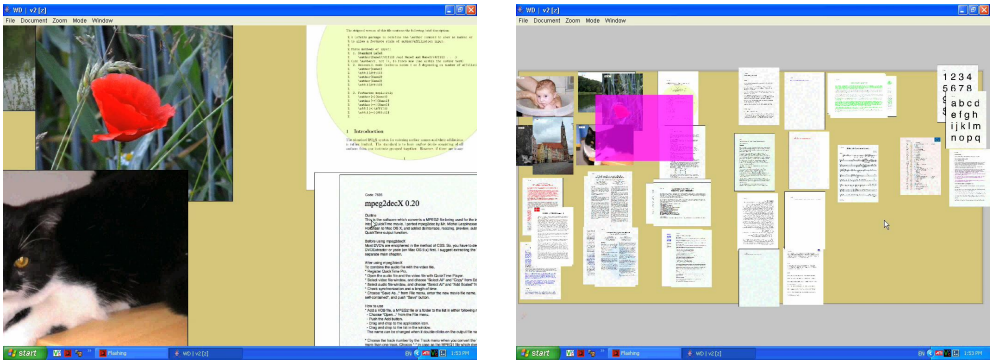


Figure 4.3.: Screenshots of viewport before (left) and after quickzoom's complete zoom out, showing quickzoom's fading out magenta colored destination rectangle (right)

```
1497        PActivity cameraMovement = getCanvas().getCamera()
1498                .animateViewToCenterBounds(aBounds, true, zoomDuration);
1499
1500        /*
1501         * Note that an activity will not run, unless it is scheduled with the
1502         * root!
1503         */
1504        if (cameraMovement != null) {
1505            getCanvas().getRoot().addActivity(cameraMovement);
1506            cameraMovement.setStartTime(System.currentTimeMillis());
1507        }
1508    }
```

The zoom duration is currently set to 300 ms (line 1495), providing fast, but smooth animation "to maintain the identity of objects in their contexts" (Ware, 2004, 343). Another orientation supporting feature is Quickzoom's fading out magenta colored transparent rectangle that indicates the destination area. This helps the user to experience his/her previous view area for a short period of time when zoomed out. An example is depicted in Fig. 4.3. The following code snippet performs the destination rectangle:

[WildDocs.java | toggleQuickZoom()]

```
1369                PBounds camBounds = getCanvas().getCamera().getViewBounds();
1370
1371                if (DEPARTUREAREAATQUICKZOOM) {
1372                    /*
1373                     * Show a fading out rectangle to show the departure area.
1374                     */
1375                    Point2D camPos = camBounds.getOrigin();
1376                    WDRubberBand rubber = new WDRubberBand(this, camPos.getX(),
1377                            camPos.getY(), camBounds.getWidth(), camBounds
1378                                    .getHeight(), Color.MAGENTA,
1379                            new BasicStroke(0), Color.MAGENTA);
1380                    rubber.removeFromWildDocs(WDRubberBand.LONGDURATION);
1381                }
```

```
1382            }
```

After the view bounds are returned, it is checked if the preference entry for departure area is set to ON (line 1371). If yes, a new WDRubberBand instance is created that fits the camera's size and origin (line 1376). Color and stroke are set to magenta. Right after its creation, the rubber band is triggered to fade out and remove itself from the WildDocs space. The constant for long duration (LONGDURATION) is currently set to 5 seconds. We will discuss rubber bands more extensively in the context of selection (Sect. 4.4.3).

Quickzoom – Zoom (Back) In If the if-clause at line 1360 shows that the previous zoom viewport is set to FULLZOOMOUT or that there is no viewpoint stored, WildDocs is already fully zoomed out. The action will zoom in to where the cursor is located at that moment. The previous viewport dimensions will be used. Zooming back is also animated:

[WildDocs.java | toggleQuickZoom()]

```
1384        } else {
1385            /*
1386             * Zoom back to the original scale frame.
1387             */
1388
1389            // Check the current cursor position.
1390            Point2D mouse = currentMousePositionOnCamera();
1391
1392            // Dummy node to center bounds (destination)
1393            PPath dummyDest = new PPath(getZoomFrameAtQuickZoomOut());
1394            dummyDest.centerFullBoundsOnPoint(mouse.getX(), mouse.getY());
```

```
1408            animateZoom(dummyDest.getFullBounds());
1409
1410            // Next time, do a full zoom out.
1411            setZoomFrameAtQuickZoomOut(FULLZOOMOUT);
1412        }
1413    }
```

The destination view is centered at the cursor's position. currentMousePositionOnCamera() simply returns the current mouse position coordinates on the camera, transformed to the view coordinate system:[9]

[WildDocs.java | currentMousePositionOnCamera()]

```
1805    public Point2D currentMousePositionOnCamera() {
```

```
1810        return getCanvas().getCamera().localToView(
1811                currentMousePositionOnCanvas());
1812    }
```

After the mouse position is returned, a dummy node is created with the previous viewport dimensions (line 1393). The node is added to the WildDocs space. Its only purpose is to be centered at the cursor position to mark the zoom destination area. Finally, an animated zoom to the dummy node's bounds is called at line 1408. The previous zoom viewport is set to FULLZOOMOUT. This causes the next quickzoom command to zoom out completely next time.

[9] Apparently, also transformations to other coordinate systems work, such as parent to local, global to local, local to global, and local to parent.

Moving Documents

Push Node Up/Down Beside moving documents on the screen via Piccolo's support for dragging, WildDocs has additional ways to alter a document's position. One is to push a node to the front ("up") or to the back ("down"). These are called by WDMainMenu, for example, through pressing `CTRL-U` or `CTRL-D`. The action affects the document below the cursor.

[WildDocs.java | pushNodeUpOrDown(int)]

```
1543    public void pushNodeUpOrDown(int aUpOrDownMark) {
1544        PNode node = getLastMouseOverOnDocument().toPNode();
1545        PNode parent = node.getParent();
```

The requested document below the cursor is returned by getLastMouseOverOnDocument(), which returns a value that is delivered by WDNodeInputEventHandler whenever the mouse is moving above a WDDocument instance. We will discuss this further in Sect. 4.4.3.

[WildDocs.java | pushNodeUpOrDown(int)]

```
1553        if (aUpOrDownMark == UP) {
1554            parent.addChild(node);
1555        } else {
1556            // Take care that the desk is still below!
1557            int idx = 0;
1558            if (DESKIMITATION)
1559                idx++;
1560            if (DESK)
1561                idx++;
1562            parent.addChild(idx, node);
1563        }
1564    }
```

If the provided move direction mark equals UP, the document is simply added to its parent. This results in assigning the highest index to the node. It is therefore displayed above its siblings.

Otherwise, the node's index is set to zero, which may be increased, depending on the existence of a desk metaphor. Indices of all siblings will be increased if this new index number is already taken by another node. A special case occurs if a desk *imitation* exists. Setting the node's index to 0 would cause the document to be displayed below the desk. To avoid this, its index is increased by 1 (line 1559).

From our current point of view, increasing the document's index additionally by 1 when a desk *binding* exists, as coded at line 1561, is not necessary (in fact, even wrong). The document that is *on* the desk will *not* be put behind the desk, because it is its parent. In such a case, increasing the document's index to 1 would cause another sibling document to be placed underneath the pushed node. If the document is a sibling to the desk,[10] it may be desired to put the document *below* the desk, which is currently not possible. Following this analysis, we claim that line 1560 and 1561 should be deleted in a future version.

Push Node Left/Right Another way to manipulate a document's position is to move the node below the cursor to the left or to the right by pressing `CTRL-L` or `CTRL-R`. Originally, this behavior was implemented to enable the user to browse stacks efficiently.

[10] This can follow the metaphor of putting a document beside the desk onto the floor.

[WildDocs.java | pushNodeSidewards(int)]

```
1572    public void pushNodeSidewards(int aLeftOrRightMark) {
1573        PNode node = currentMouseOverNode();
1574
1575        // Move only documents
1576        if (node instanceof WDDocument) {
```

The node directly below the cursor is relevant for moving. Only instances of WDDocument are processed.[11] The method currentMouseOverNode() requests the current mouse position, which is used to get the PPickpath instance that leads to the node below the cursor. PCamera's method pick(double,double,double)[12] returns a pick path that has the node below the cursor as last node.

[WildDocs.java | currentMouseOverNode()]

```
1819    public PNode currentMouseOverNode() {
1820        Point2D mousePos = currentMousePositionOnCanvas();
1821        PNode node = getCanvas().getCamera().pick(mousePos.getX(),
1822                mousePos.getY(), 3).getPickedNode();
1823        return node;
1824    }
```

The used method for calculating the mouse position at line 1820 is different to the previously described currentMousePositionOnCamera(), which we explained on page 98. In this case the coordinate system of the *canvas* is relevant, not the camera's. The method looks similar to the camera related one; however, the coordinates do not need to be transformed to another system:

[WildDocs.java | currentMousePositionOnCanvas()]

```
1793    public Point2D currentMousePositionOnCanvas() {
1794        return getCanvas().getRoot().getDefaultInputManager()
1795                .getCurrentCanvasPosition();
1796    }
```

After the node below the cursor is assigned to a variable and it is clear that it is an instance of WDDocument, WildDocs continues with calculating the destination of the active node. Currently, the offset position for the document's center is hard coded to 10 mm outside the document's left or right bound. Support for vertical offset is included, but set to zero. If the push direction is to the left, the horizontal and vertical offsets are multiplied by -1. This is calculated inside the if-clause of line 1595. This causes to change directions. Finally, a newly created WDNodeDragger instance (see Sect. 4.3.2) drags the document with its center to the calculated position. This action is animated.

[WildDocs.java | pushNodeSidewards(int)]

```
1590            double additional = new WDUnitConverter(this).wdToJava(10);
1591            double offsetX = new PPath(node.getFullBounds()).getWidth()
1592                    + additional;
1593            double offsetY = 0;
1594
```

[11] Line 1577–1581 (removed here) is a remnant of an earlier version and without function. Originally it found the most far away ancestor of the given document.

[12] The first two values represent the x and y position of the mouse cursor. The third value specifies the size of the rectangle that is used for picking.

```
1595            if (aLeftOrRightMark == LEFT) {
1596                offsetX *= -1;
1597                offsetY *= -1;
1598            }
1599
1600            WDNodeDragger dragger = new WDNodeDragger(this);
1601            dragger.dragCenterToPosition(node, offsetX, offsetY);
```

This is a quick solution for dragging documents to the side. The destination depends on the individual document size, but does not consider the stack of which the document is part of. A better solution would be to use WDClusterRecognizer (see Sect. 4.3.5) to calculate the stack's bounds and to move the document outside those. A WDNodeIndexPusher instance (see Sect. 4.3.4) could be used to update the indices so that the node that was originally below the previously moved one would be pushed on top of it in case they do not overlap.

Currently it can be the case that a pushed document still intersects partly with the node below. In order to ensure that the next node is put on top of the recent moved one, the pushed node is put above all siblings:

[WildDocs.java | pushNodeSidewards(int)]

```
1612            node.getParent().addChild(node);
1613        }
1614    }
```

Straighten Stacks An action that may be applied to several nodes at the same time is straighten stacks. The shortcut CTRL-S centers all nodes that intersect with the current cursor position. straightenStack() creates a new temporary storage object with all nodes that are located directly below the cursor:

[WildDocs.java | straightenStack()]

```
1619    public void straightenStack() {
1620        WDTempNodeStorage docsBelowCursor = nodesBelowCursorOnLayer();
```

The method nodesBelowCursorOnLayer() returns a WDTempNodeStorage object containing all requested nodes:

[WildDocs.java | nodesBelowCursorOnLayer()]

```
1705    private WDTempNodeStorage nodesBelowCursorOnLayer() {

1710        Point2D mousePos = currentMousePositionOnCamera();
1711
1712        float mouseX = (float) mousePos.getX(); // currentMousePosition().getX();
1713        float mouseY = (float) mousePos.getY(); // currentMousePosition().getY();
```

The first step is to get the mouse position by requesting it from currentMousePositionOn⤸ ⤷Camera(), as already described on page 98. The x and y values of the returned point are stored in different variables. A small rectangle is created at the mouse position. Its width and height are one pixel each. It simulates the cursor during the next steps:

[WildDocs.java | nodesBelowCursorOnLayer()]

```
1719        PPath cursor = new PPath();
1720        cursor.setPathToRectangle(mouseX, mouseY, 1, 1);
```

In the following, all nodes that are attached to the current layer are put into a temporary storage object (see Sect 4.5.3). Then, three filters are applied: Only instances of WD|↩|Document that intersect with the cursor simulation and are direct descendants of the layer are kept.[13] Finally, the storage object containing the remaining documents is returned:

[WildDocs.java | nodesBelowCursorOnLayer()]

```
1733        WDTempNodeStorage docsBelowCursor = new WDTempNodeStorage(getLayer()
1734                .getAllNodes());
1735
1736        docsBelowCursor.keepFiltered(new DocumentFilter());
```

```
1746        docsBelowCursor.keepFiltered(new IntersectionFilter(cursor));
1747
1748        // Currently only direct descendents of the layer are manipulated.
1749        docsBelowCursor.keepFiltered(new DescendentFilter(getLayer()));
1750
1751        return docsBelowCursor;
1752    }
```

After finding the set of documents below the cursor, the method straightenStack() checks now the returned array and continues performing to straighten if it is not empty:

[WildDocs.java | straightenStack()]

```
1622        if (!docsBelowCursor.isEmpty()) {
```

```
1627            PNode highestNode = (PNode) docsBelowCursor.getHighestIndexNode();
1628            double basicRotation = highestNode.getGlobalRotation();
```

```
1634            WDNodeDragger dragger = new WDNodeDragger(this);
1635            dragger.dragCenterToPosition(docsBelowCursor,
1636                    currentMousePositionOnCanvas(),
1637                    WDNodeDragger.SMALL_MAX_RANDOM_OFFSET, basicRotation);
1638        }
1639    }
```

To straighten a stack does not only include minimization of the offset among the documents, but also adjustment of their rotation. WildDocs takes the node with the highest index among the relevant ones and gets its global rotation (line 1628). A newly created WDNode|↩|Dragger instance is triggered at line 1635 to drag the center of the relevant nodes to the current mouse position, including the rotation of the topmost document and a small random offset, which is used if the preference switch for random offset is set to ON. The method for calculating the mouse position (currentMousePositionOnCanvas()) returns the coordinates in the canvas's coordinate system, as described on page 100.

Selecting Nodes There are three methods in WildDocs that handle multiple node selection and movement. We will discuss them detailed in Sect. 4.4.3.

[13] The latter filter (DescendentFilter) is only relevant if bindings are used. Only those are intended to be placed directly on the space. The underlying philosophy is based on the idea that it is the *binding* that needs to be activated in order to move also its content. After the post-supervision of our code, we criticize this. We argue that it may be intended to move a document that is placed inside a binding without moving the binding itself. For example, this would be the case with desk bindings that hold all documents that are placed on them as children. Those cases would not work with the current implementation. However, this discussion is not relevant for our usability test.

Deleting Documents

Similar to node selection, a document can be deleted by moving the cursor on top of it and pressing `CTRL-Backspace`. The node below the cursor will be determined. If it is an instance of WDDocument, it will be destroyed; otherwise, an error message is written.

[WildDocs.java | deleteDocument()]

```
1759    public void deleteDocument() {
1760        PNode node = currentMouseOverNode();
1761
1762        if (node instanceof WDDocument) {
1763            ((WDDocument) node).trash();
1764        } else {
1765            System.err
1766                    . println ("WARNING: The object that had the mouse over "
1767                            + "was not deleted, because it is not a WDDocument instance.");
1768        }
1769    }
```

Another method clears the *complete* space. This can be triggered by the menu entry "Delete ALL documents". All children of the layer will be removed. If desk imitation or desk binding are active, they will be created again afterwards:

[WildDocs.java | deleteAllDocuments()]

```
1774    public void deleteAllDocuments() {
1775        getLayer().removeAllChildren();
1776
1777        // Re-create the desk if desired
1778        if (DESK)
1779            createDesk();
1780
1781        // Re-create the desk imitation if desired
1782        if (DESKIMITATION)
1783            setDeskImitation(new WDDeskImitation(this, DESKWIDTH, DESKHEIGHT));
1784    }
```

A desk *imitation* (see Sect. 4.1.4) is not an instance of WDBinding, whereas the desk *binding* (see Sect. 4.2.3) is. The actual semantics of the menu entry "Delete ALL documents" would include the deletion of an existing desk binding. However, we assume that a desk binding, even though internally a binding and therefore a WDDocument, is not considered as what most people would denote a document. We further assume that a user would be surprised if a desk also would disappear, especially since a desk imitation would not. Instead of following the internals consistently, we implemented behavior that the user assumedly would expect.

4.1.3. Layer

WildDocs's class WDLayer extends Piccolo's PLayer to switch on or off background panning. Its central part is its constructor that checks the preference setting for panning the background at WildDocs (CANVAS_OR_DESK_PICKABLE) and sets the pickable flag accordingly to true or false. If set to true, the user can press the mouse button on the background and drag the background. This is used for navigation. If set to false, pressing the mouse on the background

and moving the mouse does not have any other visual result than moving the mouse without pressing it: The background will not change its position.

4.1.4. Desk Imitation

There is a desk imitation that is used for WildDocs *v0*, *v2*, *v3*, and *v4*, implemented via WDDeskImitation. It extends Piccolo's class PPath and implements WDAdornment. Even though it implements the same interface, it differs from the document adornments presented in Sect. 4.2.2, where we also will discuss WDDeskImitation more detailed. (Figure 4.5 on page 111 depicts relations to other classes.)

Adornments are in package documents.adornments. They are visual add-ons. They strongly depend on the object they are attached to, such as shadows. They do not have any direct user controllable behavior. Similarly, WildDocs's desk imitation does not have a directly controllable behavior. Even panning the background is internally done by the layer instance. However, its semantics differ from document adornments. The desk imitation simulates a desk. Therefore it is not strongly attached to another object. Semantically spoken, there is a stronger relationship to WDDesk, internally handled as document, than to document adornments. In fact, WDDeskImitation should be replaced by WDDesk when complex bindings are switched on. However, WDDeskImitation is not considered as a document, whereas WDDesk is. We will discuss WDDesk in Sect. 4.2.3. The main reason to implement WDDeskImitation was to have a desk metaphor already working before bindings, including WDDesk, would be finished.

The WDDeskImitation constructor calls a method that paints the desk imitation and adds it to the WildDocs space. Attributes used for painting are width and height of the rectangle. They are converted from millimeters to pixels by WDUnitConverter. Further attributes are *x* and *y* position, stroke type and color, desk imitation color and transparency. The desk imitation's pickable flag is set to false. It ignores all mouse input events.

4.2. Documents

4.2.1. General

The central objects in WildDocs are documents. This can be either a binding or a low level document. A binding is an object that can bind other documents. Low level documents are objects that hold data, such as image or text, but cannot bind other documents. Low level documents may have adornments attached, such as visible document bounds or shadows. We discuss low level documents and bindings in Sect. 4.2.2 and 4.2.3 in depth.

All documents implement the interface WDDocument, which extends Java's interface Cloneable. They are part of the package documents. WDDocument defines some constants that are relevant for all documents, such as VISIBLE, NOT_PICKABLE, or constants for paper sizes, such as ISO (e. g., A4 or A5) or ANSI (e. g., Letter or Legal) standards.

Setter and getter implementations handle the associations to related WildDocs instances and to the parent binding mechanisms. Additional implementations need to be coded for get|ClipAlignment() and getThickness(). The clip alignment method is used when a document is added to a binding. The thickness method returns the thickness of a document.

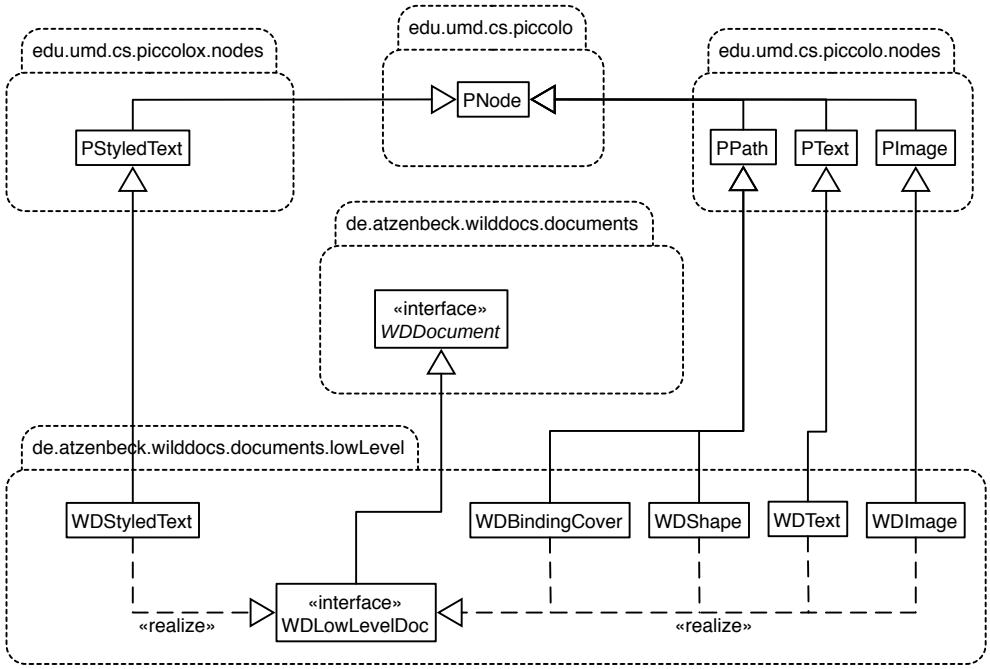


Figure 4.4.: Low level documents class diagram (package documents.lowLevel)

Furthermore, important methods include trash() to delete a document, turn() to turn it, to|PNode() to convert it from a WildDocs WDDocument to a Piccolo PNode, index() to retrieve the index path of the node including all parent nodes, and compareTo(WDDocument) to compare a document's index path to another one's. Selected parts of the interface will be discussed more detailed in those sections where we present implemented classes.

4.2.2. Low Level Documents

Supported Types

Low level document classes can be found in documents.lowLevel. They implement the interface WDLowLevelDoc, an extension of WDDocument. Originally, WildDocs had no notion of binding. Nodes were exclusively nodes provided by Piccolo, such as PText, PImage, etc. When binding were introduced, there was a need to specify the existing document types as non-binding, which only can exist at the very "lowest part" of a structure path. This is where the name "low level document" is derived from. Currently, they still extend some of Piccolo's node classes.

WDLowLevelDoc has setters and getters for setting or retrieving associated WDShadow or WDLowLevelDocBorder instances. Shadow and border are adornments and will be explained in an upcoming section. The interface has also some additional relevant constants, for example, defining default values for low level documents.

Other methods that are implemented by low level documents are removeAllInputEvent|Listener() to remove the node's input event listener, setShadowVisible(boolean) to make

the shadow visible or invisible, updateShadow() to trigger an update of the shadow, create|Border() to create a border around the node, and removeBorder() to remove the associated border line.

Currently, WildDocs has implemented the classes WDImage for images, WDText for plain text, and WDStyledText for HTML or RTF support. There are also WDBindingCover for covers used on bindings, and WDShape for support for arbitrary shapes. However, both are in a very early development state and became partly obsolete by new developments: Binding covers as low level documents are a remnant of a development phase before bindings were introduced that support arbitrary structure levels. With bindings, WildDocs is capable of using arbitrary cover pages that contain images or text or combinations of both.

Constructors of all existing WDLowLevelDoc implementing classes take at least a WildDocs instance as an argument and builds an association to it. Depending on the class, there may be additional arguments used by the constructor that are discussed in one of the following sections. All discussed low level document classes set a clip alignment position. This is used to calculate where the document clips onto a binding. Further, they add a newly created input event handler that takes care of mouse interactions with the node. The background color is set to a default value, if not set already. We use the constructor of WDImage exemplary for all discussed low level document classes:

[WDImage.java | WDImage(WildDocs,URI)]

```
78  public WDImage(WildDocs aWildDocs, URI aFileURI)
79          throws MalformedURLException {
80      super(aFileURI.toURL());
81
82      // set color if necessary
83      if (getPaint() == NO_COLOR) {
84          setPaint(DEFAULTCOLOR);
85      }
86
87      setWildDocs(aWildDocs);
88      setClipAlignment(LOWLEVELDOC_DEFAULTCLIPALIGNMENT);
89
90      addInputEventListener(new WDNodeInputEventHandler(aWildDocs));
91  }
```

The setters setWidth(double) and setHeight(double) exist for all low level documents that are currently implemented. They function equivalently. They override a method in their superclass. The following is the code for setWidth(double) in WDImage:

[WDImage.java | setWidth(double)]

```
141 public boolean setWidth(double aWidth) {
142     if (getBorder() != null) {
143         getBorder().setWidth(aWidth);
144     }
145     return super.setWidth(aWidth);
146 }
```

The purpose of this method is to adjust the change to the document's border, if existing. The border is returned by getBorder() and is an instance of WDLowLevelDocBorder. Then, the new width is passed to the superclass to change also the node itself.

All remaining setters and getters of the discussed low level documents are used to access

private variables, except for getThickness(). This method currently returns the constant LO| ↩ ↪ |WLEVELDOC_DEFAULTTHICKNESS, which contains a default value for a standard paper thickness. The thickness of documents is needed to calculate and evaluate the thickness of bindings. WildDocs does not support bindings completely yet. Therefore, thickness is currently not used.

Among others, the following methods are requested by the interface WDLowLevelDoc and inherited from WDDocument: toPNode() returns the current node as PNode. Since all current low level documents inherit from PNode, this method simply returns the casted object. trash() removes the document by calling removeFromParent(). turn() was originally planned for all documents, including low level documents. The idea was to have pairs of nodes, one front side and one back side, each simulating a sheet of paper. This became obsolete with bindings. For example, a sheet binding (discussed on page 124) would fulfill this requirement. On all discussed low level documents, turn() reports the error message that instances of this class cannot be turned. index() returns the index path of the document, calculated by WDIndex| ↩ ↪ |Comparator. Finally, compareTo(WDDocument) compares the index of the document with the index of a given one.

Other required implementations are: removeAllInputEventListener() removes all attached input event listener by iterating through the list of listeners and removing them one by one. createBorder() sets an association to a newly created instance of WDLowLevelDocBorder and adds it as child to the node. The border appears as a thin surrounding line along the node's bounds. removeBorder() removes the border by simply removing it from its parent node.

The method setShadowVisible(boolean) sets an existing shadow of a node to visible or invisible, depending on the passed parameter. Only low level documents can have a shadow. Bindings do not have shadows, but their objects at the lowest level may have. What a user may interpret as the "binding's shadow" is the sum of the shadows of all of its low level documents. The shadow update for most discussed low level documents[14] is performed by an equivalent of the following exemplary code snippet, taken from WDImage:

[WDImage.java | updateShadow()]

```
268     public WDShadow updateShadow() {
269         boolean  visibility ;
270         if  (getShadow() == NO_SHADOW) {
271             visibility   = VISIBLE;
272         } else {
273             visibility   = getShadow().getVisible();
274         }
275         setShadow(new WDShadow(this));
276         setShadowVisible( visibility ) ;
277
278         return getShadow();
279     }
```

Currently, this method is only called from within updateShadows() in class WildDocs. At this time, all shadows are already removed before the new one is set. Adding the shadow to the layer happens at the above mentioned method at line 1529, shown on page 113. This mechanism has been written over time and can be criticized for not providing an autonomously working updateShadow(). Removing the old shadow and adding the new one to the layer should be placed inside this method instead of being called within WildDocs.

[14] updateShadow() in WDText contains only line 275 and 276 of the WDImage code snippet on this page.

Image WDImage extends Piccolo's PImage, which is a wrapper around Java's class Image (package java.awt). In principle, all image formats that are supported by Image work also in WildDocs. However, we have implemented a simple file recognition that recognizes only the file extensions for PNG, JPEG, and GIF files (see also Sect 4.3.7). WDImage instances draw the image over the complete node, possibly distorting. As long as distortion over the complete node is active, setting the background during creation, as shown at line 84 at the code snippet on page 106, would not be necessary.

An additional argument passed to the constructor contains the path to the image file as URI. It is converted to an instance of URL at line 80 and passed to the superclass, which loads the image.

Plain Text Instances of WDText extend Piccolo's class PText. It supports multi-line plain text representation. There are three constructors. All of them expect a WildDocs instance as the first argument. One of them additionally expects a file path as URI that points to a text file. WDTextLoader loads a file and returns it as string, which is passed to the second constructor.

The second constructor (line 142–144) takes the WildDocs instance and the given textual content and passes them to a third constructor, adding the constant NO_LINEOFFSET as a third attribute. This constant, an integer, indicates that there is no line offset given.

The final constructor passes the given text to the superclass, which creates a node with the given string as content. Beside the method calls explained above for color, clip alignment, and input event handler, this constructor adds some additional settings:

[WDText.java | WDText(WildDocs,String,int)]

```
108         setGreekThreshold(DEFAULTGREEKTHRESHOLD);
109         setSuccessor(NO_TEXT);
110         setLineOffset(aLineOffset);
```

This code snippet shows that the greek threshold is set to a default value. This value sets a limit of font sizes below which a font is displayed "greek" looking. This prevents the system of rendering lots of small texts that possibly cannot be read, because of its size and the insufficient screen resolution. This is relevant mostly when zooming out. Further, the successor is set to NO_TEXT and the line offset to the given integer value. Both have to do with splitting up a text into several nodes automatically, as it is necessary for fixed size document when the text does not fit on one page. There will be an explanation later in this section.

[WDText.java | WDText(WildDocs,String,int)]

```
117         setConstrainWidthToTextWidth(false);
118         setConstrainHeightToTextHeight(false);
```

This code snippet enables dynamic breaking of long lines. Originally, the setting of this was dependent upon whether or not documents were of fixed or variable size, whereas WildDocs only had it on for fixed size documents. However, we decided to enable this option for all versions.

If the preference switch FIXEDSIZEDOCS is set to ON, the node's bounds are set to a given default fixed size, currently ISO A4:

[WDText.java | WDText(WildDocs,String,int)]

```
121         if (WildDocs.FIXEDSIZEDOCS == WildDocs.ON) {
124             setBounds(new WDUnitConverter(aWildDocs).wdToJava(DEFAULTSIZE));
```

We implemented behavior that adds new nodes if the loaded text does not fit onto one single instance of WDText. We copied paint(PPaintContext) from PText and modified it at three spots: Firstly, an if-clause at line 414 checks if fixed size documents are on. If yes, the changed code is used; otherwise, the given PPaintContext is passed to the superclass at line 457:[15]

[WDText.java | paint(PPaintContext)]

```
412     protected void paint(PPaintContext paintContext) {
413         super.paint(paintContext);
414         if (WildDocs.FIXEDSIZEDOCS) {
415             super.paint(paintContext);

456         } else {
457             super.paint(paintContext);
458         }
459     }
```

The other two modified parts are the initialization of the for loop at line 435 and the if-clause at line 442:

[WDText.java | paint(PPaintContext)]

```
434                 // The getLineOffset() is added for WildDocs instead of 0.
435                 for (int i = getLineOffset(); i < lines.length; i++) {
436                     TextLayout tl = lines[i];
437                     y += tl.getAscent();
438
439                     if (bottomY < y) {
440
441                         // This if-clause added for WildDocs.
442                         if (getSuccessor() == NO_TEXT) {
443                             createSuccessor(i);
444                         }
445
446                         return;
447                     }
```

Originally, the for loop initialization was set to zero, which caused the node to display the complete text, starting at its first line. Now, the changed code forces the node to display the text with an offset. The offset results in one single text possibly being spread over several nodes. Line 443 calls the creation of a new successor node with the current line offset as argument. After the new node is created, it is set as successor:

[WDText.java | createSuccessor(int)]

```
466     private void createSuccessor(int aLineOffset) {
467         WDText nextPage = new WDText(getWildDocs(), getText(), aLineOffset);
468         nextPage.setBounds(this.getBounds());
469         setSuccessor(nextPage);
470         getWildDocs().addNodeToDesk(nextPage);
471     }
```

[15] This code calls super.paint(PPaintContext) more often than necessary. It could be cleaned up by either removing line 415 and the else part at line 456–458, or by removing line 413.

Because the successor node also calls paint(PPaintContext), the text may be split several times. paint(PPaintContext) overrides the method of its superclass. It uses the private variable lines. Because it is private, we had to create this variable for WDText and copy all parts from PText that use this variable as well as any other private variable that is needed by these but cannot be accessed from WDText. These are setFont(Font), recomputeLayout(), and the constant EMPTY_TEXT_LAYOUT_ARRAY.

The code for splitting up and piping text among several documents is experimental and still shows several problems. For example, the creation of the successor node is not done by WDNodeFactory. Instead, it is directly coded in WDText. Also, the calculation is triggered when a node is resized.[16] However, the text does not flow through all connected nodes. Important code for piping text is missing in WildDocs's current version. Another possibility to add the code and possibly less confusing would be to place it inside recomputeLayout(), which is called from within paint(PPaintContext).[17]

Styled Text Styled text support in WildDocs is provided by WDStyledText. It extends Piccolo's class PStyledText and supports RTF and HTML text formats. The constructor part looks in many ways similar to the one in WDText. However, there are also some differences. One is that it takes beside an URI that points to the file, also the file type as an argument, represented as an integer:

[WDStyledText.java | WDStyledText(WildDocs,URI,int)]

```
158    public WDStyledText(WildDocs aWildDocs, URI aFileUri, int aType) {
159        this(aWildDocs, readDocument(aFileUri, aType));
160    }
```

The method readDocument(URI,int) loads the file as a string using WDTextLoader and passes it either to readRtf(String) or readHtml(String), depending on whether it is classified as RTF or HTML. Otherwise an error message appears.

A new JEditorPane is created with the appropriate MIME type and the given content as string. getDocument() returns the document, which is an instance of Document (package javax.swing.text):

[WDStyledText.java | readRtf(String)]

```
452    protected static Document readRtf(String aRtfDoc) {
453        return new JEditorPane("text/rtf", aRtfDoc).getDocument();
454    }
```

The code for readHtml(String) (line 464–487) is equivalent to readRtf(String). However, currently this works only for HTML files that have style information embedded. Support for external style files or default values needs to be added in the future. Neither RTF nor HTML support graphics in WildDocs. Despite the fact that it involves more development efforts, it is *not intended* that instances of WDStyledText include images. WildDocs's philosophy is to create combinations of different media types by using bindings.

Similar to WDText, the returned Document instance together with the WildDocs instance is then passed to another constructor, which passes them to a third constructor, adding NO_L| ↵ ↳ |INEOFFSET to indicate that there is currently no line offset. The final constructor differs

[16] There are preference switches for fixed size documents *and* resizable documents. This is why fixed size documents may be on (that is mandatory for enabling splitting of text nodes) while documents still can be resized.

[17] Adding it to recomputeLayout() was suggested by Jesse Grosjean in the mailing list posting at http://mailman.cs.umd.edu/pipermail/piccolo-chat/2005/000669.html (visited on 2006-03-16).

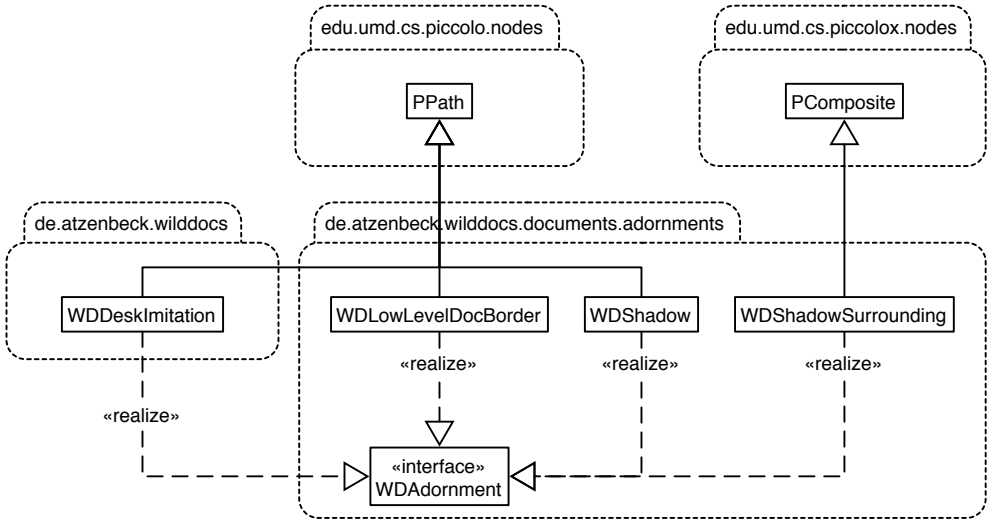


Figure 4.5.: Adornments class diagram (package documents.adornments)

from the one in WDText, in that the document content is provided as Document, not as plain text string. PStyledText does not have a constructor for passing documents. Therefore, WD| ⤸ ⤷ |StyledText sets the document explicitly. There is also no support for greek threshold as there is with PText. However, WDStyledText supports insets:

[WDStyledText.java | WDStyledText(WildDocs,Document,int)]

```
131         if (WildDocs.STYLEDTEXTINSETS) {
132             setInsets(WildDocs.INSETSVALUE);
133         }
```

If the preference switch for insets is on, default values for the inset are applied to the document. This prevents bounds being right next to the document's content.

The layout calculation that splits up a long text into several fixed sized nodes is taken from paint(PPaintContext) in PStyledText and modified with additional code, equivalent to WDText. However, because the required variable lines in PStyledText is protected, there was no need to copy additional code to WDStyledText beside that, as we had to do with WDText.

Adornments

Adornments are visual add-ons without major support for spatial structuring. They are part of the package de.atzenbeck.wilddocs.documents.adornments and implement the interface WDAdornment, which has currently no content, but is used only for classifying objects as adornments. Currently, node bounds and shadows are supported exclusively for low level documents. Figure 4.5 depicts relationships of adornment classes, explained in the following sections.

Border Line WildDocs's border adornment (WDLowLevelDocBorder) extends Piccolo's class PPath and implements WDAdornment. Its constructor associates a given WDLowLevelDoc in-

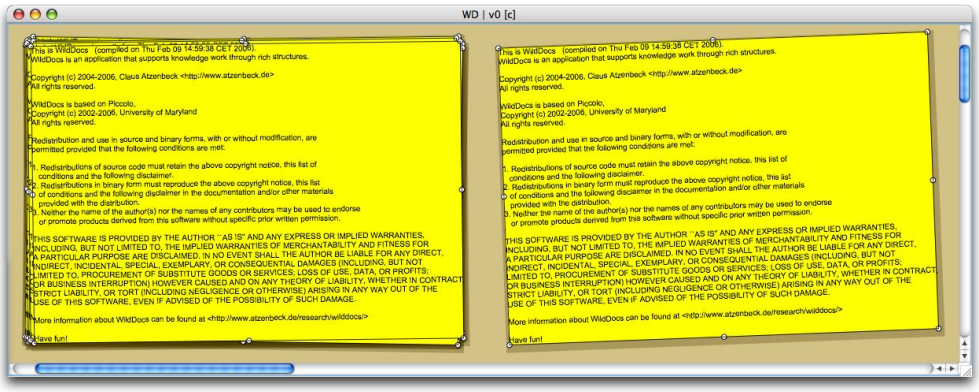


Figure 4.6.: Shadow and border line for low level documents; pile of ten documents (left) and one single document (right)

stance and calls createBorder(). It is assumed that the associated document is rectangular. The document's *x* and *y* position as well as the width and hight are taken to create a new rectangle with no paint, which is not pickable. Stroke type and color are taken from the preference settings in WildDocs.

Currently, the surrounding appears as a thin line along the document's bounds. Its main purpose is to make it easier for the user to recognize the node's border when being placed on top of another one that has the same background color. This helps to estimate the number of documents that are piled, in combination with sloppy alignment and shadows, as shown in Fig. 4.6.

Shadow A shadow is a transparent rectangle behind a node, assuming that the node itself is also rectangular. There are preference settings for color and level of transparency as well as oversize and offset relative to its associated document. The oversize constant represents how much the shadow is larger than its related document.

The shadow bounds are sharp, as Fig. 4.6 depicts. This is an easy and effective implementation for zooming; however, a fuzzy border would look more natural. Some graphics application, such as OmniGraffle, allow the user to adjust the shadow fuzziness from sharp to very smooth, beside changing offset, color, or transparency. Figure 2.26 on page 63 demonstrates shadows with different attributes.

There are two classes that support shadows for low level document: WDShadow and WD| ⤸ ⤷ |ShadowSurrounding. We will explain both in the following.

Shadow – Independent Shadow Our first implementation for shadow support was WD| ⤸ ⤷ |Shadow. It extends Piccolo's PPath and implements WDAdornment. It is neither added to its related node nor bound together with it as children of a composite node. A newly created shadow is put onto the space below all other existing nodes and exists there independent of its siblings. A method in WildDocs triggers an update of all shadows:

[WildDocs.java | updateShadows()]

```
1513    public void updateShadows() {
```

```
1514        WDTempNodeStorage shadows = new WDTempNodeStorage();
1515        getLayer().getAllNodes(new ShadowFilter(), shadows);
1516
1517        if (!shadows.isEmpty()) {
1518            getLayer().removeChildren(shadows);
1519        }
1520
1521        if (SHADOW) {
1522            WDTempNodeStorage lowLevelDocs = new WDTempNodeStorage();
1523            getLayer().getAllNodes(new LowLevelDocFilter(), lowLevelDocs);
1524
1525            if (!lowLevelDocs.isEmpty()) {
1526                Iterator iterator = lowLevelDocs.iterator();
1527                while (iterator.hasNext()) {
1528                    WDLowLevelDoc doc = (WDLowLevelDoc) iterator.next();
1529                    getLayer().addChild(0, doc.updateShadow());
1530                }
1531            }
1532        }
1533    }
```

Firstly, *all* existing shadows are removed from the space at line 1518. If shadows are activated, an iterator iterates through a list of all WDLowLevelDoc instances. They are triggered to update their shadows, and added to the current layer at the very bottom (line 1529). The update on the node's side is explained on page 107.

WDShadow's constructor receives either an instance of WDDocument, or instances of WildDocs and PNode, which are also extracted from a given WDDocument. Beside setting the associations, the shadow is painted and set to not pickable.

Drawing the shadow is performed by drawShadow(). It is based on the size and position of the associated node. Converted offset preferences and oversizes are applied to it. Also transparency, color, rotation, and scale are set. For placing the shadow correctly, the node's global coordinates are used:

[WDShadow.java | drawShadow()]

```
140        Point2D nodeGlobalCoord = node.localToGlobal(new Point2D.Double(node
141                .getX(), node.getY()));
142        Point2D shadowGlobalCoord = this.globalToLocal(nodeGlobalCoord);
143
144        float xPos = (float) shadowGlobalCoord.getX() - (oversize / 2);
145        float yPos = (float) shadowGlobalCoord.getY() - (oversize / 2);
146        float width = (float) node.getWidth() + oversize;
147        float height = (float) node.getHeight() + oversize;
148
149        setPathToRectangle(xPos, yPos, width, height);
150        translate(offsetX, offsetY);
```

This code snippet also takes care of distributing the oversize equally to the left and right as well as at the top and bottom (starting at line 144), before drawing the rectangle at line 149, and applying the offset at line 150.

Because the shadow is internally not connected to the associated document, it would not follow the document when it is dragged. Therefore, the shadow is made invisible as soon as

the mouse button is pressed on its node:

[WDNodeInputEventHandler.java | mousePressed(PInputEvent)]

```
166         if (aEvent.getPickedNode() instanceof WDLowLevelDoc) {
167             ((WDLowLevelDoc) aEvent.getPickedNode()).setShadowVisible(false);
168         }
```

On mouse release, all shadows are updated, as discussed on page 112:

[WDNodeInputEventHandler.java | mouseReleased(PInputEvent)]

```
307         getWildDocs().updateShadows();
```

Currently, any action that is not caused by releasing the mouse button on a WDLowLevelDoc does not automatically update the shadow; it stays where it was originally. Affected are, for example, resizing a document or straightening a stack using CTRL–S. This should be changed in a future version.

Shadow – Composite A more recent implementation is WDShadowSurrounding. It did not find its place anywhere else in the used code yet, because it is in an early development stage. It mainly differs from WDShadow by its superclass, behavior, and shadow creation.

WDShadowSurrounding extends Piccolo's class PComposite and extends WDAdornment. It takes the given document, creates shadow parts which surround it, and binds them together. Now, when the document is moved, the shadow also moves with it. This has two obvious changes in behavior or appearance: Firstly, the shadow does not have to be updated after moving the document, because it follows the movement. Secondly, the shadow is not painted at the very bottom and may appear on top of another document, depending on the node's index level.

The constructors are identical to those in WDShadow, but the shadow drawing differs: As mentioned above, the shadow *surrounds* the document, from whence the class retrieved its name. The method drawShadow() adds the the WDShadowSurrounding instance as child to the *document*:

[WDShadowSurrounding.java | drawShadow()]

```
119         node.addChild(this);
```

Then, the method takes the document's dimensions and oversizes and calculates the four shadow rectangles. Finally, they are added to the *composite node*. In the current version, offset calculation and oversize in millimeters do not work: Offset is switched off, and the oversize preference variable has to be given in pixels. The following paradigmatic code shows the calculation for the bottom shadow part:

[WDShadowSurrounding.java | drawShadow()]

```
149         float rect4xPos = (float) (nodeX − oversize);
150         float rect4yPos = (float) (nodeY + nodeHeight);
151         float rect4width = nodeWidth + (2 ∗ oversize);
152         float rect4height = oversize;

160         PPath rect4 = PPath.createRectangle(rect4xPos, rect4yPos, rect4width,
161                 rect4height);

166         setPathAttributes(rect4);

171         addChild(rect4);
```

After the shadow part's dimensions are calculated, a new PPath rectangle is created and the attributes set. Then, the rectangle is added to the composite. The attributes are set by setPathAttribute(PPath):

[WDShadowSurrounding.java | setPathAttributes(PPath)]

```
174  public void setPathAttributes(PPath aPath) {
175      aPath.setTransparency(WildDocs.SHADOWTRANSPARENCY);
176      aPath.setPaint(WildDocs.SHADOWCOLOR);
177      aPath.setStroke(null);
178      aPath.setPickable(false);
179  }
```

Because the document adds the composite node as a child, the latter automatically shows the same global rotation than the document. Therefore, the shadow parts that are added as children to the composite node fit around the document independent of its rotation.

There is no need to update this shadow type when moving a document, because document and shadow are bound. This includes moving the document via shortcuts, for example, by CTRL-S or rotating them. However, resizing the document does currently not include resizing the shadow around it. This needs to be implemented in a future version.

4.2.3. Bindings

Bindings are a spin-off of our core research. They were developed after we had the idea that fixed size documents in combination with bindings may be used more effectively. This follows the idea of having collection objects in spatial structure applications or binding tools, such as folders, in the real world. Binding support is implemented in its basics. However, important code is still missing, such as graphical representations.

Our implementation includes bindings (located in documents.bindings) and their binding mechanisms (located in documents.bindings.mechanisms). We made this distinction, because we aimed for easy implementation of new bindings and simple ways to attach appropriate mechanisms to them. We divided the following in two sections, discussing both, bindings and binding mechanisms. Figure 4.7 provides an overview of binding classes which are discussed in the following.

It has to be clearly stated that bindings are in an *early development state and not fully functional*, including mainly the graphical representation and everything that is closely related to it. Nevertheless, we report our code so far to show our theoretical results at least partly implemented. Even though not fully functional, most parts including central code pieces are already finished.

Supported Types and Behavior

Common Behavior The core class for most bindings is WDBinding which extends Piccolo's PNode and implements WDDocument. The only exception is WDPrimitiveBinding which extends PPath instead. This exception will be discussed on page 127. WDBinding is an abstract class and carries most of the behavior for bindings. Extending classes are mainly used for setting arguments or minor extensions. This allows system developers to add new bindings quickly and easily. The binding implementation is based on our real world analysis (see especially Sect. 2.2.3).

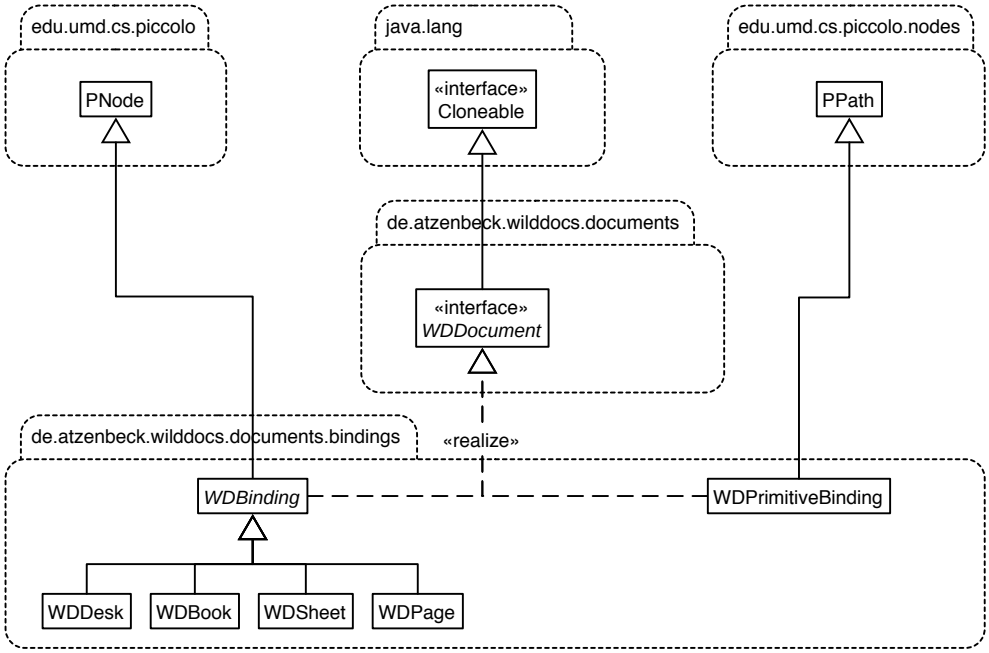


Figure 4.7.: Bindings class diagram (package documents.bindings)

For easy use at all bindings classes, WDBinding defines a public constant for every implemented binding class, for example:

[WDBinding.java | BOOK]

```
77    public static final Class BOOK = WDBook.class;
```

Most of the constructor's arguments are constants that are passed from the extending class:

[WDBinding.java | WDBinding(WildDocs,Class[],Class[],double,double,Rectangle,int]

```
168   public WDBinding(WildDocs aWildDocs, Class[] someContainableDocTypes,
169           Class[] someDissolvingBindings, double aMaxThickness,
170           double aStaticThickness, Rectangle aSize, int aClipAlignment) {
171
172       //   initialization
173       setWildDocs(aWildDocs);
174       setContainableDocTypes(someContainableDocTypes);
175       //setDependentOnBinding(aDependentOnBindingClass);
176       setDissolvingBindings(someDissolvingBindings);
177       setMaxThickness(aMaxThickness);
178       setStaticThickness(aStaticThickness);
179       setSize(aSize);
180       setClipAlignment(aClipAlignment);

188       setPickable(NOT_PICKABLE);
189   }
```

The initialization exclusively sets variables. Firstly, the WildDocs instance is associated,

then those classes of bindings that may be added, as well as bindings that dissolve when added. The maximum thickness that is set at line 177 is needed to check whether there is "space" left (metaphorical) for adding additional documents to the binding.

The static thickness that is set at the line below is the minimum thickness that the binding has in any case, independent of its contents. If the maximum thickness is larger than the static one, the binding may grow beyond its static thickness.

The setter for the binding's size expects a rectangle with values in millimeter. Therefore, the set method has to convert to the internal units using an instance of WDUnitConverter:

[WDBinding.java | setSize(Rectangle)]

```
289     protected void setSize(Rectangle aSize) {
290         WDUnitConverter converter = new WDUnitConverter(getWildDocs());
291         setBounds(converter.wdToJava(aSize));
292         size = aSize;
293     }
```

The clip alignment holds information about where the binding is added to *other* bindings and how its alignment is supposed to be. The clip alignment is *not* used when documents are added to *this* binding. Finally, the binding is set to be not pickable. This results in ignoring mouse events that occur on the binding.

Bindings support front and back covers. In principle, any low level document can be used as one of those. This code was implemented before we focused on an exclusive support for bindings as structure elements. For later WildDocs versions, covers would be instances of specialized bindings instead. All bindings with front and back cover associated can be opened. Some bindings do not have a front cover, for example, trays. A back cover exist for any binding; however, it may be invisible. This is the case for document piles that do not represent the space graphically on which they are placed.

Setting a front cover begins with removing the old one, if existing:

[WDBinding.java | setFrontCover(WDLowLevelDoc)]

```
301     protected void setFrontCover(WDLowLevelDoc aFrontCover) {
302         if (getFrontCover() != NO_COVER) {
303             removeChild(getFrontCover().toPNode());
304         }
305         frontCover = aFrontCover;
306         if (getFrontCover() != NO_COVER) {
307             new WDNodeDragger(getBindingMechanism())
308                     .dragToBindingMechanism(aFrontCover);
309             PNode cover = aFrontCover.toPNode();
310             addChild(cover);
311             cover.moveToFront();
312         }
313     }
```

Then, after the passed front cover is set, it is checked whether it has the value NO_COVER. This would mean that there was no cover passed. Otherwise, if there is a cover, a new instance of WDNodeDragger drags the front cover to the binding mechanism. Then, the cover is added as child to the binding (*not* to the binding mechanism) and moved to the front.

The method setBackCover(WDLowLevelDoc) is equivalent to setting the front cover, but uses moveToBack() instead of moveToFront() after adding the cover to the binding. Both

methods are provided by PNode. Currently, there is a problem with moving covers to the back. This works only correctly if bindings are not turned.

Thickness is important for certain binding constraints. Some code exists already, but some parts need to be added to have them fully functional and working correctly. Currently, WDBinding has four thickness related methods partly implemented. The most important one is getThickness(double), which returns the binding's thickness, considering its static height:

[WDBinding.java | getThickness(double)]

```
555  public double getThickness(double staticHeight) {
556      // TODO add code
557      if (staticHeight > getThickness()) {
558          return staticHeight;
559      } else {
560          return getThickness();
561      }
562  }
```

The thickness without considering the static height is calculated by getThickness():

[WDBinding.java | getThickness()]

```
539  public double getThickness() {
540      // TODO add code
541      double thickness = 0;
542      thickness = thicknessOfChildren() + getEmptyThickness();
543      return thickness;
544  }
```

It returns simply the sum of the thickness of all children plus the thickness of the binding itself, including front and back cover. The method getEmptyThickness() is supposed to ignore the static height, but lacks currently of content.

The thickness of a binder's contents is calculated recursively:

[WDBinding.java | thicknessOfChildren()]

```
522  public double thicknessOfChildren() {
523      // TODO add code
524      double thicknessOfChildren = 0;
525
526      Iterator iterator = getChildrenIterator();
527      while (iterator.hasNext()) {
528          WDDocument doc = (WDDocument) iterator.next();
529          thicknessOfChildren += doc.getThickness();
530      }
531      return thicknessOfChildren;
532  }
```

Binding mechanisms are involved in binding documents to a binding. They will be discussed starting on page 129. The following code snippet is called when a binding mechanism is associated to the binding:

[WDBinding.java | setBindingMechanism(WDBindingMechanism)]

```
343  protected void setBindingMechanism(WDBindingMechanism aBindingMechanism,
344          int aMechanismPosition) {
345      if (getBindingMechanism() != NO_BINDINGMECHANISM) {
346          removeChild(getBindingMechanism());
```

```
347         }
348         bindingMechanism = aBindingMechanism;
349         /*
350          * The binding mechanism becomes child of the binding. The graphical
351          * representation of the mechanism is child of the mechanism.
352          */
353         if (getBindingMechanism() != NO_BINDINGMECHANISM) {
354             WDBindingMechanism mechanism = getBindingMechanism();
355             addChild(mechanism);
356
357             /*
358              * Place the mechanism on the right place and also adjust the
359              * position dependent rotation.
360              */
361             Point2D posOnBinding = new WDBindingClipCalculator().position(this,
362                     aMechanismPosition);
363             double rotation = new WDBindingClipCalculator()
364                     .rotation(aMechanismPosition);
365             mechanism.rotate(rotation);
366             mechanism.centerFullBoundsOnPoint(posOnBinding.getX(), posOnBinding
367                     .getY());
368         }
369     }
```

Firstly, if a binding mechanism has already been associated, it will be removed at line 346. The passed binding mechanism is then set. If it is not NO_BINDINGMECHANISM, it is also added as child to the binding at line 355. Because the graphical representation of the binding mechanism is attached to the binding mechanism, but not directly to the binding, it will now appear on the screen. An instance of WDBindingClipCalculator calculates the correct position and angle, and the mechanism's location and rotation is adjusted to the correct values.

Physical bindings allow to leaf through their content. For example, pages of a book need to be turned. This is relevant to the binding mechanism implementation, because this is where documents have their fixed position. Originally, we planned to support a binding dimension for each, source and destination. This would allow to move a document from one position to another spot when turned. For example, a binder's binding mechanism moves a turned page several centimeters away. The binding dimension changes and the turned document is suddenly "placed somewhere else", as indicated in Fig. 4.8. This may include even different binding dimension locations, for example, depending on the height of a binder's filling, as indicated in the previously mentioned figure. This does not affect all bindings, for example, documents bound in books do not change their binding dimension position when being turned. However, binding lines may differ among bound documents, for example, when a book's spine bows on turning pages, as implemented by (Chu et al., 2004, 80). An alternative to source and destination binding dimension would be an offset for turned pages. However, this would not allow to change the dimension type.[18]

The method isTurned() is supposed to return true if a binding is turned and false if not. Because every binding has a back cover, the complete binding can be considered as being

[18] Changing the binding dimension when turning a document seems not to be useful from our current point of view. At this time, we are not aware of an useful use case that shows that this would support the user.

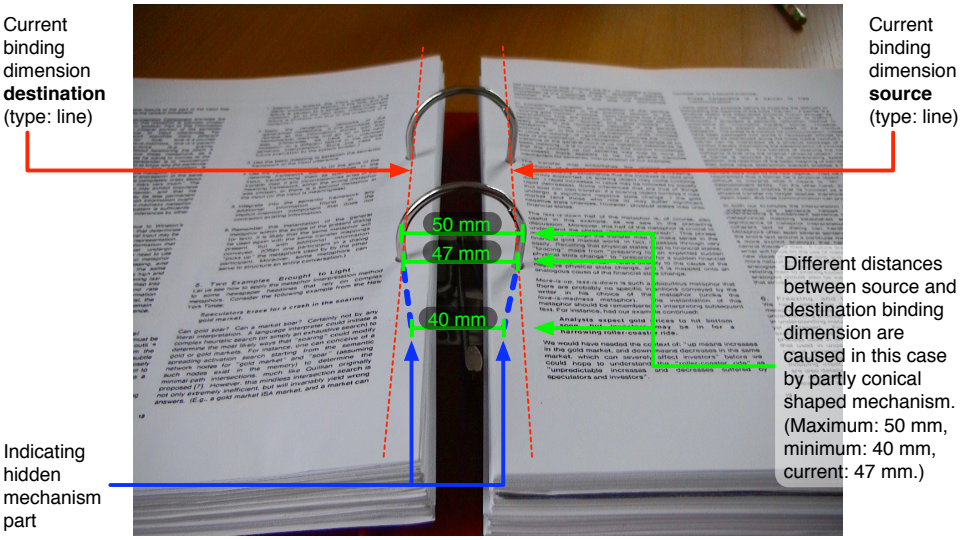


Figure 4.8.: Binder with depicted source and destination binding dimension

turned if and only if the back cover is turned.[19]

[WDBinding.java | isTurned()]

```
492     public boolean isTurned() {
493         if  (getBackCover() != NO_COVER) {
494             //  FIXME refactor!!
495             //return  getBackCover().isTurned();
496             return false;
497         } else {

504             return NOT_TURNED;
505         }
506     }
```

The original code can be seen at line 495. Changes on related parts within the code base caused us to temporarily remove this line for compiling until refactoring the depending parts.

The way of turning a complete binding has changed during development. Figure 4.9 depicts the reason. In an early development phase, contained documents were directly associated to the binding. The binding mechanism was only responsible for the graphical representation and involved in calculating the correct clip alignment. During time we put more responsibility in binding mechanisms, following our observations from the real world. This decision was supported by the fact that most behavior that is relevant for adding documents belong strongly to the mechanism and not to binding instance directly, for example, opening or closing the mechanism. Figure 4.9 draws lines between equivalent WildDocs associated binding parts of the latter development and a depicted real world binder.

The code for turning a binding (line 589–604) is not adapted yet to the new architecture. It still follows the earlier versions when contained documents were directly associated to the

[19] This said, we point out that this would not be true, if an open binding could be placed turned, with front and back cover on top. This is not supported by WildDocs.

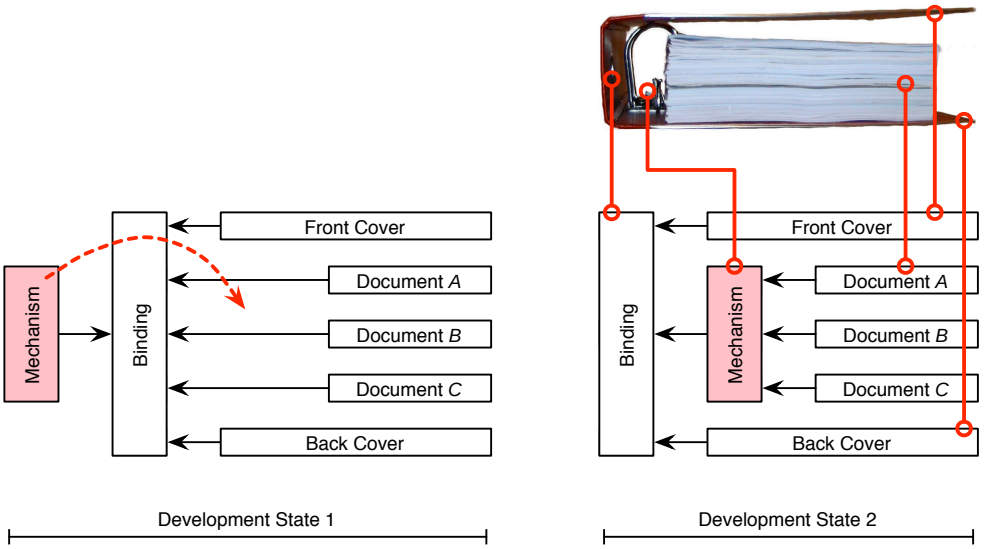


Figure 4.9.: Binding associations in early and late development, and depicted real world equivalent

binding. It creates an iterator[20] that iterates through all children of the binding and turns them recursively:

[WDBinding.java | turn()]

```
589  public void turn() {

595      Iterator iterator = new HashSet(getChildrenReference()).iterator();
596      while ( iterator .hasNext()) {
597          Object child = iterator .next() ;
598          if (child instanceof WDDocument) {
599              WDDocument docChild = (WDDocument) child;
600              docChild.turn() ;
601              docChild.toPNode().moveToFront();
602          }
603      }
604  }
```

The if-clause at line 598 prevents any other node than documents to be turned, for example, binding mechanisms. After turned, the document is put to the very front.

Browsing a binding page by page is an essential behavior for WildDocs users. WDBinding has already two methods for this, pageForward(WDDocument) (line 571–573) and pageBackward(WDDocument) (line 580–582). However none of them has content yet.

Bindings have two methods related to removing items. One is to completely remove the binding including its content; the other is to destroy the binding, but not its content. To remove everything, the method trash() calls removeFromParent(), which is provided by the

[20] The iterator iterates through a newly created HashSet that contains the binding's children. The hash set is used to avoid ConcurrentModificationException errors during runtime.

superclass PNode. Removing the binding without its content calls the relevant method at the associated WDBindingMechanism instance:

[WDBinding.java | emptyAndTrash()]

```
611    public void emptyAndTrash() {
612        getBindingMechanism().emptyAndTrashBinding();
613    }
```

For many bindings, this action is not required, because the binding mechanism can be opened and the content removed before calling trash() on the empty binding. However, some binding types, such as BOOK, do not allow opening the binding mechanism once it is bound. This represents a "final" state and must be destroyed if the user wishes to use its content differently. emptyAndTrash() performs exactly this step: The binding mechanism is forced to open and therefore semantically destroyed, whereupon the binding including its mechanism is immediately trashed. Only its content remains and can be bound again. We discuss this method in more detail on page 133.

Like in all other extensions of WDDocument, there are methods for returning the binding as casted PNode (toPNode()), its index path (index()), or to compare its index path to another document (compareTo(WDDocument).

In the following, we discuss current implementations of bindings. They extend WD| ↩ ↳ |Binding. Because of the rich implementation of the abstract WDBinding class, the code bases for concrete WildDocs bindings are rather small. Only those attributes need to be changed that differ from those in WDBinding. This demonstrates the possibility of building new binding classes with ease.

Desk Instances of the class WDDesk represent desk metaphors. A desk can hold documents and therefore is considered in WildDocs as a binding. This differs from the desk imitation as discussed in Sect. 4.1.4. Like for all other binding classes, the top part of the code contains assignments for four constants. The first one defines that only books and sheets can be placed on a desk:

[WDDesk.java | CONTAINABLE_DOCTYPES]

```
50    private static final Class[] CONTAINABLE_DOCTYPES = { BOOK, SHEET };
```

All other existing binding types cannot be added to a desk (currently pages and desks). The role of pages will be discussed later. The case that desks do not occur in CONTAINABLE_| ↩ ↳ |DOCTYPES prevents that a user piles desks, which would be realistic only when moving furniture and a metaphor break otherwise.

DISSOLVING_BINDINGS is used to declare types of bindings that dissolve when added (see discussion on page 46); however, this is empty for desks: We are not aware of any binding that would dissolve when added to a desk. CLIPALIGNMENT and MECHANISM| ↩ ↳ |POSITION are set to CENTER. The clip alignment represents where the desk would clip onto a binding mechanism. Because currently a desk cannot be part of a binding, this value is without meaning and only added because it is requested by the superclass constructor. The mechanism position carries information about where the clip alignment of a document that is added to the desk will be put to. In this case, it is the center of the desk.

The constructor takes beside the WildDocs instance also the desk's color and size:

[WDDesk.java | WDDesk(WildDocs,Paint,Rectangle]

```
66    public WDDesk(WildDocs aWildDocs, Paint aColor,
```

```
67              Rectangle aSize) {
68
69          super(aWildDocs, CONTAINABLE_DOCTYPES, DISSOLVING_BINDINGS, |
                ↪ |NO_THICKNESS,
70                NO_STATICTHICKNESS, aSize, CLIPALIGNMENT);
71
72          // set binding mechanism
73          setBindingMechanism(new WDDeskMechanism(this, aSize, aColor), |
                ↪ |MECHANISMPOSITION);
74      }
```

The WildDocs instance, all potential contained and dissolved dissolution binding types, the given size, and the clip alignment, as well as the information that the desk instance neither has a thickness nor a static thickness, are passed to the superclass WDBinding. The thickness of a desk is irrelevant, because at the current version, a desk cannot be contained by any other binding. This information is passed only because it is required by the superclass's constructor.

A new instance of WDDeskMechanism is created by passing the desk, its size, and the given color. The binding mechanism is set to the center of the desk, as defined by the constant earlier.

Turning a desk is realistically not used as an action in knowledge work. Therefore the method turn() in WDDesk overrides the one in its superclass and only prints a message that informs that a desk cannot be turned.

Book Instances of the class WDBook accept only sheets (WDSheet) as contents:

[WDBook.java | CONTAINABLE_DOCTYPES]

```
55      private static final Class[] CONTAINABLE_DOCTYPES = { SHEET };
```

Sheets do not dissolve when bound by a book. Therefore, the constant DISSOLVING_B| ↪ |INDINGS remains empty. The clip alignment is set to CENTER, the mechanism position is on the left side of the book. Currently, a book can only be added to the desk. Therefore, the clip alignment is of less importance. The code may be changed in a way that a book below a certain thickness level may be also added to a binder. In this case, the clip alignment would be at the book's left side.

WDBook sets the thickness of its front (FRONTCOVERTHICKNESS) and back cover (BA| ↪ |CKCOVERTHICKNESS) to 3 mm each. The default maximum thickness (STANDARD| ↪ |MAXTHICKNESS) is set to 100, assuming that books would be not thicker than 100 mm. However, maximum thickness is currently not used.

The constructor requests a WildDocs instance, a value for its maximum thickness, as well as color, and size:

[WDBook.java | WDBook(WildDocs,double,Paint,Rectangle)]

```
89      public WDBook(WildDocs aWildDocs, double aMaxThickness, Paint aColor,
90              Rectangle aSize) {
91
92          super(aWildDocs, CONTAINABLE_DOCTYPES, DISSOLVING_BINDINGS,
93                  aMaxThickness, NO_STATICTHICKNESS, aSize, CLIPALIGNMENT);
94
95          // set binding mechanism
96          setBindingMechanism(new WDBookMechanism(this), MECHANISMPOSITION);
97
```

```
98          // set cover
99          setFrontCover(new WDBindingCover(getWildDocs(), aSize, aColor,
100                 FRONTCOVERTHICKNESS));
101         setBackCover(new WDBindingCover(getWildDocs(), aSize, aColor,
102                 BACKCOVERTHICKNESS));
103     }
```

Most of the given information is passed to the superclass at line 92 and 93. The constant NO_STATICTHICKNESS is given to indicate that a book does not have a static thickness. Its thickness equals the sum of its parts.

After associating a new WDBookMechanism at line 96, a new front and a back cover is created, passing the WildDocs instance as well as the size, color, and the appropriate cover thickness. The use of WDBindingCover is still following an early approach in development, before we raised bindings to main structure elements, as discussed on page 106. Here, a binding cover is a low level document and should be replaced in future versions with an appropriate binding, such as WDSheet.

Sheet Instances of WDSheet represent the equivalent of what we know as sheets of paper in the real world. A sheet has two sides, a front and a back page. This is represented in WildDocs as contents of a sheet. Therefore, a sheet itself must be a binding. The only instances that a sheet can take are pages:

[WDSheet.java | CONTAINABLE_DOCTYPES]

```
54  private static final Class[] CONTAINABLE_DOCTYPES = { PAGE };
```

Pages stay pages when added. There is no dissolving behavior. The constant DISS| ↵ ↳ |OLVING_BINDINGS is empty. Clip alignment as well as mechanism position are set to CENTER. It cannot be predicted where a sheet is added, therefore the given clip alignment is only one of several possibilities. CENTER would be appropriate for adding a sheet on the desk, whereas it would have to be changed to LEFT when adding a sheet to a binder or book that have their mechanism position on the right hand side.[21] The mechanism position set to center works fine, since pages are of equal size and attached right behind of each other. The standard thickness of a sheet is currently set to $\frac{51.5}{500}$ mm ($= 0.103$ mm), assuming that 500 sheets create a pile of 51.5 mm height:

[WDSheet.java | STANDARDTHICKNESS]

```
80  public static final double STANDARDTHICKNESS = 51.5 / 500;
```

The constructor takes the same attributes as a book's constructor:[22]

[WDSheet.java | WDSheet(WildDocs,double,Paint,Rectangle)]

```
90  public WDSheet(WildDocs aWildDocs, double aThickness, Paint aColor,
91          Rectangle aSize) {
92
93      super(aWildDocs, CONTAINABLE_DOCTYPES, DISSOLVING_BINDINGS, aThickness,
94              NO_STATICTHICKNESS, aSize, CLIPALIGNMENT);
95
96      // set binding mechanism
```

[21] This applies only if the writing direction is from left to right. For example, for the opposite writing direction (e. g., Arabic or Hebrew), the clip alignment would be set to RIGHT.

[22] The variable aThickness should be named aMaxThickness in order to avoid misunderstandings.

```
97          setBindingMechanism(new WDSheetMechanism(this), MECHANISMPOSITION);
98
99          getBindingMechanism().addDocument(new WDPage(getWildDocs(), getMaxThickness() / 2,| ↩
                ↪ | aColor, aSize));
100
101         getBindingMechanism().addDocument(new WDPage(getWildDocs(), getMaxThickness() / 2,| ↩
                ↪ | aColor, aSize));
102     }
```

Most attributes are passed to the superclass. Also, a sheet does not have a thickness by itself. Its thickness is derived from the sum of its containing pages.

A new WDSheetMechanism is created and set (line 97), and the front and back page are associated at line 99 and 101. A sheet binding cannot be opened from the outside. Exactly two pages are added. This prevents of having more or less than two pages, which would be a violation of the sheet metaphor. For the construction of WDPage, we pass, among other attributes, the maximum thickness divided by two as well. This is received by WDPage as the thickness of the page. The reason for this is to simplify the setting of thickness at that point and make it more understand to humans. In the real world, most humans dealing with paper have an understanding of a sheet's thickness. However, there is no notion of how thick the front or back side of a sheet is. An alternative, possibly easier to understand, would be that WDSheet sets the given thickness and passes zero to its pages instead.

Page Pages, represented by instances of WDPage, are exclusively to be added to sheets. One page represents either a front or a back side of a sheet. They can hold instances of WDLowLevelDoc:

[WDPage.java | CONTAINABLE_DOCTYPES]

```
54      private static final Class[] CONTAINABLE_DOCTYPES = { LOWLEVELDOC };
```

In future versions, it may be possible that a page receives similar features that we know from word processing or desktop publishing applications. Different media types may be added and moved around. However, low level documents are not necessarily the only document types that may be added to pages in the future.

We can think of a binding class, that can be added to a page, for example, to be used as a "floating binding", containing a picture or table and its caption text. Other examples would be bindings that divide one page into sub-parts. Figure 4.10 depicts an example of five low level documents, three of them contained by a sub-binding ("Text body"). An important feature for such use cases would be dynamic piping of text from one to an other column, that is, from one low level document to the other in this case. Alternatively, this could be implemented using binding types that contain the complete textual low level document and take care of the split up representations. Such a binding type would be conceptually dividing the inner structure and the graphical representation.

From a user's perspective, binding low level documents on a *page* goes toward "grouping" features of chart applications; however, WildDocs is designed to offer special behavior underneath.

WDPage has constants for visible (VISIBLE) and invisible (INVISIBLE), which are used for turning documents. Clip alignment and binding mechanism position are set to CENTER. Because there are currently no bindings that can be added, the list for dissolving bindings is empty.

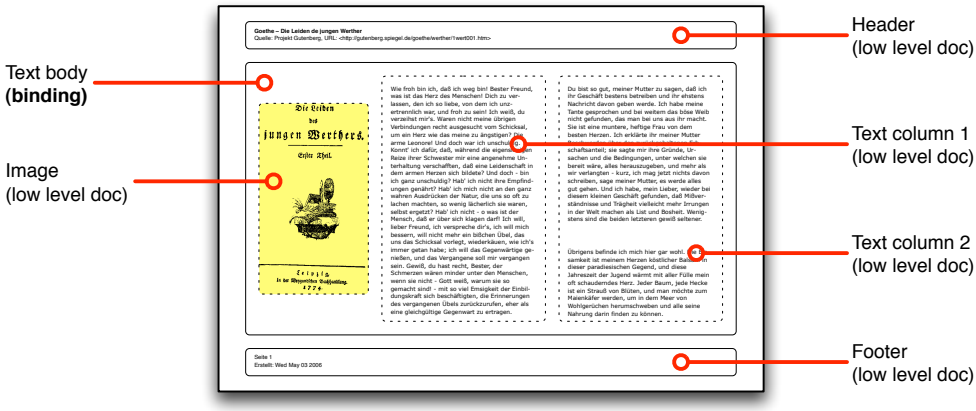


Figure 4.10.: Example of a page with a mix of binding and low level documents

The constructor is similar to the first part of the WDSheet constructor:

[WDPage.java | WDPage(WildDocs,double,Paint,Rectangle)]

```
70  public WDPage(WildDocs aWildDocs, double aThickness, Paint aColor,
71          Rectangle aSize) {
72
73      super(aWildDocs, CONTAINABLE_DOCTYPES, DISSOLVING_BINDINGS, aThickness,
74              NO_STATICTHICKNESS, aSize, CLIPALIGNMENT);
75
76      // set binding mechanism
77      setBindingMechanism(new WDPageMechanism(this, aSize, aColor), ↵
            ↳ |MECHANISMPOSITION);
78  }
```

Firstly, known attributes are passed to the superclass WDBinding. The current code shows an error at line 74, where equivalent to WDSheet the constant NO_STATICTHICKNESS is passed. However, the static value has to be the same as the maximum thickness, passed from WDSheet by the variable aThickness. The reason is that currently a page is the last possible binding. Low level documents that may be placed on pages do have zero thickness. Even bindings that may be added in the future, as mentioned above, will probably not have a thickness. Because WDSheet passes the responsibility of carrying the information about the thickness to both pages, they need to set it as static thickness. Alternatively, as mentioned on the previous page, WDSheet could set the static thickness instead; in this case, the code at line 74 would be correct.

Finally, a binding mechanism of the type WDPageMechanism is created at line 77, passing the binding, its size and color.

Turning a sheet results in showing the opposite page and hiding the previous one. There is a special case for pages, therefore, we override WDBinding's turn() method:

[WDPage.java | turn()]

```
86  public void turn() {
87      if (getVisible() == VISIBLE) {
88          setVisible(INVISIBLE);
```

```
89          } else {
90              setVisible(VISIBLE);
91          }
92      }
```

This sets a visible page invisible and vice versa. The turn() of WDBinding will turn *all* subbindings recursively, except for WDPage instances: When a sheet is triggered to turn, it will send a turn request to both pages that are associated. The one visible will become invisible and the opposite side is displayed instead.

Out of Turn: Primitive Bindings Primitive bindings are supported by WDPrimitiveBinding, which implements WDDocument. However, it does not extend WDBinding, but Piccolo's class PPath instead. In contrast to the more complex bindings of the WDBinding group, primitive bindings are – as the name suggests – primitive in appearance and behavior. They were developed recently to provide a simple binding object that allows putting documents on it and lets them move simultaneously while moving the binding. The binding is represented as a rectangle. This is similar to a blotting pad on which paper is placed and that can be moved while all objects follow its movement.

If the preference switch for primitive bindings is on, there is a menu entry "Primitive Binding" at the menu "Bindings" that lets the user create an instance that is put onto the WildDocs space. Alternatively, `CTRL-B` can be used (see Fig. 4.17 on page 154).

There are four constructors starting with WDPrimitiveBinding(WildDocs) as the least complex one. Unknown attributes, such as width, height, color, or *x* or *y* positions are replaced by predefined values, and passed to the next constructor. The following code snippet shows the final constructor:

[WDPrimitiveBinding.java | WDPrimitiveBinding(WildDocs,float,float,float,float,Paint)]

```
109     public WDPrimitiveBinding(WildDocs aWildDocs, float aXpos, float aYpos,
110             float aWidth, float aHeight, Paint aColor) {
111         super();
112         setWildDocs(aWildDocs);
113 
114         WDUnitConverter conv = new WDUnitConverter(aWildDocs);
115         float width = (float) conv.wdToJava(aWidth);
116         float height = (float) conv.wdToJava(aHeight);
117         setPathToRectangle(aXpos, aYpos, width, height);
118         setPaint(aColor);
119 
120         addInputEventListener(new WDNodeInputEventHandler(aWildDocs));
121     }
```

The first part of the constructor includes setting the passed WildDocs and creating a new WDUnitConverter instance. Then, width and height are converted from millimeters to pixels. A rectangle is created by the given *x* and *y* position, width, and height. The color is set and a new input event listener is instantiated. Primitive bindings use the same event handler than low level documents (WDNodeInputEventHandler).

Primitive bindings are not associated to binding mechanisms. Therefore, the method set|↩ ↳|ParentBindingMechanism(WDBindingMechanism) does not have any content and get|↩ ↳|ParentBindingMechanism() returns null. The method getClipAlignment() currently returns

the given default alignment and getThickness() returns zero. Neither clip alignment nor thickness are currently used. Turning a primitive binding via turn() is not possible and returns an error message. Calling trash() simply removes the primitive binding from its parent node. Bound documents are not put away and therefore are deleted as well. Other methods required by the interface WDDocument are toPNode(), index(), or compareTo(WDDocument). Those are identical to most existing implemented document classes in WildDocs.

There are methods for documents or generic nodes. A given document is casted to PNode and passed to the appropriate method:

[WDPrimitiveBinding.java | addDocument(WDDocument)]

```
177  public void addDocument(WDDocument aDoc) {
178      addNode(aDoc.toPNode());
179  }
```

The method addNode(PNode) simply adds the given node as child to the primitive binding:

[WDPrimitiveBinding.java | addNode(PNode)]

```
186  public void addNode(PNode aNode) {
187      addChild(aNode);
188  }
```

Adding a node that is dragged onto a primitive binding is planned to be triggered on mouse release:

[WDNodeInputEventHandler.java | mouseRelease(PInputEvent)]

```
283              if (WildDocs.PRIMITIVEBINDINGS) {
284                  WDTempNodeStorage primitiveBindings = new WDTempNodeStorage(
285                          getWildDocs().getLayer().getAllNodes());
286                  primitiveBindings.remove(activeNode);
287                  primitiveBindings
288                          .keepFiltered(new PrimitiveBindingFilter() );
289                  primitiveBindings.keepFiltered(new SmallerNodeIndexFilter(
290                          activeNode));
291                  primitiveBindings.keepFiltered(new IntersectionFilter (
292                          activeNode));
293
294                  if (primitiveBindings.isEmpty()
295                          && activeNode.getParent() instanceof WDPrimitiveBinding) {
296                      getWildDocs().getLayer().addChild(activeNode);
297                  } else {
298                      // primitiveBindings.getHighestIndexNode().addChild(node);
299                      ((WDPrimitiveBinding) primitiveBindings
300                              .getHighestIndexNode()).addNode(activeNode);
301                  }
302              }
```

The idea is to create a new temporary node storage object that contains all nodes, remove the active node, and keep only those primitive bindings that are below and intersect with the active node. The if-clause at line 294 checks if there are primitive bindings left. If not and the dragged node is bound by a primitive binding, it will be released from the primitive binding by adding it to the layer. Otherwise, it would be added to the primitive binding. If two or more primitive bindings are below the cursor at mouse release, the binding with the highest index

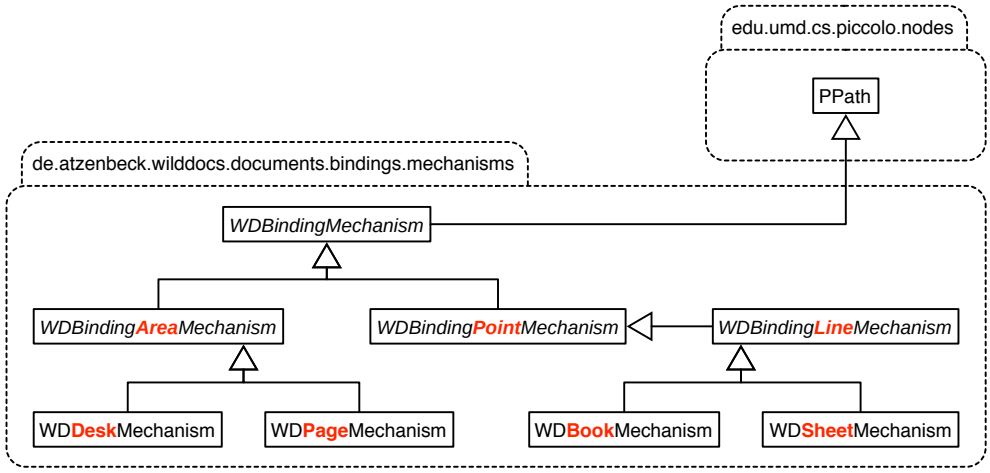


Figure 4.11.: Binding mechanisms class diagram (package documents.bindings.mechanisms)

level (among those that are left in the temporary storage object) is the one that is directly below the active node, and the one that will take the node.

This implementation works. However, problems with suddenly misplaced nodes occur after moving them onto a primitive binding and releasing the mouse button. This is probably caused by problems with moving the active node from one parent's coordinate system to another parent's. This incorrect behavior as well as the fact that we did not use bindings for our usability test were the reasons that we left out binding relevant code at the compiled version, including complex and primitive binding behavior that would be triggered on mouse release.

Binding Mechanisms

Binding mechanism related classes can be found in package documents.bindings.mechanisms. As Fig. 4.9 on page 121 depicts, we have changed binding mechanisms from a binding add-on to the class that is responsible for holding documents. This affects the set of methods for mechanisms essentially. In an earlier state, an instance of WDBindingMechanism was also the graphical representation. We changed this to be more flexible. In the current implementation, we *associate* the graphical representation, which can be any PNode instance. However, the inheritance still shows the former implementation: the abstract class WDBindingMechanism extends Piccolo's PPath, a class that wraps Java's GeneralPath (package java.awt.geom). It can be used to draw lines.

There are several classes that extend WDBindingMechanism. Figure 4.11 depicts the inheritances. We will discuss them in the following.

WDBindingMechanism introduces some constants. Some of them define a default state or value. For example, by default, the binding mechanism is open at creation time through DE| ⤾ ⤿ |FAULTOPENSTATE set to OPEN. The default binding clip alignment is set to CENTER. Two constants are related to rotation: NO_ROTATION is set to zero and FULLROTATION is set to 360 degrees.

The constructor takes and sets a binding instance, information whether the mechanism can be opened, the clip alignment to note where documents are added to the mechanism, as well as the maximum rotation and offset that may be applied to a document when added.

[WDBindingMechanism.java | WDBindingMechanism(WDBinding,boolean,int,double,double)]

```
110     public WDBindingMechanism(WDBinding aBinding, boolean ifOpenable,
111             int  aClipAlignment, double aMaxRotation, double aMaxOffset) {
112         setBinding(aBinding);
113         setOpenable(ifOpenable);
114         setClipAlignment(aClipAlignment);
115         setMaxRotation(aMaxRotation);
116         setMaxOffset(aMaxOffset);

125         forceOpen();

127     }
```

The method at line 125 forces the binding to open. The method open() is not suitable for bindings that are set to be non-openable, because it will not open them. However, in the beginning, they need to open a single time in order to fill them with documents. The method forceOpen() forces *any* binding to open:

[WDBindingMechanism.java | forceOpen()]

```
272     protected void forceOpen() {
273         boolean lastOpenableState = isOpenable();
274         setOpenable(OPENABLE);
275         open();
276         setOpenable(lastOpenableState);
277     }
```

It sets information about whether a mechanism may be opened temporarily to OPENABLE, opens the mechanism, and resets its status to the original value. The binding is now open; however, as soon as it is closed, it cannot be opened again using open().

The method open() calls open(PNode) with NO_GRAPHICS as parameter:

[WDBindingMechanism.java | open(PNode)]

```
291     public void open(PNode aGraphics) {
292         if  (isOpenable() == OPENABLE) {
293             if  (isOpen() != OPEN) {
294                 setOpen(OPEN);
295                 if  (aGraphics == NO_GRAPHICS) {
296                     updateGraphicalRepresentation();
297                 } else {
298                     updateGraphicalRepresentation(aGraphics);
299                 }
300             } else {
301                 System.out.println("This␣binding␣mechanism␣is␣already␣open.");
302             }
303         } else {
304             System.err.println("This␣binding␣mechanism␣cannot␣be␣opened.");
305         }
306     }
```

Firstly, it checks if the binding mechanism is supposed to be opened. If not, an error message will be sent. If it is openable and the mechanism is already opened, WildDocs will

inform the user about this state. Otherwise, its state is set to OPEN and the mechanism's graphical representation is updated. If NO_GRAPHICS was passed, the existing graphical representation is updated by sending it as attribute to the update method:

[WDBindingMechanism.java | updateGraphicalRepresentation()]

```
341     public void updateGraphicalRepresentation() {

346         updateGraphicalRepresentation(getGraphicalRepresentation());
347     }
```

This calls the same method than if a graphical representation would have been passed to the open method in first place:

[WDBindingMechanism.java | updateGraphicalRepresentation(PNode)]

```
355     public void updateGraphicalRepresentation(PNode aGraphics) {
356         // set the graphical stuff
357         if (getGraphicalRepresentation() != NO_GRAPHICS) {
358             removeChild(getGraphicalRepresentation());
359         }
360         setGraphicalRepresentation(aGraphics);

366         if (getGraphicalRepresentation() != NO_GRAPHICS) {
367             PNode graphics = getGraphicalRepresentation();

370             addChild(graphics);
371         }
372     }
```

Line 358 removes an existing graphical representation. Then, the passed attribute is set, which can be either a PNode instance or NO_GRAPHICS. If there is a PNode instance passed, it will be added as child to the binding mechanism (line 370) in order to have the graphical representation visible on the screen.

Closing a binding mechanism is equivalent to opening. The method close() calls close(| ↩ ↪ |PNode) with NO_GRAPHICS as attribute:

[WDBindingMechanism.java | close(PNode)]

```
320     public void close(PNode aGraphics) {
321         if (isOpen() != CLOSED) {
322             setOpen(CLOSED);

327             setOpenable(NOT_OPENABLE);
328             if (aGraphics == NO_GRAPHICS) {
329                 updateGraphicalRepresentation();
330             } else {
331                 updateGraphicalRepresentation(aGraphics);
332             }
333         } else {
334             System.out.println("This␣binding␣mechanism␣is␣already␣closed.");
335         }
336     }
```

If the binding mechanism is already closed, a warning message will be displayed. Otherwise, its open state is set to CLOSE and the graphical representation becomes updated, as discussed on this page. Setting the openable flag generally to NOT_OPENABLE at line 327 is an error in the used code base. For example, binders that should open multiple times would

not allow opening after they were closed the first time. Therefore, this line needs to be deleted in a future version.

WDBindingMechanism supports adding a single document as well as a range of documents. A given single document is wrapped into a list and sent to addDocuments(ArrayList):

[WDBindingMechanism.java | addDocument(WDDocument)]

```
382  public void addDocument(WDDocument aDocument) {
383      // FIXME DELETE THIS LINE! FOR TESTING ONLY!!
384      forceOpen();
385      ArrayList docList = new ArrayList();
386      docList.add(aDocument);
387      addDocuments(docList);
388  }
```

Line 384 remained from testing the prototype's functionality without taking care of open locks that some bindings have. This line has to be deleted in a future version.

[WDBindingMechanism.java | addDocuments(ArrayList)]

```
396  public void addDocuments(ArrayList someDocuments) {
397      if (isOpen() == OPEN) {
398          Iterator iterator = someDocuments.iterator();
399          while ( iterator .hasNext()) {
400              WDDocument doc = (WDDocument) iterator.next();

407              addChild(doc.toPNode());
408
409              // Let the document know who its binding is.
410              doc.setParentBindingMechanism(this);

419              new WDNodeDragger(this).dragToBindingMechanism(doc);
420          }
421      } else {
422          System.err
423                  . println ("Cannot␣add␣document(s),␣because␣binding␣is␣not␣open.");
424      }
425  }
```

Before adding a list of documents, the method tries to open the binding mechanism and checks if it is is open afterwards. If not, an error will be displayed. Then, an iterator iterates through the list, adding each document as a child to the mechanism and associating the mechanism at the document's side.[23] Finally, a new WDNodeDragger instance drags the node to an appropriate position. This supports the user in adding documents.

Removing documents from a binding works equivalent to adding. If a single document is passed to removeDocument(WDDocument), it is wrapped into a list, which is passed as argument to the method removeDocuments(ArrayList):

[WDBindingMechanism.java | removeDocuments(ArrayList)]

```
444  public void removeDocuments(ArrayList someDocuments) {
445      if (isOpen() == OPEN) {
446          Iterator iterator = someDocuments.iterator();
447          while ( iterator .hasNext()) {
```

[23] To avoid errors during runtime, it would be more secure to check if the individual object is an instance of WD|Document before casting at line 400.

```
448                 WDDocument doc = (WDDocument) iterator.next();
449                 getBinding().getWildDocs().addNodeToDesk(doc);
```

```
454             }
455         } else {
456             System.err
457                     . println ("Cannot␣remove␣document(s),␣because␣binding␣is␣not␣open.");
458         }
459     }
```

Also here, the method first tries to open the binding mechanism and returns an error message if it is not open afterwards. Otherwise, an iterator steps through the passed list and adds the documents individually to the *desk*.[24] This makes clear that removing is not equivalent to trash(), which removes a document completely from the space.

There is a method that removes all children of a binding:

[WDBindingMechanism.java | removeAllDocuments()]

```
465     public void removeAllDocuments() {
466         ArrayList children = new ArrayList(getChildrenReference());
467         children.remove(getGraphicalRepresentation());
468         removeDocuments(children);
469     }
```

The method gets all children of a binding mechanism, puts them in a list, and removes the graphical representation from it. Then, it calls removeDocuments(ArrayList), passing the list as attribute.

A binding can be removed from the space by calling trash() on it. This removes front and back cover, binding mechanism, and contained documents. If a user wants to trash a binding but keep the contained documents, he/she can open the mechanism and remove the complete content before calling the binding's trash method. However, this is only possible for bindings that have mechanisms that can be opened. For example, a book cannot be opened. We have implemented the method emptyAndTrashBinding() for those cases. It follows the metaphor of destroying the binding to free its content:

[WDBindingMechanism.java | emptyAndTrashBinding()]

```
476     public void emptyAndTrashBinding() {
477         forceOpen();
478         removeAllDocuments();
479         getBinding().trash() ;
480     }
```

Firstly, the binding is forced to open. Then, all documents are removed as described above. Finally, the "empty" binding is completely removed, and with it the binding mechanism and its graphical representation.

Forcing a binding mechanism to open follows the metaphor of "violently" opening the binding, even those that are not supposed to open after they are bound. The metaphor also includes that a binding that is "violently" opened is essentially "damaged"; and, therefore not to be used a second time, and because of that completely removed. A real world example would be a person ripping out all pages of a book ("violently open" and "removing pages").

[24] Like with addDocuments(ArrayList), it would be more secure to check if the passed document is an instance of WDDocument before casting at line 448, even though there should only be WDDocument instances left once the graphical representation is removed from the list.

The remaining parts are extensively damaged and will probably not be used anymore. However, the former contained pages can be rebound, for example, in binders or even as part of a newly created book.

Area The abstract class WDBindingAreaMechanism extends WDBindingMechanism. It is used by classes that support area shaped binding mechanisms. Currently, there is no additional code that goes beyond the one of the superclass. Its constructor is identical to the one in WDBindingMechanism. All attributes are passed to its superclass.

The concrete classes WDDeskMechanism and WDPageMechanism extend WDBinding|AreaMechanism. Both have areas as binding mechanisms: Documents on a desk can be moved around, so can objects on a page.

Mechanisms for desk and binding have the clip alignment set to CENTER. The constant PAGEOFFSET holds information about how much a document may overlap the mechanism area. The value is given in millimeters. Currently, the mechanism for desk has set this value to zero. This means that all documents have to be *completely* on the area; otherwise, an instance of WDNodeDragger is intended to drag it there after mouse release. The page mechanism has a maximum offset of -20 mm. This is an experimental value. It forces all documents that are placed on a page to respect a border of 20 mm around the page's bounds.

The constructor for WDDeskMechanism is identical to the one of WDPageMechanism, except for its name:

[WDDeskMechanism.java | WDDeskMechanism(WDBinding,Rectangle,Paint)]

```
65  public WDDeskMechanism(WDBinding aBinding, Rectangle aSize, Paint aColor) {
66
67      super(aBinding, NOT_OPENABLE, CLIPALIGNMENT, FULLROTATION, PAGEOFFSET);
68
69      updateGraphicalRepresentation(createRectangle(0, 0, aSize, aColor,
70              DEFAULTSTROKEWIDTH, DEFAULTSTROKECOLOR));
71  }
```

Arguments that identify the associated binding as well as size and color of the graphical representation of the mechanism are passed. The binding instance, as well as further information that is related to the binding's behavior is passed to the superclass at line 67. Those include for desk and page mechanisms the information that the binding is not to be opened after it is closed the first time, their clip alignment, as well as the information that added documents may be rotated by any angle (FULLROTATION), and the previously discussed offset for added documents.

Line 69 triggers the update of the graphical representation with a newly created rectangle of the class PPath. The called method creates and returns a rectangle with the given x and y coordiantes, size, background color, stroke width, and stroke color:

[WDDeskMechanism.java | createRectangle(float,float,Rectangle,Paint,float,Paint)]

```
90  private PPath createRectangle(float aXpos, float aYpos,
91          Rectangle aRectangle, Paint aBackgroundColor, float aStrokeWidth,
92          Paint  aStrokeColor) {
```

All lengths are converted from millimeters to pixels by WDUnitConverter. Currently, the code for creating the rectangle is identical for desk and page mechanisms.

Point and Line Figure 4.11 on page 129 depicts that the abstract class WDBindingPoint|⤶⤷|Mechanism inherits directly from WDBindingMechanism, whereas the abstract class WD|⤶⤷|BindingLineMechanism extends WDBindingPointMechanism. A point-based mechanism is used for bindings that bind documents at one single spot. A paper clip may be a valid example.

The constructor for point-based binding mechanisms takes the associated binding instance as well as information about whether it can be opened, its clip alignment, and the maximum rotation:

[WDBindingPointMechanism.java | WDBindingPointMechanism(WDBinding,boolean,int,double)]

```
56  public WDBindingPointMechanism(WDBinding aBinding, boolean ifOpenable,
57          int aClipAlignment, double aMaxRotation) {
58
59      super(aBinding, ifOpenable, aClipAlignment, aMaxRotation, NO_OFFSET);
60  }
```

It calls its superclass and passes all information unmodified, adding NO_OFFSET to indicate that an offset is not requested. Documents are fixed at the binding mechanism's position.

Line-based mechanisms are special cases of point-based ones. They also do not allow an offset; however they restrict their maximum rotation: Line mechanisms do not allow rotation. For instance, pages of a book cannot be rotated individually, because they are bound along a line. The line mechanism's constructor has fewer arguments than the point-based one:

[WDBindingLineMechanism.java | WDBindingLineMechanism(WDBinding,boolean,int)]

```
65  public WDBindingLineMechanism(WDBinding aBinding, boolean ifOpenable,
66          int aClipAlignment) {
67
68      super(aBinding, ifOpenable, aClipAlignment, NO_ROTATION);
69  }
```

It passes all given arguments unmodified to its superclass, which is WDBindingPoint|⤶⤷|Mechanism, and adds with NO_ROTATION the information that rotation is not possible for this kind of binding mechanism.

It can be criticized that in the real world, some line-based binding mechanisms show little rotation of their documents, for example, as Fig. 4.8 on page 120 depicts. Therefore, some rotation should be also possible for their implementation in a future version. It can be argued that merging point and line dimensions would be the preferable solution.

Currently, there are no examples implemented that use point-based binding mechanisms, but two implemented concrete classes are based on line mechanisms: WDBookMechanism and WDSheetMechanism. The constructor for both look identical, except for their names:

[WDBookMechanism.java | WDBookMechanism(WDBinding)]

```
59  public WDBookMechanism(WDBinding aBinding) {
60      super(aBinding, NOT_OPENABLE, WDBindingClipCalculator.CENTER);
61  }
```

It passes the binding instance to its superclass, adds information that it is not for being opened, and passes also the binding clip alignment, which is set to CENTER.

The graphical representation of book and sheet binding mechanism differ, though. WD|⤶⤷|BookMechanism overrides open() and close() and passes different colored rectangles to indicate whether a binding mechanism is open or closed:

[WDBookMechanism.java | open()]

```
72  public void open() {
```

```
73          open(createRectangle(0, 0, 5, 100, 2, Color.GREEN));
74          //open(new WDBox(getBinding().getWildDocs(), 0, 0, 5, 100, 2, Color.GREEN));
75      }
```

An open mechanism is represented in green. The rectangle creation is equivalent to the ones for WDDeskMechanism or WDPageMechanism:

[WDBookMechanism.java | createRectangle(float,float,double,double,float,Color)]

```
98      private PPath createRectangle(float aXpos, float aYpos, double aWidth,
99              double aHeight, float aStrokeWidth, Color aColor) {
```

If the mechanism is closed it changes its color from green to red:

[WDBookMechanism.java | close()]

```
80      public void close() {
81          close(createRectangle(0, 0, 5, 100, 2, Color.RED));
82      }
```

WDSheetMechanism overrides the method open() and passes a node that is returned by createGlue():

[WDSheetMechanism.java | open()]

```
70      public void open() {
71          open(createGlue());
72      }
```

The graphical representation which is named “glue” symbolizes that a sheet is created by two pages that are “glued” together. The glue is an invisible PPath object with zero size. Its only purpose is to have an object associated that stands for the graphical representation, even though it is not visible:

[WDSheetMechanism.java | createGlue()]

```
82      private PPath createGlue() {

88          PPath graphics = PPath.createRectangle(0, 0, 0, 0);
89          graphics.setVisible(INVISIBLE);
90
91          return graphics;
92      }
```

4.3. Machines

The package de.atzenbeck.wilddocs.machines contains classes that are intended to modify documents or perform related calculations. We follow the idea of extracting behavior from objects and use specialized classes for processing.

4.3.1. Rotation

Instances of the class WDNodeRotator take care of purposeful as well as incidental rotation of documents. Each instance of WDNodeInputEventHandler initializes a new node rotator, passing the mouse event that occurred on the document. The constructor in WDNode|Rotator gets information about the node from the passed attribute. The final constructor sets a reference to the node and the event to be used later as well as resets the previous random

rotation angle. setNode(aNode) also sets the rotation angle of the passed node, which can be accessed through getOriginalRotation() later.

[WDNodeRotator.java | WDNodeRotator(PNode,PInputEvent)]

```
108  public WDNodeRotator(PNode aNode, PInputEvent aEvent) {
109      setNode(aNode);
110      setRotationMiddle(aEvent);
111      setPreviousRandomRotation(0);
112  }
```

If the given event option is null, a random point somewhere within the document's area will be set as rotation middle; otherwise, the cursor position that is referenced in the event object will be taken.

Purposeful Rotation

Purposeful rotation is initiated by WDNodeInputEventHandler when the user double clicks on a document and holds the mouse button at the second click. Firstly, it calls paintRotation| ↩ |MiddleMark() in WDNodeRotator, that draws a little circle where the click occurred. This point will be used as the center of the rotation. The mode of the event handler is changed to purposeful rotation. While this mode is active, all mouse drag events will call rotateOn| ↩ |Purpose(PInputEvent), passing the input event that is used to calculate the rotation angle:

[WDNodeRotator.java | rotateOnPurpose(PInputEvent)]

```
476  public void rotateOnPurpose(PInputEvent aEvent) {
477      // Rotate only if WildDocs has rotation on.
478      if (WildDocs.PURPOSEFUL_ROTATION) {
479
480          double a = aEvent.getPositionRelativeTo(getNode()).getX()
481                  - getRotationMiddle().getX();
482          double b = aEvent.getPositionRelativeTo(getNode()).getY()
483                  - getRotationMiddle().getY();
```

Firstly, the relative distance to the rotation center is calculated, then the angle. The document's original rotation is used at line 507, because the new angle will be interpreted as absolute, not relative:

[WDNodeRotator.java | rotateOnPurpose(PInputEvent)]

```
502          double angle_pur = -Math.atan(a / b);
503          if (b > 0) {
504              angle_pur += Math.toRadians(180);
505          }
506
507          double angle = angle_pur + getOriginalRotation();
```

A static constant may be set to force purposeful rotation to snap into given angles. This is checked by the if-clause at line 512 and calculated if required. The rotation method is called at line 518:

[WDNodeRotator.java | rotateOnPurpose(PInputEvent)]

```
512          if (WildDocs.PURPOSEFUL_ROTATIONSPACING > 0) {
513              double spacing = Math
514                      .toRadians(WildDocs.PURPOSEFUL_ROTATIONSPACING);
```

Action	Mouse Event	Factor
Touching a document	mouse press	0.9
New document loaded	–	1.1
Dragging a document	mouse drag	speed/50
Alternative rotation	mouse release	5.0

Table 4.4.: Current predefined rotation factors for random rotation, based on experiments to reach realistic behavior

```
515                 angle -= angle % spacing;
516             }
517
518             rotateNode(angle, getRotationMiddle());
519         }
520     }
```

Finally, WDNodeInputEventHandler calls a method at WDNodeRotator that removes the rotation center mark and finally resets the purposeful rotation mode.

Incidental Rotation

Incidental rotation happens automatically on a document while it is being dragged, when a new document is put onto the space, or when the mouse button is pressed on document. The latter one simulates touching a document. Those actions are directly related to the user's mouse interaction and initiated by WDNodeInputEventHandler. We will discuss an alternative behavior at the end of this section, which is based on random rotation on mouse release events.

The calculation of the random rotation takes different factors, depending on the action. Table 4.4 lists those. Our model is different to the one described by Beaudouin-Lafon (2001) and depicted in Fig. 2.15 on page 52. This model imitates dragging a document with one finger, whereas WildDocs imitates moving it with the complete hand.

WildDocs's incidental rotation formula is based on experiments we made with real paper on a real desk. We used a camera to record the movement of paper on the desk. Later, we cut the movie into key frames and measured the documents' angles on those. The angle calculation formula is simple, but delivers a level of spatial sloppiness that looks realistic and is acceptable for our purpose:

[WDNodeRotator.java | calculateRandomRotation(PInputEvent)]

```
420         double angle = Math.toRadians((Math.random() * (2 * factor) - factor))
421                 - getPreviousRandomRotation();
422
423         setPreviousRandomRotation(angle);
```

The rotation angle does not accumulate, therefore the previous random rotation angle is subtracted. The subtraction of the factor makes negative values possible. Caused by that, we need to apply two times the random number to allow also positive values.[25]

[25] Assume r is a random number with $0 \leq r < 1$, as produced with Math.random(). Further, assume a factor $f > 0$. Then, rf produces only non-negative numbers with $0 \leq rf < f$. On the other side, $rf - f$ produces only negative

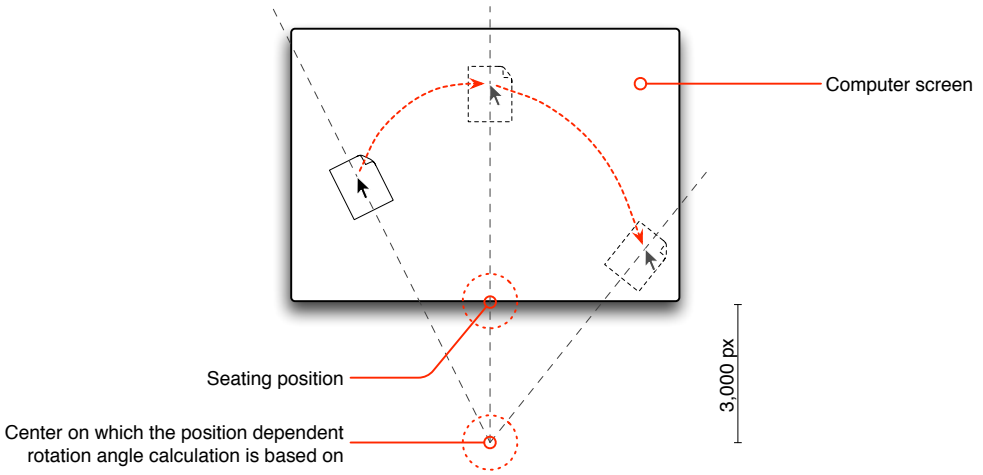


Figure 4.12.: Calculating incidental rotation angle by position

The result is an animated behavior on the screen while dragged. The incidental rotation relevant rotation middle point is the point where the mouse was clicked on the document. A pressed mouse button on a node applies a random rotation, calculated and performed by WDNodeRotator. The factor used for touch simulation ("touch factor") is set to 0.9. This has shown realistic results in our experiments. Additionally, some random offset is applied.

The above printed formula for calculating the rotation angle is also used when new nodes are created and put onto the space. However, the factor we use is 1.1. This gives slightly sloppier piles than the touch factor would.

Incidental rotation that is applied while a document is dragged takes two aspects into consideration: Firstly, the random rotation with a speed of movement; and, secondly, the position of the cursor on the space. Both are called by mouseDragged(PInputEvent) in WDNode| ↩ ↪ |InputEventHandler while the node is dragged:

[WDNodeInputEventHandler.java | mouseDragged(PInputEvent)]

```
149                 getNodeRotator().rotateByDirection(aEvent);
150                 getNodeRotator().rotateRandomly(aEvent);
```

Figure 4.12 depicts the idea behind position dependent rotation. This is based on experiments in which we have recorded and measured how paper is moved on a real desk. Partly based on the shoulder position as well as the possibility to reduce the radius of an arm, for example, by the elbow, the rotation angle of a moved document does not have its center at the person's seating position. Our experiments have shown that it is mostly a point behind the person. WildDocs infers the seating position of the user at the middle lower part of the screen. The seating position follows panning or zooming. We set the center on which the rotation angle is based 3,000 pixels below the seating position. When a document is dragged, its rotation follows a line that goes through that point and the current mouse position, as depicted in Fig. 4.12.

numbers with $-f \leq (rf - f) < 0$. The term $2rf - f$ produces both positive and negative numbers, with $-f \leq (2rf - f) < f$.

[WDNodeRotator.java | rotateByDirection(PInputEvent)]

```
448             double a = getDirectionCircleMiddle().getX()
449                     − aEvent.getPosition().getX();
450             double b = getDirectionCircleMiddle().getY()
451                     − aEvent.getPosition().getY();

463             double angle = −Math.atan(a / b) − getNode().getRotation()
464                     + getRelativeRotationAtStart();
465
466             rotateNode(angle, getRotationMiddle());
```

The angle is calculated by the absolute angle that the document should have according to its position ($-\operatorname{atan}\frac{a}{b}$) minus the previous rotation plus the rotation of the object at click time.

The random rotation calculation is the same as discussed above; however, the factor that is used for the formula is not static, but depends on the mouse speed. The faster the mouse moves, the larger the factor becomes. This results in possibly larger angles during faster movement.

[WDNodeRotator.java | calculateRandomRotation(PInputEvent)]

```
409                 factor  = getMouseSpeed(aEvent) / 50;
```

The mouse speed is based on information from the passed mouse event. The method getMouseSpeed(PInputEvent) extracts the time when the event occurred and calculates the difference to the previous event. It requests the distance that the mouse travelled since the last event notification. This is calculated by Pythagoras's formula on the delta values for x and y coordinates. Delta time and delta speed are used for gaining the speed of the cursor in pixels (dependent on canvas coordinate system) per millisecond.

[WDNodeRotator.java | getMouseSpeed(PInputEvent)]

```
609     public double getMouseSpeed(PInputEvent aEvent) {
610         long eventTime = aEvent.getWhen();
611         long deltaTime = eventTime − getPreviousEventTime();
612         double speed = getMouseDistance(aEvent) / deltaTime;
613         setPreviousEventTime(aEvent);
614
615         return speed;
616     }
```

Caused by machine dependent problems running WildDocs on Windows XP or Linux, we had to switch off the above discussed animated and mouse speed dependent incidental rotation and introduce a simple mechanism that would have a comparable result. We will explain the reasons in Sect. 5.2.2. Now, WDNodeInputEventHandler calls on each mouse release event a random rotation similar to the one for "touching" a document; however, the factor is set to 5. Tests have shown that this value produces acceptable results for our usability test. If this mode is on, the touch behavior connected to mouse press events on documents is disabled.

4.3.2. Node Dragging

Documents are dragged by the user by simply clicking and dragging using the mouse. However, some instances require WildDocs to drag an object. Those include applying a random offset or adjusting the position of a document when added to a binding. The class WDNodeDragger takes care of this.

Random Offset

Random offsets may be applied in two situations: Firstly, when a new node is created and put onto the desk; and, secondly, when a node is "touched" via pressing the mouse on it. This simulates behavior of real paper on a desk, similar to that discussed in Sect. 4.3.1 on random rotation. WDNodeInputEventHandler initiates the random offset call, passing the document that has to be dragged as well as the maximum offset size in millimeters. In our code the offset size is small, which equals a maximum offset of 5 mm.

[WDNodeInputEventHandler.java | mousePressed(PInputEvent)]

```
196         WDDocument doc = (WDDocument) aEvent.getPickedNode();
197         WDNodeDragger dragger = new WDNodeDragger(doc);
198         dragger.dragRandomly(doc, WDNodeDragger.SMALL_MAX_RANDOM_OFFSET);
```

Passing the document to the constructor is essential, because information about the WildDocs instance is extracted, which is used for unit conversions from millimeters to pixels. If m is the given maximum offset, then WDNodeDragger calculates the random offset o for x or y coordinates with $-m \leq o < m$. This is analogue to the formula discussed on page 138.

[WDNodeDragger.java | calculateRandomOffset(double)]

```
528     double xOffset = (2 * aMaxOffset * Math.random()) - aMaxOffset;
529     double yOffset = (2 * aMaxOffset * Math.random()) - aMaxOffset;
```

The WildDocs instance creates a new WDNodeDragger object, when a new document is added to the desk or space, but only if the given boolean parameter ifRandomOffset is set to true. A huge maximum random offset is applied, currently set to 100 mm.

[WildDocs.java | addNodeToDesk(WDDocument,boolean,boolean)]

```
1162    if (ifRandomOffset) {
1163        new WDNodeDragger(this).dragRandomly(aDoc,
1164                WDNodeDragger.HUGE_MAX_RANDOM_OFFSET);
1165    }
```

Binding Mechanism Support

As discussed in Sect. 4.2.3, bindings are a spin-off of our project. The binding mechanism support exists in WildDocs in its basics. This is true also for external behavior support. To avoid that users have to drag documents exactly to binding mechanisms, WildDocs infers that a document that is placed roughly with its binding side on top of an open binding mechanism is supposed to be put into the binding.

WDNodeInputEventHandler checks on every mouse release if the active node is an instance of WDDocument.[26] Only those instances can be added to a binding mechanism. Another check is whether WildDocs has complex bindings activated. Only those require binding mechanism support. Then, the active document is passed to a WDNodeDragger instance to extract the binding mechanism (line 275), and finally to the extracted binding mechanism to be added (line 278):

[WDNodeInputEventHandler.java | mouseReleased(PInputEvent)]

```
264         PNode activeNode = aEvent.getPickedNode();
```

[26] As it is now, it does not work if a user drags a binding on its mechanism onto another binding in order to add it. A binding mechanism is not an instance of WDDocument and would therefore not be considered to be possibly added. This is a bug needs to be addressed in the future.

```
271             if (activeNode instanceof WDDocument) {
272                 WDDocument activeDoc = (WDDocument) activeNode;
273
274                 if (WildDocs.COMPLEXBINDINGS) {
275 WDBindingMechanism bindingMechanism = new WDNodeDragger(
276                             activeDoc).checkDragToBindingMechanism(activeDoc);
277                     if (bindingMechanism != WDNodeDragger.NO_BINDINGMECHANISM) {
278                         bindingMechanism.addDocument(activeDoc);
279                     }
280                 }
```

The method checkDragToBindingMechanism(WDDocument) takes all objects that are on the space and keeps only those that are open binding mechanisms, and below and intersecting with the active document:

[WDNodeDragger.java | checkDragToBindingMechanism(WDDocument)]

```
552         relevantMechanisms.keepFiltered(new OpenBindingMechanismFilter());
553         relevantMechanisms.keepFiltered(new SmallerNodeIndexFilter(docNode));
554         relevantMechanisms.keepFiltered(new IntersectionFilter(docNode));
```

An iterator is run over the relevant binding mechanisms to discard those that have another object between the checked binding mechanism and the active document. This simulates the real world in which a document cannot be added to a binding if another objects is in between:

[WDNodeDragger.java | checkDragToBindingMechanism(WDDocument)]

```
561         Iterator relevantMechanismsIterator = new HashSet(relevantMechanisms)
562                 .iterator();
563         while (relevantMechanismsIterator.hasNext()) {
564             PNode mechanism = (PNode) relevantMechanismsIterator.next();
565             WDTempNodeStorage nodesBetween = new WDTempNodeStorage(root
566                     .getAllNodes());
567
568             nodesBetween.keepFiltered(new NodeInBetween(docNode, mechanism));
569             nodesBetween.keepFiltered(new IntersectionFilter(mechanism));
570             nodesBetween.keepFiltered(new DocumentFilter());
571
572             if (!nodesBetween.isEmpty()) {
573                 relevantMechanisms.remove(mechanism);
574             }
575         }
```

If there is at least one open binding mechanism that has the dragged document on top without having any other object in between, then the remaining binding mechanism which is closest to the document will be returned. That is the one with the highest index value. Otherwise, a constant will be returned, representing the state of not having any relevant binding mechanisms:

[WDNodeDragger.java | checkDragToBindingMechanism(WDDocument)]

```
583         if (!relevantMechanisms.isEmpty()) {
584             return (WDBindingMechanism) relevantMechanisms
585                     .getHighestIndexNode();
586         } else {
587             return NO_BINDINGMECHANISM;
```

```
588         }
589     }
```

WDNodeInputEventHandler uses the returned WDBindingMechanism reference to call addDocument(WDDocument).[27] After checking if the binding mechanism is open, the document is added as child to the mechanism and a reference to the mechanism at the document is set. Finally, a new WDNodeDragger instance is created with the mechanism as parameter, that drags the document to the binding mechanism.

[WDBindingMechanism.java | addDocuments(ArrayList)]

```
419                 new WDNodeDragger(this).dragToBindingMechanism(doc);
```

Before calling the animation, WDNodeDragger calculates the destination rotation. This depends on the maximum possible rotation. For example, even binders allow a margin for some rotation, whereas books do not. The document's absolute rotation is the sum of the random value plus the current rotation of the mechanism's graphical representation plus the standard clip alignment.

[WDNodeDragger.java | dragToBindingMechanism(PNode,int)]

```
381         double maxBindingRotation = getBindingMechanism().getMaxRotation();
382
383         double docRotation = (Math.random() * maxBindingRotation - (maxBindingRotation / 2))
384                 + getBindingMechanism().getGraphicalRepresentation()
385                         .getRotation()
386                 + (new WDBindingClipCalculator().rotation(aClipAlignment));
```

The span of the random rotation needs to be equal or smaller than the maximum rotation allows. However, the angle may be negative or positive. Therefore, the maximum rotation value divided by two is subtracted. The graphical representation is necessary, because the binding may be put with an angle itself. Finally, there is a clip alignment that stores on which side a document needs to be put on the binding mechanism, for example, on the left hand side, which would be zero degrees, or on the top, which would be -90 degrees. This is calculated by an instance of WDBindingClipCalculator which will be discussed in the next section.

After the destination rotation, also the destination location is calculated. Finally, the movement animation is created, scheduled, and executed. The duration passed via the constant ANIMATIONDURATION, is set to 500 ms.

[WDNodeDragger.java | dragToBindingMechanism(PNode,int)]

```
472         double docScale = aNode.getScale();
473
474         PActivity movement = aNode.animateToPositionScaleRotation(destPosX,
475                 destPosY, docScale, docRotation, ANIMATIONDURATION);
476
482         getBindingMechanism().getBinding().getWildDocs().getCanvas().getRoot()
483                 .addActivity(movement);
484         movement.setStartTime(System.currentTimeMillis());
```

4.3.3. Calculating Binding Clip Positions

Each instance of WDBindingMechanism or WDDocument has a clip alignment. They are used when documents are added to bindings: Clips of mechanism and document will be put

[27] addDocument(WDDocument) wraps the document into an ArrayList instance and passes it to addDocuments(ArrayList). This allows iteration through the list and add each document.

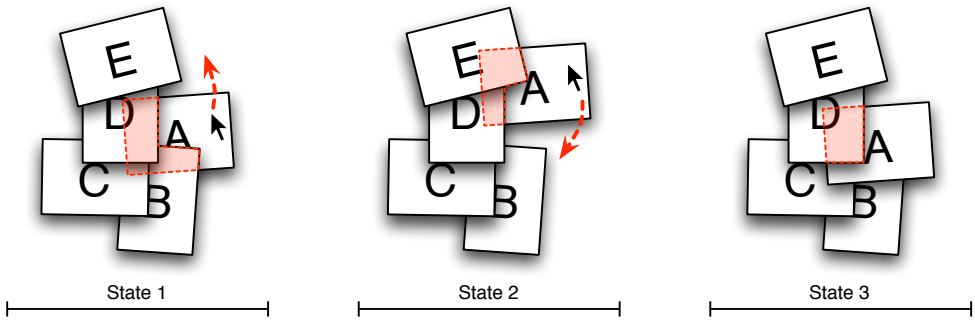


Figure 4.13.: Index pusher behavior

together by an animated movement of the document, initiated and calculated by WDNode| ⤸ ⤷ |Dragger. However, the calculation of clip alignments and locations is done by WD| ⤸ ⤷ |BindingClipCalculator.

Currently, there are predefined clip positions for center, left, right, top, or bottom. The graphical representation as well as the clip alignment are extracted from the given binding mechanism and passed to the appropriate method:

[WDBindingClipCalculator.java | position(WDBindingMechanism)]

```
80  public Point2D position(WDBindingMechanism aMechanism) {
81      return position(aMechanism.getGraphicalRepresentation(), aMechanism.getClipAlignment| ⤸
            ⤷ |());
82  }
```

Similarly, given document references are used to extract the underlying node and the clip alignment:

[WDBindingClipCalculator.java | position(WDDocument)]

```
90  public Point2D position(WDDocument aDocument) {
91      return position(aDocument.toPNode(), aDocument.getClipAlignment());
92  }
```

4.3.4. Index Pushing

Instances of WDNodeIndexPusher take care of pushing node indices automatically. They push a document up if it leaves the scope of the above positioned ones. Figure 4.13 depicts an example of how it works. An user moves document *A*, which is the bottom most document of a pile of five documents (state 1). Intersections of documents above *A* with node *A* are marked with transparent red polygons. The document never leaves its intersection with document *D*, which is below *E*. However, it leaves the intersection with *C* and *B* (state 2). When moving back the document, it is still below *D*, but appears on top of *C* and *B* (state 3). Its index is "pushed up", because it left the scope of *B* and *C*. This happens automatically while dragging a document, and is based on observations on paper that follows gravity as well as the carrier's force that holds against it.

Figure 4.14 depicts a limitation of the index pusher. This is based on a limitation of the underlying Piccolo framework. The original positions are: document *D* on top of *C*, *C* on

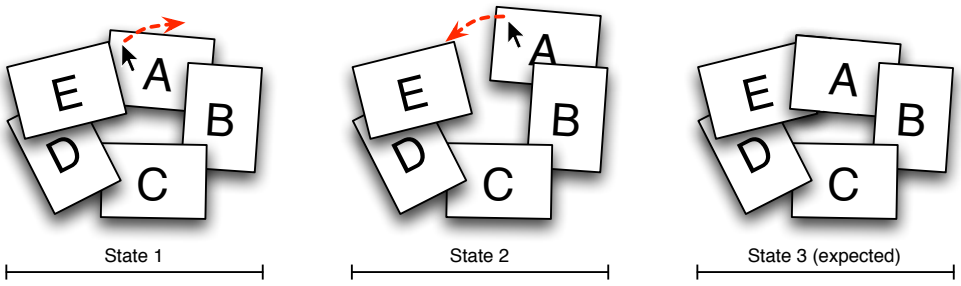


Figure 4.14.: Index pusher limits – indices chain

top of *B*, and *B* on top of *A*. Document *E* is above *D* and *A* (state 1). We assume now that the user moves document *A* outside of document *E*'s scope, still remaining below *B* (state 2). Somebody would expect the depicted state 3, where document *A* is pushed on top of *E*, but still remains below *B*. However, Piccolo's implementation of node indices, which are similar to layers, does not allow such constellations. Document *A* would be moved below *E* as long as it does not leave the scope of any of the documents, that indices are smaller than *E*'s index. Currently, the result would look similar to state 1.

There is no easy solution that addresses this problem. One idea is to over paint the area that needs to be visible. However, this draws several new problems. The area would need to be painted correctly, even while the object is moved. Once moving is finished, mouse clicks on this area would need to be passed to the right document, which is not the one that the framework would take normally. Another problem would occur if the node underneath the painted area would be moved. The painted node that is according to its index below would have to be painted constantly on top of the moved one. After thinking about cost-benefit, we decided not to invest into a solution of this problem yet.

WDNodeInputEventHandler instantiates a WDNodeIndexPusher object when the mouse button is pressed.

[WDNodeInputEventHandler.java | helperInitialization(PInputEvent)]

```
102         setNodeIndexPusher(new WDNodeIndexPusher(aEvent, getWildDocs()));
```

The passed event is used by WDNodeIndexPusher's constructor to extract the active node. The method initRelevantCluster(PNode) creates an array of all objects that are relevant, that are all above the active node and part of the same cluster. The latter clause is calculated by updateRelevantCluster(), called at line 169:

[WDNodeIndexPusher.java | initRelevantCluster(PNode)]

```
165     private void initRelevantCluster(PNode aNode) {
166         setRelevantCluster(new WDTempNodeStorage(getWildDocs().getLayer()
167                 .getAllNodes()));
168         getRelevantCluster().keepFiltered(new LargerNodeIndexFilter(getNode()));
169         updateRelevantCluster();
170         putNodesOnTop(getRelevantCluster().sortNodesOnIndex());
171     }
```

We will see later, that updateRelevantCluster() is also called sometimes during dragging. It takes the existing relevant cluster and removes all references to objects that are no longer part

of the cluster on top of the active node.

[WDNodeIndexPusher.java | updateRelevantCluster()]

```
176  public void updateRelevantCluster() {
177      getRelevantCluster().keepFiltered(
178              new ClusterOnTopFilter(new WDClusterRecognizer(getNode(),
179                      getWildDocs().getLayer().getAllNodes())));
180
181      /*
182       * The active node should be always contained by the relevant cluster
183       * list . The following if-clause is just for cases when this is not the
184       * case (e.g., caused by some program changes).
185       */
186      if (!getRelevantCluster().contains(getNode())) {
187          getRelevantCluster().add(getNode());
188      }
189  }
```

The update method passes a filter to the instance of WDTempNodeStorage, which holds all references to relevant objects. The filter of the class ClusterOnTopFilter builds the cluster. The array keeps only those that are accepted by the filter. ClusterOnTopFilter makes use of WDClusterRecognizer, which calculates boundaries of relevant clusters. This class will be explained in the next section.

Once the index pusher instance is created on pressing the mouse button, WDNodeInput|EventHandler calls the associated index pusher on mouse drag events in order to push the dragged node to a higher index level, if required:

[WDNodeInputEventHandler.java | mouseDragged(PInputEvent)]

```
155      getNodeIndexPusher().pushToHigherIndexLevel();
```

The index push procedure is based on an iterator, iterating through relevant objects, including the dragged document itself. If one of them loses the intersection with the dragged document, the cluster of all relevant objects will be updated, sorted, and put to the very top. It appears to the user as if the dragged node only would have been pushed to a higher index level:

[WDNodeIndexPusher.java | pushToHigherIndexLevel()]

```
194  public void pushToHigherIndexLevel() {
195      if (WildDocs.INDEXPUSHER) {
196          Iterator iterator = new HashSet(getRelevantCluster()).iterator();
197          while ( iterator .hasNext()) {
198              PNode checkedNode = (PNode) iterator.next();
199              /*
200               * if the intersection is lost , re-calculate the cluster on top
201               */
202              if (checkedNode.fullIntersects(getNode()
203                      .getFullBoundsReference()) == false) {
204                  updateRelevantCluster();
205                  putNodesOnTop(getRelevantCluster().sortNodesOnIndex());
206                  break;
207              }
208          }
209      }
```

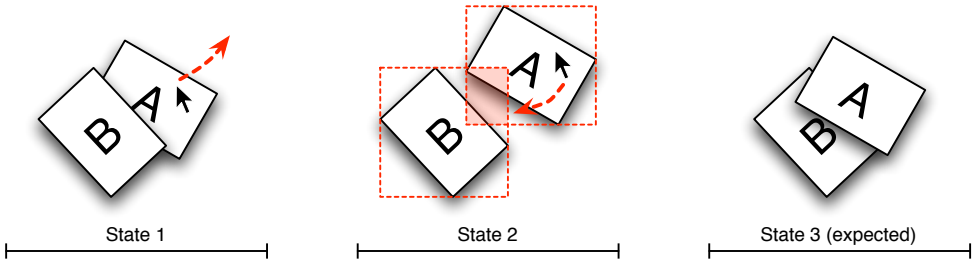


Figure 4.15.: Index pusher limits – scope and rotation

```
210     }
```

The check for intersection works only reliable for documents that are not rotated. The problem is based on Piccolo's lack of rotated bounds. Figure 4.15 depicts the problem. Assume two documents *A* and *B* with *B* on top of *A* (state 1). The user drags *A* slightly to the right. Document *A* appears to be outside *B*'s scope (state 2). However, neither *A*'s nor *B*'s bounds are rotated, but outline the graphical representation. They are depicted as red dotted lines. The transparent red area shows their intersection. When moving back document *A*, someone would expect *A* on top of *B* (state 3), especially because the actual bounds are not visible. However, *A* goes back underneath *B* again, similar to state 1. WDNodeIndexPusher only works correctly if the document's invisible bounds are moved outside the invisible bounds of the other document.

4.3.5. Cluster Recognition

WDClusterRecognizer is used for making implicit clusters explicit. The constructor takes a document or node as well as a collection of nodes. Its most important method is calc| ⤸ ⤹ |ClusterOnTop(). It draws all objects from a given collection that are on top of a given node and member of the cluster to which also the given node belongs to:

[WDClusterRecognizer.java | calcClusterOnTop()]

```
117     public WDTempNodeStorage calcClusterOnTop() {
118         WDTempNodeStorage buildClusterOnTop = new WDTempNodeStorage();
119         WDTempNodeStorage checkForIntersections = new WDTempNodeStorage();
120         WDTempNodeStorage checkedNodes = new WDTempNodeStorage();
121 
122         checkForIntersections.add(getNode());
123         buildClusterOnTop.add(getNode());
124 
125         while (!checkForIntersections.isEmpty()) {
126             PNode currentlyCheckedNode = (PNode) checkForIntersections.get(0);
127             if (checkedNodes.contains(currentlyCheckedNode) == false) {
128                 WDTempNodeStorage intersectNodes = new WDTempNodeStorage(
129                         getCollection());
130 
131                 intersectNodes.keepFiltered(new IntersectionFilter(
132                         currentlyCheckedNode));
```

```
133                 intersectNodes.keepFiltered(new LargerNodeIndexFilter(currentlyCheckedNode));
134                 intersectNodes.removeFiltered(new AdornmentFilter());
135
136                 Iterator iterator = intersectNodes.iterator();
137                 while (iterator.hasNext()) {
138                     Object o = iterator.next();
139                     if (buildClusterOnTop.contains(o) == false) {
140                         buildClusterOnTop.add(o);
141                         if (checkForIntersections.contains(o) == false
142                                 && checkedNodes.contains(o) == false) {
143                             checkForIntersections.add(o);
144                         }
145                     }
146                 }
147
148                 checkedNodes.add(currentlyCheckedNode);
149             }
150             checkForIntersections.remove(0);
151         }
```

The first part of the method takes the given node as start object (line 122). It checks all intersecting nodes that are also part of the given collection. It keeps only those that have a larger index than the currently checked node (line 133). Further, it discards all adornments, such as shadows, bound lines, etc. (line 134).

The iterator at line 136 iterates through the remaining nodes. If the node is not part of the cluster references already, it will be added (line 140). Further, it is put to the array of nodes that needs to be checked, if it is not part already of this array, and if it was not checked previously (line 143).

The last part iterates through the gained references and removes all descendants of nodes that are on the list. The reason is that their index will be pushed automatically with their ancestors' indices:

[WDClusterRecognizer.java | calcClusterOnTop()]

```
162         Iterator iterator = new HashSet(buildClusterOnTop).iterator();
163         while (iterator.hasNext()) {
164             PNode node = (PNode) iterator.next();
165             buildClusterOnTop.removeFiltered(new DescendentFilter(node));
166         }
167
168         return buildClusterOnTop;
169     }
```

An improvement of this algorithm would be to ignore those nodes from iterating that were already removed as descendants at line 165, instead of iterating through those as well.

4.3.6. Unit Conversion

Java's coordinate system and units are different to the one used in WildDocs interface. Internally, however, WildDocs uses the same as Java. In Java, the origin of the coordinate system is the top left corner. Increasing y values are represented further down. In WildDocs, the origin is the bottom left corner. Increasing y values are represented further up.

Units in Java are pixels, whereas WildDocs uses for most parameters millimeter at the canvas's level, independent of zoom. Instances of the class WDUnitConverter convert between both systems. The constructor calls an initialization method. This sets the pixels per millimeter and is used for conversions. WildDocs is thus independent of the output device's resolution.

[WDUnitConverter.java | setPixelsPerMM()]

```
230    public void setPixelsPerMM() {
231        pixelsPerMM = Toolkit.getDefaultToolkit () .getScreenResolution() / 25.4;
232    }
```

4.3.7. Node Factory

Instances of WildDocs delegate the creation of Piccolo nodes to WDNodeFactory. WildDocs's factory pattern includes support for graphics (GIF, JPEG, or PNG), plain text, as well as formatted text (HTML or RTF). Currently, WildDocs infers the type based on the file's suffix at load time.

4.3.8. Turning Documents (Obsolete)

The class WDDocTurner supports turning documents that consists of a front and a back, both instances of WDLowLevelDoc. After complex bindings were introduced, WDDocTurner became obsolete. There is still a need to turn documents; however, the notion of data at a specific structure level has been replaced with flexible binding support, especially of the type SHEET or PAGE, as discussed in Sect. 4.2.3. It is open to future work to adapt WDDocTurner to be used for complex bindings

4.4. Interaction

Basic interaction methods are provided by Piccolo, such as dragging nodes or panning the background. WildDocs classes that support interactions can be found in de.atzenbeck|↩|.wilddocs as well as in de.atzenbeck.wilddocs.util.

4.4.1. Bounds Handle

WDBoundsHandle extends Piccolo's class PBoundsHandle. They are used for resizing nodes. Examples are depicted in Fig. 4.6 on page 112 or Fig. 4.18 on page 160. The main reason why we developed an own class for WildDocs was to support removing and recalculating borders on WDLowLevelDoc objects.

Every bounds handle is a child of the node that is the target of the handle's interactions. As soon as the handle is dragged, WDBoundsHandle calls startHandleDrag(Point2D,PInput|↩|Event). The most important part of this method is the following:

[WDBoundsHandle.java | startHandleDrag(Point2D,PInputEvent)]

```
139        // Get the handler's parent.
140        PNode node = aEvent.getPickedNode().getParent();
141
142        // Remove the border if there is one.
```

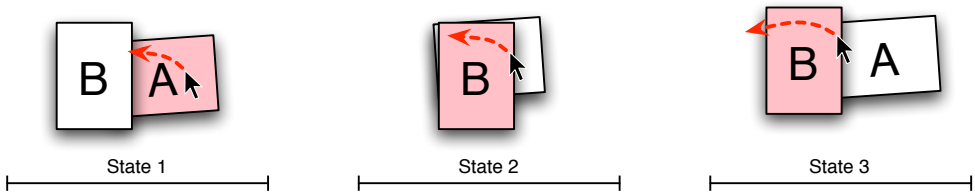


Figure 4.16.: Changing active node on mouse over

```
143         if (node instanceof WDLowLevelDoc) {
144             WDLowLevelDoc doc = (WDLowLevelDoc) node;
145             if (doc.getBorder() != null) {
146                 doc.removeBorder();
147             }
148         }
```

If the target node is an instance of WDLowLevelDoc and has a border, the border will be removed. Similarily, after releasing the handle, the method endHandleDrag(Point2D,PInput| ⤸ ⤷ |Event) is called:

[WDBoundsHandle.java | endHandleDrag(Point2D,PInputEvent)]

```
160         // Get the handler's parent.
161         PNode node = aEvent.getPickedNode().getParent();
162
163         // Repaint the border if borders are on.
164         if (node instanceof WDLowLevelDoc && WildDocs.LOWLEVELDOCBORDER) {
165             WDLowLevelDoc doc = (WDLowLevelDoc) node;
166             doc.createBorder();
167         }
```

The method checks if the node to which the bounds handle is attached is an instance of WDLowLevelDoc and if the WildDocs instance has borders switched on. If both are true, a new border will be created.

4.4.2. Change Active Node on Mouse Over

WildDocs implements behavior inspired by the real world. One is changing the active node when another node comes in between the dragged one and the mouse pointer. Figure 4.16 depicts this case. Red colored documents indicate the current mouse focus. Document *A* is behind *B*. The user drags *A* underneath *B* (state 1). As soon as the cursor is on top of *B* (state 2), *B* gets the focus and is moved (state 3) without having the user to release and press the mouse button. This behavior is similar to the real world where we cannot move a document that is behind another one "through" the one on top. The class WDCanvas takes care of changing the focus to another document. It extends Piccolo's PCanvas and is capable of adding additional or modifying existing input events.

Events are handed over from (1) Java's event queue to (2) Piccolo's PCanvas. From there they are passed to (3) PInputManager where they get converted from InputEvent (package java.awt.event) to (4) Piccolo's PInputEvent. Then, they are passed to the appropriate listener.

WDCanvas can modify events when they enter the Piccolo framework. For wrapping the input event and type, it uses a data structure, provided by the inner class EventAndType. The method sendInputEventToInputManager(InputEvent,int) overrides the method in PCanvas and is the central part of WDCanvas.

[WDCanvas.java | sendInputEventToInputManager(InputEvent,int)]

```
241  protected void sendInputEventToInputManager(InputEvent aEvent, int aType) {
242      if (WildDocs.AUTOCLICKONMOUSEOVER && aEvent instanceof MouseEvent) {
243          // pack event and type to an array and send it for adaptation
244          EventAndType eventAndType = new EventAndType(aEvent, aType);
245
246          // unpack the array and send the events
247          ArrayList adaptedEventsAndTypes = adaptInputEvent(eventAndType);
248          Iterator iterator = adaptedEventsAndTypes.iterator();
249          while ( iterator .hasNext()) {
250              EventAndType adapted = (EventAndType) iterator.next();
251              super.sendInputEventToInputManager(adapted.getEvent(), adapted
252                      .getType());
253          }
254      } else {

258          super.sendInputEventToInputManager(aEvent, aType);
259      }
260  }
```

The additional procedure is only used on mouse events. This is checked at line 242. All other events, such as keyboard, will be passed unmodified to PCanvas. If it is a mouse event and the WildDocs instance has switched on support for automatic focus change on mouse over, the event as well as the type is wrapped within a data structure that is supported by the inner class EventAndType. An additional ArrayList instance is instantiated at line 247 with the return value of adaptInputEvent(EventAndType), which decides whether additional events need to be inserted and creates them if necessary, as explained further below. Then, an iterator iterates through the array list, sending events and types to PCanvas, including newly added ones.

The method adaptInputEvent(EventAndType) checks if modifications are necessary and returns the appropriate events. Firstly, it creates an array list that is used to store all mouse events that will be returned:

[WDCanvas.java | adaptInputEvent(EventAndType)]

```
122  protected ArrayList adaptInputEvent(EventAndType aEventAndType) {
123      ArrayList eventList = new ArrayList();
```

A check is performed at line 128 to follow only mouse events. All mouse events and their types are extracted from the passed data structure at line 131 or 138.[28]

[WDCanvas.java | adaptInputEvent(EventAndType)]

```
128      if (aEventAndType.getEvent() instanceof MouseEvent) {
129
130          // unpack the information
131          MouseEvent event = (MouseEvent) aEventAndType.getEvent();
```

[28] The method event.getID() is equivalent to aEventAndType.getEvent().

```
138             int type = event.getID();
```

Another check makes sure that only mouse drag events that have the first mouse button pressed go further. The click count of two is also not allowed, because it is assigned to purposeful rotation.

[WDCanvas.java | adaptInputEvent(EventAndType)]

```
152             if (type == MouseEvent.MOUSE_DRAGGED && event.getClickCount() != 2
153                     && event.getButton() == MouseEvent.BUTTON1) {
```

The associated instance of PInputManager is referenced to find the active node as well as the current node underneath the cursor:

[WDCanvas.java | adaptInputEvent(EventAndType)]

```
165             PInputManager defaultManager = getRoot()
166                     .getDefaultInputManager();
```

```
174             PNode activeNode = defaultManager.getMouseFocus()
175                     .getPickedNode();
176
177             PNode nodeUnderCursor = defaultManager.getMouseOver()
178                     .getPickedNode();
```

The following part checks three conditions: Firstly, is the node below the cursor not the active node anymore? This happens for example if the node is dragged underneath another one in a way that the cursor is moved on top of the above node, as depicted in Fig. 4.16 (state 2). The next two conditions attempt to solve the problem that the mouse movement may be faster than the following dragged node. This delay is a Java issue. Line 189 checks if the active node is below the one directly underneath the cursor. Line 191 makes sure that it is not an instance of PCamera. The camera represents a view on layers. It may get underneath the cursor when the cursor leaves the node's bounds by moving faster than the dragged node:

[WDCanvas.java | adaptInputEvent(EventAndType)]

```
188             if (!activeNode.equals(nodeUnderCursor)
189                     && new WDIndexComparator().compare(activeNode,
190                             nodeUnderCursor) == -1
191                     && !nodeUnderCursor.getClass().equals(PCamera.class)) {
192                 MouseEvent e = (MouseEvent) event;
```

The issue of latency between mouse pointer and dragged node is also a problem when the node is dragged quickly underneath another node. It may be possible that there is no mouse event produced while the mouse was above the other node. It is also possible that the additional mouse release and press were created (discussed on the facing page), but it is performed after the mouse and the dragged node moved away from the above positioned node. These scenarios depend among others on the speed of the machine running WildDocs as well as the speed of dragging the node. One possible solution is to ensure synchronized dragging.

After those conditions are met, two additional mouse events are created. The first event represents a mouse release (line 197), the second one a mouse press (line 203):

[WDCanvas.java | adaptInputEvent(EventAndType)]

```
196             // Create a mouse event for mouse button release
197             MouseEvent eventMouseRelease = new MouseEvent(e
198                     .getComponent(), MouseEvent.MOUSE_RELEASED, e
```

```
199                         .getWhen(), e.getModifiers(), e.getX(), e.getY(), e
200                         .getClickCount(), e.isPopupTrigger(), e.getButton());
201
202                 // Create a mouse event for mouse button press
203                 MouseEvent eventMousePress = new MouseEvent(e
204                         .getComponent(), MouseEvent.MOUSE_PRESSED, e
205                         .getWhen(), e.getModifiers(), e.getX(), e.getY(), e
206                         .getClickCount(), e.isPopupTrigger(), e.getButton());
```

Even though the user does not release the mouse button, a mouse release and mouse press will be scheduled and performed later. This will change the focus to the node directly underneath the cursor, when executed. Finally, the original event as well as the two new events are added to the array:

[WDCanvas.java | adaptInputEvent(EventAndType)]

```
208                 // Add the original event
209                 eventList.add(aEventAndType);
210
211                 // Add the created mouse release
212                 eventList.add(new EventAndType(eventMouseRelease,
213                         MouseEvent.MOUSE_RELEASED));
214
215                 // Add the created mouse press
216                 eventList.add(new EventAndType(eventMousePress,
217                         MouseEvent.MOUSE_PRESSED));
218                 }
219             }
220         }
```

The array list is still empty if one of the previous conditions of this method fails. In this case, the original event is added to the list. Finally, the array list is returned:

[WDCanvas.java | adaptInputEvent(EventAndType)]

```
226         if (eventList.size() == 0) {
227             eventList.add(aEventAndType);
228         }
229
230         return eventList;
```

4.4.3. Input Event Handlers

Menu and Keyboard Shortcuts

All keyboard shortcuts are supported by the main menu (WDMainMenu). The command key may differ among various operating systems. For example, the command key on Mac OS X (⌘), as depicted for all shortcuts in Fig. 4.17, is equivalent to `CTRL` on Linux or Windows. Some originally assigned shortcut keys, for example, `CTRL-O` for importing ("open") files as documents, were removed later. They were not used for the usability tests and we wanted to avoid participants hitting them accidentally.

WDMainMenu extends Java's class MenuBar and implements ActionListener. All menus are listed in Fig. 4.17, including marked differences among WildDocs versions. WildDocs

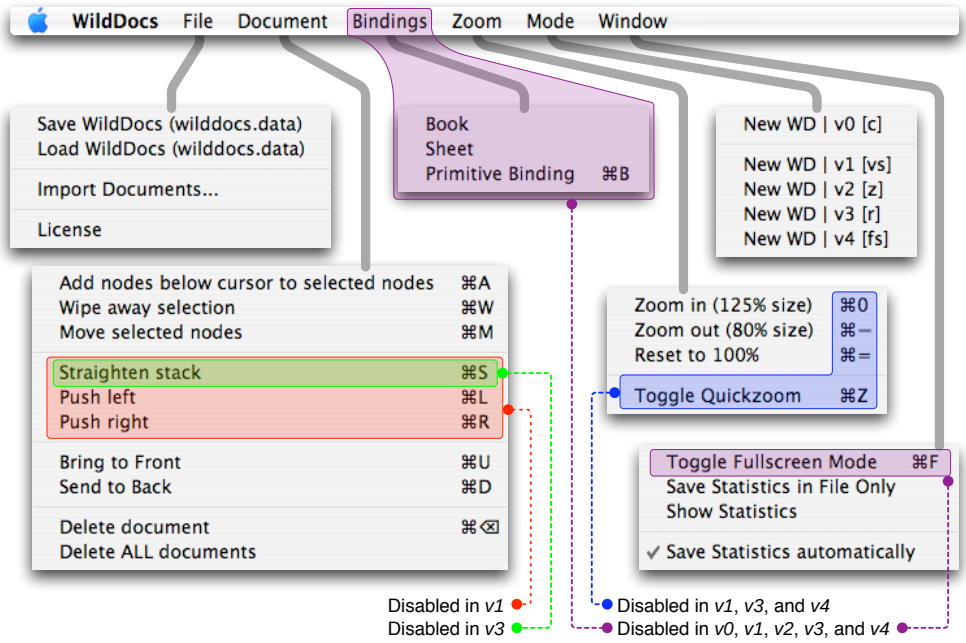


Figure 4.17.: WildDocs menus on Mac OS X

v3 has "Straighten stack" disabled, *v1* additionally also "Push left" and "Push right". Only WildDocs *v2* has "Toggle Quickzoom" enabled as well as shortcuts for zooming actions. The given file names for saving or loading, the factors for zooming in, out, or reset, as well as the version names in the "Mode" menu are statically set in WildDocs. The complete "Bindings" menu and "Toggle Fullscreen Mode" were disabled for all WildDocs versions due to the fact that they were not used for the usability test.

The method actionPerformed(ActionEvent) takes care that any accepted event in WildDocs and calls the appropriate method. All methods that are called from there are part of the class WildDocs. For counted actions, also the counter incrementation method is called, for instance:

[WDMainMenu.java | actionPerformed(ActionEvent)]

```
376         } else if (cmd.equals(STRAIGHTENSTACK)) {
377             getWildDocs().straightenStack();
378             getWildDocs().increaseStatStraightenStack(1);
379         } else if (cmd.equals(SELECTNODESBELOWCURSOR)) {
```

File Menu The "File" menu has four entries. The first two commands, "Save WildDocs (wilddocs.data)" and "Load WildDocs (wildddocs.data)", save or load the current object store. The file name is shown as part of their menu entries, currently "wilddocs.data". These actions are not completely functional yet. "Import Documents…" opens a file dialog window that lets the user select one or more files to be imported. Files of the type GIF, JPEG, PNG, plain text, HTML, or RTF are supported.[29] Finally, the menu entry "License" creates a text

[29] Currently, WildDocs infers that the file type is represented by the file extension.

document with the software license on it and puts it onto the WildDocs space.

Document Menu Commands for moving or deleting documents are placed within the "Document" menu. The first three entries support selection and movement of a range of nodes. "Add nodes below cursor to selected nodes" can be used to span a selection rectangle over an area, from one node to another. "Wipe away selection" deletes the selection rectangle. Moving all selected nodes to the current cursor position can be triggered by the menu entry "Move selected nodes" or its shortcut.

"Straighten stack" produces straight looking piles. "Push left" or "Push right" are used to move the node below the cursor to the left or to the right (including putting them it to the front afterwards). This can be used, for example, for browsing stacks.

Deleting the document below the cursor can be performed by "Delete document" or the associated shortcut key `CTRL-Backspace`. Removing all documents from the WildDocs space is initiated with the menu entry "Delete ALL documents".

Bindings Menu The entries in "Bindings" are easily extensible. WildDocs supports developers implementing new binding mechanisms. Currently, there are only the entries "Book" or "Sheet" for complex bindings as well as the entry "Primitive Binding" for primitive bindings. The activation of one of those triggers the creation of a binding of the selected type.

Zoom Menu The "Zoom" menu contains three entries for stepwise zooming: "Zoom in (125% size)", "Zoom out (80% size)", and "Reset to 100%". The displayed zoom values are taken from static variables in WildDocs. Additionally, there is an entry for quickzoom ("Toggle Quickzoom").

Mode Menu We created the "Mode" menu to easily switch between different WildDocs versions. The name of the version is part of the menu entries and appears also at the WildDocs window title. "WD" stands for "WildDocs", followed by a vertical line and the actual version name. WildDocs *v0* is a version that was not used for the usability tests. The letters in square brackets represents the main feature of the instance and therefore help memorizing the version type once it is known.[30] The letters stand for "complete feature set" [c], "variable size support" [vs], "extended zooming support" [z], "rotation and sloppiness support" [r], or "fixed size support" [fs].

A version with most features of *v1*, *v2*, *v3*, and *v4* is instantiated through the menu entry "New WD | v0 [c]". It is the default version at startup. The four used versions for the user test are created by the menu entries "New WD | v1 [c]", "New WD | v2 [vs]", "New WD | v3 [z]", or "New WD | v4 [fs]".

Window Menu The "Window" menu contains entries related to the WildDocs window or statistics. The entry "Toggle Fullscreen Mode" toggles between having the WildDocs space inside a window and full screen use. The window look and feel uses the default on the running computer.

Statistics support was implemented for usability evaluations. The entry "Save Statistics in File Only" allows to save the current statistics into a file, whereas "Show Statistics" saves

[30] It did not support the test participants in understanding the main features before or during the testing.

them and additionally creates an equivalent text document on the WildDocs space. To avoid missing statistics (e. g., if the administrator forgets to save them manually) we implemented code that saves statistics automatically on application exit or version change via “Mode” menu. This behavior is set for each newly created WildDocs instance. It can be switched off at any time via the menu entry “Save Statistics automatically”. When switched off, the mark in front of the menu entry will disappear.

Open Issues We did not change anything in look and feel after the user testing has started in order to provide the same application for every participant within the same group. Some issues concern the design of the menu entries. For example, there are violations of capitalization. Most menu entries are correctly capitalized, except some entries in “Document”, “Zoom”, or “Window”. This is caused by different development phases and certain preferences at those times. Another issue are menu entries for statistics. Semantically, they belong to the “File” menu rather than to “Window”.

Some menu entries are practicably only available by shortcut, because they require the mouse cursor to point to a node. For example, `CTRL-Backspace` deletes the node below the cursor, as discussed on the previous page. Because the mouse is used to point to the node, it cannot be used to activate the appropriate menu. Other examples include `CTRL-L`, `CTRL-R`, or `CTRL-S`, among others. However, for all relevant cases, there exists a keyboard shortcut.

Node Events

Instances of the class WDNodeInputEventHandler handle interactions with nodes. Important parts were already discussed in Sect. 4.3 with respect to the invocation of machines. Some remaining aspects will be explained in this section.

WDNodeInputEventHandler extends Piccolo’s class PBasicInputEventHandler, which provides basic support for input events on nodes. The constructor of WildDocs’s node input event handler associates the used WildDocs instance. A new node rotator (WDNodeRotator) and index pusher (WDNodeIndexPusher) are instantiated and associated on mouse click:

[WDNodeInputEventHandler.java | helperInitialization(PInputEvent)]

```
91   public void helperInitialization (PInputEvent aEvent) {

101      setNodeRotator(new WDNodeRotator(aEvent));
102      setNodeIndexPusher(new WDNodeIndexPusher(aEvent, getWildDocs()));
103  }
```

Mouse Pressed Aside of initiating some machines, mousePressed(PInputEvent) hides the shadow of the activated WDLowLevelDoc instance:

[WDNodeInputEventHandler.java | mousePressed(PInputEvent)]

```
166      if (aEvent.getPickedNode() instanceof WDLowLevelDoc) {
167          ((WDLowLevelDoc) aEvent.getPickedNode()).setShadowVisible(false);
168      }
```

The first mouse press event initiates some random offset and random rotation, simulating a similar effect as a finger touching a paper on a desk. At the second mouse press event, purposeful rotation is called, as discussed in Sect. 4.3.1, and the purposeful statistic counter

increased. The following code sets also the variable that represents the state of activated purposeful rotation. This is necessary for correctly handling mouse drag events:

[WDNodeInputEventHandler.java | mousePressed(PInputEvent)]

```
173         switch (aEvent.getClickCount()) {
174         case 2:
175             setRotationOnPurpose(true);
176             getNodeRotator().paintRotationMiddleMark();

183             if (WildDocs.PURPOSEFUL_ROTATION) {
184                 getWildDocs().increaseStatPurposefulRotation(1);
185             }
```

Mouse Dragged Mouse drag events handle several different behavior, depending on the mode or set preferences. After setting the handled state of the event, the above described purposeful rotation mode is checked. If a purposeful rotation is initiated, dragging will result in rotating the active node:

[WDNodeInputEventHandler.java | mouseDragged(PInputEvent)]

```
112     public void mouseDragged(PInputEvent aEvent) {
113         aEvent.setHandled(true);
114
115         if (isRotationOnPurpose() == true) {
116             // Rotate the node on purpose
117             getNodeRotator().rotateOnPurpose(aEvent);
```

Otherwise, the delta value of the mouse motion is calculated and used to move the node. At this point, another feature is implemented: WildDocs allows enabling of a grid to which the dragged node snaps. Even though working, this behavior is experimental. It has to be switched on at the WildDocs instance, which is checked at line 122:

[WDNodeInputEventHandler.java | mouseDragged(PInputEvent)]

```
118         } else {
119             PDimension delta = aEvent
120                     .getDeltaRelativeTo(aEvent.getPickedNode());
121
122             if (WildDocs.GRID) {
123                 PDimension account = getGridDragAccount();
124                 double deltaX = delta.getWidth() + account.getWidth();
125                 double deltaY = delta.getHeight() + account.getHeight();
126
127                 double gridspacing = new WDUnitConverter(getWildDocs())
128                         .wdToJava(WildDocs.GRIDSPACING);
129
130                 double modDeltaX = deltaX % gridspacing;
131                 double modDeltaY = deltaY % gridspacing;
132
133                 delta.setSize(deltaX − modDeltaX, deltaY − modDeltaY);
134
135                 setGridDragAccount(new PDimension(modDeltaX, modDeltaX));
136             }
137
```

```
138             // Move the node
139             aEvent.getPickedNode().translate(delta.getWidth(),
140                     delta.getHeight());
```

The heart of this behavior is a so-called "grid drag account", which stores the delta values of mouse movements that were discovered, but were not applied to the node's movement yet. On the other side, moving the node results in subtracting the moved dimension from the account. getGridDragAccount() returns the current grid account. When null, it returns a new instance of PDimension with zero values:

[WDNodeInputEventHandler.java | getGridDragAccount()]

```
448 public PDimension getGridDragAccount() {
449     if (gridDragAccount == null) {
450         setGridDragAccount(new PDimension(0, 0));
451     }
452     return gridDragAccount;
453 }
```

Line 130 and 131 calculate the current delta values modulo the grid spacing. The delta values are the sum of the latest mouse movement delta plus the grid drag account value, which are calculated at line 124 and 125. The mod delta values are subtracted from the delta values, packed into a newly created PDimension object, and associated as new grid account at line 135. The results of the subtractions at line 133 may be zero, the grid spacing value, or a multiple of the grid spacing value. Finally, the node is moved through translate(double,double). If the grid is switched off, the mouse movement delta value is passed unmodified to the translate method.

The mouse drag method also performs incidental rotation[31] calls, as discussed in Sect. 4.3.1, as well as calls for pushing the node to a higher index level, as discussed in Sect. 4.3.4. If switched on, both methods are triggered at each mouse drag event handled by WDNode|InputEventHandler:

[WDNodeInputEventHandler.java | mouseDragged(PInputEvent)]

```
142             // Rotation stuff
143             if (!WildDocs.RANDOM_ROTATION_ONLY_AT_MOUSE_RELEASE) {

149                 getNodeRotator().rotateByDirection(aEvent);
150                 getNodeRotator().rotateRandomly(aEvent);
151             }
152         }
153
154         // Take care that the node has the right index
155         getNodeIndexPusher().pushToHigherIndexLevel();
156     }
```

Mouse Released Mouse release events reset the purposeful rotation state and delete the rotation middle mark, if activated. If purposeful rotation was inactive, it triggers a random

[31] We experienced an unsolved bug in cases when direction-based rotation (line 149) and random rotation (line 150) are both switched on at the same time: After dragging the node for some time, the maximum rotation angle becomes too large. This problem did not affect the test, because the extended incidental rotation was replaced by a random rotation on mouse release, as further discussed in Sect. 4.3.1, but needs to be addressed in a future version.

rotation on the node. Finally, all shadows of WDLowLevelDoc instances are updated, as discussed in Sect. 4.2.2:

[WDNodeInputEventHandler.java | mouseReleased(PInputEvent)]

```
245     public void mouseReleased(PInputEvent aEvent) {
246         aEvent.setHandled(true);
247
248         if (isRotationOnPurpose() == true) {
249             getNodeRotator().deleteRotationMiddleMark();
250             setRotationOnPurpose(false);
251         } else {
254             if (WildDocs.RANDOM_ROTATION_ONLY_AT_MOUSE_RELEASE) {
255                 getNodeRotator().rotateRandomly(aEvent);
256             }
257         }
307         getWildDocs().updateShadows();
```

Line 259 to 305 contains experimental code for dragging nodes onto complex or primitive bindings. WildDocs currently ignores this part, because of problems in functionality.

Mouse Moved Each mouse move event sets a reference at the WildDocs instance to the document that has currently the mouse over:

[WDNodeInputEventHandler.java | mouseMoved(PInputEvent)]

```
348         PNode node = aEvent.getPickedNode();
349         if (node instanceof WDDocument) {
350             WDDocument doc = (WDDocument) node;
351             getWildDocs().setLastMouseOverOnDocument(doc);
352         }
```

This is used for pushing the document below the cursor to the foreground or to the background, for example, via `CTRL-U` or `CTRL-D`.

Mouse Entered or Exited In the current WildDocs version, methods that handle mouse enter or exit events on nodes do not add additional functionality. However, during development we had bounds handle appear when the mouse entered a node's bounds and were removed when the mouse left them. The original code still can be seen. mouseEntered(PInputEvent) triggered the creation of handles for WDLowLevelDoc instances:

[WDNodeInputEventHandler.java | mouseEntered(PInputEvent)]

```
318         // if (aEvent.getPickedNode() instanceof WDLowLevelDoc) {
319         // WDBoundsHandle.addBoundsHandlesTo(aEvent.getPickedNode());
320         // }
```

mouseExited(PInputEvent) called WDBoundsHandle to remove the bounds handle from the document:

[WDNodeInputEventHandler.java | mouseExited(PInputEvent)]

```
337         // WDBoundsHandle.removeBoundsHandlesFrom(aEvent.getPickedNode());
```

Both methods for adding or removing bounds handles were static. During the development process, removeBoundsHandlesFrom(PNode) was removed from WDBoundsHandle, because it was not longer in use.

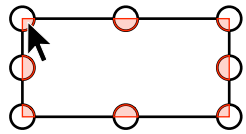


Figure 4.18.: Bounds handles with marked mouse over areas for resizing (outdated development version)

The reason why we removed this feature was that the bounds handles disappeared as soon as the mouse left the node's bounds. However, as Fig. 4.18 depicts, a handle's area is at least half outside the node's bounds. Tests have shown that it happens often that the mouse moves outside the node. Even though still inside the handle's area, all handles disappeared. It can be seen in Fig. 4.18 that the area on which the user needs to click to resize a node is small, 25 % for handles placed at edges and 50 % for the others. These areas are indicated at the figure with red transparent marks.

There are several solutions to this problem. One solution is to display all bounds handles at any time. This is what we have implemented. Another solution is to include the mouse handles' areas for adding or removing them on mouse enter or exit. This would mean that whenever the mouse is above the node *or* above any of its bounds handles, the bounds handle would be displayed. Still, this does not save the problem that the cursor may move slightly outside the bounds handle and has to be moved back to the *node* in order to bring it back again.

A third solution would be to implement a time counter that removes the handles after a certain time after the mouse moved outside the node's or the bounds handles' area. The forth possibility would be to increase the area around a node when the handles become activated on mouse over. Also magnetic behavior on handles that pull the mouse pointer onto a handle when it gets close may support the user in positioning the mouse.

Zoom Events

Mouse events that are intended for zooming are handled by WDZoomEventHandler, which extends Piccolo's class PZoomEventHandler. The method dragActivityFirstStep(PInputEvent) overrides a method in PZoomEventHandler. It is called when smooth zooming is initiated. This happens in WildDocs through pressing the right mouse button on the canvas or on a node that cannot become an active node through mouse events (pickable flag set to false), for example, a desk imitation (see Sect. 4.1.4) or rubber band (see discussion on the facing page). Beside sending the unmodified event to the superclass, it also increases the statistics counter for smooth zooming.

The second main method is dragActivityStep(PInputEvent), which also overrides a method in PZoomEventHandler. It handles incoming zoom events steps. If smooth zooming is disabled, nothing is processed there. Otherwise, the event is sent to the superclass and sets the next quickzoom command to full zoom out:

[WDZoomEventHandler.java | dragActivityStep(PInputEvent)]

```
92    protected void dragActivityStep(PInputEvent aEvent) {
93        if (WildDocs.SMOOTHZOOMING) {
94            super.dragActivityStep(aEvent);
```

```
100             if (getWildDocs() != null) {
101                 getWildDocs().setZoomFrameAtQuickZoomOut(WildDocs.FULLZOOMOUT);
102             }
103         }
```

Rubber Band

A rubber band in WildDocs is a transparent rectangle. It can be extended in size. All intersecting nodes are considered as "selected". This is used, for example, for moving several nodes at the same time.

There are two ways to create selections in WildDocs: one using the mouse; the other using the keyboard. Mostly, we call selection objects that are created or modified by mouse actions "rubber band", and the keyboard related ones "selection rectangle". Both terms have their origins in different phases of development, but have similar semantics.

Rubber bands or selection rectangles are instances of WDRubberBand. This class can be found in package de.atzenbeck.wilddocs.util.

Mouse Selection The class WDDeskInputEventHandler takes care of interactions with the desk. It extends Piccolo's PPanEventHandler. It allows the switching on or off of background panning as well as interaction with a selection rectangle (also called "rubber band") with the mouse. The mouse enabled rubber band is experimental and was switched off during the usability tests.

A rubber band is created by WDDeskInputEventHandler on mouse press, that is, when the left mouse button is pressed on the desk:

[WDDeskInputEventHandler.java | mousePressed(PInputEvent)]

```
117     public void mousePressed(PInputEvent aEvent) {
118         // Handled needs to be false, otherwise zooming does not work.
119         aEvent.setHandled(false);
120 
121         if (WildDocs.MOUSERUBBERBANDSELECTION && getRubberBandStatus() != |↵
                ↳ |RUBBERBAND_FINISHED
122                 && aEvent.isShiftDown()) {
123             aEvent.setHandled(true);
124             Point2D pos = aEvent.getPosition();
125             setRubberBandStatus(RUBBERBAND_DRAW);
126             setRubberBand(new WDRubberBand(getWildDocs(), pos));
127         } else {
128             if (WildDocs.PANNING)
129                 super.mousePressed(aEvent);
130         }
131     }
```

The WildDocs instance has a static switch that allows activation or deactivation of selections with the rubber band. This is checked at line 121. The user needs to have the shift key pressed for creating a new rubber band (line 122).

The method getRubberBandStatus() at line 121 returns the current status of the rubber band. The status can be inactive, drawing, or finished. A new rubber band is only created, if the rubber band does not have the status "finished".

If any of those conditions are not true, the mouse press event is passed to the superclass, which initiates panning the background, if WildDocs allows panning (see line 128). Otherwise, if all required conditions are true, the rubber band status is set to "draw" and a rubber band is created, starting at the position where the cursor is located.

WDRubberBand constructs a new rubber band with default values if not given otherwise, such as color, stroke type, or stroke color. The central part of the constructor calls the rubber band creation method:

[WDRubberBand.java | WDRubberBand(WildDocs,double,double,double,double,Color,Stroke,Color)]

```
100         createRubberObject(x, y, width, height, aColor, aStroke, aStrokeColor);
```

This creates a rectangle with the given visual attributes, position and size, sets transparency, and adds it to the WildDocs space. Rubber bands do not accept any mouse interactions directly as other nodes do. While the rubber band status is set to drawing mode, mouse|↩ ↪|Dragged(PInputEvent) updates the rubber band with the current cursor coordinates:

[WDDeskInputEventHandler.java | mouseDragged(PInputEvent)]

```
136     public void mouseDragged(PInputEvent aEvent) {
137         aEvent.setHandled(false);
138 
139         if (WildDocs.MOUSERUBBERBANDSELECTION && getRubberBandStatus() == |↩
                ↪|RUBBERBAND_DRAW) {
140             aEvent.setHandled(true);
141             getRubberBand().update(aEvent.getPosition());
142         } else {
143             if (WildDocs.PANNING)
144                 super.mouseDragged(aEvent);
145         }
146     }
```

The method update(Point2D) extends the opposite corner of the start position (saved at construction time) to where the cursor is located. The rubber band's position and size becomes fixed as soon as the mouse is released:

[WDDeskInputEventHandler.java | mouseReleased(PInputEvent)]

```
151     public void mouseReleased(PInputEvent aEvent) {
152         aEvent.setHandled(false);
153 
154         if (WildDocs.MOUSERUBBERBANDSELECTION && getRubberBandStatus() == |↩
                ↪|RUBBERBAND_DRAW) {
155             aEvent.setHandled(true);
156             setRubberBandStatus(RUBBERBAND_FINISHED);
157         } else {
158             if (WildDocs.PANNING)
159                 super.mouseDragged(aEvent);
160         }
161     }
```

For moving all nodes that intersect with the rubber band, the user has to click on the background without the shift key pressed. A new instance of WDNodeDragger will drag all intersecting nodes to where the click was performed (line 174):

[WDDeskInputEventHandler.java | mouseClicked(PInputEvent)]

```
166     public void mouseClicked(PInputEvent aEvent) {
```

```
167         aEvent.setHandled(false);
168
169         if (WildDocs.MOUSERUBBERBANDSELECTION && getRubberBandStatus() == |
                ↪ |RUBBERBAND_FINISHED) {
170             aEvent.setHandled(true);
171
172             if (!aEvent.isShiftDown()) {
173                 WDTempNodeStorage selected = getRubberBand().intersectDocs();
174                 new WDNodeDragger(getWildDocs()).dragCenterToPosition(selected,
175                         aEvent.getPosition());
176             }
177
178             getRubberBand().removeFromWildDocs();
179             setRubberBandStatus(RUBBERBAND_INACTIVE);
180         } else {
181             if (WildDocs.PANNING)
182                 super.mouseClicked(aEvent);
183         }
184     }
```

If the user wants to remove the selection rectangle without moving intersecting nodes, he/she can click on the background while having the shift key pressed. The if-clause at line 172 will then skip moving nodes. Finally, the rubber band will be removed (line 178) and the rubber band status is set to "inactive" again (line 179), allowing the creation of a new one on mouse press with pressed shift key.

Keyboard Selection Keyboard-based selection rectangle interactions are triggered by the class WDMainMenu, if enabled by the preference settings in WildDocs. It sets up three menu entries for creating or expanding the selection rectangle (`CTRL-A`), removing it (`CTRL-W`), or moving the selected nodes (`CTRL-M`).

[WDMainMenu.java | WDMainMenu(WildDocs)]

```
207         if (WildDocs.KEYRUBBERBANDSELECTION) {
208             addMenuItem(m, SELECTNODESBELOWCURSOR, KeyEvent.VK_A);

211             addMenuItem(m, REMOVESELECTION, KeyEvent.VK_W);
212             addMenuItem(m, MOVESELECTEDNODES, KeyEvent.VK_M);

214         }
```

The command `CTRL-A` calls the method addNodesBelowCursorToSelection() in WildDocs. Firstly, it checks if there is already an existing rubber band:

[WildDocs.java | addNodesBelowCursorToSelection()]

```
1645        if (getRubberBand() == NO_RUBBERBAND) {
1646            setRubberBand(new WDRubberBand(this, currentMousePositionOnCamera()));
1647        } else {
```

If no rubber band exists, a new one is created at the current mouse pointer position. If one exists already, the node that is directly below the cursor is passed for updating the selection rectangle:

[WildDocs.java | addNodesBelowCursorToSelection()]

```
1658        boolean result = getRubberBand().update(
```

```
1659                    nodesBelowCursorOnLayer().getHighestIndexNode());
1660
1661            if (result == false) {
1662                System.err
1663                        .println("WARNING: RubberBand was not modified, "
1664                                + "because no node was handed over that was to be ↵
                                    ↳ spanned."
1665                                + "The problem is probably caused by "
1666                                + IntersectionFilter.class.toString());
1667            }
1668        }
```

The method update(PNode) in class WDRubberBand wraps the given node into a collection object and passes it to update(Collection), which iterates through the handed over objects and increases the span to cover the full bounds of all PNode instances. If there is a change in size of the span rectangle, its graphical representation will be animated to the new size:

[WDRubberBand.java | update(Collection)]

```
188    public boolean update(Collection aCollectionOfNodes) {
189        boolean returnValue = false;
190
191        PBounds span = new PBounds();
192        span.add(this.getFullBounds());
193
194        Iterator iterator = aCollectionOfNodes.iterator();
195        while (iterator.hasNext()) {
196            Object o = iterator.next();
197            if (o instanceof PNode) {
198                PNode node = (PNode) o;
199                span.add(node.getFullBounds());
200
201                // If there is at least one PNode, the return value is true
202                returnValue = true;
203            }
204        }
205
206        if (returnValue == true) {
207            PActivity resizement = animateToBounds(span.getX(), span.getY(),
208                    span.getWidth(), span.getHeight(), DEFAULTDURATION);
209
210            getRoot().addActivity(resizement);
211            resizement.setStartTime(System.currentTimeMillis());
212        }
213
214        getParent().addChild(this);
215
216        return returnValue;
217    }
```

Removing Selection As an alternative to the menu entry for removing the selection rectangle, CTRL−W, can be used. Both call removeSelection() at the WildDocs instance, which

firstly checks if a rubber band exists:

[WildDocs.java | removeSelection()]

```
1674    public void removeSelection() {
1675        if (getRubberBand() != NO_RUBBERBAND) {
1676            getRubberBand().removeFromWildDocs();
1677        }
1678    }
```

An existing rubber band will be removed from the WDRubberBand instance by calling removeFromWildDocs(), which finally triggers removeFromWildDocs(long), passing the default duration for fading out.

One problem occurred by writing the code for fading out and removing the selection rectangle from the WildDocs space. Because fading takes some time, the deletion was performed before the fading finished. The user did not see any fading.

One possible way to solve this problem would be Java threads. However, Piccolo is not thread safe. We used Piccolo's nested class PActivity.PActivityDelegate instead. It allows the setting of specific behavior that is called during the activity:

[WDRubberBand.java | removeFromWildDocs(long)]

```
241    public void removeFromWildDocs(long aFadeOutDuration) {

250        PActivity activity = animateToTransparency(0f, aFadeOutDuration);
251        PActivity.PActivityDelegate delegate = new PActivity.PActivityDelegate() {
252            public void activityFinished( PActivity a) {
253                getWildDocs().getLayer().removeChild(getThisRubberBand());
254            }
255
256            public void activityStarted( PActivity a) {
257            }
258
259            public void activityStepped( PActivity a) {
260            }
261        };
262        activity.setDelegate(delegate);
263
264    }
```

The actual fading out is triggered at line 250. Line 262 adds the information to remove it after it is finished. Without removing, it would be completely transparent and therefore invisible, but still existing.

Moving Selected Nodes Finally, moving all nodes that intersect with the selection rectangle can be initiated by a menu entry or preferably by CTRL−M. Both interactions call moveSelectionToCurrentPosition() at the associated WildDocs instance:

[WildDocs.java | moveSelectionToCurrentPosition()]

```
1683    public void moveSelectionToCurrentPosition() {
1684        if (getRubberBand() != NO_RUBBERBAND) {
1685            WDTempNodeStorage selected = getRubberBand().intersectDocs();

1692            getLayer().addChildren(selected);
1693
```

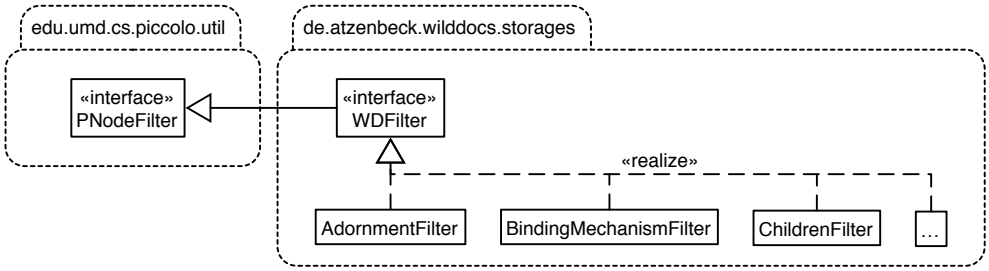


Figure 4.19.: Filters class diagram (package filters)

```
1694                new WDNodeDragger(this).dragCenterToPosition(selected,
1695                        currentMousePositionOnCamera());
1696                getRubberBand().removeFromWildDocs();
1697            }
1698        }
```

If the check for the existence of the rubber band at line 1684 turns out positive, an array of all selected nodes is returned by the WDRubberBand instance. This is done by adding all nodes and filtering those that are documents and intersect with the rubber band:

[WDRubberBand.java | intersectDocs()]

```
271     public WDTempNodeStorage intersectDocs() {
272         WDTempNodeStorage intersect = new WDTempNodeStorage();
273         intersect.addAll(getWildDocs().getLayer().getChildrenReference());
274         intersect.keepFiltered(new DocumentFilter());
275         intersect.keepFiltered(new IntersectionFilter(this));
276         return intersect;
277     }
```

In order to have all selected nodes on top, they are added again to the layer (line 1692). Through that, their index is higher than the remaining nodes and they appear above. Then, a new instance of WDNodeDragger drags all selected nodes to the current mouse position (line 1694). Finally, the rubber band calls the fading out action and is removed from the space.

4.5. Miscellaneous

4.5.1. Filters

WildDocs makes extensive use of filters for isolating groups of nodes. All filters are part of the package de.atzenbeck.wilddocs.filters. A complete list of WildDocs filter classes can be found in Tab. 4.1 on page 80. They implement the interface WDFilter, which extends PNodeFilter. The latter one is provided by Piccolo. Figure 4.19 shows the class relationships.

WDFilter classes implement two methods: accept(PNode) and acceptChildrenOf(PNode). Both return boolean values. The first one returns true if a node is accepted by the filter. The second method returns true if the filter should also test a node's children for acceptance.

Filters are widely used, for instance, by WDClusterRecognizer. It uses AdornmentFilter, DescendentFilter, IntersectionFilter, and LargerNodeIndexFilter. Many filters require an additional attribute that is used for the comparison.

Paradigmatically, we explain how NodeInBetween works. It filters all nodes that are in between two given nodes. The constructor takes both nodes and associates the top and bottom one according to their index. An instance of WDIndexComparator does the calculation:

[NodeInBetween.java | NodeInBetween(PNode,PNode)]

```
53  public NodeInBetween(PNode aNode1, PNode aNode2) {
54      if (new WDIndexComparator().compare(aNode1, aNode2) == 1) {
55          setTopNode(aNode1);
56          setBottomNode(aNode2);
57      } else {
58          setTopNode(aNode2);
59          setBottomNode(aNode1);
60      }
61  }
```

The method accept(PNode) will return true if the instance of WDIndexComparator returns the information that the given node is below the top node and above the bottom node. Otherwise, false will be returned:

[NodeInBetween.java | accept(PNode)]

```
88  public boolean accept(PNode aNode) {
89      WDIndexComparator comparator = new WDIndexComparator();
90      if (comparator.compare(aNode, getTopNode()) == -1
91              && comparator.compare(aNode, getBottomNode()) == 1) {
92          return true;
93      } else {
94          return false;
95      }
96  }
```

The filter always returns true for acceptChildrenOf(PNode) to force all nodes to be checked. However, this is not necessary, because a node's children return the same result as the node itself. Therefore, false should be returned instead.

4.5.2. Index Comparison

The package de.atzenbeck.wilddocs.comparators contains classes used for comparisons. Currently, only WDIndexComparator exists. It implements Java's interface Comparator. This class compares two nodes with respect to their index sequence.

Each Piccolo node has a unique index ID among siblings of the same parent. Piccolo draws nodes with higher index above those with lower numbers. The class WDIndexComparator creates a path containing the indices of all ancestors of a given node. Those paths can be compared to find the correct relation among two nodes.

The method index(PNode) returns the inverse path sequence of a given node.

[WDIndexComparator.java | index(PNode)]

```
66  public WDTempNodeStorage index(PNode aNode) {
67      WDTempNodeStorage reverseIndices = new WDTempNodeStorage();
68      PNode node = aNode;
```

```
69          while (!node.equals(aNode.getRoot())) {
70              reverseIndices
71                      .add(new Integer(node.getParent().indexOfChild(node)));
72              node = node.getParent();
73          }
74
75          /*
76           * Return the reversed sequence, so that the node comes last, after all
77           * its parents.
78           */
79          return reverseIndices.reverseSequence();
80      }
```

An instance of the class WDTempNodeStorage is created at line 67. The index of the passed node and the indices of its ancestors will be added to the temporary storage. Finally, the reversed sequence is returned.

The interface Comparator requests the method compare(Object,Object):

[WDIndexComparator.java | compare(Object,Object)]

```
127     public int compare(Object o1, Object o2) {
128         return compareNodes((PNode) o1, (PNode) o2);
129     }
```

Assuming that it receives only PNode instances, it casts the objects and passes them to compareNodes(PNode,PNode):[32]

[WDIndexComparator.java | compareNodes(PNode,PNode)]

```
90      private int compareNodes(PNode aNode1, PNode aNode2) {
91          WDTempNodeStorage path1 = index(aNode1);
92          WDTempNodeStorage path2 = index(aNode2);
93
94          /*
95           * Add delimiter -1. This takes care if one path is shorter than the
96           * other and delivers the correct return value.
97           */
98          path1.add(new Integer(-1));
99          path2.add(new Integer(-1));
100
101         if (path1.equals(path2)) {
102             return 0;
103         } else {
104             int i = 0;
105             int idxPath1;
106             int idxPath2;
107             /*
108              * This loop should terminate in any case, since path1.equals(path2)
109              * has been checked already. The delimiter -1 will take care in case
110              * one path is shorter than the other.
111              */
112             do {
113                 idxPath1 = ((Integer) path1.get(i)).intValue();
```

[32] There should be an exception handling implemented for cases when the passed object is not an instance of PNode.

Node	Index path (including delimiter)					
	0	1	2	3	4	5
A	1	0	0	3	8	−1
B	1	0	0	1	−1	

Table 4.5.: Example of index paths

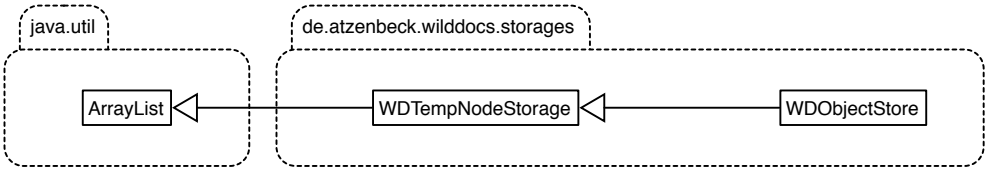


Figure 4.20.: Storage classes diagram (package storages)

```
114                 idxPath2 = ((Integer) path2.get(i)).intValue();
115                 i++;
116             } while (idxPath1 == idxPath2);
117
118             return (idxPath1 < idxPath2) ? -1 : 1;
119         }
120     }
```

Firstly, the method requests the index paths of both given nodes and adds −1 as delimiter to them at the very end. If both paths are equal, zero will be returned; otherwise, the method loops through index by index and compares them to each other. As soon as there are two indices that differ, it leaves the loop and returns either −1 or 1, depending on whether the index of the first or second path is smaller.

Table 4.5 shows an example. Two nodes, *A* and *B*, depict their index path as delivered by index(PNode). The node's index stands right before the delimiter −1, which is index 8 for node *A* and index 1 for node *B*. The comparison starts at position 0, where both indices are equal. The loop proceeds until position 3, where *A* has 3 and *B* has 1 at its index path. Assuming that node *A* was the first passed parameter, the return value will be 1.

4.5.3. Storage

The package de.atzenbeck.wilddocs.storages includes classes for temporarily or persistently storing documents. Figure 4.20 depicts the relationships of WDTempNodeStorage and WD| ⤸ ⤷ |ObjectStore, discussed below.

Temporal Object Store

WildDocs uses WDTempNodeStorage for temporary storage, intended for nodes. It extends Java's class ArrayList. Its constructor calls the superclass:

[WDTempNodeStorage.java | WDTempNodeStorage()]

```
64     public WDTempNodeStorage() {
```

```
65          super();
66      }
```

A given collection is passed to ArrayList:

[WDTempNodeStorage.java | WDTempNodeStorage(Collection)]

```
68      public WDTempNodeStorage(Collection aCollection) {
69          super(aCollection);
70      }
```

In the following, we describe the most important extensions of WDTempNodeStorage: The method sortNodesOnIndex() passes the node storage instance including a newly created instance of WDIndexComparator to sort(List,Comparator) in Java's Collections class (package java.util).

[WDTempNodeStorage.java | sortNodesOnIndex()]

```
110     public WDTempNodeStorage sortNodesOnIndex() {
111         Collections.sort(this, new WDIndexComparator());
112         return this;
113     }
```

Two methods return the node with the lowest or highest index of those stored within the temporary storage. For the node with the lowest index, a clone of the instance is sorted and the first entry returned:

[WDTempNodeStorage.java | getLowestIndexNode()]

```
79      public PNode getLowestIndexNode() {
80          WDTempNodeStorage sortedList = ((WDTempNodeStorage) this.clone())
81                  .sortNodesOnIndex();
82          if (sortedList.isEmpty()) {
83              return NO_NODE;
84          } else {
85              return (PNode) sortedList.get(0);
86          }
87      }
```

The method for finding the node with the highest index looks similar, except for line 85. Instead of the first entry, the last one is returned:

[WDTempNodeStorage.java | getHighestIndexNode()]

```
100             return (PNode) sortedList.get(sortedList.size() - 1);
```

reverseSequence() reverses the current order of references. Two other important methods are keepFiltered(WDFilter) and removeFiltered(WDFilter). The first keeps only those references that are accepted by a given filter. The second one removes the accepted object references from the WDTempStorage instance. Both call the method processWithFilter(WDFilter,boolean) with the given filter and a switch as parameter. The parameter is either KEEP (set to true) or DELETE (set to false):

[WDTempNodeStorage.java | processWithFilter(WDFilter,boolean)]

```
165     private WDTempNodeStorage processWithFilter(WDFilter aFilter,
166             boolean aInverseSwitch) {
167         WDTempNodeStorage deletedObjects = new WDTempNodeStorage();
168         if (aFilter == null) {
169             return null;
```

```
170         } else {
171             Iterator iterator = new HashSet(this).iterator();
172             while ( iterator .hasNext()) {
173                 PNode node = (PNode) iterator.next();
174 
175                 // false for remain, true for delete
176                 if ( aFilter .accept(node) == aInverseSwitch) {
177                     this.remove(node);
178                     deletedObjects.add(node);
179                 }
180 
181                 // false for remain, true for delete
182                 if ( aFilter .acceptChildrenOf(node) == aInverseSwitch) {
183                     ArrayList children = new ArrayList();
184                     children.addAll(node.getChildrenReference());
185                     this.removeAll(children);
186                     deletedObjects.addAll(children);
187                 }
188             }
189             return deletedObjects;
190         }
191     }
```

This method iterates through all stored object references and removes all nodes and their children that are accepted or that are not accepted by the given filter, depending on whether KEEP or DELETE was passed (see line 176 and 182). All deleted objects are returned.

Object Store

The package storages contains classes for storing WildDocs objects. The class ObjectStore was an early and basic implementation of a persistent object store. WDObjectStore is a newer version with special support for WildDocs; however, it is still in an early development state and has not reached its full functionality yet. It is intended for saving or loading objects. It extends WDTempNodeStorage. Its main methods are update(), saveObjects(String), and loadObjects(String).

update() takes all nodes and keeps only instances of WDDocument and WDBinding|↩↪|Mechanism. All other instances, for example, adornments (WDAdornment) are ignored, because they can be recalculated at any time later:

[WDObjectStore.java | update()]

```
84     public void update() {
85         // clear and get all nodes that are on layer
86         clear();
87         WDTempNodeStorage all = new WDTempNodeStorage(getWildDocs().getLayer()
88                 .getAllNodes());
89 
90         if (! all .isEmpty()) {
91             // filter all documents
92             WDTempNodeStorage docs = (WDTempNodeStorage) all.clone();
93             docs.keepFiltered(new DocumentFilter());
94 
```

```
95              // filter all binding mechanisms
96              WDTempNodeStorage mech = (WDTempNodeStorage) all.clone();
97              mech.keepFiltered(new BindingMechanismFilter());
98
99              // add documents and binding mechanisms
100             addAll(docs);
101             addAll(mech);
102
103             // sort sequence
104             sortNodesOnIndex();
105         }
106     }
```

saveObjects(String) saves all objects that are referenced by the WDObjectStore instance to a file. loadObjects(String) loads objects from a given file path into the object store instance. The file paths for saving or loading are represented as strings.

4.5.4. File Access

File Selection Dialog Window

When loading new documents to the WildDocs space, FileChooser takes care of the GUI supported selection of files. It is part of the package util. Multiple file selections are supported. The user can call the file dialog box by selecting the "Import Documents..." menu entry. The class contains only one method:

[FileChooser.java | getFiles(WildDocs)]

```
47  public static File [] getFiles(WildDocs aWildDoc) {
48      JFileChooser fc = new JFileChooser();
49      fc.setMultiSelectionEnabled(true);
50      fc.showDialog(aWildDoc, "Load");
51      return fc.getSelectedFiles();
52  }
```

Loading and Saving Text

The class WDTextLoader loads a given text file and returns its content as string. The file is represented as an instance of Java's class URI and handed over to the method load(URI). WDTextLoader is used, for example, by loading plain text documents onto the WildDocs space.

WDTextSaver saves a given string as plain text file. This class is used, for example, for saving the gathered statistics to an external file. The file name is accepted as an instance of URI or as string: save(URI,String) or save(String,String).

4.5.5. Boxes and Rotation Point (Obsolete)

For the sake of completeness, also the classes WDBox and WDRotationPoint shall be mentioned. Both can be found in package util. WDBox was designed as multi-purpose box, for example, for drawing graphical representations of binding mechanisms. It extends Piccolo's

class PPath and supports unit conversions by using WDUnitConverter. It is currently not in use.

The idea behind WDRotationPoint is to have a class for center marks including their graphical representation. They would appear on purposeful rotation. This class is not implemented yet. Instead, WDNodeRotator creates a Piccolo PPath instance and sets the desired attributes. The method paintRotationMiddleMark() is triggered at WDNodeInputEventHandler, as explained on page 157.

Chapter 5.

Experimental Design and Evaluation

"Enough research will tend to support your theory."

(Murphy's Law of Research)

5.1. Goals

The main goal of our experiment is to try to falsify our hypotheses (discussed in Chap. 3), following the philosophy of science theories of Karl Popper (Popper et al., 2001). We want to test the influence of variable sized documents, extended zooming, and incidental rotation for information organizing and finding on WildDocs, a 2D spatial application. Additionally, we want to evaluate some other observations, discussed in Sect. 5.3. Section 5.2 describes the test setup, Sect. 5.4 concludes both, the statistical evaluation as well as the descriptive observations. Statistics is supported by Field & Hole (2003), which we consulted mainly for finding appropriate tests and answers to other statistics related questions.

5.2. Method

5.2.1. Test Laboratory

We set up a laboratory for performing tests on WildDocs. Figure 5.1 shows a picture of the laboratory from the investigator's viewpoint when standing. The front row was for the participant, the tables behind for the investigator. We used four computers, three external 17-inch and two built-in screens, and two Digital Video (DV) cameras. The participants worked on a Pentium IV machine (3.4 GHzHT, 2 GB DDR2 RAM, ATI Radeon X600) running WildDocs on Java 1.5 on Windows XP. Additionally, a VNC server (RealVNC Ltd, 2005) and a screen capture application were started. The latter one produced screen capture movies of the full screen. There was a second screen for the participants that showed an introduction movie, task descriptions, or questions to be worked on. This screen was connected to the investigator's main computer, an Apple PowerBook G4 (15-inch, 1.67 GHz).

This portable computer was used to control the content of the introduction screen, such as starting an introduction video or displaying questions to the participant. It also captured the main part of this external screen as a movie. Furthermore, the PowerBook was used to time organization and search periods as well as to add notes, answers, or comments which participants gave to pre or post-test questions. Additionally, it saved the DV video stream from camera 1 to an external hard drive.

We used a second Apple PowerBook G4 (12-inch, 887 MHz) to store the DV video stream from camera 2. Another Pentium IV computer (2.5 GHz) on the investigator's desk served as

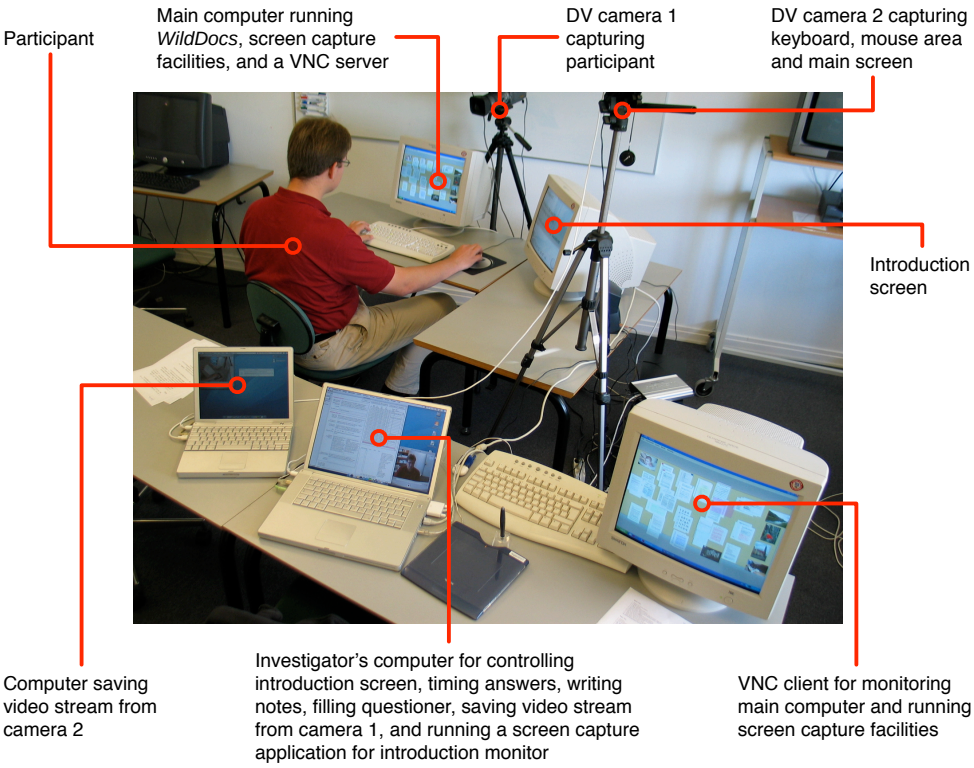


Figure 5.1.: Test laboratory setup

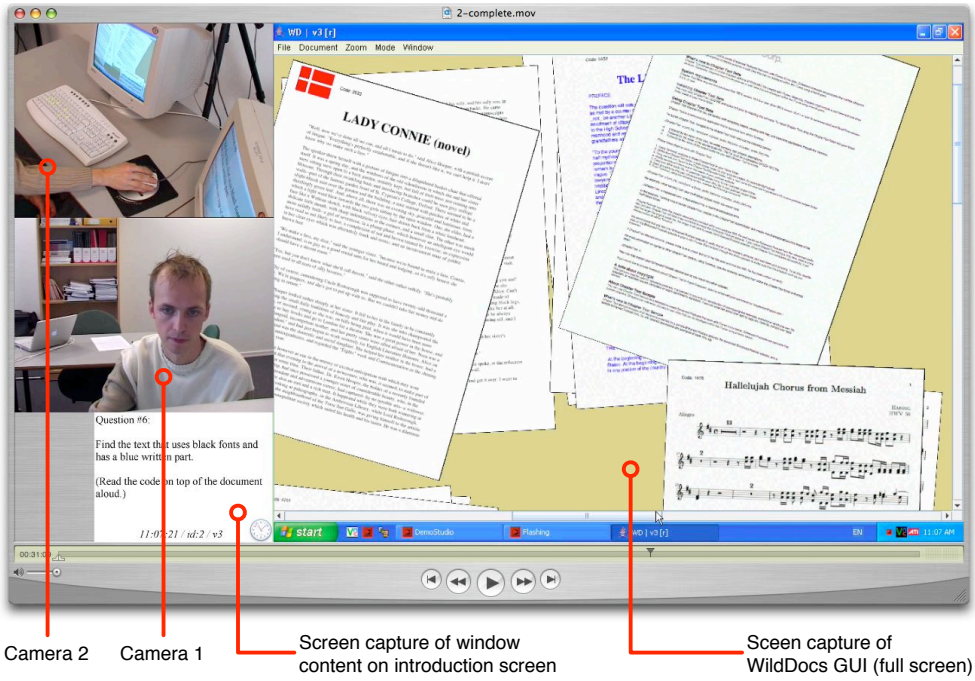


Figure 5.2.: Compilation of video material captured during a session

remote control for the participant's machine that ran WildDocs. A VNC client allowed the investigator a closer look at the participant's screen as well as easy setup tasks on WildDocs without leaving his workspace, for example, loading a new document set or changing the application version before the next participant came in. Because the screen capture on the main WildDocs machine was not reliable,[1] a screen capture movie of the VNC client window in full screen mode was recorded additionally.

As mentioned above, we installed two cameras, each of them sending the video stream to one of the PowerBooks to be stored on a hard drive directly. Camera 1 captured the participant's face, camera 2 the area where the participant's keyboard and mouse were located as well as the screen of the machine running WildDocs. The stream was stored in raw DV format and was compressed to H.264 and MPEG-4 AAC codec overnight or on weekends. For most sessions, we compiled completely or partly the gathered video material into a synchronized single movie file, as depicted in Fig. 5.2. This turned out to be very useful for post-analysis of video data.

5.2.2. Test Applications

There are four different WildDocs versions. Each of them differs slightly from the others. Table 5.1 gives an overview of the implemented features for each version. We will evaluate some of them in Sect. 5.3. There are three main groups of features addressing the focus of this

[1] Occasionally, the saved screen capture was damaged and could not be opened or repaired.

Feature	*v1*	*v2*	*v3*	*v4*
Zooming *(main feature)*				
– Menu zoom (stepwise zooming)	•	•	•	•
– Keyboard shortcut for menu zoom		•		
– Smooth zooming		•		
– Quickzoom		•		
Rotation and sloppiness *(main feature)*				
– Incidental rotation and sloppiness			•	
– Purposeful rotation			•	
Object size *(main feature)*				
– Resizable objects	•			
– Fixed size		•	•	•
– Desk metaphor		•	•	•
Object movement				
– Drag object	•	•	•	•
– Push object to left/right		•	•	•
– Push object to foreground/background	•	•	•	•
– Auto push object to foreground		•	•	•
– Move selected objects shortcut	•	•	•	•
Navigation				
– Drag background	•	•	•	•
– Scrollbars	•		•	•
Straighten stacks		•		•

Table 5.1.: WildDocs application features

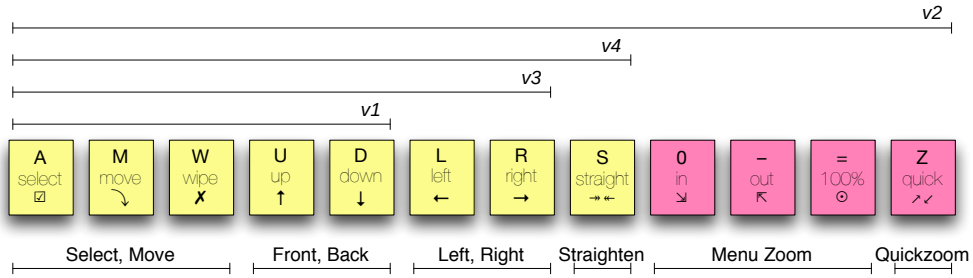


Figure 5.3.: Keyboard labels used for different WildDocs versions

work: zooming, rotation and sloppiness, and fixed versus variable size objects. All versions have log file support to report the used area dimensions or counts of selected feature usage, for example, quickzoom, smooth zooming, purposeful rotation, or clicks on bounds handles. A complete list of logged information can be found in Sect. B.2.

Version dependent shortcut keys support the user in working with WildDocs. All shortcut keys are to be pressed in combination with `CTRL`. Figure 5.3 depicts all the labels that were put on the physical keyboards. Every WildDocs version had its individual keyboard with all shortcuts marked. The intention was to help the user remembering the keys, since the learning phase was short. Figures 5.4 to 5.7 show pictures of the different keyboards. Even though all keyboards were Danish, the software-based keyboard mapping was US English. This only affected some keys for *v2*, whereas the keys for all other used shortcuts had identical positions on Danish and US English keyboard layouts.

WildDocs version designations consist of the letter "v", followed by a number between 1 and 4: *v1*, *v2*, *v3*, and *v4*. "v" is an abbreviation of "version", the number stands for the version number. Because our experiment had groups each assigned to one specific WildDocs version only, we named those groups also by the WildDocs version they have used.

WildDocs *v4*

WildDocs *v4* is the basic WildDocs application. Its main feature is support for fixed sized documents. There is a space on which fixed sized pages can be dragged using the mouse. It is not possible to resize those. There is a brown rectangle in the background that symbolizes a desk. However, the space outside the desk metaphor can also be used to place pages. We will statistically compare results of this version to the three other WildDocs versions.

Beside using the mouse to drag and drop single pages, there are keyboard shortcuts for moving objects. Those include pushing the node below the cursor to the very back (`CTRL-D` for "down") or to the very front (`CTRL-U` for "up") as well as pushing the node below the cursor to the left (`CTRL-L`) or to the right (`CTRL-R`). Figure 5.4 depicts those and others marked on the keyboard.

There is also behavior implemented that pushes a dragged page automatically to the foreground when it leaves the scope of the above positioned pages. Details of the relevant class WDNodeIndexPusher are discussed in Sect. 4.3.4. A selection feature (`CTRL-A`) selects all pages that intersect with the selection rectangle. Another shortcut (`CTRL-M`) moves them to the position where the cursor is located. The so moved documents will be aligned to a straight

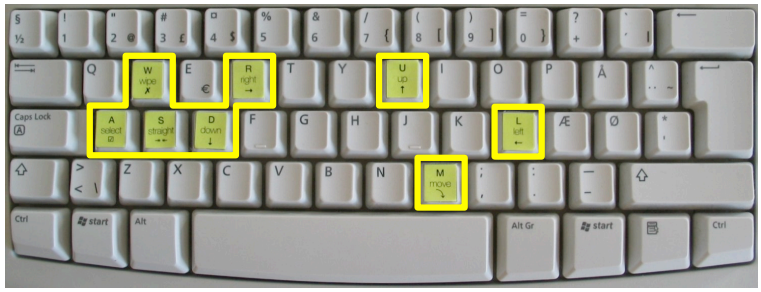


Figure 5.4.: Picture of WildDocs *v4* keyboard with marked shortcut keys (W, R, U, A, S, D, L, M)

looking pile. After CTRL–M is pressed, the selection rectangle will fade out. The shortcut CTRL–W removes a selection area without moving documents.

Navigation of the space is possible through dragging the background. Additionally, scrollbars are switched on when the occupied area does not fit completely into the visible area.

Another keyboard shortcut (CTRL–S) allows to straighten stacks. All documents that intersect with the cursor's position will be centered at that spot. This results in a straight looking pile. It looks exactly the same as if those pages would have been selected (CTRL–A) and moved (CTRL–M).

Like all other versions, *v4* allows stepwise zooming via the zoom menu. The user can decide to zoom in to 125 % or zoom out to 80 % relative to the current scale, or reset the zoom scale to 100 %. The zoom menu entries cannot be accessed via keyboard shortcuts.

WildDocs *v3*

WildDocs *v3* is similar to *v4*; however, it has incidental rotation and sloppiness as well as purposeful rotation implemented. Incidental rotation was originally implemented in a way that rotation was applied to the dragged node in real time while being dragged, depending on speed, direction, virtual position of the user, and a random factor, as described in Sect. 4.3.1. However, due to technical limitations, we reduced this behavior to random rotations on mouse press and release actions only.[2]

Purposeful rotation allows the user to rotate an object easily by double clicking on it and holding the mouse button at the second click. The rotation angle can now be changed by moving the mouse toward the desired direction. The document's angle will follow the mouse.

Another difference to *v4* is that *v3* does not support to straighten piles with CTRL–S. This would be in contradiction with the intended emerging sloppiness. The move shortcut (CTRL–M) for moving selected nodes works also for *v3*; however, realistic random offsets as well as random rotation angles will be applied to each moved page. Random offset as well

[2]We developed WildDocs in Java on Mac OS X. All tests on this machine were performed without problems. The final test took place on a computer running Windows XP. On this machine, incidental rotation frequently caused the window content to grey out. We experienced the same malfunction occasionally on Linux. Because this problem could not be solved within a reasonable amount of time, we decided to base incidental rotation on clicks (mouse release events). We consider this as a good alternative that also provides spatial structures that look similar to paper on a desk.

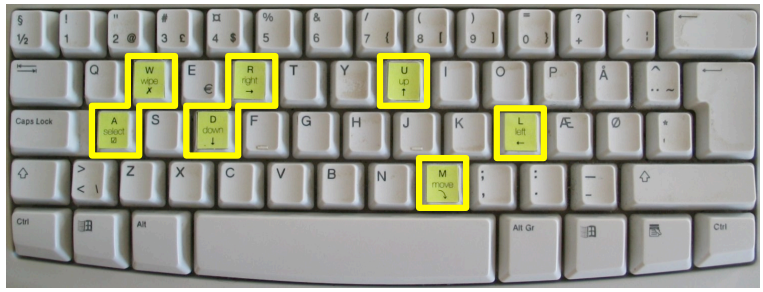


Figure 5.5.: Picture of WildDocs *v3* keyboard with marked shortcut keys (W, R, U, A, D, L, M)

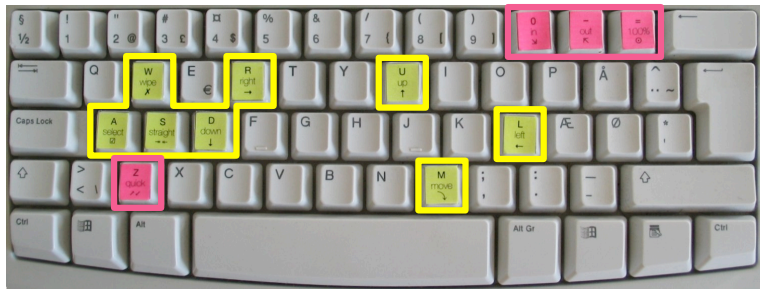


Figure 5.6.: Picture of WildDocs *v2* keyboard with marked shortcut keys (0, –, =, W, R, U, A, S, D, L, Z, M)

as random rotation are both based on simple real world observations. Figure 5.5 shows all shortcuts marked on the keyboard that are available for *v3*.

WildDocs *v2*

The main feature of WildDocs *v2* is enhanced zooming. There are four different zooming interactions implemented: menu zoom via mouse, menu zoom via keyboard shortcut, smooth zooming, and quickzoom.

The menu zoom exists as in *v4*; however, the user can activate those commands also by keyboard shortcuts: `CTRL-0` for zoom in, `CTRL--` for zoom out, and `CTRL-=` for zoom reset. As already mentioned above, the test application follows an US English keyboard layout whereas the physical keyboard used for testing was Danish. This only affected the menu zoom shortcuts. Because we used marked keys and informed the participants of group *v2* about the US English layout at the beginning of every test session, we did not experience any problem. Figure 5.6 shows a picture of the keyboard. The menu zoom keys are side by side. All zooming related shortcuts were marked in pink, whereas the others were marked in bright yellow, such as all shortcut labels of other versions.

Smooth zooming is another zoom method implemented in *v2*. It is accessible through pressing the right mouse button on the background and moving the mouse to the right for

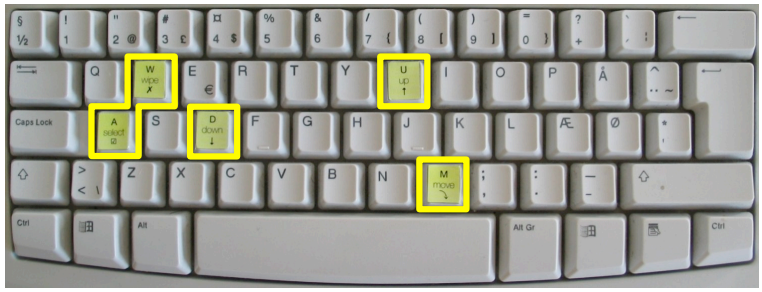


Figure 5.7.: Picture of WildDocs *v1* keyboard with marked shortcut keys (W, U, A, D, M)

zooming in or the left for zooming out. The further the mouse moves away from the original point where it was pressed, the faster zooming becomes. This action is stopped as soon as the right mouse button is released.

The forth method for zooming is quickzoom. Pressing `CTRL-Z` zooms out until all objects, including the desk metaphor, are visible on the screen.[3] The next quickzoom call zooms back to the former scale level to the position where the cursor is located at activation. `CTRL-Z` enables the user to zoom out completely and zoom back to a desired position very quickly. Zooming back is only active if no other zoom method, such as menu zoom or smooth zooming, is called. Other zoom actions cause the next quickzoom call to zoom to the level where all objects fit into the visible area. Quickzoom is animated. This supports the user in understanding the spatial relation of departure and destination location.

Because we intended to encourage navigation through zooming, scrollbars are disabled for this version.

WildDocs *v1*

The main difference between *v1* and *v4* concerns the object size. WildDocs *v1* does not support fixed size objects, but they can be resized by the user through dragging one of eight bounds handles located on the the object's corners or sides. The left screenshot of Fig. 5.8 depicts those bounds handles. By increasing an object's size, text objects show more of its content whereas images become magnified and show a distortion if not resized proportionally.

Versions supporting fixed size documents are based on the idea of implementing limitations in a way that users are familiar with. Because *v1* does not follow this idea, we did not implement a desk metaphor.

Shortcuts for pushing the object below the cursor to the right or left, automatically pushing a dragged object to the front when leaving the scope of the ones above, or the possibility to straighten piles are disabled. We left those features out, because they are closely related to the use of large sized documents, as they existed in our test for versions supporting fixed sizes, where `CTRL-L` or `CTRL-R` as well as `CTRL-S` were implemented to browse piles more efficiently or to save space due to neatly shaped stacks. Because of the small size as well as the rather unequal rectangular shapes of objects once they were resized, those features would not support the main focus of this version. This is similar to automatically pushing a node

[3] This could also be activated by selecting the "Toggle Quickzoom" menu entry located at the zoom menu. However, we did not experience any participant using the menu entry instead of the keyboard shortcut for quickzoom.

to the front, because we expected that objects of this version would be kept rather small and therefore would lose the scope of the above positioned objects quickly.

5.2.3. Documents and Questions Sets

Documents

Document IDs (DID) consist of the letter "d" for "document", followed by a unique two digit number.

Document vs Page We distinguish between documents and pages for versions supporting fixed size. A *document* is a coherent text block that may include graphics. It may consist of several pages, that are non-resizable rectangular objects on the screen. *Pages* can be recognized through cohesion supporting visual attributes, such as similar typesetting, same graphics on each page, or same page format. If neither coherence nor cohesion play a role in our argumentation, we may refer to "objects on the screen" instead of using the term "pages".

We did not use binding mechanisms for the experiment. Users were not able to bind pages explicitly, for example, as book or put them into a binder. Even though pages are bound through coherence and cohesion, they may be intentionally or unintentionally placed apart.

Variable size versions (*v1*) do not have the notion of pages. All objects are documents. In our experiment, every participant had 29 documents for organizing and finding phases, consisting of 56 single pages for versions supporting fixed size (*v2*, *v3*, and *v4*).

Document Types There were two different document sets, one for variable size versions (*v1*), the other for versions supporting fixed size (*v2*, *v3*, and *v4*). The reason for this was different capabilities of WildDocs versions[4] as well as a tight coupling of certain attributes to page-based or versions supporting fixed size, such as the notion of landscape vs portrait page formats. Each WildDocs version had 29 documents loaded during the main test. Table 5.2 marks both sets at the top (rows "Variable size" and "Fixed size"). Document *d24* to *d29* were exclusively for versions supporting fixed size, document *d30* to *d35* exclusively for *v1*. Document *d01* to *d23* were used by all versions.

The table shows a list of attributes. Symbols in parentheses next to the attribute names indicates whether an attribute exists for variable size (◐), fixed size (◑), or both document sets (●). The additional letter "o" or "h" within the parentheses indicates whether we interpret questions to this attribute as "obviously" visible or rather "hidden". This is used and further discussed in Sect. 5.3.3.

Marks within document columns (○, ◐, ◑, or ●) show which attributes a document owns. This is dependent on the document set as well as on the availability of attributes for them. For example, *d07* is available in both document sets. One attribute is "small graphical mark". However, this attribute is marked as existing only for fixed size versions (◑). That means that there is no small graphical mark on document *d07* for variable size documents, whereas there is one for fixed size versions.

Different symbols are used in document columns, representing if there was a question for the finding part that asked about an specific attribute. It also shows to which document set

[4] For example, *v1* cannot display graphics included in text. This would request advanced support in calculating complex layouts in real time.

Attribute	d01	d02	d03	d04	d05	d06	d07	d08	d09	d10	d11	d12	d13	d14	d15	d16	d17	d18	d19	d20	d21	d22	d23	d24	d25	d26	d27	d28	d29	d30	d31	d32	d33	d34	d35
Document set and number of pages																																			
– Variable size (◐)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
– Fixed size (◑)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○						
– Pages (◑)	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	4	4	4	2	2	2	2	3	4	*1*	2	2	2	2	*4*
Page format																																			
– Landscape (◑, o)	○	○	○	○																			◑												
– Portrait (◑, o)					○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○			○	○	○	○	○	○						
Font color																																			
– Completely (●, o)									●	●	○	●	○									●													
– Partly (●, h)														●				○			●								◐	◐					
Foreign or special font (sheet music)																																			
– Completely (●, o)									○	○	○				●	●	●								○	○									
– Partly (●, h)																		●				●													
Graphical mark																																			
– Small (◑, o)							○																	◑		◑									
– Large (◑, o)																													○	◑					
Content type																																			
– Grayscale (●, o)	●																			●				○				○	○						
– Large font (●, o)																			○	●															
– Short text (●, o)								●	○	○	○																								
– Layout (●, o)							●																	○				○	○						
Content media type																																			
– Picture only (●, o)	○	●	○	○	●	○																		◑											
– Text only (●, o)							○	○	○	○	○	●	◐	○	○	○	○	○	○	○	●	○	○		○	○	○	○	○	◐	○	○	○	○	◐
– Text w/figure (◑, h)																																			

(◐) = attr. exists for variable size; (◑) = attr. exists for fixed size; (●) = attr. exists for all doc sets; (o) = "obvious" attr.; (h) = "hidden" attr.;
○ = attribute exists for document; ◐ = attribute asked in *v1* tests; ◑ = attribute asked in *v2*, *v3*, and *v4* tests; ● = attribute asked in all tests

Table 5.2.: Visual attributes of documents

a question was connected. An empty circle ○ indicates that this particular attribute existed; however, it was not part of any question. A filled circle ● indicates that an attribute of the particular document was used for a question for all versions. If the symbol shows a partly filled circle on the left side ◐, the attribute was used only for a question of the variable size document set. If the filled part is on the right hand side ◑, it was used for a fixed size related question.

Page is an attribute of fixed size versions only. We used 11 single page, 13 two page, one three page, and three four page documents. Even though *v1* does not have the notion of multiple pages, Tab. 5.2 shows numbers in italic font in the page row for document *d30* to *d35*. They represent the equivalent number of fixed size pages for those documents.

We used documents with visual attributes, such as page format, font color, foreign or special font, graphical mark, specific content type, or media type. *Page format* is dependent on pages, since objects in *v1* can have arbitrary rectangular shapes. *Font color* was applied to whole texts or parts of them. This also applies to *foreign or special fonts*. We included documents completely or partly written in Arabic, Hebrew, Greek, and Japanese[5]. There was also sheet music as fixed size document.

There were also *graphical marks* on pages of some fixed size documents. We categorize them in small or large marks. Small marks included logos or flags on top of the pages, whereas large marks appeared as big yellow circles or large blue rhombi behind the text, covering most of the width or height of the pages.

Some attributes were based on what we call *content type*, such as grayscale, large font, short text, or layout. We had one question (*q56*) asking about a picture in grayscale (*d01*). This attribute does not fit to any other category; therefore, we created a new one. Two documents had exclusively large fonts (140 pt monospace font on document *d19* and *d20*) and four had a very short texts, only consisting of one sentence (*d08*, *d09*, *d10*, and *d11*). Some attributes were layout related, such as a two-column (*d24* or *d28*) or list layout (*d07*). The two-column layout depends on fixed size documents, because currently WildDocs does not support rendering of multiple column layouts in real time, which would be necessary for variable size objects of this kind.

We name the last attribute group *content media type*. It classifies documents as picture, text, or mix of both, according to their main appearance. For our experiment, we did not consider graphical marks as a mixed type, because the mark was not part of the original content, but had independent semantics and was added later by us. There was only one document that had both (*d24*): a scientific article with a user pictogram on the second page.

We used documents of various content: Pictures showed different objects, such as nature, buildings, humans, or animals. The other documents were taken form various areas, such as computer science, fiction, or history.

Even though there were two document sets, we aimed to make both comparable. We were able to use 23 of 29 documents for both sets and only added six additional ones per set. We put efforts in designing and choosing those individual additional documents in a way that they were equal with respect to our test setup and statistical evaluation.

Code Each document had a unique four digit code on top, as shown in Fig. 5.8. For fixed size supporting versions with several pages per document, the code only appeared on the first

[5] Japanese (document *d25*) was only available for fixed size documents.

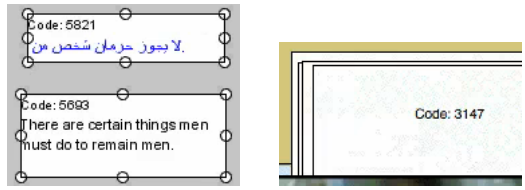


Figure 5.8.: Codes on top of documents: variable size (*v1*, left) and fixed size (*v4*, right)

page. The participant was asked to read this code aloud after finding the correct document to a question.

There were three reasons for using codes on documents and asking the participant to read them aloud. Firstly, a document can be identified by that number. However, this was not the most important argument, because the investigator was familiar with all documents and were able to see instantly on the document's appearance whether the participant found the right one or not.

Secondly, and more importantly, the participant was forced to find the beginning of a document. This was only relevant for fixed size documents that contained more than one page.

Thirdly, the most important argument for codes is to make sure that the user is forced to zoom in to a level where he/she can read the content. For example, when asked about an easy to recognize, large graphical symbol on a document, the user may find the document while being fully zoomed out. However, he/she still needs some time to zoom in and adjust the position so that the first page can be read.

This was an improvement of our pre-tests, where we asked participants to read the first sentence of a document aloud. However, some of them interpreted "sentence" as "headings", others as "first sentence of the main text". Because most headings were written in a larger font size than normal text, we encountered unequal situations among those who read the heading versus those who read the actual text. By introducing a code of equal size and position on all documents, we unified this particular condition for all participants and made it comparable for statistical evaluation.

The code was not considered as part of the document with respect to its visual attributes. For example, a document that was completely written in blue font was still considered to be completely written in blue, even though the code on top was in black letters.

Questions

Question IDs (QID) consist of the letter "q" for "question", followed by a unique two digit number.

Table 5.3 shows a list of question asked during the finding phases. The last column contains the document IDs (DID) to which the questions refer to. They draw a connection to Tab. 5.2.

We divide questions into two groups, regarding how easy it is to see a certain visual attribute. One group contains attributes that are easy to see, for example, the use of *exclusively* red fonts on a document or a *large* yellow circle behind the text of each page. We refer to this group as "obvious" visual attributes, in Tab. 5.3 marked as "(o)". The other group contains visual cues that are rather hidden, for example, a green written paragraph on the second page or a paragraph written in Hebrew on the third page. We name this group "hidden" visual

QID	Question	DID
Questions for versions supporting variable size only (◐)		
q03 (o)	Find the document about Abraham Lincoln.	*d13*
q48 (o)	Find the document that is about an application "iTeXMac".	*d30*
q49 (o)	Find the document about post-structuralism.	*d35*
q47 (h)	Find the text that uses black fonts and has an orange written part.	*d32*
q53 (h)	Find the text that uses black fonts and has a green written part.	*d31*
Questions for versions supporting fixed size only (◑)		
q11 (o)	Find the document that uses black fonts on landscape format.	*d23*
q38 (o)	Find the document that has a large yellow circle behind the text.	*d29*
q39 (o)	Find the document that has an AUE logo[a] on top.	*d25*
q44 (o)	Find the document that has the Danish[b] flag on top.	*d27*
q31 (h)	Find the document that has a figure on it that shows two user pictograms as part of the figure.	*d24*
Questions for all versions (●)		
q04 (o)	Find the document that uses exclusively red fonts and covers more than 3 lines.	*d12*
q05 (o)	Find the document about interpretation of nature.	*d12*
q09 (o)	Find the document about the application "mpeg2decX".	*d21*
q12 (o)	Find the document that is written in Arabic and is longer than 3 lines.	*d15*
q13 (o)	Find the document that is written in Greek and is longer than 3 lines.	*d16*
q14 (o)	Find the document that is written in Hebrew and is longer than 3 lines.	*d17*
q19 (o)	Find the document that uses exclusively green fonts.	*d22*
q22 (o)	Find the document that has large numbers and signs on it.	*d20*
q23 (o)	Find the document that contains only one sentence. This sentence is written in English.	*d08*
q26 (o)	Find the document that contains only one sentence. This sentence is written in dark blue font.	*d09*
q30 (o)	Find the document that contains only one sentence. This sentence is written in brown font.	*d10*
q41 (o)	Find the document that shows a list of smilies.	*d07*
q54 (o)	Find the picture that shows a red flower.	*d05*
q56 (o)	Find the picture that is in grayscale.	*d01*
q57 (o)	Find the picture that shows a cat.	*d57*
q06 (h)	Find the text that uses black fonts and has a blue written part.	*d14*
q08 (h)	Find the text that uses black fonts and has a red written part.	*d21*
q15 (h)	Find the document that is partly written in Arabic.	*d18*
q17 (h)	Find the document that is partly written in Hebrew.	*d22*

(o) = "obvious" visual attribute; (h) = "hidden" visual attribute

[a] The abbreviation "AUE" stands for "Aalborg University Esbjerg". All participants were affiliated to this institute and therefore were aware of the abbreviation and its logo.

[b] All participants lived in Denmark and were aware of the appearance of the Danish flag.

Table 5.3.: Questions asked during finding parts of test sessions

attributes, marked as "(h)". Table 5.2 relates those categories to the visual attributes of the documents.

All questions start with "Find the . . . " and end with "(Read the code on top of the document aloud.)"[6] Answers to questions were unique. There was only one correct answer per question. However, there were three pairs of questions, each asking about the same document: *q04* and *q05* both asked about *d12*, *q08* and *q09* about *d21*, and *q17* and *q19* about *d22*. Except for the first mentioned pair, they contained one question related to an "obvious" and one related to a "hidden" attribute.

There were the same set of 19 questions assigned for all sessions. Additionally, we added five separate questions for each of both document sets. The question types were slightly different; however, we attempted to find comparable ones for both document sets. For the additional questions of variable size versions, we added three "obviously" visible attribute questions. Each of them asked about the content of the document. The central terms of the question[7] was part of the document's headline and therefore visible at the top of each document. The remaining two questions, *q47* and *q53*, were about documents that had partly colored text.

Questions that were only asked in sessions with versions supporting fixed size include four questions related to "obvious" attributes. *q11* referred to the document's format, *q38* to large marks, and *q39* as well as *q44* to small marks on documents. One additional question (*q31*) was classified as being related to a "hidden" visual attribute.

5.2.4. Design

Pre-tests have shown that participants were not able to preform tests on more than one WildDocs version in a concentrated way. The reason was that learning and organizing phase for each tested version were necessary, but increased the total time spent drastically. The participants' concentration dropped after testing the first version. Asking participants to attend to several tests at several times was not applicable due to organizational reasons. Therefore, we designed a between-group test; each participant was ask to test exactly one WildDocs version.

We expected between 40 and 50 people to attend. In order to have an equal distribution of participants per group, we created a pool of 40 pointers, ten assigned to each WildDocs version. Each participant received one of the unassigned pointers randomly. After more than 35 tests were done, we added additional two pointers for each version and increased the total number to 48.

5.2.5. Procedure

The test procedure was divided into three main parts: an introduction; an organizing phase; and, a finding phase. The whole session was surrounded by pre- and post-test questionnaire and explanations.

Pre-Test Phase

After welcoming the participant, the investigator gave a brief introduction on the test workflow and asked to read and sign a form that briefly describes the experiment and explains our

[6] The part that tells the participant to read the code aloud is not written in Tab. 5.3.

[7] Central terms were "Abraham Lincoln" for *q03*; "iTeXMac" for *q48*; and, "post-structuralism" for *q49*.

information policy about the gathered data. The form is printed in Sect. A.1. Then, the investigator asked general questions, such as age, nationality, profession, experiences in computer usage, etc. The results of most of those questions are discussed in Sect. 5.2.6. The complete pre-test questionnaire can be found in Sect. A.2.

Introduction Phase

The introduction phase started with a movie that explained the used WildDocs version and described briefly the different experiment phases. There was an individual introduction movie for each version. They lasted between 5:44 (*v1*) and 9:20 Minutes (*v2*). The movie's text for each version can be read in Sect. A.3. The movie was displayed on the screen to the right of the participant. He/she had the possibility to test WildDocs's features while watching the introduction. During that time, the WildDocs version had a small set of documents[8] loaded that were independent of the document set used later. The second part of the movie explained the different phases of the test, organization and finding, and asked the participant to think aloud when problems occur. For versions supporting fixed size, it also shows the difference between documents and pages, as discussed in Sect. 5.2.3.

After the movie was finished, the investigator recalled the presented functions and asked the participant to try them at least one time on the test document set. We wanted to be sure that the participant was aware of all features after the introduction.

Because most of the first five participants had difficulties with recognizing foreign language fonts, such as Arabic, Hebrew, or Greek, we introduced sample pages of all used foreign fonts[9] at the end of the introduction phase, starting with participant *s06*. Caused by this change of the test workflow, we had to modify the set of data we took for evaluation, as discussed in Sect. 5.3.1.

The participant was told that the four digit code on top of each document is not considered as part of the document with respect to its visual attributes, as described in Sect. 5.2.3. For participants with versions supporting fixed size (*v2*, *v3*, and *v4*), the investigator also pointed out again the difference between documents versus pages and explained that pages of documents with more then one page will be put in a reverse sequence in the very beginning. The participant was not told how many documents or objects there were on the screen.

Organization Phase

The investigator loaded the appropriate set of documents into WildDocs. They appeared at the upper left corner of the screen as straight pile in version *v1*, *v2*, or *v4*, as sloppy pile in version *v3*. Multiple page documents in versions *v2*, *v3*, or *v4* had pages that belonged to the same document next to each other, but their sequence was reversed, the last page was on top. The organization phase was introduced with the task description on the introduction monitor: "Organize the objects in a way that allows you finding them quickly afterwards." The time of organizing documents was measured. The results are discussed in Sect. 5.3.2. Statistics were written in a log file after the participant finished organizing.

The test database allowed the investigator to add comments related to the organization phase. Those comments were intended to be used for further analysis later. There were

[8] All participants had four objects on the screen for training. Users of *v1* had two pictures and two RTF documents. Users of the other versions had two pictures and one text document, containing two pages.

[9] Used foreign fonts were Arabic, Hebrew, Greek, and Japanese. The sample pages can be seen in Sect. A.4.

also fields for questions about organizing or finding that the investigator wanted to ask the participant after the test, in addition to the predefined questionnaire.

Finding Phase

The finding phase consisted of 24 preselected questions in random order. As discussed in Sect. 5.2.3, there were two sets of questions, depending on whether the participant used a variable size or fixed size version.

Each question was displayed individually on the introduction screen at the participant's right hand side and stayed there until the question was completed. The participant was asked to read the question aloud. The time between the user started searching till he/she started reading the four digit code on top of the document was measured.

Occasionally, the investigator added further question specific information into the WildDocs experiment database. This was intended to be used for later analysis. If a completed question was not correctly answered, the investigator added "wrong" or "gave up", depending on whether the user has chosen the wrong answer or gave up before finding the correct document.

Similar to the comments about the organization phase, the investigator also added general comments dedicated to the finding phase. Also the additional fields for adding questions during the test to be asked after completion were available for finding related remarks.

Post-Test Phase

The post-test phase completed the session with a questionnaire. The participant was asked to describe how he/she organized and found documents. Furthermore, the application was rated by the user.[10] The questions that were noted by the investigator during the test were asked. A complete list of post-test questions can be found in Sect. B.1.

At the very end, the participant was asked whether he/she would like to have further information or has specific questions about the project or the experiment. He/she was also asked not to tell anyone who is scheduled to perform the test later about the project or session procedure until the experiment was finished. After the investigator's appreciation for taking part of the research project, the participant was released.

5.2.6. Participants

Each session was assigned to one single participant. The session ID contains the letter "s", followed by a continuous two digit number, starting with "01". Participants are named after their session number.

We had 45 volunteers for our experiment. All of them were affiliated to the Department of Software and Media Technology, Aalborg University Esbjerg, Denmark. Most of them were male (37 male, 8 female). They had various nationalities (29 Danish, 5 Chinese, 2 Indians, and a Belgian, Canadian, German, Italian, Jordanian, Portuguese, Russian, American, Venezuelan). There were 20 undergraduate and 17 graduate student volunteers. The remaining 8 volunteers were staff members, all academic except of one who worked in the administration.

[10] The statistical evaluation of the user ratings will be discussed in Sect. 5.3.6.

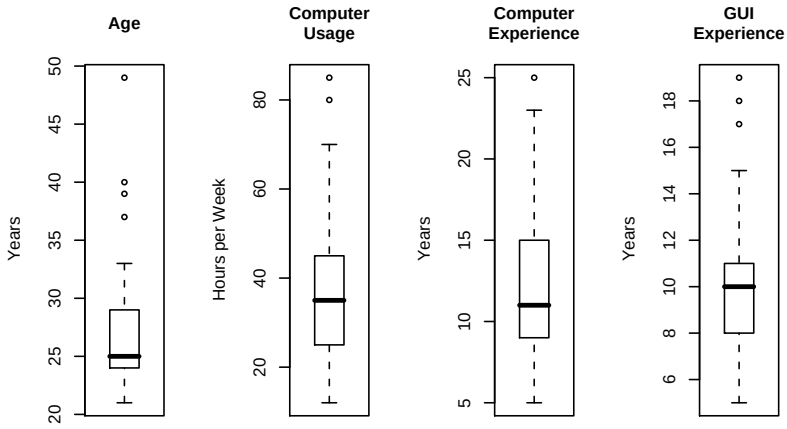


Figure 5.9.: Information about participants

Figure 5.9 depicts information about their age ($Mdn = 25.0$, $M = 27.0$, $SE = .83$)[11] and their experience in computers, such as hours of computer usage in a typical week ($Mdn = 35$, $M = 38$, $SE = 2.54$), years of computer experience ($Mdn = 11.0$, $M = 12.11$, $SE = .73$), or years of GUI experience ($Mdn = 10.0$, $M = 10.27$, $SE = .51$). Most of the participants ($N = 27$) stated that they did not have experiences in spatial organization of objects on the screen, whereas the other participants ($N = 18$) had. All of the participants confirmed that nobody talked to them about the experiment before the test. They did not get any information about the purpose of the test until the end of the session.

5.3. Statistical Results

5.3.1. Remarks on Skipped or Taken Out Questions

Most of the statistical calculations are based on finding phase related data, such as failure rate for finding or time for finding correctly. Almost every participant had 24 questions. There were a few exceptions to this. Some had fewer questions to answer. Therefore, all statistics that are related to the number of questions are calculated as rate per asked questions.

We distinguish between *skipped* and *taken out* questions. Skipped questions are questions that were not asked during the test. Taken out questions were asked during the test, but taken out for most tests afterwards. Those cases are discussed below. There were 1080 questions in total, 24 per participant. 16 of them (= 1.5 %) are marked as skipped and 28 (= 2.6 %) as taken out.

Most statistical calculations are based on questions that were neither skipped nor taken out. Otherwise, it is explicitly mentioned in the text. Examples for which we do not consider skipped questions, but taken out ones are described in Sect. 5.3.4.

[11] The abbreviations stand for *Median* (Mdn), *Mean* (M), and *Standard Error* (SE).

Participant *s27*

Participant *s27* had WildDocs *v4* (fixed size document support) assigned. He/she spent 7:55 Minutes for organizing. However, he/she did not spread the documents out on the virtual desk, but kept them all visible on the screen. He/she did not zoom out and changed the viewpoint on the visible are only slightly. Because of the small area used for organizing, the document density was high. Finding documents took long. The investigator was not supposed to interfere or give a time-out. Because this participant was patient, he/she spent up to 7:18 Minutes trying to answer a single question. The finding phase for the first eleven questions took 40 Minutes. Six of them were not answered correctly.

Participant *s27* was the only one who spent lots of time for finding and was patient to do so. Because the next participant was scheduled around that time, we decided to skip the remaining 14 questions, without telling the participant that the test was planned to take longer. This decision was also based on the assumption that *s27* was an outlier. However, we still consider the completed questions for our statistical evaluation. The post-test phase was completed normally.

Participant *s41*

As mentioned in Sect. 5.2.3, there were slightly different sets of documents for WildDocs *v1* and the other versions. Participant *s41* had *v2* assigned that would have required questions for fixed size documents. However, because of a mistake by the investigator, questions of the variable size document set were assigned. This mistake became obvious during the test. There were five questions that were not part of the right question set. The first one was *q03*. Even though it was not part of the question set, this question was answered correctly, because the document existed also for fixed size versions. However, we still marked it as "taken out", to avoid differences to question sets of other fixed size document sessions. The next two questions, *q48* and *q49*, could not be answered, because the requested documents were not part of the used set. They were also considered as "taken out". The investigator realized that a mistake happened and skipped the remaining two questions of the wrong question set, *q47* and *q53*.

Participants *s01*, *s02*, *s03*, *s04*, and *s05*

As described in Sect. 5.2.3, there were documents that were partly or completely written in Hebrew, Arabic, Greek, and Japanese. Some questions existed for the first three mentioned font types. During the first five sessions it became clear that it was very difficult for participants to answer questions that were based on those foreign language fonts. In most cases, those failures were based on the participant's lack of prior knowledge rather than based on the created spatial structure. We did not investigate people's existing knowledge of non-Latin fonts. Therefore, we decided to introduce those four foreign font types before the test started. We showed sample texts of one page per language to make the participant aware of the used foreign fonts. The sample pages were not part of any document set.

There were only five questions related to foreign fonts, *q12*, *q13*, *q14*, *q15*, and *q17*, each a member of both question sets. After the test was completed, we counted the number of foreknowledge-based failures of those questions. The first five participants who did not see the font sample pages did not answer 8 of 25 related questions correctly due to lack of prior

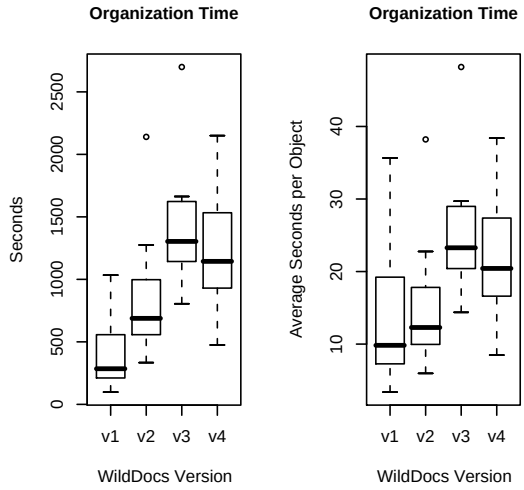


Figure 5.10.: Time spent for organization task

knowledge. This is a rate of 32 %. The remaining participants had 198 related questions to answer[12] and got 13 of them wrong due to lack of existing knowledge, that is 6.57 %. Because of this obvious change, we decided to take out the above mentioned five questions for the first five participants.

5.3.2. Organizing Documents in WildDocs

Even though it is not part of our hypotheses, looking at the organization part helps us see some differences and similarities among the used WildDocs versions. In this section, we focus on the time spent for organizing the objects on the screen as well as on the occupied area.

Time

The independent variable is the WildDocs version; the dependent variable is the used time in seconds for organizing. We consider *v1* ($Mdn = 285.0$, $M = 390.9$, $SE = 80.9$), *v2* ($Mdn = 688.0$, $M = 838.6$, $SE = 141.3$), *v3* ($Mdn = 1304$, $M = 1408$, $SE = 170.7$), and *v4* ($Mdn = 1144$, $M = 1235$, $SE = 159.8$). Shapiro-Wilk tests and the evaluation of Q-Q plots and histograms unveiled that *v1*, *v3*, and *v4* are normally distributed, whereas *v2* is not ($p = .01$). We apply a transformation of $\log(x)$ on all samples in order to receive normal distributions.[13]

Analysis of variance shows a significant effect ($F(3,41) = 16.92$, $p < .001$, $r = .74$). Levene's test reports the assumption of homogeneous variances ($F(3,41) = 1.27$, $p = .30$) for log transformed organization time. This is confirmed by F-tests. Therefore, we use the standard t-test with Bonferroni correction ($\alpha = \frac{.05}{3} = .0167$) to try to falsify our hypotheses.

[12] It would have been 200 questions; however, two of them were skipped in session *s27*.

[13] Before the transformation, only *v1* ($p = .052$), *v3* ($p = .08$), and *v4* ($p = .89$) were normally distributed; *v2* ($p = .01$) was not. After transforming, the p-value increased for *v1* ($p = .98$), *v2* ($p = .92$), and *v3* ($p = .68$), but not for *v4* ($p = .58$). Those tendencies are confirmed by examining Q-Q plots and histograms.

Members of group *v1* were significantly faster in organizing documents compared to those of our control group *v4*. The effect size is large[14] ($t(21) = 5.08$, $p < .001$, $r = .74$).[15] The comparisons of *v4* to *v3* ($t(19) = -.92$, $p = .37$, $r = .21$) or *v2* ($t(21) = 2.02$, $p = .06$, $r = .40$) do not show a significant effect; however, the latter one reports a rather small p-value and a medium to large effect size.

As mentioned in Sect. 5.2.1, the number of documents was equal for all versions. However, caused by fixed sizes and the notion of *pages*, *v2*, *v3*, and *v4* each had 56 objects on the screen, *v1* only 29. Now, we consider the average seconds for organization per object, including a log transformation. Levene's test result does not change to the above mentioned one. The analysis of variance still shows a significant effect ($F(3,41) = 5.22$, $p = .004$, $r = .53$). Figure 5.10 depicts the absolute time as well as the average seconds per object used for organization, both without transformation. The interesting evaluation is how *v1* compares to our control group *v4*. The relations of *v2* or *v3* to *v4* stay the same, because the number of objects on the screen did not change. However, because two of the above mentioned comparisons apply also for this test, we accept the same Bonferroni correction on the per comparison error rate ($\alpha = .0167$). A t-test shows that *v1* is not significantly different from *v4* anymore ($t(21) = 2.45$, $p = .02$, $r = .47$).[16]

We conclude that the implementation of zooming (*v2*) or rotation (*v3*) apparently does not significantly affect the time used for organizing documents, whereas variable sized documents without the notion of pages (*v1*) does. However, this seems to be mainly based on the fact that *v1* had fewer objects in our experimental setup. Our evaluation of the average organization time per object shows that *v1* is not significantly different to *v4*. This draws the question of whether binding mechanisms would reduce the organization time period for WildDocs versions supporting fixed size to a similar level than we experienced with *v1*. This hypothesis could be based on the argument that bindings reduce the number of objects at a certain structure level. Even though it is not statistically significant, there is a tendency that users of *v2* were faster in organizing documents compared to our control group. This raises the question about what improvements of zooming mechanisms in WildDocs *v2* would create a significant difference.

Area

In this section, we discuss our results for the occupied area after the organization task. The area in pixels[17] squared is the dependent variable, the WildDocs version the independent variable. We consider *v1* ($Mdn = 633.3\,\text{k}$, $M = 599.3\,\text{k}$, $SE = 29.08\,\text{k}$), *v2* ($Mdn = 22.49\,\text{M}$, $M = 19.21\,\text{M}$, $SE = 2.216\,\text{M}$), *v3* ($Mdn = 24.49\,\text{M}$, $M = 24.96\,\text{M}$, $SE = 724.80\,\text{k}$), and the control group *v4* ($Mdn = 18.58\,\text{M}$, $M = 15.78\,\text{M}$, $SE = 2.433\,\text{M}$). Shapiro-Wilk tests on the occupied area after organizing shows normal distributions for *v1* ($p = .58$), *v3* ($p = .56$), and *v4* ($p = .29$), but not for *v2* ($p = .004$). This is confirmed by Q-Q plots and histograms.

[14] The labels *small* ($r = .10$), *medium* ($r = .30$), and *large* ($r = .50$) for effect sizes are suggested by Cohen (1992, 156).

[15] The significant effect between *v4* and *v1* holds also for non-transformed data samples ($t(14.90) = 4.71$, $p < .001$, $r = .77$). We applied Welch's t-test because the F-test showed that the assumption of homogeneity of variance among both groups is violated ($p = .048$).

[16] This is more obvious for non-transformed samples ($t(21) = 2.14$, $p = .04$, $r = .42$). We used the standard t-test for testing without transformation, because an F-test lets us assume homogeneity of variance ($p = .95$).

[17] Pixels refer to a scale level at 100 %.

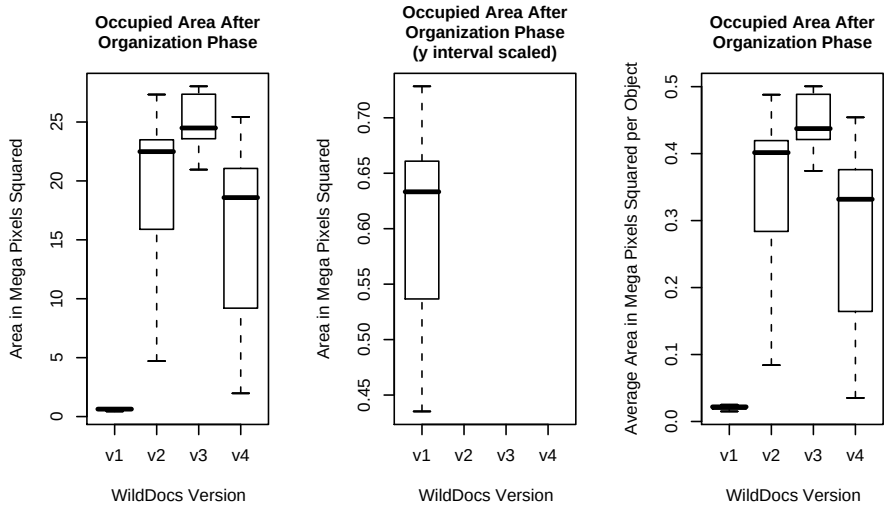


Figure 5.11.: Occupied area after organization phase

We apply an x^3 transformation on all groups and receive normal distributions.[18] We use the transformed data for the following tests.

Levene's test assumes a violation of homogeneous variances ($F(3,40) = 4.89$, $p = .005$). Nevertheless, we report F-tests, following the suggestion in Field & Hole (2003). The violation is caused by *v1*; Levene's test on *v2*, *v3*, and *v4* signifies homogeneous variances ($F(2,30) = .36$, $p = .70$). The results show a significant effect as well as a large effect size among all WildDocs versions ($F(3,40) = 19.44$, $p < .001$, $r = .77$).

To find individual differences between *v1*, *v2*, or *v3* and our control version *v4*, we apply a Bonferroni correction ($\alpha = \frac{.05}{3} = .0167$). As the left boxplots of Fig. 5.11 depict,[19] there is a large difference between *v1* and the other versions. An F-test supports the assumption that *v1* and *v4* do not have homogeneous variances ($p < .001$). Therefore, we use Welch's t-test. As expected, the result shows that the area used for *v1* is significantly smaller than for *v4* and the effect size is large ($t(10) = 3.62$, $p = .005$, $r = .75$).[20]

As argued above, Levene's test assumes homogeneous variances among *v2*, *v3*, and *v4*. This is confirmed by F-tests. We base the following t-tests on this assumption. It turns out that participants of group *v2* and *v4* used a similar amount of space – the t-test result shows no significance ($t(21) = -1.27$, $p = .22$, $r = .27$). However, members of *v3* used significantly more space for placing documents compared to *v4* ($t(19) = -4.09$, $p < .001$, $r = .68$).[21] The effect size is large.

[18] We examined Q-Q plots, histograms, as well as Shapiro-Wilk tests for *v1* ($p = .62$), *v2* ($p = .12$), *v3* ($p = .43$), and *v4* ($p = .17$).

[19] Figure 5.11 depicts samples before the transformation was applied.

[20] Also the result of Welch's t-test of *v1* and *v4* before transformation is significantly different and shows a large effect size ($t(10.0) = 6.24$, $p < .001$, $r = .89$). We used Welch's t-test under the assumption of non-homogeneous variances, based on an F-test result ($p < .001$).

[21] The significance of *v3* compared to *v4* can also be shown for non-transformed samples ($t(11.75) = -3.62$, $p = .004$, $r = .73$). We used Welch's t-test, because of the assumption of non-homogeneous variances ($p < .001$).

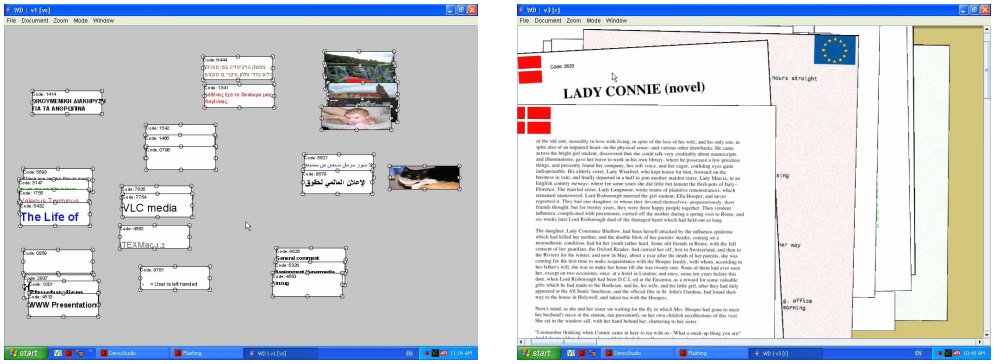


Figure 5.12.: Screenshots of *v1* (left) and *v3* (right), each at 100 % zoom level

As the right boxplot in Fig. 5.11 suggests,[22] there remains a significant effect between *v1* and *v4* when testing the average area per object ($t(10) = 3.62$, $p = .005$, $r = .75$).[23]

We conclude that group members of *v2* and *v4* used not significantly more space for organizing objects. Further, we can see that members of group *v3* used a larger area, those of *v1* a smaller area, both significant compared to observations of *v4*. The large effect between *v1* and *v4* (and as Fig. 5.11 shows also to the remaining two versions) was based on the default object size, as depicted in Fig. 5.12. It shows two screenshots, one of *v1* and one of *v3*. Both have a zoom level of 100 %. Whereas *v1* shows small objects, *v3* has fixed size pages that do not fit on the screen in portrait format at the default zoom level. The result also shows that the participants working with WildDocs *v1* did not enlarge nodes and occupy a larger area, similar to what members of other groups were forced to do by having fixed size documents.

The difference of *v4* to *v3* apparently was based on the incidental rotation behavior of the latter version, also depicted in Fig. 5.12. It seems that participants of this group cared less about occupied space and did not adjust documents manually in order to have a tidy looking spatial structure. It can be assumed that more widely spread and sloppily arranged objects give a better overview, because they show more of its content than those that are arranged as straight piles on small areas. To some extent, WildDocs *v3* forces the user to create sloppy spatial structures and therefore occupy more space due to its automatic incidental rotation behavior.

5.3.3. Finding Documents in WildDocs

Time for Correct Answers

This section evaluates exclusively correctly given answers of the finding part. As discussed in Sect. 5.2.3, we divide questions into two categories, regarding how easy it is to see the asked visual attribute. Figure 5.13 depicts boxplots[24] for both categories with respect to the time spent for finding documents.

[22] The boxplot shows samples before transformation.
[23] We used Welch's t-test due to the assumption of non-homogeneous variances ($p < .001$).
[24] The boxplots depict data before transformation.

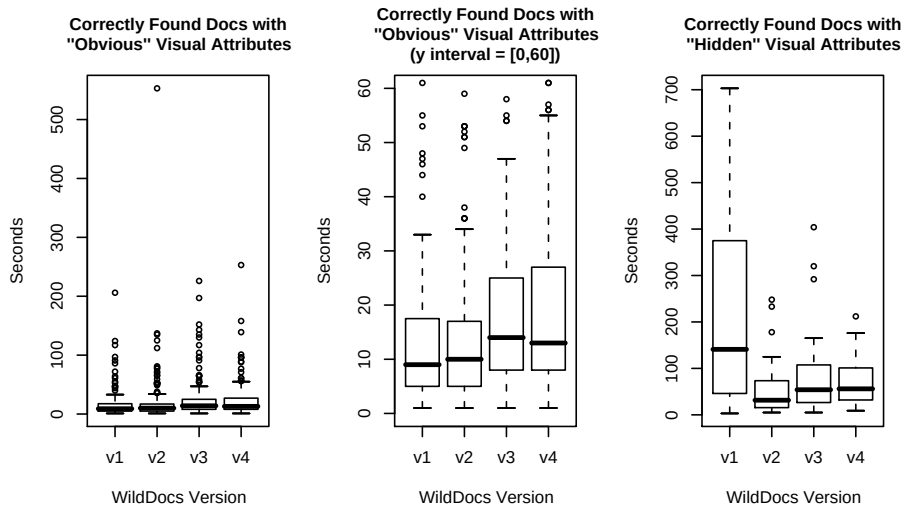


Figure 5.13.: Time spent for correctly finding documents

The independent variable for the following is the WildDocs version; the dependent variable the time for finding the correct document.

Documents with "Obvious" Visual Attributes The descriptive statistics are: *v1* ($Mdn = 9.0$, $M = 15.8$, $SE = 1.67$), *v2* ($Mdn = 10.0$, $M = 20.1$, $SE = 3.10$), *v3* ($Mdn = 14.0$, $M = 25.1$, $SE = 2.53$), and *v4* ($Mdn = 13.0$, $M = 24.2$, $SE = 2.53$). Q-Q plots as well as Shapiro-Wilk tests show that none of the given time-based data for correctly finding documents with "obvious" visual attributes is normally distributed ($p < .001$ for all groups). A log transformation solved the problem partly. Shapiro-Wilk tests assume normal distribution for *v1* ($p = .08$), but still reject *v2* ($p < .001$), *v3* ($p = .01$) and *v4* ($p = .01$). However, our examination of Q-Q plots and histograms of transformed data let us assume that it is reasonable to use log transformed values with ANOVA or t-tests instead of using non-parametric procedures. The following tests are based on transformed data.

Levene's test assumes homogeneity of variance ($F(3,721) = .81$, $p = .49$). The analysis of variance indicates significant differences between two or more groups ($F(3,721) = 12.62$, $p < .001$, $r = .22$). In order to see the differences, we used standard t-tests to compare our control group *v4* to the remaining groups, assuming homogeneous variances. We apply a Bonferroni correction for performing three tests ($\alpha = \frac{.05}{3} = .0167$).

Our results show that members of group *v1* ($t(347) = 4.71$, $p < .001$, $r = .24$) and group *v2* ($t(353) = 3.03$, $p = .003$, $r = .16$) were significantly faster in finding documents with "obvious" visual attributes compared to members of group *v4*, whereas members of group *v3* ($t(327) = -.28$, $p = .78$, $r = .02$) were not. This tendency can also be seen on the boxplots in the middle of Fig. 5.13 that depict the lower part up to 60 seconds. The effect size for *v1* is small, for *v2* between small and medium.

We have shown that variable document size (*v1*) and zooming features (*v2*) support finding of documents with easily recognizable visual attributes compared to our control group (*v4*). The effect size for zooming is slightly larger than the one for fixed size documents. We

also pointed out that there is no effect of incidental rotation or sloppiness (*v3*) regarding the finding time. This falsifies two of our hypotheses for the used document and question sets: Despite our assumption, neither incidental rotation (*v3*) nor fixed size documents (*v4*, compared to variable size support in *v1*) seem to help the user in finding information. We were not able to falsify our claim that WildDocs with zooming enabled (*v2*) supports a user in finding information more efficiently than without (*v4*).

Documents with "Hidden" Visual Attributes Now, we consider *v1* ($Mdn = 141.0$, $M = 211.0$, $SE = 33.30$), *v2* ($Mdn = 31.5$, $M = 57.8$, $SE = 12.46$), *v3* ($Mdn = 54.0$, $M = 89.7$, $SE = 19.26$), and *v4* ($Mdn = 56.0$, $M = 72.8$, $SE = 10.59$) with respect to the finding time of documents with "hidden" visual attributes. Similar to the observation for documents with "obvious" visual attributes, the gathered time data is not normally distributed.[25] A $\sqrt[4]{x}$ transformation causes a positive result of the data's distributions. Shapiro-Wilk still rejects *v1* ($p = .048$), but assumes for all other groups normal distributions.[26] Based on our examination of Q-Q plots and histograms of the transformed sets, we have good reason to assume that the distribution of the different data sets, including *v1*, is close enough to normal that we can accept parametric statistics. In the following, we use $\sqrt[4]{x}$ transformed data sets.

Levene's test rejects the assumption of homogeneity of variance ($F(3,110) = 6.14$, $p < .001$), nevertheless, we report F-tests, as suggested by Field & Hole (2003). ANOVA reports a significant effect among the tested groups with a medium effect size ($F(3,110) = 6.06$, $p < .001$, $r = .38$). We apply a Bonferroni correction for three comparisons ($\alpha = \frac{.05}{3} = .0167$) and use t-tests to point out differences.

Based on the assumption of non-homogeneous variance,[27] we use Welch's t-test for calculating the effect between *v1* and our control group *v4*. As the right boxplots in Fig. 5.13 suggest,[28] members of group *v1* were significantly slower in finding documents with "hidden" visual attributes correctly than those of group *v4* ($t(49.19) = -2.63$, $p = .01$, $r = .35$). The effect size is medium. F-tests for the remaining comparisons assume homogeneity of variance.[29] Standard t-tests did not discover significant differences between *v4* and *v2* ($t(51) = 1.75$, $p = .09$, $r = .24$) or *v4* and *v3* ($t(50) = .009$, $p = .99$, $r = .001$).

We conclude that finding documents on the basis of "hidden" visual attributes correctly takes significantly longer for variable size (*v1*) versions. Although our tests do not report significant effects for *v2* or *v3* compared to *v4*, there is a tendency that zooming (*v2*) supports the user, whereas rotation (*v3*) does not show any effect for finding "hidden" attributes. We are able to falsify two of our hypotheses: Neither zooming nor rotation significantly decrease the time for finding documents on the basis of "hidden" visual attributes. However, WildDocs with fixed size document support (*v4*) is significantly faster than without (*v1*, variable size objects).

[25] Shapiro-Wilk support our evaluation of Q-Q plots and histograms: *v1* ($p = .003$), *v2* ($p < .001$), *v3* ($p < .001$), and *v4* ($p = .01$).

[26] The results were *v2* ($p = .21$), *v3* ($p = .50$), and *v4* ($p = .93$).

[27] The F-test assumes that the homogeneity of variance of *v1* and *v4* is violated ($p < .001$).

[28] The boxplots depict data before transformation.

[29] This is for the comparison of *v4* with *v2* ($p = .17$) and *v4* with *v3* ($p = .06$).

Incorrect Answers

We use the WildDocs version as the independent variable and the rate of not correctly given answers as the dependent variable.

All questions for which the participant did not find the correct document were classified either as "wrong" (this is when a wrong document was chosen) or as "gave up" (this is when the participant did not find the right document and gave up). We experienced several cases where participants chose a document that they thought would be the closest to the asked one, even though they were aware of the fact that it was not the correct one. Therefore, we decided afterward to combine both groups for the following evaluation.

After the test we classified all incorrect answers according to the reason why a participant did not find the right document into "condition-based problem" or "structure-based problem". The goal is to isolate problems that were based on the spatial structure, not on other conditions. The following sections will discuss them separately.

Condition-Based Problems Failures due to condition problems were caused by lack of prior knowledge that would have been needed to find a specific document or some physical limitations of the participant. For example, one question asked about a document with a "pictogram" on it, but some participants did not know the meaning of this term. Other examples are participants who were not able to recall how a certain foreign language looks (e. g., Hebrew or Arabic). A case of physical limitation was color blindness, for example, a participant chose the document with brown font, thinking that this would be red. We were able to point out conditional problems, because users were asked to think aloud whenever they encounter a problem.[30] We also classify incorrectly understood questions as conditional problems. For example, the question "Find the document that is written in Arabic and is longer than 3 lines." was read by some participants as "... and is *no* longer than 3 lines."

We consider group *v1* ($Mdn = 0$, $M = .017$, $SE = .008$), *v2* ($Mdn = 0$, $M = .036$, $SE = .014$), *v3* ($Mdn = .083$, $M = .058$, $SE = .014$), and *v4* ($Mdn = .042$, $M = .056$, $SE = .021$). Q-Q plots, histograms, and Shapiro-Wilk tests[31] report that none of them are normally distributed. Transforming the data does not solve this problem. Therefore, we use non-parametric statistics.

A Kruskal-Wallis rank sum test shows that there is no statistical significance between WildDocs versions regarding condition-based failures ($H(3) = 5.11$, $p = .16$). This is what we expected, because of the random assignment of participants to WildDocs versions. The descriptive statistical data of all condition-based failure rates show rather low values ($Mdn = .042$, $M = .041$, $SE = .008$).

Structure-Based Problems Descriptive statistics for the rate of structure-based failures show mostly higher values than condition-based problems: *v1* ($Mdn = .14$, $M = .16$, $SE = .03$), *v2* ($Mdn = .12$, $M = .14$, $SE = .03$), *v3* ($Mdn = .08$, $M = .08$, $SE = .02$), and *v4* ($Mdn = .13$, $M = .23$, $SE = .08$). Figure 5.14 depicts boxplots of those four groups. Shapiro-Wilk tests assume normal distributions for all samples, except for *v4* ($p = .01$). We apply $\sqrt{x}$ transformations on all samples before performing statistical tests, because it shows positive

[30] Participants were asked during the introduction phase to think aloud when problems occur (see Sect. 5.2.5).
[31] The results were: *v1* ($p < .001$), *v2* ($p = .002$), *v3* ($p = .03$), and *v4* ($p = .006$).

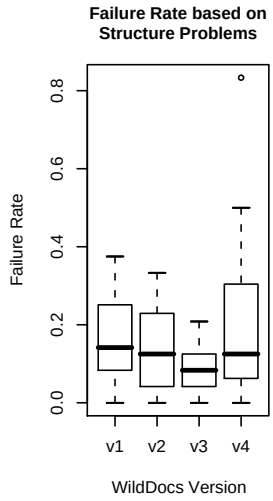


Figure 5.14.: Failure rates due to structure problems

effects on the distributions of most data sets.[32] This is confirmed by Q-Q plots and histograms.

Levene's test assumes homogeneity of variance ($F(3,41) = .96$, $p = .42$). The analysis of variance does not report a significant difference of structure-based failure rates among WildDocs versions ($F(3,41) = 1.26$, $p = .30$, $r = .29$).[33]

We were able to falsify our hypothesis. Apparently, there is no significant effect in structure-based failure rates among the versions for variable sizes (*v1*), enhanced zooming (*v2*), incidental rotation (*v3*), or fixed sizes (*v4*). However, the tendency of fewer failure rates for *v3* ($t(19) = 1.68$, $p = .11$, $r = .36$)[34] leaves the question open, if or how emerging spatial structures can be improved that the effect becomes statistically significant.

Figure 5.15 depicts a bar chart of the rate of failed questions due to structure problems. In combination with Tab. 5.3, we can see that the first six positions (*q17*, *q06*, *q08*, *q47*, *q53*, *q15*) as well as position nine (*q31*) are taken by questions that are related to "hidden" visual attributes. That means that all seven questions of this type appear within the first nine positions of structure-based failures.

5.3.4. Use of WildDocs Specific Features

This section describes how participants used specific WildDocs features. This gives some information about how comfortable they felt with them.

[32] The $\sqrt{x}$ transformation causes Shapiro-Wilk results to changed from *v1* ($p = .56$), *v2* ($p = .11$), *v3* ($p = .56$), and *v4* ($p = .01$) to *v1* ($p = .73$), *v2* ($p = .48$), *v3* ($p = .19$), and *v4* ($p = .78$).

[33] Without transformation, the effect is still not significantly differently ($F(3,41) = 1.59$, $p = .21$, $r = .32$). Also here, we assumed the homogeneity of variance ($F(3,41) = 2.01$, $p = .13$).

[34] Based on an F-test we assumed homogeneous variances ($p = .11$) and used a standard t-test.

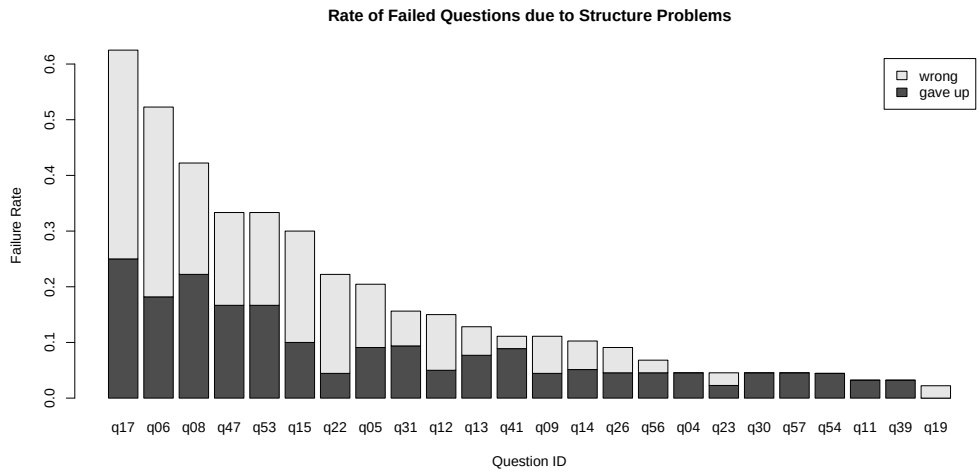


Figure 5.15.: Rate of failed questions due to structure problems

Activation of Zooming

Zooming of *v2* and the other versions can hardly be compared directly, because participants of *v2* had alternative zoom features. Therefore, we divide this section into two parts, one for zooming in *v1*, *v3*, and *v4* and another section for zooming in *v2*.

Zooming in WildDocs *v1*, *v3*, and *v4* Figure 5.16 depicts zoom menu activation counts for *v1*, *v3*, and *v4*, for both, organizing and finding phase. We counted *zoom in*, *zoom out*, and *zoom reset* individually in our log file during testing; however, we decided to evaluate them together. Any single click on those three commands at the zoom menu was counted as one zoom menu activation.

Descriptive statistics for the organization part show the following values: *v1* ($Mdn = 0$, $M = .25$, $SE = .18$), *v3* ($Mdn = 35.0$, $M = 41.6$, $SE = 10.54$), and *v4* ($Mdn = 9.0$, $M = 16.2$, $SE = 4.82$). We discovered normal distributions for the menu zoom count for *v3* ($p = .69$) and *v4* ($p = .06$), confirmed by Q-Q plots and histograms. *v1* is not normally distributed ($p < .001$), mainly because only two participants used the zoom menu while organizing: Participant *s22* used it once, *s25* twice. Because the significance to version *v4* is obvious,[35] we test only *v4* against *v3*.

The F-test assumes non-homogeneous variances. Welch's t-test reports that members of group *v3* used menu zoom significantly more often than users of *v4* ($t(12.66) = -2.19$, $p = .048$, $r = .52$). The effect size is large.

The boxplots for menu zoom activation rates[36] for the finding part look similar for *v3* ($Mdn = 2.23$, $M = 2.15$, $SE = .50$) or *v4* ($Mdn = 1.0$, $M = 1.56$, $SE = .54$); however, *v1*

[35] Eight of eleven members of group *v4* used menu zoom. The lowest menu zoom activation count among them is seven.

[36] The rates are calculated by the total number of zoom menu activations divided by *all asked* questions. Those include also questions that we have taken out later (see Sect. 5.3.1). The reason is that we did not have the possibility to isolate those menu zoom activations that were counted during a finding task that was taken out later.

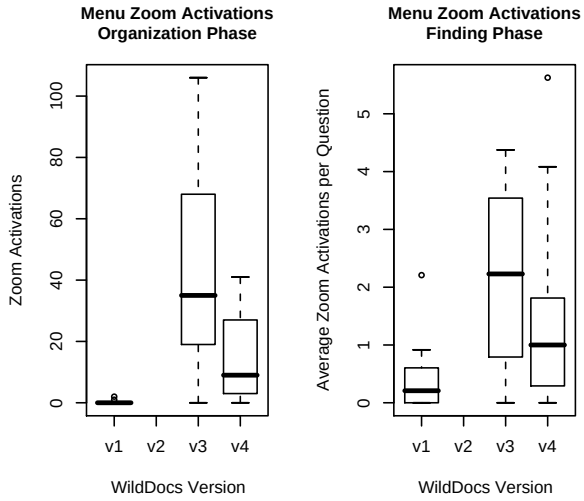


Figure 5.16.: Menu zoom activations during organization (left) and finding phase (right) in *v1*, *v3*, and *v4*

($Mdn = .21$, $M = .45$, $SE = .20$) indicates higher values relative to the other groups than it has for the organization phase. Confirmed by Shapiro-Wilk tests, Q-Q plots and histograms show that *v1* ($p = .001$) and *v4* ($p = .02$) do not have normally distributed data, whereas *v3* has ($p = .47$). A $\sqrt{x}$ transformation produces acceptable distributions for *v1* ($p = .11$), *v3* ($p = .36$), and *v4* ($p = .36$). We use the transformed data for the following tests.[37]

Levene's test assumes homogeneity of variance ($F(2,29) = .55$, $p = .58$). Analysis of variance reports a significant difference and a large with tendency to medium effect size ($F(2,29) = 4.09$, $p = .03$, $r = .47$). We apply a Bonferroni correction ($\alpha = \frac{.05}{2} = .025$) and use t-tests to compare the use of menu zoom during finding to our control group *v4*. For testing *v1* against *v4*, we used Welch's t-test, because the F-test reports a possible violation of the homogeneity of variance ($p < .001$). The result shows that apparently users of *v1* activate menu zoom significantly less than users of *v4* ($t(10.68) = 3.99$, $p = .002$, $r = .77$). The effect size is large. Variances of *v3* and *v4* are homogeneous ($p = .46$). The standard t-test shows that they are not significantly different from each other ($t(19) = -1.98$, $p = .06$, $r = .41$), even though similar to the organization part, members of group *v3* tend to use the zoom menu more often than those of *v4*. The effect size is medium with a tendency to large.

We conclude that menu zoom is used significantly less for variable size versions (*v1*) compared to fixed size (*v4*). This result applies for both organizing and finding. We expected this, because small nodes in *v1* do not force the user to zoom as much as with *v4* in order to to get a better overview or navigate conveniently. We further conclude that users of versions supporting incidental rotation (*v3*) used menu zoom significantly more often during the organization phase than the control group (*v4*). For the finding part, there is a tendency that *v3* shows a higher number of menu zoom calls than *v4*. It seems that sloppy structures, as produced with *v3*, increase the user's desire to use zooming. Also the question about a possible

[37] Boxplots in Fig. 5.16 depict data before being transformed.

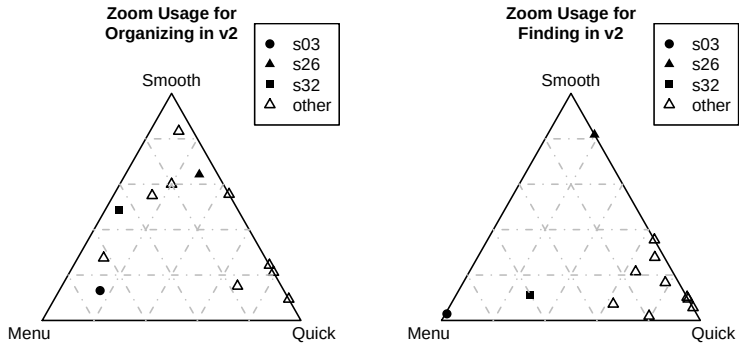


Figure 5.17.: Zoom function usage during organization (left) and finding phase (right) in *v2*

relation between the frequency of menu zoom usage and the occupied area comes in mind. We will discuss this in Sect. 5.3.5.

Zooming in WildDocs *v2* WildDocs *v2* has several zoom methods built in. It is possible to zoom via menu or keyboard shortcuts for stepwise zooming, keyboard shortcut for "quick-zoom", or right mouse button for "smooth zooming". We want to see how participants used those. We do not calculate significance, but describe tendencies, because the different zoom methods are of different type. For example, it takes one activation to zoom out completely for quickzoom, whereas it may take several when using the zoom menu. If we consider one single zoom act as moving from a "departure" to a desired "destination" scale level, the several counts of menu zoom activations compared to the single quickzoom call would be misleading. This view on zooming would cause distortions on the axes of the ternary plots in Fig. 5.17, which we will discuss in the following.

Figure 5.17 depicts the relation of menu zoom, smooth zooming, and quickzoom with regards to the number of activation. Each symbol represents a participant of group *v2*. At the organization part, four users strongly used quickzoom, three of them menu zoom not at all. A wider spread field of six participants activated mostly smooth zooming; however, most of them did not stick with it as strongly as the previous group did with quickzoom. Two participants tended to activate mostly zoom menu functions.

The situation changed for the finding part: All four participants who used mainly quick-zoom for organizing stayed with it. All other participants, except three, also moved toward quickzoom. Those three activated mainly other techniques for zooming. They are marked with special symbols in Fig. 5.17. Participant *s03* used menu zoom already to a high extend during organizing; however, for the finding part he/she decided to use it almost exclusively. Participant *s26* used mainly smooth zooming, but also quickzoom and to some extend menu zoom for finding. During the organization, he/she did not use menu zoom at all and moved slightly toward a higher rate of smooth zooming calls. Participant *s32*'s activation count for organizing was almost equal for smooth zooming and menu zoom. He/she used quickzoom very little. For the finding part, he/she changed the strategy toward activating among all zoom methods mostly menu zoom, quickzoom more often, and smooth zooming only to a small extent. Five of the participants did not call zoom menu methods at all during finding.

We can conclude that six participants changed their main zoom method between organiza-

tion and finding part. Five decided to change to mainly quickzoom, one moved away from smooth zooming toward menu zoom and quickzoom. All participants who used mainly quickzoom during the organization phase did not change their strategy for finding. The group of participants who mainly activated quickzoom grew from four at the organization part to nine at the finding part. According to the activation counts for finding, quickzoom became the most popular zoom method in WildDocs *v2*.

We further can see that four participants during organizing and five during finding did not use menu zoom at all. This includes menu zoom keyboard shortcuts (`CTRL-0`, `CTRL--`, `CTRL-=`). Menu zoom is the only zoom method that has zero counts for some sessions. Figure 5.17 also depicts that participants activated menu zoom slightly less often relative to the other methods for finding than they did during the organization part. There is also a strong tendency of users away from smooth zooming toward quickzoom.

Participants were aware of all zooming methods; however, only some were using quickzoom extensively from the very beginning, but more switched to it later during the experiment. This indicates that the participants were not used to methods such as quickzoom, even though it became popular during the test. This leads to the question about the efficiency of different zooming methods: Would quickzoom improve existing spatial-based knowledge management applications? This question becomes especially interesting, because quickzoom is not widespread in other applications. We are not aware of any application that has implemented a zoom method similar to quickzoom, that is based on the observation we made by analyzing paper work on a real desk.

Purposeful Rotation in WildDocs *v3*

WildDocs *v3* has incidental as well as purposeful rotation implemented, as explained in Sect. 4.3.1. We counted how often a participant activated the purposeful rotation feature. For the organization part, six of ten used this feature ($Mdn = 2.5$, $M = 3.0$, $SE = 1.01$). The maximum is eight counts. During the finding part, three users called rotation on an object, two of them three times, one of them one time. None of the count collections are normally distributed. This is mainly caused by only little use and a small sample size ($N = 10$).

We do not have the possibility to distinguish between intentional and unintentional activations; however, we assume that at least some of the counted rotation calls happened through accidentally double clicking a document. This assumption is based on our observations during the test phase when we did not experience many rotations that were performed by the participant. We conclude that the purposeful rotation feature was only little used by the participants.

Object Resizing in WildDocs *v1*

We counted how often participants of group *v1* clicked on bounds handles. Bounds handles are used to resize objects. The descriptive data shows that most participants did not use handles extensively during organization ($Mdn = 25.0$, $M = 34.6$, $SE = 10.58$). This changed during the finding phase ($Mdn = 9.17$, $M = 9.17$, $SE = 1.72$). The measurement for the data collected during the finding phase is based on the average handle count per question.[38] Q-Q

[38] There were 24 questions in total per participant. The total count of handle clicks for finding was between 40 and 493 ($Mdn = 220.0$, $M = 220.2$, $SE = 41.37$). Our evaluation includes data on questions that were taken out after the test (see Sect. 5.3.1 for further details). Affected are participant *s04* and *s05*, each of them with five questions.

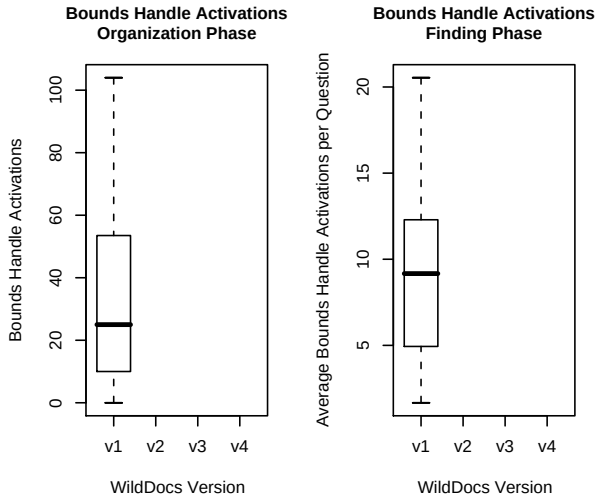


Figure 5.18.: Bounds handle usage during organization (left) and finding phase (right) in *v1*

plots as well as Shapiro-Wilk tests indicate normal distributions for counts during organization ($p = .08$) as well as during finding phase ($p = .78$).

The high number of handle counts for finding documents is mainly caused by questions that ask about a visual attribute that was outside the visual area of an object. The user had to enlarge the object in order to see the content further down. Interestingly, most participants aimed to put the object back to its original position and in its original size, even though he/she may have had to enlarge the same objects several times for solving other questions.

We conclude that participants rather use bounds handles several times on each object instead of enlarging documents once and placing them on a larger area. One reason may be that they were not used to work with full sized documents, similar to fixed size ones, on this type of application.

5.3.5. Relations

In this section we discuss relations between different pieces of data gathered in our experiment. We counted feature activations during organization as well as during finding phases. We convert all feature counts for finding to the average counts per asked question. Because the counters include also feature activations for working on questions that were taken out later, we consider *all* questions, excluding skipped ones, for calculating the average count per asked question. A detailed discussion of skipped and taken out questions can be found in Sect. 5.3.1.

Relations of Special Features to Time

We investigate in finding relations between the time spent for organizing or finding objects and the following version specific features: *Bounds handle usage* for *v1*, *quickzoom* or *smooth*

The reason is that we calculate the average handle activation count per questions. This includes the one that were taken out later. We do not have the possibility to subtract the handle count performed for those five questions.

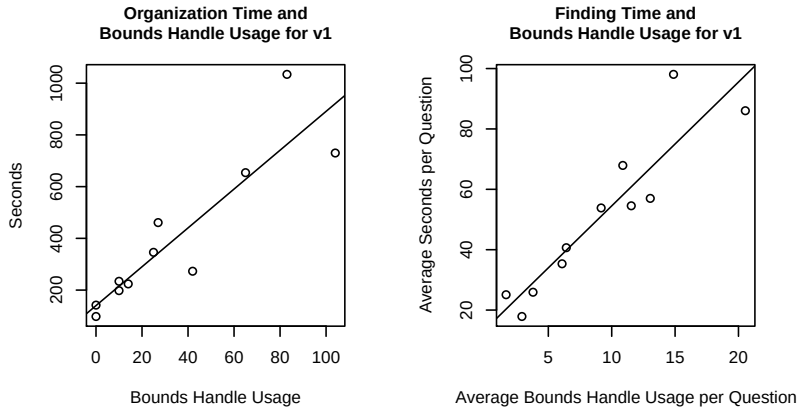


Figure 5.19.: Relation of bounds handle usage to organizing (left) and finding time (right) in *v1*

zooming usage for *v2*, or *zoom menu* usage for *v1*, *v3*, and *v4*. Scatter plots as well as evaluations of F-statistics and R^2 let us assume that there is no linear relationship among them,[39] except for bounds handle usage to time.

Figure 5.19 suggests a linear relation between bounds handle activations and time spent for organizing ($F(1 \text{ and } 9) = 38.84$, $p < .001$, $R^2 = .81$) as well as for bounds handle usage and the average time spent for working on a question ($F(1 \text{ and } 9) = 47.77$, $p < .001$, $R^2 = .84$). A simple linear regression $y = \alpha + \beta x$ with $\alpha = 140.08$, $\beta = 7.51$ for organizing and $\alpha = 13.48$, $\beta = 4.10$ for the average finding time per question describes the relations. This means that one bounds handle activation costs approximately 7.5 seconds for organizing or approximately 4.1 seconds while working on one question. This is not to be seen in isolation. There are other related effects, such as time for reading the content of the resized object, etc. However, it points out that resizing is expensive in time.

A related question is whether the failure rate of questions for the finding part is related to the number of bounds handle activations. An useful result would only contain structure-based failures. However, our data is based on the total count of bounds handle clicks. There is no possibility to distinguish clicks that were performed for searches failed by structure and those failed by condition problems. Therefore, we did not investigate in this question.

Relations to Occupied Area

We assumed that there would be a relation of occupied area measured after the organization phase to *organization time* ($F(1 \text{ and } 42) = 15.04$, $p < .001$, $R^2 = .26$), *spent time for correctly finding documents* ($F(1 \text{ and } 42) = 7.52$, $p = .009$, $R^2 = .15$), or the *rate of structure-based*

[39] The results for F-statistics and R^2 are the following for the organization time related features: quickzoom for *v2* ($F(1 \text{ and } 10) = .10$, $p = .75$, $R^2 = .01$); smooth zooming for *v2* ($F(1 \text{ and } 10) = .03$, $p = .87$, $R^2 = .003$); menu zoom for *v1* ($F(1 \text{ and } 10) = .48$, $p = .51$, $R^2 = .05$), *v3* ($F(1 \text{ and } 8) = .40$, $p = .54$, $R^2 = .05$), and *v4* ($F(1 \text{ and } 9) = .62$, $p = .45$, $R^2 = .06$). The results for F-statistics and R^2 of the average time of working on a document in relation to the following features are: quickzoom for *v2* ($F(1 \text{ and } 10) = .01$, $p = .92$, $R^2 = .001$); smooth zooming for *v2* ($F(1 \text{ and } 10) = .23$, $p = .64$, $R^2 = .02$); menu zoom for *v1* ($F(1 \text{ and } 9) = 8.30$, $p = .02$, $R^2 = .48$), *v3* ($F(1 \text{ and } 8) = 9.81$, $p = .01$, $R^2 = .55$), and *v4* ($F(1 \text{ and } 9) = .15$, $p = .71$, $R^2 = .02$).

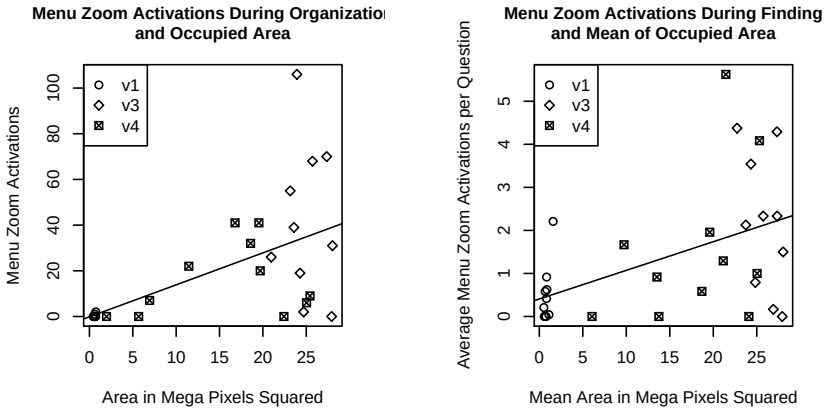


Figure 5.20.: Relation of occupied area to zoom menu activations during organization (left) and finding phase (right) for *v1*, *v3*, and *v4*

failures ($F(1 \text{ and } 42) = 9.33$, $p = .004$, $R^2 = .18$). However, scatter plots as well as evaluation of linear models do not show a positive result.

Now, we look into the relation of occupied area after the organization phase and the number of menu zoom activations. Figure 5.20 shows scatter plots for the organization and finding phase. For the evaluation of the finding phase, we use the mean of the occupied area at the beginning[40] and at the end of the finding part. The underlying data comes from *v1*, *v3*, and *v4* and is calculated as average zoom menu count per asked question.[41] We do not use *v2*, as discussed in Sect. 5.3.4.

We apply a linear model for the used area and zoom menu activations captured during the organization phase. The result reports significance; however, R^2 is weak ($F(1 \text{ and } 30) = 16.41$, $p < .001$, $R^2 = .35$). There is a similar result for the area and its relation to the number of zoom menu calls during the finding phase ($F(1 \text{ and } 29) = 8.55$, $p = .007$, $R^2 = .23$). The simple linear regression model $y = \alpha + \beta x$ fits for the organization part with $\alpha = -.14$, $\beta = 1.40$ and for the finding phase with $\alpha = 0.41$, $\beta = 0.07$.

We conclude that even though there is no tight linear relationship between menu zoom activations, there is an overall tendency that there is a higher frequency of menu zoom calls for larger areas. This explains why menu zoom calls as well as the occupied area after the organization part for *v3* are significantly or at least tendentiously different to those for *v4*.[42] Apparently, zooming becomes more important to most users when dealing with larger areas. Our tests discussed in Sect. 5.3.2 have shown that fixed size documents need significantly more space than variable sized objects. Therefore, zooming becomes especially important for fixed size versions.

[40] That is the occupied area at the end of the organization phase.

[41] Asked questions include those that were taken out later.

[42] See discussion in Sect. 5.3.2 for further details about the occupied area or Sect. 5.3.4 for further information about the menu zoom usage.

Test	*v1*	*v2*	*v3*	Focus
Time for organizing	▼	▽	□	time used
Time for organizing per object	▽	▽	□	time used
Occupied area after organization	▼	□	▲	occupied area
Average occupied area after organization per object	▼	□	▲	occupied area
Correctly found docs with “obvious” visual attributes	▼	▼	□	time used
Correctly found docs with “hidden” visual attributes	▲	▽	□	time used
Incorrect answers, based on condition problems	◇	◇	◇	rate
Incorrect answers, based on structure problems	□	□	▽	rate
Subjective satisfaction	◇	◇	◇	users’ rating
Use of menu zoom (organization)	▼	–	▲	activation count
Use of menu zoom (finding)	▼	–	△	activation count

▼ = significantly less; ▲ = significantly more; ▽ = tendentiously less; △ = tendentiously more;
□ = no obvious tendency; ◇ = not significant, no or unknown tendency; – = not part of comparison

Table 5.4.: Summary of statistical tests compared to WildDocs *v4*

5.3.6. Participants’ Ratings

We asked participants after the test to rate the application on a scale of whole numbers from one to five. One is the worst rating and five is the best. The independent variable is the WildDocs version; the dependent variable is the rating given by the participants. Shapiro-Wilk tests confirmed by histograms show that except for *v4* ($p = .18$), none of the data sets is normally distributed: *v1* ($p = .004$), *v2* ($p = .003$), and *v3* ($p < .001$). Q-Q plots show discrete distributions. Transformation does not solve the problem. Therefore, we used non-parametric statistics for evaluation. A Kruskal-Wallis test reports no significant effects among the four groups ($H(3) = 4.49$, $p = 0.21$). Apparently, most users tended to like the application ($Mdn = 4.0$, $M = 3.73$, $SE = .11$).

5.4. Summary and Conclusion on Statistical Results

Table 5.4 gives a summary of the previously discussed results. Effects are differently marked for significantly less (▼) and significantly more (▲) in comparison to the control group *v4*. Non-significant results are divided into those that show a tendency[43] of less (▽) or more (△) in comparison to *v4*, those that do not show a tendency (□), and finally those that may have a tendency (◇). The result of the latter group was detected by Kruskal-Wallis tests on condition-based incorrect answers and subjective ratings. However, we did not apply individual comparisons to the control group, because it is of no interest to us for condition-based incorrect answers and difficult to see tendencies on the discrete distribution of participants’ ratings. As discussed in Sect. 5.3.4, *v2* was not tested against *v4* for zooming activations. This is marked in the table as “–”.

The results show that *v1* has more in common with *v2* than with *v3*. The effects of *v2* are often in between *v1* and *v3*. Whereas members of group *v1* were significantly faster in

[43] The reason why we point out tendencies for non-significant results is to mark those hypotheses that are still interesting for future research investigations.

organizing objects on the screen, those of group *v2* showed a tendency to be faster than *v4*. The evaluation of the occupied area shows that both, *v1* and *v3*, significantly differ from *v4*, but in opposite directions. Similar results exist for menu zoom usage, even though for the finding part, *v3* members used menu zoom only tendentiously more often. We calculated linear models that report that menu zoom tendentiously is more often used when objects occupy a larger area.

There are different effects for documents with "obvious" versus those with "hidden" visual attributes. For the "obvious" ones, it took significantly less time for *v1* and *v2*. For the "hidden" ones, it took significantly more time for *v1*, whereas the results on *v2* show at least a tendency to be still faster than *v4*. There is no effect for *v3*.

The rates of incorrect answers do not show significant differences; however, the evaluation of *v3* reports a tendency to have a lower structure-based failure rate. There is also no significance among the participants' ratings. As mentioned above, we did not explore tendencies for condition-based failures or ratings.

Based on the statistical results, we have learned that variable size objects (*v1*) help to organize information quickly as well as to find them correctly, as long as the asked visual attribute is "obvious", that is basically, visible on the screen. It is not a good tool for browsing documents looking for "hidden" visual attributes. It turned out that the fixed size feature (*v4*) supports the knowledge worker better in this context.

Tests on zooming have shown that *v2* tends to reduce the time for organization as well as the time for finding documents based on "hidden" visual attributes. Similar to *v1*, *v2* is also significantly faster for finding "obvious" attributes. In general, the tests report better results for zooming than for variable size features.

Rotation (*v3*) does not show many significant effects compared to *v4*. However, participants of this group used a larger area. They also zoomed more often, significantly for the organization part and tendentiously for finding. In our experiment, rotation has no significant positive effects for knowledge workers. However, it could be shown to improve the user's workflow, if the tendency of a lower rate of structure-based problems can be made significant, possibly via minor improvements on WildDocs *v3* or a higher number of participants.

We tested WildDocs with respect to well defined problem statements. The results do not tell how WildDocs features would affect current spatial hypertext applications, such as VKB or Tinderbox, because the novel features would have to be seen in *combination* with existing ones, such as search facilities or agents. However, our statistical evaluation shows that some of the selected, isolated aspects have an effect. This makes it valuable and interesting to test those in combination against supported features of existing applications.

Chapter 6.

Summary, Future Work, and Conclusion

> "It is quite nice. It is easy to learn and to use. It is a strong tool to organize documents."
>
> *(Participant s21 about WildDocs)*

6.1. Summary

6.1.1. Statistical Evaluation

We had some assumptions about the outcome of the experiment (see Tab. 3.1 on page 73). However, the results, shown in Tab. 5.4 on page 208, turned out slightly different. We expected the time used for organizing or finding to be significantly faster with zooming enabled (*v2*). The only case where we found this to be true was for correctly found documents with "obvious" visual attributes. For all other organizing or finding cases there was no significance, even though our test reported a tendency of being faster.

We have to differentiate also for variable size support (*v1*). Our assumption of significantly faster organizing turned out to be correct when taking the total amount of objects into consideration. The time for finding differs among "obvious" and "hidden" visual attributes. Documents with "hidden" attributes fulfilled our expectation of being significantly slower, whereas such with "obvious" attributes were found significantly more quickly. We learned that variably resizable small nodes support finding only for nodes with "obvious" visual attributes.

We did not expect incidental rotation or sloppiness (*v3*) to affect significantly time spent for organizing. We expected the time for finding information to be significantly lower than without sloppiness, but this turned out not to be true. However, we experienced the tendency of fewer incorrect answers based on structure problems.

Our assumption that *v1* would have a significantly smaller occupied area and *v2* would have no significant difference in this regard were met. However, we were surprised that emerging sloppiness turned out to occupy significantly larger areas. We have learned that more space is needed when spatial structures emerge sloppily.

We further expected to have no significance in the number of zoom calls for *v3*; however, users activated zoom methods significantly more often for organizing and tendentiously more often for finding. Apparently, zooming is more important for users when dealing with sloppy spatial structures. Our assumption of experiencing significantly less zooming for *v1* was met. This shows that zooming for members of *v1* was not as important as for the members of the control group. We further found out that there is an overall tendency that larger occupied spaces draw more zoom method calls and vice versa.

We assumed that members of *v2* would be significantly higher satisfied with the application. However, there was no significance among the different groups.

The usability test delivered results for short term use scenarios. However, we assume to have more implicit metainformation gathered over time. Therefore, long term tests are likely to report different results.

6.1.2. Real World Observations

We have analyzed real world paper structures including bindings. The most important aspect we discovered are emerging effects of spatial structure during or after creation. We argued that this type of metainformation is attached automatically and without additional effort, for example, sloppiness of piles, paper with a tinge of yellow through sunlight, etc.

We further argued that there are various limitations that are based on laws of nature. One example is space. Humans perceive any point in space to be taken only by one item. For example, the space that a book takes cannot be taken by another book at the same time. Another example is the inside of a box, which cannot be larger than the box's outside dimensions. Other laws of nature include physical forces, such as gravity, friction, or inertia. Those limitations affect the creation of spatial structures with paper to a high degree. Humans have an understanding or feeling of how those forces act on items.

Limitations or physical forces lead to complex behavior. We isolated two important aspects: Auto conversion and structure dissolution. Auto conversion happens when a structure is converted to another one when added. For example, a heap that is added to a tray becomes a stack, because the documents need to have a neat shape in order to fit into the tray. Structure dissolution happens when a structure disappears and gives its contained documents to the other structure. For example, open placed objects that are put onto a heap will not be perceived as pages that belong together, but rather directly as child documents of the heap. We also pointed out specific behavior of structures' thickness growth.

We described constraints for adding bindings to other bindings. Only certain combinations are possible, as shown in Tab. 2.3 on page 46. Partly, this depends on structure conversion, structure dissolution, or space-related constraints, such as thickness growth.

We explained the idea of structure that is pushed to its next structure level. An example is depicted in Fig. 2.11 on page 47. Mainly, this becomes necessary because of physical limitations. People came up with a variety of devices or add-ons that can be used to group or bind documents, such as binders, or marks or inscriptions *on* binders.

We further developed a classification for bindings. Table 2.1 on page 40 shows an overview. We used this taxonomy to describe various structure or binding types that we discovered in offices. Furthermore, we developed the concept of *binding mechanism* and created a classification for the described bindings. Those classifications are necessary for building application classes efficiently.

We argue that the described paper structures or binding types are only a small snapshot of a large variety. Different types are used in different cultures, times, or locations. For example, we mentioned scripture roles as a binding type that is not used frequently in the western world today.

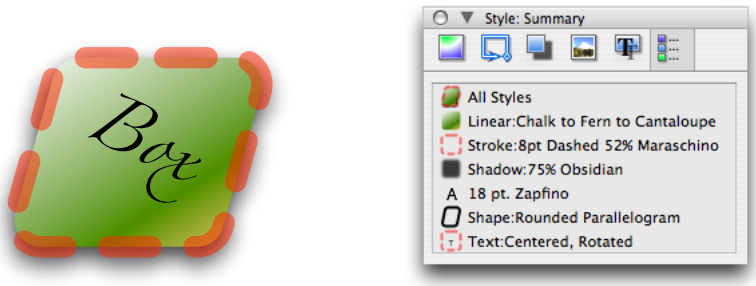


Figure 6.1.: OmniGraffle object (left) and style summary inspector window with the object's visual attributes listed (right)

6.1.3. Application Analysis

The summary in the previous section points out the variety and complexity of real paper structures or bindings, as well as the ability of office workers to deal with that through experience. In this thesis, we compared some aspects to spatial structure supporting applications, especially to spatial hypertext applications.

We argued that those applications are based on metaphors, for example, cards on a table, but their implementations are at high abstraction of levels. Most real world behavior is either mostly or completely unimplemented, such as natural-like emerging behavior or sloppiness, physical forces, or realistic bindings. The latter includes limitations, such as fixed node, desk, or collection size.

Another discussed issue is zooming. People in the real world change their focus constantly, for example, browsing a book shelf, picking a book, and finally browsing its pages. We argue that smooth zooming would simulate this most realistically among other focus–context supporting interactions. However, today's 2D spatial hypertext applications mostly support only stepwise zooming. OmniGraffle, a chart application which was "misused" for the purpose of our research as a "spatial hypertext application" had the most advanced zooming support among the analyzed applications. Our implementation of smooth zooming or quickzoom in WildDocs has shown positive effects in our usability test.

Essential for spatial hypertext applications beside spatial arrangement are visual cues. Those include color, shape, or border (color, width, or stroke). However, VKB and Tinderbox support only basic types. For instance, the only supported shape for nodes is rectangular. Only some border settings are available in VKB; none are available in Tinderbox.

On the other side, OmniGraffle supports a variety of visual cues for objects and therefore offers better graphical and visual support. Figure 6.1 depicts an example. Visual objects with on or more of those visual cues that are used as data variables are called *glyphs* (see Ware, 2004, 176–185).

6.2. Future Work

6.2.1. Open Questions

We discovered many questions that remain open for future work. One main topic concerns bindings and how they would affect users in organizing or finding information. Would bindings reduce the time for organizing fixed size documents? There are also implementation issues on how emerging behavior or physical forces can be implemented and supported by the system.

Other open questions include rotation or zooming. We found a tendency of fewer incorrect answers in relation to incidental rotation (*v3*). It is still open for further investigation. For example, how can we improve existing incidental rotation or sloppiness to support the user? This is closely related to questions about general limitation or emerging metainformation, such as whether limited space in binders would support knowledge workers. Since emerging metainformation is created over time, the appropriate approach would include long term tests.

We implemented different zooming methods in WildDocs (smooth zooming, quickzoom, and menu zoom). There was a tendency among those users who had the choice between them (i. e., *v2*) toward the use of quickzoom for the finding part (see Fig. 5.17 on page 203). However, it is still an open question as to whether positive or negative aspects can be found among the different variants with respect to the situation they are used in. How would people use them? Which method would show a significant effect for organizing or finding time?

We argue that there is a large difference between VKB or Tinderbox on the one side and OmniGraffle on the other regarding the variety of visual cues for objects. It is open for further discussion which and how many visual cues are optimal to use for spatial knowledge structure. A relevant issue is the limitation of visual working memory capacity (Ware, 2004, 355). It is for future work to discover this aspect with respect to spatial hypertext applications.

An interesting question is how WildDocs could be used in international context. How do offices look in other cultures? Which paper structures or workflows can be observed? How is the understanding of bindings to people in other cultures? Answers to those questions would be relevant to see the need for specific bindings and to implement them in WildDocs.

6.2.2. Improving WildDocs

WildDocs is a prototypic implementation. In Chap. 4 we pointed several times to potential improvements or bugs that are to be addressed in the future.

WildDocs follows the philosophy of managing many documents on different levels. Especially smooth zooming on a space with many objects causes heavy load on the machine. Beside appropriate hardware, also software issues play a role. Beside more efficient frameworks, internal improvements to reach better efficiency are needed. In the analysis phase of available frameworks or components, we discussed writing the application on Mac OS X using Objective-C as the main programming language and Quartz (Apple Computer, 2006) for 2D or OpenGL (Apple Computer, 2005b) for 3D rendering (as opposed to Java, which we used). This may be an option in the future.

There are problems that are based on the underlying framework Piccolo, such as index pusher limits (see Figures on page 145 or 147). Other important features, such as text piping among different documents, extended file import (e. g., native PDF), or more realistic shadows

Figure 6.2.: Inertia/friction simulation in DynaWall demonstrating by "pushing" a document, which moves to the other side of the display – pictures taken from the Roomware demonstration movie (Streitz et al., 2002b), used with permission

are essential for productive long term use of WildDocs. Additional features, such as support for annotation or visual cues would be desirable.

Even though a spin-off of our work, bindings play an important role. We implemented basic parts. However, as mentioned in Chap. 4, some more coding needs to be done in order to have them ready to be used.

6.2.3. Extending WildDocs

Paper-like Movement

Beside the discussed features (zooming, rotation, and fixed size documents), also physical forces could be implemented in WildDocs, such as gravity, friction, or inertia. They would support emerging structures without causing unexpected behavior.[1]

An example of the implementation of physical forces is DynaWall (Streitz et al., 2002a), "an interactive electronic wall, representing a touch-sensitive vertical information display and interaction device that is 4.50 m wide and 1.10 m high" (Streitz et al., 2002a, 1). Figure 6.2 shows pictures of the installation. Another example is BumpTop (Agarawala & Balakrishnan, 2006), a prototypic application that simulates the effects of friction, gravity, and inertia on icons placed in a 3D environment.

Caused by its width, it may become necessary to to drag a document several meters. However, this would be inconvenient. Therefore, DynaWall simulates what would be experienced in the real world as a mix of inertia and friction. The users can push documents into a certain direction. Even though the object is released, it still moves further. The above mentioned figure depicts a sequence of pictures that represent the movement of a pushed object.

This is an example for partly implemented real world forces. WildDocs would use those forces to support emerging structures with simulations of which users are aware of due to their world knowledge.

Mixed Environments, Large Displays, and Multi-User

WildDocs could be implemented in mixed environments, such as described in Sect. 2.3.1 (e. g., Wellner, 1993; Ashdown, 2004). Projections of digital paper on the desk would show similar behavior to that of real paper. The interesting question in this context is what structures

[1] Unexpected behavior may be caused by users who may not expect a computer to simulate physical forces.

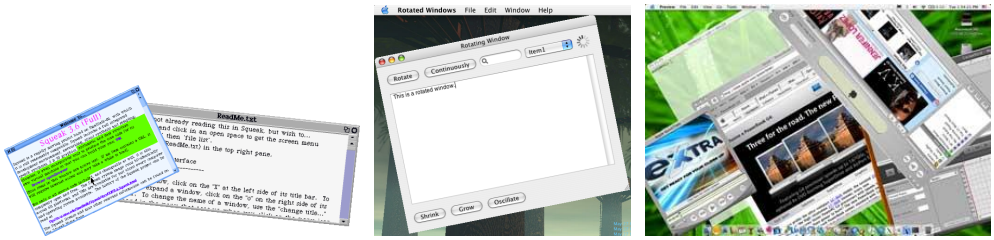


Figure 6.3.: Two windows in Squeak (left), rotated Wild Windows window on Mac OS X (center), and the Perturbed Desktop on Mac OS X (right) – third screenshot taken from Singh (2005, Fig. 4)

people would create and which metainformation could be gained from applied real world behavior. The same question appears for large touchscreen.

Furthermore, we are interested in finding out how emerging structures affect collaborations. Our usability test did not include the exchange of information among participants. It is unknown how WildDocs spatial structures would be perceived and interpreted among members of a group of people. A special question arises about the relevance of cultural background especially with respect to sloppiness and the used binding types. All people have experiences with the same physical forces, but they may have different preferences about disorder of documents on a desk.

Desktop and Window Manager

Another idea that we discussed in the beginning of our research was to add rotation, sloppiness, and zooming to the computer desktop and application windows. An essential feature is rotation or zooming of individual windows and keeping them fully functional. Some environments are already capable of doing this. For example Squeak, a Smalltalk implementation, can rotate or scale windows individually. The left screenshot in Fig. 6.3 depicts two windows in Smalltalk. Both are rotated, the left one is additionally scaled down. Both windows are fully functional.

Our main development was on Mac OS X. Therefore, we took a first look at possibilities on this machine. We initiated a prototype for turning and scaling an individual window, named "Wild Windows"[2] (Atzenbeck, 2005). This prototype was a proof of concept to show the possibility of scaling and rotating windows on Mac OS X. The second screenshot in Fig. 6.3 depicts the application's window. It could rotate stepwise or continuously, shrink, grow, or oscillate. The window was mostly functional, even while rotating or oscillating.[3] The implementation is in Objective-C and uses unofficial Apple APIs.

Another example of rotated windows on Mac OS X is depicted at the right screenshot in Fig. 6.3. It shows the Perturbed Desktop (Singh, 2005). It uses Apple's Sudden Motion Sensor (Apple Computer, 2005a). The Sudden Motion Sensor is a hard drive protection for some of Apple's mobile computers that aims to parks the hard drive head on severe vibra-

[2] We would like to thank Wade Tregaskis for his effort in coding the prototype.

[3] Code for translating mouse events was implemented, but the underlying APIs (particularly relating to drawing) could not easily be modified. This left many actions not supported, such as opening of pop-up menus.

tions, for example, when the computer is accidentally dropped. However, it also can be used to detect the computer's orientation. A triplet $\{x, y, z\}$ symbolizes the inclination left/right, back/forward, and the angle of the machine's bottom relative to the ground (Singh, 2005).

Windows on the Perturbed Desktop rotate in dependency of the computer's physical movement. "The perturbed desktop starts life normally, but soon gets very ... *perturbed*. The orientation of on-screen windows is set to be a strangely complicated function of several things: the physical orientation of the PowerBook[4], the amount of resources consumed by the application, and how much the user is using the application" (Singh, 2005). All windows are functional, independent of their orientation.

These examples have shown that a WildDocs desktop or window manager version respectively with emerging sloppiness and rapid zooming would be possible. Additionally, instead of icons, the application windows (i. e., document) would exist. Their size would be a matter of scale. Bindings could collect any windows.

Currently, there are no plans for WildDocs to support 3D environments, because apparently "3D effects make no difference to the effectiveness of spatial memory in monocular static displays" (Cockburn, 2004, 30). However, it is supposed to have adornment-like visualization of 3D interaction implemented, such as turning a page. 3D window managers, such as described in van Dantzich et al. (1999) or Robertson et al. (2000), or semi-3D window managers, such as Looking Glass[5] (Kawahara et al., 2004) are related examples.

The research group around Jeff Raskin had a similar idea about having all documents open and visible on a virtual desk, using zooming to bring a document closer or to put it further away:

> "We believe that Archy[6] is capable of doing everything that now requires applications scattered about your desktop and program files. Archy contains only your content, and the commands that modify your content – no desktop, no applications, no files. In the future, you will zoom in to the content you wish to modify, and then modify it directly. No launching applications, no opening files, and only one set of principles governing every operation." (Raskin Center, 2006)

6.2.4. Input Device

WildDocs supports a 2D space. Users can navigate freely in any direction. It also provides rapid zooming. Furthermore, documents can be moved, and in a future version possibly edited.

Today's common input devices, such as mouse, keyboard, graphic table, or trackpad do not support WildDocs features well. Therefore, we designed a special input device. We first published our idea in Atzenbeck & Nürnberg (2005c, 292). Figure 6.4 depicts a sketch.

The output device is also used as input device. It is a large touchscreen. Users can directly move or annotate documents. By using a touchscreen we avoid to split input location (e. g., mouse) and output location (i. e., screen), and provide a more natural way of working.

[4] PowerBook is the name of a series of portable computers manufactured by Apple Computer, Inc. – Author's note.

[5] See http://www.sun.com/software/looking_glass/ for the project site or https://lg3d-core.dev.java.net for the developer site (visited on 2006-03-10).

[6] Archy is an extensible text editor. It is designed to support also non-text media, such as images, but has currently no implementation for it. – Author's note.

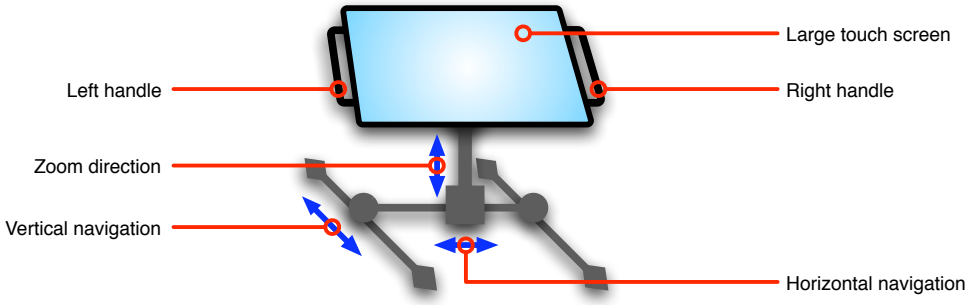


Figure 6.4.: Sketch of an input device for WildDocs

One issue is that many touchscreens are restricted to one single input location, that is, the cursor. WildDocs needs to be able to distinguish whether the user wants to move, rotate, or annotate a document. Explicitly changing modes is unnatural. Therefore, we support the idea of implicitly changing modes by either the input device or dual touch recognition.

A user's intention to annotate can be easily recognized by the system, because a special pen is only used for annotations. The hardware must be able to distinguish between the pen input and others, for example, a finger. Moving a document would be performed by using one or more fingers or the complete hand. However, the screen has to be capable of detecting multiple spots. WildDocs would then calculate the offset as well as the rotation based on the given input. For example, documents can be rotated, if the system tracks the movement at least two input coordinates. One single finger may not give enough data for that. There is research done in the development and usage of two or more input coordinates on touch screens (Wu & Balakrishnan, 2003; Rekimoto, 2002; Yee, 2004; Matsushita et al., 2000). Alternatively, also pens could be used that can express rotation, such as the Intuos3 Art Marker, a recently developed pen that is rotation-sensitive at 360 degrees.[7]

Beside a given manual rotation or offset, the movement would also depend on additional behavior, such as simulation of friction or gravity. For example, a movement similar to the one shown in Fig. 2.15 on page 52 would be possible.

Navigation on the pane is supported by the input device itself. Figure 6.4 depicts two handles on both sides of the monitor. The user can grab one and move the monitor freely along x or y directions. The other hand can be used for annotations on documents or for moving documents on the screen. The navigation appears similar to users as if they would have a frame in their hands that is moved on top of a desk: They could see part of the desk through the frame. By moving it further to the right, the viewport would also move to the right. We believe that this input device would support the user by its intuitive and realistic behavior for navigation.

The remaining feature is zooming. People get closer toward an object if they want to see it in greater detail. We transferred this idea to our input device. Using one of the handles, the user can press the screen (possibly perpendicularly) toward the virtual desk. This will result in getting closer to the scene (i. e., zooming in). Pulling the screen toward the user's head will

[7] The Intuos3 Art Marker product page is available at http://www.wacom-europe.com/int/products/intuos/input.asp?lang=en&pdx=44 (visited on 2006-04-04).

result in zooming out. The mechanical machinery needs to be easy to press or pull, but also must have some resistance. For example, it should not be pressed down by the weight of a user's hand, for example, when annotating. It must stay at the position where a user stopped pressing or pulling the monitor.

We summarize that the designed input device is based on natural behavior and ease of use. Modifications on structures can be performed easily by using fingers or a hand on the touch-screen. Annotations can be written with an appropriate pen. Navigation and zooming are supported by an underlying mechanical part. The design allows rapid and natural navigation and zooming, both applicable synchronously with one hand, possibly even during annotating or moving documents with the other.

6.2.5. Integration in Structural Computing Environments

Currently, WildDocs is a monolithic system. There are no defined interfaces of its services visible from the outside. However, internally it aims to divide structure (bindings), data (low level documents), and behavior (machines). This idea is taken from structural computing, a research branch that was born within the hypertext community and stated originally the "*primacy* of structure over data in computing" (Nürnberg et al., 1997, 96). Later, this view was broadened by realizing that "system infrastructure must assume that any entity is at all times (at least potentially) both *structured* and *structuring*" (Nürnberg et al., 2004a, 2).

This demands different services for WildDocs. Component-based architectures are built to offer support for this. Examples include Construct, which provides multiple open middleware services (Wiil et al., 2001) and special support for service developers (Wiil et al., 2000), or Callimachus (Tzagarakis et al., 1999), which provides a P2P architecture for services (Tzagarakis et al., 2000) and support for structuring primitives (Tzagarakis et al., 2003).

WildDocs's bindings implementation follows the idea of being structured and structuring. Even though it supports "data" (instances of WDLowLevelDoc) at the very end, the application is made to replace them with bindings (i. e., structure). This compares to the EAD model[8], which provides support for elucidating or hiding structure parts (Nürnberg et al., 2004a, 2005). Its development is based on the criticism of node–link structures, which have the focus on nodes. EAD focuses exclusively on structure.

In a component-based system, WildDocs would consist of several services. There would be a structure server that handles spatial structure. Because of the complexity of bindings (beside arguments of reuse), we tend to support the idea of a binding structure server that includes also binding related behavior, such as opening binding mechanisms. Another service could be a spatial parser, which generically can be interpreted as an information retrieval system that makes implicit knowledge explicit.

Distributed architectures challenge latency. This becomes especially important in environments that require immediate feedback. Some behavior, such as incidental rotation *while* a document is moved, needs to be calculated in real time. Therefore, we argue that the behavior has to be processed at the client side. However, conceptually spoken, the behavior should be handled by a server.

A possible solution would be to send a behavior processor to the client at initialization time. This module would be able to parse behavior patterns that are handled by a behavior

[8]The EAD model is "named for the three most important operations exposed by the interface to the model: elucidate; analogize; and, delete." (Nürnberg et al., 2004b, 241)

or structure server. The patterns would hold information about how behavior is applied to objects. In order to provide this, a model that can be used to describe all possible required behavior is needed.

For example, when a document movement is initialized that requires incidental rotation during movement, the client would request the behavior pattern to be parsed at the behavior processor at the client side, which extracts the information and instantiates a machine that handles the behavior in real time. Caching may help to provide better overall performance. The results would then be sent back to the structure service.

This architecture leads to problems that have to be solved. One is keeping consistency, which includes locking parts while the structure is modified. The question of which parts are relevant for the behavior arises. Another issue is history support. The question about whether behavior should be part of tracking changes is to be answered. For example, behavior may include a random factor, such as incidental rotation while moving.

6.2.6. Summary

Beside potential future work on improving the existing WildDocs application, we proposed projects that go beyond what it can provide today. This include implementation of document behavior based on physical forces. This might lead to better simulations of emerging spatial structures that could be interpreted by the users' world knowledge.

Furthermore, WildDocs could work as enhanced computer desktop and window manager replacement, providing support for zooming, emerging sloppiness, or fixed size documents. Icons would be replaced by windows to where the user can zoom. WildDocs also could be used for mixed environments that support projections of digital documents on a real desk. It would provide emerging metainformation by applying natural behavior on the digital part of the installation.

Caused by special needs for rapid zooming and navigation on a 2D space that has the size of a real desk, we designed an input device that can be used for zooming, navigation, and object modification at ease. This is an alternative to mixed environments and closely based on natural behavior.

Finally, the question about the underlying architecture is important to answer. Currently, WildDocs is implemented as a monolithic single user application. However, cutting edge research on component-based architectures provides solutions to extend WildDocs to an extensible and scalable and open application. It could provide information spaces of medium or large scale in multi-user environments.

6.3. Conclusion

Shipman (2001) points to seven directions for future spatial hypertext research. One is to "determine good design practices for spatial hypertext" (Shipman, 2001, 4). In this thesis we took one step back by discussing how spatial structure may look like. This may be the "eighth research direction" of spatial hypertext. We explained simulating behavior and richness of physical documents on spatial-based knowledge management applications, having in mind that users' world knowledge helps understanding them. Even though we are aware that our description is an abstraction of the physical world itself, it provides a step toward a system

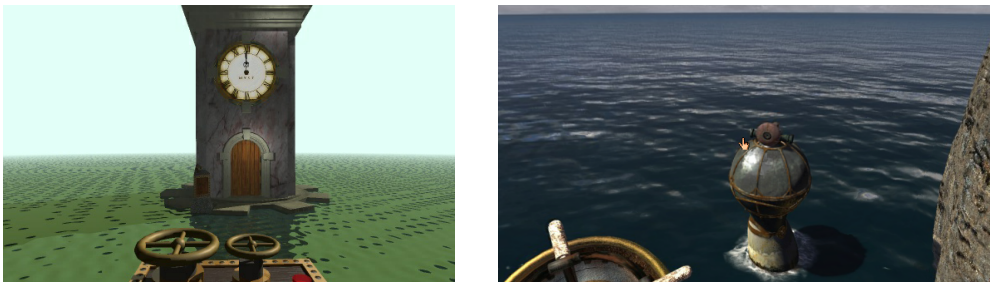


Figure 6.5.: Screenshots of Myst (left) and Myst III: Exile (right)

with more "natural" behavior. We have implemented WildDocs to see if we could experience effects of "naturally" behaving virtual documents.

We do not claim that mimicking the physical world in metaphor-driven applications is positive per se. Instead, we claim that we need to analyze properties of physical objects implemented in applications, in order to see which ones show positive and which ones negative effects. This is supported by the fact that paper is a very old medium compared to computers and therefore may be better understood by people.

However, many applications used for office work are based on metaphors, but implement a high level of abstraction which causes many metaphor breaks. These were criticized already two decades ago (Halasz & Moran, 1982). On the other side, "the use of metaphor ... can also cripple the interface with irrelevant limitations and blind the designer to new paradigms more appropriate for a computer-based application" (Gentner & Nielsen, 1996, 72).

A potential reason for many metaphor breaks or highly abstract presentations or behavior may be found in the fact that computers did not have the capacity to display and offer highly realistic interaction until recently. This becomes obviously in the development of computer games. Compared to games a decade ago, current releases offer highly realistic graphics and movements. Figure 6.5 depicts two examples: Myst, a graphic adventure computer game, was first released in 1993 (Wikipedia, 2006a). The user interaction for movement reminds of an interactive slide show. The player can move to the next scene by clicking on the desired direction on the current picture shown. The first screenshot at the mentioned figure depicts such a scene. The picture is static. The water does not look realistically. The next version of Myst had improved graphics. Its sequel (Myst III: Exile, released 2001) introduced 360 degree scenes and moving objects. For example, the second screenshot depicts a scene next to the sea. The realistic look is improved by showing the waves moving, including naturally looking reflections in real time. The final version of Myst (Myst V: End of Ages, released 2005) continues the realistic look and feel, but allows also moving around freely in real time.

Games show that realistic look and feel are possible already on today's computers. Now we need to find out which simulations of real world properties would have positive effects in computer applications. With this research, we found effects on realistic behavior of digital "paper" used in knowledge work.

Our work has contributed to the current research by the development of a detailed description and terminology of physical structures and bindings as well as novel behavior and interactions (e. g., quickzoom). Our tests about realistic behavior of real world behavior have shown some positive effects as well as some tendencies. However, an important issue is still

not solved: How does emerging behavior effect the creation and retrieval of knowledge over time? This requires a long-term test; the used application must be stable and have features for productivity use.

The bigger context of our work includes the automatic generation of metainformation, as briefly discussed in Sect. 2.2.4. Our work focuses especially on attributes and behavior that are supported by users' world knowledge. It is a challenge to develop applications that follow this paradigm. It puts the main focus of interpreting information on users, not computers.

Many other research directions follow a different approach by making machines more "intelligent" and increasing their capability of interpreting information. Examples include information retrieval or the Semantic Web, which form large research communities currently and influence many others fundamentally.

There is no competition between both research directions. It is a valid goal to improve machines in their "understanding" of content or structure. On the other side, it has to be understood that computers in the context of our work are tools for knowledge workers in offices. In this respect, it is important to think about how they can match and support the ability of human minds and thinking best. This takes away the focus of creating applications that are expected to find out what the users may want. As opposed to this, the center of our interest is to develop presentations and interactions with the machine that would fit human minds most appropriately.

Humans cannot compete with computer in the amount of data they can process. However, in most cases they are still better in interpreting information, especially when world knowledge is required. Computers work well in small universes of discourse, but in many other cases they do not reach the level of human understanding. With respect to this, our work proposes to put more resources toward creating applications that support the human capability of interpreting. This goes well in combination with improved data processing on computers.

WildDocs, our prototypic application, follows this direction by providing rich structures that can be understood due to the users' world knowledge. This supports office workers in creating and finding knowledge by "augmenting human intellect", as Doug Engelbart wrote almost half a century ago (Engelbart, 1962).

Part III.

Appendix

Appendix A.

Pre-Work and Introduction

A.1. Participant Agreement

Everyone who participated in the WildDocs usability test was asked to agree with the form printed on the next page. The document is based on the *Code of Practice for Research involving Human Participants* by the Faculty of Engineering and Physical Sciences, University of Dundee (FEPS Ethics Committee, 2005, 10–12).

The WD Experiment

Dear Participant:

Thank you for your interest in this project. This page describes what you will be asked to do for the study. Please read through it and then sign at the bottom to say that you understand and accept the conditions of this study. If you have questions, feel free to ask the researcher after the test is finished. Please understand that questions cannot be answered during the test.

The researcher will begin by asking you for some general information about yourself. You will learn from a short introduction movie how the application that will be tested is working. You also will learn more details about your task.

Basically, you will see a series of documents (text or pictures) on the screen and you will be asked to organize them on the screen in a way that allows you to find them quickly afterward. Then you will receive questions that ask you to find a specific documents that are among the ones you have organized.

You will then be asked to talk about what you think of the tested application. You will also be asked some questions about the experience of using it.

You as well as your actions on the screen will be recorded during the experiment as this will allow the researchers to learn more about how easy or how difficult it is to use the system. It will help them to understand what works and does not work so that they can improve this as well as other systems. Some video clips that show an important aspect may be used in presentations at research conferences or meetings.

Please note that you are helping the researchers to evaluate a new system. *You are not being tested – it is the system that is being tested.* There are therefore no right or wrong answers to the questions you will be asked.

Your participation in this study is voluntary and you can leave the study at any time without penalty or giving reasons. No undue risk arises from the participation in this study.

All the information which you give us and the video recordings (that is all data) will be stored safely and kept separate from information about your identity. Access to your data is minimized to the people involved in this research. If information about you is used for publications or presentations, we will ensure that no reference to your identity is made. If a photograph or video clip is used for presentation, your name will be changed. With reference to the use of photographic or video data for presentations, please tick one of the following boxes:

- ○ I agree that my likeness (e. g. on video tape) can be used for presentations.
- ○ I do not agree that my likeness (e. g. on video tape) can be used for presentations.

It is important that you understand that this is a research project and the prototypic application has only features implemented that help to answer well defined research questions.

The researchers are very grateful for your help.

In order to obtain an equal start situations for all participants, we ask you not to mention anything about the test or the tested system to any person that will perform the test after you.

Please date and sign this page below to indicate that you understand and accept the conditions of this study. (The "Participant/Session ID" will be given by the researcher.) Thank you!

Name of Participant: Participant/Session ID:

Signature:

Date:

A.2. Pre-Test Questionnaire

Before the test, the participant was asked the following questions:

- Sex (male/female)
- Nationality
- Profession (Student/Lecturer)
- Profession Level (Graduate/Undergraduate)
- Class at University
- How many hours do you use the computer in a typical week?
- What are you doing with the computer?
- How many years of computer experience do you have?
- How many years of GUI experience do you have?
- In which OS are you experienced in?
- With which input devices are you experienced with?
- Are you experienced with spatial organization of objects on the screen? (yes/no)
- Did anyone tell you about this test? (yes/no)
- If somebody told you, what was it?

A.3. Introduction Movie Manuscript

The following text was part of the introduction movies shown to participants. There was a separate introduction movie for each WildDocs version. Each paragraph indicates at the beginning for which versions it has been used.

> [*all*:] You can move a document by pressing the left mouse button on it and moving the mouse.
>
> [*all*:] Documents can be moved even behind other documents.
>
> [*v2*, *v3*, *v4*:] However, if a document is moved outside the pile's scope, it is pushed above it. You will recognize this by moving back the document. Now, it will be moved on top of the pile. You can use this behavior for moving documents to the front.
>
> [*all*:] You also can drag the background by clicking and dragging.
>
> [*v1*, *v3*, *v4*:] Scrollbars can be used to navigate, too. They disappear if all objects are visible on the screen.

[*v2*, *v3*, *v4*:] A shortcut to browse stacks is `CTRL-L` or `CTRL-R` for moving the node below the cursor to the left or to the right. The moved document is put to the very front.

[*all*:] All shortcuts that are available for you are marked on the keyboard.

[*all*:] If you want to push a document to the very front manually, press `CTRL-U` (for "move up"). For moving it into the very back, press `CTRL-D` (for "move down").

[*all*:] A selection tool gives you the opportunity to move more than one document at a time. To do this, move your mouse onto the document that you would like to move and press `CTRL-A`. A transparent white rectangle with a red border appears. All documents that intersect with this rectangle will be moved. You can expand this rectangle by moving the mouse onto any document and pressing `CTRL-A` again. Immediately, the rectangle expands so that it includes the complete topmost document on which the cursor is currently located. You may continue until all documents that should be moved intersect at least partly with the rectangle.

[*all*:] To move the selected nodes, move the cursor to the desired destination and press `CTRL-M` (for "move"). All documents that intersect with the transparent rectangle will be automatically dragged to this spot. After pressing `CTRL-M`, the selection rectangle fades out.

[*all*:] If you have some documents selected and want to get rid of the selection rectangle without moving the documents, press `CTRL-W` (for "wipe out"). The transparent rectangle will fade out.

[*v2*, *v4*:] Moving a stack results also in straightening the stack. If you want to straighten a stack without moving documents via the selection rectangle, move the mouse on top of it and press `CTRL-S` (for "straighten"). All documents that are directly below the cursor will be aligned to a straight looking stack.

[*v2*, *v3*, *v4*:] The system has a desk metaphor. That is a brown rectangle in the background that symbolizes a desk. You may move documents temporarily outside this area, as you can do in the real world by putting them onto the floor. However, finally, they have to be on the desk again. They may overlap the desk imitation's border to some extend, as they could also in the real world without falling down. This behavior of falling down is not implemented; the user has to take care of this rule.

[*all*:] The application allows basic zooming through the Zoom menu. You can zoom in to 125 % relative to the current view as often as you want as well as zoom out to 80 % relative to the current view. If you want to zoom back to 100 %, select the appropriate menu entry.

[*all*:] Even though there are several other menu entries, do not use any other than zoom.

[*v2*:] As you can see on the Zoom menu entries, you may use shortcuts to zoom in or out. Those are `CTRL-0` to zoom in, `CTRL--` to zoom out, and `CTRL-=` to zoom to 100 %.

[*v2*:] Smooth zooming can be used to zoom in or out seamlessly. For zooming in press the right mouse button somewhere on the background and keep it pressed. Then, move the mouse to the right. The more you move it to the right, the faster zooming becomes. If you want to zoom out, keep the right mouse button pressed, but move the mouse to the left. Also here, the further you move to the left, the faster zooming becomes. To stop smooth zooming, just release the right mouse button.

[*v2*:] Quickzoom is a way to zoom very quickly to view the used space completely and zoom back again to a designated spot. In order to see all objects, press `CTRL-Z` (for "quickZoom"). A fading out magenta colored rectangle shows the area of the previous screen in order to give you some more orientation. If you want to zoom back, move the mouse to the desired spot and press `CTRL-Z` again. The view will zoom to where the mouse is located. You need to know that if you use the Zoom menu or smooth zooming after quickzoom has fully zoomed out, the next quickzoom command `CTRL-Z` will show the full zoom-out view again next.

[*v3*:] There are two types of rotation for documents: One is emerging rotation. This rotation happens automatically while you click on documents, for example, for moving them. You also will experience this when documents are moved automatically, for example, through selection `CTRL-A` and move shortcut `CTRL-M`. Additionally, some random offset is applied to automatic movements.

[*v3*:] The other rotation type is rotation on purpose. Double click a document with the left mouse button and hold the button at the second click. A circle will be put to where the rotation center is located. Then, move the mouse in order to rotate the document, still having the left mouse button pressed. After the desired angle is reached, release the mouse button. (You will experience that a rotated document is pushed on top if it leaves a stack's scope.)

[*v1*:] Objects have handlers, displayed as small circles. You can press the mouse button on one of them and resize the object. Pictures show a distortion when resized.

[*v2*, *v3*, *v4*:] The documents that you see on the screen may be split into several individual pages.

[*all*:] Your task is divided into two parts. Part I: Organize – Part II: Find. Think aloud if you experience any problem at any time during the test.

[*all*:] Part I: Organize.

[*all*:] You will see a pile of documents on the screen.

[*v2*, *v3*, *v4*:] They are split up into several pages. Their sequence is reversed: The last page is above the previous one.

[*all*:] Wait until the staff tells you to start organizing.

[*all*:] Place the documents in a way that allows you to find them quickly afterwards. You will be ask questions like: "Find the document that has this or that specific visual attribute or content." You may use all previously explained features.

[*all*:] Think aloud if you experience any problem at any time. As soon as you finish, announce it to the experiment staff.

[*all*:] Part II: Find

[*all*:] The experiment staff will now show you finding tasks one by one that you are asked to do. Before you start searching, read the question number and the complete task description aloud. Then, start searching. Try to complete the tasks as quickly as possible.

[*all*:] In this stage of the test, perform only the requested finding task. Do not start re-organizing.

[*all*:] After you found the document, read the requested part of it aloud, as requested in the task description. Then, continue with the next task.

[*all*:] Think aloud if you experience any problem at any time. That may include problems in understanding questions.

[*all*:] Thank you for your support!

A.4. Foreign Language Sample Documents

The following pages show the four samples of foreign languages that we presented to the participants, as discussed in Sect. 5.2.5. All samples are taken from the *Human Rights* (United Nations, 1948). The first one is Greek[1] on the facing page, then Arabic[2] on page 232, Hebrew[3] on page 233, and finally Japanese[4] on page 234.

[1] Page 9 from the PDF version at http://www.unhchr.ch/udhr/lang/grk.htm
[2] Page 5 from the PDF version at http://www.unhchr.ch/udhr/lang/arz.htm
[3] Page 1 from the PDF version at http://www.unhchr.ch/udhr/lang/hbr.htm
[4] Page 8 from the PDF version at http://www.unhchr.ch/udhr/lang/jpn.htm

ΑΡΘΡΟ 28

Καθένας έχει το δικαίωμα να επικρατεί μια κοινωνική και διεθνής τάξη, μέσα στην οποία τα δικαιώματα και οι ελευθερίες που προκηρύσσει η παρούσα Διακήρυξη να μπορούν να πραγματώνονται σε όλη τους την έκταση.

ΑΡΘΡΟ 29

1. Το άτομο έχει καθήκοντα απέναντι στην κοινότητα, μέσα στα πλαίσια της οποίας και μόνο είναι δυνατή η ελεύθερη και ολοκληρωμένη ανάπτυξη της προσωπικότητάς του.
2. Στην άσκηση των δικαιωμάτων του και στην απόλαυση των ελευθεριών του κανείς δεν υπόκειται παρά μόνο στους περιορισμούς που ορίζονται από τους νόμους, με αποκλειστικό σκοπό να εξασφαλίζεται η αναγνώριση και ο σεβασμός των δικαιωμάτων και των ελευθεριών των άλλων, και να ικανοποιούνται οι δίκαιες απαιτήσεις της ηθικής, της δημόσιας τάξης και του γενικού καλού, σε μια δημοκρατική κοινωνία.
3. Τα δικαιώματα αυτά και οι ελευθερίες δεν μπορούν, σε καμία περίπτωση, να ασκούνται αντίθετα προς τους σκοπούς και τις αρχές των Ηνωμένων Εθνών.

ΑΡΘΡΟ 30

Καμιά διάταξη της παρούσας Διακήρυξης δεν μπορεί να ερμηνευθεί ότι παρέχει σε ένα κράτος, σε μια ομάδα ή σε ένα άτομο οποιοδήποτε δικαίωμα να επιδίδεται σε ενέργειες ή να εκτελεί πράξεις που αποβλέπουν στην άρνηση των δικαιωμάτων και των ελευθεριών που εξαγγέλλονται σε αυτήν.

لكل شخص الحق في الراحة، أو في أوقات الفراغ، ولا سيما في تحديد معقول لساعات العمل وفي عطلات دورية بأجر.

المادة 25

1. لكل شخص الحق في مستوى من المعيشة كاف للمحافظة على الصحة والرفاهية له ولأسرته. ويتضمن ذلك التغذية والملبس والمسكن والعناية الطبية وكذلك الخدمات الاجتماعية اللازمة. وله الحق في تأمين معيشته في حالات البطالة والمرض والعجز والترمل والشيخوخة وغير ذلك من فقدان وسائل العيش نتيجة لظروف خارجة عن إرادته.
2. للأمومة والطفولة الحق في مساعدة ورعاية خاصتين. وينعم كل الأطفال بنفس الحماية الاجتماعية سواء أكانت ولادتهم ناتجة عن رباط شرعي أم بطريقة غير شرعية.

المادة 26

1. لكل شخص الحق في التعلم. ويجب أن يكون التعليم في مراحله الأولى والأساسية على الأقل بالمجان، وأن يكون التعليم الأولي إلزاميا وينبغي أن يعمم التعليم الفني والمهني، وأن ييسر القبول للتعليم العالي على قدم المساواة التامة للجميع وعلى أساس الكفاءة.
2. يجب أن تهدف التربية إلى إنماء شخصية الإنسان إنماء كاملا، وإلى تعزيز احترام الإنسان والحريات الأساسية وتنمية التفاهم والتسامح والصداقة بين جميع الشعوب والجماعات العنصرية أو الدينية، وإلى زيادة مجهود الأمم المتحدة لحفظ السلام.
3. للآباء الحق الأول في اختيار نوع تربية أولادهم.

المادة 27

1. لكل فرد الحق في أن يشترك اشتراكا حرا في حياة المجتمع الثقافي وفي الاستمتاع بالفنون والمساهمة في التقدم العلمي والاستفادة من نتائجه.
2. لكل فرد الحق في حماية المصالح الأدبية والمادية المترتبة على إنتاجه العلمي أو الأدبي أو الفني.

المادة 28

لكل فرد الحق في التمتع بنظام اجتماعي دولي تتحقق بمقتضاه الحقوق والحريات المنصوص عليها في هذا الإعلان تحققا تاما.

المادة 29

1. على كل فرد واجبات نحو المجتمع الذي يتاح فيه وحده لشخصيته أن تنمو نموا حرا كاملا.
2. يخضع الفرد في ممارسته حقوقه لتلك القيود التي يقررها القانون فقط، لضمان الاعتراف بحقوق الغير وحرياته واحترامها ولتحقيق المقتضيات العادلة للنظام العام والمصلحة العامة والأخلاق في مجتمع ديمقراطي.

הכרזה לכל באי עולם בדבר זכויות האדם

הואיל והכרה בכבוד הטבעי אשר לכל בני משפחת האדם ובזכויותיהם השוות והבלתי נפקעות הוא יסוד החופש, הצדק והשלום בעולם.

הואיל והזלזול בזכויות האדם וביזוין הבשילו מעשים פראיים שפגעו קשה במצפונה של האנושות; ובנין עולם, שבו ייהנו כל יצורי אנוש מחירות הדיבור והאמונה ומן החירות מפחד וממחסור, הוכרז כראש שאיפותיו של כל אדם.

הואיל והכרח חיוני הוא שזכויות האדם תהיינה מוגנות בכוח שלטונו של החוק, שלא יהא האדם אנוס, כמפלט אחרון, להשליך את יהבו על מרידה בעריצות ובדיכוי.

הואיל והכרח חיוני הוא לקדם את התפתחותם של יחסי ידידות בין האומות.

הואיל והעמים המאוגדים בארגון האומות המאוחדות חזרו ואישרו במגילה את אמונתם בזכויות היסוד של האדם, בכבודה ובערכה של אישיותו ובזכות שווה לגבר ולאשה; ומנוי וגמור אתם לסייע לקדמה חברתית ולהעלאת רמת החיים בתוך יתר חירות.

הואיל והמדינות החברות התחייבו לפעול, בשיתוף עם ארגון האומות המאוחדות, לטיפול יחס כבוד כללי אל זכויות האדם ואל חירויות היסוד והקפדה על קיומן.

הואיל והבנה משותפת במהותן של זכויות וחירויות אלה הוא תנאי חשוב לקיומה השלם של התחייבות זו.

לפיכך מכריזש העצרת באזני כל באי העולם את ההכרזש הזאת בדבר זכויות האדם כרמת הישגים כללית לכל העמים והאומות, כדי שכל יחיד וכל גוף חברתי ישווה תמיד לנגד עיניו וישאף לטפח, דרך לימוד וחינוך, יחס של כבוד אל הזכויות ואל החירויות הללו, ולהבטיח באמצעים הדרגתיים, לאומיים ובינלאומיים, שההכרה בעקרונות אלה וההקפדה עליהם תהא כללית ויעילה בקרב אוכלוסי המדינות החברות ובקרב האוכלוסים שבארצות שיפוטם.

סעיף א. כל בני אדם נולדו בני חורין ושווים בערכם ובזכויותיהם. כולם חוננו בתבונה ובמצפון, לפיכך חובה עליהם לנהוג איש ברעהו ברוח של אחוה.

סעיף ב. (1) כל אדם זכאי לזכויות ולחרויות שנקבעו בהכרזש זו ללא הפליה כלשהיא מטעמי גזע, צבע, מין, לשון, דת, דעה פוליטית או דעה בבעיות אחרות, בגלל מוצא לאומי או חברתי, קנין, לידה או מעמד אחר.

(2) גדולה מזו, לא יופלה אדם על פי מעמדה המדיני, על פי סמכותה או על פי מעמדה הבינלאומי של המדינה או הארץ שאליה הוא שייך, בין שהארץ היא עצמאית, ובין שהיא נתונה לנאמנות, בין שהיא נטולת שלטון עצמי ובין שריבונותה מוגבלת כל הגבלה אחרת.

סעיף ג. כל אדם יש לו הזכות לחיים, לחרות ולבטחון אישי.

סעיף ד. לא יהיה אדם עבד או משועבד; עבדות וסחר עבדים יאסרו לכל צורותיהם.

סעיף ה. לא יהיה אדם נתון לעינויים, ולא ליחס או לעונש אכזריים, בלתי אנושיים או משפילים.

סעיף ו. כל אדם זכאי להיות מוכר בכל מקום כאשיות בפני החוק.

סעיף ז. הכל שווים לפני החוק וזכאים ללא הפליה להגנה שווה של החוק. הכל זכאים להגנה שווה מפני כל הפליה המפירה את מצוות ההכרזש הזאת ומפני כל הסתה להפליה כזו.

2. 母と子とは、特別の保護及び援助を受ける権利を有する。すべての児童は、嫡出であると否とを問わず、同じ社会的保護を享有する。

第26条

1. すべて人は、教育を受ける権利を有する。教育は、少なくとも初等の及び基礎的の段階においては、無償でなければならない。初等教育は、義務的でなければならない。技術教育及び職業教育は、一般に利用できるもでなければならず、また、高等教育は、能力に応じ、すべての者にひとしく開放されていなければならない。
2. 教育は、人格の完全な発展並びに人権及び基本的自由の尊重の教科を目的としなければならない。教育は、すべての国又は人種的もしくは宗教的集団の相互間の理解、寛容及び友好関係を増進し、かつ、平和の維持のため、国際連合の活動を促進するものでなければならない。

3. 親は、子に与える教育の種類を選択する優先的権利を有する。

第27条

1. すべて人は、自由に社会の文化生活に参加し、芸術を鑑賞し、及び科学の進歩とその恩恵とにあずかる権利を有する。
2. すべて人は、その創作した科学的、文学的又は美術的作品から生ずる精神的及び物質的利益を保護される権利を有する。

第28条

すべて人は、この宣言に掲げる権利及び自由が完全に実現される社会的及び国際的秩序に対する権利を有する。

第29条

1. すべて人は、その人格の自由かつ完全な発展がその中にあつてのみ可能である社会に対して義務を負う。

Appendix B.

Post-Work and Analysis

B.1. Post-Test Questionnaire

After the test, the participant was asked the following questions:

- What do you think of this application?
- Describe your way of organizing.
- Rate this application on a scale from 1 (worst) to 5 (best).
- Explain your rating.
- (Additional questions that were noted by the administrator during the test.)
- Could you imagine feature for applications that exist already today? Which ones?
- Do you have questions or additional comments?

B.2. Log Files

The following snippet shows the log file for session *s01*. The first block shows the logged data after the organization phase, the second block after the finding phase. Counts are accumulating, that means that counts at the second block include also those from the first one.

[WildDocs-1127117012734msec_id2161283.log]

```
1
2  *** STATISTICS @ Mon Sep 19 09:35:50 CEST 2005 (= 1127115350750 msec)
3  WildDocs version compile time: Mon Sep 19 08:19:33 CEST 2005
4  WildDocs instance (hash code): 2161283
5  Active window title: WD | v4 [fs]
6  56 low level docs on screen
7  5185 px area width of low level doc
8  3584 px area height of low level doc
9  1371 mm area width of low level doc
10 948 mm area height of low level doc
11 164x straighten stack (STRAIGHTENSTACK)
12 2x select node below cursor (SELECTNODESBELOWCURSOR)
13 2x move selected nodes (MOVESELECTEDNODES)
14 0x side push left (SIDEPUSHLEFT)
```

```
4x side push right (SIDEPUSHRIGHT)
12x index push up (INDEXPUSHUP)
6x index push down (INDEXPUSHDOWN)
1x zoom in (ZOOMIN)
26x zoom out (ZOOMOUT)
5x zoom reset (ZOOMRESET)
0x toggle quick zoom (QUICKZOOMTOGGLE)
0x purposeful rotation (PURPOSEFULROTATION)
0x smooth zooming (SMOOTHZOOMING)

*** STATISTICS @ Mon Sep 19 10:03:32 CEST 2005 (= 1127117012734 msec)
WildDocs version compile time: Mon Sep 19 08:19:33 CEST 2005
WildDocs instance (hash code): 2161283
Active window title : WD | v4 [fs]
56 low level docs on screen
5185 px area width of low level doc
3624 px area height of low level doc
1371 mm area width of low level doc
958 mm area height of low level doc
164x straighten stack (STRAIGHTENSTACK)
2x select node below cursor (SELECTNODESBELOWCURSOR)
2x move selected nodes (MOVESELECTEDNODES)
0x side push left (SIDEPUSHLEFT)
4x side push right (SIDEPUSHRIGHT)
17x index push up (INDEXPUSHUP)
118x index push down (INDEXPUSHDOWN)
6x zoom in (ZOOMIN)
34x zoom out (ZOOMOUT)
6x zoom reset (ZOOMRESET)
0x toggle quick zoom (QUICKZOOMTOGGLE)
0x purposeful rotation (PURPOSEFULROTATION)
0x smooth zooming (SMOOTHZOOMING)
```

Appendix C.

Acknowledgements

I would like to thank my family, to whom I dedicate this thesis. Thank you for your support, especially through the time of heavy research with long working hours. Thank you, Christiane, Janik, and Noah. I also would like to thank my Mother and Father, who gave me the freedom early on in choosing my way without trying to push me into a certain direction. This freedom was a prerequisite for finding my place as an academic, which I highly enjoy. Sadly, my Father died before he saw me finishing this thesis. I will remember him.

My advisor was Uffe K. Wiil, before he followed a call of another university. Thank you, Uffe, for giving me the opportunity to continue my path as a research scientist. Peter J. Nürnberg agreed to become my new advisor. Pete, I appreciate that you decided to advise me in my work, introduced me to the research community, and supported me in various specialist and organizational questions. I enjoyed working with you very much.

Forty-five volunteers tested my application WildDocs. They were promised anonymity; therefore, I cannot mention them here by name. I would like to thank them for their willingness to support my research, also those who participated in pre-testing WildDocs.

The following people supported my work with ideas, discussions, or other help. I would like to thank them, plus many others in the research community.

Alexander Wagner
Anders Schmidt Kristensen
Anton J. Atzenbeck
Bernd Raichle
Britta Jensen
David L. Hicks
Diana F. Hansen
Dil Muhammad Akbar Hussain
Frank M. Shipman
Fredrik H. Madsen
Giorgos Gkotsis
Jamie Blustein
Keith Andrews
Kim C. Kristoffersen
Kim H. Esbensen
Kumiyo Nakakoji
Manolis M. Tzagarakis
Mark Bernstein
Markus Kohm
Michael Rosenørn
Nasrullah Memon
Peter J. Nürnberg
Siegfried Reich
Steffen Podlech
Tanja Keller
Uffe K. Wiil
Ulla Tradsborg
Wade Tregaskis
Wolfgang Kienreich

Bibliography

Adobe Systems. *PDF Reference. Adobe® Portable Document Format Version 1.6*. Adobe Systems, 5th edn., 2004. URL http://partners.adobe.com/public/developer/en/pdf/PDFReference16.pdf

A. Agarawala, R. Balakrishnan. *Keepin' it real: pushing the desktop metaphor with physics, piles and the pen*. In: *CHI '06: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM Press, 2006; pp. 1283–1292. URL http://doi.acm.org/10.1145/1124772.1124965

Apple Computer. *About the Sudden Motion Sensor*. WWW, 2005a. URL http://docs.info.apple.com/article.html?artnum=300781, visited on 2006-03-26

Apple Computer. *Macintosh OpenGL Programming Guide*, 2005b. URL http://developer.apple.com/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/index.html, visited on 2006-03-26

Apple Computer. *Quartz 2D Programming Guide*, 2006. URL http://developer.apple.com/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/, visited on 2006-03-26

M. S. D. Ashdown. *Personal projected displays*. Tech. Rep. 585, University of Cambridge Computer Laboratory, 2004. URL http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-585.pdf

C. Atzenbeck. *Wild Windows on Mac OS X*. WWW, 2005. URL http://www.atzenbeck.de/research/wildWindows/, visited on 2006-03-26

C. Atzenbeck, P. J. Nürnberg. *Approaching structure interoperability*. In: *Procedings of the 4th International Conference on Knowledge Management (I-KNOW '04)*, J.UCS Conference Proceedings. 2004; pp. 269–278. URL http://www.atzenbeck.de/publications/atzenbeck+04b.pdf

C. Atzenbeck, P. J. Nürnberg. *Constraints in spatial structures*. In: *Proceedings of the 16th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2005a; pp. 63–65. URL http://doi.acm.org/10.1145/1083356.1083368

C. Atzenbeck, P. J. Nürnberg. *Looking beyond computer applications: Investigating rich structures*. In: U. K. Wiil (ed.), *Proceedings of the International Metainformatics Symposium 2004*, vol. 3511 of *Lecture Notes in Computer Science*. 2005b; pp. 51–65. URL http://dx.doi.org/10.1007/11518358_5

C. Atzenbeck, P. J. Nürnberg. *WildDocs – emerging metainformation support*. In: *Procedings of the 5th International Conference on Knowledge Management (I-KNOW '05)*,

J.UCS Conference Proceedings. 2005c; pp. 286–293. URL http://www.atzenbeck.de/publications/atzenbeck+05c.pdf

C. Atzenbeck, U. K. Wiil, D. L. Hicks. *Toward a structure domain interoperability space*. In: D. L. Hicks (ed.), *Proceedings of the International Metainformatics Symposium 2003*, vol. 3002 of *Lecture Notes in Computer Science*. Springer, 2004; pp. 66–71. URL http://www.springerlink.com/link.asp?id=vn0y51tmc9rlbr2h

R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, B. MacIntyre. *Recent advances in augmented reality*. IEEE Computer Graphics and Applications 21 (2001) 34–47. URL http://www.cs.unc.edu/~azuma/cga2001.pdf

M. Beaudouin-Lafon. *Novel interaction techniques for overlapping windows*. In: *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST '01)*. ACM Press, 2001; pp. 153–154. URL http://doi.acm.org/10.1145/502348.502371

B. B. Bederson, J. Grosjean, J. Meyer. *Toolkit design for interactive structured graphics*. IEEE Transactions on Software Engineering 30 (2004) 535–546. URL http://doi.ieeecomputersociety.org/10.1109/TSE.2004.44

R. Bergman, T. Hastings. *The printer working group. Standard for media standardized names*. Tech. Rep. IEEE-ISTO 5101.1-2002, IEEE Industry Standards and Technology Organization, 2002. URL ftp://ftp.pwg.org/pub/pwg/standards/pwg5101.1.pdf

M. Bernstein. *Patterns of hypertext*. In: *Proceedings of the 9th ACM Conference on Hypertext and Hypermedia*. ACM Press, 1998; pp. 21–29. URL http://doi.acm.org/10.1145/276627.276630

M. Bernstein. *Collage, composites, construction*. In: *Proceedings of the 14th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2003; pp. 122–123. URL http://doi.acm.org/10.1145/900051.900077

N. Brügger. *Internet: Medium and text*. In: *Proceedings of the 15th Nordic Conference on Media and Communication Research*. 2001; p. unknown

V. Bush. *As we may think*. The Atlantic Monthly 176 (1945) 101–108. URL http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm

S. K. Card, L. Hong, J. D. Mackinlay, E. H. Chi. *3Book: a 3D electronic smart book*. In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. ACM Press, 2004a; pp. 303–307. URL http://doi.acm.org/10.1145/989863.989915

S. K. Card, L. Hong, J. D. Mackinlay, E. H. Chi. *3Book: a scalable 3D virtual book*. In: *Extended Abstracts of the 2004 Conference on Human Factors and Computing Systems*. ACM Press, 2004b; pp. 1095–1098. URL http://doi.acm.org/10.1145/985921.985997

Y.-C. Chu, D. Bainbridge, M. Jones, I. H. Witten. *Realistic books: a bizarre homage to an obsolete medium?* In: *Proceedings of the 4th Joint ACM/IEEE Conference on Digital Libraries*. ACM Press, 2004; pp. 78–86. URL http://doi.acm.org/10.1145/996350.996372

Y.-C. Chu, I. H. Witten, R. Lobb, D. Bainbridge. *How to turn the page*. In: *Proceedings of the 3rd Joint ACM/IEEE Conference on Digital Libraries*. IEEE Computer Society, 2003; pp. 186–188. URL http://www.nzdl.org/html/open_the_book/p186-chu.pdf

A. Cockburn. *Revisiting 2D vs 3D implications on spatial memory*. In: *Proceedings of the 5th Conference on Australasian User Interface (CRPIT '04)*. Australian Computer Society, Inc., 2004; pp. 25–31. URL http://doi.acm.org/976314

J. Cohen. *A power primer*. Psychological Bulletin 112 (1992) 155–159. URL http://content.apa.org/journals/bul/112/1/155

I. Cole. *Human aspects of office filing: Implications for the electronic office*. In: *Proceedings of the 26th Annual Meeting of the Human Factors Society*. 1982; pp. 59–63

T. T. A. Combs, B. B. Bederson. *Does zooming improve image browsing?* In: *Proceedings of the 4th ACM International Conference on Digital Libraries*. ACM Press, 1999; pp. 130–137. URL http://doi.acm.org/10.1145/313238.313286

J. Conklin, M. L. Begeman. *gIBIS: a hypertext tool for team design deliberation*. In: *Proceeding of the ACM Conference on Hypertext*. ACM Press, 1987; pp. 247–251. URL http://doi.acm.org/10.1145/317426.317444

J. Conklin, M. L. Begeman. *gIBIS: a hypertext tool for exploratory policy discussion*. In: *Proceedings of the 1988 ACM Conference on Computer-Supported Cooperative Work*. ACM Press, 1988; pp. 140–152. URL http://doi.acm.org/10.1145/62266.62278

R.-A. de Beaugrande, W. Dressler. *Introduction to Text Linguistics*. Addison-Wesley, 1981

D. C. De Roure, D. G. Cruickshank, D. T. Michaelides, K. R. Page, M. J. Weal. *On hyperstructure and musical structure*. In: *Proceedings of the 13th Conference on Hypertext and Hypermedia*. ACM Press, 2002; pp. 95–104. URL http://doi.acm.org/10.1145/513338.513366

P. Dragicevic. *Combining crossing-based and paper-based interaction paradigms for dragging and dropping between overlapping windows*. In: *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*. ACM Press, 2004; pp. 193–196. URL http://doi.acm.org/10.1145/1029632.1029667

A. D. Dunn. *Notes on the standardization of paper sizes*. Dunn, 1972. URL http://www.cl.cam.ac.uk/~mgk25/volatile/dunn-papersizes.pdf

Eastgate Systems. *Tinderbox™ for Macintosh v. 2.2. User's Manual & Reference*, 2004. URL http://www.eastgate.com/Tinderbox/

EDS Inc. *Guide to international paper sizes. Concise tables of measurements*. WWW, 1997. URL http://home.inter.net/eds/paper/papersize.html, last updated 2004-01-15

D. C. Engelbart. *Augmenting human intellect: A conceptual framework*. Summary Report AFOSR-3233, Standford Research Institute, 1962. URL http://www.bootstrap.org/augdocs/friedewald030402/augmentinghumanintellect/ahi62index.html

D. C. Engelbart. *Collaboration support provisions in AUGMENT*. In: *Proceedings of the AFIPS Office Automation Conference (OAC '84)*. 1984; pp. 51–58. URL http://www.bootstrap.org/augdocs/oad-2221.htm

FEPS Ethics Committee. *Code of Practice for Research Involving Human Participants*. University of Dundee, 2005. URL http://www.computing.dundee.ac.uk/staff/awaller/fepsethics/Code_of_Practice_2005.pdf, edition 2005-07-22

A. Field, G. Hole. *How to Design and Report Experiments*. Sage Publications, 2003. Reprinted 2004

L. Francisco-Revilla, F. Shipman. *Parsing and interpreting ambiguous structures in spatial hypermedia*. In: *Proceedings of the 16th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2005; pp. 107–116. URL http://doi.acm.org/10.1145/1083356.1083376

D. Frohlich, M. Perry. *The paperful office paradox*. Tech. Rep. HPL-94-20, Hewlett-Packard Laboratories, 1994. URL http://www.hpl.hp.com/techreports/94/HPL-94-20.html

G. W. Furnas. *Generalized fisheye views*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*. ACM Press, 1986; pp. 16–23. URL http://doi.acm.org/10.1145/22627.22342

L. N. Garrett, K. E. Smith, N. Meyrowitz. *Intermedia: Issues, strategies, and tactics in the design of a hypermedia document system*. In: *Proceedings of the 1986 ACM Conference on Computer Supported Cooperative Work (CSCW '86)*. ACM Press, 1986; pp. 163–174. URL http://doi.acm.org/10.1145/637069.637090

D. Gentner, J. Nielsen. *The Anti-Mac interface*. Communications of the ACM 39 (1996) 70–82. URL http://doi.acm.org/10.1145/232014.232032

L. Good, B. B. Bederson. *Zoomable user interfaces as a medium for slide show presentations*. Information Visualization 1 (2002) 35–49. URL http://dx.doi.org/10.1057/palgrave/ivs/9500004

H. P. Grice. *Logic and conversation*. In: P. Cole, J. L. Morgan (eds.), *Speech Acts*, vol. 3, pp. 41–58. Academic Press, 1975;

K. Grønbæk, J. F. Kristensen, P. Ørbæk, M. A. Eriksen. *"Physical hypermedia": Organising collections of mixed physical and digital material*. In: *Proceedings of the 14th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2003; pp. 10–19. URL http://doi.acm.org/10.1145/900051.900056

K. Grønbæk, R. H. Trigg. *Design issues for a Dexter-based hypermedia system*. Communications of the ACM 37 (1994) 40–49. URL http://doi.acm.org/10.1145/175235.175238

K. Grønbæk, P. P. Vestergaard, P. Ørbæk. *Towards geo-spatial hypermedia: Concepts and prototype implementation*. In: *Proceedings of the 13th Conference on Hypertext and Hypermedia*. ACM Press, 2002; pp. 117–126. URL http://doi.acm.org/10.1145/513338.513370

K. Gupton, F. Shipman. *Visual Knowledge Builder version 0.70. The user's manual*. Center for the Study of Digital Libraries, Texas A&M University, 2000. URL http://www.csdl.tamu.edu/VKB/Download/VKBManual.PDF

F. Halasz, T. P. Moran. *Analogy considered harmful*. In: *Proceedings of the Conference on Human Factors in Computing Systems*. ACM Press, 1982; pp. 383–386. URL http://doi.acm.org/10.1145/800049.801816

F. Halasz, M. Schwartz. *The Dexter hypertext reference model*. Communications of the ACM 37 (1994) 30–39. URL http://doi.acm.org/10.1145/175235.175237

F. G. Halasz. *Reflections on NoteCards: Seven issues for the next generation of hypermedia systems*. In: *Proceeding of the ACM Conference on Hypertext*. ACM Press, 1987; pp. 345–365. URL http://doi.acm.org/10.1145/317426.317451

F. G. Halasz, T. P. Moran, R. H. Trigg. *NoteCards in a nutshell*. In: *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface (CHI '87)*. ACM Press, 1987; pp. 45–52. URL http://doi.acm.org/10.1145/29933.30859

R. Hammwöhner. *Offene Hypertextsysteme. Das Konstanzer Hypertextsystem (KHS) im wissenschaftlichen und technischen Kontext*, vol. 32 of *Schriften zur Informationswissenschaft*. Universitätsverlag Konstanz, 1997

R. Hobbs. *Mark Lombardi. Global Networks*. Independent Curators International, 2003

A. Hoeben, P. J. Stappers. *Flicking through page-based documents with thumbnail sliders and electronic dog-ears*. In: *CHI '00 Extended Abstracts on Human Factors in Computing Systems*. ACM Press, 2000; pp. 191–192. URL http://doi.acm.org/10.1145/633292.633397

K. Hornbæk, B. B. Bederson, C. Plaisant. *Navigation patterns and usability of zoomable user interfaces with and without an overview*. ACM Transactions on Computer-Human Interaction (TOCHI) 9 (2002) 362–389. URL http://doi.acm.org/10.1145/586081.586086

W. Johnson, H. Jellinek, L. Klotz, R. Rao, S. K. Card. *Bridging the paper and electronic worlds: the paper user interface*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '93)*. ACM Press, 1993; pp. 507–512. URL http://doi.acm.org/10.1145/169059.169445

H. Kawahara, P. Byrne, D. Johnson. *Project Looking Glass. API Design Overview (Draft)*, 2004. URL https://lg3d-core.dev.java.net/files/documents/1834/7596/LG3DAPIOverview-DRAFT.pdf

F. Khan. *A survey of note-taking practices*. Tech. Rep. HPL-93-107, Hewlett-Packard Laboratories, 1994. URL http://www.hpl.hp.com/techreports/93/HPL-93-107.html

A. Khella, B. B. Bederson. *Pocket PhotoMesa: a zoomable image browser for PDAs*. In: *Proceedings of the 3rd International Conference on Mobile and Ubiquitous Multimedia (MUM '04)*. ACM Press, 2004; pp. 19–24. URL http://doi.acm.org/10.1145/1052380.1052384

A. Kidd. *The marks are on the knowledge worker*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Press, 1994; pp. 186–191. URL http://doi.acm.org/10.1145/191666.191740

J. Kim, S. M. Seitz, M. Agrawala. *The office of the past: Document discovery and tracking from video*. In: *2004 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW '04)*. 2004a; pp. 157–157. URL http://dx.doi.org/10.1109/CVPR.2004.461

J. Kim, S. M. Seitz, M. Agrawala. *Video-based document tracking: Unifying your physical and electronic desktops*. In: *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04)*. ACM Press, 2004b; pp. 99–107. URL http://doi.acm.org/10.1145/1029632.1029650

J. Kim, S. M. Seitz, M. Agrawala. *Video-based document tracking: Unifying your physical and electronic desktops (demonstration movie)*. WWW, 2004c. URL http://grail.cs.washington.edu/projects/office/doc/kim04unifying_video_divx52.avi, in combination with Kim et al. (2004b); visited on 2006-03-27

J. C. King. *A format design case study: PDF*. In: *Proceedings of the 15th ACM Conference on Hypertext and hypermedia*. ACM Press, 2004; pp. 95–97. URL http://doi.acm.org/10.1145/1012807.1012810

M. Kuhn. *International standard paper sizes*. WWW, 1996. URL http://www.cl.cam.ac.uk/~mgk25/iso-paper.html, visited on 2006-03-12

J. Lamping, R. Rao, P. Pirolli. *A focus+context technique based on hyperbolic geometry for visualizing large hierarchies*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*. ACM Press/Addison-Wesley, 1995; pp. 401–408. URL http://doi.acm.org/10.1145/223904.223956

M. Lansdale. *The psychology of personal information management*. Applied Ergonomics 19 (1988) 55–66. URL http://dx.doi.org/10.1016/0003-6870(88)90199-8

J. P. Lewis, R. Rosenholtz, N. Fong, U. Neumann. *VisualIDs: automatic distinctive icons for desktop interfaces*. ACM Transactions on Graphics 23 (2004) 416–423. URL http://doi.acm.org/10.1145/1015706.1015739

M. Lombardi. *The recent drawings: an overview (artist statement)*. WWW, 1997. URL http://www.pierogi2000.com/flatfile/lombardidrawingshow.html, visited on 2006-04-06

T. W. Malone. *How do people organize their desks? Implications for the design of office information systems*. ACM Transactions on Information Systems (TOIS) 1 (1983) 99–112. URL http://doi.acm.org/10.1145/357423.357430

R. Mander, D. E. Rose, G. Salomon, Y. Y. Wong, T. Oren, S. Booker, S. Houde. *Method and apparatus for organizing information in a computer system*. United States Patent US 6,243,724 B1, Apple Computer, 1994. URL http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=6243724, date of Patent 2001-06-05

R. Mander, G. Salomon, Y. Y. Wong. *A 'pile' metaphor for supporting casual organization of information*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Press, 1992; pp. 627–634. URL http://doi.acm.org/10.1145/142750.143055

C. C. Marshall. *Annotation: From paper books to the digital library*. In: *Proceedings of the 2nd ACM International Conference on Digital Libraries*. ACM Press, 1997; pp. 131–140. URL http://doi.acm.org/10.1145/263690.263806

C. C. Marshall. *Toward an ecology of hypertext annotation*. In: *Proceedings of the 9th ACM Conference on Hypertext and Hypermedia*. ACM Press, 1998; pp. 40–49. URL http://doi.acm.org/10.1145/276627.276632

C. C. Marshall, F. G. Halasz, R. A. Rogers, W. C. Janssen. *Aquanet: a hypertext tool to hold your knowledge in place*. In: *Proceedings of the 3rd Annual ACM Conference on Hypertext*. ACM Press, 1991; pp. 261–275. URL http://doi.acm.org/10.1145/122974.123000

C. C. Marshall, F. M. Shipman, J. H. Coombs. *VIKI: Spatial hypertext supporting emergent structure*. In: *Proceedings of the 1994 ACM European Conference on Hypermedia Technology*. ACM Press, 1994; pp. 13–23. URL http://doi.acm.org/10.1145/192757.192759

N. Matsushita, Y. Ayatsuka, J. Rekimoto. *Dual touch: a two-handed interface for pen-based PDAs*. In: *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*. ACM Press, 2000; pp. 211–212. URL http://doi.acm.org/10.1145/354401.354774

P. Mogensen, K. Grønbæk. *Hypermedia in the virtual project room – toward open 3D spatial hypermedia*. In: *Proceedings of the 11th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2000; pp. 113–122. URL http://doi.acm.org/10.1145/336296.336340

T. H. Nelson. *Complex information processing: a file structure for the complex, the changing and the indeterminate*. In: *Proceedings of the 20th National Conference*. ACM Press, 1965; pp. 84–100. URL http://doi.acm.org/800197.806036

P. J. Nürnberg, K. C. Kristoffersen, U. K. Wiil, D. L. Hicks. *EAD revisited: First experiences*, 2005. Proceedings of the International Metainformatics Symposium 2005 (in press)

P. J. Nürnberg, J. J. Leggett, E. R. Schneider. *As we should have thought*. In: *Proceedings of the 8th ACM Conference on Hypertext*. ACM Press, 1997; pp. 96–101. URL http://doi.acm.org/10.1145/267437.267448

P. J. Nürnberg, U. K. Wiil, D. L. Hicks. *A grand unified theory for structural computing*. In: D. L. Hicks (ed.), *Proceedings of the International Metainformatics Symposium 2003*, vol. 3002 of *Lecture Notes in Computer Science*. Springer, 2004a; pp. 1–16. URL http://www.springerlink.com/link.asp?id=rqtq28uakc9l0ler

P. J. Nürnberg, U. K. Wiil, D. L. Hicks. *Rethinking structural computing infrastructures*. In: *Proceedings of the 15th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2004b; pp. 239–246. URL http://doi.acm.org/10.1145/1012807.1012868

Omni Group. *OmniGraffle 4*, 2005. URL http://www.omnigroup.com/ftp/pub/software/MacOSX/Manuals/OmniGraffle-4-Manual.pdf, visited on 2006-03-12

H. V. D. Parunak. *Don't link me in: Set based hypermedia for taxonomic reasoning*. In: *Proceedings of the 3rd Annual ACM Conference on Hypertext*. ACM Press, 1991; pp. 233–242. URL http://doi.acm.org/10.1145/122974.122998

K. Popper, V. Friedrich, W. Hochkeppel, T. Rotstein. *Kritik und Vernunft. Von der Unendlichkeit des Nichtwissens. Reden und Gespräche*. Der Hörverlag, 2001. 5 Audio CDs

W. Porstmann. *Das metrische Formatsystem*. Mitteilungen des Normenausschusses der Deutschen Industrie (1918) 200–202, 226–228. URL http://www.cl.cam.ac.uk/~mgk25/volatile/DIN-A4-origins.pdf

D. Raggett, A. Le Hors, I. Jacobs (eds.). *HTML 4.01 Specification*. W3C, 1999. URL http://www.w3.org/TR/1999/REC-html401-19991224/, W3C Recommendation 1999-12-24

Raskin Center. *Archy in the future*. WWW, 2006. URL http://rchi.raskincenter.org/index.php?title=Archy_in_the_Future, visited on 2006-03-26

RealVNC Ltd. *VNC Free Edition 4.1*, 2005. URL http://www.realvnc.com/products/free/4.1/, visited on 2006-03-12

J. Rekimoto. *SmartSkin: an infrastructure for freehand manipulation on interactive surfaces*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Press, 2002; pp. 113–120. URL http://doi.acm.org/10.1145/503376.503397

G. Robertson, M. van Dantzich, D. Robbins, M. Czerwinski, K. Hinckley, K. Risden, D. Thiel, V. Gorokhovsky. *The Task Gallery: a 3D window manager*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Press, 2000; pp. 494–501. URL http://doi.acm.org/10.1145/332040.332482

D. E. Rose, R. Mander, T. Oren, D. B. Poncéleon, G. Salomon, Y. Y. Wong. *Content awareness in a file system interface: Implementing the "pile" metaphor for organizing information*. In: *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM Press, 1993; pp. 260–269. URL http://doi.acm.org/10.1145/160688.160735

P. Russo, S. Boor. *How fluent is your interface? Designing for international users*. In: *Proceedings of the Conference on Human Factors in Computing Systems*. Addison-Wesley Longman, 1993; pp. 342–347. URL http://doi.acm.org/304459.304713

F. Shipman, R. Airhart, H. Hsieh, P. Maloor, J. M. Moore, D. Shah. *Visual and spatial communication and task organization using the Visual Knowledge Builder*. In: *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*. ACM Press, 2001a; pp. 260–269. URL http://doi.acm.org/10.1145/500286.500325

F. Shipman, J. M. Moore, P. Maloor, H. Hsieh, R. Akkapeddi. *Semantics happen: Knowledge building in spatial hypertext*. In: *Proceedings of the 13th Conference on Hypertext and Hypermedia*. ACM Press, 2002; pp. 25–34. URL http://doi.acm.org/10.1145/513338.513350

F. M. Shipman. *Seven directions for spatial hypertext research*. In: *1st International Workshop on Spatial Hypertext in Conjunction with the 12th ACM Conference on Hypertext and Hypermedia*. 2001; p. n/a. URL http://www.csdl.tamu.edu/~shipman/SpatialHypertext/SH1/shipman.pdf

F. M. Shipman, H. Hsieh, P. Maloor, J. M. Moore. *The Visual Knowledge Builder: a second generation spatial hypertext*. In: *Proceedings of the 12th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2001b; pp. 113–122. URL http://doi.acm.org/10.1145/504216.504245

F. M. Shipman, C. C. Marshall. *Spatial hypertext: an alternative to navigational and semantic links*. ACM Computing Surveys 31 (1999). URL http://doi.acm.org/10.1145/345966.346001

F. M. Shipman, C. C. Marshall, M. LeMere. *Beyond location: Hypertext workspaces and non-linear views*. In: *Proceedings of the 10th ACM Conference on Hypertext and Hypermedia*. ACM Press, 1999; pp. 121–130. URL http://doi.acm.org/10.1145/294469.294498

F. M. Shipman, C. C. Marshall, T. P. Moran. *Finding and using implicit structure in human-organized spatial layouts of information*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Press/Addison-Wesley, 1995; pp. 346–353. URL http://doi.acm.org/223904.223949

B. Shneiderman. *User interface design for the Hyperties electronic encyclopedia (panel session)*. In: *Proceeding of the ACM conference on Hypertext*. ACM Press, 1987; pp. 189–194. URL http://doi.acm.org/10.1145/317426.317441

A. Singh. *The PowerBook Sudden Motion Sensor*. WWW, 2005. URL http://www.kernelthread.com/software/ams/ams.html, visited on 2006-03-26

T. Smith, S. Bernhardt. *Expectations and experiences with HyperCard: a pilot study*. In: *Proceedings of the 6th Annual International Conference on Systems Documentation (SIGDOC '88)*. ACM Press, 1988; pp. 47–56. URL http://doi.acm.org/10.1145/358922.358931

U. Springfeld. *Gescheit, gescheiter, gescheitert. Wie lernt der Mensch?* SWF2 Radio, 2004. URL http://www.swr.de/swr2/sendungen/radioakademie/wer-weiss-was/themen/thema3/index.html, broadcasted on 2004-08-11 (first broadcast on 2003-05-17)

N. Streitz, T. Prante, C. Müller-Tomfelde, P. Tandler, C. Magerkurth. *Roomware®: the second generation*. In: *CHI '02 Extended Abstracts on Human Factors in Computing Systems*. ACM Press, 2002a; pp. 506–507. URL http://doi.acm.org/10.1145/506443.506452

N. Streitz, T. Prante, C. Müller-Tomfelde, P. Tandler, C. Magerkurth. *Roomware®: the second generation (demonstration movie)*. WWW, 2002b. URL http://ipsi.fraunhofer.de/ambiente/paper/2002/Roomware-ubicomp02.avi, in combination with Streitz et al. (2002a); visited on 2006-03-26

M. Toyoda, E. Shibayama. *HishiMochi: a zooming browser for hierarchically clustered documents*. In: *CHI '00 Extended Abstracts on Human Factors in Computing Systems*. ACM Press, 2000; pp. 28–29. URL http://doi.acm.org/10.1145/633292.633312

D. Tsichritzis. *Form management*. Communications of the ACM 25 (1982) 453–478. URL http://doi.acm.org/10.1145/358557.358578

M. Tzagarakis, D. Avramidis, M. Kyriakopoulou, M. M. C. Schraefel, M. Vaitis, D. Christodoulakis. *Structuring primitives in the Callimachus component-based open hypermedia system*. Journal of Network and Computer Applications 26 (2003) 139–162. URL http://dx.doi.org/10.1016/S1084-8045(02)00064-4

M. Tzagarakis, N. Karousos, D. Christodoulakis, S. Reich. *Naming as a fundamental concept of open hypermedia systems*. In: *Proceedings of the 11th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2000; pp. 103–112. URL http://doi.acm.org/10.1145/336296.336338

M. Tzagarakis, M. Vaitis, A. Papadopoulos, D. Christodoulakis. *The Callimachus approach to distributed hypermedia*. In: *Proceedings of the 10th ACM Conference on Hypertext and Hypermedia*. ACM Press, 1999; pp. 47–48. URL http://doi.acm.org/10.1145/294469.294482

United Nations. *Universal declaration of human rights*. Resolution 217 a (iii), United Nations, 1948. URL http://www.unhchr.ch/udhr/lang/eng.htm

M. van Dantzich, V. Gorokhovsky, G. Robertson. *Application redirection: Hosting Windows applications in 3D*. In: *Proceedings of the 1999 Workshop on New Paradigms in Information Visualization and Manipulation in Conjunction with the 8th ACM International Conference on Information and Knowledge Management*. ACM Press, 1999; pp. 87–91. URL http://doi.acm.org/10.1145/331770.331791

W. Wang, A. Fernández. *A graphical user interface integrating features from different hypertext domains*. In: S. Reich, M. M. Tzagarakis, P. M. E. De Bra (eds.), *Hypermedia: Openness, Structural Awareness, and Adaptivity. International Workshops OHS-7, SC-3, and AH-3*. Springer, 2002; pp. 141–150. URL http://springerlink.metapress.com/link.asp?id=kw55yxlvqjwknhp1

C. Ware. *Information Visualization*. Morgan Kaufmann, 2nd edn., 2004

M. J. Weal, G. V. Hughes, D. E. Millard, L. Moreau. *Open hypermedia as a navigational interface to ontological information spaces*. In: *Proceedings of the 12th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2001a; pp. 227–236. URL http://doi.acm.org/10.1145/504216.504270

M. J. Weal, D. E. Millard, D. T. Michaelides, D. C. De Roure. *Building narrative structures using context based linking*. In: *Proceedings of the 12th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2001b; pp. 37–38. URL http://doi.acm.org/10.1145/504216.504231

P. Wellner. *Interacting with paper on the DigitalDesk*. Communications of the ACM 36 (1993) 87–96. URL http://doi.acm.org/10.1145/159544.159630

S. Whittaker, J. Hirschberg. *The character, value, and management of personal paper archives*. ACM Transactions on Computer-Human Interaction (TOCHI) 8 (2001) 150–170. URL http://doi.acm.org/10.1145/376929.376932

U. K. Wiil, D. L. Hicks, P. J. Nürnberg. *Multiple open services: a new approach to service provision in open hypermedia systems*. In: *Proceedings of the 12th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2001; pp. 83–92. URL http://doi.acm.org/10.1145/504216.504241

U. K. Wiil, P. J. Nürnberg, D. L. Hicks, S. Reich. *A development environment for building component-based open hypermedia systems*. In: *Proceedings of the 11th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2000; pp. 266–267. URL http://doi.acm.org/10.1145/336296.336507

Wikipedia. *Myst*. WWW, 2006a. URL http://de.wikipedia.org/wiki/Myst, visited on 2006-06-27

Wikipedia. *Schloss Schönbrunn*. WWW, 2006b. URL http://de.wikipedia.org/wiki/Schloss_Sch%C3%B6nbrunn, visited on 2006-03-10

Working Party on Facilitation on International Trade Procedures. *United Nations layout key for trade documents*. UNECE Recommendation 1 (ECE/TRADE/137, Edition 96.1), United Nations, 1981. URL http://www.unece.org/cefact/recommendations/rec01/rec01_ecetrd137.pdf

M. Wu, R. Balakrishnan. *Multi-finger and whole hand gestural interaction techniques for multi-user tabletop displays*. In: *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology*. ACM Press, 2003; pp. 193–202. URL http://doi.acm.org/10.1145/964696.964718

Y. Yamamoto, K. Nakakoji, A. Aoki. *Spatial hypertext for linear-information authoring: Interaction design and system development based on the art design principle*. In: *Proceedings of the 13th ACM Conference on Hypertext and Hypermedia*. ACM Press, 2002a; pp. 35–44. URL http://doi.acm.org/10.1145/513338.513351

Y. Yamamoto, K. Nakakoji, A. Aoki. *Visual interaction design for tools to think with: Interactive systems for designing linear information*. In: *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI2002)*. 2002b; pp. 367–372. URL http://www.kid.rcast.u-tokyo.ac.jp/~kumiyo/mypapers/AVI2002.pdf

K.-P. Yee. *Two-handed interaction on a tablet display*. In: *Extended Abstracts of the 2004 Conference on Human Factors and Computing Systems*. ACM Press, 2004; pp. 1493–1496. URL http://doi.acm.org/10.1145/985921.986098

Index