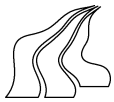
Designing an autostereoscopic display

Based on a study of three-dimensional languages

Lars Holm Jensen

May 29, 2006

Faculty of Engineering and Science
Aalborg University



Department of Computer Science

TITLE:

Designing an autostereoscopic display
Based on a study of three-dimensional languages

PROJECT PERIOD:

DAT6
February 1st 2006
- May 29th, 2006

PROJECT GROUP:

D624A

GROUP-MEMBERS:

Lars Holm Jensen

SUPERVISOR:

Lone Leth Thomsen

NUMBER OF COPIES: 5

NUMBER OF PAGES: 36

APPENDIX PAGES: 16

TOTAL PAGE NUMBER: 62

SYNOPSIS:

This project focuses on designing a 3D display that is suitable for use with three-dimensional programming languages. The distinct requirements of these are gathered through an analysis of three three-dimensional languages. The result is a design for a 3D display and an implementation of a simulator of the display.

Preface

This report is made by Lars Holm Jensen on the second Master Thesis semester at the Department of Computer Science, Aalborg University.
Lars Holm Jensen is associated with the research unit of Programming Technology. To comply to the Study Board Regulations a résumé of the report is available in Appendix B.
Basic mathematical skills and knowledge of general programming concepts are considered a prerequisite for reading this report.
A PDF-document of this report is available at

`http://www.larsholm.net/Publications/thesis.pdf`

with hyperlinked internal and external references.

Lars Holm Jensen

Contents

List of Figures

Chapter 1

Preanalysis

1.1 Motivation

This project focuses on designing a 3D display that is suitable for use with three-dimensional programming languages. The reasons for this are manifold.

In [9] it is argued that 3D models of software is superior to 2D models in communicating information and provides an effective method of layout for 3D UML diagrams. Furthermore [25] shows that a true 3D view is likewise superior to 2D representations of three-dimensional models.

A method for translating `Executable UML` (xUML) into code is described in [21]. When implemented this enables business analysts to compile xUML diagrams, modelling software, directly into executable code.

A declarative approach to transforming UML control flow diagrams into BPEL4WS is presented in [12], which enables business analysts to generate the BPEL4WS code necessary to implement business processes via webservices.

In [14] it is demonstrated that 3D UML diagrams of software systems with many hundreds of classes can be effectively visualised.

These arguments and new developments make it likely that programming in xUML in 3D might be the next development in the programming world. Even the paradigm of computing is shifting towards 3D [13]. The paradigms of computing have gone from electromechanical technology through relay, vacuum tube, transistor, and integrated circuit technology. According to

Kurzweil the sixth paradigm will be three-dimensional molecular computing. This will very likely affect the programming environment, both when designing the three-dimensional CPUs and the software that will run on them. In order to design a 3D display that is suitable for three-dimensional languages, this report investigates some of serious attempts to create these languages.

1.2 Problem Statement

This project investigates the field of three-dimensional programming languages, with focus on what they require of a 3D display in order to be beneficial to the field of programming in 3D. Based on this a weighty amount of focus is given to designing a 3D display that fulfils these requirements. The display will be *autostereoscopic*[1], as viewing artifacts seem to be a significant hindrance for widespread use. Further the project will compare the design to similar designs, and evaluate the design's implications on hardware and/or software requirements. One method to investigate the functionality of the design is to implement a display simulator.

1.3 Success Criteria

The main objective of the project is to design an autostereoscopic display based on a study of three-dimensional languages. To obtain this first the history of 3D programming languages is shortly reviewed, afterwards a few three-dimensional languages will be singled out, in order to determine if they have any distinctive requirements for an autostereoscopic display. Based on this a design of an autostereoscopic display is proposed. A theoretical proof of the design's effectiveness is given and the significance of the parameters of the design is examined. A simulator is implemented to give an impression of the functionality of the design. Next the project will give an account for the advantages and disadvantages of the design, in hardware/software terms as well as in a 3D programming context. Furthermore a comparison with existing 3D display designs will be given.

[1] Capable of providing a true 3D view without the use of artifacts, such as glasses

1.4 Delimitation

This project will not present optimisation techniques for rendering graphics on stereoscopic displays.

Chapter 2

Analysis

2.1 Introduction

This chapter briefly describes the history of three-dimensional programming languages and subsequently examines a few of the three-dimensional programming languages to determine if these have any distinct requirements for autostereoscopic displays. In order for this, it is here argued that a thorough understanding of these languages is needed. A thorough understanding of a language might help to conjecture typical use patterns, which ultimately would lead to an identification of any specific display requirements. Finally the state-of-the-art of 3D displays is reviewed and the findings are concluded upon.

2.2 Visual programming languages

Programming languages have been around for a long time. Longer even than computers some argue. However visual programming languages (VPLs), that is languages 'in which more than one dimension is used to convey semantics'[1], first began to take form around 1965, when William Sutherland of MIT created a visual executable dataflow language and later when David Canfield Smith made Pygmalion in 1975. Smith was trying to rid programming of the tedious compile-run-debug-edit cycles and instead introduces two new

[1] Encyclopedia of Electrical and Electronics Engineering

concepts; program by demonstration and by using icons. This resulted in an icon-based programming environment in which users demonstrate how a task is performed in a 'record mode', after which the program can operate.VPLs began to gain more momentum after Backus asked 'Can Programming Be Liberated from the von Neumann Style?' in 1978 [2]. Though not advocating for visual languages, the article questioned the one-dimensional word-at-a-time style of conventional programming languages and hereby sparked a search for new ways of expressing semantics. After this VPLs started to flourish, such as Prograph of 1982 inspired by functional languages and dataflow diagrams [6], and Pict/D of 1984 based on flow chart, and Blox of 1986 which pieced **while** and **if** statements together as a puzzle.

2.3 Three-dimensional programming languages

Three-dimensional visualisation in visual languages was used in GL (Geometry Language) in 1968. GL was aimed at solving the placement problem, when designing three-dimensional structures. Namely helping engineers finding spatial overlaps of objects at design time [5]. However GL was never implemented. There are other examples of multidimensional visual languages, for instance the `Structured Analysis and Design Technique` (SADT) language by Douglas T. Ross, which modelled systems comprising 'things', 'happenings' and their interrelations used for requirements definitions [8]. However SADT was only three-dimensional in the sense that two-dimensional diagrams could be layered on top of each other. Furthermore though GL and SADT where referred to as programming languages they both were meant for specifications rather than computation. In 1991 Marc Najork created Cube, which is the first attempt to make a general-purpose three-dimensional visual language [17]. Several followed later, such as Lingua Graphica, CAEL-3D, 3D BTTL, 3D-Visulan, ToonTalk, SAM and 3D-PP [3, 10, 11, 18, 22, 24, 26].

2.3.1 The Cube language

Cube is the first programming language with three-dimensional notation capable of computation. Cube is a higher order language based on dataflow and logic techniques, and as such shares many similarities with one-dimensional

languages such as Prolog, Lisp, and ML. These similarities include the extensive use of recursion and a declarative approach to problem solving. The following sections are based on [17] and [16] and will give an insight of the syntax of Cube.

Cube uses `cubes` as logic variables capable of holding both values and predicates. Cubes can have `ports`, which themselves are cubes, that correspond to arguments of the predicate. The ports can be connected by `pipes` that facilitate dataflow. In Figure 2.1 the use of the factorial function in Cube is shown.

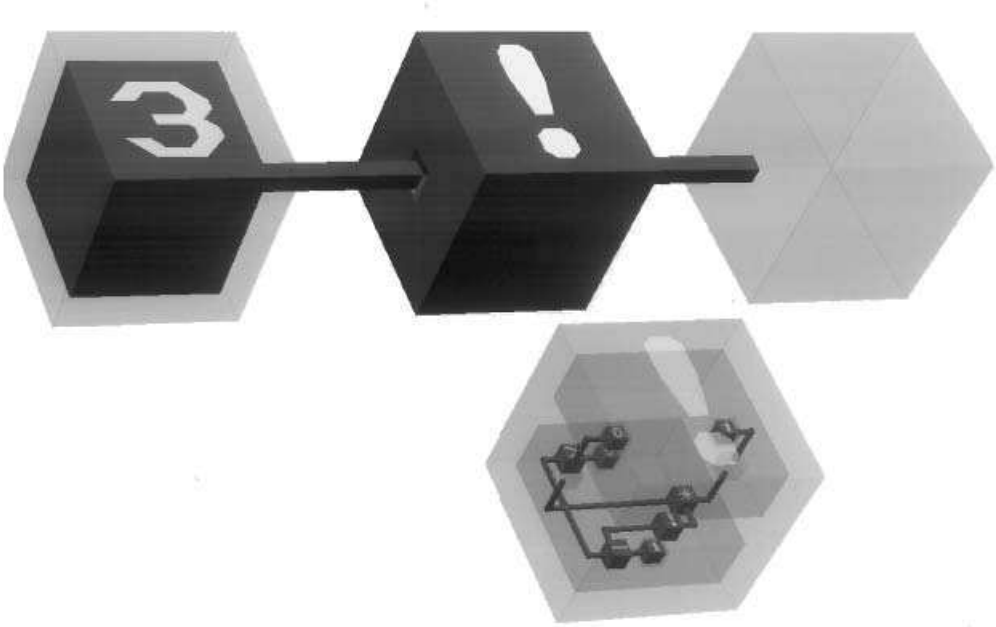


Figure 2.1: Using the factorial function in Cube

The dataflow in pipes in Cube has no direction. Data should be thought of as flowing through pipes in both directions or rather values of holder cubes are unified through pipes. For instance a holder cube containing the integer 1 connected to an empty holder cube will evaluate to the two holder

cubes containing the same value, in this case the integer 1. If the values are not unifiable dataflow will fail. This is related to another element in Cube, `planes`. Planes are boxes inside cubes that confine dataflow failures within themselves. Planes can be stacked vertically or horizontally, indicating disjunctions and conjunctions, respectively. This quality of confining dataflow failures is very powerful and enables *conditionality*. Now consider the close-up of the factorial function in Figure 2.2. Assume the value n is flowing in the left side port. If n is anything but 0 the upper plane will fail, because n and 0 are not unifiable. If n is 0, however, the upper plane will remain and the lower plane will fail due to the condition on the first left turn of the input pipe. This leaves the 1 of the upper plane to flow out the result port. If n is lower than 0 both planes will fail, thus causing the entire function to fail. On the other hand if n is greater than 0, the value of n will flow both to the multiplication cube and the negation cube. At the negation cube the value 1 flows in the right side port of the cube and the result $n-1$ will flow out the result port. Now $n-1$ will flow into the factorial cube and $(n-1)!$ will flow out. Finally the values n and $(n-1)!$ will be multiplied and flow out the result port of the function.

Cube has a static polymorphic type system and uses a variant of the Hindley-Milner type inference algorithm [7] that ensures that Cube programs are well-typed. The user can initiate type inference at design time, when this happens the Cube system will place a `type cube` within all empty holder cubes. Three types are native to Cube; integers, floating-point numbers and propositions. However Cube allows definitions of new types, such as characters, strings, lists and trees. This is done in the same manner as regular Cube coding, here `type definition cubes` and `type planes` are used instead. A thorough description of Cube can be found in [15].

2.3.2 The 3D-Visulan language

The 3D-Visulan language was created in an attempt to totally abandon the expression of programs in text and symbols. A 3D-Visulan program is expressed purely in three-dimensional bitmap of pixels. 3D-Visulan is a very simple language. It is rewrite-rule-based and has very few programming con-

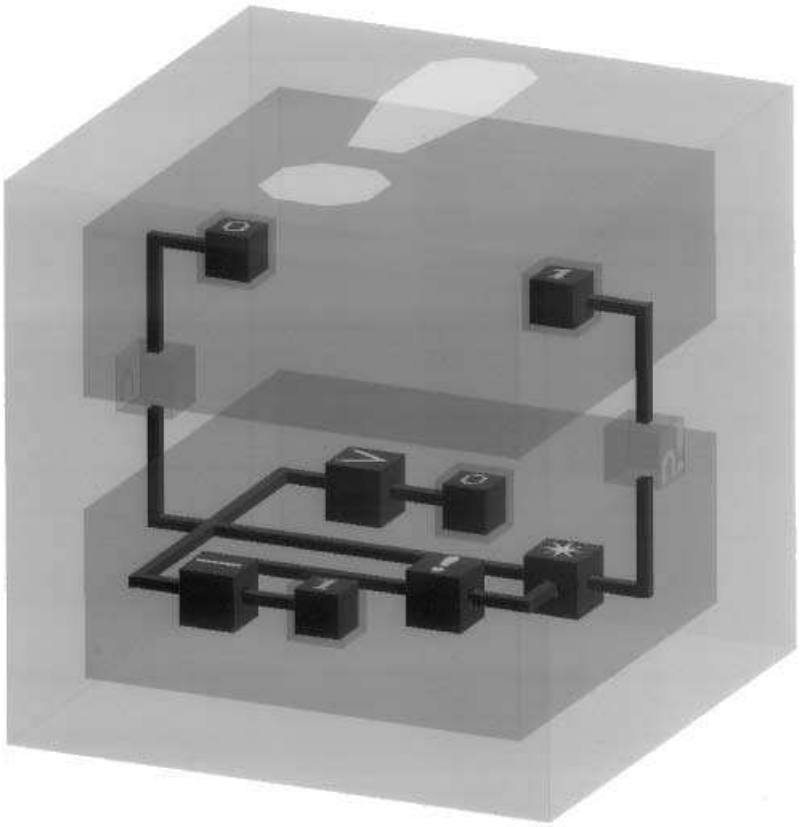


Figure 2.2: Factorial function in Cube

structs. There are `rules`, `definitions` and `data`. The programming constructs are recognised by having a rectangular base and special signature rises, like walls or bars. Figure 2.3 shows a program in 3D-Visulan. When executed the white X-figures and O-figures move continuously left and right between the dark walls.

The data in 3D-Visulan is located in a data-world, in Figure 2.3 the top-right platform, recognised by two vertical bars in each upper corner. All execution takes place here. The data-world is a three-dimensional array of pixels, which can have different colours. There are four composite rules the program in Figure 2.3 and one definition. The definition is the construct at the bottom of the picture. A rule can consist of any number of rules, connected by a single pixel, and has priority in accordance with its location. Rules are recognised by the T-shaped wall, dividing the before-world and the after-world. If the before-world, to the left of the T-wall, is matched in the data-world, the matching data is replaced by the data in the after-world of the rule. In connected rules all before-worlds must be matched in the data-world, before replacement can take place. All replacements take place simultaneously. The definition construct, closest in the picture, defines that X-figures and O-figures can be matched by a white square in any rule. The upper left rule states that whenever a white figure touches a line of dark pixels on the left, the arrow should change from pointing left to pointing right. The rule just next to states the opposite. The two rules below, state that any white figure should be moved one step sideways in the direction the arrow is pointing. The result during execution is that the white figures keep moving as a group, left and right between the dark walls.

This simple language of data (states) and substitution rules appears like cellular automata or other finite state machines, but that is only appearance. The modest possibility of the conjunction of rules makes 3D-Visulan computationally equivalent to Turing-machines, which means that any problem that any computer can solve can be solved by a 3D-Visulan program.

Figure 2.4 shows an implementation of a Turing-machine in 3D-Visulan.

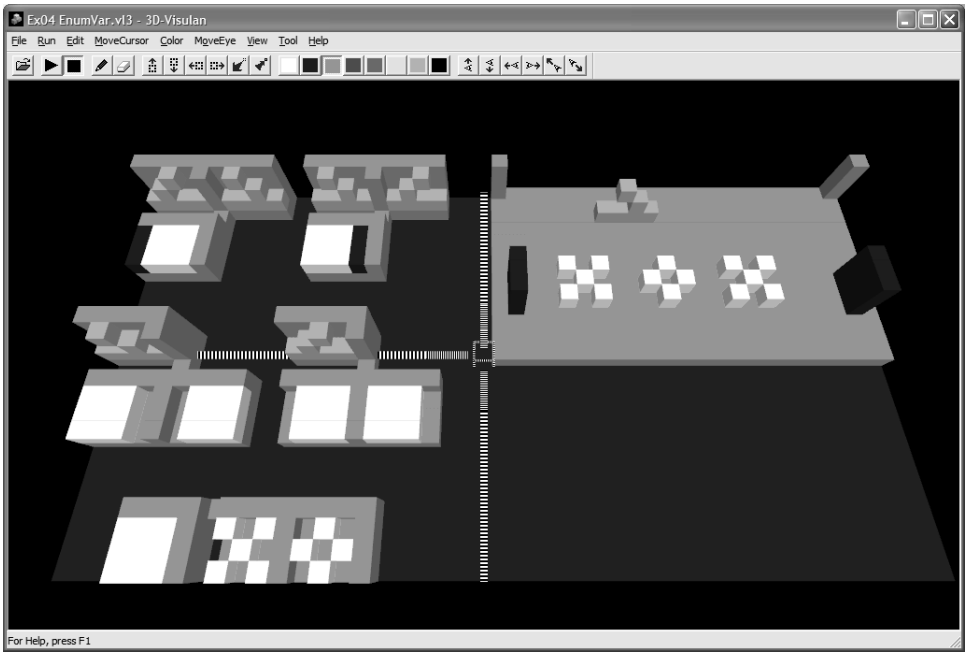


Figure 2.3: Program moving the shapes continuously left and right

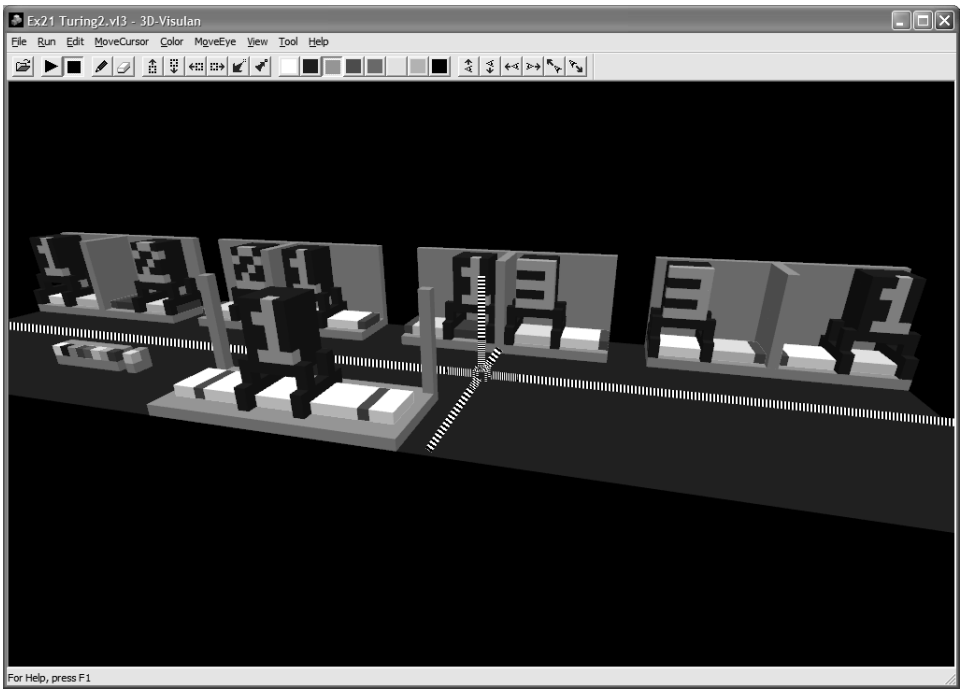


Figure 2.4: Implementation of a Turing Machine in 3D-Visulan

2.3.3 The 3D-PP language

The 3D-PP language was created in an effort to improve the use of `screen real estate`[2]. Through the years this was identified as a serious problem for visual programming languages in particular. Pictorial elements seemed to take up much more space than text. 3D-PP tries to address this with two ideas. Base the language entirely on a known and compact programming language, and simply use miniature pictorial elements, shown in Figure 2.5.

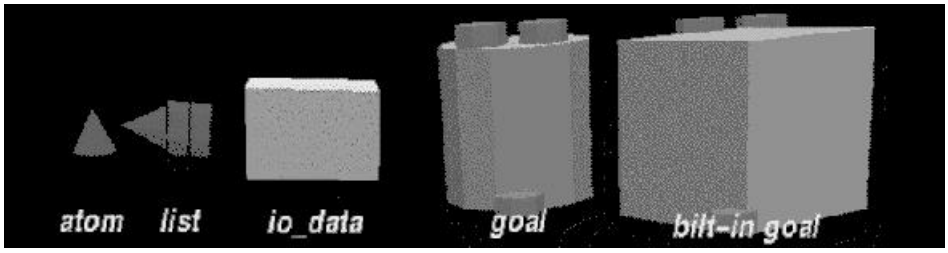


Figure 2.5: Pictorial elements in 3D-PP

The language chosen was Haskell, a declarative, polymorphically typed, lazy, purely functional language based on lambda calculus. This closely resembles the characteristics of Cube and this is no coincident. Declarative languages seem to require less programming elements than imperative languages, as in general the problem to solve is often easier to describe than the solution to the problem. So this is a convenient choice for a visual programming language. A clause in 3D-PP is composed in the same manner as a clause in Haskell:

```
predicates(arguments, ... ) :- guard | body.
```

The predicates are represented by a parent `Goal`-element and the guards and the bodies as child `Goal`-elements inside the parent. The use of miniature programming elements to support large programs is not without consequences. Such an interface is subject to unmanageable and confusing code, to counter this 3D-PP employs a `direct manipulation`[3]-style interaction as

[2]The amount of screen space available to a program

[3]Wikipedia: »Direct manipulation is a human-computer interaction style that ... involves continuous representation of objects of interest, and rapid, reversible, incremental actions and feedback«

defined by [23]. This is among other things manifested in the development of a new drag-and-drop technique specially suited for use in three-dimensional spaces.

Figure 2.6 shows a program calculating the 1000th prime in 3D-PP. As opposed to Cube, 3D-PP does not to expose the same level of details regarding the nesting of programming elements.

2.4 Sub-conclusion

The study of the three-dimensional languages has ascertained that such languages need not be similar, neither in appearance nor in behaviour. However all the three studied languages have connecting constructs, which are crucial to execution, and which require a certain level of details in a display. All languages have elements, which are sensitive to their location in 3D-space. This seems to enforce the requirement of a good depth perception in an autostereoscopic display. The perception of the placement and connections of the three-dimensional objects would benefit from features that enable the programmer to look around objects in autostereoscopic displays. Finally if these languages are to be used in common work environments, a display that supports multiple observers seems advantageous, likewise that the display is compact, for instance incorporable in a laptop, would be favourable.

2.5 3D displays

In this section the principles different autostereoscopic displays are described. These are the parallactic and volumetric displays. These are chosen because they constitute the far greater part of commercial autostereoscopic displays.

2.5.1 Parallactic displays

The term parallactic displays is commonly used to describe autostereoscopic displays which have discrete viewing zones, although this is not etymologically correct. A discrete viewing zone is one in which all viewpoints receive the same image. Parallactic displays can be based on different technologies, for instance lenticular and diffractional displays.

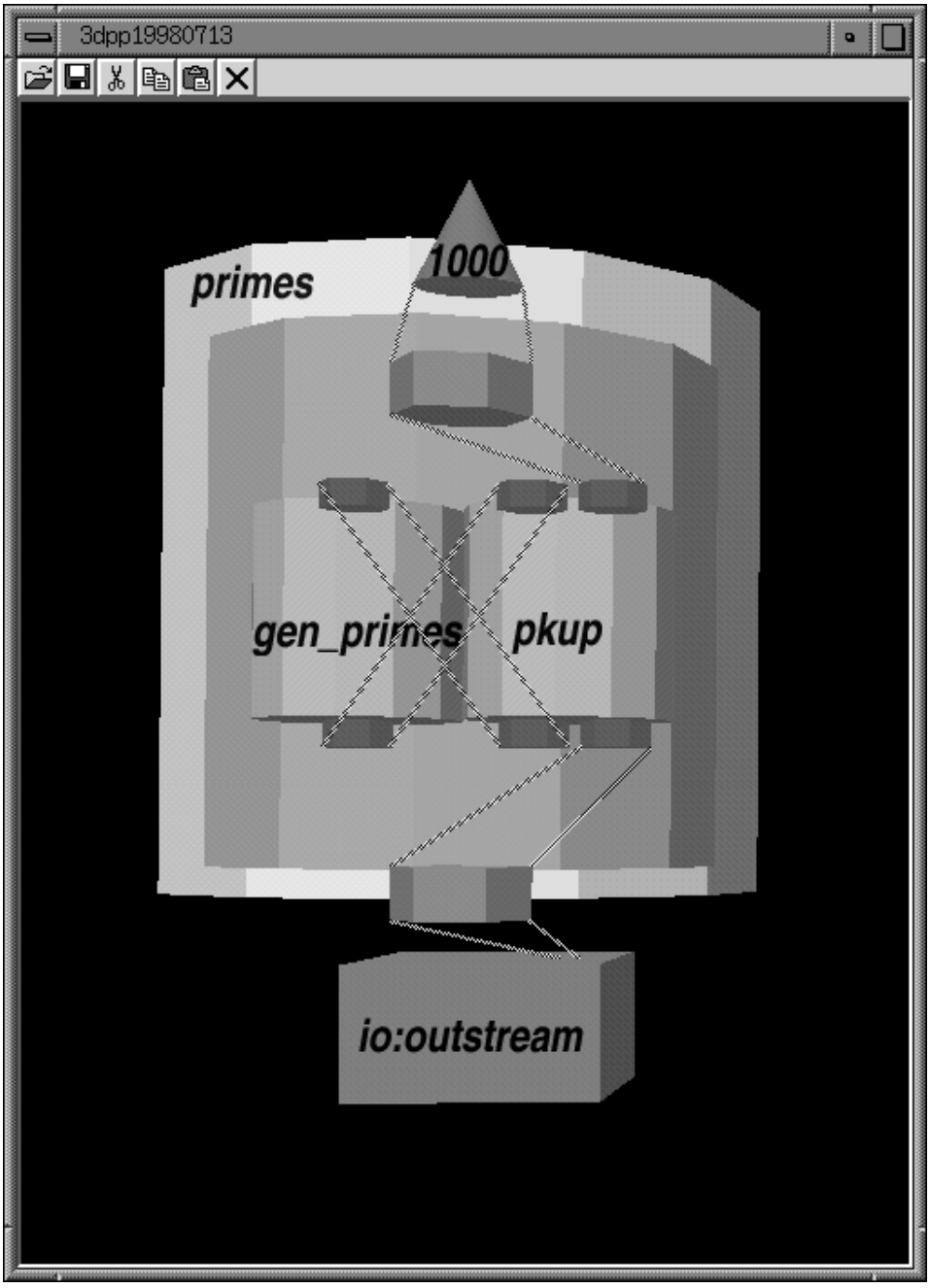


Figure 2.6: Program generating the 1000th prime in 3D-PP

Lenticular displays work by applying a film of lenses in front of or behind the layer of pixels in an ordinary liquid crystal display. The film usually consists of vertical series of cylindrical lenses that disperse light according to the location of the light source behind the film. Figure 2.7 illustrates the principle. In this manner the individual vertical pixel line becomes part of a distinct viewing zone. The lenticular film is fashioned such that the observer's eyes receive two different images at the appropriate viewing position. [4]

One such lenticular display is Philips 42-3D6C01 display

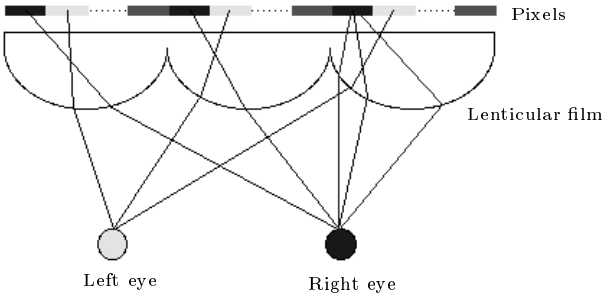


Figure 2.7: The principle of a lenticular display

The diffractional displays create discrete viewing zones by diffraction. Diffraction is the bending or redirection of waves that occurs when waves meet any obstruction or gap. This effect is utilised in a special film or sheet, which is then placed between the liquid crystals and their backlighter in a liquid crystal display.

Ultimately both types function in very much the same manner and their principles facilitate the same features in 3D displays. They are horizontal look-around capabilities (horizontal parallax), 2D/3D-switchable, and support for multiple observers. The disadvantages for both are a low level of details. The allocation of each vertical line of pixels to a separate viewing zone decreases the horizontal resolution proportionally. For instance a lenticular or diffractional display with 10 viewing zones based on a LCD with a horizontal resolution of 1600 pixels will have a horizontal resolution of 160 pixels. This is less, than most contemporary mobile telephones, which often has the resolution QVGA (Quarter Video Graphics Array) to support the popular podcasts. Optionally the lenses can be slanted, which distributes the

resolution loss evenly.

2.5.2 Volumetric displays

The field of volumetric displays is filled with different technologies. There are among others swept volume, emissive volume, laser-based, and layered LCD displays.

The swept volume display, in some cases referred to as swept surface display, functions by spinning a translucent screen inside a globe. The screen itself may emit light or light is projected upon it. In this fashion a three-dimensional image is perceived.

The emissive volume displays rely on a translucent light-emitting substance in which points can be individually addressed. So this is a rather straightforward approach to creating 3D images.

There exist various different laser-based 3D display systems. Most depend on some kind of substance, in which the intersection or focal point of laser beams trigger illuminance. This substance could be air, as demonstrated by [1], or dust illuminated by a visible laser [19], however not all laser-based displays are based on illuminating substances. Computer generated holographics use laser in combination with a diffractional screen to create the same wave front of light as a real object would.

The last of the volumetric displays described here are the layered LCDs. They simply consist of several layers of transparent liquid crystal displays, which can then produce images of three-dimensional object within the layers.

The display types discussed here all have look-around capabilities and support for multiple observers. Some have a low level of details, such as the emissive volume and laser-substance-based displays, while others are comparable to present 2D displays. The disadvantage of all of them is their volume. None of the displays discussed here are flat. Some have nuisances such as background noise and a few of them are not entirely without hazards, such as the laser plasma display by [1] that would cause serious burns if anybody were to touch the plasma flashpoints.

2.6 Sub-conclusion

In Table 2.1 the characteristics of the examined displays have been assembled. Full parallax in the first column means that one can look around objects both horizontally and vertically. Detail level is divided into four categories; poor, low, normal, and high, in ascending order, normal being comparable to the present level of details in 2D displays. The rightmost column indicates whether the display can be switched into 2D viewing. Two fields are marked N/A. Laser-substance in the Noisy-column as there exist both laser-substance display, which are silent and noisy, and Layered LCD in the Flat-column, as the display could be made flat, but that would limit the level of depths portrayable in the display.

	Parallax	Detail level	Depth	Flat	Mechanical	Noisy	2D
Lenticular	Horizontal	Low	Unlimited	✓	×	×	✓
Diffractional	Horizontal	Low	Unlimited	✓	×	×	✓
Swept volume	Full	Normal	Limited	×	✓	✓	×
Emissive volume	Full	Poor	Limited	×	×	×	×
Laser-substance	Full	Poor	Limited	×	×	N/A	×
Holographic	Full	High	Unlimited	×	×	×	✓
Layered LCD	Full	Normal	Limited	N/A	×	×	✓

Table 2.1: Comparison of the described displays

When reviewing the requirements for three-dimensional languages in autostereoscopic displays in Section 2.4, it is clear that none of the examined displays fulfils all the requirements. A ranking of the most suitable displays will naturally be depending on a subjective evaluation, however it seems that lenticular, diffractional or holographic displays are the most apt in a working environment, due to their accumulated qualities and depending on whether they should replace a workstation monitor or a laptop display. The holographic displays can only replace workstation monitors as the display cannot be flat, due to the laser cannon creating the holographic images.

Chapter 3

Design

3.1 Introduction

In this chapter a design for an autostereoscopic display will be presented. The design is based on the one presented in [20] by Ken Perlin. The proposed design generalises Perlin's concept from one viewer to several observers. This, unfortunately, results in an increased sensitivity to factors such as position, viewing angle and the tilting of the user's head. This sensitivity, however, is not greater than currently commercialised autostereoscopic displays. The advantages of Perlin's and this design are noticeable. The properties of these are to a greater degree controllable by software. This means that the characteristics, such as the width of the viewing angle and the quality of the display image, can be suited to fit the surroundings and the computing power available. Moreover the display can easily be used as a normal 2D display, just by clearing the front display.

3.2 Concept

In order for the illusion of 3D to be created all that has to be achieved is the ability to provide each eye of the observer with two different images. This is obtained by placing a display, which can be switched between opaque and transparent, in front of a regular image producing display. On the front display a striped pattern is created. Now an observers eyes would see through the transparent gaps between the opaque bars and see two different vertical

lines of the display in the back. This effect is utilised to continuously feed two separate images to each eye. In Figure 3.1 the concept of the design is illustrated. The figure shows two heads and the front and background displays. The figure also shows which parts of the background display that can be seen from each eye through the gaps in the front display.

However the pattern on the front display has to move for the observer to be able to see vertical lines of the entire display in the back. So the pattern is continuously and very rapidly shifted sideways. This is like the electron cannon of a television creating the impression of a still picture, while still generating different images in different directions within a restricted area in front of the display. This area will subsequently be referred to as the `effective zone` and its location and measurements are defined by the various parameters of the design.

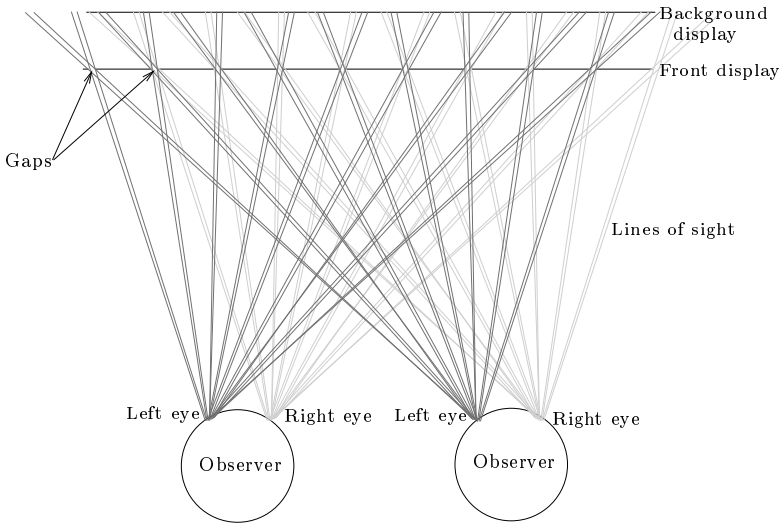


Figure 3.1: Illustration of the design concept

3.3 Model

The effective zone and its reach are crucial to the display's value as a commercial product, both in terms of quality and market range. In order to calculate the effective zone and overall reason about the design a model is

needed. In this section such a model is presented.

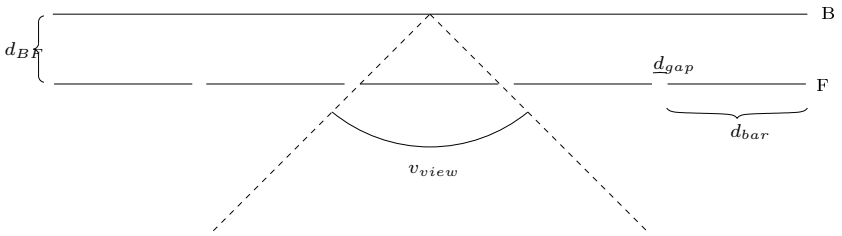


Figure 3.2: Illustration of the design model

In Figure 3.2 a rudimentary model is illustrated. Here the line segment of the background display is denoted **B**, the line segment of the front display **F**, the distance between them d_{BF}, the gap width d_{gap}, the bar width d_{bar} and the angle within which all viewing directions corresponds to one and only one point on **B** is denoted v_{view}.

3.4 Delimiting the effective zone

In this section the rudimentary model is used to delimit the effective zone. In order to calculate the range a more precise definition of the effective zone is needed.

Definition 1. The effective zone is the area in front of the display, in which it is possible to place two points, p_{left} and p_{right}, from where two points, $p_{left,B}$ and $p_{right,B}$, on **B** can be seen, where

- the distance between p_{left} and p_{right} is d_{eyes}
- none of the point are identical,
- none of the points are collinear.
- $p_{left,B}$ and $p_{right,B}$ may be no closer than d_{pixel}, the width of the pixels of the display

□

With this definition it is straightforward to show that the effective zone exists and is within v_{view} of all points on $\mathbf{B}$. To calculate v_{view} we need no further information than the given. If v_{view} is the top angle in an `isosceles`[1] triangle with base d_{bar} and altitude d_{BF} then

$$v_{view} = \pi - 2\arctan(\frac{2d_{BF}}{d_{bar}})$$

by simple trigonometry.

The point on the border of the effective zone nearest to the display is the top point of the isosceles triangle with base $\mathbf{B}$ and top angle v_{view} and which is in front of the display. This point, p_{start}, is shown in Figure 3.3. The distance between p_{start} and $\mathbf{B}$ is

$$d_{min} = \frac{|\mathbf{B}|d_{BF}}{d_{bar}}$$

since corresponding parts of similar triangles are proportional.

The effective zone does not extend infinitely away from the display. There exists a distance from where p_{left} and p_{right} cannot be placed without seeing the same pixel on $\mathbf{B}$. In other words, where $p_{left,B}$ and $p_{right,B}$ is closer than d_{pixel}. Though the curve delimiting the effective zone is irregular, the shortest distance between this curve and p_{start} is

$$d_{max} = \frac{d_{BF}d_{eyes}}{d_{pixel}}$$

by proportionality.

In this section a model has been presented for the design of an autostereoscopic display and several significant parameters have been identified that have impact on the qualities of the display. Among these the most influential are the distance between the two displays d_{BF} and the bar width d_{bar}.

3.5 Parameters

In this section the parameters identified in the previous section are considered in a hardware context. Explicit values are designated based on a trade-off analysis and present hardware capabilities.

[1] An isosceles triangle is a triangle with (at least) two equal sides

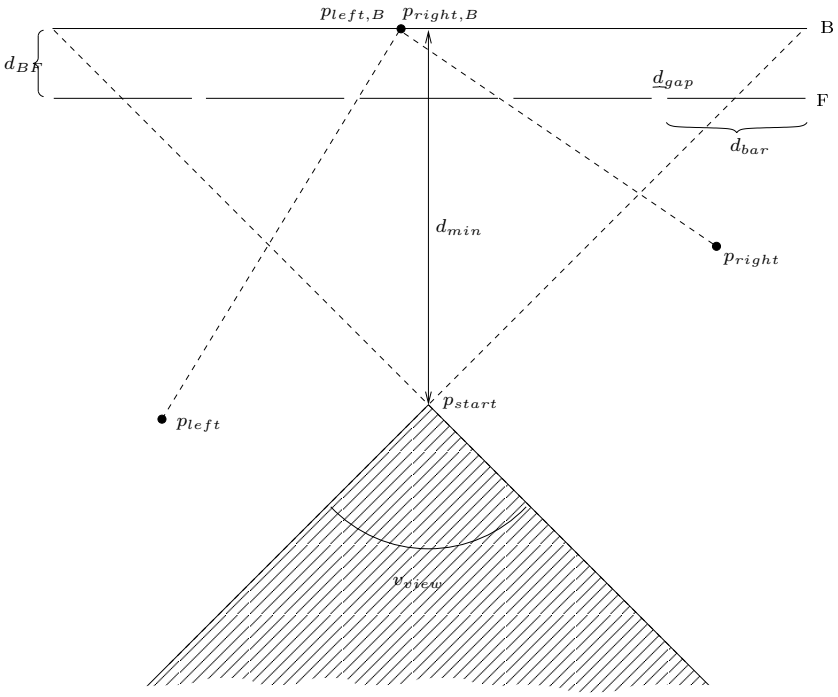


Figure 3.3: Illustration of the effective zone

3.5.1 The d_{gap} parameter

The d_{gap} parameter affects neither d_{min} and d_{max}, nor v_{view}. Nonetheless it does present a role for the display. A low value for d_{gap} yields a higher number of discrete viewing zones and a higher image quality, so if the front and background display have the same pixel density then d_{gap} should be 1 pixel width.

3.5.2 The d_{BF} parameter

The d_{BF} parameter is the most influential parameter of the proposed design. As Section 3.4 shows this parameter partakes in both the determination of d_{min} and d_{max}, as well as v_{view}. A low value for d_{BF} enhances both v_{view} and d_{min}, as they both increase. This makes the display viewable from a closer range and a greater angle. However a low value for d_{BF} also decreases d_{max} rendering the display ineffective at greater distances. Furthermore d_{BF} in combination with d_{pixel} limits the number of discrete viewing zones, which needs to be high for the display to be effective. The present commercial pixel width seems to be 0.2 millimetres, thus d_{BF} should be at least the same size and with displays intended for greater audiences, thus greater viewing distances, d_{BF} should be 2 millimetres or more.

3.5.3 The d_{bar} parameter

The d_{bar} parameter is present in both the determination of v_{view} and d_{min}. A high value of d_{bar} results in a higher v_{view}, but a lower d_{min}. So there exists a trade-off between a wide viewing angle and at how close a distance the display should be effective. Additionally the d_{bar} parameter is limited by the desirable display update frequency and brightness. This is due to the fact that the d_{gap}-d_{bar} ratio determines the number of cycles the display has to run through in order to render one complete image. This in turn reduces both the frequency and the brightness of the display proportionally. The choice of the value of the d_{bar} parameter is a matter of hardware capabilities, price and image quality. Nevertheless a d_{bar} value less than 1 pixel width would be ineffective, due to the fact that the display has to be able to display a different image to each eye. The d_{bar} value is limited upwards first and

foremost by computer rendering power and bandwidth limits. Consider a plausible setting of a 1280×1024 resolution display with 14 viewing zones operating at 60 Hz with 24 bit colours. In this setting the computer would have to render the 1280×1024 3D images at 840 frames per second and send the data at

$$1280 \times 1024 \times 24 \times 60 \times 14 \approx 26.4\text{Gbit/s}$$

to the display. This example shows the violent increase in bandwidth and computational demands these displays have.

3.6 Architecture of the simulator

This section describes how a simulator of the proposed design might be instantiated. This means that the algorithms described subsequently are not to be viewed as a part of the proposed design, rather as one way of using the proposed design to create a stereoscopic view. Of the same reason, and as it is not the main objective of this project, the description will be brief. Figure 3.4 shows a class diagram of the simulator. The diagram is not exhaustive, for instance the typical `Get`/`Set`-methods for variables are not depicted. The simulator consists of a main class `Simulator`, the classes `Front`, `Back`, `Head`, and `Canvas`. The main class `Simulator` is responsible for instantiating the other objects and drawing them on the graphical user interface. The class `Front` models the front display with the pinstriped pattern. The class `Back` models the background display. The class `Head` models the head and eyes of the observer. Finally the `Canvas`-class models the bitmaps on which all images are rendered. The interesting parts of the simulator that are not described in the proposed design, are the **Render**- and the **View**-methods. They are responsible for rendering the images of the background display and the eyes of the observer respectively. The rendered images of the background display are in the figure referred to as *slides*. They are rendered from the precompiled *shots* of a three-dimensional object viewed from different angles. The rendered images of the eyes of the observer are simply referred to as *eyes* in the `Head`-class.

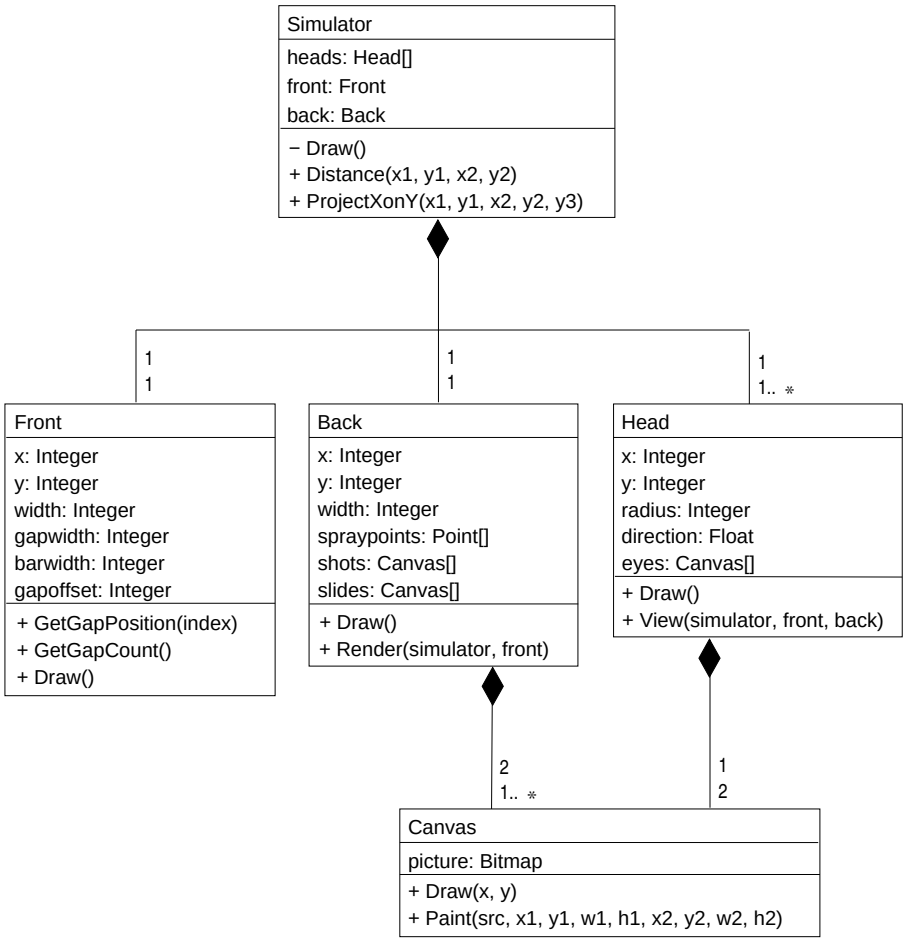


Figure 3.4: Class diagram of the simulator

3.6.1 The Render-method

The **Render**-method is, as mentioned, responsible for rendering the images of the background display. It is based on what you might call the idea of *»what you put in is what you get out«*. Figure 3.5 shows the pseudo-code for the method.

```
1  Create an array of spraypoints in a segmental arch around the centre of the display;
2  Create an array of slides to paint on;
3  For every slide in slides{
4       Enumerate t over spraypoints{
5           For every gap in Front{
6               xleft = the far left point on Back seen from spraypoint[t] through gap;
7               xright = the far right point on Back seen from spraypoint[t] through gap;
8               Paint the vertical stripe between xleft and xright of shot[t] on slide;
9           }
10      }
11      Increase Front.gapoffset by Front.gapwidth;
12 }
```

Figure 3.5: Pseudo code for the **Render**-method

Essentially this algorithm paints what is supposed to be seen of an image from a specific angle on the area on the background display, which can be seen through a gap from a specific point. This way one sees the depicted object from exactly the same angle as one is viewing when looking at the display.

3.6.2 The View-method

The **View**-method renders the image seen from two specific locations, namely the locations of the eyes of the observer. This is done very much in the same manner as the **Render**-method paints, just in this case the graphical data is flowing in the opposite direction. The pseudo code for the **View**-method is shown in Figure 3.6.

```
1  Create two canvases, leftview and rightview, to paint on;
2  For every slide in slides{
3      For every gap in Front{
4          xleft = the far left point on Back seen from lefteye through gap;
5          xright = the far right point on Back seen from lefteye through gap;
6          Paint the vertical stripe between xleft and xright of slide on leftview;
7      }
8      For every gap in Front{
9          xleft = the far left point on Back seen from righteye through gap;
10         xright = the far right point on Back seen from righteye through gap;
11         Paint the vertical stripe between xleft and xright of slide on rightview;
12     }
13     Increase Front.gapoffset by Front.gapwidth;
14 }
```

Figure 3.6: Pseudo code for the **View**-method

Chapter 4

Implementation

4.1 Introduction

In the section the choice of programming language for implementing the simulator is presented and argued for, next the implementation of the simulator of the proposed design is described and the most interesting code parts are reviewed and explained.

4.1.1 The choice of programming language

The programming languages considered to implement the simulator were the three-dimensional programming languages presented, Java, C# and Visual Basic, and the respective versions of these. The choice was based on requirements such as the ability to program tasks both object-oriented and sequential imperative in nature, bitmap handling capabilities and the time it takes to implement a working prototype. Of these languages Visual Basic 6 (VB6) was chosen. The three-dimensional programming languages presented in this report have no newly developed programming tools, indeed any suitable development environments. Java and C# fulfil the required qualities, but to a slightly lesser degree than VB6. VB6, as opposed to VB.Net, is not truly object-oriented, rather it is a sequential imperative language with added on object-oriented constructs. This makes VB6 very suitable for this specific task.

4.1.2 Implementing the proposed design

The simulator was implemented very much in accordance with the class diagram in Figure 3.4. The class `Simulator` was implemented using the Win32 ThunderFormDC class simply called Form in VB6. The source code for `Simulator` is included in Appendix A.1. The `Back`-class was implemented using the `Line`-class, therefore the **Render**-method and other code segments were moved to `Simulator`. The `Front`- and `Head`-classes were implemented as regular classes and the source codes are included in Appendix A.3 and A.2 respectively, notice that the **View**-method was moved to `Simulator`. The `Canvas`-class was implemented using PictureBox controls.

Implementing the Render-method

The **Render**-method starts at line 22 in `Simulator` and ends at line 86. The lines 23 through 30 are variable declarations and the lines 32 through 38 are the calculations of the used parameters. The lines 40 to 50 declare and calculate spray points. The shots used to render the slides are loaded in the lines 52 to 57 and the empty slides themselves are created in the lines 59 to 62, however the lines 66 through 84 is the code most similar the pseudo code of the **Render**-method in Figure 3.5. For instance are the pseudo code lines

```
6  xleft = the far left point on Back seen from spraypoint[t] through gap;
7  xright = the far right point on Back seen from spraypoint[t] through gap;
8  Paint the vertical stripe between xleft and xright of shot[t] on slide;
```

directly implemented in the lines

```
71 x1 = projectXonY(spraypoints(t).X, spraypoints(t).Y, _
   myFront.Gappos(i).X - myFront.Gap * 0.499) - myFront.X
72 x2 = projectXonY(spraypoints(t).X, spraypoints(t).Y, _
   myFront.Gappos(i).X + myFront.Gap * 0.499) - myFront.X
74 picSlides(s).PaintPicture picShots(t).Image, x1, 0, _
   x2 - x1, .ScaleHeight, x1, 0, x2 - x1
```

of the `Simulator`-class. The `projextXonY`-function calculates the intersection between the line going through two points and the line of the background display. In line 74 the viewable vertical stripe of `picShots(t)` is

painted on `picSlides(s)`. These slides are the ones used when reconstructing the image from a specific angle in the **View**-method.

Implementing the View-method

The **View**-method begins at line 103 and ends at line 176 in `Simulator`. The lines 105 through 112 are variable declarations and the lines 114 through 121 are the calculations of the used parameters. The lines 136 to 143 loads the saved slides from the **Render**-method into memory. Again there is a part, which implements the pseudo code of the **View**-method in Figure 3.6, this is the part from line 152 to line 174. Notice the lines 4, 5, 6, and 9, 10, 11 of the pseudo code. They are implemented in the lines 160, 161, and 164 in the `Simulator`-class. When the loops have been run through, two canvases have been painted over and are now showing the view from the two eyes.

4.2 Experiments and results

The three-dimensional object used during experiments in the simulator is a graph, from a three-dimensional graph viewer, viewed from different angles. The graph viewer was developed previous to the writing of this report and was integrated in the simulator. This particular object was chosen due to its high level of details and yet having a simple and recognisable shape; the parabola on one axis and the sine curve on the other. With these properties rendering errors and imprecisions are easily spotted. Figure 4.1 shows one of the shots of the graph.

The **Render**-method renders these shots into slides. Figure 4.2 shows one such slide. Remember these slides are not shown directly to the observers. These slides are seen through the gaps of the front display and there is one slide per gap offset. Each time the gap pattern is shifted one gap width sideways, a new slide is shown on the background display.

When the gap pattern has gone through one cycle, every observer in front of the display has received an entire and unique view of the display. Figure 4.3 shows a rendered view from one specific point in front of the display. Notice that the graph is somewhat deteriorated. There are several reasons for this. One reason is that the simulator works in a discrete environment,

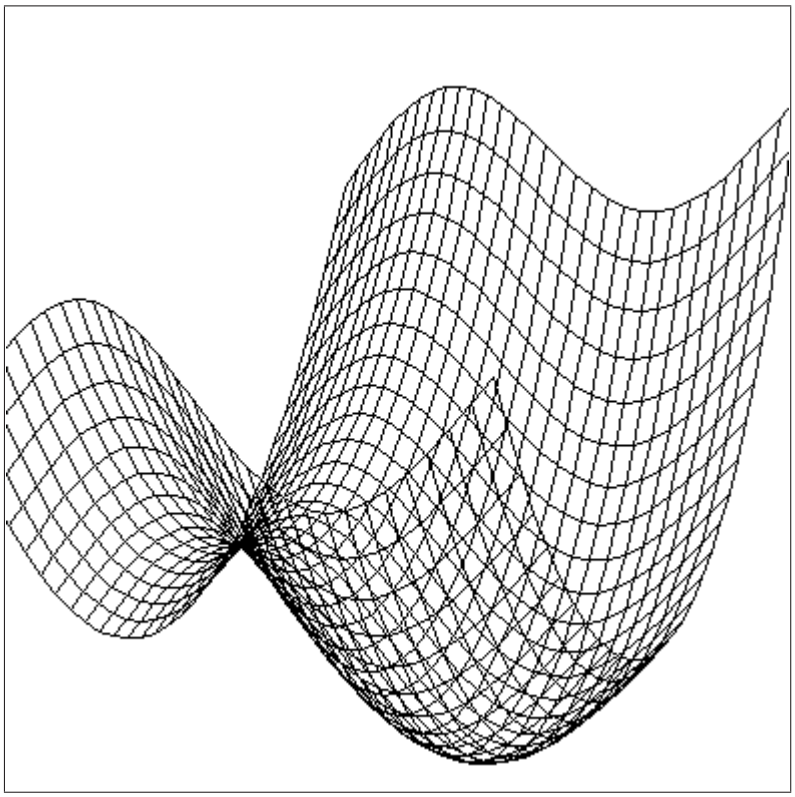


Figure 4.1: A shot of the 3D graph

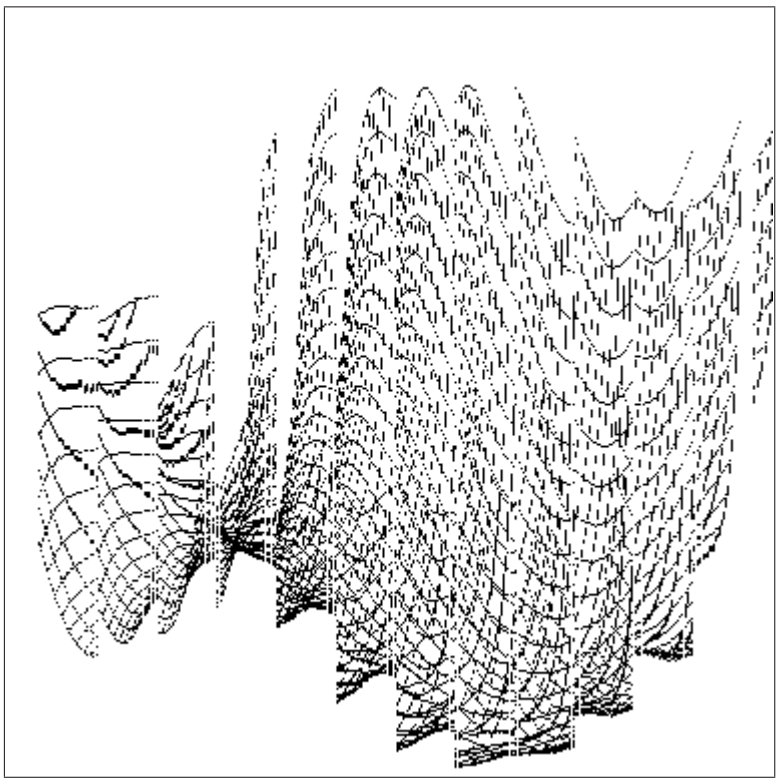


Figure 4.2: A slide of the 3D graph

this means that the view rendered is not precisely the view seen in front of an actual display. This, to a certain extent, accounts for the black and white imperfections of the graph. Another deterioration is the distortion of the depth perception. This particular view has an amplified depth perspective. This happens when the observation point is closer to the display than the spray points. A greater distance will »flatten« the image of the display. This effect will also appear on an actual display and is an unavoidable side effect of allowing several observers, although it may be minimised by adjusting the parameters of the display or by choosing another rendering technique. The third deterioration is the missing part of the graph on the far left of the image. This is due to the rendering method used and may most likely be corrected or limited by optimisation. Since neither the rendering method nor the optimisation of one such is an objective in this report, nothing further will be done to correct this.

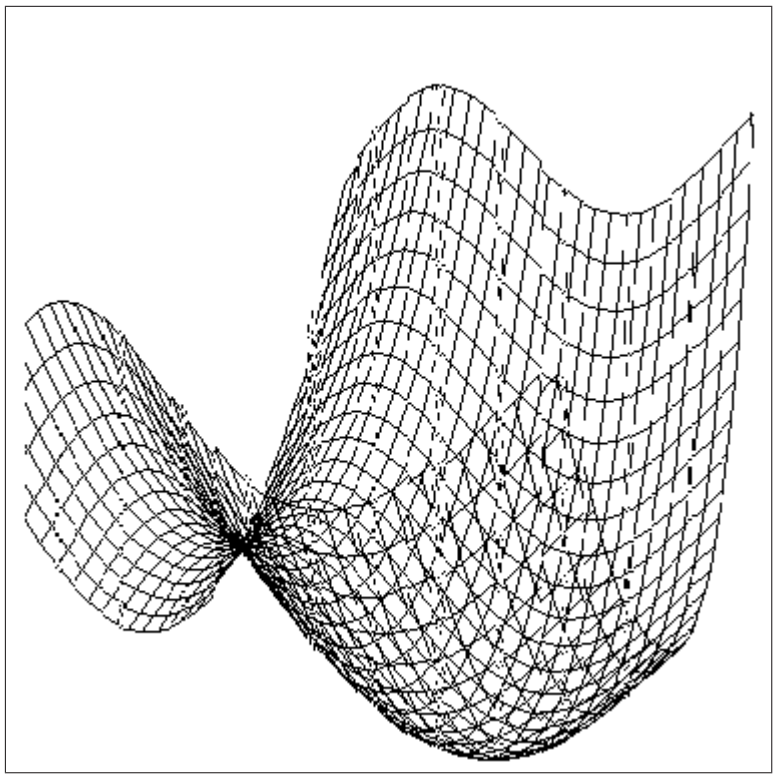


Figure 4.3: A rendered view

Figure 4.4 shows a red/cyan anaglyph[1] stereogram of such a view.

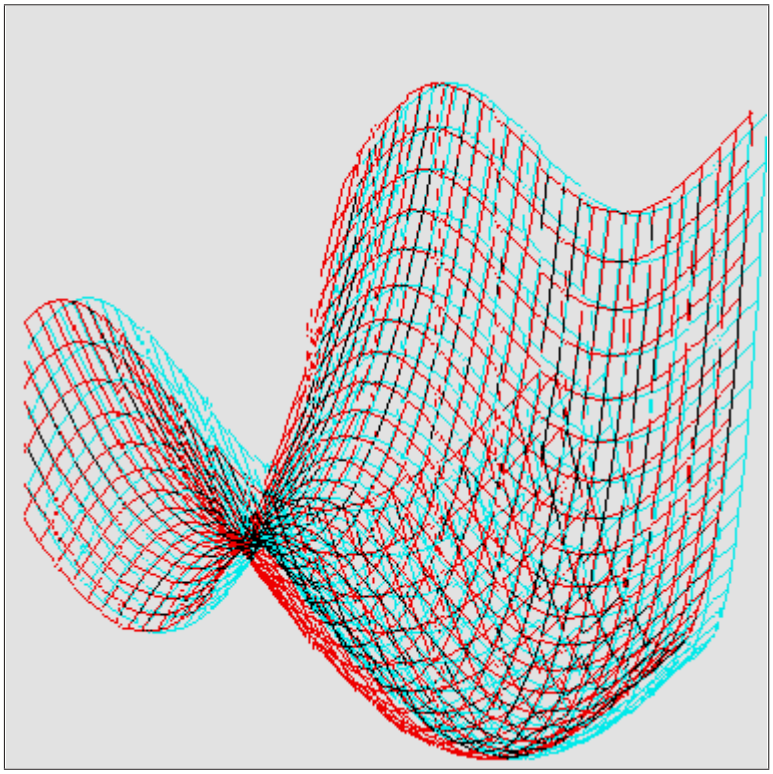


Figure 4.4: An anaglyph image of a rendered view

[1] A three-dimensional image created by superimposing two different views with different colour schemes upon each other

Chapter 5

Evaluation

5.1 An evaluation of the proposed design

In this section the proposed design will be evaluated and compared to other 3D displays in a three-dimensional programming context. Additionally the proposed display will be evaluated in the context of hardware and software requirements.

5.1.1 Comparison to other display types

Table 5.1 shows the assembled characteristics of the examined displays along with the characteristics of the proposed design. Compared to the two most suitable displays in Section 2.6, the proposed display may not supersede the holographic display as a workstation monitor, as the holographic display has full parallax and a higher detail level. However the proposed display certainly surpasses the lenticular and diffractional display in the level of details. Furthermore the display fulfils the remaining requirements quite well. The display can portray depths from somewhere between the display and the heads of the observers and endlessly into the display. The display has look-around and multiple observers capabilities, while being incorporable into a laptop.

	Parallax	Detail level	Depth	Flat	Mechanical	Noisy	2D
Lenticular	Horizontal	Low	Unlimited	✓	×	×	✓
Diffractional	Horizontal	Low	Unlimited	✓	×	×	✓
Swept volume	Full	Normal	Limited	×	✓	✓	×
Emissive volume	Full	Low	Limited	×	×	×	×
Laser-substance	Full	Low	Limited	×	×	N/A	×
Holographic	Full	High	Unlimited	×	×	×	✓
Layered LCD	Full	Normal	Limited	N/A	×	×	✓
Proposed design	Horizontal	Normal	Unlimited	✓	×	×	✓

Table 5.1: Comparison of the described displays

5.1.2 Hardware and software requirements

The software requirements for the proposed display are no greater than different types of autostereoscopic displays with the same level of quality. As mentioned earlier in this report the software requirements for autostereoscopic displays can be quite high, and even though the proposed design excels in leaving further parameters, such as viewing zones, angle, and distance, up to dynamic software settings, this adds little to the computational demand.

The hardware requirements for the presented display are another matter. For instance a display, based on the proposed design, with a refresh rate of 65 Hz, typical to LCD monitors, and 10 viewing zones, would require an refresh rate of 650 Hz on both the front and background displays. Moreover the light intensity of the display, will likewise have been reduced by a factor 10. While this is technically possible, it is certainly not a mass-produced component. These two factors seem like the most important hindrances for widespread use. However, all in all, the hardware seem no higher than the advanced technologies necessary for holographic displays.

Chapter 6

Conclusion

6.1 Conclusion

This report has presented a design for an autostereoscopic display, suited for, but not limited to, three-dimensional languages. The history of three-dimensional languages has been reviewed, along with specific languages, in order to acquire their particular requirements for autostereoscopic displays. The effective area of the display has been theoretically accounted for and the display's parameters have been examined. An impression of the design's functionality has been given by implementing a simulator. Finally the advantages and disadvantages of the design have been accounted for, both by comparison to existing display technologies, and by evaluating the hardware and software requirements for the design. Two factors were was identified as the greatest hindrances for widespread use, they were the reduction in illuminance and refresh rate. Nonetheless the proposed design was found to be the most suitable for use with three-dimensional languages of the investigated displays.

Appendix A

Source code of the simulator

A.1 The `frmSimulator` code

```
1   Dim heads As clsArrayHead
2   Dim myFront As clsFront
3   Dim ismoving As Boolean
4   Dim isturning As Boolean
5   Dim currenthead As Single
6   Dim backfrontdist As Single
7
8   Private Type PointSingle
9       X As Single
10      Y As Single
11      angle As Single
12  End Type
13
14  Const PI = 3.14159265358979
15  Const SRCCOPY = &HCC0020
16
17  Private Sub cmdCycle_Click()
18  Timer1.Enabled = Not Timer1.Enabled
19
20  End Sub
21
22  Private Sub Render()
23  Dim d_BF As Single
24  Dim d_gap As Single, d_bar As Single
25  Dim d_view As Single
```

```
26  Dim d_eye As Single
27  Dim n_view As Single
28  Dim myHead As New clsHead
29  Dim spraypoints() As PointSingle
30  Dim t As Long, s As Long, i As Long
31
32  d_BF = myFront.Y - myBack.y1
33  d_gap = myFront.Gap
34  d_bar = myFront.Bar
35  d_eye = dist(myHead.lefteye.X, myHead.lefteye.Y, myHead.righteye.X, _
    myHead.righteye.Y)
36
37  d_view = PI - Atn(d_BF / (d_bar / 2)) * 2
38  n_view = (d_bar + d_gap) / d_gap
39
40  ReDim spraypoints(n_view)
41
42  'Calculate spraypoints
43  midfrontx = myFront.X + myFront.Width / 2
44  For t = 0 To n_view - 1
45      With spraypoints(t)
46          .angle = -1 / 4 * PI - t * 1 * (PI / 2) / n_view
47          .X = midfrontx + Cos(.angle) * 500
48          .Y = myFront.Y + Sin(.angle) * 500
49      End With
50  Next t
51
52  'Load shot
53  picShots(0).Picture = LoadPicture("C:\Work\VB\Concept\0.bmp")
54  For s = 1 To n_view - 1
55      Load picShots(s)
56      picShots(s).Picture = LoadPicture("C:\Work\VB\Concept\" & s & _
        ".bmp")
57  Next s
58
59  'Create memory slides
60  For pbs = 1 To n_view - 1
61      Load picSlides(pbs)
62  Next pbs
63
64  myFront.Gapoffset = 0
```

```
65
66  For s = 0 To n_view - 1
67      If Dir("C:\Work\VB\Concept\r" & Format(s, "0#") & ".bmp") = "" _
        Then
68      With picSlides(s)
69          For t = 0 To n_view - 1
70              For i = 0 To myFront.Gaps
71                  x1 = projectXonY(spraypoints(t).X, spraypoints(t).Y, _
                    myFront.Gappos(i).X - myFront.Gap * 0.499) - myFront.X
72                  x2 = projectXonY(spraypoints(t).X, spraypoints(t).Y, _
                    myFront.Gappos(i).X + myFront.Gap * 0.499) - myFront.X
73                  If x1 > 0 And x2 > 0 And x1 < 383 And x2 < 383 Then
74                      .PaintPicture picShots(t).Image, x1, 0, x2 - x1, _
                        .ScaleHeight, x1, 0, x2 - x1
75                  End If
76              Next i
77          Next t
78          .Left = s * 200
79          .Visible = True
80          SavePicture .Image, "C:\Work\VB\Concept\r" & Format(s, "0#") _
            & ".bmp"
81      End With
82      End If
83      myFront.Gapoffset = myFront.Gapoffset + myFront.Gap
84  Next s
85
86  End Sub
87
88  Private Sub cmdRender_Click()
89  Render
90
91  End Sub
92
93  Private Sub cmdShow_Click()
94  frm3DGraphCalc.Show
95  End Sub
96
97  Private Sub cmdStep_Click()
98  myFront.Gapoffset = myFront.Gapoffset + myFront.Gap
99  Draw
100
```

```
101 End Sub
102
103 Sub View()
104
105 'Declare variables
106 Dim d_BF As Single
107 Dim d_gap As Single, d_bar As Single
108 Dim d_view As Single
109 Dim d_eye As Single
110 Dim n_view As Single
111 Dim eyepoints() As POINTL
112 Dim t As Long, s As Long, i As Long
113
114 'Calculate initial parametres
115 d_BF = myFront.Y - myBack.y1
116 d_gap = myFront.Gap
117 d_bar = myFront.Bar
118 d_eye = dist(heads.getValue(0).lefteye.X, heads.getValue(0).lefteye.Y, _
           heads.getValue(0).righteye.X, heads.getValue(0).righteye.Y)
119
120 d_view = PI - Atn(d_BF / (d_bar / 2)) * 2
121 n_view = (d_bar + d_gap) / d_gap
122
123 'Show the views from the eyes
124 picEyes(0).Visible = True
125 picEyes(1).Visible = True
126
127 DoEvents
128
129 'Load the eye locations into two local variables
130 ReDim eyepoints(2)
131
132 eyepoints(0) = heads.getValue(0).lefteye
133 eyepoints(1) = heads.getValue(0).righteye
134
135
136 If picSlides.Count = 1 Then
137     'Load saved slides into memory
138     picSlides(0).Picture = LoadPicture("C:\Work\VB\Concept\r00.bmp")
139     For pbs = 1 To n_view - 1
140         Load picSlides(pbs)
```

```
141         picSlides(pbs).Picture = LoadPicture("C:\Work\VB\Concept\r" _
            & Format(pbs, "0#") & ".bmp")
142     Next pbs
143 End If
144
145 'Always make sure the offset is the same
146 myFront.Gapoffset = 0
147 picEyes(0).Cls
148 picEyes(1).Cls
149
150 pb1.value = 0
151
152 'For every slide
153 For s = 0 To picSlides.Count - 1
154     With picSlides(s)
155         'For both eyes
156         For t = 0 To 1
157             'For every gap
158             For i = 0 To myFront.Gaps
159                 'Calculate which part of B the eye sees
160                 x1 = projectXonY(eyepoints(t).X, eyepoints(t).Y, _
                    myFront.Gappos(i).X - myFront.Gap * 0.499) _
                    - myFront.X
161                 x2 = projectXonY(eyepoints(t).X, eyepoints(t).Y, _
                    myFront.Gappos(i).X + myFront.Gap * 0.499) _
                    - myFront.X + 1
162                 If x1 > 0 And x2 > 0 And x1 < 382 And x2 < 382 Then
163                     'Paint the corresponding part
164                     picEyes(t).PaintPicture .Image, x1, 0, x2 - x1, _
                        .ScaleHeight, x1, 0, x2 - x1
165                 End If
166             Next i
167         Next t
168     End With
169     'Shift gap positions one gap width right
170     myFront.Gapoffset = myFront.Gapoffset + myFront.Gap
171
172     pb1.value = s / (picSlides.Count - 1) * 100
173
174 Next s
175
```

```
176 End Sub
177
178 Private Sub cmdView_Click()
179 View
180
181 End Sub
182
183 Private Sub Form_Load()
184 Set heads = New clsArrayHead
185 Set myFront = New clsFront
186 With myFront
187     .Bar = 30
188     .Gap = 1
189     .Gapoffset = 5
190     .Width = 500
191     .X = 200
192     .Y = 35
193 End With
194
195 With myBack
196     .x1 = 200
197     .y1 = 20
198     .x2 = 700
199     .y2 = 20
200 End With
201
202 backfrontdist = myFront.Y - myBack.y1
203 ismoving = False
204 isturning = False
205 currenthead = -1
206
207 End Sub
208
209 Private Sub DrawHeads()
210 Dim skincolor As Single
211
212 For i = 0 To heads.size - 1
213     With heads.getValue(i)
214         If i = currenthead Then skincolor = vbBlue Else skincolor _
            = vbBlack
215         Me.DrawWidth = 10
```

```
216         PSet (.lefteye.X, .lefteye.Y), vbRed
217         PSet (.righteye.X, .righteye.Y), vbGreen
218         PSet (.nose.X, .nose.Y), skincolor
219         Me.DrawWidth = 1
220         Circle (.X, .Y), .Radius, skincolor, BF
221     End With
222 Next i
223
224 End Sub
225
226 Private Sub DrawFront()
227 Dim i As Long
228
229 Line (myFront.X, myFront.Y)-(myFront.Gappos(0).X, myFront.Gappos(0).Y)
230 For i = 1 To myFront.Gaps
231     Line (myFront.Gappos(i - 1).X + myFront.Gap, _
        myFront.Gappos(i - 1).Y)-(myFront.Gappos(i).X, myFront.Gappos(i).Y)
232 Next
233
234 End Sub
235
236 Private Sub DrawLines()
237 Dim i As Long, hi As Long
238
239 For hi = 0 To heads.size - 1
240     With heads.getValue(hi)
241
242         For i = 0 To myFront.Gaps
243             Line (projectXonY(.lefteye.X, .lefteye.Y, _
                myFront.Gappos(i).X - myFront.Gap * 0.499), myBack.y1)- _
                (.lefteye.X, .lefteye.Y), vbRed
244             Line (projectXonY(.lefteye.X, .lefteye.Y, _
                myFront.Gappos(i).X + myFront.Gap * 0.499), myBack.y1)- _
                (.lefteye.X, .lefteye.Y), vbRed
245         Next
246
247         For i = 0 To myFront.Gaps
248             Line (projectXonY(.righteye.X, .righteye.Y, _
                myFront.Gappos(i).X - myFront.Gap * 0.499), myBack.y1)- _
                (.righteye.X, .righteye.Y), vbGreen
249             Line (projectXonY(.righteye.X, .righteye.Y, _
```

```
                myFront.Gappos(i).X + myFront.Gap * 0.499), myBack.y1)- _
                (.righteye.X, .righteye.Y), vbGreen
250         Next
251
252     End With
253 Next hi
254
255 End Sub
256
257 Private Sub Draw()
258 Me.Cls
259
260 DrawHeads
261 DrawFront
262 DrawLines
263
264 End Sub
265
266 Private Function projectXonY(ByVal x1 As Single, ByVal y1 As Single, _
    ByVal x2 As Single) As Single
267 projectXonY = x2 - (x1 - x2) / (y1 - CSng(myFront.Y)) * _
    CSng(backfrontdist)
268
269 End Function
270
271 Private Function dist(ByVal x1 As Single, ByVal y1 As Single, _
    ByVal x2 As Single, ByVal y2 As Single) As Single
272 dist = Sqr((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
273
274 End Function
275
276 Private Sub Form_MouseDown(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
277
278 'currenthead = -1
279 hit = False
280 ismoving = False
281 For i = 0 To heads.size - 1
282     With heads.getValue(i)
283         If dist(.X, .Y, X, Y) < .Radius Then
284             currenthead = i
```

```
285             hit = True
286         End If
287     End With
288 Next i
289
290 If Button = 2 Then
291     If Not hit Then
292         Dim myHead As New clsHead
293
294         myHead.X = X
295         myHead.Y = Y
296
297         heads.addvalue myHead
298     Else
299         heads.Remove currenthead
300         ismoving = False
301         currenthead = -1
302     End If
303 Else
304     If Not hit Then
305         If currenthead <> -1 Then
306             isturning = True
307         End If
308     Else
309         ismoving = True
310     End If
311 End If
312
313 Draw
314
315 End Sub
316
317 Private Sub Form_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
318
319 If ismoving Then
320     With heads.getValue(currenthead)
321         .X = X
322         .Y = IIf(Y > (myFront.Y + .Radius + 1), Y, myFront.Y + _
            .Radius + 1)
323     End With
```

```
324     Draw
325 End If
326
327 If isturning Then
328     With heads.getValue(currenthead)
329         If Y < .Y Then
330             .Direction = -Atn((.X - X) / (.Y - Y + 0.0000000001))
331         Else
332             .Direction = PI - Atn((.X - X) / (.Y - Y + 0.0000000001))
333         End If
334     End With
335     Draw
336 End If
337
338 Caption = "(" & X & ", " & Y & ")"
339
340 End Sub
341
342 Private Sub Form_MouseUp(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
343 ismoving = False
344 isturning = False
345
346 End Sub
347
348 Private Sub Form_Paint()
349
350 Draw
351
352 End Sub
353
354 Private Sub Form_Unload(Cancel As Integer)
355
356 For i = 1 To picSlides.UBound - 1
357     Unload picSlides(i)
358 Next i
359
360 Unload frm3DGraphCalc
361
362 End Sub
363
```

```
364 Private Sub Timer1_Timer()
365 myFront.Gapoffset = myFront.Gapoffset + myFront.Gap
366 Draw
367
368 End Sub
```

A.2 The clsHead code

```
1   Private myX As Single
2   Private myY As Single
3   Private myRadius As Single
4   Private myDirection As Single
5   Private Const PI = 3.14159265358979
6
7   Public Property Get lefteye() As POINTL
8   Dim mypoint As POINTL
9
10  mypoint.X = myX + Cos(-2 / 3 * PI + myDirection) * myRadius
11  mypoint.Y = myY + Sin(-2 / 3 * PI + myDirection) * myRadius
12
13  lefteye = mypoint
14
15  End Property
16
17  Public Property Get righteye() As POINTL
18  Dim mypoint As POINTL
19
20  mypoint.X = myX + Cos(-1 / 3 * PI + myDirection) * myRadius
21  mypoint.Y = myY + Sin(-1 / 3 * PI + myDirection) * myRadius
22
23  righteye = mypoint
24
25  End Property
26
27  Public Property Get nose() As POINTL
28  Dim mypoint As POINTL
29
30  mypoint.X = myX + Cos(-0.5 * PI + myDirection) * (myRadius + 5)
31  mypoint.Y = myY + Sin(-0.5 * PI + myDirection) * (myRadius + 5)
32
33  nose = mypoint
```

```
34
35  End Property
36
37  Public Property Get X() As Single
38  X = myX
39  End Property
40
41  Public Property Get Y() As Single
42  Y = myY
43  End Property
44
45  Public Property Get Radius() As Single
46  Radius = myRadius
47  End Property
48
49  Public Property Let X(X As Single)
50  myX = X
51  End Property
52
53  Public Property Let Y(Y As Single)
54  myY = Y
55  End Property
56
57  Public Property Let Direction(Direction As Single)
58  myDirection = Direction
59  End Property
60
61  Public Property Get Direction() As Single
62  Direction = myDirection
63  End Property
64
65  Private Sub Class_Initialize()
66  myRadius = 100
67  myDirection = 0
68
69  End Sub
```

A.3 The clsFront code

```
1   Private myWidth As Single
2   Private myBar As Single
```

```
3   Private myGap As Single
4   Private myX As Single
5   Private myY As Single
6   Private myGapoffset As Single
7
8   Public Property Let Bar(Bar As Single)
9   myBar = Bar
10
11  End Property
12
13  Public Property Let Gap(Gap As Single)
14  myGap = Gap
15
16  End Property
17
18  Public Property Get Bar() As Single
19  Bar = myBar
20
21  End Property
22
23  Public Property Get Gap() As Single
24  Gap = myGap
25
26  End Property
27
28  Public Property Let Width(Width As Single)
29  myWidth = Width
30
31  End Property
32
33  Public Property Get Width() As Single
34  Width = myWidth
35
36  End Property
37
38  Public Property Let X(X As Single)
39  myX = X
40
41  End Property
42
43  Public Property Get X() As Single
```

```
44  X = myX
45
46  End Property
47
48  Public Property Let Y(Y As Single)
49  myY = Y
50
51  End Property
52
53  Public Property Get Y() As Single
54  Y = myY
55
56  End Property
57
58  Public Property Get Gapoffset() As Single
59  Gapoffset = myGapoffset
60
61  End Property
62
63  Public Property Let Gapoffset(Gapoffset As Single)
64  myGapoffset = Gapoffset Mod myBar
65
66  End Property
67
68
69  Public Property Get Gappos(Index As Long) As POINTL
70  Dim mypos As POINTL
71
72  mypos.Y = myY
73
74  mypos.X = myX + myGapoffset + Index * (myBar + myGap)
75
76  Gappos = mypos
77
78  End Property
79
80  Public Property Get Gaps() As Long
81  Gaps = (myWidth - myGapoffset) \ (myBar + myGap)
82
83  End Property
```

Appendix B

Résumé

The focus of this project is the design of a 3D display that is suitable for use with three-dimensional programming languages. In order to derive the specific requirements of these, the history of three-dimensional programming languages are reviewed and three three-dimensional languages are analysed. These are the first three-dimensional programming language Cube, the minimalistic 3D-pixel-based 3D-Visulan, and the 3D-PP language, which focuses on optimal screen area usage. Successively a set of requirements is compiled.

Subsequently the principles of different autostereoscopic displays are described. These are the parallactic displays, both lenticular and diffractional, and the volumetric displays, which include the swept volume displays, the emissive volume displays, the laser-substance displays, the holographic displays, and the layered LCD displays. Based on the display analysis an assembly of characteristics is devised and the examined displays are considered in the context of the three-dimensional programming languages' requirements for autostereoscopic displays.

Upon the analysis a design for an autostereoscopic display is presented. This design is a generalisation of a design by Ken Perlin, from supporting one observer to multiple observers. The model for the design includes mathematical formulae to calculate the extent of the effective zone, the area in which it is possible to obtain a 3D view. Additionally a model for a simulator is presented including pseudo code for a rendering technique.

Succeeding an account is given for the implementation of the simula-

tor and, as a result from the simulator, rendered graphics is presented and reviewed.

Moreover the design is evaluated both by comparison to the examined display technologies, and by evaluating the hardware and software requirements for the design. Finally the works of this project are concluded upon.

Bibliography

[1] AIST2006. Three dimensional images in the air. Press release, February 2006. URL `http://www.aist.go.jp/aist_e/latest_research/2006/20060210/20060210.html`. 2.5.2

[2] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. URL `http://www.stanford.edu/class/cs242/readings/backus.pdf`. 2.2

[3] Alberto Del Bimbo, Luigi Rella, and Enrico Vicario. Visual specification of branching time temporal logic. *IEEE 1995*, 1995. URL `http://www.cs.concordia.ca/~haarslev/vl95www/ieee/delbimbo.ps.gz`. 2.3

[4] Paul Bourke. *Autostereoscopic lenticular images*, December 1999. URL `http://astronomy.swin.edu.au/~pbourke/stereographics/lenticular/index.html`. 2.5.1

[5] P. G. Comba. A language for three-dimensional geometry. *IBM Systems Journal*, 7(3 and 4):292–307, 1968. URL `http://www.research.ibm.com/journal/sj/073/ibmsj3a4N.pdf`. 2.3

[6] P. T. Cox and I. J. Mulligan. Compiling the graphical functional language prograph. *Proceedings of the 1985 ACM SIGSMALL symposium on Small systems*, pages 34–41, 1985. URL `http://portal.acm.org/citation.cfm?id=317169&dl=ACM&coll=portal`. 2.2

[7] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Lan-*

guages, pages 207–212. ACM, 1982. URL http://pag.csail.mit.edu/6.883/readings/p207-damas.pdf. 2.3.1

[8] Melvin E. Dickover, Clement L. McGowan, and Douglas T. Ross. Software design using: Sadt. *Proceedings of the 1977 ACM Annual Conference*, pages 125–133, 1977. URL http://portal.acm.org/ft_gateway.cfm?id=810192&type=pdf&coll=GUIDE&dl=GUIDE&CFID=76205547&CFTOKEN=8189622. 2.3

[9] Tim Dwyer. Three-dimensional uml using force directed layout. In Eades and Eds Tim Pattison, editors, *Conferences in Research and Practice in Information Technology*, volume 9, 2001. URL http://citeseer.ist.psu.edu/cache/papers/cs/26540/http:zSzzSzwww.jrpit.flinders.edu.auzSzconfpaperszSzCRPITV9Dwyer.pdf/dwyer01three.pdf. 1.1

[10] Christian Geiger, Wolfgang Muller, and Waldemar Rosenbach. SAM - an animated 3d programming language. In *Visual Languages*, pages 228–235, 1998. URL http://jerry.c-lab.de/~wolfgang/vl98a.pdf. 2.3

[11] Ken Kahn. Toontalk - an animated programming environment for children. In *The Journal of Visual Languages and Computing*, volume 7, number 2, 1996. URL http://www.toontalk.com/Papers/jvlc96.pdf. 2.3

[12] Jana Koehler, Rainer Hauser, Shane Sendall, and Michael Wahler. Declarative techniques for model-driven business process integration. *IBM SYSTEMS JOURNAL*, 44(1), 2005. URL http://www.research.ibm.com/journal/sj/441/koehler.pdf. 1.1

[13] Ray Kurzweil. *The singularity is near*. Penguin, 2005. ISBN 0-7156-3561-1. 1.1

[14] Paul McIntosh, Margaret Hamilton, and Ron van Schyndel. X3d-uml: Enabling advanced uml visualisation through x3d. Association for Computing Machinery, Inc., 2005. URL http://goanna.cs.rmit.edu.au/~ronvs/papers/WEB3D05.PDF. 1.1

[15] Marc A. Najork. *Programming in Three Dimensions.* Technical report, Univ. of Illinois, Dept. of Computer Science, October 1993. URL `http://research.microsoft.com/~najork/thesis.pdf`. 2.3.1

[16] Marc A. Najork. Programming in three dimensions. *Journal of Visual Languages and Computing*, 7(2):219–242, June 1996. 2.3.1

[17] Marc A. Najork and Simon M. Kaplan. The cube language. *1991 IEEE Workshop on Visual Languages*, pages 218–224, October 1991. URL `http://ieeexplore.ieee.org/iel2/376/6139/00238829.pdf?isnumber=&arnumber=238829`. 2.3, 2.3.1

[18] Takashi Oshiba and Jiro Tanaka. "3d-pp": Visual programming system with three-dimensional representation. In *Proceeding of International Symposium on Future Software Technology (ISFST '99)*, pages 61–66, October 1999. URL `http://www.iplab.cs.tsukuba.ac.jp/~ohshiba/paper.lnk/isfst99-ohshiba.pdf`. 2.3

[19] Ken Perlin. Princess leia, in a beam of light. Website. URL `http://mrl.nyu.edu/~perlin/experiments/holodust/index.html`. 2.5.2

[20] Ken Perlin, Salvatore Paxia, and Joel S. Kollin. An autostereoscopic display. In *SIGGRAPH 2000 Conference Proceedings.* Media Research Laboratory, Dept. of Computer Science, New York University, July 2000. URL `http://www.mrl.nyu.edu/publications/autostereo/autostereo.pdf`. 3.1

[21] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML.* Library of Congress, 2004. ISBN 0521537711. 1.1

[22] Frank Van Reeth and Eddy Flerackers. Three-dimensional graphical programming in cael. *IEEE 1993*, 1993. URL `http://ieeexplore.ieee.org/iel2/467/6709/00269554.pdf?isnumber=&arnumber=269554`. 2.3

[23] Ben Shneiderman. Direct manipulation: A step beyond programming

languages. In *Proceedings of the joint conference on Easier and more productive use of computer systems*, page 143. ACM Press, 1981. 2.3.3

[24] Randy Stiles and Michael Pontecorvo. Lingua graphica: A visual language for virtual environments. *IEEE 1992*, 1992. URL `http://ieeexplore.ieee.org/iel2/895/6833/00275759.pdf?arnumber=275759`. 2.3

[25] Sabine Volbracht, K. Shahrbabaki, Gitta Domik, and Gregor Fels. Perspective viewing, anaglyph stereo or shutter glass stereo? In *Proceedings of the 1996 IEEE Symposium on Visual Languages (VL '96)*, 1996. URL `http://csdl2.computer.org/comp/proceedings/vl/1996/7508/00/75080192.pdf`. 1.1

[26] Kakuya Yamamoto. 3d-visulan: A 3d programming language for 3d applications. In *Pacific Workshop on Distributed Multimedia Systems (DMS96)*, pages 199–206, 1996. URL `http://www.yuasa.kuis.kyoto-u.ac.jp/ylab/yamakaku/Dms96/dms96.html`. 2.3