Technical report to the master’s project

Supporting methodic design practices with interactive organization and visualization of design artifacts

ANITA HÖCHTL
(01/744142)

INFORMATION ENGINEERING

University of Konstanz
Department of Computer and Information Science

Subject of Study: Information Engineering
Specialization: Computer Science
Topic: Applied Informatics
Submission Date: September 30th, 2011
Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Konstanz, September 30th, 2011

Anita Höchtl
Contents

Erklärung iii

1 Introduction 1

2 Hardware / Software 4
   2.1 Microsoft Surface 1.0 ................................................. 4
   2.2 ZOIL ................................................................. 4
      2.2.1 MVVM pattern .................................................. 5
      2.2.2 Device Handlers ............................................... 5
      2.2.3 Attached Behaviors ......................................... 5
      2.2.4 Interaction Logic Modification ............................. 6
      2.2.5 VisualProperties .............................................. 7
      2.2.6 Animations .................................................... 7

3 Interfaces 9

4 Blub 12
   4.1 Idea ............................................................... 12
   4.2 Visual Representation ........................................... 13
      4.2.1 Performance .................................................. 15
      4.2.2 Adjustment of Parameters ................................ 16
   4.3 Interaction Techniques ....................................... 18
      4.3.1 Grouping ..................................................... 18
      4.3.2 Transferring ................................................. 19
      4.3.3 Splitting .................................................... 21
      4.3.4 Spreading ................................................... 23

5 Bin 25
   5.1 Idea ............................................................... 25
   5.2 Interaction Techniques ....................................... 25

iv
Contents

5.2.1 Holing .................................................. 26
5.2.2 Collecting .............................................. 27
5.2.3 Adjusting ............................................. 28
5.2.4 Transferring ......................................... 28
5.2.5 Spreading ............................................. 29

6 Logging ..................................................... 30
   6.1 Implementation ....................................... 30

7 Conclusion .................................................. 32

Bibliography ................................................. 33
# List of Figures

1.1 Examples of primitive spatial structures [10]. ........................................ 1
1.2 ControlBox of the Microsoft Surface 1.0 SDK. ....................................... 2
2.1 Changes in the interaction logic of ZOIL for Blub. ................................. 6
2.2 Changes in the interaction logic of ZOIL for Bin. ................................... 6
3.1 Program loop. ...................................................................................... 9
3.2 (1) Starting Blub and Bin interface. (2) Blub interface before initializing landscape. (3) Bin interface before initializing landscape. ............ 9
3.3 (1) Positioning objects randomly. (2) Result of grouping. (3) Shuffling colors of objects. (4) Result of regrouping. ................................. 10
3.4 Procedure of one session. ................................................................. 11
4.1 Bubble containing two objects. .......................................................... 12
4.2 (1) Calculate the bounding box. (2) Estimate each pixel's energy. (3) Extract all points between a min and max energy value. (4) Sort all points according their distance to each other and draw a spline. .................. 13
4.3 (1) Potential field of a single object. (2) Potential field around more objects. 13
4.4 Parameter adjustment for the designer. ................................................ 16
4.5 Force Field Threshold is the radius (R₁) of the potential field. ............... 16
4.6 (1) Contour Threshold defines the energy where to draw a spline. (2) Contour Threshold Offset sets an offset around this value. .................... 17
4.7 The Sampling Size indicates the number of pixels to consider inside the bounding box. ................................................................. 17
4.8 ((1) Estimate the current point. (2 - 3) Define a Min Distance and a Max Distance to restrict the distance to the current point. ............... 17
4.10 (1) Dragging an object. (2) Touching boundaries. (3) Melting of bubbles... 18
4.9  (A-4) *Force Field Threshold* has great influences to the number of bubbles for more objects. (B-4) *Contour Threshold* decides on the spline position in the potential field. (C-1) shows that a too low *Contour Threshold Offset* causes missing spline points. (D-3) A high *Sampling Size* results in an inaccurate visualization especially for bigger bubbles. (E-3) *Min Distance* and (F-1) *Max Distance* influences the number of points in the spline. .......... 19

4.11 (a) Touching the bubble. (b) Dragging the bubble over objects. (c) Release bubble on the desired position. (d) Merging of bubbles. .......... 20

4.12 (a) Pushing the scissor button. (b) Drawing a stroke. (c) Moving bubbles apart from each other. (d) Result. ................. 21

4.13 (a) Creating a line. (b) Determine \( \alpha \), calculate and rotate offset vector. (c) Animate objects. ................................. 21

4.14 (1) Placing thumb and index finger on the bubble. (2) Pulling them apart. 23

4.15 (1) Triangulation. (2) Scale difference vector. (3) Set position. .......... 24

5.1 Bin containing two objects. .................................................. 25

5.2 (1) Touching and dragging the object. (2) Releasing the object. (3) Resizing object in bin. (4) Kick-off object. (5) Object moves. (6) Holing object into bin. .................................................. 26

5.3 (1) Touching the bin. (2) Dragging the bin. (3) Releasing the bin. (4) Holing and resizing the object. .................................................. 27

5.4 (1) Touching and dragging the handle. (2) Releasing the handle. (3) Resizing object in bin. .................................................. 28

5.5 (1) Pushing the lasso button. (2) Drawing a selection. (3) Touching and dragging the selection. (4) Releasing the selection. (5) Resizing objects and removing the selection. .................................................. 28

5.6 (1) Placing thumb and index finger on the bin. (2) Pulling them apart. 29

6.1 Sample data of the *Session Logger*. .................................. 31
List of Algorithms

4.1 Calculate bounding box. .................................................. 14
4.2 Estimate pixel energies. .................................................... 14
4.3 Extract contour points. .................................................... 14
4.4 Sort points according to their distance to each other. ................. 15
4.5 Estimate closest bubble. Update all relevant bubbles. ................ 20
4.6 Splits a bubble. .............................................................. 22
4.7 Animates objects apart. ....................................................... 23
4.8 Spreads objects. .............................................................. 24
5.1 Checks, to which bin one object belongs. .......................... 26
5.2 Checks, whether any object is new in a bin. ........................ 27
5.3 Checks, whether any bin has any new objects. ...................... 29
List of Source Code Snippets

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Animations: Defines the necessary variables.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Animations: Attaches/Releases the animation’s updating method.</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Animations: Initializes the animation.</td>
<td>8</td>
</tr>
<tr>
<td>6.1</td>
<td>Logging: Configure logger.</td>
<td>30</td>
</tr>
<tr>
<td>6.2</td>
<td>Logging: Initialize logger.</td>
<td>31</td>
</tr>
<tr>
<td>6.3</td>
<td>Logging: Log header and events.</td>
<td>31</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This technical report describes the implementation of two concepts for manipulating objects and groups on digital tabletops. Organizing information efficiently is still a topic in research. Current desktops require titling of files, which is rather than what humans do, when they organize information in the physical life. An early work of Mander [7] shows that e.g. folders in real life were structured in random, alphabetical or color-coded manner. In digital life, folders have to be titled, random order is difficult, alphabetical order is common used and color-coded ordering is currently not possible. So, the goal should be to identify aspects which work good in real life and transfer those to the digital life.

Desk organization, for example, has to major goals: reminding the user of things to do and the categorization of information [6]. The problem with the latter one is the process of categorization, as not each piece of information belongs to a specific category, and the category itself, as sometimes the meaning is not uniquely. User-constructed spatial layouts can be divided into four different types [10] (see Figure 1.2): Lists share common features across objects. Stacks are compact, include only one type of objects and are used when less space is available. Composites are a kind of regular hierarchical structures, which typically have a 1 : n relationship. Heaps are similar to stacks, but allow different types of objects to be stacked. The last category is the most interesting one for us, as it allows to pile all kind of information types (e.g. books, magazines, reports, cards, images,...). Furthermore, it may helps in organizing messy workspaces, for example those from designers [12]. We think that especially for managing visual artifacts such as images, videos or sketches, digital tabletops have the potential to support the user in spatial layouting tasks as well as in object and group manipulation. Nevertheless, the success is highly dependent on the interaction concept. For this purpose, we introduce two ideas of how interaction can work on digital tabletops in a grouping task, in the following.

Figure 1.1: Examples of primitive spatial structures [10].
1. Introduction

To structure stuff in real life, all kind of boxes can be used. These boxes act as container, where things were stored in a structured way. These containers also act as transport medium. In digital life, folders meet this purpose. Nevertheless, the user does often not exactly know the content of the folder, which is still a problem, in physical and digital life. Especially for visual content such as images, sketches, videos, etc., digital tabletops could contribute the task of creating an own mental image of content instead of having black boxes. But currently, there are less systems which meet these demands. The Microsoft Surface SDK\(^1\) provides a **ControlBox** which supports storing and retrieving items in grid or pile layout. The main drawback is that this control restricts the user in creating a more flexible layout. Therefore, we want to create something in-between too restrictive and too flexible. Malone [6] argues, that classification can be done in three different manners: *Multiple classifications* allow more than one membership of one object, which is a 1:n relationship. *Deferred classification* means that the position of the object has a meaning for the user. *Automatic classification* is supported through automatic clustering (e.g. location, object type, name,...) through the system. The position of an object on a digital tabletop meets already the needs of deferred classification depending on the size of the table and the size of the objects. But concerning multiple classification, there is less support with the current state-of-the-art storage mediums such as the Control Box on the Microsoft Surface. In the following, we address this problem in the Blub interface. Depending on the amount of data, automatic classification could be a future direction, but should be combined with manual structuring on digital tabletops to give greater flexibility to the user particularly in managing visual artifacts. The reason for the combination is that the user might have a different view on objects than the system automatically calculates. But automatic classification can be used to presort objects on the digital tabletop, so that the user has not to start by zero.

As we have shown, current containers are restrictive. To meet the need of more flexibility, we transfer the concept of Bubble Clusters [13] from mouse to touch input. By means of the Gestalt law of proximity [11], spatial aggregated objects were automatically recognized as groups and the system visualizes the result as a bubble surrounding the objects. The second concept, we introduce, is based on Storage Bins [9], a mobile, adjustable container widget allowing the user to store and retrieve workspace content such as images, documents or thumbnails anywhere on a digital tabletop. The roots of bins lie in the container schemata [5]. By using eight handles, a bin's shape can be adjusted and the container has a more flexible form. Our main goal is to support the user's need of organizing objects and groups on digital tabletops. Thereby, we want to retain the user's flexibility of creating an own mental picture.

This technical report is structured into seven chapters. This first chapter is dedicated to our motivation and our goals. Chapter 2 clarifies our hardware and software decisions. In Chapter 3, a general interface introduction is given and the study procedure is delineated. Chapter 4 and Chapter 5 provide descriptions about the ideas behind both interfaces, the visual representations and the interaction techniques. To understand the technical part of

\(^{1}\text{http://www.microsoft.com/surface/en/us/default.aspx}\)
both concepts, precise implementation details illustrate the single processes of visualizing and interacting. For the user study, a logging mechanism was implemented which can be found in Chapter 6. Finally, Chapter 7 sums up this report and provides ideas for future improvements in a technical sense.
Chapter 2

Hardware / Software

This chapter clarifies our hardware and software decisions. In addition, we give a short introduction into the interaction relevant concepts of ZOIL, such as the MVVM pattern, device handlers, attached behaviors and demonstrate the changes, which we made for the Blub and Bin interface. Furthermore, we present the VisualProperties and show how they can be used for animations.

2.1 Microsoft Surface 1.0

For a proof of concept, a digital tabletop was necessary. For our task, the size of the Microsoft Surface was appropriate, because we had 30 small colored shapes for grouping. There was also enough space to draw non-overlapping bubbles around those shapes in the Blub condition. Once someone wants to use these interaction concept in a more complex task (e.g. detailed sketches with descriptions instead of colored shapes), a larger system should be considered. The size of the table depends on the number of users, the number of artifacts, which should be organized, and the size of them. Another reason for using the Microsoft Surface 1.0 was that we needed a stable system. As it is a consumer product, it is more solidly built than lab prototypes.

2.2 ZOIL

Both interfaces were developed with the ZOIL Framework 1. ZOIL (Zoomable Object-Oriented Information Landscape) is a design paradigm and a software framework, written in C# for the .NET/WPF platform. ZOIL is developed by the Human-Computer-Interaction Group of the University of Konstanz2. Several external frameworks and APIs, e.g. Versant’s db4objects (db4o) database and the Microsoft Surface SDK were integrated. As ZOIL is a ZUI, an infinite information landscape can be used to show content. Thereby, visual zooming allows to focus on a region of interest. The main advantages of ZOIL are the semantic zooming for smoothly changing content by zooming in, persistency and dis-
2. Hardware / Software

*tributed and real-time synchronisation* as a data model can be shared across devices by using the d4o database [14].

The idea behind using ZOIL for our interfaces was the reusability. Blub and Bin are both interaction concepts and are not mandatory related to a specific project. Future developers should be able to pass both interaction concepts without massive effort to their projects. According to Watanabe [13], Bubble Clusters, which is the base of Blub, could be interesting in combination with ZUIs to create an infinite amount of space. That is due to our choice of using ZOIL.

### 2.2.1 MVVM pattern

As currently introduced, one of the main advantages of ZOIL is the *persistence*, which allows users of different clients independent navigation and manipulation of content. Therefore, ZOIL uses the *MVVM pattern (Model-View-ViewModel Pattern)*. If a component is created by the user, one *Model* is stored in the database and one *View* is produced for each client. The *ViewModel* lies in-between and is synchronized with the *Model* and bound to the *View* [3,14].

### 2.2.2 Device Handlers

ZOIL provides different handlers in its architecture. Handlers concern events from different input devices such as mouse or touch and were either related to the information landscape or to the objects. To illustrate just one example: ZOIL is a ZUI and so the user can zoom into a region of interest in the landscape. Therefore, he places thumb and index finger on the landscape and pulls them apart, which corresponds a pinching gesture. While this gesture is performed, the handler listens to the event of touching the surface and gets information about the the distance between thumb and index finger. This distance is the factor used for zooming. Another example is when the user wants to jostle an object across the surface, he just pokes the object and the object moves without further control through the user. The handler gets the information about the poking and triggers this information to the physical behavior, which passes the event to the corresponding method and calculates all further movement for the involved objects.

### 2.2.3 Attached Behaviors

In ZOIL, each object in the information landscape has a visual representation, properties (e.g. width, height) and different behaviors. A behavior is a kind of black box attached to an object, which takes care of object-related events. That is always, when something happens, which has consequences to the object. For example, if the user touches an object and wants to drag it to a specified position, the system should know that the object should be moved underneath the user's finger. Therefore, the behavior of the object gets an event from the device handler and routes this to the appropriate processing method. So, a behavior is a form of distribution center for processing special incoming messages to the respective processing methods. An incoming message can also be created inside the system. For example, if one object collides with another object, both should move. To do
this, the behavior of the first one toggles an event, which receives the behavior of the second object as incoming message and triggers a movement event for the second object. For more flexibility, a behavior can be adjusted to its purpose. In other words, processing methods can be cut off, that incoming messages were caught in the behavior, but do not produce any consequences. E.g. a specific type of objects act as anchors and the user should not be able to move these objects. For this purpose, the manipulation property (translation, rotation, scalability) is disabled in the behavior of this object type.

Because, Blub and Bin affect the interaction with objects, we decided to create two new behaviors in ZOIL - one for each interaction concept. Each behavior has a bundle of properties, which can be attached to objects and a number of events.

### 2.2.4 Interaction Logic Modification

In the last two sections, we have shown that ZOIL has handlers for the different devices which route events to the corresponding behaviors to trigger actions. In our implementation, we used partly the existing interaction logic and did some changes in the device handlers and the behaviors.

Blub supplies four different operation modules: Grouping, Transferring, Splitting and Spreading. As Figure 2.1 shows, Grouping and Transferring were attached to the DragDropBehavior. For Splitting, we abused the SelectionBehavior, because the process of drawing a selection shape and drawing a free-form splitting path are similar. For both cases, a figure is created, which is used to identify all selected objects while selecting or cuts an object through the defined path while splitting. For fanning objects apart, we used the ResizeBehavior.

Bin provides five different operation modules: Holing, Collecting, Adjusting, Transferring and Spreading. Holing and Collecting were attached to the DragDropBehavior as they are related to moving objects. As adjusting works with circular handles at the edges of a bin, the behavior therefore is implemented directly in the bin component. For transferring objects, Bin provides a lasso. This lasso is part of the SelectionBehavior. As selected objects should also be able to move, we created a component for visualizing the selection and attached this to that behavior. Finally, Spreading is again connected to the ResizeBehavior.
2.2.5 VisualProperties

As introduced in Section 2.2, one of the main advantages of ZOIL is persistency. For that reason, objects were stored in a database. To sustain information concerning rendering (e.g. position, width, height, opacity,...) each object has attached visual properties. The model of the object is stored together with the visual properties to create the same view for the object on different displays. E.g. if one object’s position is $P(100/100)$, this position concerns only the information landscape. So, the relative position will stay the same across a variety of displays with different solutions as the size of the information landscape between displays should be the same.

The purpose of both interfaces is to allow structuring objects on digital tabletops via a storage medium which indicates the membership of each object to the storage medium. As this information is accessed very often during using one of both interfaces, because the task is grouping objects, we decided to store the id of the storage medium (bubble or bin) in the attached visual properties of an object, to have easy access to this information.

2.2.6 Animations

To raise the user experience, we decided to create animations for smoother transitions in object movements (see Section 4.3.3, 4.3.4 and 5.2.5). To do so, an object’s visual properties (see Section 2.2.5) were used. To simplify the process of building animations, a VisualPropertiesAnimation$^3$ undertakes the execution and calculation of simple movement animations. In the following, we give a short tutorial of how to create a VisualPropertiesAnimation:

First, the variables for animation (anim), start time (startTickCount) were defined and the general animation duration (duration) is set.

```
1. VisualPropertiesAnimation anim; // animation
2. float duration = 200; // animation duration, e.g. 20ms
3. float startTickCount; // start time
```

Source Code 2.1: Animations: Defines the necessary variables.

Next, a method for updating the animation has to be attached to the rendering method in the same class. Therefore, methods for starting (startAnimation(...)) and stopping an animation (stopAnimation(...)) were created. For updating the process of animation, a method (onRenderReset(...)) is attached to the rendering process, calculates the current progress of the animation and either triggers this to the animation or stops the animation.

```
1. static void startAnimation()
2. {
3.     CompositionTarget.Rendering += onRenderReset; // attaches this method to the rendering event
4. }
5. static void onRenderReset(object sender, EventArgs e)
6. {
7.     float curTickCount = Environment.TickCount; // current time
8.     float progress = (curTickCount - startTickCount) / duration; // progress of the animation
9.     anim.updateProgress(progress);
10. }
11. if (progress > 1) {
12.     anim.stopAnimation();
13. }
```

$^3$Originally developed by Marcus Specht. Adapted by Anita Höchtl.
2. Hardware / Software

Source Code 2.2: Animations: Attaches/Releases the animation’s updating method.

```c
static void stopAnimation()
{
  CompositionTarget.Rendering = comTargetReset; // releases this method from the rendering event
}
```

Finally, an animation is created. Therefore, the object itself and original visual properties (objVSP) as well as final visual properties (endVSP) were needed. Furthermore, the start time is set, to start the animation.

Source Code 2.3: Animations: Initializes the animation.

```c
// create an animation with the object, the original position and the end position
mim = new VisualPropertiesAnimation(mj, objVSP, endVSP);

mim.prepareStart(); // prepare the animation to start

startTickCount = Environment.TickCount; // sets the start time
```
Chapter 3

Interfaces

To proof both interaction concepts, we decided to conduct a user study. In this chapter, we describe the general interface used for this and the procedure of the study.

As Figure 3.1 shows, a start screen is loaded, at the beginning. Afterwards, pushing of the A-Button starts the Blub interface and pushing the B-Button starts the Bin Interface. Pressing the Play-Button initializes the landscape. In the Blub condition, a scissor button is provided for splitting bubbles (see Section 4.3.3). To activate the selection tool in the Bin condition (see Section 5.2.4), a lasso button is on the bottom left corner. The start screen and both interfaces were shown in Figure 3.2.

![Diagram](image)

**Figure 3.1:** Program loop.

![Images](images)

**Figure 3.2:** (1) Starting Blub and Bin interface. (2) Blub interface before initializing landscape. (3) Bin interface before initializing landscape.
3. Interfaces

The study was separated into two main parts for each participant - one for each interface. One part consisted of one task to be completed four times. The task was again divided into two subtasks: grouping and regrouping. The procedure of one task was the following (see also Figure 3.3):

1. Random distribution of 2D shapes, colored in yellow, green, blue, pink and violet, on the surface.
2. Grouping of shapes according their colors.
4. Regrouping of shapes according their colors.

![Figure 3.3: (1) Positioning objects randomly. (2) Result of grouping. (3) Shuffling colors of objects. (4) Result of regrouping.](image)

Both subtasks were successfully completed when all 2D shapes were grouped according their colors. If participants needed more than five minutes for a subtask, the next subtask started automatically. Figure 3.4 sums up schematically the implementation of the whole procedure for one interface condition per participant.
Figure 3.4: Procedure of one session.
Chapter 4

Blub

This chapter describes the basic idea behind the Blub interface and explains the visualization as well as the interaction techniques in detail. For this purpose, algorithms were discussed, illustrated and enriched by visual representations.

4.1 Idea

Blub is the approach to bring the existing interaction concept of Bubble Clusters [13] from the desktop to the digital tabletop. The idea behind Bubble Clusters was to support object and group manipulation through a dynamic visualization. At the very beginning, each object is surrounded by a bubble, which means that each object is one group. A bubble is a kind of container, which dynamically adjusts its shape according number and positions of objects in it. The roots of the bubble concept lie in the Gestalt law of proximity as spatial or temporal aggregated objects were perceived as a collection [11]. This proximity defines whether bubbles melt together or not. Figure 4.1 shows two objects with a surrounding bubble.

During a dragging operation, a bubble dynamically updates its visual representation and gives feedback about its membership. Dragging a bubble allows to transfer a collection of stored objects to a desired position. Items can be added or removed to or from a bubble. Therefore the object itself or the surrounding bubble can be dragged. In addition, bubbles can be easily split through drawing a free-form stroke across the bubble. For getting an overview of all the objects inside a bubble, objects can be spread out with a pinching gesture.

Figure 4.1: Bubble containing two objects.
4.2 Visual Representation

The process of visualizing a bubble can be separated into four main stages (also see Figure 4.2):

1. Calculate the bounding box around the objects, which should be grouped.
2. Estimate each pixel’s energy in the bounding box.
3. Extract all points between a min and max energy value.
4. Sort all points according their distance to each other and draw the contour.

![Figure 4.2](image)

**Figure 4.2**: (1) Calculate the bounding box. (2) Estimate each pixel’s energy. (3) Extract all points between a min and max energy value. (4) Sort all points according their distance to each other and draw a spline.

**Implementation** First, a circular potential field with the radius $R_1$ is created for each object (see Figure 4.3). Then, a bounding box limits the potential fields of the objects, which should be included (Figure 4.2-(1)).

To estimate each pixel’s energy, a 2D version of blobby shapes [1] is used. From the center ($f(R_0) = 1$) to the edge ($f(R_1) = 0$), the energy drops out in a potential field. So, $R_0$ is the center of an object and $R_1$ is the radius of the potential field.

Now, to establish each pixel’s energy value in the bounding box (see Figure 4.2-(2)), the sum of influences of the nearby objects is calculated with Equation 4.1. Algorithm 4.2 shows the implementation therefore.

$$energy(\text{pixel}) = \sum_{i \in s_{\text{pixel}}} (R_1 - distance_{i, \text{pixel}})^2 / (R_1 - R_0)^2$$  \hspace{1cm} (4.1)

Subsequently, all values between a min and max value were extracted and stored together with the x/y - position (see Figure 4.2-(3) and Algorithm 4.3).
Algorithm 4.1: Calculate bounding box.

1: \textbf{CALCULATEBOUNDINGBOX}(O) \hspace{1cm} \triangleright \text{All objects.}
2: \hspace{1cm} \textbf{for all} \: o \in O \hspace{1cm} \textbf{do}
3: \hspace{1.5cm} \textbf{if} \: posOf(o).X - R_1 < boundingBox.Left \hspace{1cm} \textbf{then}
4: \hspace{2.5cm} boundingBox.Left \leftarrow posOf(o).X - R_1 \hspace{1cm} \triangleright \text{Sets left boundary.}
5: \hspace{1.5cm} \textbf{end if}
6: \hspace{1.5cm} \textbf{if} \: posOf(o).X + R_1 > boundingBox.Right \hspace{1cm} \textbf{then}
7: \hspace{2.5cm} boundingBox.Right \leftarrow posOf(o).X + R_1 \hspace{1cm} \triangleright \text{Sets right boundary.}
8: \hspace{1.5cm} \textbf{end if}
9: \hspace{1.5cm} \textbf{if} \: posOf(o).Y - R_1 < boundingBox.Top \hspace{1cm} \textbf{then}
10: \hspace{2.5cm} boundingBox.Top \leftarrow posOf(o).Y - R_1 \hspace{1cm} \triangleright \text{Sets top boundary.}
11: \hspace{1.5cm} \textbf{end if}
12: \hspace{1.5cm} \textbf{if} \: posOf(o).Y + R_1 < boundingBox.Bottom \hspace{1cm} \textbf{then}
13: \hspace{2.5cm} boundingBox.Bottom \leftarrow posOf(o).Y + R_1 \hspace{1cm} \triangleright \text{Sets bottom boundary.}
14: \hspace{1.5cm} \textbf{end if}
15: \hspace{1cm} \textbf{end for}
16: \hspace{1cm} \textbf{return} \: boundingBox
17: \textbf{end}

Algorithm 4.2: Estimate pixel energies.

1: \textbf{ESTIMATEPIXELENERGIES}(boundingBox, O) \hspace{1cm} \triangleright \text{Bounding Box and all objects.}
2: \hspace{1cm} \textbf{for all} \: \text{pixel} \in \text{Pixel}(boundingBox) \hspace{1cm} \textbf{do} \hspace{1cm} \triangleright \text{For each pixel in the Bounding Box.}
3: \hspace{1.5cm} \textbf{for all} \: o \in O \hspace{1cm} \textbf{do}
4: \hspace{2.5cm} distance \leftarrow posOf(o) - posOf(pixel)
5: \hspace{2.5cm} \textbf{if} \: \text{distance} < \text{threshold} \hspace{1cm} \textbf{then}
6: \hspace{3.5cm} \text{pixel.Energy} \leftarrow \text{pixel.Energy} + ((R_1 - \text{distance})^2/(R_1 - R_0)^2)
7: \hspace{2.5cm} \textbf{end if}
8: \hspace{1.5cm} \textbf{end for}
9: \hspace{1cm} \textbf{end for}
10: \hspace{1cm} \textbf{return} \: \text{Pixel}
11: \textbf{end}

Algorithm 4.3: Extract contour points.

1: \textbf{EXTRACTIONSPOINTEMATRICE}(boundingBox) \hspace{1cm} \triangleright \text{Bounding Box.}
2: \hspace{1cm} \textbf{for all} \: \text{pixel} \in \text{Pixel}(boundingBox) \hspace{1cm} \textbf{do} \hspace{1cm} \triangleright \text{For each pixel’s energy.}
3: \hspace{1.5cm} \textbf{if} \: \text{pixel.Energy} > \text{min} \: \& \: \text{pixel.Energy} < \text{max} \hspace{1cm} \textbf{then}
4: \hspace{2.5cm} Points \leftarrow posOf(pixel)
5: \hspace{1.5cm} \text{end if}
6: \hspace{1cm} \text{end for}
7: \hspace{1cm} \textbf{return} \: \text{Points}
8: \textbf{end}
Now, all points were sorted according their distance to each other. To start, the nearest neighbor point of the first point is searched and taken to repeat this procedure. All used points were saved to new points \((\text{Points}_{\text{sorted}})\) and removed from the old points \((\text{Points})\). The sorting procedure (see Algorithm 4.4) is essential for the drawing, as otherwise, the points will be drawn line-by-line, from left to right or from top to bottom, which ends in a zig-zag line. Finally, \(\text{Points}_{\text{sorted}}\) were used to draw the line (see Figure 4.2-(4)).

**Algorithm 4.4:** Sort points according to their distance to each other.

1. \(\textbf{Sort}(\text{Points})\) \(\triangleright\) Points.
2. \(\textit{point}_\text{tmp} \leftarrow \text{Points}_0\) \(\triangleright\) Initialize with the first point.
3. \(\textbf{while} \ \text{Count}(\text{Points}) \neq 0 \ \textbf{do}\)
4. \(\textit{point}_\text{nn} \leftarrow \text{FindNearestNeighbourPoint}(\text{point}_\text{tmp}, \text{Points})\)
5. \(\textit{point}_\text{tmp} \leftarrow \textit{point}_\text{nn}\)
6. \(\text{Points}_{\text{sorted}}.\text{Add}(\textit{point}_\text{nn})\)
7. \(\text{Points}.\text{Remove}(\textit{point}_\text{nn})\)
8. \(\textbf{end while}\)
9. \(\textbf{return} \ \text{Points}_{\text{sorted}}\)
10. \(\textbf{end}\)

**4.2.1 Performance**

The presented algorithm is too slow for Blub. The reason therefore is the high computational effort for the estimation of each pixel’s energy and the sorting procedure of the points in the last part of the algorithm. Section 4.3.1 shows that the permanent visualization of a bubble is important for this interaction concept as the visual feedback is essential and increases the user experience. For this purpose, we decided to reduce and simplify some of the parts in the algorithm and to pre-calculate some values to save time. The following paragraphs describe these reductions, simplifications and pre-calculations.

Currently, there is a huge amount of pixel energy values which has to be estimated for each bounding box. To decrease the number pixels used for the calculation, a subsampling factor was introduced. This factor defines how many rows and columns of the pixel bounding box should be considered for the calculation. E.g. a subsampling of 2 reduces the number of pixel calculations by half, which means that also half of the time of this step in the algorithm is saved. An interface allows to adjust this factor (see Section 4.2.2).

Furthermore, to decrease time, the pixel energy calculation (see Equation 4.1) was divided into three subparts: distance-dependent part (see Equation 4.2), static part (see Equation 4.3) and the summing up of all potential field influences (see Equation 4.4).

\[
\text{part}_1 = (R_1 - \text{distance}_{i, \text{pixel}})^2 \quad (4.2)
\]
\[
\text{part}_2 = (R_1 - R_0)^2 \quad (4.3)
\]
\[
\text{energy(pixel)} = \sum_{i \in s_{\text{pixel}}} \langle \text{part}_1/\text{part}_2 \rangle \quad (4.4)
\]
Although, the distance-dependent part varies according the distance between the current pixel position and the center of a potential field, values are the same for same distances and can be pre-computed. For this purpose, a stepsize was introduced, which defines the interval between the pre-calculated values. So, to decrease the computational effort for mathematical operations during run time, distance-dependent values \((part_1)\) were pre-estimated. Currently, a stepsize of \(1/\text{threshold}\) is used for this procedure. The static part \((part_2)\) was calculated once at the beginning of the program, as it stays the same for all pixel energy values, because \(R_0\) and \(R_1\) are static values and are independent from the current pixel position. During run time, the system accesses the corresponding distance-dependent value \((part_1)\), takes the static value \((part_2)\), divides those and sums up the pixel energy influences from the potential field on the current pixel (see Equation 4.4).

Finally, the points for the spline were downsized in order save time while sorting them in ascending distance to each other, as a low number of points means also less potential nearest neighbor candidates (see Algorithm 4.4). Additionally, less points create a smoother spline as bezier splines were used at the moment.

### 4.2.2 Adjustment of Parameters

The bubble visualization is fine adjustable. During the developing process, we created an interface, made up of six sliders, which have different influences to the surrounding contour of a bubble. Figure 4.4 shows this interface. The developer can display this tool through an expander on the top right corner of the Blub interface. Especially during the development of Blub, we used this tool to create a good-looking visualization and tune the performance. In the following, we introduce the different parameters and show their influences.

The **Force Field Threshold** slider is the most influential parameter, as it is used to specify the radius \((R_1)\) of the potential field around a bubble (see Figure 4.5). In our implementation, this value is 2.25 times the size of an object.

A potential field has an decreasing energy from the center to the edge. The contour is drawn on a specific pixel energy value between center \((R_0)\) and edge \((R_1)\). The **Contour Threshold** slider allows to adjust this pixel energy value. By default, this value is set to 0.3.

The **Countour Threshold Offset** gives an amount of leeway to the contour threshold. The reason is that otherwise only pixel with the specific contour threshold value were considered for the spline. The offset extends the value range around the contour threshold. But, caution is called, as a too low value ends in an inaccurate contour and a too high
value produces a less smooth line as contour. Figure 4.6 shows the potential field with the contour threshold and its offset in circular view and illustrates the defined value range through beams. In our current implementation, we used a contour threshold offset of 0.0035.

\[\text{Figure 4.6:} (1) \textit{Contour Threshold} defines the energy where to draw a spline. (2) \textit{Contour Threshold Offset} sets an offset around this value.\]

Since the procedure of calculating a bubble’s visualization requires a very high computational effort, the process therefore can be simplified. Figure 4.2-(2) has demonstrated the estimation of each pixel’s energy. Doing this for each pixel inside a bounding box is very expensive. As a consequence, the \textit{Sampling Size} slider allows to define a sampling factor, which means that only every $ns$ pixel is calculated. Reducing the sampling size has proved an effective instrument for tuning the performance. However, the sampling size should not be too high, as otherwise there were too less pixels to calculate a spline. For our implementation, a value of 2 was appropriate, which corresponds to the half number of pixels and the half time. Figure 4.7 shows the procedure of sampling schematically.

\[\text{Figure 4.7: The Sampling Size indicates the number of pixels to consider inside the bounding box.}\]

\[\text{Figure 4.8:} (1)\text{Estimate the current point.} (2 - 3)\text{ Define a Min Distance and a Max Distance to restrict the distance to the current point.}\]
To sort the points in ascending order to the each other, the distance between them was used. The \textit{Min Distance} therefore defines the minimal distance between a current point and the next point. This is to reduce the number of points for the drawing the spline. The \textit{Max Distance} restricts the distance to the next point. The reason is again to shrink the number of points. Figure 4.8 demonstrate the examination of a nearest neighbor point.

Figure 4.9 shows the effects of changing the parameter values. Therefore, each row describes one parameter and each column illustrates one specific value for the parameter. Column (4) combines all three results in one picture.

### 4.3 Interaction Techniques

To understand, each of the following sections is structured as follows: First, on the basis of figures and short descriptions, we explain how the interaction technique works for the user. Afterwards, we provide some implementation details in the form of algorithms and illustrate mathematical procedures with equations and figures.

#### 4.3.1 Grouping

To group objects in one bubble, bubbles melt as their boundaries touch each other. Figure 4.10 shows the general procedure for a user to group objects.

The user starts with touching an object. To move the object and the surrounding bubble, the user slides the finger with the object underneath to the desired position. As the bubble partially touches another bubble, they melt. To depart an object from a bubble, the user again touches the desired object and slides over the surface. As soon as the object is sufficiently far from the bubble, the object get its own surrounding bubble.

![Figure 4.10: (1) Dragging an object. (2) Touching boundaries. (3) Melting of bubbles.](image)

**Implementation**

During the dragging operation, the system searches the nearest neighbor of an object. Therefore, the distances between the object \((o_i)\) and each candidate object \((o_j)\) is calculated. As the distance between two objects is lower than the \textit{threshold}\(^1\), objects get grouped in a bubble. Finally, old and new bubble were updated. If no neighboring object was found within the \textit{threshold} and the object is located out of its current surrounding bubble, a new bubble for the object is created. Algorithm 4.5 illustrates the whole process of grouping.

\(^1\text{The threshold is the maximum distance between two objects, without getting grouped.}\)
Figure 4.9: (A-4) Force Field Threshold has great influences to the number of bubbles for more objects. (B-4) Contour Threshold decides on the spline position in the potential field. (C-1) shows that a too low Contour Threshold Offset causes missing spline points. (D-3) A high Sampling Size results in an inaccurate visualization especially for bigger bubbles. (E-3) Min Distance and (F-1) Max Distance influences the number of points in the spline.

4.3.2 Transferring

A bubble is a kind of storage medium, which can be used to transfer one or more objects. Figure 4.11 demonstrates how this works.
Algorithm 4.5: Estimate closest bubble. Update all relevant bubbles.

1: \(o_i\) means original object, \(o_j\) means nearest neighbor object
2: \(\text{FINDNEARESTNEIGHBOUR}(o_i)\) \(\triangleright\) Searches for the nearest bubble of the object.
3: \(\text{for all candidate \(\in C(candidates)\)}\) \(\triangleright\) All except the current object.
4: \(\text{distance} \leftarrow \text{posOf}(o_i) - \text{posOf}(\text{candidate})\)
5: \(\text{if distance < threshold then}\)
6: \(o_j \leftarrow c\) \(\triangleright\) Sets the nearest neighbor of the object.
7: \(\text{end if}\)
8: \(\text{end for}\)
9: \(\text{return } o_j\)
10: \(\text{end}\)
11: \(\text{UPDATEBUBBLES}(o_i, o_j)\) \(\triangleright\) Updates bubbles, if object is added/removed.
12: \(\text{bubble}_i \leftarrow \text{bubbleOf}(o_i)\)
13: \(\text{bubble}_j \leftarrow \text{bubbleOf}(o_j)\)
14: \(\text{if } \text{bubble}_j = \text{null} \&\& \text{bubble}_i.\text{ChildrenCount} > 1 \text{ then}\) \(\triangleright\) The next object is too far away and the bubble has more than one object inside.
15: \(\text{bubble}_{new}.\text{Create}(o_i)\) \(\triangleright\) Creates a bubble with the child \(o_i\).
16: \(\text{end if}\)
17: \(\text{if } \text{bubble}_i \neq \text{bubble}_j \text{ then}\)
18: \(\text{bubble}_i.\text{Remove}(o_i)\) \(\triangleright\) Removes \(o_i\) from the old bubble.
19: \(\text{bubble}_j.\text{Add}(o_i)\) \(\triangleright\) Adds \(o_i\) to the new bubble.
20: \(\text{bubble}_i.\text{Update}()\) \(\triangleright\) Updates the old bubble.
21: \(\text{bubble}_j.\text{Update}()\) \(\triangleright\) Updates the new bubble.
22: \(\text{end if}\)
23: \(\text{end}\)

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure}
\caption{(a) Touching the bubble. (b) Dragging the bubble over objects. (c) Release bubble on the desired position. (d) Merging of bubbles.}
\end{figure}

The user has the option to transfer one or more objects to a position anywhere on the surface. In order to do this, he touches the bubble with one or more fingers and drags it to the desired position. As long as the bubble is moved, bubbles do not automatically melt. Only after the user takes away his fingers, bubbles merge.
**Implementation**

As we currently introduced, during a dragging operation, bubbles do not merge. As soon as a bubble stands still, it can merge with any other bubble. The procedure therefore can be found in Section 4.3.1.

### 4.3.3 Splitting

A hand-drawn free-form-stroke allows the user to split a bubble. Figure 4.12 illustrates the process for splitting a bubble.

![Figure 4.12](image)

**Figure 4.12:** (a) Pushing the scissor button. (b) Drawing a stroke. (c) Moving bubbles apart from each other. (d) Result.

The user pushes the scissor button on the bottom left corner of the interface and runs one finger across the desired bubble. As soon as the finger is taken away, the bubble splits. Finally, the two new bubbles move apart from each other.

**Implementation**

At the beginning, all points, describing the stroke path, were saved to identify if the stroke intersects any bubble or not. As soon as the stroke is finished, the system looks whether an object is on the (A) side or on the (B) side of the path. To do so, all objects in one bubble were fetched and the splitting path is separated into subparts, each defined by two points of the path. To decide to which side one object belongs, the system applies the preparation step of the Graham Scan\(^3\). Therefore, the differences between two rectangles, constructed from the points of the subpart \((P_0, P_1)\) and the center of the current object \((P_2)\) were calculated (see Equation 4.5). The sign of the expression

---

\(^2\)Note: Here, we avoid to say left and right side as this depends on the angle \((\alpha)\) of the splitting line (see Figure 4.13).

\(^3\)Graham Scan [4] is a method for calculating a convex hull.
\[ T(P_0, P_1, P_2) = (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0) = \begin{cases} 
< 0 & \text{if } P_2 \text{ on the one side from } P_0, P_1, \\
0 & \text{if } P_2 \text{ on } P_0, P_1 \\
< 0 & \text{if } P_2 \text{ on the other side from } P_0, P_1
\end{cases} \] (4.5)

**Algorithm 4.6:** Splits a bubble.

1. \( A \): objects \( a \) from the path, \( B \): objects \( b \) from the path
2. \( \text{SPLIT(splitPoints, } O) \) \( \triangleright \) Points of the splitting path, all objects.
3. \( \text{for } i \leftarrow 0, \text{numberOf(splitPoints)} \) \( \text{do} \)
   4. \( \text{for all } o \in O(\text{objects}) \) \( \text{do} \) \( \triangleright \) All Objects in the bubble.
   5. \( p_0 = \text{splitPoints}_i \)
   6. \( p_1 = \text{splitPoints}_{i+1} \)
   7. \( p_2 = \text{posOf}(o) \)
   8. \( \text{result} \leftarrow \text{Sign}((p_1.X - p_0.X) * (p_2.Y - p_0.Y) - (p_1.Y - p_0.Y) * (p_2.X - p_0.X)) \) \( \triangleright \) Estimates a sign.
   9. \( \text{if } \text{result} \leq 0 \text{ then} \)
   10. \( A.\text{Add}(o) \)
   11. \( \text{else} \)
   12. \( B.\text{Add}(o) \)
   13. \( \text{end if} \)
14. \( \text{end for} \)
15. \( i \leftarrow i + + \)
16. \( \text{end for} \)
17. \( \text{bubble}_O.\text{Remove}() \) \( \triangleright \) Removes original bubble.
18. \( \text{if } \text{count}(A) \neq \text{null} \text{ then} \) \( \triangleright \) Not empty.
19. \( \text{bubble}_A.\text{Create}(A) \) \( \triangleright \) Bubble for all \( a \) side objects.
20. \( \text{end if} \)
21. \( \text{if } \text{count}(B) \neq \text{null} \text{ then} \) \( \triangleright \) Not empty.
22. \( \text{bubble}_B.\text{Create}(B) \) \( \triangleright \) Bubble for all \( b \) side objects.
23. \( \text{end if} \)
24. \( \text{end} \)

Finally, to provide more visual feedback, bubbles move apart from each other. For this purpose, a line is defined between first and last point of the splitting path. Then, the angle \( \alpha \) is calculated to get the direction of the line. Next, an offset vector \( (\text{offsetVector}) \) is determined for the animations. This offset vector is turned around 90° and added to all positions of \( o \in A \) and subtracted from all positions of \( o \in B \). Figure 4.13 and Algorithm 4.7 introduce the procedure for doing this.

\(^4\text{Note: In our implementation, case } T(P_0, P_1, P_2) = 0 \text{ counts as } T(P_0, P_1, P_2) < 0.\)
Algorithm 4.7: Animates objects apart.

1: A objects on the a side of the path, B objects on the b side of the path
2: \text{ANIMATE}(A, B, offset) \quad \triangleright \text{Objects on the a side, objects on the b side, offset.}
3: \quad x \leftarrow p_0.X - p_{n-1}.X \quad \triangleright \text{Adjacent leg.}
4: \quad y \leftarrow p_0.Y - p_{n-1}.Y \quad \triangleright \text{Opposite leg.}
5: \quad \alpha \leftarrow \cot \frac{y}{x}
6: \quad \text{offsetVector}.X \leftarrow \cos \alpha \ast \text{offset}
7: \quad \text{offsetVector}.Y \leftarrow \sin \alpha \ast \text{offset}
8: \quad \text{rotate}(\text{offsetVector}, 90) \quad \triangleright \text{Rotates to normal position to splitting path.}
9: \quad \text{for all } a \in A \text{ do}
10: \quad \quad \text{animateTo}(\text{posOf}(a) + \text{offsetVector})
11: \quad \quad \text{end for}
12: \quad \text{for all } a \in B \text{ do}
13: \quad \quad \text{animateTo}(\text{posOf}(a) - \text{offsetVector})
14: \quad \quad \text{end for}
15: \quad \text{end}

4.3.4 Spreading

To avoid overlapping objects in a bubble, objects can fan apart inside a bubble. Figure 4.14 shows how it works for the user.

The user is able to spread superimposed objects through placing thumb and index finger directly on the bubble and pulling them apart. The objects animate to their new positions and the bubble adapts its surrounding contour. Afterwards, all objects stay in their new position and do not animate backward to their original position. The reason for that is that this technique should help to get an overview in a structuring process. The user may wants to move objects apart from the expanded version. If all objects move back to their original position, users might be confused as they are already one step further at that time.

Implementation

Starting with the original object centers, the system applies the Delaunay triangulation to find all neighbors of each object. For this purpose, an existing C# library was taken. To fit the application, some methods were adapted for allowing the usage of Visual Properties (see Section 2.2.5). Here, we do not mention the implementation and the algorithm, as a detailed description and some samples can be found with the original library\(^5\).

So, in the first step, Delaunay Triangulation is applied. As a result, triangles with one object on each vertex were received (see Figure 4.15-(1)). In the next step, every combination of two objects in each triangle is proofed for overlappings. If this is the case,

\(^5\)Originally developed by Paul Bourke (pbourke@swin.edu.au) as Fortran 77 Program. Converted to a standalone C# 2.0 library by Morten Nielsen (www.iter.dk). http://astronomy.swin.edu.au/~pbourke/terrain/triangulate/.
the difference vector \((\text{diffVector})\) between the first object \((o_i)\) and the second object \((o_j)\) is calculated and scaled up to 120%\(^6\) (see Figure 4.15-(2)). Then, the first object stays in the original position and the absolute position of the second object is shifted along the difference vector (see Figure 4.15-(3)). This procedure is repeated as long as both objects were overlapping.

Thereafter, the next two objects in the triangle were compared. As soon as all objects of one triangle are visually separated from each other, the next triangle is considered. While any position of one object in any triangle is updated, the process of updating begins anew for all triangles. To provide a smooth transition, each object animates smoothly from its old to its new position. The whole procedure of spreading is shown in Algorithm 4.8.

![Figure 4.15](image)

**Figure 4.15**: (1) Triangulation. (2) Scale difference vector. (3) Set position.

**Algorithm 4.8**: Spreads objects.

1. \(T\) are all triangles.
2. \(\text{SPREAD}(O)\) ▷ All objects.
3. \(T \leftarrow \text{Delaunay.Triangulate}(O)\)
4. \(\text{adapted} \leftarrow \text{true}\)
5. \(\text{while } \text{adapted} \text{ do}\)
6. \(\text{for all } t \in T \text{ do}\)
7. \(\text{ADAPT}(t.o_1, t.o_2)\)
8. \(\text{ADAPT}(t.o_2, t.o_3)\)
9. \(\text{ADAPT}(t.o_3, t.o_1)\)
10. \(\text{end for}\)
11. \(\text{end while}\)
12. \(\text{end}\) ▷ Objects to compare.
13. \(\text{while } \text{Intersecting}(o_i, o_j) \text{ do}\) ▷ Looks, whether objects were intersecting.
14. \(\text{diffVector} \leftarrow \text{posOf}(o_i) - \text{posOf}(o_j)\)
15. \(\text{scale}(\text{diffVector}, 1.2)\) ▷ Scale the vector up to 120%.
16. \(\text{posOf}(o_j) \leftarrow \text{posOf}(o_i) + \text{diffVector}\)
17. \(\text{adapted} \leftarrow \text{true}\)
18. \(\text{end while}\)
19. \(\text{return } \text{adapted}\)
20. \(\text{end}\)

\(^6\)120% as a smaller factor costs performance and a larger factor moves the position of the object too far away.
Chapter 5

Bin

This chapter describes the implementation of Bin. We introduce the roots of this interface and give a short introduction to the interaction techniques. For better explanation, algorithms were discussed, illustrated and enriched by visual representations. We do not focus on the general visual representation of a bin here, because the implementation has no specific algorithmic needs.

5.1 Idea

On the basis of Storage Bins [9], we developed a bin interface. A bin is a mobile, adjustable container, which allows to store and retrieve workspace content on a digital tabletop [8]. In opposite to real-world bins or containers, items were resized by moving them into the bin’s space to save storage space. Items can be easily added / removed as a group or individually by dragging, using a lasso or adjusting the bin’s shape. By moving a storage bin, all items inside will be transferred to the desired position.

A bin is a storage medium represented by an octagon. Objects can be added or removed to and from a bin. As an object is stored in a bin, it is resized to save space. Furthermore, a bin can be used to transfer objects right across the surface. To accommodate varying amounts of objects, a bin provides eight handles for changing the bin’s shape. Figure 5.1 shows a bin with eight handles and two objects inside.

5.2 Interaction Techniques

To understand, each of the following sections is structured as follows: First, we explain how the interaction technique works for the user through figures and a short description. Afterwards, we provide some implementation details in the form of algorithms and illustrate mathematical procedures with equations and figures.
5.2.1 Holing

To add an object to a bin, a user can hole it. Therefore, the user has two options.

![Diagram](image)

**Figure 5.2:** (1) Touching and dragging the object. (2) Releasing the object. (3) Resizing object in bin. (4) Kick-off object. (5) Object moves. (6) Holing object into bin.

The first option is touching and moving objects directly into a bin (see Figure 5.2-(1-3)). The second option is to use physicality. Therefore, an object is kicked off and begins to move. As it stands still, it gets holed (see Figure 5.2-(4-6)).

**Implementation**

Until the object \( (o) \) is released, the system permanently checks, if it belongs to any bin \( (\text{bin} \in Bins) \) on the landscape. Therefore, the system looks, whether any bin’s figure contains this specific object’s figure. If this is the case, the object is added to the new bin \( (\text{bin}) \) and removed from the old bin \( (\text{bin}_o) \). Moreover, to save space in the bin, the object is resized. Algorithm 5.1 shows how to check this.

**Algorithm 5.1:** Checks, to which bin one object belongs.

```plaintext
1: OBJECTINBins(o, Bins) ▷ Object and all Bins.
2: \text{bin}_o ← binOf(o) ▷ Saves the old bin.
3: \text{bin}_o.Remove(o) ▷ Removes the object from the old bin.
4: \text{bin}_o.Update() ▷ Updates the old bin.
5: for all \text{bin} ∈ Bins do ▷ All bins.
6: if \text{bin}.Contains(o) then ▷ Looks, if the visual bin contains the visual object.
7: \text{bin}.Add(o) ▷ Adds the object to the new bin.
8: \text{bin}.Update() ▷ Updates the new bin.
9: \text{o}.Resize() ▷ Resizes the object in the bin.
10: end if
11: end for
12: end
```
5.2.2 Collecting

Dragging and releasing a bin allows the user to collect items.

Figure 5.3: (1) Touching the bin. (2) Dragging the bin. (3) Releasing the bin. (4) Holing and resizing the object.

Therefore, the user first touches the bin. Then, he slides his finger to the desired object. As soon as the bin is released, all objects under the bin were collected.

Implementation

In contrast to the holing technique (see Section 5.2.1), the user can collect more than one item through dragging and releasing a bin over some objects. During the dragging operation, the system does not look, if any object underneath should be added to the bin. As the bin is released, it checks, whether any new object \((o)\) is inside the bin's \((bin)\) boundaries. If this is the case, the object is removed from the old bin \((bin_o)\) and added to the new one \((bin)\) even if the object is already in a bin underneath the new bin. Afterwards, both bins were updated. If the object was not inside a bin, the system resizes it. Algorithm 5.2 illustrates the procedure.

Algorithm 5.2: Checks, whether any object is new in a bin.

```
1:   ObjectsInBin(bin, O)                         ▷ Bin and all Objects.
2:    for all \(o \in O\) do
3:      if bin.Contains\((o)\) then ▷ Looks, if the visual bin contains the visual object.
4:        if binOf\((o)\) \(\neq\) null then ▷ Does \(o\) currently belong to a bin.
5:            bin_o \(\leftarrow\) binOf\((o)\) ▷ Saves the old bin.
6:            bin_o.Remove\((o)\) ▷ Removes the object from the old bin.
7:            bin_o.Update() ▷ Updates the old bin.
8:      else
9:          o.Resize() ▷ Adds the object to the new bin.
10:     end if
11:    end if
12:   bin.Add\((o)\) ▷ Updates the new bin.
13:   bin.Update() ▷ Updates the new bin.
14: end for
15: end
```
5.2.3 Adjusting

Another possibility to add an object to a bin is to adjust the bin through the manipulation of one of the eight handles.

![Diagram](image)

*Figure 5.4:* (1) Touching and dragging the handle. (2) Releasing the handle. (3) Resizing object in bin.

To adjust the bin, the user touches one of the handles with a finger and drags this handle to the desired position. As the handle is released by the user, the system checks, whether there is any new object directly under the bin. If this is the case, the object gets resized.

**Implementation**

When the handle is released, the system checks if any new object is inside the bin’s boundaries. This is the same process as for Collecting (see Section 5.2.2). Algorithm 5.2 demonstrates the procedure for doing this.

5.2.4 Transferring

To drag more objects, a selection tool allows to move several objects. Figure 5.5 shows how the user has to interact.

![Diagram](image)

*Figure 5.5:* (1) Pushing the lasso button. (2) Drawing a selection. (3) Touching and dragging the selection. (4) Releasing the selection. (5) Resizing objects and removing the selection.

First, the user pushes the lasso button on the bottom left corner. Afterwards, a selection can be drawn. Then, the user drags the selection to the desired position (e.g. into a bin).
As soon as the user takes away his fingers, the system checks all bins, if there are any new objects underneath it.

**Implementation**

After releasing the selection, the process is similarly to the collecting procedure (see Section 5.2.2). The only difference is, that the system checks every bin \((bin \in Bins)\) with all objects \((O)\) upon new containments (see Algorithm 5.3).

**Algorithm 5.3:** Checks, whether any bin has any new objects.

1: \textbf{ObjectsInBins}(Bins, \(O\)) \quad \triangleright \text{All Bins and all Objects.}
2: \textbf{for all} \(bin \in Bins\) \textbf{do}
3: \textbf{ObjectsInBin}(bin, \(O\)) \quad \triangleright \text{Checks the bin with all objects.}
4: \textbf{end for}
5: \textbf{end}

5.2.5 Spreading

To avoid overlapping objects in a bubble, objects can fan apart inside a bubble. The main difference is that the bin’s boundaries stay the same and not automatically adjust to the content. Figure 5.6 shows how this works for the user.

![Figure 5.6](image)

**Figure 5.6:** (1) Placing thumb and index finger on the bin. (2) Pulling them apart.

**Implementation**

The spreading of objects is already described in Section 4.3.4.
Chapter 6

Logging

For quantitative data analysis, data regarding used interaction techniques and single- or multi-finger input was logged. For this purpose, we decided to use the SLF framework. SLF (Simple Logging Facade)\(^1\) is a framework for easily plug in logging functionality. This framework is simple to integrate by using the SLF Facade Libraries. The SLF developer team provides a common interface, which allows to use different logging frameworks (e.g. log4net, EntLib, or NLog). Moreover, custom logging frameworks can be plugged-in easily. In the following, we introduce the usage of the SLF Framework exemplary by means of our Session Logger.

6.1 Implementation

The SLF.dll and SLF.Log4netFacade.dll were referenced in our Visual Studio Project Client_Blub.csproj to use the SLF Framework. For the logging procedure, two new loggers were defined. Therefore, a Session Logger is configured in the app.config (see Source Code Snippet 6.1). The named logger SessionLogger can now be used for gathering data. Source Code Snippet 6.2 shows the initialization of the logger. At the beginning of a session, the header is logged. During the session, a logging method is called for each occurring event logging all relevant values. Source Code Snippet 6.3 shows how a logging method can look like.

```xml
<configuration>
  <configSections>
    <section name="log4net" type="log4net.Config.Log4ConfigSection,log4net"/>
  </configSections>
  <log4net>
    <appender name="FileAppender" type="log4net.Appender.FileAppender">
      <file>session-log.txt</file>
    </appender>
    <appender name="RollingFileAppender" type="log4net.Appender.RollingFileAppender">
      <file>session-log.txt</file>
      <appendToFile>false</appendToFile>
      <rollingStyle>SizeBasedRotation</rollingStyle>
      <maxSizeBytes>1048576</maxSizeBytes>
      <maxBackupIndex>2</maxBackupIndex>
    </appender>
    <root>
      <level value="Info"/>
      <appender-ref ref="FileAppender"/>
      <appender-ref ref="RollingFileAppender"/>
    </root>
  </log4net>
</configuration>
```

\(^1\)http://slf.codeplex.com/
6. Logging


```java
<appender name="BasiliskAppender" type="log4j.Appender.FileAppender">
  <param name="File" value="log-session.csv" />
  <param name="AppenderRef.ref" value="BasiliskAppender" />
</appender>
</layout>
</configuration>
```


```java
private static string my = ";/4; // separator
// logging the header for the session file
public static void logSessionHeader()
{
  logItem = new LogItem();
  logItem.Timestamp = Time.Now.Nanoseconds;
  logItem.Message = "SessionId" + date + Interface + my + Task + my + Status + my + Operation + my + Comment1 + my + Comment2;
  sessionLogger.log(logItem);
  Console.WriteLine();
}
```

Source Code 6.3: Logging: Log header and events.

Finally, the logger's output is saved in a comma-separated file (*.csv). In table format, sample output of the SessionLogger is illustrated in Figure 6.1.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>UserID</th>
<th>Interface</th>
<th>Task</th>
<th>Status</th>
<th>Operation</th>
<th>Comment 1</th>
<th>Comment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:40:35</td>
<td>1 BLUB</td>
<td>GROUPING</td>
<td>STARTED</td>
<td>TASK</td>
<td>STARTED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08:40:45</td>
<td>1 BLUB</td>
<td>GROUPING</td>
<td>STARTED</td>
<td>GROUP</td>
<td>STARTED</td>
<td></td>
<td>OBJECT</td>
</tr>
<tr>
<td>08:41:49</td>
<td>1 BLUB</td>
<td>GROUPING</td>
<td>STARTED</td>
<td>SPLIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 6.1: Sample data of the Session Logger.
Chapter 7

Conclusion

This technical report covers the implementation of two concepts for group and object manipulation on digital tabletops. Blub is the approach to transfer Bubble Clusters [13] from mouse to touch input. Based on the Gestalt law of proximity [11], spatial aggregated objects were automatically recognized as groups. Additionally, a surrounding bubble provides more feedback to the users. The second concept was a kind of mobile, adjustable container, which allows to add or remove different objects (e.g. images, documents or thumbnails). Bin is based on Storage Bins [9] and has its roots in the container schemata [5]. The main goal was to find an interaction concept for structuring objects on digital tabletops, which meets the need of greater flexibility. Both concepts, Blub and Bin can be improved in a technical sense. Therefore, we provide some suggestions, in the following.

The heart of the this technical report is the visualization of a bubble (see Section 4.2). Drawing a surrounding iso-contour is a frequent topic in computer graphics. For static bubbles, the current algorithm is efficient enough. But as Blub allows to interact with bubbles and adapts bubbles permanently while the user drags an object (see Section 4.3.1), the calculation of the surrounding bubble is very expensive. Section 4.2.1 already discusses the performance issue and gives some advice how we solved the problem. At this point, we show some more ideas, how performance could be tuned less or more easily. First, pixel energy estimation (see Algorithm 4.2) could be parallelized in the bounding box. Therefore, the bounding box can be divided into subregions and then, pixel energy is calculated in parallel for the different regions. Second, sorting points according their distance to each other is currently too expensive (see Algorithm 4.4). To save time, the Graham Scan method [4] would be an alternative solution. Finally, mathematical operations such as the pixel energy estimation can be outsourced to the GPU, which is especially made for these mathematical operations. This would save a large amount of time.

A more algorithmic and graph drawing related problem is spreading (see Section 4.3.4). Actually, we use the Delaunay triangulation, which is the easiest way and works for simple equal-sized objects such as our rectangles good. But, as soon as objects have similar sizes, this implementation will not be satisfactory. Therefore, we suggest PRISM [2], which is a node overlap removal algorithm. Based on a proximity graph of the nodes in the original layout, PRISM provides proximity relations. According to the authors this algorithm is effective, efficient and provides the relations between nodes, which sounds promising.
Bibliography


