Amra Džombić, BSc

Pocket Toon
A Smart Flip book

MASTER'S THESIS

to achieve the university degree of
Master of Science
Master's degree programme: Computer Science

submitted to
Graz University of Technology

Supervisor
Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Institute for Softwaretechnology

Graz, April 2016

AFFIDAVIT

Acknowledgments

I would first like to thank my mother, Safija Ćatović-Džombić. Without her selflessness, her support, her concern, and her love, none of this would be possible. This one is for you mom!

I would like to thank my supervisor, Prof. Wolfgang Slany, for the guidance, encouragement and advice he has provided throughout my time as his student. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly.

Kurzfassung

Ein Daumenkino ist eine Art Buch, das eine Serie von zusammenhängenden Bildern enthält. Wenn dieses schnell durchgeblättert wird, kreiert es so die Illusion einer Bewegung. Das Daumenkino stellt eine der frühesten Formen von Animation dar.

Diese Arbeit verwendet als Grundlage das genius design, das ausschließlich auf der Erfahrung und der Kreativität eines einzelnen Designers basiert, um das Konzept des traditionellen auf Papier basierenden Daumenkinos in eine App für Smartphones – Pocket Toon – zu verwandeln. Diese App erlaubt es den Benutzern Bild-für-Bild-Animationen samt musikalischer Untermalung zu erstellen. Die Herausforderung hierbei ist, nicht nur eine App zu kreieren, welche Benutzern dies ermöglicht, sondern diese App mit einem überzeugenden User-Interface auszustatten, das von Effizienz und einfacher Handhabe geprägt ist.

Um eben genannte Probleme zu lösen, werden analoge Daumenkinos sowie Nintendos Flipnote Studio^{TM} analysiert und deren Eigenschaften und Funktionalitäten als Basis für jene von Pocket Toon herangezogen. Diese Eigenschaften werden mittels Personas, User stories und Scenarios verfeinert. Das konzeptuelle Design und die Gruppierung der Anforderungen basieren auf Information, die aus der Analyse bestehender Daumenkino-Apps gewonnen wurde. Den agilen Prinzipien folgend, welche die Priorisierung einzelner Merkmale eines idealen Produktes als essentiell ansiehen, wurden erste, minimale, jedoch mit den wichtigsten Funktionen ausgestattete Prototypen erstellt und diese als Basis für die Software-Umsetzung der ersten Version herangezogen.

Abstract

A flip book is a book that contains a series of related images, which when flipped rapidly create the illusion of motion. It is one of the earliest forms of animation.

This thesis applies principles of genius design, which solely relies on the experience and the creativity of an individual designer, to translate the paper based flip book concept into a smart mobile application — Pocket Toon, which will allow users to create image-by-image animations in combination with sound. The challenge ahead is not merely to provide an app that allows users to create a series of images, but to design a compelling user interface, which is characterized by both efficiency and ease of use.

To solve the interaction problem at hand, analog flip books and Nintendo's Flipnote StudioNintendo's Flipnote Studio $^{\text{TM}}$ are analyzed and their existing features are used as a basis for the features of Pocket Toon. These features are refined by personas, user stories and scenarios. The conceptual design and the grouping of requirements are based on the information acquired in the analysis of existing flip book applications. Following the agile principles of relentlessly prioritizing the features that a theoretical "finished" product might eventually get, minimal prototypes were designed and used as basis for the software implementation of the first version.

Contents

List of Tables

List of Figures

Listings

1 Introduction

1.1 Motivation

Flip books or flip notes are one of the earliest forms of animation. A flip book is a book that contains a series of related images, which when flipped rapidly create the illusion of motion [8]. This illusion is based on two optical phenomena known as persistence of vision and the phi phenomenon [3].

The visual persistence causes the brain to retain the image cast upon the retina for about a fraction of the second, even after the image has disappeared from view. This phenomenon occurs because the chemical representation of an image takes time to dissolve. As a result, there is a threshold below which the eye cannot detect changes in repeated visual stimuli. Thus a series of images arriving in rapid enough succession appear as one continues image. Alternatively, if the series of images arrives too slowly, the continuous image will have a jerking quality, i.e., flicker [19].

While the visual persistence creates the illusion of a single, continuous image, the phi phenomenon is responsible for conceptually filling the gaps between the alternating images that progressively differ, thus creating the illusion of motion. However, if the frequency of alternation exceeds a certain threshold, the visual persistence will take over and images will be shown at once [19].

In order to create the illusion of a motion, the rate of the still images, i.e., the number of images per second, has to be sufficiently timed so that visual persistence and the phi phenomena can take effect. In the film industry, for instance, movies are filmed and played at a rate of 24 frames per second, which is enough to create the illusion of motion.

The goal of this thesis is to translate the old fashioned flip book into a smart mobile application, which allows users to create image-by-image animations in combination with sound. The challenge ahead is not merely to provide an app that allows users to create a series of images, exploiting above mentioned optical phenomena, but to design a compelling user interface, which is characterized by both efficiency and ease of use.

1.2 Problem Statement

1.2.1 The user interface design problem

Designing a good user interface includes many distinct challenges. These days a good user interface design is less about pretty pictures and more about the people: What do they like? What kind of applications do they use? Why do they use them? And most importantly: how do they use them [24]?

In the same way that a written exam can be seen as a dialog between the students and their teacher, the application is a dialog between users and the machine. And just as it is up to the teacher to script this dialog between him and his students, by preparing exam questions and writing them on the paper sheet, it is up to the interface designer to script the conversation between the users and the artificial entity, by designing the user interface. Consequently, the interface designer's main task is to understand the user in every respect: What motivates the users? What are their expectations? What kind of keywords, icons and layout do they prefer? And after extracting all these pieces of information, how does one go about writing the perfect dialog or the perfect application [24]?

1.2.2 The user interface design problem

User centered design provides not only the means for a great user interface, but also the means for quality software.

The software quality can be measured by different factors, such as:

- Correctness - the ability to perform according to specification
- Extendibility - the ability to adapt to changes of specification
- Reusability - the ability to reuse existing software components
- Efficiency - the ability to place as few demands as possible on hardware resources
- Maintainability - the ability to be modified and corrected after it has been delivered
- Readability and clarity - the ability to be read and understood easily
- Documentation and test coverage – the presence of good documentation and tests
- Scalability - the ability to perform well when the workload increases
- etc.

Focusing on the wants and needs of the user, designers are able to observe the software from the user's point of view. This helps to identify desired software features, i.e., distinct elements of functionality, and determine the implementation order according to their importance, i.e., prioritizing certain features over others. This does not only help software developers to focus on features that matter, but also to focus on one feature at a time. A single feature is usually easier to design, improve, and test, raising the software quality at a single point. Thus, qualitative implementation of features equals qualitative software.

1.2.3 Thesis Outline

This thesis shows the translation of the old fashioned flip book into an intuitive mobile application – Pocket Toon. Pocket Toon allows users to create image-by-image animations in combination with sound. Chapter 2 discusses the animation software which inspired Pocket Toon. Chapter 3 examines interaction design. The conceptual design and the development of the first prototype are discussed in Chapter 4. Chapter 5 describes the software implementation. Finally, Chapter 6 provides a summary and discusses possible future developments.

2 Related Work

This thesis has been inspired by several existing animation software, including Nintendo's Flipnote Studio $^{\mathrm{TM}}$, Toon Boom's Storyboard Pro ©and Autodesk Sketchbook v7 ©: Flipbook feature.

2.1 Nintendo's Flipnote Studio $^{\mathrm{TM}}$

The Flipnote Studio is an application for the Nintendo DSi, which allows users to easily create a flip book -like animation with sound. It provides tools for creating flip book pages, either by drawing images with the Nintendo DSi stylus or by importing existing pictures taken with the Nintendo DSi Camera application, and adding a soundtrack by using the Nintendo DSi Sound application or the built-in microphone.

Until May 31, 2015 [18] Nintendo, together with Hatena Co., Ltd., provided a bulletin-board website, called Ugomemo Hatena (Flipnote Hatena), which allowed users to share their creations, as well as collaborate on projects.[15]

The Flipnote Studio tools can be grouped in 5 categories 2.7:

1. **Playback tool.** This tool is visible throughout the application. Users can start the animation, adjust speed, add new pages, jump to first/last frame and set a repeat flag.
2. **Single page tools.** This tool allows users to draw, modify foreground, background and draft layer, copy drawings across other pages, modify paper color, show/hide previous pages and select, as well as modify different objects on a certain page.

3. **Animation overview tool.** The animation overview tool enables users to rearrange pages, add, delete insert and copy/paste pages.
4. **Camera tool.** Users can import images from the system memory and modify them by changing brightness and contrast.
5. **Sound tool.** The sound tool allows users to add music or sound effects to the animation.

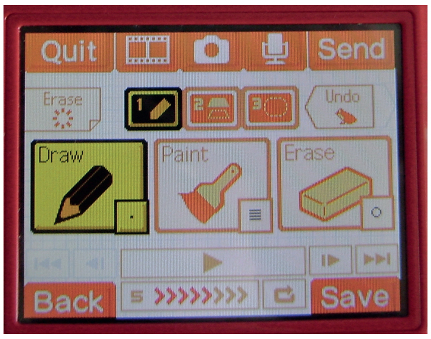


Figure 2.1: Single page tools - drawing tools.

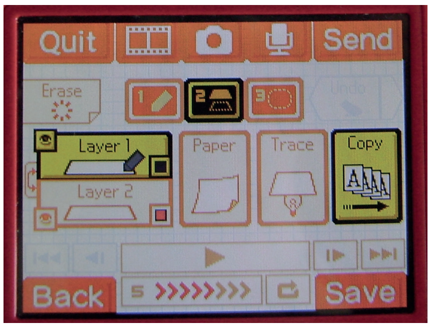


Figure 2.2: Single page tools - page tools.

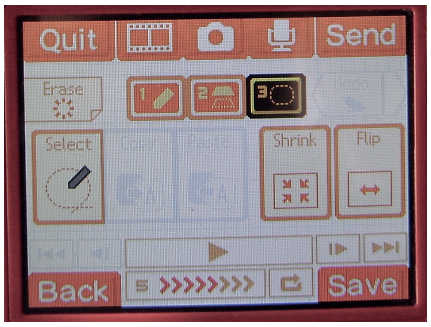


Figure 2.3: Single page tools - selection tools.

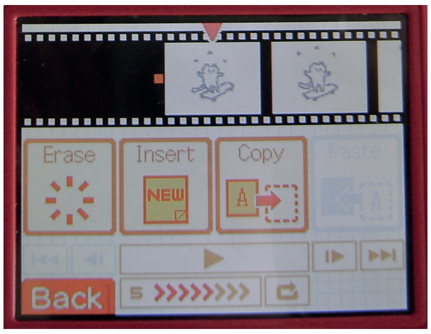


Figure 2.4: Animation overview tool.

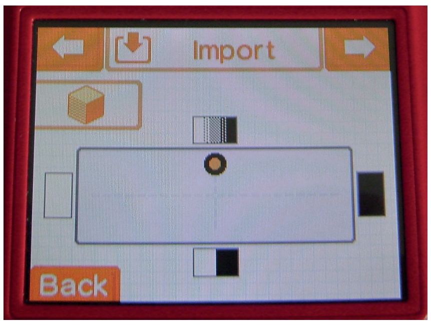


Figure 2.5: Camera tool.

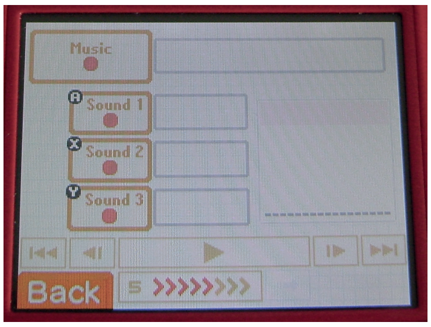


Figure 2.6: Sound tool.

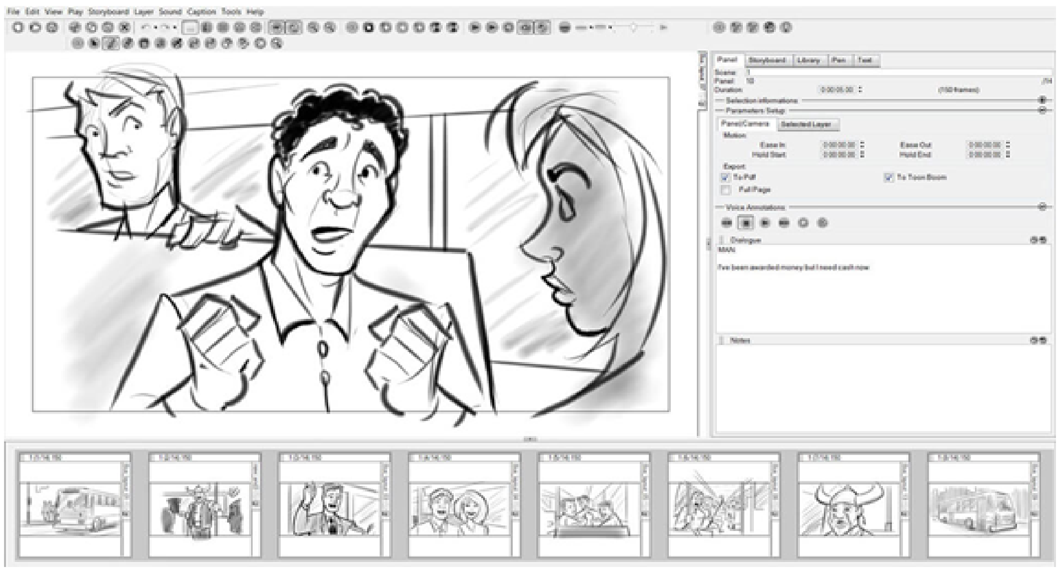
Figure 2.7: Nintendo's Flipnote Studio ^{TM}Tools

2.2 Toon Boom's Storyboard Pro ©

The Toon Boom Storyboard Pro ©is a software for creating digital storyboards and animatics for 2D/3D animations, live-action productions, games or events with advanced features. It incorporates many features from drawing and importing images to camera movements and sound editing.

The storyboard is a comic-like illustration, where each panel represents a particular scene in the animated cartoon or movie. Each scene consists of an action, a dialog and a camera angle.

The Storyboard Pro ©provides a simple and intuitive interface for creating and visualizing storyboards. Beside importing the script and adding captions, users can easily import, draw, rearrange and modify panels. Each panel represents a single action i.e. a single image.



MAN

I've been awarded money but I need cash now

Figure 2.8: Storyboard Pro user interface. The interface consist of different toolbars, stage view (containing stage specific tools, a drawing area and the layer tabs), panel view (displaying different captions and information related to a current panel), and thumbnail view (displaying all the panels in the project) [2]. (Source: [20])

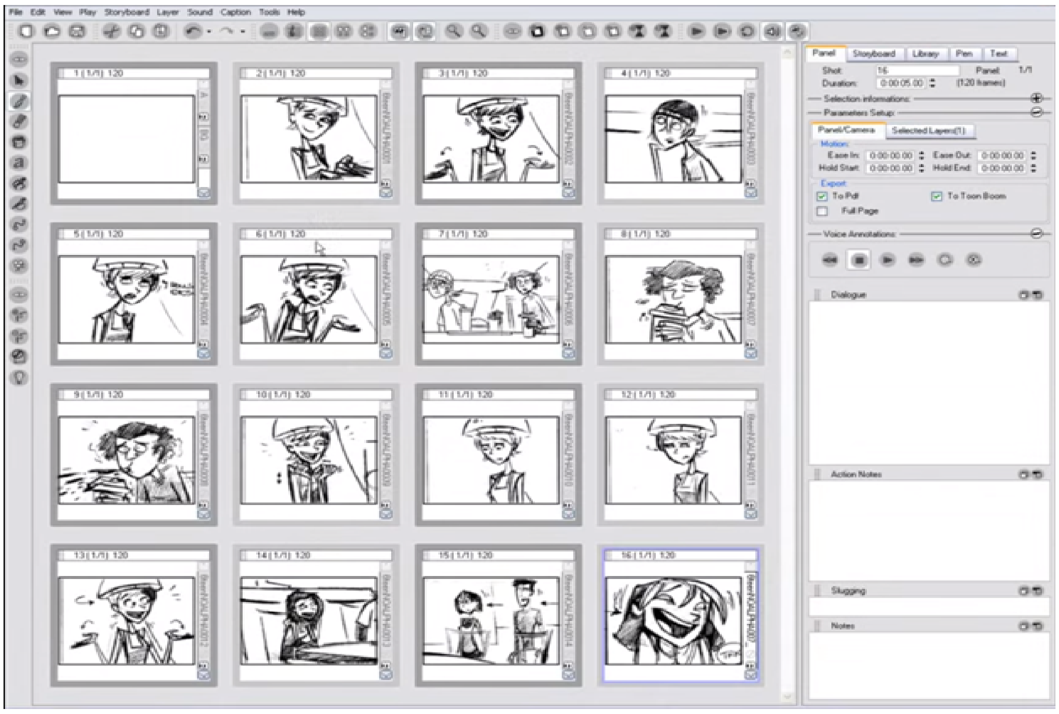
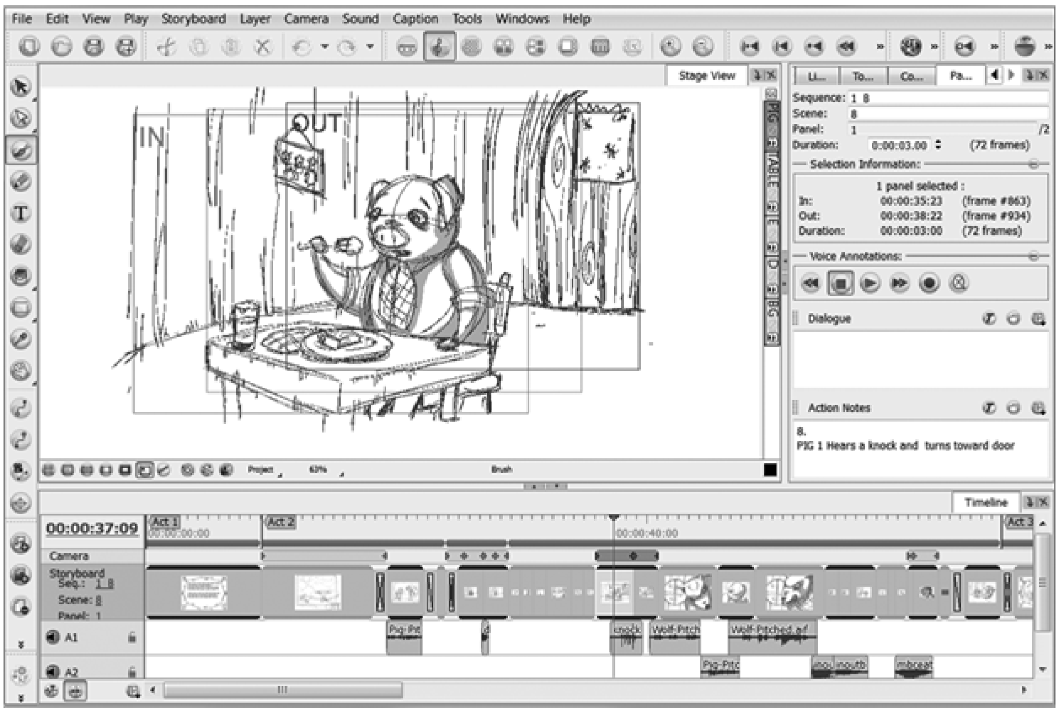


Figure 2.9: Storyboard Pro ©overview of the project. Panels are shown as thumbnails and can easily be rearranged by drag action [2]. (Source: [2])

An animatic is an animated storyboard. The Storyboard's timeline view provides all the necessary tools for creating an animatic. Users can assemble timing of the scene's visuals and sounds by changing the duration of a single panel, modify size and camera position, or add camera movements, transitions between the scenes and the soundtrack to the timeline. Users can also control the playback of selected panels or the entire storyboard [2].



Sequence: 1 8
Scene: 8
Panel: 1 /2
Duration: 0:00:03.00 (72 frames)

1 panel selected :
In: 00:00:35:23 (frame #863)
Out: 00:00:38:22 (frame #934)
Duration: 00:00:03:00 (72 frames)

8.
PIG 1 Hears a knock and turns toward door

00:00:37:09
00:00:00:00
00:00:40:00
Storyboard
Seq.: 1 8
Scene: 8
Panel: 1

Figure 2.10: Storyboard Pro ©timeline view. Creating animatics(Source: [2])

2.3 Autodesk Sketchbook v7 ©: Flipbook feature

Autodesk SketchBook ©is a professional sketching and painting software. It provides a very powerful and intuitive user interface with a variety of tools, allowing artists to easily focus on artistic tasks at hand.

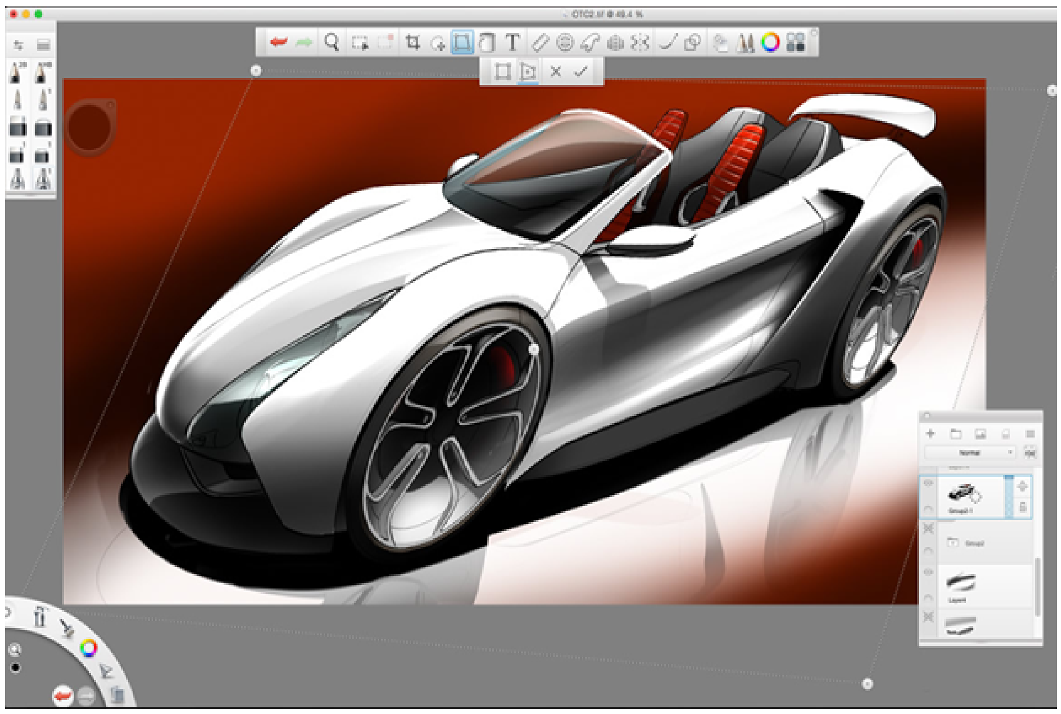


Figure 2.11: Autodesk SketchBook ©graphical user interface (Source: [1])

The flip book feature allows users to enter the animation mode, where they can create a traditional flip book animation within the SketchBook interface. The only difference between the SketchBook and the animation mode is that the layer editor has been modified, containing a fixed number of four layers per frame and a timeline.

Users can either draw a sequence of images or import an existing sequence from software such as Maya or 3DMax.

The animation mode focuses on animating a single scene. Each animation frame (a single image) consists of four layers: foreground, midground,

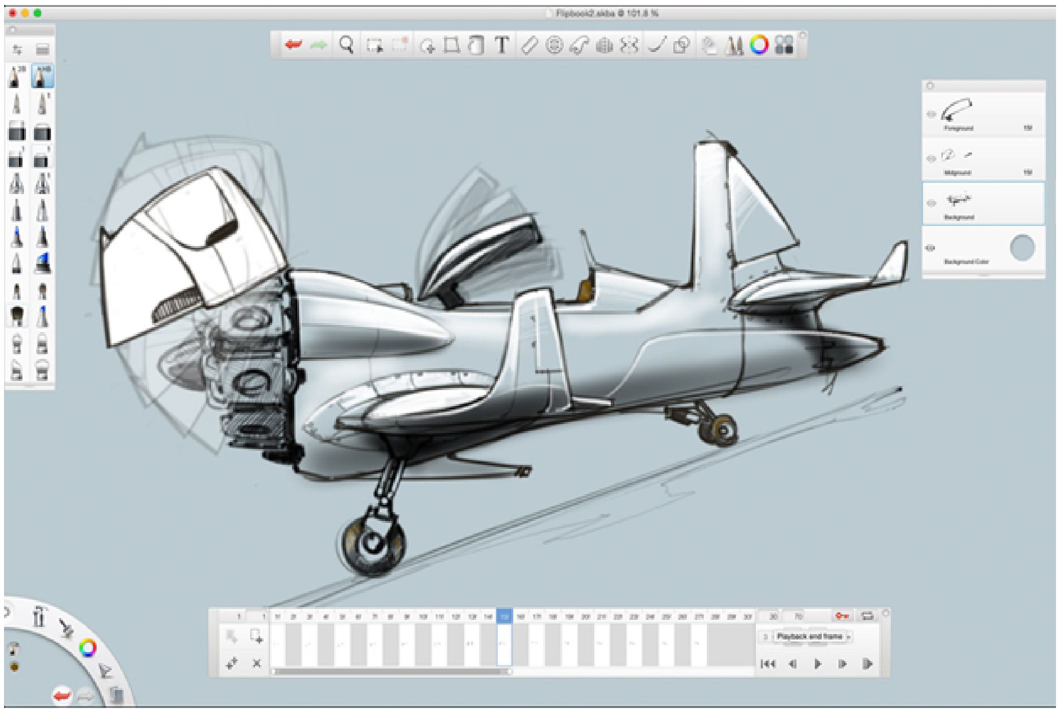


Figure 2.12: Autodesk SketchBook ©animation mode (Source: [1])

background and background color. Users can neither add new layers nor delete existing ones. The foreground and midground layers are the only ones which can be animated, while the background layer is a static image which cannot be changed. As a consequence, it stays the same throughout the animation. In general, the background color layer only sets the background color of the scene.

The timeline tool allows users to add, duplicate, move, rearrange, clear or delete key frames. In addition, it enables the user to insert empty frames, scrub through or playback animation, turn ghosting on and off and set the number of frames ghosted, as well as set the playback range.

3 Designing For Interaction

Design - verb
: to plan and make (something) for a specific use or purpose
: the arrangement of elements or details in a product or work of art

– *Merriam-Webster dictionary*

Interaction - noun
: mutual or reciprocal action or influence
: the act of talking or doing things with other people
: the action or influence of things on one another

– *Merriam-Webster dictionary*

Interact - verb
: to talk or do things with other people
: to act together : to come together and have an effect on each other

– *Merriam-Webster dictionary*

3 Designing For Interaction

This chapter focuses on interaction design. In its essence, interaction design is about building a dialog between a human and an artificial entity. It is about creating a unique and satisfying user experience, through products and services that are easy, effective and pleasurable to use.

Beginning by examining what interaction actually is, this chapter looks at its building blocks or ”raw materials” and characterizes good interaction design. Later, the four major approaches to interaction design are described, namely user-centered design (UCD), activity-centered design, system design and genius design. Finally, the importance of the users is emphasized and the four basic activities of interaction design are introduced.

3.1 What Is Interaction Design?

"Interaction design is the art of facilitating interactions between humans through products and services." [23]

The key to interaction design lies in behavior patterns. Designing a behavior tends to be a tedious task since it requires a deeper understanding of the fluidity of interaction, the fluidity which is described through motion, space, time, appearance, texture and sound [23].

Motion is the basis of all interaction. It is expressed though our unique behavior patterns, which are influenced by emotion, attitude, culture, personality and context. Thus, human conducts tend to vary, which is evident in even the simplest of actions, such as talking (e.g. some people use hand gestures others do not) or walking (e.g. pace) [23].

Motion takes place in a **space**. More specifically, space provides the context for an interaction. In interaction design this context is often a combination of physical and digital space. For example swipe gestures (physical space) when viewing images on our smart phone enable us to see the previous or the next image on our screen (digital space) [23].

Naturally, movement through space requires **time**. The amount of time it takes to process a single task is directly connected to the complexity of the task at hand. Pressing a single button, for instance, can take less than a second, whereas downloading a game via Steam® can take up to a couple of hours, depending on the broadband speed [23].

Appearance provides affordances , meaning that "the qualities or properties of an object define its possible uses or make clear how it can or should be used". Merriam-Webster A physical object is described by properties such as size, shape, weight, color, proportion and structure. A button, for example, has affordance of pushing because of its shape and movement capability. Affordances are contextual and cultural. We know, for instance, how to push buttons because we have pushed at least one before, but a person who sees chopsticks for the first time may be confused and unable to use them properly [23].

Texture describes how an object feels when being held. Thus, the surface and the feel add key features to a devices' appearance. In the world of mobile devices, vibration (vibrating alerts) and heat would be examples of texture.

Even though **sound** is only a minor element of interaction design,it provides audio information, which can enhance the overall user experience. For instance, audio indication when locking the phone can reassure the owner that the phone has been locked properly, without the additional glance at the screen.

These six elements represent the building blocks of interaction design. It is up to designers to manipulate them in order to create interactivity. But the fluidity of interaction cannot be accomplished by mere manipulation. To achieve a good quality, interaction design needs to reflect some or all of the following adjectives [23]:

Trustworthy. The product or service performs as expected and promised by the manufacturer or provider. We are certainly not going to take a bungee jump with suspiciously worn out cords, nor use a mobile phone with the battery life of half an hour. People are less likely to engage with things that could injure them, break lightly, leak their personal data or betray their trust in any way. Trust is the basis for deeper exploration, as well as identification with the product or service.

Appropriate. A product or service should be appropriate for the culture, situation and context in which people live in. Different cultures provide different contexts. For instance, the Japanese and Koreans are advanced mobile phone users and can use expert features that people in many other countries would struggle with. The Europeans use the metric system: meters, grams, liters, Celsius, etc., whereas Americans use the system of imperial units, such as yards, miles, pounds, Fahrenheit, etc.

Professor Geert Hofstede conducted one of the most comprehensive studies on how values in the workplace are influenced by culture. He defined five dimensions [13]:

- **Power distance.** The first dimension describes how a society handles inequalities among people or "The extent to which the less powerful members of institutions and organizations within a country expect and accept that power is distributed unequally.”

- **Individualism versus collectivism.** This dimension describes the ”I” versus ”We” in society. ”Individualism pertains to societies in which the ties between individuals are loose: everyone is expected to look after him- or herself and his or her immediate family. Collectivism as its opposite pertains to societies in which people from birth onward are integrated into strong, cohesive in-groups, which throughout people’s lifetime continue to protect them in exchange for unquestioning loyalty.”
- **Masculinity versus femininity.** The third dimension focuses on the influence of gender roles on society. According to Hofstede, ”A society is called masculine when emotional gender roles are clearly distinct: men are supposed to be assertive, tough, and focused on material success, whereas women are supposed to be more modest, tender, and concerned with the quality of life. A society is called feminine when emotional gender roles overlap: both men and women are supposed to be modest, tender, and concerned with the quality of life.”
- **Uncertainty avoidance.** This dimension describes society’s tolerance towards ambiguity and uncertainty or ”The extent to which the members of a culture feel threatened by ambiguous or unknown situations”
- **Long-term versus short-term orientation.** The last dimension concentrates on how much society values the future over the past and present. ”The long-term orientation stands for the fostering of virtues oriented toward future rewards - in particular, perseverance and thrift. Its opposite pole, short-term orientation, stands for the fostering of virtues related to the past and present - in particular, respect for tradition, preservation of ”face,” and fulfilling social obligations.”

On balance, understanding the context, including cultural, technological and emotional, in which a product or service operates, is essential for a good design [23].

Smart. A product or service needs to be smarter than the users. Products and services are intended to assist humans, therefore, they have to be able to take as much workload off the users as possible [23].

Responsive. Regardless of whether referring to people, products or services, there is nothing more annoying and frustrating than unresponsiveness. Unresponsiveness is simply rude. Users need to know that a product or service ”heard” them and that it is working on the task given by the user. If action is taking time to compute, a good design provides mechanism to inform users that a task at hand is being processed. For instance, an indicator telling users how many minutes/seconds it will take until a certain task is completed [23].

The responsiveness of digital products can be divided in four categories [23]:

- **Immediate.** The product or service responds in 0.1 seconds or less. Users consider response as immediate, i.e. with no perceived interruption.
- **Stammer.** The product or service takes between 0.1 and 1 second to complete. Users notice delay, but if this type of behavior is not frequent, they will probably overlook it. Otherwise the product or service will be perceived as ”laggy ”.
- **Interruption.** The product or service does not respond for more than 1 second. Users perceive the task as being interrupted, and shift their attention from the task to the product or service. For instance, if submit action takes time to compute without any indication, users will start to wonder if the product or service is malfunctioning. Repeated interruption could lead to disruption.
- **Disruption.** The product or service is unresponsive for more than 10 seconds. Users perceive the task as completely disrupted. In such cases, a simple indication, for example a progress bar, would diminish concerns greatly.

Clever. A clever product or service is able to predict the needs of its users and fulfill those needs in pleasing ways [23].

Ludic. A ludic or playful product or service invites its users to ”play” without serious consequences. In this way, users are able to explore the product or service within a safe environment. The ability to undo previous actions, for instance, would reinforce playfulness [23].

Pleasurable. A product or service has to be pleasing, otherwise it will only be used when necessary. This can be accomplished in two ways: aesthetically and functionally. Even though people are arguably more forgiving if something is pleasing to the eye, users are still dissatisfied if a product or service is not doing what it is supposed to do, no matter how sophisticated its design might be. Therefore, products need to be both: pleasing to the eye and efficient [23].

3.2 Four Approaches of Interaction Design

The four major approaches in finding a solution to interaction design problems are [23]:

1. User-centered design (UCD)
2. Activity-centered design
3. System design
4. Genius design

There are a few assertions that apply to all these approaches [23]:

- They can be applied to different products and services, from websites to devices.
- Applying one of these approaches can improve even the most problematic situation.
- The best designers tend to be those who can move between approaches and/or combine them.
- An individual designer may tend to use one of these approaches more than others. Even though one approach can seem more attractive than others, it is important for interaction designer to know all four approaches, because some problems may require different approaches than the preferred one.

The four approaches will now be briefly examined.

3.2.1 User-Centered Design (UCD)

The user-centered design philosophy is based on the premise ”users know best.” [23] Rather than requiring users to adapt in order to learn how to use it, a product or service is designed to support the goals, needs and preferences of users, thus making the users the center of the design process.

The goals disclose the intention of the users and play a central part in the design process. It is up to interaction designers to provide the means necessary for accomplishing these goals. The means, however, are again described by the needs and preferences of users [23].

3.2.2 Activity-Centered Design

Unlike the User-centered design, which focuses on the goals and preferences of the users, the activity centered design focuses on activities, i.e. a cluster of tasks (actions and decision) that are performed for a specific purpose. The purpose of an activity is not necessarily a goal, even though they may overlap. For instance, when raking the leaves, the gardener may have the goal to tidy the yard, but the purpose of using a rake is to collect the leaves. In contrast, making tea has the same goal and purpose: to drink tea [23].

Activity-centered design is guided by activity, not by people doing the activity. Therefore, the main purpose is to design solutions to help users accomplish certain tasks, rather than to achieve a specific goal [23].

3.2.3 System Design

System design is simply a process of designing a system, i.e. a set of entities (people, devices, machines and/or objects) that act upon each other. It is a rigorous design methodology suitable for tackling complex problems [23].

The main focus of the system design are not users, but the context in which users act. However, the goals and needs of users are not completely disregarded. They are used to set the goal of the actual system [23].

3.2.4 Genius Design

Genius design relies solely on the experience and the creativity of an individual designer. All design decisions are being made by the designer himself/herself. The designers use their creativity and experience to determine wants, needs and expectation of users. Users, if involved, are usually there to test the end product or service and validate the design concept. Reasons for choosing this approach include a lack of resources or the inclination to involve users in the design process [23].

Needless to say, this approach is recommended exclusively for experienced designers [23].

Approach	**Overview**	**Users**	**Designer**
User-centered design	Focuses on user needs and goals	Guides the design	Translates user needs and goals
Activity-centered design	Focuses on user activity	Performs activities	Creates tools for action
Systems design	Focuses on the parts of the system - context	Sets goals of the system	Assembles the system
Genius design	Relies on experience of designer	Validates design concept	Makes all design decisions

Table 3.1: Overview of the four interaction design approaches (Saffer 2006)

3.3 The process of Interaction Design

Interaction design is a practical and creative activity with the aim of developing products or services that support users needs, wants and goals, its main purpose always being to establish a dialog between a person and an artificial entity.

Every design discipline includes three fundamental activities [22]:

1. Understanding the requirements.

2. Producing a design that satisfies those requirements.
3. Evaluating the design.

The interaction design is guided by the wants, needs and goals of users. The users are involved and are directing the development of the design itself. To completely integrate users in the design process, interaction design extends the above mentioned list by one extra activity: prototyping. Prototyping allows users to interact with a design before it reaches its end stage [22].

The following chapter is going to explore the significance of user involvement and the four major activities of interaction design.

3.3.1 The Importance of Users

Interaction design focuses on the actual users of the products or services, but why is it so important to involve the users in the design process?

Often, when eliciting requirements, developers talk to higher level managers. Even though their input provides significant information about the system, their perspective is often very different to the end users', i.e., the users who interact with the product or service on a regular basis [22]. For instance, several years ago, I worked on a project, where all design decisions were made by higher level managers. These decisions were based on their own understanding of the underlying workflow. The actual users of the system were completely left out of the decision making process. Once the first version of the software was deployed, there was a lot of confusion. It turned out that the workflow the managers were describing was different to the one being actually used. Therefore, the higher level managers might provide a significant insight to the problem that interaction design is trying to solve, but unless they are performing the task on a daily basis, their perspective will be very different from someone who is really using the product or service.

Involving users in design development plays an important role in creating usable products and services, not only in terms of functionality but also with reference to expectation management and ownership, two equally important aspects [22].

Expectation management ensures that the expectation of users, with reference to the product or service, is realistic. The goal is to avoid misrepresentation of a product or service by making empty promises, which could lead to frustration and complete rejection of the product or service [22]. Unfortunately, making profit is often perceived as more important than meeting users' expectations. This often leads to exaggerated marketing strategies. However, if a product or service is below users' expectations, sales will drop and the product or service will no longer be used.

An early involvement of users in the development process has two advantages [22]:

(A) The users can see from an early stage on what the product or service is capable of.
(B) The users can understand better what to expect once the final version of a product or service is available.

Sense of *ownership* rises from the feeling that users have contributed to the development of a product or service. The users are more likely to support the product or service in whose development they were personally involved in [22].

3.3.2 The four basic activities of interaction design

The process of interaction design involves four basic activities [22]:

1. Establishing requirements
2. Designing alternatives
3. Prototyping, or creating an interactive version of design
4. Evaluating

Establishing Requirements

The basis of all engineering is answering the questions who, what, why and how. The establishing requirements has two aims [22]:

1. Understanding as much as possible about the users, their activities and the context in which the product will be used.
2. Based on the understanding of the users and the context they work in, one can produce a set of requirements for the product or service.

This can be achieved in two individual steps [22]:

1. Determine the target user group and their needs, preferences and goals.
2. Establish requirements that form a sound basis for design, using the information obtained in step one.

What are requirements?

"A requirement is a statement about an intended product that specifies what it should do or how it should perform." Rogers et al. That being said, it is vital for interaction design to make the requirements specific, unambiguous and as clear as possible [22].

In general, interaction design involves a wide range of requirements, which broadly can be divided in the following categories [22]:

Functional requirements - fundamental requirements which describe what the product or service should do.

Data requirements - describe in detail the data that a product or service is going to handle.

Environmental requirements - describe the context in which the product or service will operate. There are four aspects to consider when establishing environmental requirements:

1. **Physical environment** - light, noise, movement etc., i.e. everything concerning physical interaction with interface.
2. **Social environment** - social aspects, such as collaboration and coordination.
3. **Organizational environment** - organizational hierarchy and user support.
4. **Technical environment** - everything regarding the technology that is being used.

User characteristics - capture the essence of the target user group, everything from cultural background, nationality, preferences, education, mental and physical disabilities to user expertise: novice, expert, casual or frequent user.

In order to capture the essence of targeted user group, users are often transformed into personas. Personas is a profiling technique that models the typical users of the product or service. Persona represent a fictional person who captures the most important aspects of the targeted user group, allowing designers to focus more precisely on the target users.

Each persona has a name, a biography, hobbies and unique goals. The goals represent the objectives persona wants or needs to fulfill by using product or service. Beside goals, the descriptions of personas include behavior patterns, attitudes and environment.

During design processes, a small set of personas is required to achieve the needed diversity, but the designer should always choose one primary persona, who represents a large section of the target group.

Usability goals and user experience goals - identify requirements that meet usability and user experience goals.

Data Gathering for requirements

To establish or clarify existing requirements, interaction designers need to collect sufficient, relevant and appropriate data They need to cover a broad spectrum of issues, such as the tasks that users are currently performing, their goals, the context in which tasks are being performed and the rationale for the current situation [22].

The common methods for data gathering are [22]:

- **Interviews** - asking users about their goals and tasks directly.
- **Focus groups** - gathering users for a discussion about a certain product in order to learn their wants, needs and goals.
- **Questionnaires** - used to get initial response about a product or service.
- **Direct observation** - gaining insight about tasks and the context in which these tasks are performed by observing users directly.

- **Indirect observation** - using diaries and interaction logging to gain insight into tasks and the context in which they are executed.
- **Studying documentation** - gathering activity data by reading manuals and other documentations.
- **Researching similar products** - looking at similar products in order to develop alternative designs or to induce requirements.

Task Description

Task description is used to envision the product or service that is being developed. The common description types are scenarios, use cases and essential use cases (also known as task cases) [22].

This section introduces two description types that are used in this thesis: the user story and the scenario.

The user story is a short, simple and goal-oriented one-sentence statement that describes a feature that a user would like to have. They are usually written on a small index card and have the following structure:

As a <type of user>, I want <some goal>so that <some reason> [21]

"Scenarios are prototypes built of words."Saffer A scenario is a story about fictional users (personas) using a product or service to achieve their goals. The purpose of such a story is to explore the contexts, needs and requirements.

An example of a Pocket Toon scenario:

Sophie just finished her toon. She has been drawing for hours now, bringing her character to life with every page. She even added some music for dramatic purposes. It is a half-minute tale about her cat Tiger and his mischievous ways. As she plays the video, she notices that one of the scenes requires small adjustments. She opens the animation's "filmstrip" screen, selects the scene and corrects it. She plays the video again. It is perfect and ready for sharing with her friends.

Designing Alternatives

Once requirements have been established we begin with the design activity. The design activity is an iterative three-step process: design-evaluate-redesign.

There are two types of design: *conceptual design* and *physical design*.

Conceptual Design

Conceptual design is a process of translating previously defined requirements into conceptual model. The conceptual model contains all the concepts necessary to understand what a product is capable of and how to interact with it. Which concepts are going to be used depends on the target users, type of interaction, type of interface, terminology, metaphors, etc [22].

There are four key guiding principles [22]:

1. Keep an open mind without forgetting the users and their context
2. Discuss ideas with users as much as possible
3. Use low-fidelity prototyping to reduce feedback time
4. Iterate

The first step when working with conceptual design is getting familiar with the data collected about users and their goals, in order to try empathizing with them. By doing so, the designer gets a better understanding of the desired user experience. Needless to say, at this stage, all design decisions have to be technologically feasible [22].

Next, the designer develops an initial conceptual model. There are a couple of approaches that the designer could consider at this stage, namely [22]:

Interface metaphors help users to understand the product or service by combining familiar and new concepts. An example of interface metaphor would be a magnifying glass to zoom. Erickson proposed a three-step process for generating a good interface metaphor: *identify functional requirements, understand user problems and generate metaphor*.

Interaction types determine which type of interaction is provided by a product or service. There are four different types of interaction: instructing, conversing, manipulating and exploring. It is up to the designer to determine which type or combination of types suit the product or service best.

Interface types highlight some aspects of a product or service. Experimenting with different interfaces at this stage is encouraged, because the designer will be able to observe different aspects of a product or service. This could lead to alternative designs.

And finally, before the designer can assemble a prototype, the initial conceptual model needs to be expanded by answering the following questions:

1. **What functions will the product perform?**
2. **How are the functions related to each other?**
3. **What information needs to be available?**

Physical Design

The physical design focuses on physical or visual aspects of the product or service such as colors, sounds, images, materials, interface layout, icon design and so forth [22].

Prototyping

Prototypes are interactive draft versions of a product or service. In essence, they are used for simulation. They can be of low or high quality. The purpose of prototyping is to explore some characteristics of the product or service, thus emphasizing one set of characteristics over another. At first glance, this may seem very restrictive, but that is not the case. When designers prototype one set of characteristics, they present one possible design approach. Designers can use these different approaches later, in order to experiment and see what works best for the users. Often, multiple well-working prototypes are combined into a single hybrid-prototype, capturing the best of all designs [23].

There are three types of prototype [23]:

Paper prototypes. These are low-quality prototypes, which are very simple, cheap and quick to produce or modify. They are used to quickly demonstrate the workings of a product or service. Each piece of paper represents a different aspect of the design. For instance, each piece of paper can be used to demonstrate different screens of a mobile application. With each interaction, e.g. pressing a button, users can either update the current screen or go to different a screen.

Digital prototypes. Digital prototypes are easy to distribute and can be of either high or low quality. There are two kinds of digital prototypes:

- Low functional. There is no real interaction and users are only able to click through images - very similar to paper prototypes.
- High functional. Users are able to interact with the product or service. In this way, the prototype is very similar to the final version of a product or service. The drawback of high quality prototyping is that the prototypes often get mistaken for real products or services.

Physical prototypes. This type of prototypes can be as small as a control and as large as rooms. Physical prototypes can be of high quality and low quality. For instance, a prototype can be made of exactly the same material as the final product, or it can be made of alternative materials, such as wood or clay.

Evaluating

Evaluation is a process of collecting information about user experience in order to determine the usability and acceptability of design [**?**]. The main purpose of evaluation is to expose design flaws. Once exposed, the flaws will be dealt with in the redesign process, which again will be included in the evaluation process and so on. This is due to aforementioned iterative nature of the design: design-evaluate-redesign, the purpose being to achieve user approved quality.

The evaluation can be classified in three categories [22]:

1. **Controlled settings involving users.** Evaluates hypotheses or behaviors. All activities are controlled by evaluators. The evaluators decide which task users should perform, as well as its duration and the settings of its execution. The main methods are usability testing and experiments.

Quality	**Advantages**	**Disadvantages**
Low quality	Lower development cost Evaluate multiple design concepts Useful communication device Address screen layout issues Useful for identifying market requirements Proof-of-concept	Limited error checking Poor detailed specification to code to Facilitator-driven Limited utility after requirements established Limited usefulness for usability tests Navigational and flow limitations
High quality	Complete functionality Fully interactive User-driven Clearly defines navigational scheme Used for exploration and test Look and feel of final product Serves as a living specification Marketing and sales tool	More expensive to develop Time-consuming to create. Inefficient for proof-of-concept designs Not effective for requirements gathering

Table 3.2: High vs Low Quality Prototyping (Rogers et al. 2011)

2. **Natural settings involving users.** Evaluates usage of a product or service in the real world. Evaluators have little or no control over activities. The main method is the use of field studies.
3. **Any settings not involving users.** The interface flaws are detected by consultants and researchers. The range of methods includes inspections, heuristics, walkthroughs, models, and analytics.

4 Pocket Toon Design Process

This chapter describes the design process of Pocket Toon. First, the general approach of Pocket Toon, with reference to interactive design problems is explained. Second, the analog flip book and Nintendo's Flipnote Studio^{TM} are analyzed, and their existing features are used as a basis for the features of Pocket Toon. These features are refined by personas, user stories and scenarios. The conceptual design and the grouping of requirements are based on the information acquired in the analysis of existing flip book applications. Third, the process of determining the features of the prototype is outlined. Finally, the feedback of a small group of test users will be evaluated and used for further considerations concerning the conceptual design and the prototype.

4.1 Design development process

The previous chapters discussed interaction design, its basic building blocks or ”raw materials”, as well as qualities of good design. Further, the four approaches of interaction design and the importance of the user have been highlighted. Moreover, the four basic activities of interaction design, namely establishing requirements, designing alternatives, prototyping and evaluating, have been defined.

Using this body of theoretical background on interaction design, the following chapter is going to apply interaction design to the smart flip book Pocket Toon and describes related design problems.

The first step was to decide on a target user group. In the case of Pocket Toon, it was decided to focus on teenagers, i.e., users from 13 to 19 years, as this is also the target user group on Pocket Code on which Pocket Toon is based. The chosen platform was Android, simply because Pocket Code’s Android version is the only one already released to the public. In the future, once Pocket Toon’s Android version and as Pocket Code’s versions for other platforms, e.g., iOS, will been released, it is planned to develop also versions of Pocket Toon for other platforms so that users can share and remix their creations without having to worry about supported platforms.

In order to solve the design problem, it was decided to take a slightly modified genius design approach. In this context, slightly modified means that users were involved before the final version of the app was designed. The needs, preferences and goals of the users were defined by the designer without user-involvement in order to work on the app. However, during the first design iteration, the prototype was tested by a small group of interns, whose age correlated with the target user group. The reason for choosing the genius design approach was due to the fact that I, the designer, have been an end user of similar applications for over a decade, i.e., also when I was a teenager myself, and that my interests thus were and still are similar to those of the targeted audience.

After choosing the approach, the requirements for the app were decided on. As mentioned before, the design development process had minimal user involvement, therefore it was up to the designer to determine the needs, preferences and goals of the users. The first step was to examine the actual analog flip book, which has been around since 1869. After evaluating the analog version, the existing software solutions were analyzed. Nintendo's Flipnote Studio$^{\text{TM}}$, which has already been discussed in Chapter 2, turned out the be a subtle basis for the Pocket Toon application.

There are two reasons for this:

- Nintendo's Flipnote Studio$^{\text{TM}}$ is a digital flip book.
- It gained a lot of popularity in a short amount of time, which suggests that its interaction design is comparatively attractive to users.

Finally, personas, user stories and scenarios were created, which were used in order to guide decisions in the design process.

In essence, establishing requirements for the application was a three step process:

1. Analysis of analog flip book interaction.
2. Study of Nintendo's Flipnote Studio$^{\text{TM}}$ which accordingly inspired the Pocket Toon design.
3. Deciding on the target audience, and creating personas, user stories and scenarios, in order to improve user-related design decisions.

The above mentioned analysis was used to build the conceptual design. To form features, requirements were defined and grouped, based on their functionality. Each feature has been illustrated with interaction schematics.

Once the conceptual model had been established, the agile approach was applied to prototyping:

1. The features were ranked based on their importance.
2. Based on this ranking it was decided which features were going to be designed and implemented first.
3. The prototype for selected features was designed.

At this stage, an analysis of the Pocket Family, i.e., Pocket Paint and Pocket Code, has been conducted for the following reasons:

1. **Reusability.** The task was to determine if these apps implement some of the already established features.
2. **Familiarity.** Both apps have a solid user base. By reusing already familiar concepts, the learning curve of the users will be much shorter, since they will be working with something they are already familiar with – something they already ”know”.
3. **Compatibility.** Users will be able to open flip books created with Pocket Toon in Pocket Code. Thereby, they will be empowered to extend the functionality of their creations with, for instance, more interactivity. This also provides a way towards more advanced coding concepts for users.

The prototype was tested with interns, whose age correlated with the target group. The testing provided valuable feedback, which inspired some additional research and led to a refinement of the design.

The refined prototype was used as the basis of the software implementation. Based on previous functionality and the analysis of the exisiting Pocket family of apps, it became obvious that Pocket Paint could provide the complete drawing functionality. Thus, it was decided to completely integrate it in Pocket Toon. This is highly beneficial because:

(A) There is no need to ”reinvent the wheel” by implementing yet another paint editor.
(B) The benefit of reusing the concepts that are familiar to 200,000 users (more than 2,000 users reviewed Pocket Paint in Play Store as of April 2016) is that if users get interested in Pocket Toon, they will only have to learn a few new features.
(C) Using the already existing user basis enables a broader adoption.

However, the analysis also revealed that:

(A) Pocket Paint is missing one crucial feature – layers. Layers allow an artist to separate different elements of the image. With layer functionality, users could separate foreground and background elements of flip book pages. In addition, this functionality could be used to allow tracing of previous images.

(B) If Pocket Paint is integrated into the Pocket Toon app, there is also a need for a library, the reason being that designer would reuse the core functionality of Pocket Paint but add designs and interaction specific to Pocket Toon..

Therefore, the software implementation is a three step process:

1. Implementation of missing feature in Pocket Paint, i.e., layers.
2. Refactoring Pocket Paint into a library.
3. Implementing Pocket Toon.

Due to the limitations of the thesis' scope, it was not possible to fully complete all three steps. The implementation of the layers, however, was successful. In order to create the library, Pocket Paint had to be refactored, completely separating the core functionality and GUI. This turned out to be a very complex task, because the current implementation did not support the library requirements. Thus, Pocket Paint has been prepared to be used as a basis for Pocket Toon, and the first version of Pocket Toon has been conceptually designed, but the implementation of Pocket Toon exceeded the scope of this thesis.

4.1.1 Establishing the Interaction Design Requirements

Analog Flip Book

When considering the functionality of the analog flip book, the following activities come to mind: One can:

1. Turn pages and move forward or backward, similar to a regular book.
2. Draw on each page or add images.
3. Add a new page to the flip book animation.
4. Remove the pages one does not need anymore.
5. Disassemble the flip book into pages.
6. Rearrange the pages.
7. Flip the pages at a different speed, thus creating the animation.

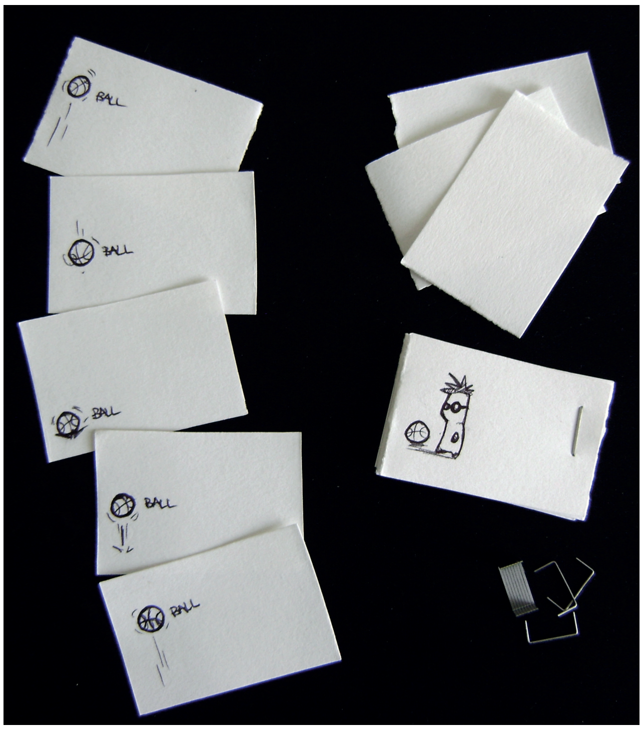


Figure 4.1: Analog flip book – can be made with paper and staples.

In order to gain an understanding of the needs, preferences and goals of the users, the functionality has to be translated into user stories. In addition, one has to follow interaction reverse engineering, starting by observing the end product and its interaction. Later the needs, wants and goals of the users are taken under consideration.

Thus, the following user stories were extracted as a basis for the work on the application described in this thesis:

As a flip book enthusiast, I want be able to flip forward so that I can either see the next drawing or get to the next blank page.

As a flip book enthusiast, I want be able to flip backwards so that I can see the previous drawing(s).

As a flip book enthusiast, I want be able to draw on a blank page so that I can create one frame of my animation.

As a flip book enthusiast, I want be able to add an image the size of my flip book page so that I can create one frame of my animation.

As a flip book enthusiast, I want be able to add blank pages so that I can expand my animation.

As a flip book enthusiast, I want be able to remove pages I do not need anymore so that I can refine my animation.

As a flip book enthusiast, I want be able to disassemble my flip book so that I can get an overview of all the pages I have created so far.

As a flip book enthusiast, I want be able to rearrange my flip book pages so that I can refine my animation.

As a flip book enthusiast, once the flip book is assembled, I want be able to flip the pages at different speeds so that I can create an animation.

This list established the first set of functionality requirements for a flip book's digital counterpart.

Animation Software

Chapter 2 provided a rough overview of Nintendo's Flipnote Studio™. This section analyzes its features in greater detail and describes which features were selected to be included in Pocket Toon, thus refining and expanding the first set of functionality requirements.

After being started/opened, the Flipnote Studio welcomes users with the following screen:

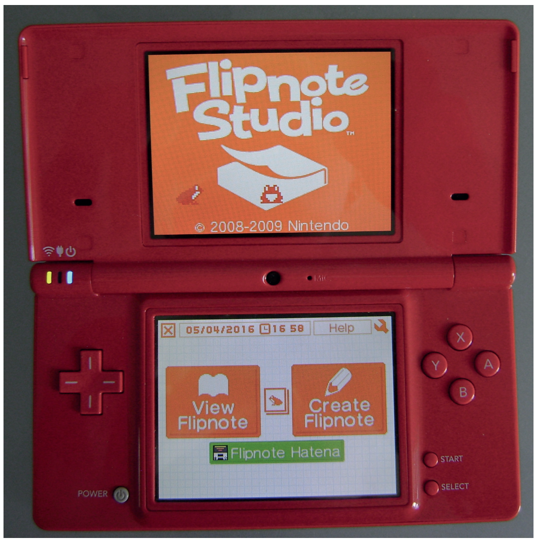


Figure 4.2: Flipnote Studio™main menu

Users can either view previously created flipnotes (gallery feature), create a new flipnote or visit the Flipnote Hatena site to review flipnotes of other users or embed their animations in other web pages.[26] Unfortunately, the Flipnote Hatena service was terminated on May 31, 2013. As a consequence, its features could not be analyzed, and their analysis thus is not part of this thesis.

Beside allowing users to create new flipnotes, as well as save them in the gallery, Nintendo's Flipnote Studio™permits users to share their works and review the animations of others, thus incorporating a social factor. It is this social interaction that the Pocket Toon application also aspires to.

When starting a new flipnote project, one can see the following features on the screens:

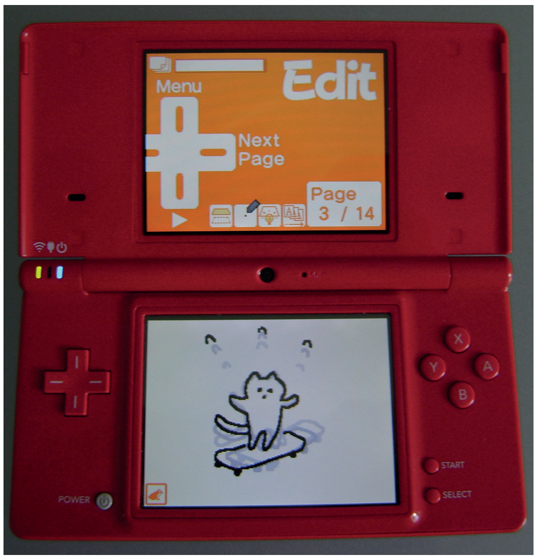


Figure 4.3: Flipnote Studio^{TM}: create a new flipnote.

The bottom screen provides a surface for drawing and stylus interaction, while the upper screen provides the following pieces of information:

- Which options are bound to which arrow:
  - Up – Tool menu
  - Right – Create new page
  - Down – Play animation
- Information page and tools that are being used:
  - Whether the foreground or background layer is selected
  - Which tool is currently being used
  - Whether the current page should be copied to all following pages
  - Whether the tracing [1] has been activated

[1]In traditional animation, the lightbox is used to create an accurate sequence of drawings by tracing the existing drawing. The new paper sheet would be placed on existing drawings, allowing the animator to trace the non-moving parts and slightly reposition parts that should be animated, thus illustrating the movement. For example, in order to

- Flipnote page counter.

The user can access the tool menu either by pressing the up arrow or by clicking the frog icon in the lower left corner of the interaction screen. Once in the menu, the user can either select between different tools, quit to main menu, send the flipnote to other users or save the flipnote.

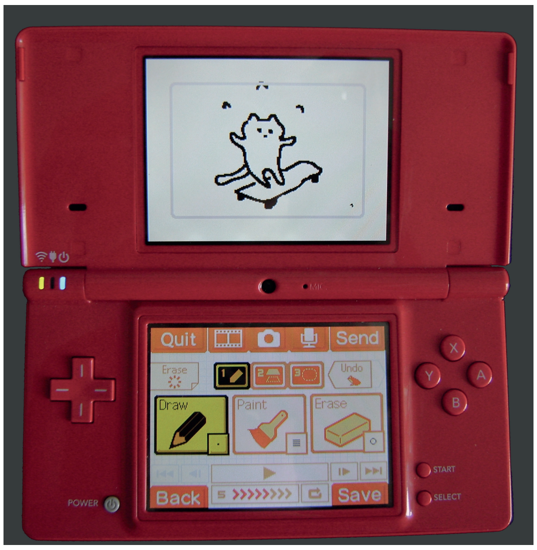


Figure 4.4: Flipnote Studio^{TM}tool menu.

As mentioned in Chapter 2, the tools can be grouped in five categories:

1. **Playback tool.** This tool is visible throughout the application. Users can start the animation, adjust speed, add new pages, jump to first/last frame and set a repeat flag.
2. **Single page tools.** This tool allows users to draw, modify foreground, background and draft layer, copy drawings across other pages, modify paper color, show/hide previous pages and select, as well as modify different objects on a certain page.

animate characters, the key animator would first draw key poses of the characters on different sheets. E.g., if a character waves, two key positions on two different paper sheets are needed: the hand has to be positioned at the far right and on the second sheet at the far left. Then the assistant animator would draw "Inbetweens", meaning the remaining drawings that link the key poses. Taking the example of the waving character, this would include all the hand positions between the two key positions – far right and far left [25].

3. **Animation overview tool.** The animation overview tool enables users to rearrange pages, add, delete insert and copy/paste pages.
4. **Camera tool.** Users can import images from the system memory and modify them by changing brightness and contrast.
5. **Sound tool.** The sound tool allows users to add music or sound effects to the animation.

The process of creating a flipnote is very straightforward. Users can select a tool they would like to work with and select the layer on which they would like to draw – background or foreground. Once finished with the current page, the users can create a new page by pressing the right arrow. If the tracing option is activated, the users will see a transparent version of previous drawings. The foreground and/or background can be copied on the next pages by activating the "Copy" option, otherwise they have to be redrawn by hand each time a new page is added. Beside drawing, users can also import images from the system memory into their flipnote pages.

In order to add music and sound effects to their animation, users have to use the sound tool. Users can add sounds by recording them with the Nintendo DSi microphone. There are two types of sounds that can be added to the animation:

- Sound track – whose length corresponds to animation length.
- Sound effects – max. 3 sound effects, each max. 2 seconds.

The animation can be played either in the page edit mode by pressing the down arrow, which allows users a quick review of the animation while still working on it, or in the main menu by using the playback tool.

The last tool interesting for designing Pocket Toon, is the animation overview tool, which allows users to fully review and edit their animation by adding, deleting, copying or rearranging pages.

Compared to the analog flip book, overlaps concerning functionality can be detected. However, the execution differs between the analog and the digital version. Similar to the analog flip book, users are able to flip through pages, draw foreground or background, insert images, add or delete pages, get an overview of the animation, rearrange pages and play the animation. The only new functionality regarding the creation of the flip book is adding the sound.

The gallery allows users to revisit their previously created flipnotes.

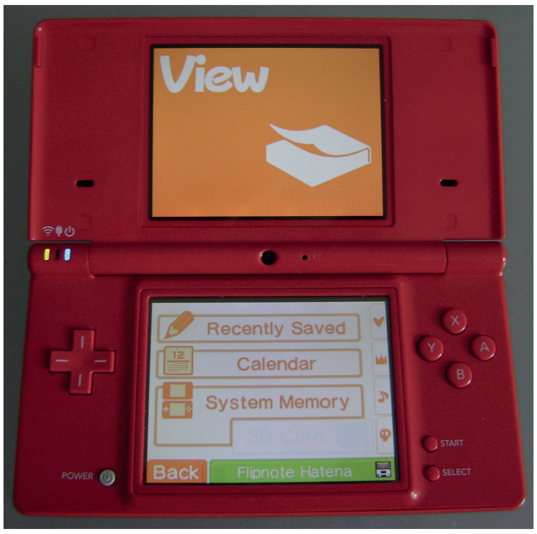


Figure 4.5: Flipnote Studio^{TM}gallery menu

The gallery offers filter functionality. The users can either view recently saved flipnotes, search flipnotes by date or view all flipnotes that have been stored on the device or an SD Card. They can also visit the Flipnote Hatena sharing platform, or view previously grouped flipnotes by selecting one of the icons on the right menu bar.

In the gallery overview, selecting a flipnote will automatically result in starting the flipnote animation on the top screen. The flipnotes can either be edited or viewed for details. In the Details view, users can delete a selected flipnote, send it to other users, play it or assign a sticker to it (form of labeling and grouping).

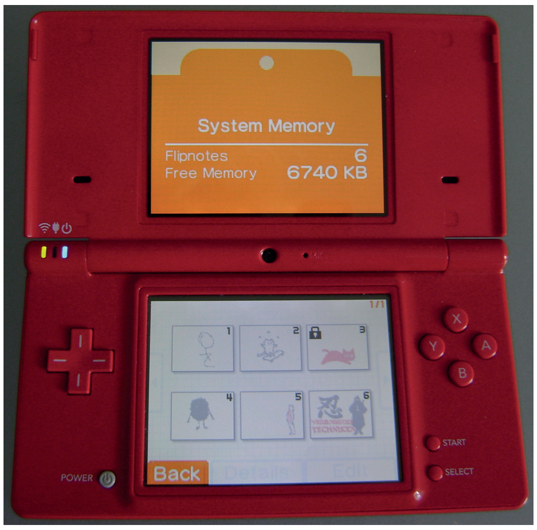


Figure 4.6: Flipnote Studio^{TM}gallery overview.

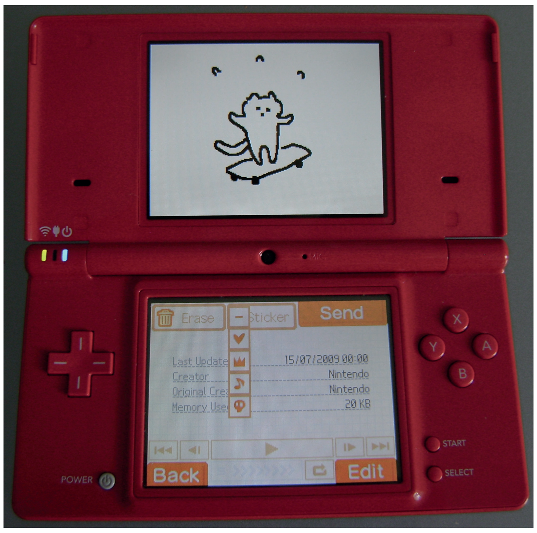


Figure 4.7: Flipnote Studio^{TM}Details View.

After reviewing Nintendo's Flipnote Studio^{TM}, it was concluded that the gallery and the possibilities of publishing and reviewing animations on social platforms should be added to Pocket Toon as well. The users can benefit greatly from social platforms since they can:

- Share their work with the rest of the community
- Learn from each other by providing feedback via comments and ratings
- Find people who inspire them
- Exchange ideas and collaborate
- Specific to Catrobat's Pocket family of apps:
  - Pocket Toon users will be able to share their creations not only on Pocket Toon's sharing platform, since their creations are automatically available on Pocket Code's sharing platform as regular Pocket Code programs, because the internal data format of Pocket Toon animations is a subset of and upwards compatible to the internal data format of programs created with Pocket Code. This will allow Pocket Code users to extend animations originally created with Pocket Toon with interactions such as tapping or reactions to sensor input that cannot be realized with the means provided by Pocket Toon alone. The intended goal is that, by becoming aware of Pocket Code's possibilities (e.g. they get notified if their toons have been remixed by Pocket Code users), Pocket Toon users might become interested in learning more advanced concepts of programming and thus motivated to start using Pocket Code themselves in order to create interactive art, animations, and game apps.
  - Pocket Toon users could also contribute to collaboratively created games by providing Pocket Code users with animations, which the latter could then easily integrate in their own games.

Therefore, the following user stories have been added:

As a Pocket Toon user, I want be able to add music to my animation so that I can portray the emotions of the scene better.

As a Pocket Toon user, I want be able to view all the animations I've created *so far, so that I can play or edit them again.*

As a Pocket Toon user, I want be able to share my animations with other users so that I can give and receive valuable feedback, e.g., in the form of reviews or comments.

4.1.2 Personas

This section is going to present personas that were used in order to describe the target audience. The personas method is a profiling technique that models the typical users of the product or service. A persona represents a fictional person who captures the most important aspects of the targeted user group, allowing a designer to focus more precisely on the real needs of one typical user as compared to the compound needs of an inhomogeneous group of different users.

Each persona has a name, a biography, behavioral patterns, interests and goals. The goals represent the objectives a persona wants or needs to fulfill by using a specific product or service

Our primary persona is Sophie, a young artist whose dream is to become a recognized mangaka – a japanese comic book artist. Sophie conveys her emotions through pen and paper, i.e., she speaks through her art.

The second persona, Emily, is outgoing and very articulate. She likes hanging out with her friends and capturing every precious moment with her camera. She represents the more general audience assumed to be interested in the Pocket Toon app.

Sophie

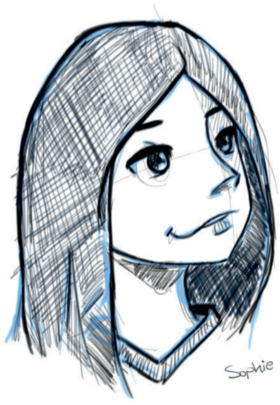


Figure 4.8: Sophie. She does not like to be photographed, so she draws a fast sketch of herself.

Sophie is 13 years old and lives with her parents on the outskirts of the Graz. She is attending the 3rd year of lower secondary school. Every morning, she takes a 45 minutes bus ride to school, but she does not mind, since she gets to see her friends, especially her best friend Marie. She is interested in arts, music and all sorts of creative endeavors, hoping that one day she will be a recognized mangaka (author of manga – Japanese comics). In her free time, she plays piano, draws and reads manga, watches anime, and pets her cat Tiger.

General Information
Programing experience: None
Smartphone: Samsung Galaxy S3
Daily usage: Approx. 4h/day
Favorite apps: Deviantart
Download Behavior: Good Play Store rating

Smartphone Behavior:

Sophie owns a smartphone for about a year. She mainly uses her phone to take pictures of her drawings, edit her newly crafted artwork and post it on social platforms such as Facebook, Instagram and Deviantart. Therefore, she likes to try out different editing and drawing software. She even bought a stylus. Sophie also uses WhatsApp to chat with her friends and make appointments.

Her end goals:

- Wants to create art easily without reading complex manuals
- To publish her art and see what others think of it
- Wants to have fun, and drawing is fun to her

Emily

Figure 4.9: Emily

Emily is 16 years old and lives with her parents in Graz. She is attending the 2nd year of high school. Every morning, she meets with her friends and takes a 15 minutes walk to school. She likes hanging out with her friends and taking pictures of their daily activities. She likes to print and pin these pictures to a board, right above her desk, making a collage of her favorite memories. In her free time, she rides her horse Chester.

General Information
Programing experience: None
Smartphone: Sony Xperia Z3 Compact
Daily usage: Approx. 5h/day
Favorite apps: Facebook, Instagram and WhatsApp
Download Behavior: Based on suggestions from friends

Smartphone Behavior: Emily owns a smartphone for about three years. She mainly uses her phone to take pictures of memorable moments with her friends and broadcast these experiences through Facebook, Instagram and WhatsApp.

Her end goals:

- Wants to create ”moving photo collage” easily and be able to add music to it
- Wants to publish her videos among friends
- Wants to have fun, while sharing her experience with others

4.1.3 User Stories and Scenarios

The user stories represent a short description of features the user would like to have. They are usually described with a few keywords written on small index cards. The goal is to have a short and simple description of a feature told from the perspective of the user.

In the previous section, the following user stories have been collected:

As a flip book enthusiast, I want be able to flip forward so that I can either see the next drawing or get to the next blank page.

As a flip book enthusiast, I want be able to flip backwards so that I can see the previous drawing(s).

As a flip book enthusiast, I want be able to draw on a blank page so that I can create one frame of my animation.

As a flip book enthusiast, I want be able to add an image the size of my flip book page so that I can create one frame of my animation.

As a flip book enthusiast, I want be able to add an image the size of my flip book page so that I can create one frame of my animation.

As a flip book enthusiast, I want be able to add blank pages so that I can expand my animation

As a flip book enthusiast, I want be able to remove pages I do not like so that I can refine my animation.

As a flip book enthusiast, I want be able to disassemble my flip book so that I can get an overview of all the pages I have created so far.

As a flip book enthusiast, I want be able to rearrange my flip book pages so that I can refine my animation.

As a flip book enthusiast, once the flipbook is assembled I want be able to flip the pages at different speed so that I can create an animation.

As a Pocket Toon user, I want be able to add the music to my animation so that I can portray the emotions of the scene better.

As a flip book enthusiast, once the flipbook is assembled, I want be able to flip the pages at different speed so that I can create an animation.

As a Pocket Toon user, I want be able to share my animations with other users so that I can give and receive valuable reviews.

This section is going to describe the user stories in greater detail. However, before proceeding with the user description, the following points have to be clarified:

(A) Sophie is the primary persona, therefore all the stories are told from her perspective.
(B) The term animation will be replaced with **toon**, which is an informal definition for the cartoon, which is also the explanation for the app's name Pocket Toon.

From a high-level view, the user stories can be grouped in four categories, i.e., top stories:

1. Create New Toon
2. Continue Toon
3. Toon Gallery
4. Online Sharing Platform

The sections below will describe the stories for each category.

Create New Toon – As an artist I want to be able to create a new toon...

As an artist I want to be able to create new pages (frames).

As an artist I want to be able to draw or insert images into a new page.

As an artist I want to be able to draw or insert an image into the background.

As an artist I want to be able to draw or insert an image into the foreground.

As an artist I want to be able to use different drawing tools, as well as the camera.

As an artist I want to be able to select the size of drawing tools such as for the brush eraser or the pen.

As an artist I want to be able to select the color with which I draw.

As an artist I want to be able to select objects I have drawn so far.

As an artist I want to be able to rotate selected objects.

As an artist I want to be able to move selected objects.

As an artist I want to be able to color closed surfaces with a fill tool.

As an artist I want to be able to completely undo my drawing actions easily.

As an artist I want to be able to completely redo my drawing actions easily.

As an artist I want to be able to delete the whole content of the current page and start over.

As an artist I want to be able to view previous pages so that the movement of objects on my current page is accurate.

As an artist I want to be able to hide previous pages so that once I got the movement of objects, I can concentrate on the drawing.

As an artist I want to be able to hide the foreground so I can focus on the background.

As an artist I want to be able to show the foreground to see if the composition is right.

As an artist I want to be able to view the background of my drawing to see if the composition is right.

As an artist I want to be able to hide the background of my drawing so I can focus on the foreground.

As an artist I want to be able to see all pages I've created so far.

As an artist I want to be able to insert a new page and edit it instantly.

As an artist I want to be able to edit a page instantly.

As an artist I want to be able to delete pages I don't need anymore.

As an artist I want to be able to copy some of the current pages.

As an artist I want to be able to click/scroll through the pages, and reorganize them to my liking.

As an artist I want to be able to see all pages I've created so far – to view my flipbook animation.

As an artist I want to be able to change the speed rate of individual pages.

As an artist I want to be able to add a soundtrack to my toon.

As an artist I want to be able to add sound effects.

Continue Toon – As an artist I want to be able to continue where I left off.

Toon Gallery – As an artist I want to be able to see all my toons.

As an artist I want to be able to play my toon.

As an artist I want to be able to edit my toon.

As an artist I want to be able to delete my toon.

As an artist I want to be able to make a copy of my toon.

As an artist I want to be able to publish my toon and share my artwork with others.

Online Sharing Platform – As an artist I want to be able to share and view toons online.

As an artist I want to be able to view toons created by others.

As an artist I want to be able to download toons created by others.

As an artist I want to be able to rate toons created by others.

As an artist I want to be able to comment on toons created by others.

As an artist I want to be able to receive comments related to my toon.

As an artist I want to be able to receive ratings on my toon.

As an artist I want to be able to add other toons to my favorites.

As an artist I want to be able to name my toon.

As an artist I want to be able to add a short description to my toon.

A scenario is a story about fictional users (personas) using a product or service to achieve their goals.

The following paragraphs describe Sophie's story:

Sophie is inspired. She'd like to make a small toon of her cat Tiger destroying the Christmas tree's decorations. This has been his favorite activity for years now. She takes her phone, unlocks it and starts Pocket Toon. She selects the new toon option, picks the brush, selects the color green and choses the background layer. She starts drawing a Christmas tree. She switches between colors, brush sizes and tools, in order to create a most glorious Christmas tree. There, it's done. She creates a new page, selects the foreground and starts drawing Tiger's paw. He needs to enter the scene slowly. Couple of pages later, he's finally beneath the Christmas tree. Suddenly she hears her mother calling for her. She closes the application, places her phone on the table and runs downstairs.

Sophie is back in her room. She picks up her phone, unlocks it and restarts Pocket Toon. This time she selects "Continue". She flips through her already created pages to get herself reconnected. Finally, she creates a new page and continues drawing Tiger climbing up a Christmas tree.

Sophie just finished her toon. She has been drawing for hours now, bringing her character to life with every page. She even added some music for dramatic purposes. It is a half-minute tale about her cat Tiger and his mischievous ways. As she plays the video, she notices that one of the scenes requires small adjustments. She opens the animation's "filmstrip" screen, selects the scene and corrects it. She plays the video again. It is perfect and ready for sharing with her friends.

Sophie is on the bus on her way to school. Her best friend Marie is sitting next to her. Marie asks Sophie about her thoughts on their favorite show which aired the *night before. Sophie says she forgot all about it because she was animating Tiger's Christmas activity. Marie wants to see the toon and jokes that the toon better be good since last night's episode was amazing. Sophie takes her phone out, unlocks it, opens Pocket Toon and selects the gallery. She scrolls until she sees the title "Oh, Christmas tree". She clicks on the toon and presses play. Suddenly the bus echoes with Maries' laughter.*

4.1.4 Conceptual Design

This thesis describes the conceptual model through schematics, because it is believed that schematics can capture interaction far better than words and tables [12].

To begin with, the high-level schematic, which describes a complete application interaction, is presented. Next, smaller schematics, representing the lower-level concepts, namely the Gallery, Create Page, Toon Overview, Add Sound and Online Sharing Platform, are introduced.

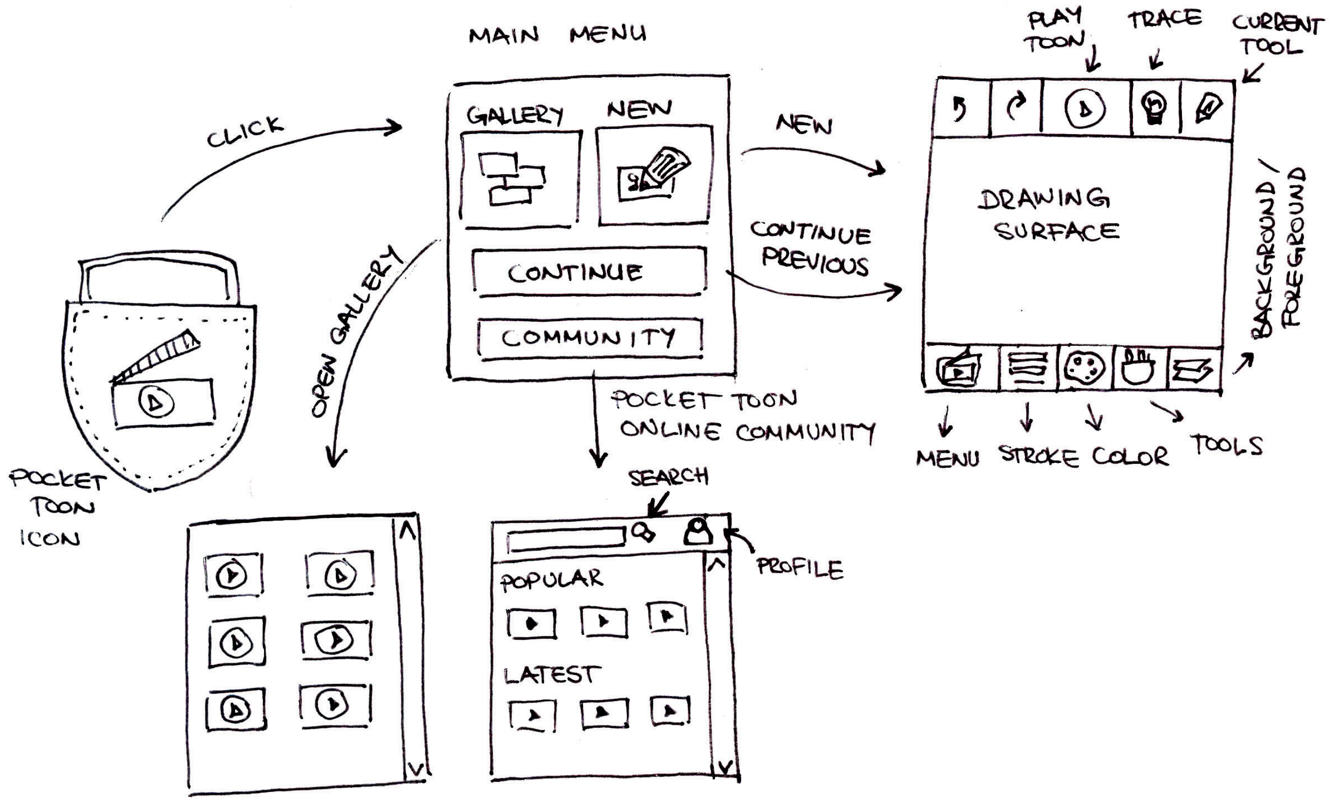


Figure 4.10: High-level schematics: Application interaction

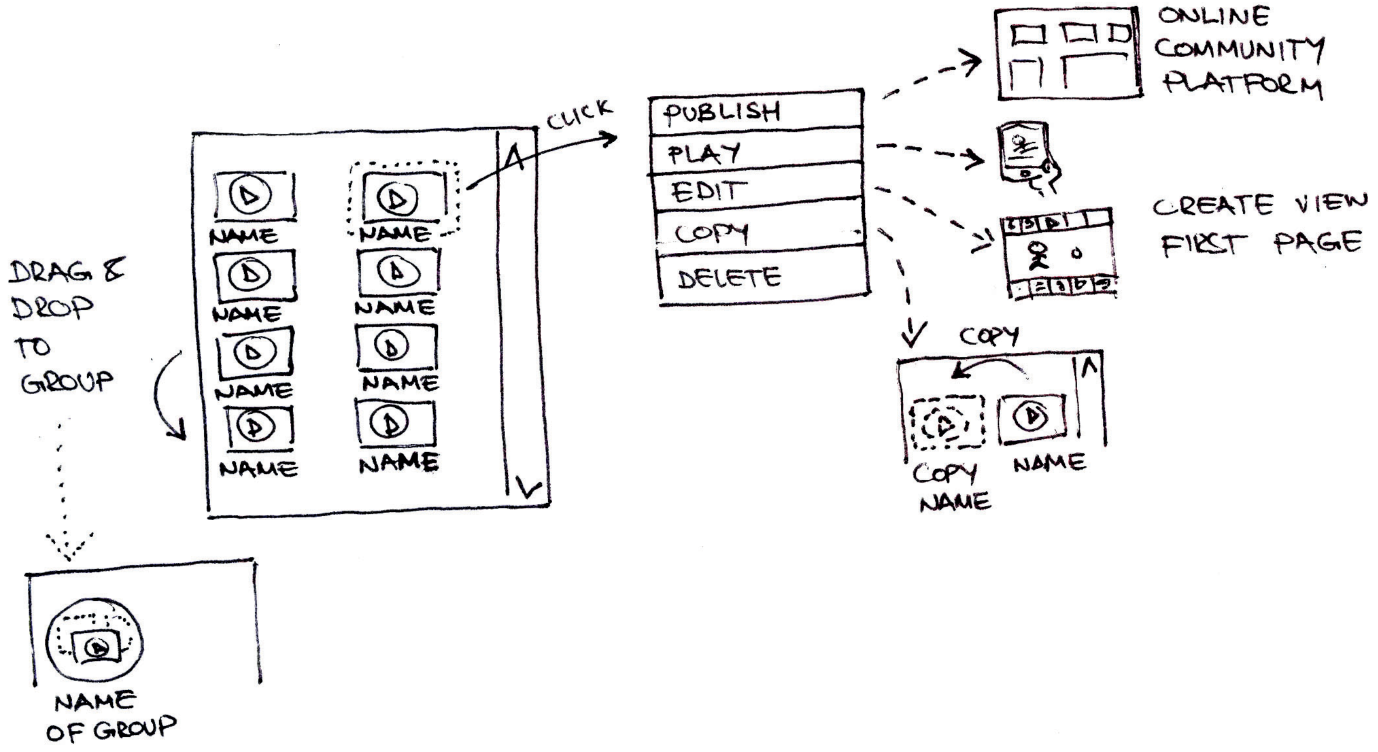


Figure 4.11: Low-level schematics: Gallery

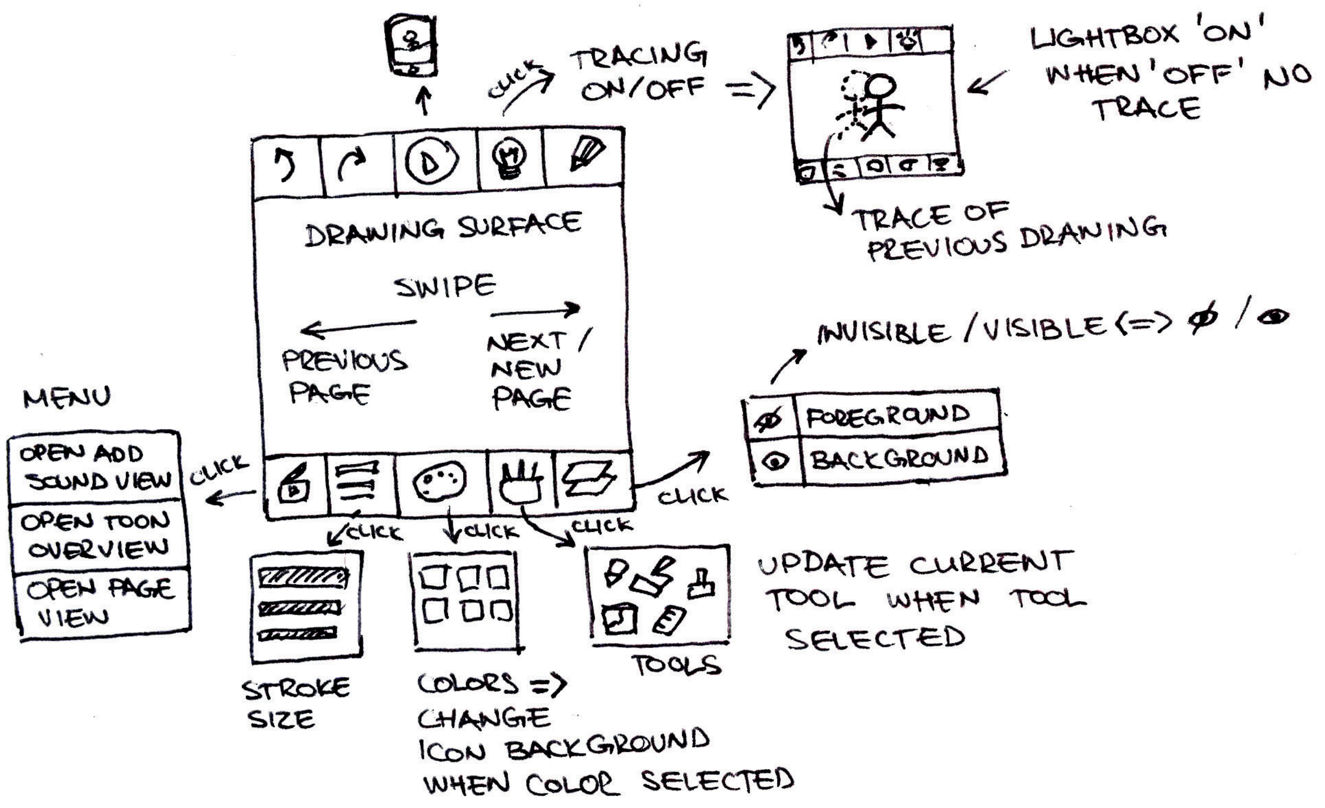


Figure 4.12: Low-level schematics: Create page view

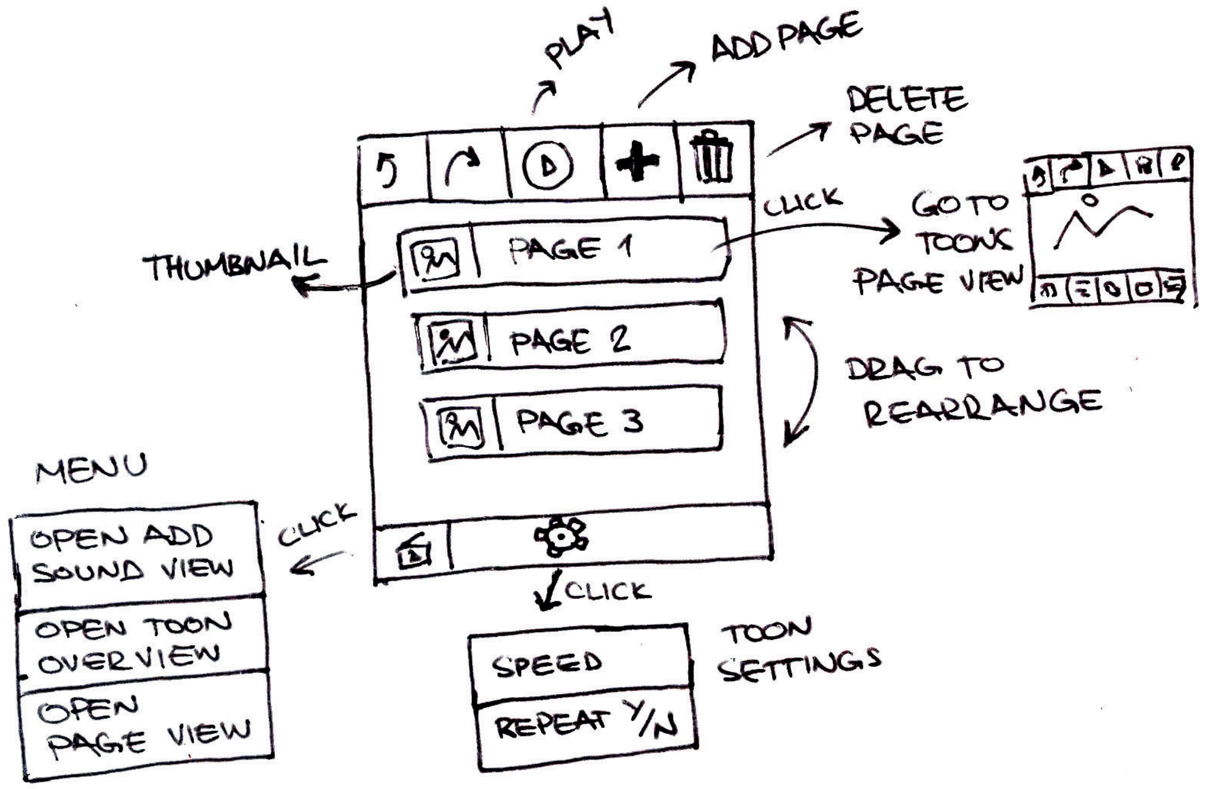


Figure 4.13: Low-Level schematics: Toon overview

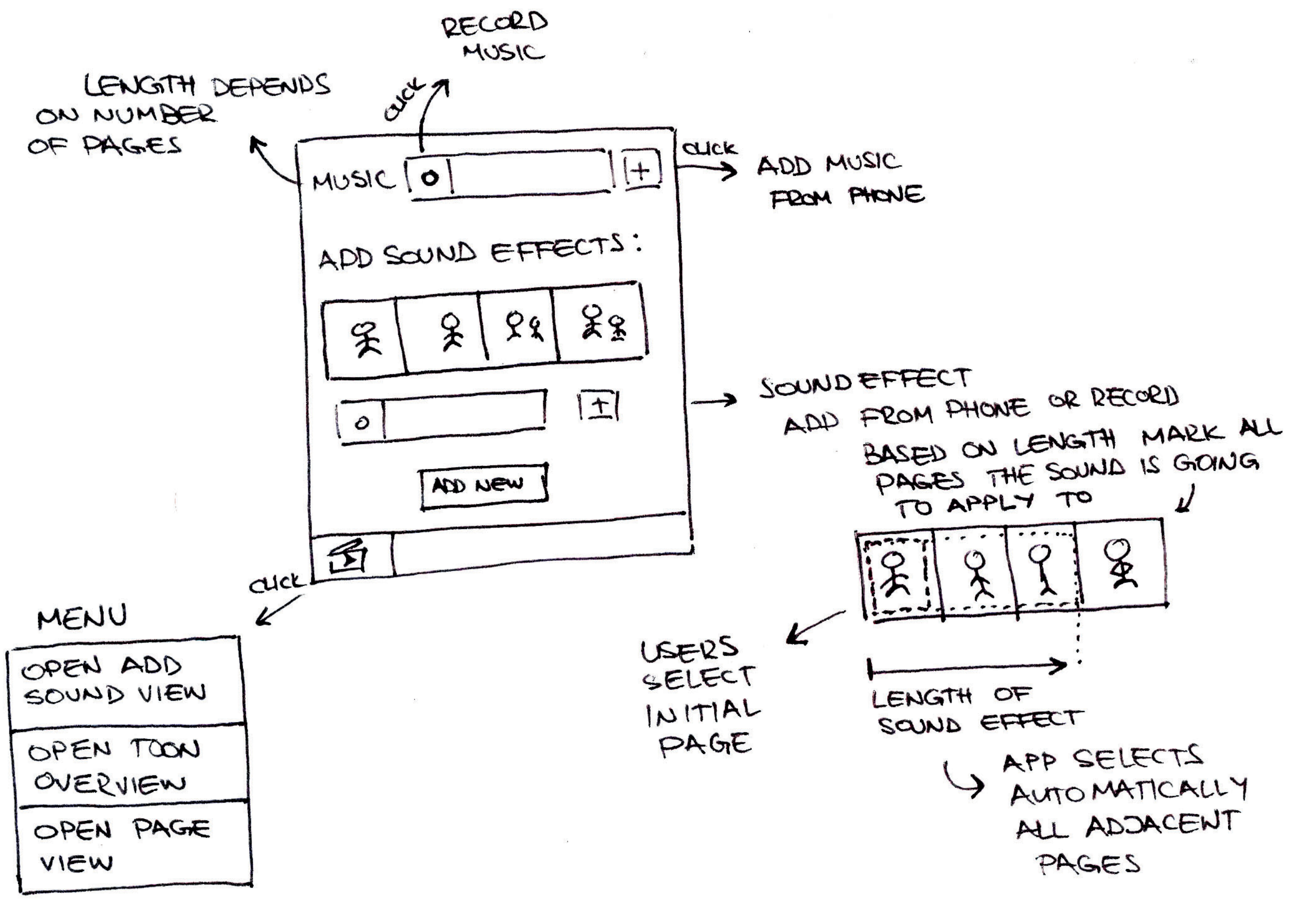


Figure 4.14: Low-Level schematics: Add sound

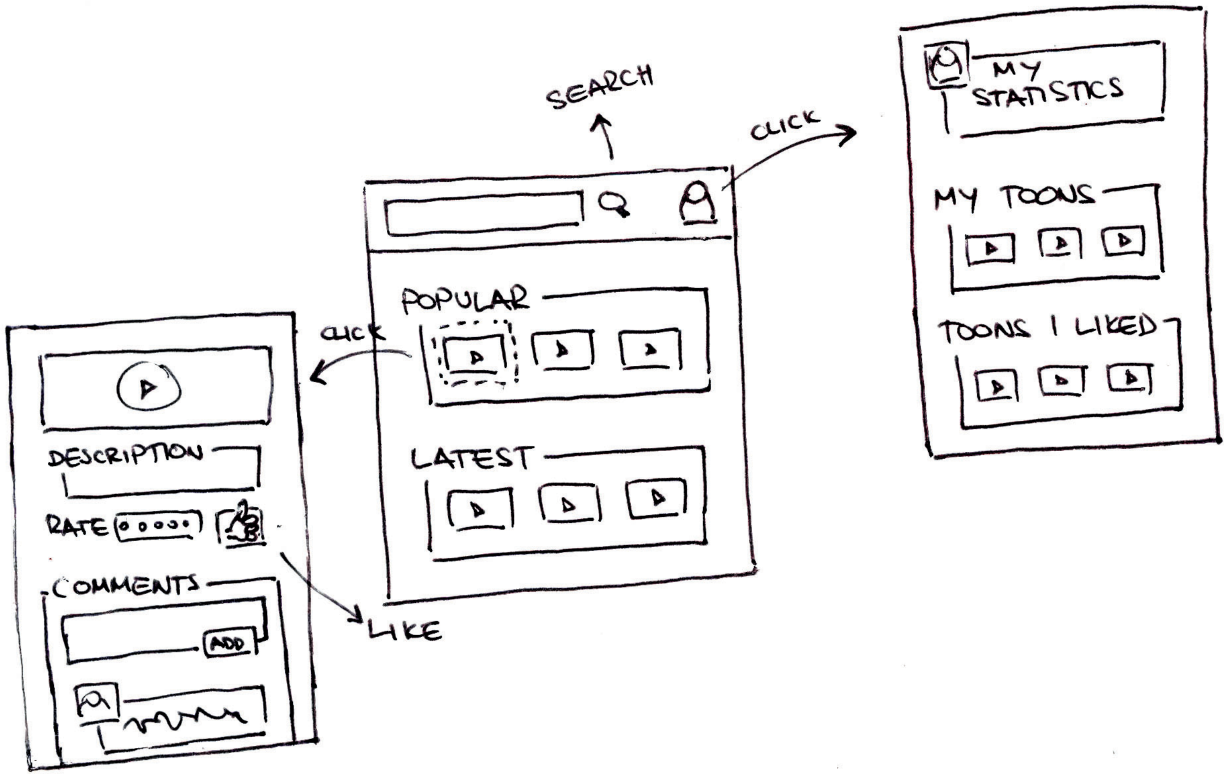


Figure 4.15: Low-Level schematics: Online sharing platform

4.1.5 Focusing on What is Important First

Up to this section, the requirements for Pocket Toon have been described. Based on those requirements, the complete conceptual design was built, visualizing interaction. The next step is materializing the app.

At this stage, it was decided to take an agile approach and apply the following steps:

1. Break big problems down into smaller ones.
2. Focus on the really important points and forget everything else.

Through user stories, the main problem has already been broken into smaller parts. With conceptual design, requirements have been grouped into standalone entities, i.e. features, based on their functionality:

1. Gallery
2. Online Sharing Platform
3. Create Page View
4. Toon Overview
5. Toon Sound

The most important feature of a digital flip book is that the users are able to create at least a flip book in the traditional sense, including the following options:

1. Turn pages and move forward or backward, similar to a regular book.
2. Draw on each page or add an image.
3. Add a new page to the flip book animation.
4. Remove pages.
5. Disassemble flip book into pages.
6. Rearrange the pages.
7. Flip the pages at different speed, thus creating the animation.

As a consequence, the Create Toon Page View and Toon Overview are the only two features that the first version of the Pocket Toon app is going to have. The other three elements are going to be implemented in later versions of the app.

4.1.6 Prototype and Evaluation

The prototype design process of this thesis included the following activities:

1. Review of Pocket Paint and Pocket Code
2. Creating the prototype

Pocket Paint provides a sophisticated basis for drawing activities. It has all the functionality Sophie wishes for in a paint editor. She can work with different tools and colors, adjust their properties, undo/redo her actions, and import photos from her gallery or camera and edit them. Beside functionality, Pocket Paint provides a simple and easy interaction. Both, functionality and design concepts are taken under consideration, aiming at benefiting from reusability and familiarity.

There were far less similarities between Pocket Toon and Pocket Code but the analysis revealed that the Pocket Code's player (interpreter) and some IDE parts of Pocket Code could be candidates for reusability.

I used the Create Page View conceptual model to build a high-fidelity, low function prototype. I built this prototype using InVision [14], an online prototyping tool, and tested it with a very small group of interns, whose age correlated with the target user group.

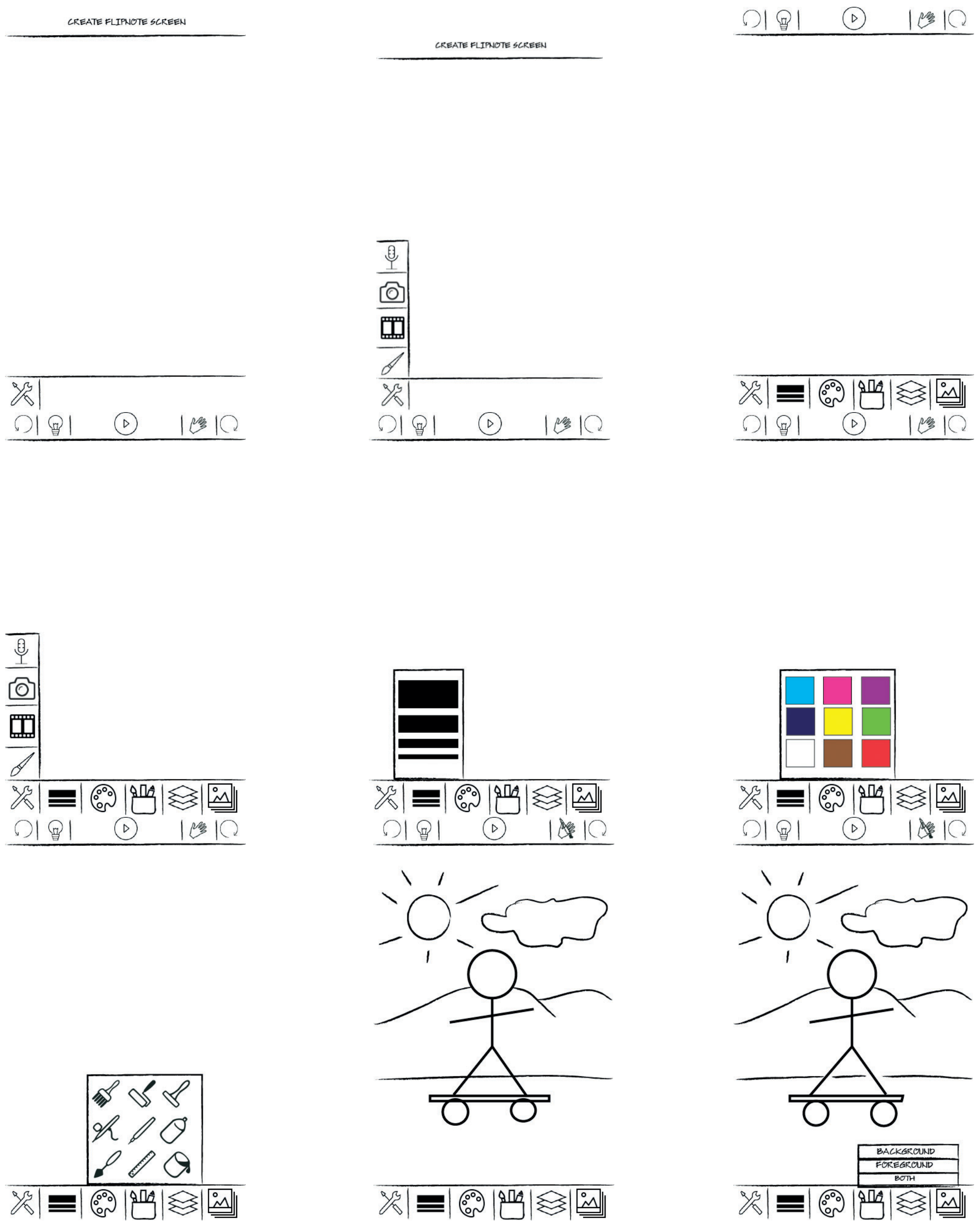
The test was conducted with a group of three interns. They were all girls between the age of 14 and 19. They were asked to perform the following tasks:

1. Go to Create Page View.
2. Select Tool.
3. Draw.
4. Show/Hide foreground.
5. Show/Hide background.
6. Copy content of the current page to all the following pages.
7. Go to next page.
8. Hide/Show tracing of previous page.
9. Play toon.
10. Go to Toon Overview.

11. Add new page.
12. Delete existing page.
13. Explain how they would rearrange pages.
14. Change settings of the toon such as speed.
15. Use Camera.

The following results were obtained through the test:

1. All three subjects found the "Copy content of the current page to all the following pages" task confusing. The icon used did not provide the necessary clarification. It only added to the existing task confusion.
2. None of them found using swipe gestures to create new pages or move forwards/backwards intuitive. They were all looking for some kind of indicator such as an arrow, three dots to continue or dog ears, suggesting that they could move to another page.
3. Two of them had difficulties turning tracing on and off, i.e. understanding the purpose of the light bulb. When I later explained the lightbox metaphor, they remarked that they would have needed something more analogous to a real light bulb. As a consequence, additional interaction has to be integrated.
4. One of the subjects suggested that I should change the Toon Overview icon, because the chosen film strip is very similar to the gallery, resulting in confusion.
5. All three maintained that the design was simple and apart from the previously mentioned problems, very intuitive.
6. Only one subject confirmed interest in using the app. The other two had no interest in using it, but found it suitable for a younger audience or art classes.


CREATE FLIPNOTE SCREEN
CREATE FLIPNOTE SCREEN
BACKGROUND
FOREGROUND
BOTH

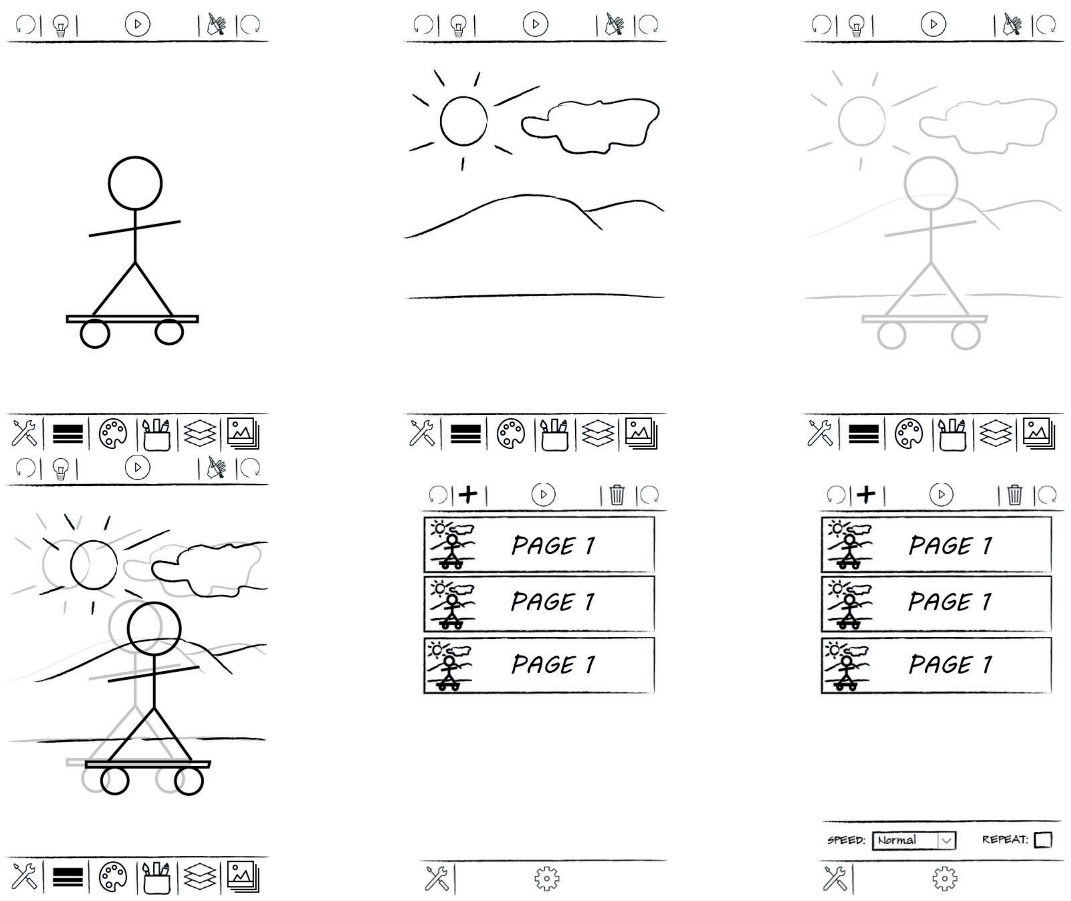


Figure 4.15: Pocket Toon: High-fidelity low functional prototype

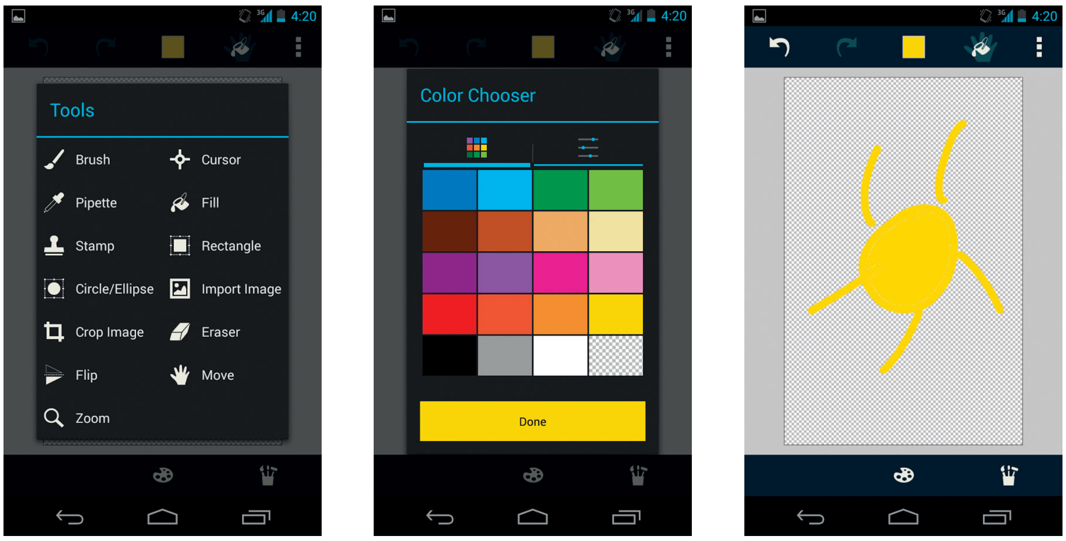


Figure 4.16: Pocket Paint interface

First design iteration

Even though I conducted a test with a very small group of interns, the result provided valuable feedback. I used this feedback in order to refine the conceptual design schematics of Create Page View and Toon Overview.

Specific problems that needed to be solved was improving the flipping through Pocket Toon pages. The interns told me that they were expecting some kind of visual indicator for flipping the pages forward or backward. The risk when solving this problem was cluttering the design with too many visual elements. So, I decided to study Google's Material design guidelines [10] and winners of the Material Design Awards [9] for possible guidance.

The solution to my "flipping pages"-problem was to use arrows which disappear as soon as users touch the drawing surface. I also decided to add an animation effect to the right arrow button, transforming it to a plus symbol as soon as the user reaches the last page. This decision was inspired by the Evernote design [7], which has been recognized for its achievements in design through the Material Design Showcase Award [9].

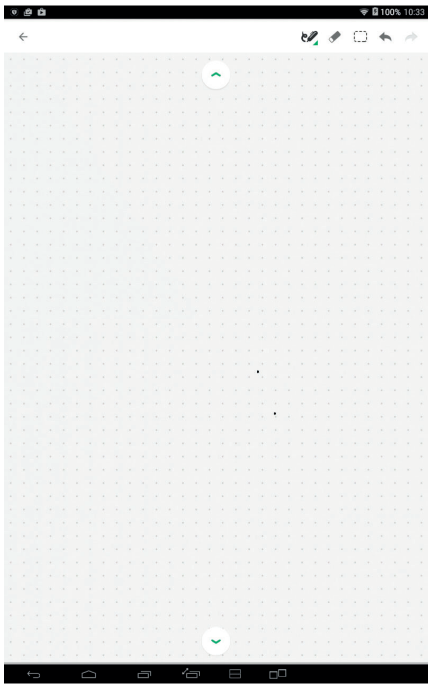


Figure 4.17: Evernote

The ”copy content of the current page to all the following pages”-functionality was removed completely. I decided that the first version would only have a single static background (similar to Autodesk Sketchbook v7 ©: Flipbook feature) and if the users want to copy the foreground to multiple pages, they would have to use the Toon Overview feature.

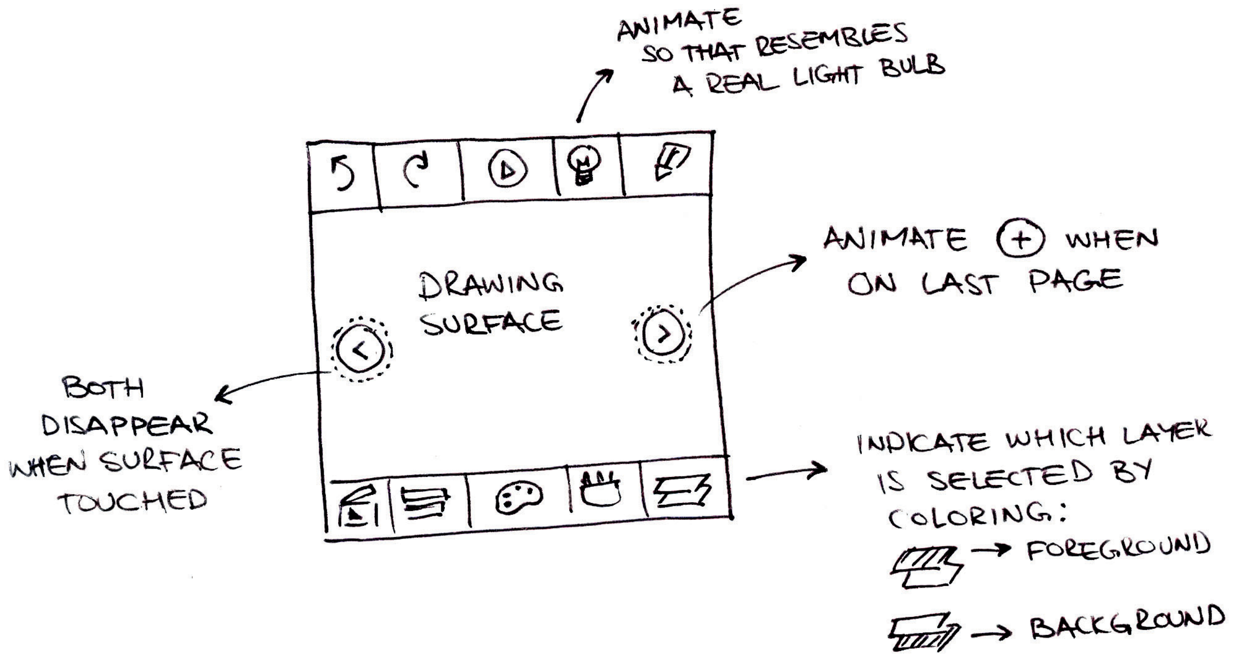

ANIMATE SO THAT RESEMBLES A REAL LIGHT BULB
DRAWING SURFACE
ANIMATE ⊕ WHEN ON LAST PAGE
BOTH DISAPPEAR WHEN SURFACE TOUCHED
INDICATE WHICH LAYER IS SELECTED BY COLORING:
→ FOREGROUND
→ BACKGROUND

Figure 4.18: Pocket Toon: Create page view redesigned

5 Implementation

As outlined in Section 4.1., the software implementation can be described as a three step process:

1. Implementation of missing feature in Pocket Paint (layers).
2. Refactoring Pocket Paint into library.
3. Implementing Pocket Toon.

Due to limitation of the thesis' scope, only the first step was completed, in other words, adding the layer feature to Pocket Paint. The refactoring, however, turned out to be a very complex task, because of the restrictions of the current implementation. Thus, the transformation into a library, which demands a clean separation between GUI and Core, could not be realized. However, in spite of the current limitations, I plan to continue contributing in a leading role to the implementation of Pocket Toon together with other members of the Catrobat team as part of the Catrobat free open source software development efforts.

Therefore the last chapter of this thesis describes the layer implementation.

5.1 Defining Layers

Layers represent a virtual stacking of transparent images, [16] where each layer represents a separate image, whose content can be modified without influencing any other layer in the stack. This is known as ”nondestructive editing”. Changes made in one layer remain isolated from other layers [16]. For instance, Sam is on vacation in South Africa. He just took a picture of the amazing wildlife and now wants to create a postcard with his personal greetings on it in order to send it to his family and friends. He opens the wildlife image with his favorite image editing software, which supports layers. The image is automatically added to the first layer (often called ”Background” layer). Sam creates another layer on top of the Background layer and adds his handwritten greetings to it. The greetings did not turn out quite right, so Sam erases them. Even though the handwritten greetings are gone, his wildlife image is still intact. This is due to the fact that the changes made by Sam only affected the top layer, the ”foreground” layer. Sam tries again. This time, his greetings are written perfectly but the position is a bit off. Sam selects the layer with his greetings, moves it around, rotates it, even scales it, until he finds a position that ”feels right”. He prints his post card. Without a layer system, it would not have been possible to isolate and modify separate elements of the image. The changes would have affected the image as whole.

Layers can have many different characteristics depending on the software solution and its implementation, the fundamental ones being [11]:

1. Opacity. Values lie between 0% (transparent) and 100% (completely opaque).
2. Holes. Allow details of underlying layers to be visible when either the image data of the higher-level or top layer has been erased, or the higher-level or top layer is smaller than the layer beneath.
3. Ability to be merged (blended) with other layers.

5.2 Pocket Paint and Layers Requirement

In order to extract implementation requirements, it is necessary to examine the current state of Pocket Paint. The current implementation of Pocket Paint provides the following features:

It supports a single image (Bitmap) only.

A Bitmap can be modified with several tools. The properties of a tool can be adjusted, for instance, stroke size, color, etc.

Pocket Paint implements the command pattern. Each command describes an interaction between a tool and an underlying Bitmap. The commands are necessary in order to implement the Undo and Redo functionality.

The undo and redo commands are stored on two separate command stacks, which are the property of the CommandManager. Every time users undo their action, Pocket Paint moves a command from the Undo stack to the Redo stack, wipes the Bitmap clean, and reuses the remaining Undo stack commands to repaint the image. If the users decide to redo a previously undone action, the action is moved from the Redo stack to the Undo stack. Basically, Undo and Redo stacks mirror each other.

The users can save the modified image to their device's file system.

The enhanced version of Pocket Paint modifies the basic functionality and adds extra layer operations (add, remove, merge etc), namely:

1. Tools can only modify the Bitmap of the currently active layer. The only exception to this rule is a crop tool, which resizes the whole image.
2. The undo and redo actions apply to the layer(s) in which the respective actions were taken. E.g., undoing a merge of two layers regenerates the two original layers.
3. Support for the following layer operations:
   a) Add layer.
   b) Delete layer.
   c) Merge layers.
   d) Change opacity of the layer.

   e) Lock the layer.
   f) Hide/show the layer.
   g) Name layer.
   h) Copy layer.
   i) Move layer up or down in the list of layers, changing visibility precedence.
   j) Move layer to the top or bottom in the list of layers.

4. The users need to be able to save the image. Two possible solutions are:
   a) Merging all the layers (flattening the image) and saving it as png.
   b) Maintaining layer structure.

5.3 Implementation

The implementation of the layers has been initiated three years ago by several members of the Catrobat team but was far from being completed and had stalled for several months before I started working on it.

Figure 5.1 shows the layer dialog, which provides the layer interaction. The layers are stacked in a list. The upper layers have visibility precedence over lower layers. The users can either add, delete, merge, or rename, layers, change visibility, lock, as well as change layer opacity. They can also move layers up and down the stack, thus changing their visibility precedence.

Therefore, the Layer Dialog has been implemented as follows:

Each layer object holds information about its opacity, name, visibility, lock, and unique ID, in order to differentiate between layers and drawing surface.

In order to create a drawing surface in Android, two components are needed [4]:

- Canvas – which is an interface to the actual surface upon which the graphics will be drawn – it holds the ”draw” calls [5].

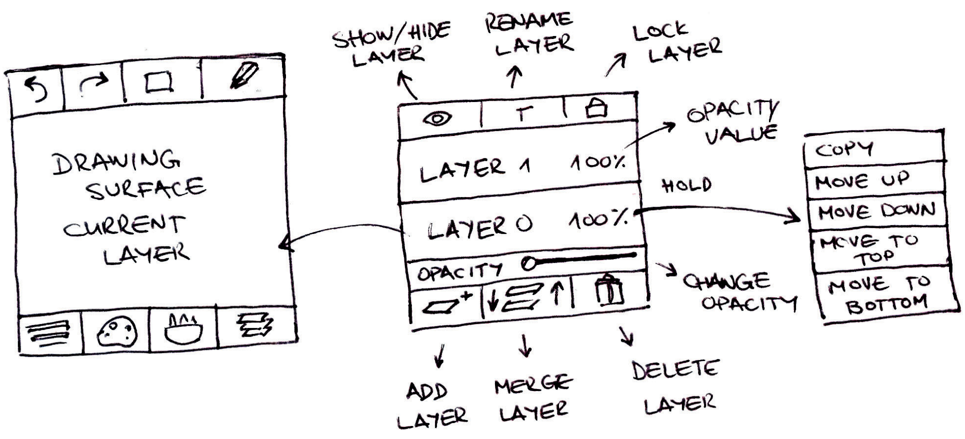


Figure 5.1: Pocket Paint: Layer Dialog

- Underlying Bitmap – which holds the pixels and upon which the drawing is performed [5].

Therefore, the layer object needs to contain Canvas and its underlying Bitmap.

The stack with layer objects is implemented using GridView. GridView displays layers in a two-dimensional, scrollable grid. In order to perform a stack manipulation (adding, removing, rearranging), the LayerAdapter, which extends the BaseAdapter, was implemented[1].

Once the layer basis has been established, the implementation of the remaining features, which have been defined in Section 5.2., can begin.

[1]Notice that merge has not been mentioned, because merge is special version of add – the properties of two selected layers are merged into new layer, after which selected layers are removed. The new is then added to LayerAdapter.

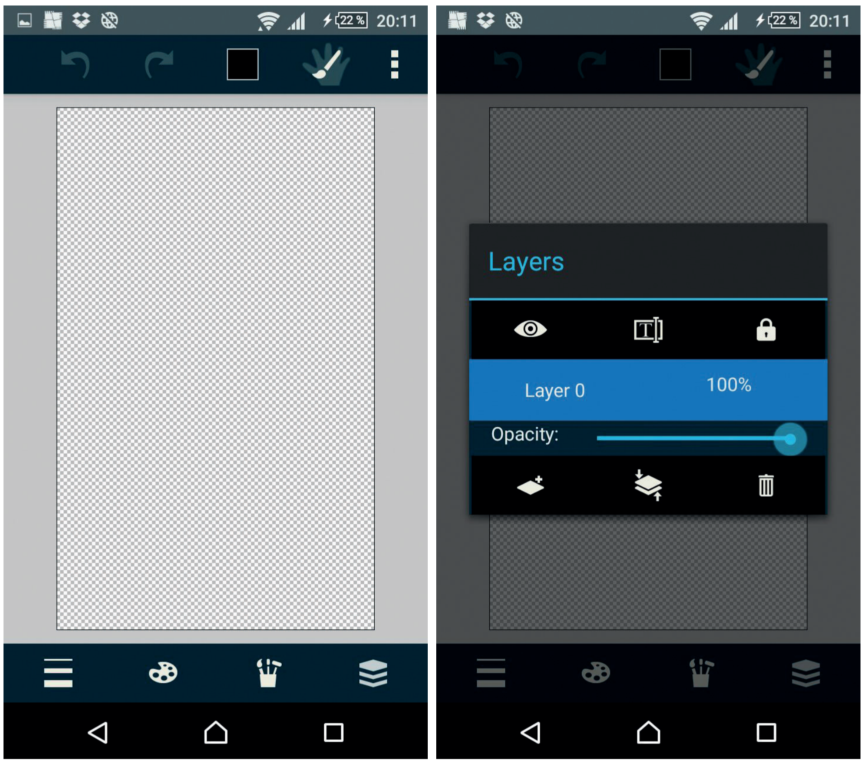


Figure 5.2: Pocket Paint: Layer Dialog Implemented

Tools can modify only the Bitmap of currently active layer

In order to adapt the tools to the layer implementation, it is important to gain basic understanding of how the ”tool action”, for instance, drawing a line, is executed in Pocket Paint.

The ”tool action” consists of two activities:

The first activity is executed on the drawing surface and is visible to the user. The surface tracks finger motion of the user, for example, when the brush tool is being used, the users will see the ”paint” trace on the screen, which corresponds to their finger movement, without changing the underlying Bitmap.

The changes on the Bitmap happen during the second activity. Section 5.2.

illustrated that interaction between the Bitmap and the tool is described by a command. The "tool action" begins as soon as users touch the drawing surface and ends as soon as they lift their finger from it. Every time users perform a "tool action", the tool generates a new command, which is then executed on the underlying Bitmap. Therefore, the commands are responsible for Bitmap modifications.

The same activities apply to layers. The only difference is that the second activity modifies the Bitmap of the currently selected layer. Nevertheless, the drawing surface class, all tools and all commands classes needed to be modified

The undo and redo actions apply only to the corresponding layer(s)

The modification of undo and redo functionality provided a greater challenge because it required a complete redesign of the CommandManager, which is responsible for undo, redo and the execution of commands.

In Section 5.2., undo/redo operations were described in detail. The basic principle of mirroring stays the same, however, the complexity of the task is much higher.

Since the Pocket Paint implements the command pattern, the first step is to identify new commands, i.e. extending the existing set.

Figures 5.4, 5.5 and 5.6 revealed that in order to implement undo/redo functionality it is necessary to differentiate between the layer commands and the Bitmap command (the current command implementation in Pocket Paint), as well as between different layer command types. The following design decisions were made and implemented:

The Bitmap command represents a "tool action" which is associated to some layer, for example, the. DrawTool creates a PathCommand which draws a line on the Bitmap of a currently active layer.

LayerBitmapCommands act as a Bitmap command manager for the corresponding layer. This manager contains the associated layer, the undo and redo stack and it manages all the Bitmap commands associated with the layer.(Listing: 5.1).

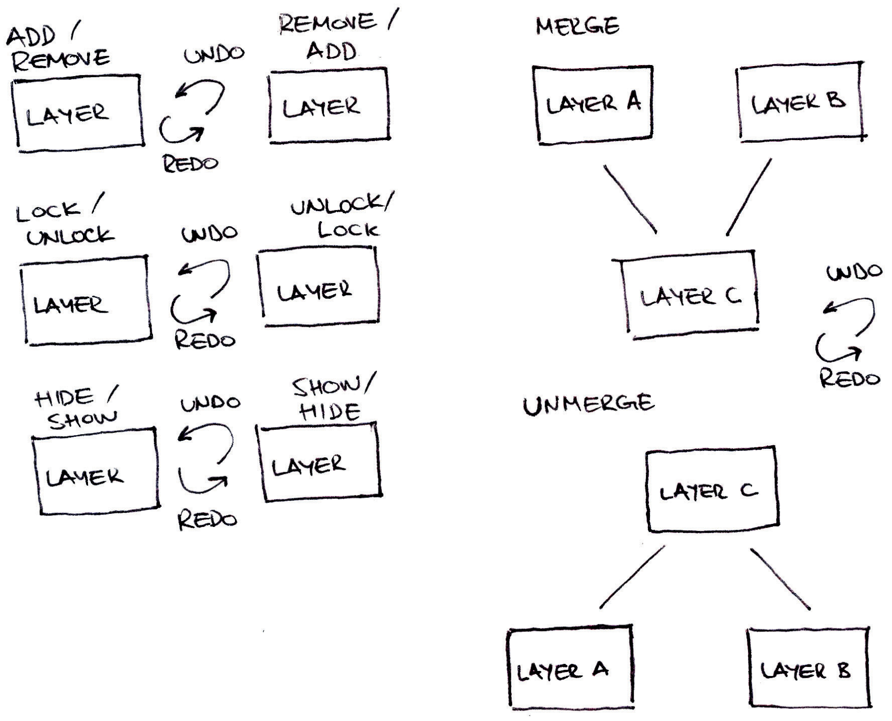


Figure 5.3: Layer undo/redo

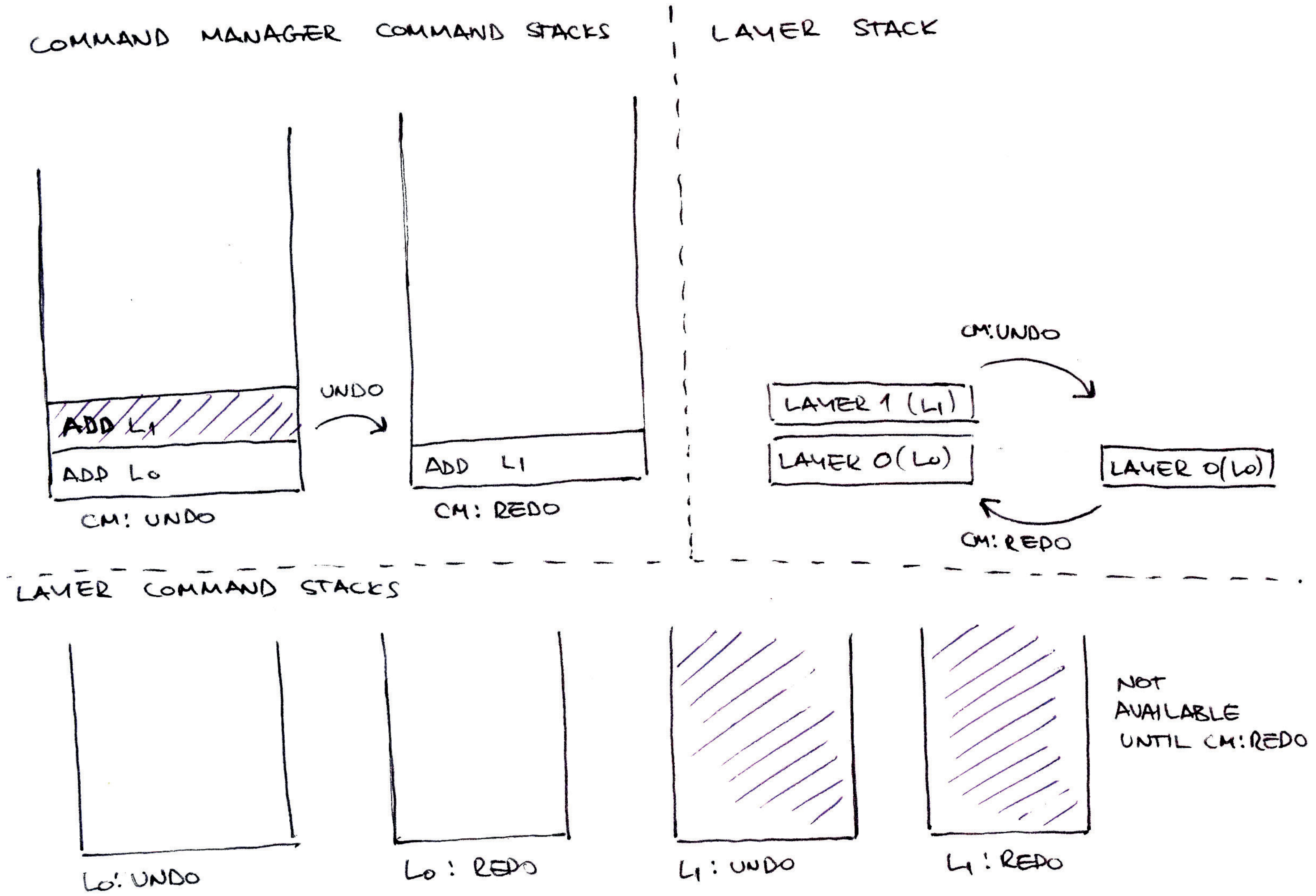


Figure 5.4: Layer implementation: CommandManager add layer.

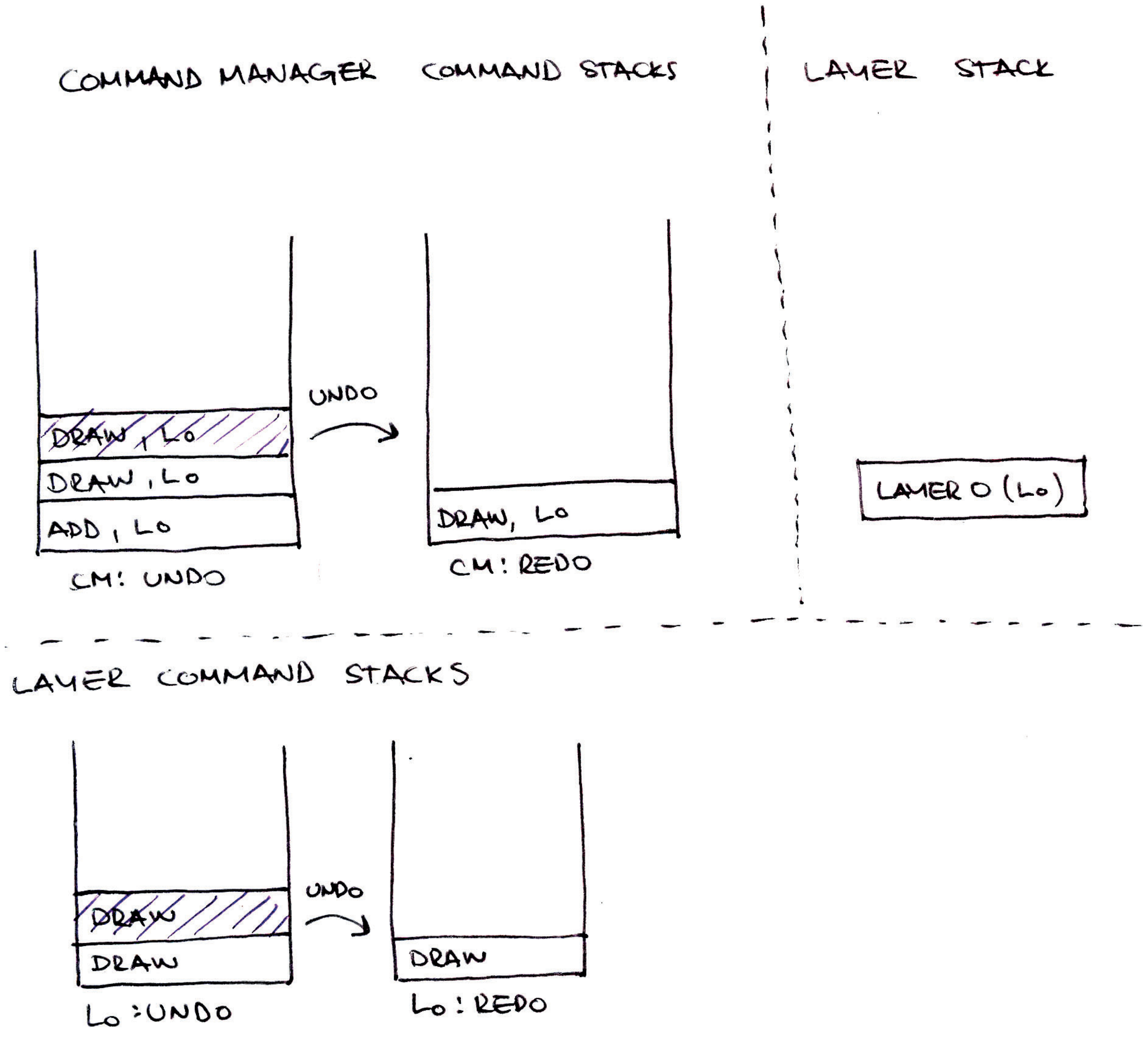


Figure 5.5: Layer implementation: CommandManager layer bitmap command.

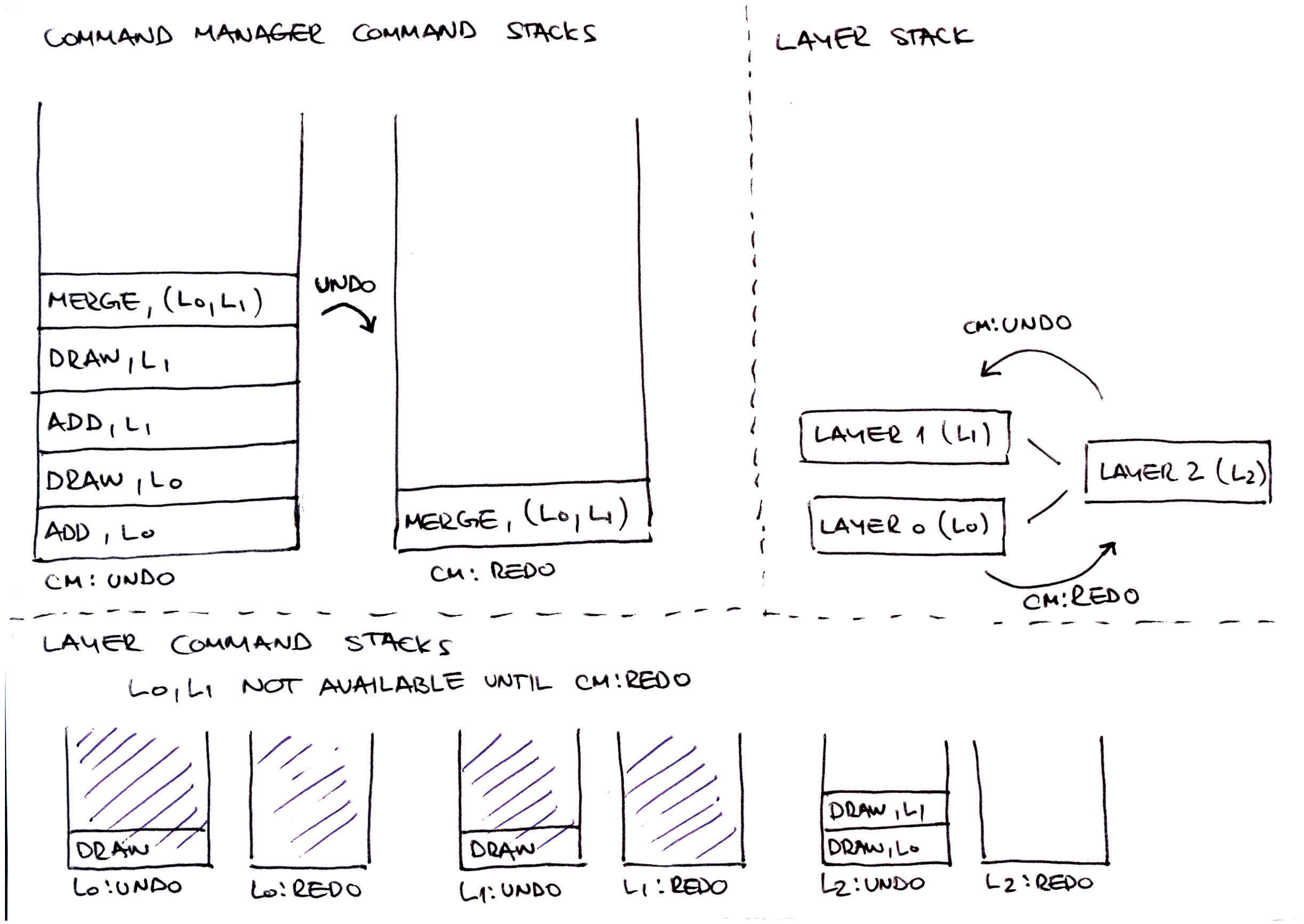


Figure 5.6: Layer implementation: CommandManager merge layers.

LayerCommand describes layer operations such as add, remove, merge and rename. Its initialization depends on the operation with which it has been associated with (Listing: 5.2):

- ADD/REMOVE – contains a single layer to which add/remove operation applies.
- MERGE/UNMERGE – contains merged layers, list of layers to be merged and their associated LayerBitmapCommands.
- RENAME – contains the layer to which the change has been applied and its former name.

The CommandManager (Listing: 5.3) has been implemented as follows:

The command of the undo/redo stack has been implemented as `Pair<CommandType,CommandLayer>`.

The undo and redo stack have been implemented as linked lists of `Pair<CommandType,CommandLayer>`.

Beside the undo and redo stack, the CommandManager contains a list of LayerBitmapCommands, which represents the list of all currently active layers i.e. correlates to the layers in the Layer Dialog. The reason for this is that every time a remove or merge operation is undone, the layers that have been previously removed from the system need to be restored in their original state (prior to removal or merge). There is also a benefit of such a design: such data structure could be used in order to save images with layers. However, this will be discussed later in this section.

The CommandManager supports two different workflows:

1. Commit command workflow – command is added to command stack.
2. undo/redo workflow, where undo and redo are mirroring operations:
   a) Undo – command is removed from undo stack and pushed on redo stack.
   b) Redo – command is removed from redo stack and pushed back to undo stack.

As a consequence of the support of both workflows, each of these workflows has to be described with its own set of methods.

```
/**
 * Describes layer commands responsible for drawing. These
    commands are performed on layer's bitmap.
 */
public interface LayerBitmapCommands
{
    /**
     * Retrieves layer assigned to command manager.
     * @return Layer which has been assigned to command manager.
     */
    Layer getLayer();

    /**
     * Commits command for assigned layer.
     * @param command which has been performed on layer.
     */
    void commitCommandToLayer(Command command);

    /**
     * Retrieves all the commands performed on layers bitmap.
     * @return layer bitmap commands.
     */
    List<Command> getLayerCommands();

    /**
     * Copies layer commands to current LayerBitmapCommands.
     * @param commands commands to be copied.
     */
    void copyLayerCommands(List<Command> commands);

    /**
     * Undo drawing command for assigned layer.
     */
    void undo();

    /**
     * Redo drawing command for assigned layer.
     */
    void redo();
}

/**
 * Contains all the commands that are to be executed on the layer'
    s bitmap.
 */
public class LayerBitmapCommandsImpl implements
    LayerBitmapCommands {
     private Layer mLayer;

     private LinkedList<Command> mCommandList;
     private LinkedList<Command> mUndoCommandList;


     public LayerBitmapCommandsImpl(LayerCommand layerCommand)
     {
          mLayer = layerCommand.getLayer();
          mCommandList = new LinkedList<Command>();
          mUndoCommandList = new LinkedList<Command>();
     }


     @Override
     public Layer getLayer() {
          return mLayer;
     }

     @Override
     public void commitCommandToLayer(Command command)
     {
          synchronized (mCommandList)
          {
               mUndoCommandList.clear();
               mCommandList.addLast(command);
               synchronized (mLayer.getLayerCanvas())
               {
                    command.run(mLayer.getLayerCanvas(), mLayer.
                        getBitmap());
               }

               PaintroidApplication.currentTool.resetInternalState(
                   Tool.StateChange.RESET_INTERNAL_STATE);
          }
     }

     @Override
     public List<Command> getLayerCommands() {
          return mCommandList;
     }
```

```
@Override
public void copyLayerCommands(List<Command> commands) {
   for(Command command : commands)
   {
       mCommandList.add(command);
   }
}

@Override
public synchronized void undo()
{
    synchronized (mCommandList)
    {
        Command command = mCommandList.removeLast();
        mUndoCommandList.addFirst(command);
        executeAllCommandsOnLayerCanvas();
    }
}

@Override
public synchronized void redo() {
    synchronized (mUndoCommandList) {

        if (mUndoCommandList.size() != 0) {
            Command command = mUndoCommandList.removeFirst();
            mCommandList.addLast(command);

            synchronized (mLayer.getLayerCanvas())
            {
                command.run(mLayer.getLayerCanvas(), mLayer.
                    getBitmap());
            }

            PaintroidApplication.currentTool.
                resetInternalState(Tool.StateChange.
                RESET_INTERNAL_STATE);
        }
    }
}

private void executeAllCommandsOnLayerCanvas()
{
    synchronized (mLayer.getLayerCanvas())
```

```
        {
            clearLayerBitmap();
            for (Command command : mCommandList)
            {
                command.run(mLayer.getLayerCanvas(), mLayer.
                    getBitmap());
            }

            PaintroidApplication.currentTool.resetInternalState(
                Tool.StateChange.RESET_INTERNAL_STATE);
        }
    }

    private void clearLayerBitmap()
    {
        synchronized (mLayer.getLayerCanvas())
        {
            mLayer.getBitmap().eraseColor(Color.TRANSPARENT);
        }
    }
}
```

Listing 5.1: LayerBitmapCommands implementation

```
/**
 * Describes Layer command. It can contain either simple layer on
     which some operation is being
 * performed, or list of merged layers ids, along with the new
     layer created by merge and
 * merged layers bitmap command managers.
 */
public class LayerCommand
{
    private Layer mLayer;
    private ArrayList<Integer> mListOfMergedLayerIds;
    private ArrayList<LayerBitmapCommands> mLayersBitmapCommands;

    private String mLayerNameHolder;

    public LayerCommand(Layer layer)
    {
        mLayer = layer;
    }
```

```
public LayerCommand(Layer newLayer, ArrayList<Integer>
    listOfMergedLayerIds)
{
     mLayer = newLayer;
     mListOfMergedLayerIds = listOfMergedLayerIds;
     mLayersBitmapCommands = new ArrayList<LayerBitmapCommands
         >(mListOfMergedLayerIds.size());
}

public LayerCommand(Layer layer, String layerNameHolder)
{
     this.mLayer = layer;
     this.mLayerNameHolder = layerNameHolder;
}

public Layer getLayer() {
     return mLayer;
}

public ArrayList<Integer> getLayersToMerge()
{
     return mListOfMergedLayerIds;
}

public void setLayersBitmapCommands(ArrayList<
    LayerBitmapCommands> layersBitmapCommandManagerList)
{
     this.mLayersBitmapCommands =
         layersBitmapCommandManagerList;
}

public ArrayList<LayerBitmapCommands> getLayersBitmapCommands
    () {
     return mLayersBitmapCommands;
}

public String getLayerNameHolder() {
     return mLayerNameHolder;
}

public void setLayerNameHolder(String layerNameHolder) {
     this.mLayerNameHolder = layerNameHolder;
}
```

```
}
```

Listing 5.2: LayerCommand implementation

```
/**
 * Describes undo/redo command manager responsible for
    applications layer management.
 */
public interface CommandManager
{

    /**
     * Adds the new command (draw path, erase, draw shape) to
        corresponding layer.
     * @param bitmapCommand command to commit to layer bitmap.
     * @param layerCommand contains layer to which command should
        be commited.
     */
    void commitCommandToLayer(LayerCommand layerCommand, Command
        bitmapCommand);

    /**
     * Adds new layer to application.
     * @param layerCommand contains layer to add.
     */
    void commitAddLayerCommand(LayerCommand layerCommand);

    /**
     * Removes corresponding layer from application.
     * @param layerCommand contains layer to remove.
     */
    void commitRemoveLayerCommand(LayerCommand layerCommand);

    /**
     * Merges two layers.
     * @param layerCommand contains layer to be merged.
     */
    void commitMergeLayerCommand(LayerCommand layerCommand);

    /**
     * Changes visibility of corresponding layer.
     * @param layerCommand contains layer which visibility should
        be changed.
     */
    void commitLayerVisibilityCommand(LayerCommand layerCommand);

    /**
     * Locks the corresponding layer.
     * @param layerCommand contains layer which should be (un)
         locked.
     */
    void commitLayerLockCommand(LayerCommand layerCommand);

    /**
     * Renames corresponding layer.
     * @param layerCommand contains layer to rename.
     */
    void commitRenameLayerCommand(LayerCommand layerCommand);

    /**
     * Undo last command applied to specific layer.
     */
    void undo();

    /**
     * Redo last command applied to specific layer.
     */
    void redo();

    /**
     * Clears manager command lists.
     */
    void resetAndClear();
}


public class CommandManagerImplementation implements
    CommandManager, Observer
{
    private static final int MAX_COMMANDS = 512;
    private static final int INIT_APP_LAYER_COUNT = 1;

    enum CommandType {COMMIT_LAYER_BITMAP_COMMAND
         ,ADD_LAYER, REMOVE_LAYER
         ,MERGE_LAYERS
         ,CHANGE_LAYER_VISIBILITY
         ,LOCK_LAYER
         ,RENAME_LAYER}
```

```
enum Action {UNDO, REDO}

private LinkedList<Pair<CommandType, LayerCommand>>
    mLayerCommandList;
private LinkedList<Pair<CommandType, LayerCommand>>
    mLayerUndoCommandList;
private ArrayList<LayerBitmapCommands>
    mLayerBitmapCommandsList;

private RefreshLayerDialogEventListener
    mRefreshLayerDialogListener;
private RedrawSurfaceViewEventListener
    mRedrawSurfaceViewListener;
private UpdateTopBarEventListener mUpdateTopBarListener;
private ArrayList<ChangeActiveLayerEventListener>
    mChangeActiveLayerListener;
private LayerEventListener mLayerEventListener;

public CommandManagerImplementation()
{
     mLayerCommandList = new LinkedList<Pair<CommandType,
         LayerCommand>>();
     mLayerUndoCommandList = new LinkedList<Pair<CommandType,
         LayerCommand>>();
     mLayerBitmapCommandsList = new ArrayList<
         LayerBitmapCommands>();
}

public void setRefreshLayerDialogListener(
    RefreshLayerDialogEventListener listener)
{
     mRefreshLayerDialogListener = listener;
}

public void setRedrawSurfaceViewListener(
    RedrawSurfaceViewEventListener listener)
{
     mRedrawSurfaceViewListener = listener;
}

public void setUpdateTopBarListener(UpdateTopBarEventListener
    listener)
```

```
{
    mUpdateTopBarListener = listener;
}

public void addChangeActiveLayerListener(
   ChangeActiveLayerEventListener listener)
{
    if(mChangeActiveLayerListener == null)
    {
        mChangeActiveLayerListener = new ArrayList<
            ChangeActiveLayerEventListener>();
    }

    mChangeActiveLayerListener.add(listener);
}

public void setLayerEventListener(LayerEventListener listener)
{
    mLayerEventListener = listener;
}

@Override
public void commitCommandToLayer(LayerCommand layerCommand,
   Command bitmapCommand)
{
    synchronized (mLayerCommandList)
    {
        clearUndoCommandList();
        enableUndo(true);

        ArrayList<LayerBitmapCommands> result =
            layerIdToOneElementBitmapCommandList(layerCommand.
            getLayer().getLayerID());
        result.get(0).commitCommandToLayer(bitmapCommand);
        layerCommand.setLayersBitmapCommands(result);

        mLayerCommandList.addLast(createLayerCommand(
            CommandType.COMMIT_LAYER_BITMAP_COMMAND,
            layerCommand));
    }

    drawingSurfaceRedraw();
}
```

```
@Override
public void commitAddLayerCommand(LayerCommand layerCommand)
{
    synchronized (mLayerCommandList)
    {
        clearUndoCommandList();

        LayerBitmapCommands bitmapCommand = new
            LayerBitmapCommandsImpl(layerCommand);
        layerCommand.setLayersBitmapCommands(
            layerBitmapCommandToOneElementList(bitmapCommand));

        mLayerBitmapCommandsList.add(bitmapCommand);
        mLayerCommandList.addLast(createLayerCommand(
            CommandType.ADD_LAYER, layerCommand));

        if(mLayerCommandList.size() > INIT_APP_LAYER_COUNT)
        {
             enableUndo(true);
        }
    }

    drawingSurfaceRedraw();
}

@Override
public void commitRemoveLayerCommand(LayerCommand layerCommand
    )
{
    synchronized (mLayerCommandList)
    {
        clearUndoCommandList();
        enableUndo(true);

        ArrayList<LayerBitmapCommands> result =
            layerIdToOneElementBitmapCommandList(layerCommand.
            getLayer().getLayerID());
        layerCommand.setLayersBitmapCommands(
            layerIdToOneElementBitmapCommandList(layerCommand.
            getLayer().getLayerID()));

        mLayerBitmapCommandsList.remove(result.get(0));
        mLayerCommandList.addLast(createLayerCommand(
            CommandType.REMOVE_LAYER, layerCommand));
    }

    drawingSurfaceRedraw();
}

@Override
public void commitMergeLayerCommand(LayerCommand layerCommand)
{
    synchronized (mLayerCommandList)
    {
        clearUndoCommandList();
        enableUndo(true);

        ArrayList<LayerBitmapCommands> result =
            getLayerBitmapCommands(layerCommand.
            getLayersToMerge());
        layerCommand.setLayersBitmapCommands(result);

        LayerBitmapCommands bitmapCommand = new
            LayerBitmapCommandsImpl(layerCommand);
        for (LayerBitmapCommands manager: result)
        {
            bitmapCommand.copyLayerCommands(manager.
                getLayerCommands());
            mLayerBitmapCommandsList.remove(manager);
        }

        mLayerBitmapCommandsList.add(bitmapCommand);
        mLayerCommandList.addLast(createLayerCommand(
            CommandType.MERGE_LAYERS, layerCommand));
    }

    drawingSurfaceRedraw();
}


@Override
public void commitLayerVisibilityCommand(LayerCommand
    layerCommand)
{
    synchronized (mLayerCommandList)
    {
        clearUndoCommandList();
        enableUndo(true);
```

```
            mLayerCommandList.addLast(createLayerCommand(
                CommandType.CHANGE_LAYER_VISIBILITY, layerCommand))
                ;
        }

        drawingSurfaceRedraw();
    }

    @Override
    public void commitLayerLockCommand(LayerCommand layerCommand)
    {
        synchronized (mLayerCommandList)
        {
            clearUndoCommandList();
            enableUndo(true);

            mLayerCommandList.addLast(createLayerCommand(
                CommandType.LOCK_LAYER, layerCommand));
        }
    }

    @Override
    public void commitRenameLayerCommand(LayerCommand layerCommand
        )
    {
        synchronized (mLayerCommandList)
        {
            clearUndoCommandList();
            enableUndo(true);

            mLayerCommandList.addLast(createLayerCommand(
                CommandType.RENAME_LAYER, layerCommand));
        }
    }

    private ArrayList<LayerBitmapCommands>
        layerBitmapCommandToOneElementList (LayerBitmapCommands
        command)
    {
        ArrayList<LayerBitmapCommands> result = new ArrayList<
            LayerBitmapCommands>(1);
        result.add(command);
        return result;
```

```
}

private ArrayList<LayerBitmapCommands>
    layerIdToOneElementBitmapCommandList(int layerId)
{
     ArrayList<Integer> ids = new ArrayList<Integer>(1);
     ids.add(layerId);
     return getLayerBitmapCommands(ids);
}

private ArrayList<LayerBitmapCommands> getLayerBitmapCommands(
    ArrayList<Integer> layerIds)
{
     synchronized (mLayerBitmapCommandsList)
     {
          ArrayList<LayerBitmapCommands> result = new ArrayList<
              LayerBitmapCommands>();

          for (LayerBitmapCommands layerBitmapCommands :
              mLayerBitmapCommandsList)
          {
               for(int id : layerIds)
               {
                    if (layerBitmapCommands.getLayer().getLayerID
                        () == id) {
                         result.add(layerBitmapCommands);
                    }
               }
          }

          return result;
     }
}

private Pair<CommandType, LayerCommand> createLayerCommand(
    CommandType operation, LayerCommand layerCommand)
{
     return new Pair<CommandType, LayerCommand>(operation,
         layerCommand);
}

@Override
public synchronized void resetAndClear()
{
        mLayerCommandList.clear();
        mLayerUndoCommandList.clear();
        mLayerBitmapCommandsList.clear();
        enableRedo(false);
        enableUndo(false);
    }

    @Override
    public void undo()
    {
        synchronized (mLayerCommandList)
        {
            if (mLayerCommandList.size() > INIT_APP_LAYER_COUNT)
            {
                Pair<CommandType, LayerCommand> command =
                    mLayerCommandList.removeLast();
                mLayerUndoCommandList.addFirst(command);
                processCommand(command, Action.UNDO);
                enableRedo(true);

                if(mLayerCommandList.size() ==
                    INIT_APP_LAYER_COUNT)
                {
                     onFirstCommandReached();
                }
            }
        }
    }

    private void onFirstCommandReached()
    {
        changeActiveLayer(mLayerCommandList.get(0).second.getLayer
            ());
        enableUndo(false);
    }

    @Override
    public void redo()
    {
        synchronized (mLayerUndoCommandList)
        {
            if (mLayerUndoCommandList.size() != 0)
            {
                enableUndo(true);
                Pair<CommandType, LayerCommand> command =
                    mLayerUndoCommandList.removeFirst();
                mLayerCommandList.addLast(command);
                processCommand(command, Action.REDO);
                if(mLayerUndoCommandList.size() == 0)
                {
                    enableRedo(false);
                }
            }
        }
    }

    private void clearUndoCommandList()
    {
        synchronized (mLayerCommandList)
        {
            enableRedo(false);
            mLayerUndoCommandList.clear();
        }
    }

    private void processCommand(Pair<CommandType, LayerCommand>
        command, Action action)
    {
        switch (action)
        {
            case UNDO:
                processUndoCommand(command);
                break;
            case REDO:
                processRedoCommand(command);
                break;
        }
    }

    private void processUndoCommand(Pair<CommandType, LayerCommand
        > command)
    {
        switch (command.first)
        {
            case COMMIT_LAYER_BITMAP_COMMAND:
                handleUndoCommitLayerBitmapCommand(command.second)
                    ;
                break;
            case ADD_LAYER:
                handleRemoveLayer(command.second);
                break;
            case REMOVE_LAYER:
                handleAddLayer(command.second);
                break;
            case MERGE_LAYERS:
                handleUnmerge(command.second);
                break;
            case CHANGE_LAYER_VISIBILITY:
                handleLayerVisibilityChanged(command.second);
                break;
            case LOCK_LAYER:
                handleLayerLockedChanged(command.second);
                break;
            case RENAME_LAYER:
                handleLayerRename(command.second);
                break;
        }
    }

    private void processRedoCommand(Pair<CommandType, LayerCommand
        > command)
    {
        switch (command.first) {
            case COMMIT_LAYER_BITMAP_COMMAND:
                handleRedoCommitLayerBitmapCommand(command.second)
                    ;
                break;
            case ADD_LAYER:
                handleAddLayer(command.second);
                break;
            case REMOVE_LAYER:
                handleRemoveLayer(command.second);
                break;
            case MERGE_LAYERS:
                handleMerge(command.second);
                break;
            case CHANGE_LAYER_VISIBILITY:
                handleLayerVisibilityChanged(command.second);
                break;
            case LOCK_LAYER:
                handleLayerLockedChanged(command.second);
                break;
            case RENAME_LAYER:
                handleLayerRename(command.second);
                break;
        }
    }

    private void handleUndoCommitLayerBitmapCommand(LayerCommand
        command)
    {
         command.getLayersBitmapCommands().get(0).undo();
         changeActiveLayer(command.getLayer());
         drawingSurfaceRedraw();
    }

    private void handleRedoCommitLayerBitmapCommand(LayerCommand
        command)
    {
         command.getLayersBitmapCommands().get(0).redo();
         changeActiveLayer(command.getLayer());
         drawingSurfaceRedraw();
    }

    private void handleAddLayer(LayerCommand command)
    {
         mLayerBitmapCommandsList.add(command.
             getLayersBitmapCommands().get(0));
         addLayer(command.getLayer());

         changeActiveLayer(command.getLayer());
         layerDialogRefreshView();
         drawingSurfaceRedraw();
    }

    private void handleRemoveLayer(LayerCommand command)
    {
         mLayerBitmapCommandsList.remove(command.
             getLayersBitmapCommands().get(0));
         removeLayer(command.getLayer());

         changeActiveLayer(getNextExistingLayerInCommandList(
             command.getLayer().getLayerID()));
         layerDialogRefreshView();
         drawingSurfaceRedraw();
    }
```

```
/**
 * Undo -- Redo operations are reflections of one another. By
     merge the previously  merged layer
 * needs to be re-added along with its LayerBitmapCommands,
     while origin layers need to be
 * removed along with their LayerBitmapCommands.
 * @param command Layer command containing merged layer and
     its LayerBitmapCommands.
 */
private void handleMerge(LayerCommand command)
{
    ArrayList<LayerBitmapCommands> result =
        getLayerBitmapCommands(command.getLayersToMerge());

    for (LayerBitmapCommands bitmapCommand: result)
    {
        removeLayer(bitmapCommand.getLayer());
        mLayerBitmapCommandsList.remove(bitmapCommand);
    }

    addLayer(command.getLayer());
    mLayerBitmapCommandsList.add(command.
        getLayersBitmapCommands().get(0));

    command.setLayersBitmapCommands(result);

    changeActiveLayer(command.getLayer());
    layerDialogRefreshView();
    drawingSurfaceRedraw();
}

/**
 * Undo -- Redo operations are reflections of one another. By
     un-merge the previously merged layer
 * needs to be removed along with its LayerBitmapCommands,
     while origin layers need to be
 * re-added along with their LayerBitmapCommands.
 * @param command Layer command containing origin layers and
     their LayerBitmapCommands.
 */
private void handleUnmerge(LayerCommand command)
{
    ArrayList<LayerBitmapCommands> result =
        layerIdToOneElementBitmapCommandList(command.getLayer()
        .getLayerID());

    mLayerBitmapCommandsList.remove(result.get(0));
    removeLayer(command.getLayer());

    ListIterator<LayerBitmapCommands> iterator = command.
        getLayersBitmapCommands().listIterator();
    LayerBitmapCommands bitmapCommand;
    while (iterator.hasNext())
    {
        bitmapCommand = iterator.next();
        addLayer(bitmapCommand.getLayer());
        mLayerBitmapCommandsList.add(bitmapCommand);
        iterator.remove();
    }

    command.setLayersBitmapCommands(result);

    changeActiveLayer(getNextExistingLayerInCommandList(
        command.getLayer().getLayerID()));
    layerDialogRefreshView();
    drawingSurfaceRedraw();
}

private void handleLayerVisibilityChanged(LayerCommand command
    )
{
    command.getLayer().setVisible(!command.getLayer().
        getVisible());

    changeActiveLayer(command.getLayer());
    layerDialogRefreshView();
    drawingSurfaceRedraw();
}

private void handleLayerLockedChanged(LayerCommand command)
{
    command.getLayer().setLocked(!command.getLayer().getLocked
        ());

    changeActiveLayer(command.getLayer());
    layerDialogRefreshView();
}
```

```
private void handleLayerRename(LayerCommand command)
{
    String layerName = command.getLayer().getName();
    command.getLayer().setName(command.getLayerNameHolder());
    command.setLayerNameHolder(layerName);

    changeActiveLayer(command.getLayer());
    layerDialogRefreshView();
}

private Layer getNextExistingLayerInCommandList(int
    originLayerId)
{
    synchronized (mLayerCommandList)
    {
        ListIterator<Pair<CommandType, LayerCommand>> iterator
            = mLayerCommandList.listIterator(mLayerCommandList
           .size());

        Layer commandsLayer;
        while (iterator.hasPrevious())
        {
            commandsLayer = iterator.previous().second.
                getLayer();

            if (commandsLayer.getLayerID() != originLayerId )
            {
                if(layerIdToOneElementBitmapCommandList(
                    commandsLayer.getLayerID()).size() == 1)
                {
                    return commandsLayer;
                }
            }
        }

        return null;
    }
}

private synchronized void deleteFailedCommand(Command command)
{

}
```

```
private void drawingSurfaceRedraw()
{
    if(mRedrawSurfaceViewListener != null)
    {
        mRedrawSurfaceViewListener.onSurfaceViewRedraw();
    }
}

private void layerDialogRefreshView()
{
    if(mRefreshLayerDialogListener != null)
    {
        mRefreshLayerDialogListener.onLayerDialogRefreshView()
            ;
    }
}

private void enableUndo(boolean enable)
{
    if(mUpdateTopBarListener != null)
    {
        mUpdateTopBarListener.onUndoEnabled(enable);
    }
}

private void enableRedo(boolean enable)
{
    if(mUpdateTopBarListener != null)
    {
        mUpdateTopBarListener.onRedoEnabled(enable);
    }
}

private void changeActiveLayer(Layer layer)
{
    if(mChangeActiveLayerListener != null)
    {
        for(ChangeActiveLayerEventListener listener :
            mChangeActiveLayerListener)
        {
            listener.onActiveLayerChanged(layer);
        }
    }
    }

    private void removeLayer(Layer layer)
    {
        if(mLayerEventListener != null)
        {
            mLayerEventListener.onLayerRemoved(layer);
        }
    }

    private void addLayer(Layer layer)
    {
        if(mLayerEventListener != null)
        {
            mLayerEventListener.onLayerAdded(layer);
        }
    }

    @Override
    public void update(Observable observable, Object data) {
        if (data instanceof BaseCommand.NOTIFY_STATES)
        {
            if (BaseCommand.NOTIFY_STATES.COMMAND_FAILED == data)
            {
                if (observable instanceof Command)
                {
                    deleteFailedCommand((Command) observable);
                }
            }
        }
    }
}
```

Listing 5.3: CommandManager implementation

6 Conclusion and future work

This thesis discussed the process of translating the old fashioned flip book into an intuitive mobile application, which allows users to create image-by-image animations in combination with sound. The goal was not merely to provide an app that allows users to create a series of images, exploiting two optical phenomena - persistence of vision and the phi phenomenon, but to design a compelling user interface, which is characterized by both efficiency and ease of use.

To solve the interaction problem at hand, this thesis applied the genius design approach of interaction design, which relies solely on the experience and the creativity of an individual designer. Even though seemingly limited in comparison to other interaction design approaches, the genius approach, in combination with user-centered requirements gathering techniques; for instance, persona, user stories and scenarios, provided sufficient data to build a complete conceptual model of an interactive mobile flip book app. Agile principles of prioritizing were used in order to build a first prototype, which was simple and intuitive to use and provided the basis for the first version of the app. Further, the thesis demonstrates that Pocket Toon could capitalize from reusability, familiarity and compatibility by integrating design concepts and functionality of its Pocket "siblings" - Pocket Code and Pocket Paint.

Despite its final outcome, this thesis provides a solid base for design and implementation of an intuitive flip book mobile app:

1. It gathered the necessary requirements.
2. It provided detailed descriptions of features necessary for creating and sharing flip books.
3. It provided a conceptual design, thus described interaction.

4. It prioritized important features and built a low-fidelity prototype which can be used for initial implementation.
5. Finally, it integrated existing design concepts and the functionality of Pocket Code and Pocket Paint, thus benefiting from reusability, familiarity and compatibility.

Thus, future designers and developers can use this thesis as the basis for the further implementation of Pocket Toon.

Appendix

Bibliography

[1] Autodesk. Sketchbook pro features - autodesk, 2013. http://www.autodesk.com/products/sketchbook-pro/features/all/list-view.

[2] T. Boom. Toon boom storyboard pro 4.2 user guide, 2013. http://docs.toonboom.com/help/storyboard-pro/Content/SBP/SBP_Documentation_Cover_Page_UG.html.

[3] Britannica. History of the motion picture — britannica.com, 2015. http://www.britannica.com/art/history-of-the-motion-picture.

[4] A. Developers. Canvas, 2009. http://developer.android.com/reference/android/graphics/Canvas.html.

[5] A. Developers. Canvas and drawables, 2009. http://developer.android.com/guide/topics/graphics/2d-graphics.html.

[6] T. D. Erickson. Human-computer interaction. chapter Working with Interface Metaphors, pages 147–151. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995. ISBN 1-55860-246-1. URL http://dl.acm.org/citation.cfm?id=212925.212939.

[7] Evernote. Evernote: The workspace for your life's work, 2008. https://evernote.com/.

[8] P. Fouché. History - flipbook.info. http://www.flipbook.info/history.php, 2006.

[9] R. Fulcher. Material design awards - articles - google design., 2015. https://design.google.com/articles/material-design-awards/.

[10] Google. Introduction - material design - google design guidelines, 2014. https://www.google.com/design/spec/material-design/introduction.html.

[11] J. Gulbins. *Mastering Photoshop Layers: A Photographer's Guide*. Rocky Nook, 2013. ISBN 9781492001287. URL https://books.google.at/books?id=xD-4BAAAQBAJ.

[12] H. Hartson and P. Pyla. *The UX Book: Process and Guidelines for Ensuring a Quality User Experience*. Elsevier, 2012. ISBN 9780123852410. URL https://books.google.at/books?id=5KqoHjeEKkkC.

[13] G. Hofstede, G. Hofstede, and M. Minkov. *Cultures and Organizations: Software of the Mind, Third Edition*. McGraw-Hill Education, 2010. ISBN 9780071770156. URL https://books.google.at/books?id=o4OqTgV3V00C.

[14] InVision. Invision: Free web and mobile prototyping, 2011. https://www.invisionapp.com.

[15] S. Iwata. Iwata asks : Nintendo dsi : Volume 7 : Flipnote studio, 2013. http://iwataasks.nintendo.com/interviews/.

[16] R. Lynch. *The Adobe Photoshop CS4 Layers Book: Harnessing Photoshop's Most Powerful Tool*. Safari Books Online. Focal Press/Elsevier, 2009. ISBN 9780240521558. URL https://books.google.at/books?id=L2Op956Iz_8C.

[17] Merriam-Webster. Affordance, 2008. http://www.merriam-webster.com/dictionary/affordance.

[18] Nintendo. Official site - flipnote studio 3d for nintendo 3ds, 2013. http://flipnotestudio3d.nintendo.com.

[19] M. of the Moving Image. Museum of the moving image - shutters, sprockets, and tubes, 2007. http://www.movingimage.us/sprockets/.

[20] L. Rains. Efficient and powerful storyboarding wins, 2013. https://www.toonboom.com/community/success-stories/larry-rains.

[21] J. Rasmusson. *The Agile Samurai: How Agile Masters Deliver Great Software*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010. ISBN 9781934356586. URL https://books.google.at/books?id=KjmXSQAACAAJ.

[22] Y. Rogers, H. Sharp, and J. Preece. *Interaction Design: Beyond Human - Computer Interaction*. Interaction Design: Beyond Human-computer Interaction. Wiley, 2011. ISBN 9780470665763. URL https://books.google.at/books?id=b-v_6BeCwwQC.

[23] D. Saffer. *Designing for Interaction: Creating Smart Applications and Clever Devices*. Voices That Matter. Pearson Education, 2006. ISBN 9780132798105. URL https://books.google.at/books?id=Ckrd3Hoi4IsC.

[24] J. Tidwell. *Designing Interfaces*. O'Reilly Media, 2010. ISBN 9781449302733. URL https://books.google.at/books?id=5gvOU9X0fu0C.

[25] T. White. *Animation from Pencils to Pixels: Classical Techniques for Digital Animators*. NetLibrary Inc. Focal, 2006. ISBN 9780240806709. URL https://books.google.at/books?id=oyzBquu0ULMC.

[26] F. A. Wika. Flipnote hatena, 2015. http://flipnote.wikia.com/wiki/Flipnote_Hatena.