Folio: Augmenting Books on Interactive Tabletops

Bachelor Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Christian Cherek

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Karin Herrmann

Registration date: Jan 15th, 2012
Submission date: Apr 25th, 2012

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, April 2012
Christian Cherek

Contents

List of Figures

List of Tables

Abstract

In the context of digital humanities literary scholars have to work more and more in digital environments. Many publishers decide to do both, print the real book, and give access to digital contents regarding the work.

In this thesis we present a project plan for a software on a multitouch table that combines the advantages of real world physical books with the high accessibility of digital contents. It is even possible to attach digital materials to real books. To create this plan we did several interviews with literature professionals.

Furthermore we implemented the first steps to get this system running and present them in this thesis as well. We describe the organization of the workspace to keep the project clean and comprehensible.

Future work will be the implementation of the remaining steps suggested in the project plan in chapter 4—"Planning *Folio*". Also there are open research questions on how our system changes the workflow of literary scholars.

Überblick

Im Kontext der "digital humanities" arbeiten Literaturwissenschaftler heute mehr und mehr in digitalen Umgebungen. Viele Herausgeber entscheiden sich inzwischen ihre Werke sowohl in gedruckter, als auch in digitaler Form im Internet zur Verfügung zu stellen.

In dieser Bachelorarbeit präsentieren wir die Projektplanung für eine Software auf einem Multitouch-Tisch, die einerseits die Vorteile der Arbeit mit gedruckten Büchern und andererseits die Möglichkeiten der digitalen Vernetzung und Darstellung von Materialien miteinander vereint. Es ist sogar möglich, digitale Materialien so an physikalische anzubinden, dass diese immer zusammen erscheinen.

Um den Projektplan zu erstellen, wurden mehrere Interviews mit Literaturwissenschaftlern durchgeführt und hier präsentiert. Ausserdem wurden die ersten Schritte zur Implementierung dieser Software gemacht und hier vorgestellt. Wir beschreiben die Ordnung des Projektes, um den Programmcode möglichst übersichtlich und verständlich zu halten.

Zukünftige Arbeit an dem Projekt beinhaltet vor allem die Durchführung des in Kapitel 4—"Planning *Folio*" erarbeiteten Projektplans. Ausserdem gibt es aus Sicht der Human Computer Interaction offene Fragen, inwiefern ein System wie *Folio* die Arbeitsabläufe und Gewohnheiten von Literaturwissenschaftlern ändern könnte.

Acknowledgements

There were a lot of people who helped me to get this thesis done, here I want to name the most important among them.

Thank you Prof. Jan Borchers and Prof. Karin Herrmann for your supervising support and your illuminating feedback on my cautious first steps in this thesis.

Thank you very much Moritz Wittenhagen and Max Möllers. Your feedback on my progress, the tips you gave when I got stuck, your motivation and the constructive criticism during this thesis meant really a lot to me.

Thank you Lara without you nothing of this would have happened at all.

And big thanks to my whole family, especially Papa, I could ask for your help every day and night, no matter what, you always had a helping hand, ear or eye for me.

Thank you all.

Conventions

Throughout this thesis we use the following conventions.

Names of software or widgets are written in italic text.

Definitions of technical terms or short excursus are set off in coloured boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Source code and implementation symbols are written in typewriter-style text.

```
myClass
```

The whole thesis is written in American English.

Download links are set off in colored boxes.

File: myFile[a]

[a]http://hci.rwth-aachen.de/public/folder/file_number.file

Chapter 1

Introduction

This thesis presents the design and implementation process for a software, called *Folio*, on a 1,8x1,4 m^2 scaled high definition multitouch table. *Folio* is intended to support literary scholars in research projects, by building the bridge between real world books and digital media.

As Deininghaus [2010] found in his thesis on interactive surfaces for literary criticism, it is necessary for literary scholars to maintain on their habit of using real world books instead of switching to a complete digital version of their working material.

Deininghaus also designed and created several prototypes of a system, which accomplishes these needs. His software prototype had been developed further for two years. During this development we found several reasons to rebuild the software.

In this thesis we present a project plan to rebuild this software. We did the next step through the DIA cycle, analyzing the existing software, designing the next iteration, and started to develop the next implementation step.

Definition:
DIA Cycle

DIA CYCLE:
The "Design Implement Analysis Cycle" (DIA Cycle) is a software development process that iteratively improves a software. In the "Design" step a prototype of the software is designed. This could include a bug list that needs to be fixed, or just brainstorm sessions on what the software needs to do. The "Implement" step uses the design to implement a prototype. This implementation gets more and more concrete with each cycle. The "Analysis" step tests the prototype and evaluates it. This could be done in a user study or with other feedback. The analysis is used to design the next prototype.

We did several interviews with literary scholars from the Humtec institute at RWTH Aachen University, to examine how they would like the system to be. Based on these, we created the project plan that is presented here. The interviews are presented in chapter 4.1—"Initial Interviews".

In chapter 5—"*Folio*" the implementation progress is described. We name the important classes, describe which parts are implemented already and what should be done in the future. Most of this classes are part of the graphical user interface, but there are also descriptions for data management classes and the workspace organisation.

At the end we summarize our progress on the project plan and give a suggestion on the workflow for succeeding colleagues. For this workflow we recommend a close cooperation with the literary scholars to build a system, that can be used for literary research projects.

Chapter 2

Related work

2.1 The Multitouch Table

Folio is a software for a multitouch table, designed and built at Media Computing Group. In figure 2.1 you can see a conceptual drawing of this multitouch table.

Folio runs on a multitouch table.

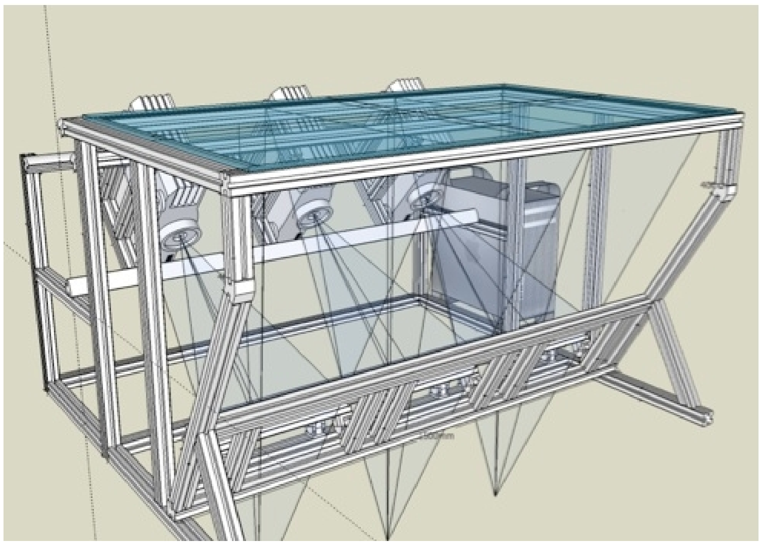


Figure 2.1: Conceptual drawing of our multitouch table.

Three high definition projectors are used to display the screen contents. Via mirrors at the bottom of the table the

Three projectors create the screen.

images are projected to the downside of an acrylic surface. These projectors are connected to a desktop computer next to the table. This computer is used to start the system, after that our users will not need to work at the computer anymore.

A finger touch is detected with the FTIR technique.

To detect finger touches on the surface we use a technique called *Frustrated Total Internal Reflection* or *FTIR*, figure 2.2 shows how this works. The acrylic surface of the table is filled with infrared (IR) light sideways. This light stays inside the surface until it is disturbed by a touch. This can be seen by IR-cameras on the bottom of the table, thus finger touches can be detected.

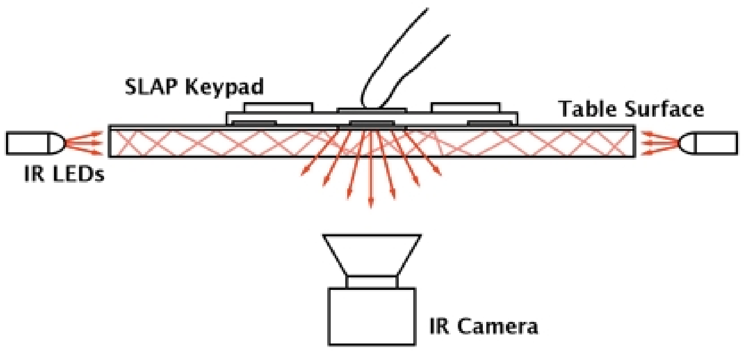


Figure 2.2: *Frustrated Total Internal Reflection* is used to detect finger touches.

Objects need markers to be seen by the camera.

Physical Objects on the surface are detected, using a technique called *Diffuse Illumination* or *DI*. All objects that should be detected by our system need to have markers on their downside. These markers get illuminated by IR-Leds on the bottom of the table, and the reflection on the markers is detected by the camera. This way even light weighted objects can be detected on the surface. Figure 2.3 shows the concept of *DI*.

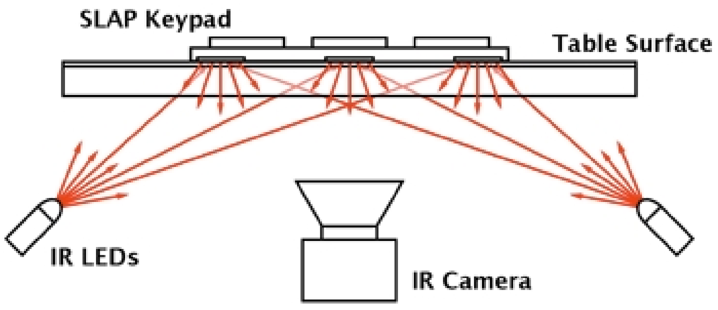


Figure 2.3: The concept of *Diffuse Illumination*, this way markers under objects are detected.

2.2 Deininghaus's Diploma Thesis

Folio builds up on Deininghaus's diploma thesis.

Stephan Deininghaus did the first two DIA cycles on a system that supports literary scholars in research, by combining physical and digital material, in [2010]. He created the first software prototype on our multitouch table. This software was called *Apparatchik*.

His studies showed that a system like this arouses a great deal of interests within the humanities. In his future work section he motivated a further development of the system, that should include an integration of web services and databases. Also he named several situations where collaboration or sharing features would be useful.

We used his experiences to developed a system that offers the desired features and extends it further. We added a web browser and save and reload features, which could be used to create and save multiple different working processes. This was not possible with *Apparatchik* until now.

2.3 *SLAP*

SLAP offers the possibility to attach physical objects to our multitouch table.

In [2008] Weiss et al. presented *SLAP* which is a shortcut for **S**ilicone **I**lluminated **A**chive **P**eripherals. *SLAP* is a framework that offers multitouch tables the possibility to attach physical widgets.

The great improvement above just simple onscreen widgets is, that these widgets offer haptic feedback. At the same time they benefit from the onscreen possibilities. A knob, that can be turned can first be used to select a functionality and then secondly control the selected functionality. On the webpage of Media Computing Group you can find additional information[1] , that shows all possibilities of SLAP.

We used this framework to build *Folio* on top of it. It comes with touch detection for fingers, as well as with marker detection for physical objects. Also it offers a set of widgets that can be used to display different contents on the screen. Therefore we decided to work with *SLAP* for our next iteration on *Folio*.

[1]http://hci.rwth-aachen.de/slap

Chapter 3

Motivation

3.1 The Task

Folio is created for literary scholars. Specifically professionals who analyze literary texts by referring to different versions of a existing text. Therefore scholars create so called editions, which are defined by Plachta [1997] as "a reliable text that provides the basis for any historical or interpretative examination". Our system can be used to do both, creating an edition and working with an edition to analyze texts. The exact process was described by Deininghaus [2010] in chapter three of his diploma thesis.

Folio will be used for literary research.

EDITION:
"A reliable text that provides the basis for any historical or interpretative examination." (Plachta [1997])

Definition: *Edition*

When creating an edition a, scholar collects many different versions of a literary text. He subscribes handwritten manuscripts and chronologically orders the different versions. That can be quite difficult. For example old handwritten versions can be hard to read or the initial order is not clear. In figure 3.1 you can see a manuscript of a poem by Ernst Meister, his handwritings sometimes are really difficult to read.

Working at a desk with movable items is indispensable for our users.

Literary scholars usually spread their material on a desk.

As Deininghaus [2010] examined, most users traditionally solve this task by spreading all versions on a table and trying to sort them in the correct order. To transcript a hardly readable text they make notes and discuss their ideas with other scholars.

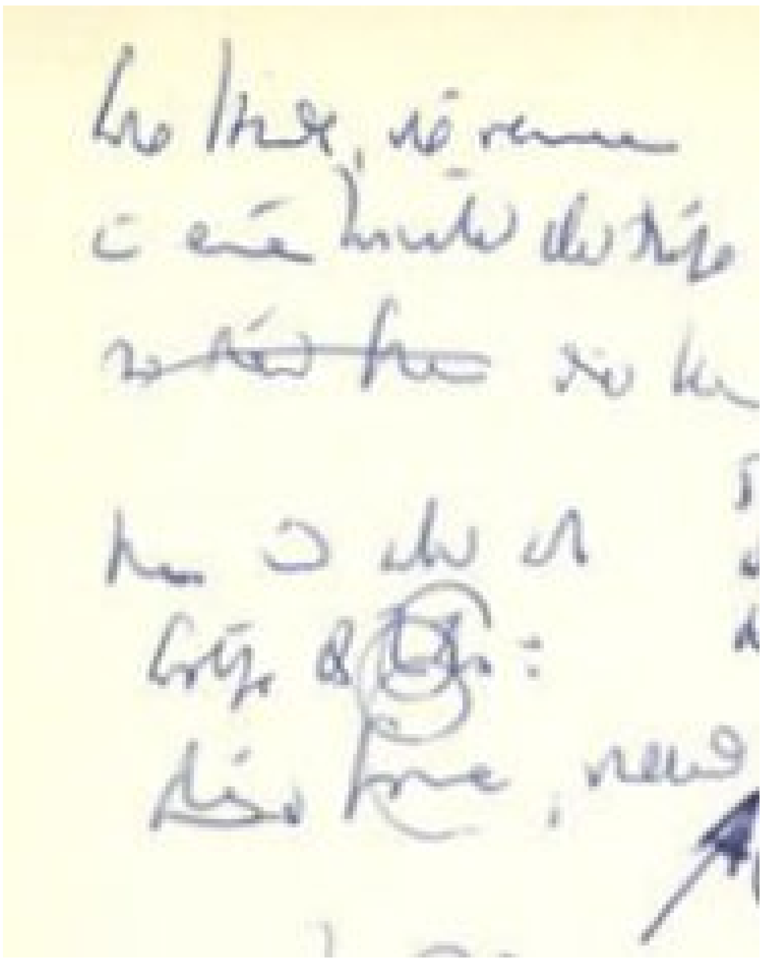


Figure 3.1: A handwritten manuscript of a poem by Ernst Meister [2011].

Online research takes a lot of time.

An edition is used to analyze literary texts. The scholar takes certain milestones in the creation of e.g. a poem, and examines the differences between these versions. Doing this, it is possible to get a better understanding of the researched text. To solve this task our users also make heavy usage of the internet. They search referenced texts or try to find out the meaning of outstanding terms in a literary text. Therefore they need access to online databases or library systems where these references might be stored. Also some publishers decide to release their edition not only in a printed version but also in a digital version. In this digital version the scholar is able to see copies of the original manuscripts, which were not printed in the actual edition. These copies can be enlarged on the computer to get a more detailed view as a printed version could offer it.

3.2 *Apparatchik*

The current version of our Software is called *Apparatchik*.

In [2010] Deininghaus created some first prototypes to build a system, which achieves our goals. His software prototype was named *Apparatchik*. *Apparatchik* was able to display digital representations of books, and scanned single pages. Figure 5.6 in chapter 5.2.4—"`Book`, `BookController` and `BookView`" is a screenshot of a *Book* in *Apparatchik*.

Also there is a possibility to connect books via *Links*. These *Links* were displayed as green arrows next to the *Book* (see figure 3.2). Pressing on these arrows brings up either the referenced page as a single *Clip* item, when the actual book is not placed on the table, or a popup pointing on the referenced book, if this is anywhere on the table.

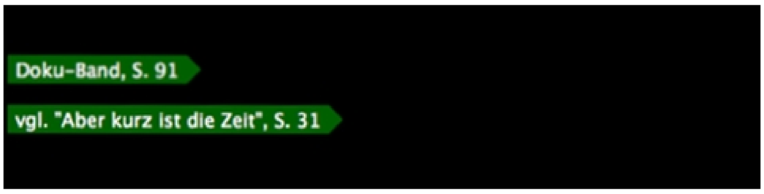


Figure 3.2: The green arrow represents a direct link to another book.

Apparatchik has a lot of the desired functionalities.

Another feature Deininghaus already implemented was the possibility to attach digital, single page copies to a specific page of a book. The enhanced page got a *Tab* next to it, which could be enlarged by pressing the tab symbol. In figure 3.3 the minimized and maximized version of the tab can be seen. If this *Tabs* hold more items than displayable, the *Tabs* could be scrolled up and down via touch gestures.

After Deininghaus left the Media Computing Group several colleagues built up on *Apparatchik*. *Apparatchik* was enhanced with an internet browser and a marker detection to detect which book is placed on the table and attach the corresponding digital representation to it. The browser is attached to a bluetooth keyboard, so that it always appears above this keyboard. For a more detailed view on *Apparatchik*'s features see chapter 5—"*Folio*".

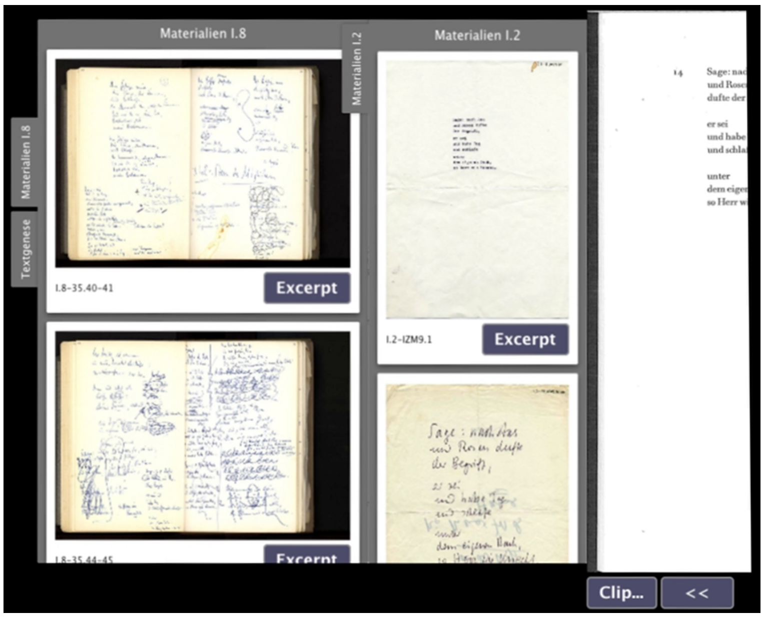


Figure 3.3: The *Tab*-functionality of the *Book* widget. Two *Tabs* are opened, the third one "Textgenese" is closed.

Deininghaus's user tests helped to improve our design.

In the current version *Apparatchik* is already really advanced. The running version was used to demonstrate a typical working situation of a literary scholar, as it could look like with a multitouch table. Also Deininghaus did some user studies where our users showed great interest in the system. These studies also provided feedback for our implementation. For example the *Link* reference pop up for a not yet placed book was mistaken for an error message. That showed us that this feature needs to be realized differently this time.

Unfortunately the latest version of *Apparatchik* has some drawbacks. Some of these drawbacks are hardware constrained and we are not able to solve these without rebuilding the whole system. But others of these are software based. Improving these problems is the main task of this thesis.

Apparatchik is not yet finished.

3.2.1 Drawbacks

The biggest drawback the system has are performance problems. Especially when there are many items on the table the responsiveness gets more and more lost. Sometimes the system even loses user input or confuses two actions as a single one. This can lead to completely unpredictable behavior.

A lack of performance is the biggest disadvantage.

The second problem is a very unstructured code. The original software architecture is weaken up on many places within the code. This leads to crashes of the whole system which are hard to detect. Even the try to find some memory issues with *Instruments* - a debugging tool provided by Apple - could not help to find the crash cause.

To use the system as a complete working environment for literary scholars there are still a lot of missing features. Deininghaus [2010] did some interviews on how the system should work, and which functionalities should be provided. Now two years later the literary scholars have some experience with the running system, so we reinterviewed them to find out if their preferences are still the same.

There are missing features as well.

The interaction with the table elements should get improved too. The way to choose a page in the digital representation of a book for example is not intuitive enough right now. In this version the user has to choose the page via a slider widget, which is placed under the digital book (see figure 3.4). This is quite difficult to handle especially when the book has lots of pages, because the sensitivity of the slider widget is to high there. That leads to problems choosing a single page.

Some interaction flaws complicate the usage.

In [2005] Hurst et al. presented a solution, the *Zoomslider*, that could improve the performance on our slider, however for *Folio* we aim to get a automatic recognition of the current page by taking photos of the screen and reading the selected page automatically. With automatic page recognition we would not need this slider anymore, since our users are able to let the system check which page should be displayed.

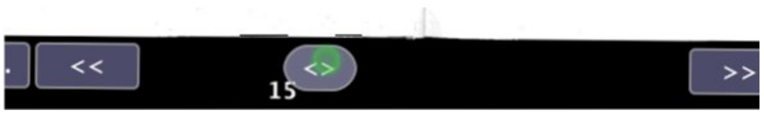


Figure 3.4: Below a book widget is a slider to change the displayed page. The green dot represents a touch.

3.2.2 Where do they come from

There are two big performance leaks.

The performance issues have two origins. First the hardware used to detect the touch events needs some time to process the information, especially when there are very many touches or objects on the surface (e.g. more than 30 touches). Unfortunately we cannot handle these delays now, since we would need to change the hardware of the system. Because this is not possible right now, we concentrated on improving the software side performance problems.

Exact delays are hard to measure.

Measuring the exact delay was difficult. We started by adding log messages into the system. But these messages could not measure the time *Apparatchik* needs to render the contents on the screen. Doing this we found that our touch detection, the time between the actual touch and the touch event in *Apparatchik*, needs 0.2 - 0.3 seconds.

We made some informal measurements with a stopwatch, that showed delays of more than 0.5 seconds with only three active components on the table. With two books, a browser and two clips most actions on, for example the browser, got completely lost.

Unfortunately we could not come up with a proper way of measuring the exact delay. But our informal measurements together with the fact that the touch detection does not slow down with this amount of touches definitely show, that the major performance issue is rooted in *Apparatchik*.

Delays of more than 0.5 seconds were not acceptable for us.

We spotted the main reason for the delay in *Apparatchik* is rooted in the decision to build up the system on the Core Animation framework provided by Apple. This framework encounters large performance issues when displaying contents over multiple screens, which are moreover connected to more than one graphic card. Since this framework is provided by Apple, we are not able to improve this any further ourselves; that's why we decided to rebuild the whole system on another framework.

The used framework is not able to display fast graphics on more than one screen.

CoreAnimation:
CoreAnimation is a framework consisting of a collection of Objective-C classes for graphics rendering, projection, and animation.

Definition: *CoreAnimation*

The messed up code structure is another reason for us to rebuild the whole system. Since Deininghaus finished his thesis at the Media Computing Group, his software prototype was developed further. This was done by many different people, everybody with his own understanding of the code and his own opinion how this should be structured. There was no project plan our colleagues could refer to. Since *Apparatchik* is a quite big software (approximately 30 000 lines of code) this led to a really unstructured workspace.

Many different developers messed the code structure.

We provide a project plan that includes a structured workspace to prevent the new code from getting cluttered as well. Also we provide a work plan when which feature should be implemented, so that colleagues in the future can refer to it.

Our interviews with literary scholars showed some changes in the preferences which functionality is needed for the system. At Deininghaus's interviews the participants had no clear idea off how the system could look like, and which features are possible to create. Now with

The users needs developed over the past years.

two more years experience the scholars have a clearer idea which functionality is needed and possible for our system.

The improvement of the interaction with the table is a big topic. There are already some ideas on automating for example the page detection of a physical book. And there is ongoing research at the Media Computing Group on improving the clarity of the workspace.

3.3 Chose Another Framework

We built up on a framework, developed at Media Computing Group

Since we are not able to improve the Core Animation framework by Apple, we decided to build up on another framework. At Media Computing Group Malte Weiss et al. [2008] created a framework which is designed to run on multitouch tables.

3.3.1 SLAP

SLAP was developed for similar systems, and offers faster graphic rendering.

The SLAP Framework created by Weiss et al. is a good framework to build our project up on. It is made for software that runs on multitouch tables and brings the possibility to combine physical objects with digital contents. It provides a detection mechanism for physical objects we can adopt to detect books on the table.

Furthermore since it was developed at the Media Computing Group we were able to extend or improve it if this was needed. The SLAP framework uses OpenGL to render contents which should fix the rendering problems we had with Core Animation.

Chapter 4

Planning *Folio*

Folio has been and will be implemented following a specific project plan. This plan was created to prevent the source code from becoming as unstructured as the previous system. Furthermore the realization of *Folio* will take more time than we had to write this thesis and the plan could be used to finish the implementation. Succeeding colleagues will be able to use this plan to finish *Folio*.

This thesis provides a project plan for *Folio*.

To create a reference on which we can implement this system we made the following steps. First we had some brainstorm sessions in the computer science team which steps are needed to get a system like this running. In figure A.1 you can see a diagram we created in one of our sessions. Second we did several interviews with literary scholars to examine their vision of the system. To create the plan we put the gathered information together and created a plan that supports the wishes of literary scholars and is reasonable for computer scientists alike.

The plan was created with interviews and brainstorm sessions.

4.1 Initial Interviews

Since the whole project is done in a close cooperation with the literary scholars, we regarded their wishes which features and in which order they should be implemented. Therefore we did three interviews with literary scholars

We set great value to the needs of literary scholars.

to investigate their needs. We interviewed the scholars in their normal working environment. Furthermore we let them try out the old version to let them think about the system. We did the interviews consecutively and asked them not to talk about the interviews, before all interviews were concluded.

The interviews consisted of three parts.

The interviews were structured in three parts. The first task we gave our interviewees was to brainstorm which features are needed to create a system that will be useful to them. The interviewees were asked to name functionalities they need to do their research at the multitouch table. We collected all of their answers on a board so that they were able to see them all the time. In the second part the participants were asked to sort the answers they had given earlier in a reasonable way. Reasonable in this context should mean, if they could decide which feature should work first to do something, which is the most important. If the interviewees had some other ideas they forgot in the first part they were allowed to add them as well.

1. Naming all desired functionality.

2. Sorting their ideas in a reasonable order.

3. Collect some drawbacks of the current version.

The last part consisted of a short collection on which problems or drawbacks they found while using the existing system. They should mention every problem they could come up with the big really annoying as well as the little just a little unhandy ones.

On all of these parts they were allowed to change or add something to their previous answers. Everything they said was collected on a whiteboard; we explicitly encouraged the interviewees to think about the previous questions over the whole interview. We did this because we wanted to let them think about their experiences they had with the system. This way they were able to come up with things they thought about some time ago as well.

Our goal was to find out, what they want to do, which features they think they need, and if these ideas changed in the last two years.

Within the interviews we tried to find out three major questions. First what do they want to do with the system, and what their imaginations are what is possible with the multitouch table. Second which features they need to do research with it, and third in which order they think they need their functionality back. The first two questions towards the functionality and the use case were also treated in the interviews Deininghaus [2010] did for his thesis. On this part we payed close attention, if, and how, our participants ideas of the system changed since Deininghaus interviewed them.

All interviewees are literary scholars who are creators of editions. That means they are deeply into the process of working on and with an edition and have a deep understanding of the process at working on editions.

The task our interviewees want to solve is alike among them.

The answers to the first question, what to do with the system were quite similar. The system was created to fulfill a specific task. Namely to do research in the context of creating and working with editions. That means heavy online research, and equal efforts using printed books or handwritten manuscripts of the author under research. All participants described their workflow as spreading out the materials on their desk, sorting them in a possible order and making notes on each of them. All participants expressed their disapproval about loosing this collection when they clean up their desk. Their normal workflow contains of a second part in which they transcript the handwritten manuscripts of the author and their own notes into digital text. For the transcription of the manuscripts they need a word processor with lots of possibilities to edit the text.

4.1.1 Gather Ideas

The desired features were more diverse.

The answers on our question, which features they need to fulfill this task, were more diverse. The results are collected in table 4.1, 4.2, and 4.3. Every participant expressed a high need for connection to the internet. Also everyone mentioned the possibility to attach digital contents to a physical book as a required function. Also the table should have the ability to display digital copies of their research material.

One interviewee mentioned a word processor as desired feature.

One interviewee mentioned the system would only be fully useful if it would also be possible to do the actual writing processes at the table. That includes a text editor with extensive possibilities for editing and writing texts. The other participants wanted to use the table not that much for actual writing the edition, but more sorting the materials and making small annotations on the books and digital materi-

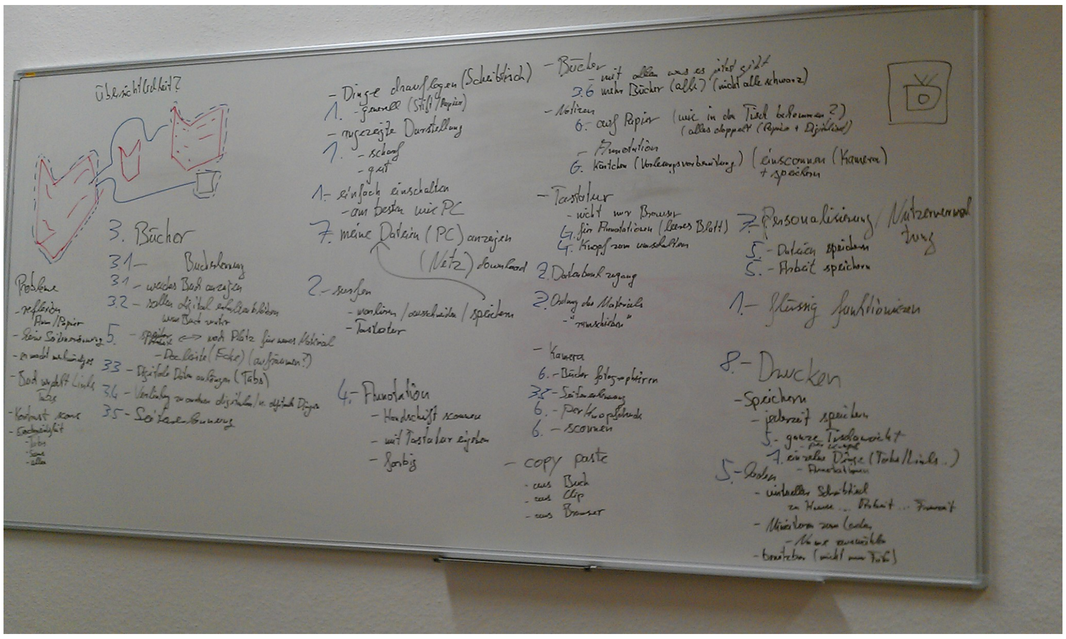


Figure 4.1: A mindmap created within our initial interviews.

als. Therefore these interviewees mentioned a connection to their desktop computer or even mobile devices as a need to do research at the table.

Saving and restoring the work process was mentioned by all interviewees.

Also a possibility to save and restore previous work perhaps even on user based working environment was mentioned as needed. To get copies of handwritten manuscripts or annotations an author made every interviewee mentioned on the one hand the connection to the internet with access to databases. And on the other hand a fast scan function, which just creates a photo of the table and offers the possibility to create digital representatives of physical materials. You can see a collection of desired functionalities in figure 4.1 collected on a mind map.

4.1.2 Sort Ideas

The last task we gave to the interviewees was to sort the functionalities in a way they would desire them to work again. Although real research can only happen when the whole system is working with all functionalities there are more important features and less important ones.

The feature lists in table 4.1, 4.2, and 4.3 are already sorted as our interviewees thought how important each feature is.

	Interviewee 1
1.	Browser
2.	Books
3.	Attaching digital materials on books
4.	Clips
5.	Links between materials
6.	Page recognition
7.	Digital annotations
8.	Save and restore working environment
9.	Camera scans of physical documents
10.	User management
11.	Connection to other working stations (home PC)
12.	Printing

Table 4.1: List of features our first interviewee found necessary

Internet connection is essential for al interviewees.

All participants rated the internet connection with a very high importance. They described a widget that is more or less like an internet browser used on a normal desktop pc. It should be able to connect to online databases and search engines.

Displaying materials like books or single page copies was rated second.

The second and third features should be displaying the materials. Single paged contents like scanned manuscripts on the one hand, and books on the other hand. The single page contents should be freely movable and resizable on the desk. Interviewee 2 rated the creation of a word processor more important than displaying material, but for him displaying contents had the highest need right behind the text editor.

These materials should be created by downloading them from online databases or just by creating a screenshot of the browser or a book on the table. Two participants also mentioned that it would be great to create these screenshots of physical material that is placed on the table, like a piece of paper with some handwritten notes, but this functionality

	Interviewee 2
1.	Clips
2.	Browser
3.	Automatic sorting
4.	Word processor to subscribe manuscripts
5.	Edit digital material (e.g. with text marker)
6.	Books
7.	Attaching digital materials on bookss
8.	Links between material
9.	Searching on the table
10.	Searching the internet
11.	Connect materials (to create an order)
12.	Digital annotations

Table 4.2: List of features our second interviewee found necessary

was rated less important than displaying downloaded data.

The connection of digital book representation to the real physical book should be the next step.

The books should be displayed digitally, but also be connected to a real book if it is placed on the table. To extend the possibilities of the book representative is the next desired step. The books have to be able to attach the other materials next to them. All participants described a folder like extension on each page of the book to save the materials in it. "I just want to drag a manuscript into this folder and save it there" one participant described it.

Annotating the digital materials was ranked lower.

The next step should be a possibility to create "Post-it" like annotations on the table. They do not need lots of editing functionality, but two participants expressed a wish to copy them to their desktop easily for further usage. These annotations should be created with a wireless keyboard connected to the multitouch table.

Other functionalities like saving the work situation, automatic sorting, printing the material and different user profiles were not that important to our participants.

	Interviewee 3
1.	Browser
2.	Searching the internet
3.	Clips
4.	Books
5.	Attaching digital material on books
6.	Save and restore working environment
7.	Connection to other working stations (home PC)
8.	Digital annotations
9.	Edit digital material (e.g. with text marker)
10.	Camera scans of physical documents
11.	Automatic sorting
12.	Links between material
13.	Printing
14.	Page recognition
15.	User management

Table 4.3: List of features our third interviewee found necessary

4.2 Create a Reasonable Project Plan

Out of our interviews and meetings within the computer scientists we created a project plan.

With these initial deliberations in mind, we created a project plan for the realization of *Folio*. Since our interviewees were no computer scientists they had no software architecture in mind when they thought about their desired features. For example the "save and restore the working environment" functionality needs to be done in several steps. First the system needs a consistent data management for material we created for the system as well as downloaded material.

In figure A.2 you can see a time schedule with a feature list we created out of these interviews. The time schedule also displays the estimated time each implementation step should need in our opinion. The rhombi represent certain milestones in the implementation of *Folio*.

4.3 Progress at the Actual Implementation

In the actual realization of our project plan we worked on some of the described milestones simultaneously. The creation of the *Browser*, *Book* and *Clip* widgets is quite progressed. This is indicated by the darker blue timelines in figure A.2.

Also the data structure to save the context of *Folio* is already working, but the functionality to save online material and attach it to books needs to be done after this thesis.

For a detailed description on how far *Folio* is developed, and which parts still need to get done see chapter 6—"Progress Evaluation"

Chapter 5

Folio

In this chapter we shall describe *Folio* as it will look like when it will be finished. Each section will describe another part of the *Folio* user interface, and how far the implementation progress has come so far.

5.0.1 The Model View Controller Paradigm

Folio uses the MVC Paradigm.

Folio makes heavy usage of the Model View Controller(MVC) Pattern Erik M. Buck [2012] 5.1. All interface widgets always consist of these three parts.

The model holds the data.

The model is a container for the data. Holding the information which is displayed, as the current document, the page or the current position on the screen. The Model is independent of the presentation or the controller of the data. It only provides getters and setters for the data.

The view displays the data on the screen

The View is the onscreen representation of the Model. It sets the appearance of the interface widget. Which colors are used for the buttons, where a button is, and calls the methods to handle button presses. Every view knows its model and its controller. The view displays the model data and calls the controller to access the data.

The controller is responsible for the interaction

The controller is responsible for the interaction. It manipulates the data corresponding to the user input, and responses to the actions the view called. In *Folio* every view controller is a delegate for it's view and has to implement the `-(void) executeCommand:(int) commandNamefromSource:(id)source;` method. In *Folio* the controller also is responsible to set up its view when it is created. That means there is a close one to one relationship between every view controller the corresponding view and model.

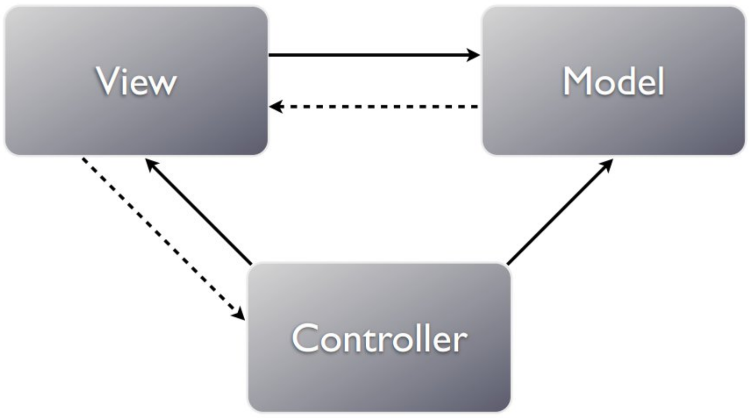


Figure 5.1: The Model View Controller Paradigm

Folio has a model, view and controller class for every interface object.

In *Folio* the whole interface is build up on this paradigm. For each instance of an user interface object, there is an instance of a specific model class, the corresponding view class and the controller class (e.g. `Book`, `BookView` and `BookController`.

In our implementation every model and every view has a weak reference to the associated controller class. The controller takes the data from the model, updates the view and takes the user input to update the model.

Our model for example holds the information where on the table the view is placed. This is done to be able to save the working situation, and bring it back for later usage.

In figure A.3 you can see a class diagram of *Folio*. The different parts of the MVC paradigm are marked as boxes.

5.1 Data Structure

Folio has to manage three different types of consistent data. The arrangement over the whole table, the contents which enhances a specific book, and the pdf and jpg files which are displayed by *Books* and *Clips*.

There are three types of information *Folio* has to store.

System State

The current working situation in *Folio* is defined as the digital objects currently displayed and the position of each of them. Objects are *Browsers*, *Books*, *Clips*, and *Annotations*. The model class for each object contains a `affineTransform` property, which saves position, rotation and zoom factor.

The current state describes every active widget, it's position and inner state.

When Folio gets started the latest state is loaded; the `LivingObjectController` is responsible for this. Every object is recreated and brought back to the saved state. That includes position, rotation and zoom. For the browser the internet page is loaded, a book will display the latest page.

These information get loaded at startup.

Of course we are not able to bring back the real world items which were placed on the table, but since the loaded table state is fully interactive the widgets will respond to replaced books, or touch input. This way a working situation of the previous day can be continued without a lot of rethinking how the working materials were placed on the table.

Until now the current table state is saved when the user hits a button placed at the edge of the table. We decided to implement it this way, since we wanted to offer the user a possibility to decide if and when a working situation is saved. A time triggered saving could also save useless working situations.

Saving is done by button press, to avoid useless savings.

We are aware, that this could lead to losing contents the user did not want to lose, to solve this problem we refer to the future work section. Perhaps additional user studies could give a hint which version is more useful.

Displayed Contents

All files that are displayed in *Books* and *Clips* need to be stored separately.

The displayed contents consists of pdf and or jpg files. The scanned books and manuscripts are pdf and the browser can download jpg or pdf files. All these files do not change very often. Sometimes users will download additional files, but its unlikely that they change a scanned manuscript or book. Thus we decided to manage these files with the Core-Data framework provided by Apple. The model class of a table widget is responsible to handle this.

Enhancements on our *Books*

Attached items in *Books* need to be stored as well.

The enhancements on a book will get managed by the `BookController` class. Every `BookController` manages a `plist` file similar to the one, the `LivingObjectsController` has for the whole table. These changes will be saved automatically, when the user changes something. We are of the opinion that this will not lead to useless savings, since both the situation before and after are fixed states and not steadily changed.

These saving operations should be done automatically.

Attaching Objects to a book will be done by dragging the object into the *Tab*. To remove it, the user simply has to drag it out of the tab again. This way *Clips* and in the future *Annotations* can be added to the page.

The `LivingObjectsController`

The `LivingObjects-Controller` is used to manage the widget creation, destruction and saving.

The LivingObjectsController takes care of the creation, saving, loading and destruction of every widget on the table. Table 5.1—"Overview for the `LivingObjectsController` class" shows a overview for the `LivingObjectController` class.

The `NSMutableArrays` are containers for the currently active widgets on the table. These arrays are filled when *Folio* starts and the `loadLivingObjectsOnUITK` method is called.

Class
 	
inherits from
		`LivingObjects Controller`
		`NSObject`

Properties:		
`livingBooks`	strong	NSMutableArray*
`livingClips`	strong	NSMutableArray*
`livingBrowsers`	strong	NSMutableArray*
`livingAnnotations`	strong	NSMutableArray*
Class methods:		
`(id)init;`		
`(void)loadLivingObjectsOnUITK: (SLAPUITK*) uiToolKit;`		
`(void)saveLivingObjects;`		
`(void)addLivingBook:(Book*) book onUITK:(SLAPUITK*)uiTollKit;`		
`(void)removeLivingBook:(BookController*) book fromUITK:(SLAPUITK*) uiToolKit;`		

Table 5.1: Overview for the `LivingObjectsController` class

First these four arrays were implemented as one array, which should contain all objects. We changed this to the current solution, because this made the saving and reloading a lot easier. Each widget on the table needs to save different values for it's current state. By putting them in independent lists, the conversion between the *plist* file and the running system is much nicer now.

Individual arrays for each object ease the saving and reloading.

By changing these we made it harder to add additional widgets on the table. To build in new widgets, the `LivingObjectsController` now needs to get an additional array and needs changes in the `saveLivingObjects` and the `loadLivingObjectsOnUITK:` methods, but changing these should not be that difficult, therefore we decided to stay with multiple arrays.

The methods `addLivingBook: (Book*) book onUITK: (SLAPUITK*) uiToolKit;` and `removeLivingBook: (BookController*) book` `fromUITK: (SLAPUITK*) uiToolKit;` are listed representative for similar methods to add or remove the different widgets on the table.

5.2 The Graphical User Interface

We created a set of interface widgets for *Folio*.

The graphical user interface (GUI) of *Folio* consists of several different graphical widgets, which grant the possibility to interact with *Folio*. Until now there are three types of widgets used; the *Book*, the *Browser* and the *Clip* widget. The *Book* widget can be attached to real world objects (e.g. the *Book* widget). The others are stand alone.

A fourth widget, the *Annotation* will be implemented in the future. This widget will allow to write simple texts similar to a real world Post-it paper. These *Annotations* will also be attachable to *Books*, like the *Clips* are now.

5.2.1 `RoundedRectButton`

Our Rounded Rect Button is similar to the one in *Apparatchik*.

The SLAP Framework did not come with a `RoundedRectButton`. Therefore we subclassed the existing SLAP-button and overwrote the draw function. The new button can be seen in figure 5.2. The design is taken from Deininghaus's [2010] latest version. The button was created as a first step of implementing the interface, since it is used for nearly every widget.

The implementation of our `RoundedRectButton` is finished, and took the estimated time in the project plan. Changeable characteristics are the colors of font, stroke, button not pressed background, and button pressed background. Every button gets an identification number (ID) which is used to trigger the correct function in its parent widget. This parent widget has to be a `SLAPCommandReceiver` and implement the `-(void) executeCommand:(int) commandName fromSource:(id)source;` method. Where the integer `commandName` represents the button ID set for the triggered button, handed over in `source`. You can see an overview of the class we created in table 5.2.

Figure 5.2: `RoundedRectButton` created for *Folio*

Class	`Rounded Rect Button`
inherits from	`SGOButton`
Properties:	(all inherited from SGOButton)
`ID`	integer
`buttonMode` (Push/Toggle)	SGOButtonMode
`buttonState` (Up / Down)	SGOButtonState
Delegate:	`SLAPCommandReceiver`
Delegate methods	`-(void) executeCommand:(int) commandName fromSource:(id)source;`

Table 5.2: Overview for the `RoundedRectButton`

5.2.2 `Document`, `DocumentController` and `DocumentView`

The document classes are the parent classes for every interface widget in *Folio*. Where the `Document` class represents the parent model class, `DocumentController`represents the controller parent, and `DocumentView` the parent view class.

These classes are superclasses for all our interface objects.

`Document`

In table 5.3 you can see the collection of attributes, the `Document` class is responsible for. The `affineTransform` property saves the actual position, rotation, and translation on the multitouch screen. By saving this in the model as well, we are able to restore

`Document` is the model class.

the table on relaunch as it was left behind when the last saving point was made. The controller property is `weak` to prevent a retain cycle between controller and model. The mutable array will hold the links to other materials on the table.

Class		`Document`
inherits from		`NSObject`
Properties:		
`myController`	weak	DocumentController
`affineTransform`		CGAffineTransform
`myReferences`	strong	NSMutuableArray

Table 5.3: Overview for the `Document` class

`DocumentView`

`DocumentView` holds the connection to the controller and affine transform on the screen.

Table 5.4 shows the class definition of the `DocumentView` class. The `delegate` property holds a weak reference to a controller who implements the `-(void) executeCommand:(int) commandName fromSource:(id)source;` method. Again the reference is weak to prevent a retain cycle between view and controller.

Class		`DocumentView`
inherits from		`SGORect`
Properties:		
`delegate`	weak	DocumentController

Table 5.4: Overview for the `DocumentView` class

`DocumentController`

The controller class manages the interaction between `Document` and `DocumentView`.

The class overview for the `DocumentController` class can be seen in table 5.5. `MyDocument` and `MyView` are the references to the model and the view class corresponding to the MVC Paradigm. The `(id) initWithDocument:(Document*)document;` method creates a new DocumentController and sets the myDocument property. The (void) affineTransformChangedTo:(CGAffineTransform)transform method gets called on the touchesMoved event, and sets the transform property of the model. This property is used to save the actual position on the screen in the LivingObjectsController.

Class inherits from		Document NSObject
Properties:		
myDocument	strong	Document
myView	strong	DocumentView
Class methods		
(id) initWithDocument:(Document*)document;		
(void) affineTransformChangedTo: (CGAffineTransform)transform		

Table 5.5: Overview for the DocumentController class

5.2.3 Browser, BrowserController and BrowserView

Connection to the internet, search functionality and bookmarking were high rated features over all our interviews. Therefore we enhanced *Folio* with a web browser. To save and load different browser windows we added our browser into the document management of *Folio*. Every browser consists of a Browser model class, a BrowserView and a BrowserController.

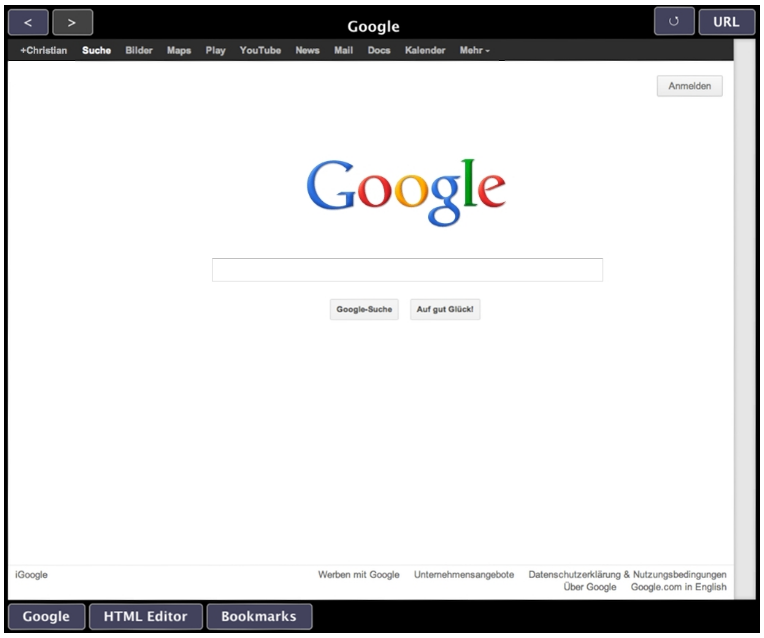
As Deinighaus and our interviewees suggested, we implemented a web browser.

Browser

In Table 5.6 you can see an overview over the Browser class. The only added property until now is the url-string. But this list will need to be extended when the actual browser functionalities increase. To add a tab bar the browser needs a list of urls. It would also be highly appreciable to add a scroll position variable, which saves the

The Browser class holds all the data of the current state the *Browser* widget is in.

scroll position of the displayed page, and can be saved and reloaded as well. Also there is no history management right now; the browser should be able to go back and forth in a history to be really useful.



<
>
Google
URL
+Christian Suche Bilder Maps Play YouTube News Mail Docs Kalender Mehr
Anmelden
Google
Google-Suche
Auf gut Glück!
iGoogle
Werben mit Google Unternehmensangebote Datenschutzerklärung & Nutzungsbedingungen
Über Google Google.com in English
Google
HTML Editor
Bookmarks

Figure 5.3: The *Apparatchik* version of our *Browser*.

Class		`Browser`
inherits from		`Document`
Properties:		
`url`	strong	NSString*

Table 5.6: Overview for the Browser class

`BrowserView`

The `BrowserView` displays web contents.

In contrast to the old version our browser is not attached to a keyboard on the table. Instead the user will be able to create a new browser window by hitting a button on the screen. Also users will be able to create more than one browser. In figure 5.3 you can see the *Apparatchik* version of the browser in comparison to figure 5.4 the *Folio Browser*. Table 5.7 contains the class description of our BrowserView.

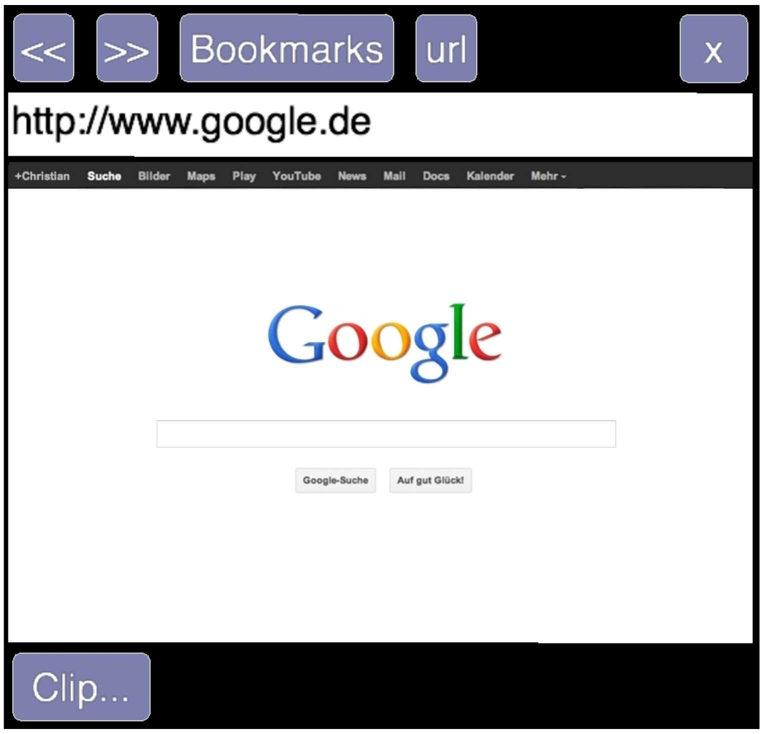


Figure 5.4: The *Folio* version of our *Browser*.

The `webRect` is the actual contents part of the `BrowserView` it is a web renderer provided by the SLAPFramework.

The `forward-` and `backButton` access the history of the browser. The `backButton` brings the latest page back, the `forwardButton` undoes the change a user made by pressing the `backButton`. All these should work to the very maximum of all user inputs. That means the user should be able to undo every page change he did from the launch of a browser window and should be able to also reburying every undid page.

`forward-` and `back-Buttons` are used to access the history.

The `bookmarksButton` brings up a new page, this page contains a list of bookmarked pages. This list will be editable when *Folio* is complete. Therefore the `Browser` model class should be extended further in the later development of *Folio*.

Class		BrowserView
inherits from		DocumentView
Properties:		
webRect	strong	SGOBrowser*
backButton	strong	RoundedRectButton*
forwardButton	strong	RoundedRectButton*
bookmarksButton	strong	RoundedRectButton*
closeButton	strong	RoundedRectButton*
clipButton	strong	RoundedRectButton*
urlButton	strong	RoundedRectButton*
urlBar	strong	SGOText*

Table 5.7: Overview for the `BrowserView` class

The `urlButton` needs to be pressed to change the url.

The `urlButton` gives access to the `urlBar`. If the user hits the button, the focus of the keyboard changes to the `urlBar` and the user is able to enter a new url. When the user hist the enter-button on the keyboard, the `urlBar` loses the focus, and the browser loads the new entered url. We took this behavior from the old version of the multi-touch table, to be consistent with the system our users already knew. Nevertheless this technique of entering should be rethought for the next implementation step. Perhaps a touch on the `urlBar` is more intuitive to get the focus and change the url.

To create a screenshot of the browser contents the `clipButton` is used.

The `clipButton` is used to create a new `Clip` out of the browser. The first press enables a changeable mask over the browser, where the user can select the area he wants to copy. This area can be changed by pressing on the browser with two fingers. One finger represents the top left corner of the clip, the other represents the bottom right. When the user hits the `clipButton` again, the masked part will be copied into a new `Clip` and displayed next to the browser. The default mask is set over the whole browser contents, that means the user is able to copy the whole page by just pressing the `clipButton` twice.

BrowserController

The BrowserController on table 5.8 is responsible for the interaction with the *Browser*. It has weak references to the LivingObjectsController, which saves and creates all objects on the table, and to the uiToolKit provided by the SLAP Framework. These references are weak because we do not want to create retain cycles between a BrowserController object and it's delegates.

Class inherits from		BrowserController DocumentController
Properties:		
expectsKeyboardEvents		BOOL
myLivingObjectController	weak	LivingObjectController*
uiToolKit	weak	SLAPUITK*
Class methods		
initialization		
(id) initWithBrowser:(Browser*)browser OnUITK:(SLAPUITK*)aToolKit;		
Event handling		
(void) executeCommand:()int) commandName fromSource:(id)source;		
(void) getKeyEvent:(NSEvent*)event;		
(void) gotTwoTouches:(NSMutableSet*)t;		
Browser methods		
(void) closeBrowser		
(void) loadNewUrl		
(NSRect) selectNewClip		
(void) createNewClipFromRect:(NSRect)rect;		

Table 5.8: Overview for the BrowserController class

The BrowserController is a SLAPCommandReceiver delegate, to the button presses on a button of the SLAP Framework. This is done in the executeCommand method. The getKeyEvent: method and expectsKeyboardEvents property are used to catch keyboard events when the url is changed.

The BrowserController class is a delegate for button events.

The selectNewClip and createClipFromRect methods are used to create a screenshot of the current screen. They get triggered when the user hits the clipButton on the bottom of the `BrowserView`. With a first press, the user is able to chose the part of the contents which should get copied, with a second press on the `clipButton` the actual `Clip` is created.

5.2.4 `Book`, `BookController` and `BookView`

These classes represent the *Book* widget.

In figure 5.7 you can see the actual version of the *Book* widget. We tried to keep the look and feel of Deininghaus's version of the software, since he put a lot of effort and research into it we wanted to use his findings. Deininghaus's book can be seen in figure 5.6

Marker Detection

The marker detection is handled by the *SLAP* framework.

We added the functionality to attach the digital representation of a book to a real world book. The SLAP framework comes with a possibility to detect certain combination of touch markers. In figure 5.5 the Markers attached to a book can be seen. These markers are recognized by the system like normal finger touches. And the exact arrangement is used to decide which book is actually placed on the surface. Using information the digital representation of a book is attached to it. Moves when the book is moved and remains on its position if the book is lifted of the table.

In SLAP this is done by assigning a unique `SLAPFootprint` object to the widget. This footprint consists of several touch objects and can be detected by the system. This is done in the `SLAPUITK` every time all touch events get handled.

Old and New Version

Editing the attached material can be done in *Folio* now.

As seen in figure 5.6 and 5.7 the old and the new version of a book look similar. The only changes we made are in the feel of this widget. With *Apparatchik* users had to add materials, collected in a tab next to a page, by hand. This was done via

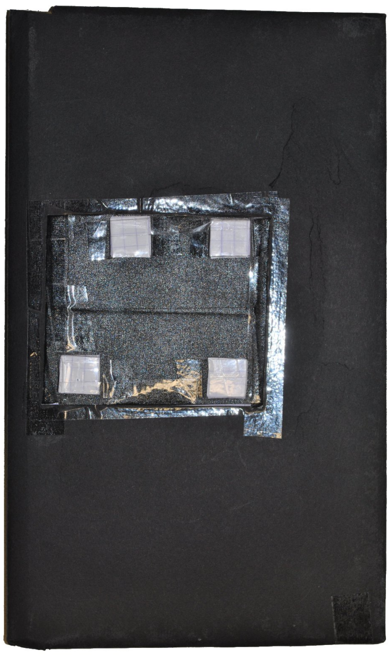
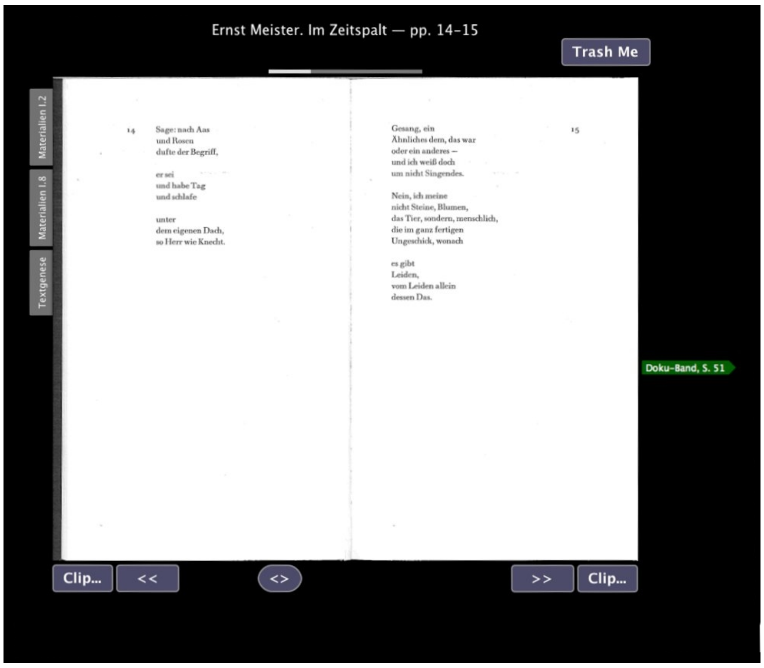


Figure 5.5: An arrangement of markers to distinguish which book is placed on the table.

an interface which is displayed on the screen of the desktop system next to the multitouch table. In our version users will be able to add, remove and arrange the contents of the tabs while the system is running. This will be possible with standard multitouch gestures. Dragging a manuscript into a folder will attach this manuscript to this position in the folder. All these changes are saved automatically. Doing this we hope to ease the work with *Folio*, this feature was requested in our initial interviews.

Book

Table 5.9—"Overview for the `Book` class" shows the class diagram for the *Book* Model. It holds the information which book it represents, how big it is and how to handle the page count. `frontMatter` is the actual number of pages which are not counted as a page in front of the text, `pageCount` represents the overall pages of a book, including the for the



14

Sage: nach Aas
und Rosen
dufte der Begriff,

er sei
und habe Tag
und schlafe

unter
dem eigenen Dach,
so Herr wie Knecht.

Gesang, ein
Ähnliches dem, das war
oder ein anderes –
und ich weiß doch
um nicht Singendes.

Nein, ich meine
nicht Steine, Blumen,
das Tier, sondern, menschlich,
die im ganz fertigen
Ungeschick, wonach

es gibt
Leiden,
vom Leiden allein
dessen Das.

15

Figure 5.6: The *Book* Widget in *Apparatchik*

page numbering of the printed version not counted pages. The `Book` needs the two properties `widthWhenClosed` and `widthWhenOpen` to be able to get displayed also when the actual book is closed, but still lying on the table.

For future work this will be extended with properties for the contents of the book. The parts used in the class `BookView` are able to display pdfs. The displayed books will be scanned, saved as pdf file and displayed on the multitouch table.

`BookView`

We omitted the slider widget, since we want to implement automatic page recognition soon.

The `BookView` holds the properties to display a digital representation of a book. The `bookRect` is the actual container for the scanned pdf. The `left/rightPane` rects will be used to realize the Tabs.

Turning the pages is done as in *Apparatchik* with the `pageTurnButtons`. We decided to omit the slider widget

Class		`Book`
inherits from		`Document`
Properties:		
`title`	strong	NSString*
`widthWhenClosed`	strong	NSNumber*
`widthWhenOpen`	strong	NSNumber*
`height`	strong	NSNumber*
`pageCount`	strong	NSNumber*
`frontMatter`	strong	NSNumber*

Table 5.9: Overview for the `Book` class

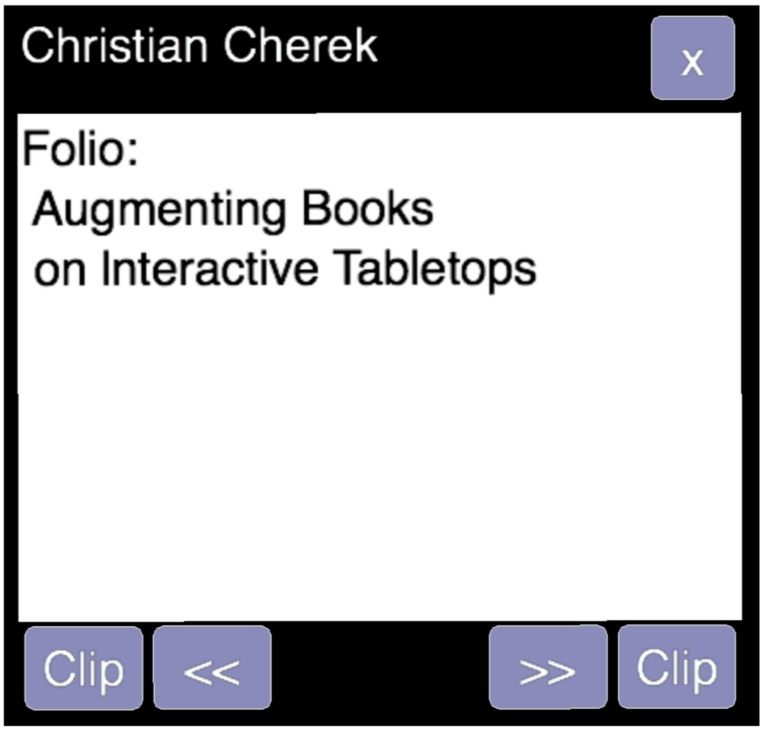


Figure 5.7: The *Book* Widget in *Folio*

used in the earlier version, since the automatic page recognition is one of the first implemented features after this thesis is finished.

The left and right `ClipButtons` are used similar to the one in the *Browser*. A touch on this button will open a new window, where the user selects the clipped area, and then cre-

Class		BookView
inherits from		DocumentView
Properties:		
bookRect	strong	SGOImage*
titleRect	strong	SGOText*
leftPane	strong	SGORect*
rightPane	strong	SGORect*
closeButton	strong	RoundedRectButton*
leftClipButton	strong	RoundedRectButton*
rightClipButton	strong	RoundedRectButton*
leftPageTurnButton	strong	RoundedRectButton*
rightPageTurnButton	strong	RoundedRectButton*
Class methods		
(id)initWithFrame:(NSRect) frame onGUIObject:(SLAPGUIObject*)parentObject)		

Table 5.10: Overview for the `BookView` class

ates a new *Clip*.

The `initWithFrame: onGUIObject:` method overrides the inherited method to set up the widget when it is created.

In table 5.10—"Overview for the `BookView` class" you can see collection of the class properties and methods.

BookController

The `BookController` class as listed in table 5.11—"Overview for the `BookController` class" is responsible for the interaction with the *Book* widget. There are weak references to the delegate `LivingObjectsController` and since the `BookController` is again a `SLAPCommandReceiver` also a weak reference to the system wide `uiToolKit`.

The `closeBook` method is called when the `closeButton` of the corresponding view is pressed, and simply tells the `LivingObjectController` to delete this instance of a *Book*.

Class		`BookController`
inherits from		`DocumentController`
Properties:		
`myLivingObjectsController`	weak	LivingObjectsController*
`uiToolKit`	weak	SLAPUITK*
Class methods		
`(id)initWithBook:(Book*) book onUITK:(SLAPUITK*) uiToolKit`		
`(void) closeBook`		

Table 5.11: Overview for the `BookController` class

Tabs

The tabs will be used to store contents next to a page of a book. By dragging contents into them the user will be able to store contents in this tab. The contents that can be stored here are the clips taken out of other books, the internet browser or in a later version annotations and camera scans. The contents in a tab is saved automatically so the user does not need to care about loosing his work anymore. In figure 5.8 this functionality can be seen. The tabs can be opened and closed by pressing them, or by dragging them in and out of the book.

Tabs are used to attach digital copies to the *Book*.

If a tab holds more items than can be displayed at once, the user is able to move the contents of a tab up and down. This is done by dragging the contents to the top or the bottom of the tab.

An item within such a tab can be enlarged and moved independently by pressing on it. This creates a new *Clip* with the contents of the item. This *Clip* gets added to the `LivingObjectsController` and gets stored and brought back as every other widget.

Until now the *Tab* functionality is not implemented in *Folio*.

This functionality is not yet implemented in *Folio* but finishing the *Book* widget and adding these functionality should be done right after this thesis.

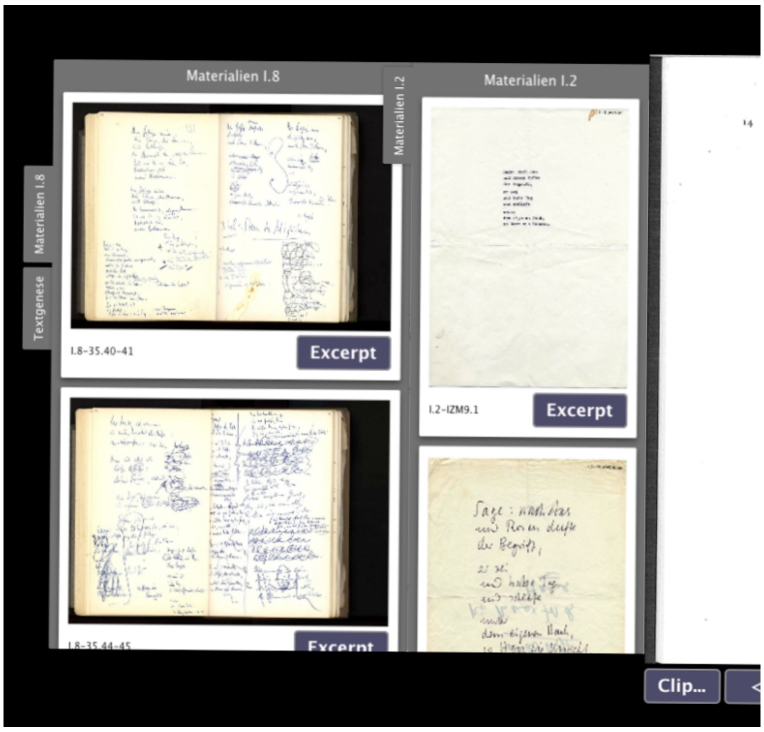


Figure 5.8: Part of a *Apparatchik* book with opened *Tabs* on the left side.

5.2.5 Clip, ClipController and ClipView

This widget is used to display single page contents.

The *Clip* widget is used to display single paged contents. That could be a copy of a manuscript, or a screenshot of a *Browser*. This widget will be used to spread out the materials on the table. If a user tries to analyze a certain text, and wants to include the different versions of this text. It is useful to spread out these versions, and move them around to get a overview and order them correctly.

Deininghaus [2010] found that this is frequently done in a normal working situation. Therefore we want our users to be able to do this with *Folio* as well. We even extend this possibility with a zooming function.

Clips are freely movable and resizable on the screen.

Our *Clips* can be moved on the table with standard multi-touch gestures. Pinching on a *Clip* will enlarge it, so that our users can investigate it in a more detailed version. Rotating the fingers will rotate the *Clip*, so it is possible to work with more than one person or arrange the items in a circle around the user.

`Clip`

Class		`Clip`
inherits from		`Document`
Properties:		
`title`	strong	NSString*

Table 5.12: Overview for the `Clip` class

In table 5.12—"Overview for the `Clip` class" you see the class overview for the model part of the *Clip* widget. The `title` property is used to store the the contents of the *Clip*. These list will get extended when the actual contents needs to be saved as well.

`ClipView`

Class		`ClipView`
inherits from		`DocumentView`
Properties:		
`closeButton`	strong	RoundedRectButton*
`contentImage`	strong	SGOImage*
`titleBar`	strong	SGOText*

Table 5.13: Overview for the `ClipView` class

The `ClipView` 5.13—"Overview for the `ClipView` class" displayed the contents of the `Clip` model. The `contentImage` can hold pdf or image files and similar to the *Book* and *Browser* a button is used to remove the widget from the table.

Class		`ClipController`
inherits from		`DocumentController`
Properties:		
`myLivingObjectsController`	weak	LivingObjectsController*
`uiToolKit`	weak	SLAPUITK*
Class methods		
`(id)initWithClip:(Clip*) clip onUITK:(SLAPUITK*) uiToolKit`		
`(void) closeClip`		

Table 5.14: Overview for the `ClipController` class

`ClipController`

In table 5.14—"Overview for the `ClipController` class" a summary of the `ClipController` class can be seen. As in the `Browser-` and `BookController` the `ClipController` has a weak reference to the `LivingobjectsController` and is a delegate for events from the `uiToolKit`.

5.3 Workspace Organisation

The introduced workspace structure should be maintained as good as possible.

The workspace of *Apparatchik* got more and more confused. Thus it was difficult for new programmers to understand the project correctly.

To prevent this in *Folio* the workspace is clearly structured. For each part of the model view controller paradigm there is a folder, and classes that do not fit in there should get grouped as well.

Also naming conventions make the belongings to MVC more visible.

For classes of the widgets there are naming conventions that let each model class be named as the widget itself (e.g. `Book` for the *Book* widget.) Each view class should be named as the widget with a following "`View`". And the controller with following "`Controller`".

Additional resources, like the `plist` file used to store the table state are placed in the `Resources` folder.

In the uppermost folder there are only the `FolioAppDelegate` and the `FolioViewDelegate`. Which are only used to delegate actions that concern the whole application. In *Apparatchik* this was weaken so that the `AppDelegate` had methods for starting the App, managing touch input, managing keyboard input, rendering the screen, managing the data function, and a lot more. In *Folio* this should be stronger segregated from each other.

The `FolioAppDelegate` should not be misused as servant for every interaction.

Chapter 6

Progress Evaluation

We implemented the first milestones of *Folio*. In figure A.2 the already reached targets are marked in dark blue. The first version can be used at the multitouch table. In this chapter we will describe the milestones and how far each of them is reached.

The first milestones are already under testing.

6.1 Data Structure Milestone

This milestone is not yet completed. We implemented the current State savings. Each widget even the not yet implemented *Annotation* can be saved and brought back as is was before.

Saving the system state is working, the remaining steps in progress.

This includes the position on the table, the rotation and zoom scale. For the *Book* this also includes the current selected page and the whole contents; for the *Browser* we save the url and in the future the whole browsing history as well. Clips are just brought back as they were before.

To implement the data manager for contents in *Books* and *Clips* took more time than we suggested. Our first try to implement this on a `plist` structure as well went not good, so we decided to change this and implement a Core Data managed system instead. This is not finished right know, but should be one of the first steps for our succeeding colleagues.

Also saving the edits users can make to books is not yet implemented. We skipped the realization of this feature until the contents management works correct.

6.2 Browser Milestone

After merging the newest version of the *SLAP* framework the *Browser* should be complete.

The *Browser* widget is quite complete. We are able to create and move several browsers, and each of them exists independently from the others.

Also it is possible to catch keyboard inputs and change the url with this. We did not implemented any short cuts or expert functionality yet, but if these are needed should be discussed with our users again.

The only thing that still needs to be done is some testing on this widget, but we are confident that the *Browser* widget does not need a lot of effort any more.

6.3 Clip Milestone

Clips are running and able to display contents.

Similar to the *Browser*, the *Clip* widget is nearly finished. The whole interaction works fine and the widget can display every contents we want to.

The *Clip* will be complete as soon as the contents managing system is finished. In the latest version of *Folio* our *Clip* widget does not display anything, since we want to add the correct contents with the next step but our tests showed that it is possible to display our material on a *Clip*.

6.4 Book Milestone

The Book Milestone is the milestone that is under progress right now. We already finished the widget layout. The *Book* can be placed, deleted and moved as desired. Also the contents display is complete.

The *Book* widget marks the current implementation step. Right now this widget gets developed further.

What needs to be done next is page turning interaction and the connection to the real world books. We recommend to finish these steps right after the contents manager. Than the *Book* widget will be fully usable. Also the interaction with other widgets is not yet running, therefore the realization of the third part of the data structure milestone is required.

6.5 Further Milestones

After finishing the milestones we recommend to follow the other created milestones. These consist of additional needed, but less fundamental features.

Annotation, user management and automatic page recognition will be done in the future.

For example a automatic page recognition, or the *Annotation* widget could be the next steps there. The project plan A.2 In chapter 4—"Planning *Folio*" gives a overview how the next steps should look like.

And after that the interview lists in chapter 4—"Planning *Folio*" give additional feedback what should be the next steps.

Chapter 7

Summary and future work

In this thesis we motivated and planned the reimplementation of a software for literary criticism. We implemented the basic functionality and gave a project plan, succeeding colleagues can refer to.

7.1 Summary and contributions

The evaluation of the running system, showed that improving the old system is not a reasonable step. And our experiences during the development process of *Folio* strengthened this assertion. The new software, *Folio*, although not as sophisticated as *Apparatchik* now, is already a lot more responsive. Even with multiple objects on the screen we do not get similar performance issues.

Also we created a clean and structured workspace which can be used to build *Folio's* next implementation steps. Chapter 5—"*Folio*" can be used to understand the code, and the clear usage of design patterns as the Model View Controller pattern facilitate the adjustment of new colleagues.

7.2 Future work

For future work we suggest to abide by our project plan. Following this it is possible to create a working environment, literary scholars could actually use for their research.

The multitouch table right now is attached to a computer and needs some effort to get running. This makes it difficult for our users to simply turn it on and start working with the table. It would be nicer to have a single button that can be pressed, and all the initialization process is done for the user. This contains initialization of the cameras detecting the touches, and running the correct application. After this initialization the user should not be forced to use the computer next to the table at all.

Until now our system is able to communicate over the internet. There is no application to existing databases since the internet browser. Since these databases are heavily used by our users it could be desirable to create applications within *Folio* that provide easy access of these databases. Textgrid[1] is an example for such a online database, where professionals in the humanities can access material and tools for their research.

Since some of our interviewees mentioned they would like to be able to create the whole edition at our table. It is worth thinking of a possibility to add a vertical screen to the multitouch table, where a word processor is running on. The user should be able to use the horizontal surface as suggested in this thesis and the vertical as a usual computer screen in parallel. In [2009] Weiss et al. presented a multitouch table that combines horizontal and vertical screens with each other. This could be a possible way of integrating research for a edition and the actual writing of the book.

From the human computer interaction perspective this project offers interesting research fields as well. There is ongoing research at the Media Computing Group which tries to find out, how workspaces like our multitouch tables could be enhanced with better search or ordering func-

[1]http://www.textgrid.de/

tionality. Also the research question if the habits of literary scholars change using this system is unanswered yet. We hope that *Folio* once it is finished can offer the possibility to do user studies on this kind of research projects.

Appendix A

ADDITIONAL DOCUMENTS

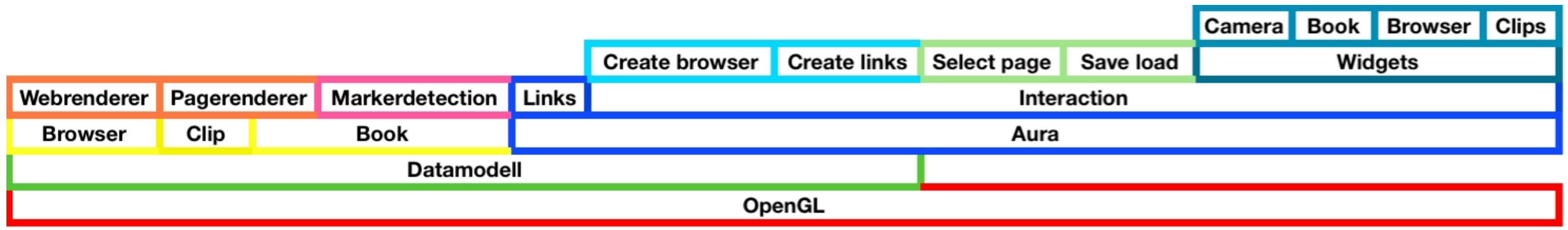
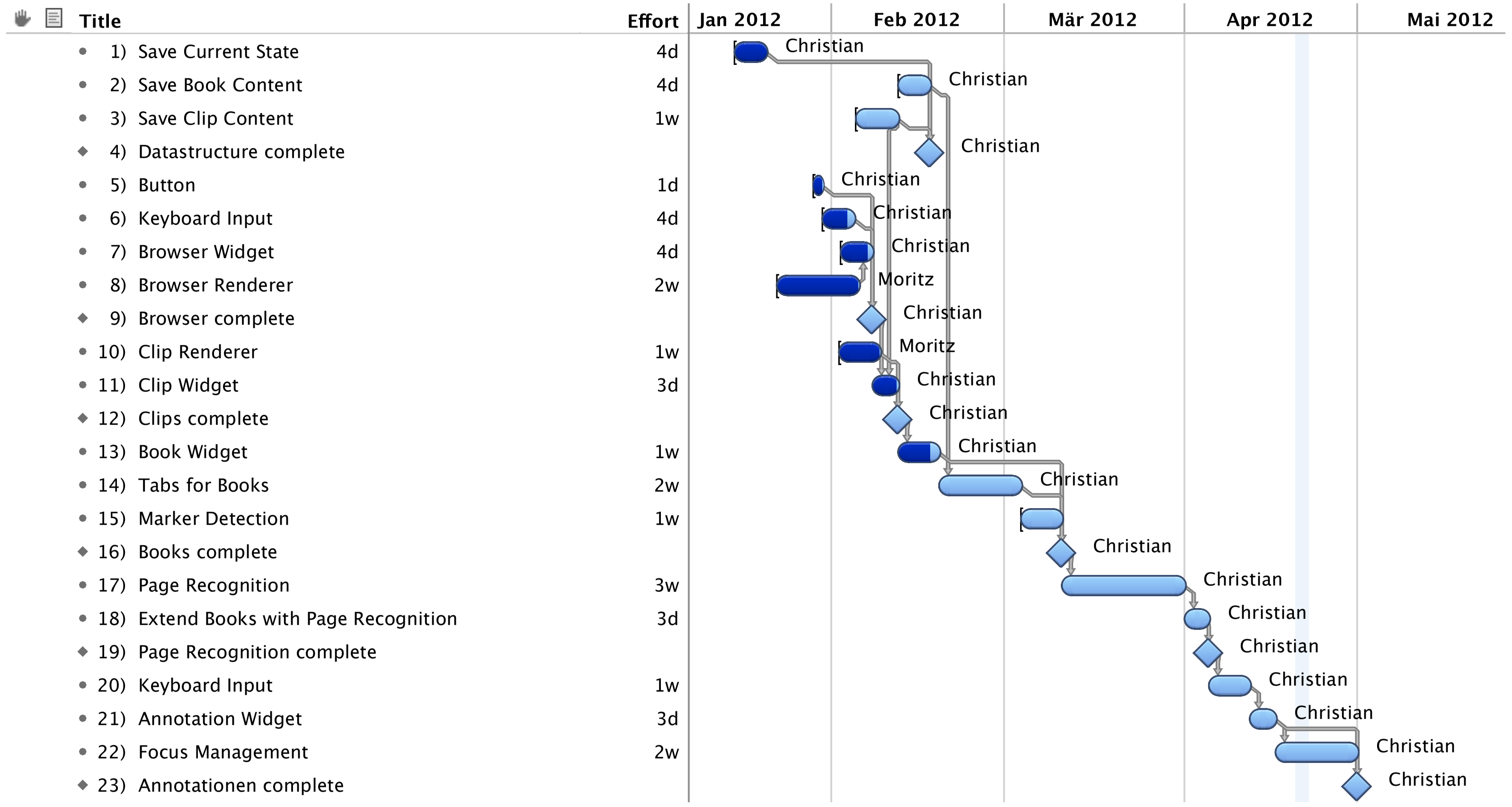


Figure A.1: A diagram displaying the structure of *Folio*, created in one of our team sessions.



Title	Effort
• 1) Save Current State	4d
• 2) Save Book Content	4d
• 3) Save Clip Content	1w
◆ 4) Datastructure complete	
• 5) Button	1d
• 6) Keyboard Input	4d
• 7) Browser Widget	4d
• 8) Browser Renderer	2w
◆ 9) Browser complete	
• 10) Clip Renderer	1w
• 11) Clip Widget	3d
◆ 12) Clips complete	
• 13) Book Widget	1w
• 14) Tabs for Books	2w
• 15) Marker Detection	1w
◆ 16) Books complete	
• 17) Page Recognition	3w
• 18) Extend Books with Page Recognition	3d
◆ 19) Page Recognition complete	
• 20) Keyboard Input	1w
• 21) Annotation Widget	3d
• 22) Focus Management	2w
◆ 23) Annotationen complete	

Figure A.2: A Time Schedule for the Implementation of *Folio*. Not containing additional features which are not needed for basic research with *Folio*.

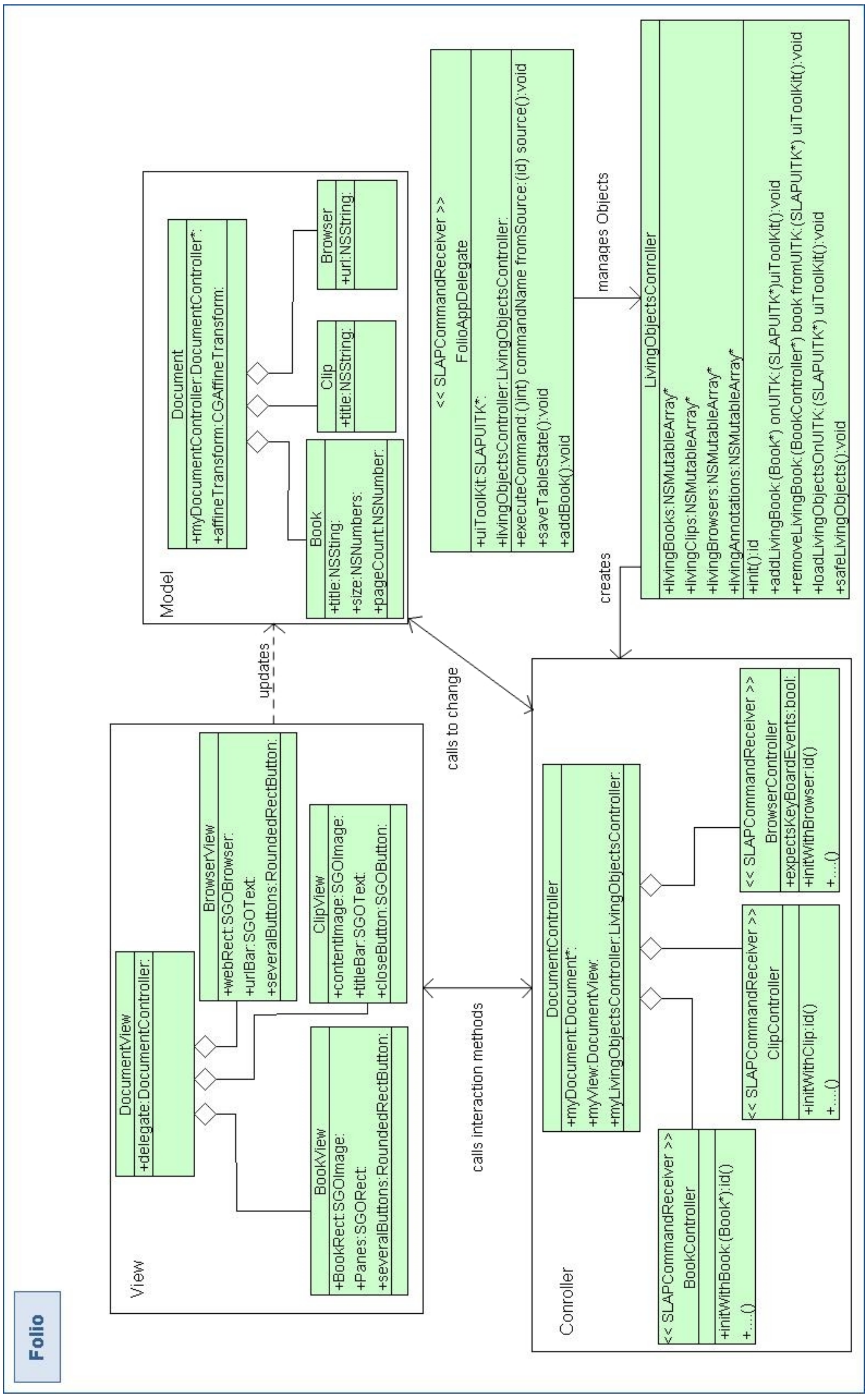

Folio
Model
Document
+myDocumentController:DocumentController*:
+affineTransform:CGAffineTransform:
Book
+title:NSSting:
+size:NSNumbers:
+pageCount:NSNumber:
Clip
+title:NSString:
Browser
+url:NSString:
<< SLAPCommandReceiver >>
FolioAppDelegate
+uiToolkit:SLAPUITK*:
+livingObjectsController:LivingObjectsController:
+executeCommand:()int) commandName fromSource:(id) source():void
+saveTableState():void
+addBook():void
manages Objects
LivingObjectsConroller
+livingBooks:NSMutableArray*
+livingClips:NSMutableArray*
+livingBrowsers:NSMutableArray*
+livingAnnotations:NSMutableArray*
+init():id
+addLivingBook:(Book*) onUITK:(SLAPUITK*)uiToolKit():void
+removeLivingBook:(BookController*) book fromUITK:(SLAPUITK*) uiToolKit():void
+loadLivingObjectsOnUITK:(SLAPUITK*) uiToolKit():void
+safeLivingObjects():void
creates
updates
calls to change
View
DocumentView
+delegate:DocumentController:
BrowserView
+webRect:SGOBrowser:
+urlBar:SGOText:
+severalButtons:RoundedRectButton:
ClipView
+contentImage:SGOImage:
+titleBar:SGOText:
+closeButton:SGOButton:
BookView
+BookRect:SGOImage:
+Panes:SGORect:
+severalButtons:RoundedRectButton:
calls interaction methods
Conroller
DocumentController
+myDocument:Document*:
+myView:DocumentView:
+myLivingObjectsController:LivingObjectsController:
<< SLAPCommandReceiver >>
BrowserController
+expectsKeyBoardEvents:bool:
+initWithBrowser:id()
+...()
<< SLAPCommandReceiver >>
ClipController
+initWithClip:id()
+...()
<< SLAPCommandReceiver >>
BookController
+initWithBook:(Book*):id()
+...()

Figure A.3: A class diagram of *Folio*. Classes belonging to the MVC paradigm are connected in boxes.

Bibliography

Stephan Deininghaus. An interactive surface for literary criticism. Master's thesis, RWTH Aachen University, April 2010.

Donald A. Yacktman Erik M. Buck. *Cocoa Design Patterns fuer Mac und iPhone*. mitp, 2012.

W. Hurst and P. Jarvers. Interactive, dynamic video browsing with the zoomslider interface. In *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, 2005.

Ernst Meister. *Gedichte: Textkritische und kommentierte Ausgabe*. Axel gellhaus, Stephanie Jordans, Andreas Lohr, Göttingen: Wallstein, 2011.

Bodo Plachta. *Editionswissenschaft: Eine Einführung in Methode und Praxis der Edition neuerer Texte.* Reclam, 1997.

Malte Weiss, Roger Jennings, Julie Wagner, James D. Hollan, Ramsin Khoshabeh, and Jan Borchers. Slap: Silicone illuminated active peripherals. In *Extended Abstracts of Tabletop '08*, pages 37–38, 2008.

Malte Weiss, Simon Voelker, and Jan Borchers. Benddesk: Seamless integration of horizontal and vertical multitouch surfaces in desk environments. In *Extended Abstracts of Tabletop '09*. ACM, 2009.

Typeset April 24, 2012