TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

M A G I S T E R A R B E I T

Visualization of Complex Function Graphs in Augmented Reality

ausgeführt am

Institut für
Softwaretechnik und Interaktive Systeme
der Technischen Universität Wien

unter der Anleitung von

Univ.Ass. Mag. Dr. Hannes Kaufmann

durch

Robert Liebo

Brahmsplatz 7/11
1040 Wien

_____ _____

Datum Unterschrift

Abstract

Understanding the properties of a function over complex numbers can be much more difficult than with a function over real numbers. This work provides one approach in the area of visualization and augmented reality to gain insight into these properties.

The applied visualization techniques use the full palette of a 3D scene graph's basic elements, the complex function can be seen and understood through the location, the shape, the color and even the animation of a resulting visual object. The proper usage of these visual mappings provides an intuitive graphical representation of the function graph and reveals the important features of a specific function.

Augmented reality (AR) combines the real world with virtual objects generated by a computer. Using multi user AR for mathematical visualization enables sophisticated educational solutions for studies dealing with complex functions.

A software framework that has been implemented will be explained in detail, it is tailored to provide an optimal solution for complex function graph visualization, but shows as well an approach to visualize general data sets with more than 3 dimensions. The framework can be used in a variety of environments, a desktop setup and an immersive setup will be shown as examples.

Finally some common tasks involving complex functions will be shown in connection with this framework as example usage possibilities.

Dedication

Dedicated to everybody who took more than a year to finish his/her master thesis.

Acknowledgments

I want to thank everybody who made this work possible, first of all my family, Barbara, the people I live with, my friends and my colleagues who supported my all the time.

This work would also have been impossible without the work of my supervisor Hannes Kaufmann and Dieter Schmalstieg who guided me into the world of VR, the people of the VR-group in Vienna and Graz and the computer graphics institute with its master and the CGClub.

Table of Contents

List of Figures

1 Introduction

1.1 Motivation

Visualization is a field of research that aims to provide solutions to generate images from a set of data/numbers in such a way, that insight into the important features can be achieved.

The idea for the topic of this master thesis came to the author when he was trying to understand equations from the field of signal processing, filters and Fourier transformations. The behaviour of filters is connected to features of a complex function (its zeros and poles, its values around the unit circle, etc.), so the author generated a plot of such a function using different methods provided by the MathWorks Matlab© software. Unsatisfied with the interactivity provided to change the parameters in real time, a limited set of visualization techniques and no integration into a standard 3D graphics environment that would allow stereo viewing and combination with other scene graph objects on the one hand and having a virtual reality (VR) / augmented reality (AR, see section 2.2) background due to experience with the Studierstube framework (see section 3.2) on the other hand the author started this project to implement a complex function graph visualization framework for Studierstube.

The conceptual idea is generally speaking a function plotter software. This is a long existing almost trivial type of software, so it is hardly a subject of scientific research. Still the available applications are good examples for the desired workflow: input a function and a range from the domain set, then view the result and optionally manipulate the function or the range in order to gain more information (feedback loop: input ⇔ view). When this familiar workflow is followed, the user interaction of a function graph visualization software becomes very intuitive.

Because of their conceptual relationship an extensive survey on function plotters has been done (see appendix A). The more advanced ones already enable to some extent complex function visualization.

Next to the function plotter applications some dedicated complex function visualization solutions exist (see section 2.1). Manifold examples show different approaches that developed over a long time. A motivator for this project was to unify as many of their visualization features as possible in one framework.

A project related to this work as another mathematical visualization in AR is Construct3D (see reference [1] and section 2.2.1). Extensive user studies showed that Construct3D is a very useful solution to make high school students

understand 3D constructive geometry, even if they had problems to imagine the same content when doing these tasks with pen and paper on the desk. These results motivated to implement the complex function graph visualization framework in a similar manner. Understanding complex functions is sometimes quite a challenge for undergraduate students who have to deal with them for example in electrical engineering courses. They could be a target user group for an educational complex mathematics AR visualization software with user scenarios similar to those in Construct3D.

1.2 Applications

1.2.1 Math Education

A complex function graph visualization would have a similar application in mathematics education as the function plotters presented in section 7.2. Teachers could use the software in the classroom to visualize learning content whereas students could use it at home when revising the last lectures content and as a help to understand text book information and assignment tasks. Those assignments could be already prepared for the software, so that insight can be won in very little time of exploring the virtual objects and possibly manipulating parameters. With to related application Construct3D (see section 2.2.1) further scenarios have been studied. These scenarios include pairs of students solving a task together. This requires some social interaction and communication, which in turn massively increases the durability of the lessons learned.

1.2.2 Electrical Engineering Math

Complex mathematics has an important role in electrical engineering. One ubiquitous application are calculations concerning circuits with alternating current. Here one complex number can represent a sinusoid's amplitude and phase. The electric impedance for example can be easily computed by a complex number, the complex electric impedance. In a passive electric circuit it is the relation of the complex effective value (i.e. the amplitude value) of a sinusoidal voltage and the complex effective value of a sinusoidal electric current: $Z_e = \frac{\dot{U}}{\dot{I}}$.

All calculations can now be done using complex numbers without having to expand cosine and sine terms. This makes equations much shorter and easier to read and understand.

1.2.3 Filter Design Math

Another application arises from the area of digital signal processing (DSP). The behaviour of linear time invariant systems [2], a common type of DSP systems, in

how far it affects the amplitudes and phases of the frequency components of a signal (frequency domain behaviour), can be calculated using complex mathematics. This behaviour can be described by the transfer function $H(z)$ and the frequency response $H(f)$ of the digital system.

Section 6.2 will present an usage example dealing with these functions as a result of this work. To explain the mathematical background in a short way: two transforms are needed, the Z-transform and the discrete time Fourier transform (DTFT): (pay attention to the imaginary number i being denoted in the fields of electrical engineering and signal processing as j)

Z-transform: $X(z)=\sum_{n=-\infty}^{\infty} x(n)\cdot z^{-n}$ for the signal x(n)

DTFT: $X(e^{j\Omega})=\sum_{n=-\infty}^{\infty} x(n)\cdot e^{-j\Omega n}$

with $\Omega=\omega T=2\pi\frac{f}{f_s}$

for the frequency f and sample rate f_s

The DTFT is Z-transform evaluated around the unit circle in the complex plane, which is given by $z=e^{j\Omega}$.

The transfer function of a system is the Z-transform applied to the impulse response $h(n)$ and the frequency response is the DTFT applied to it :

transfer function: $H(z)=\sum_{n=-\infty}^{\infty} h(n)\cdot z^{-n}$

frequency response: $H(e^{j\Omega})=\sum_{n=-\infty}^{\infty} h(n)\cdot e^{-j\Omega n}$

$H(f)=\sum_{n=-\infty}^{\infty} h(n)\cdot e^{-j2\pi\frac{f}{f_s}n}$

Visualizing these functions, observing features as zeros, poles and values around the unit circle while tuning the parameters of $h(n)$ for a desired frequency response of the system would be a helpful application of a complex function graph visualization software.

2 Related Work

This chapter presents previous work on the topics of complex function visualization, augmented reality and visualization in scene graph APIs, which are the three main fields that were brought together in this work.

Additionally a survey on function plotting solutions that are not bound to complex mathematics can be found in appendix A.

2.1 Complex Function Visualization

Existing examples from three categories of complex function graph visualizing solutions (stand-alone applications, web based applications, usage of programs not specialized on complex functions) are explained in the following. Their key features that influenced this work are highlighted. The list is ordered by the number of references found to each solution.

2.1.1 Applications

f(z) [3]

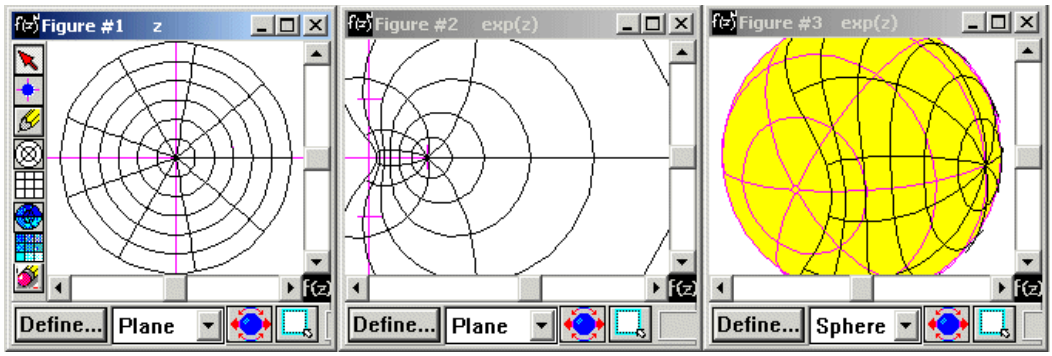


Figure 1: f(z) screenshots

f(z) is a commercial Windows package for visualizing complex mappings. It shows images of Cartesian grids, polar grids and single points. Function parameters can be animated and the animations can be controlled with a slider, in this way also a parameter could be controlled with that slider. The graph can be mapped to a plane, a sphere, a 3D and a 4D representation.

g(z) [4]

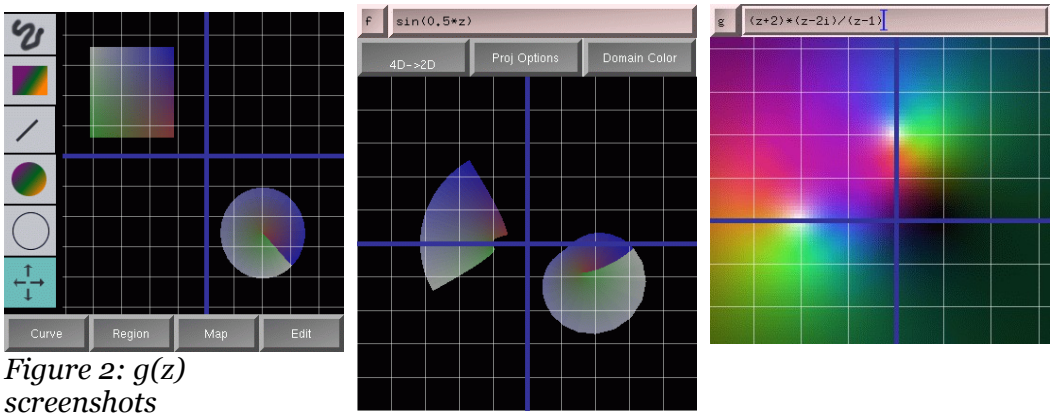


Figure 2: g(z) screenshots

This is a tool for visual complex analysis developed by David L. Akers as an honors thesis project at Brown University. The software was used successfully in Brown's undergraduate complex analysis course. It allows mapping of Cartesian and polar grids, that can be dragged with the mouse as well as drawing custom shapes in the domain set. An additional window displays another visualization using domain coloring (see section 2.1.3 for an explanation of this concept).

Real-Time Zooming Math Engine (rtzme) [5]

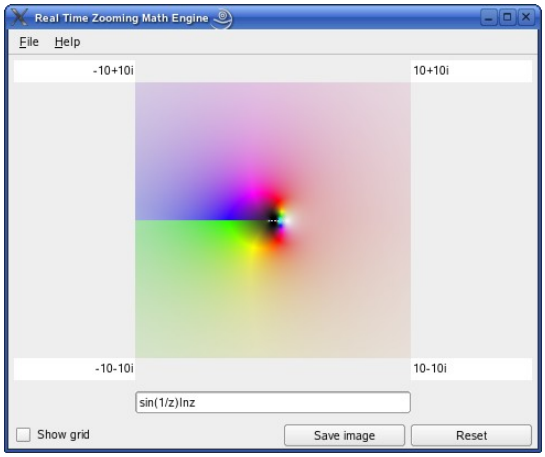


Figure 3: rtzme screenshot

This is another program for making domain coloring plots. This one doesn't show much novelty from the method of visualization, but distinguishes from other solutions by being implemented with particular emphasis on computation speed.

2.1.2 Web Based Complex Function Visualization

A Complex Function Viewer (Java) [6]

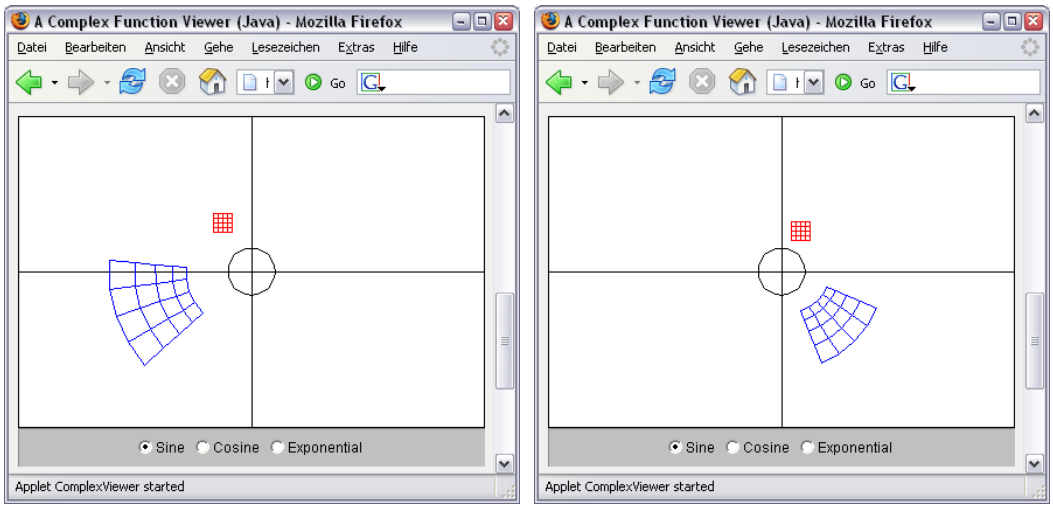


Figure 4: A Complex Function Viewer screenshots

This simple java applet implements a purely 2 dimensional visualization of a fix set of 3 predefined example functions (sine, cosine, exponential). The user can move the red square grid around with the mouse pointer, it represents a neighbourhood of a point in the domain. The twisted (blue) grid represents the image of the square grid. The source code of this applet is available, so a user who can do some programming in java could implement arbitrary functions.

Bombelli [7]

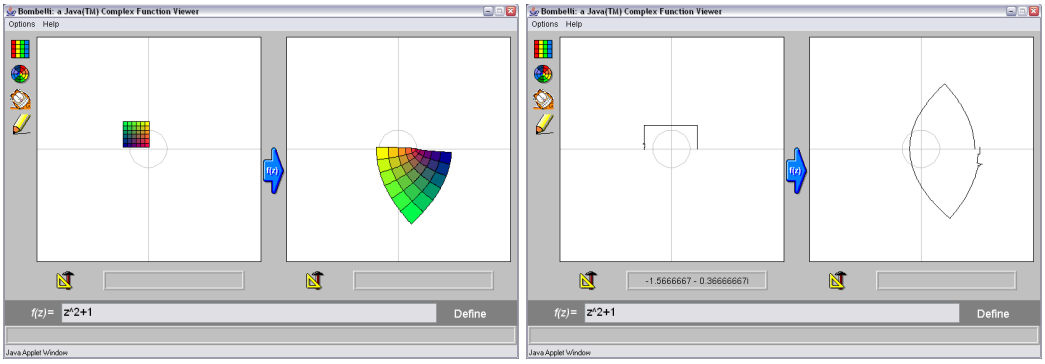


Figure 5: Bombelli screenshots

Bombelli is another 2 dimensional complex function grapher, conceptually similar to the one shown above, but with more features. Arbitrary functions can be defined. It maps Cartesian (square) and polar (round) grids from one plane to another. The grids are color coded, so the user can see which corner of the grid is mapped to which corner in the codomain. A little inconvenient is, that no dragging of the grid using the mouse pointer is possible, to see the grid at another position, a new grid has to be drawn. The user can also draw custom shapes in the domain and see its mapping in the codomain (right side of fig. 5).

Complex Function Grapher [8]

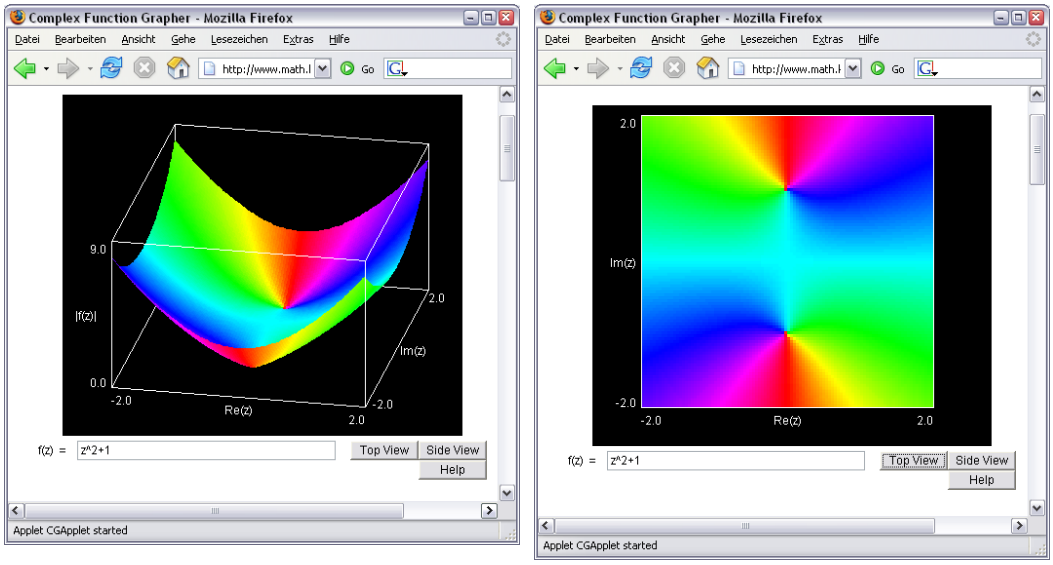


Figure 6: Complex Function Grapher applet screenshots

This is simple java applet that maps the codomain's magnitude to height above the complex plane and the codomain's argument to color. It allows only very limited interaction (two viewpoints, that can't be changed).

2.1.3 Visualizations Generated by Existing Software

Table of Conformal Mappings Using Continuous Coloring [9]

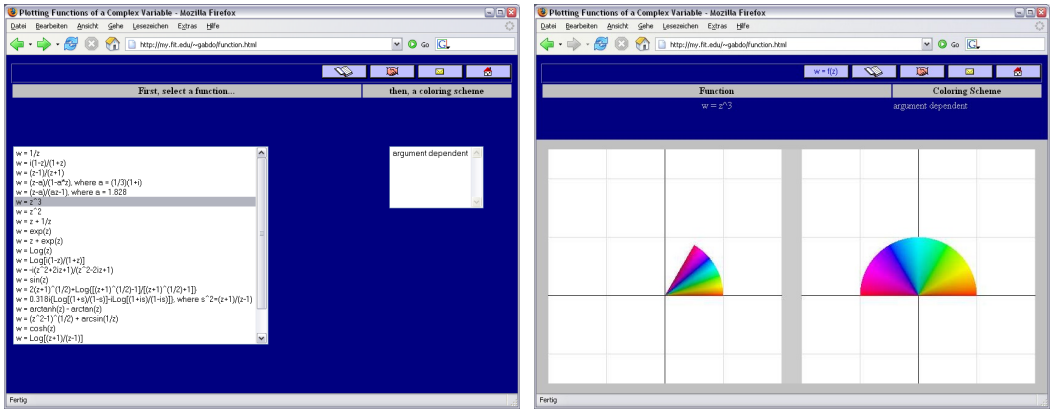


Figure 7: Table of Conformal Mappings... web page

This is web site with a short article by George Abdo and Paul Godfrey on using coloring schemes to plot functions of a complex variable and a large set of example images for interesting functions. These images were generated with Matlab. The authors made the Matlab files available for download. The article additionally gives a historic background, that René Descartes introduced the vertical placing of the y-axis to plot functions of real numbers.

Graphics for Complex Analysis [10]

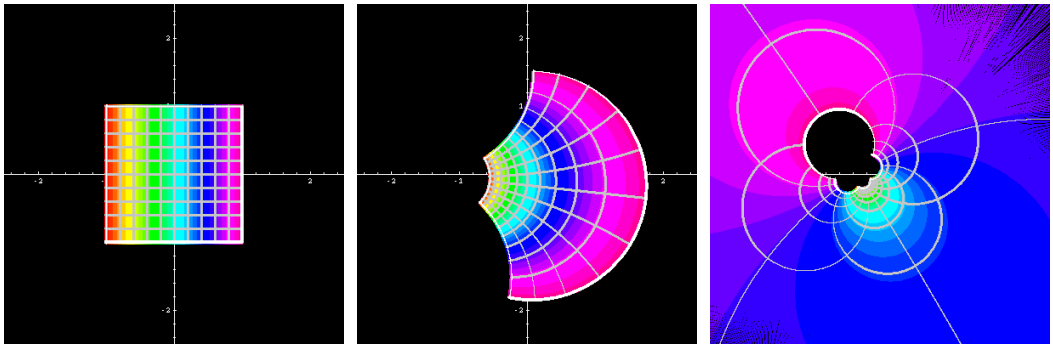


Figure 8: Graphics for Complex Analysis example animation

This short article on graphical demonstrations of concepts in complex analysis by Douglas N. Arnold features example images and animations that illustrate selected functions and problems generated with Mathematica. The Mathematica files that were used are available for download.

Understanding Complex Function Graphs [11]

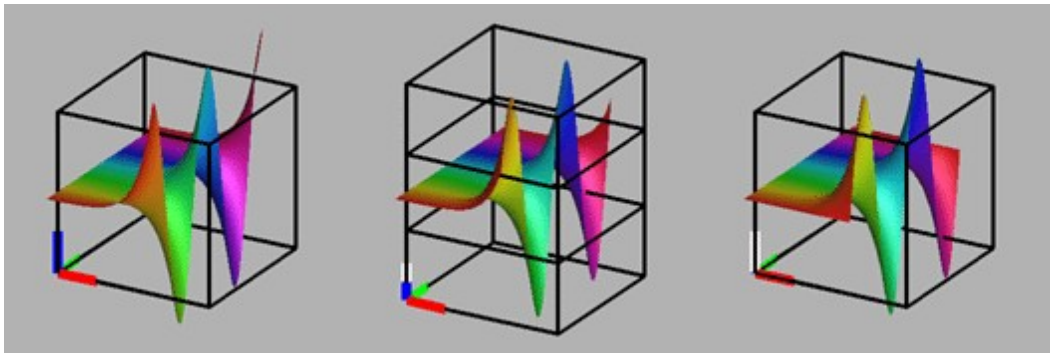


Figure 9: example graph animation by Banchoff and Cervone

Tom Banchoff and Davide Cervone provide an article, example animations and VRML models that illustrate complex function graphs on their web site. The animations are described by them as follows: “This movie shows projections of the complex [...] function into three-space as the graph is rotated in four-space. The initial position projects into the x, y, u space, so this is the graph of the real part of the complex exponential. The final position is the projection into the x, y, v space, which is the graph of the imaginary part of the complex exponential. The intermediate frames follow the rotation in the uv-plane that takes the v-axis to the u-axis. ”

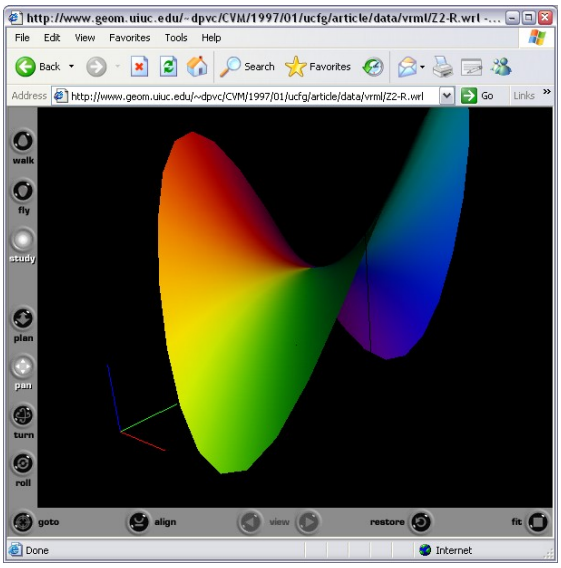


Figure 10: example graph VRML file by Banchoff and Cervone (screenshot)

Complex Function Visualization [12]

Figure 11: Complex Function Visualization images by Frank Farris

This approach by Frank A. Farris from Santa Clara University has been quoted by many others who created 2d color visualizations. He developed it in conjunction with a review [13] of the book Visual Complex Analysis by Tristan Needham [14]. He associates colors to the complex numbers in the domain set in the following way (see left image in fig.11): "Think of the complex plane as having colors similar to those in a traditional color wheel. We put red at the complex number 1, with green and blue at the other two cube roots of unity as shown. Hues are interpolated, giving secondary and tertiary colors. A continuous blending would be possible, but here we show just twelve hues. Then we blend toward white at the center, toward black going outwards. Thus, each complex number has a color associated to it."[12]. The resulting images are referred to as "domain coloring diagrams". Besides their strong capabilities to exhibit important properties of the function, the generated images inhere some astounding aesthetic component. The center image in fig. 11 shows "a sixth degree polynomial with 4 simple zeroes and one double zero. You can spot the double zero because the colors cycle around twice when you make a circuit of that point in the domain."[12] The right image is a rational function with two zeroes (the white points) and two poles (the black ones); the function is: $f(z)=\frac{z^2-i}{z^2+i}$

Resources for the Teaching of Complex Variables [15]
Paul Fishback collects one this web page resources for teaching an introductory, undergraduate course in complex variables. Therein he includes files for interesting functions and problems for the f(z) application (see section 2.1.1).

visualizing complex analytic functions using domain coloring [16]

Figure 12: Hans Lundmark's approach using an image in the domain set

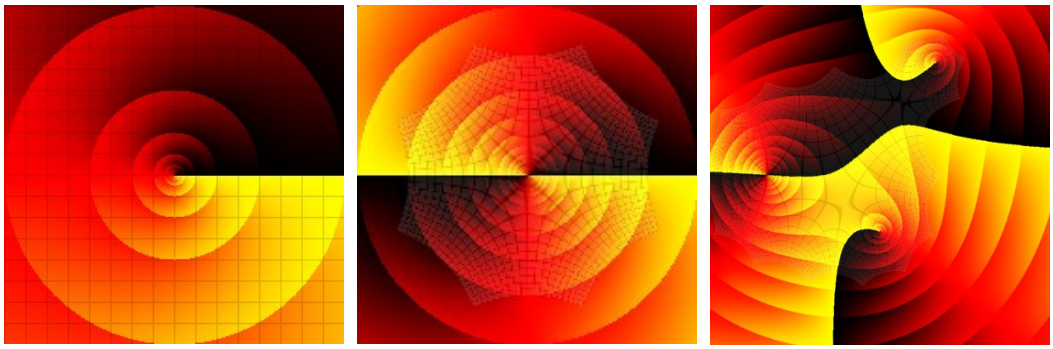


Figure 13: Hans Lundmark's approach using a more useful coloring scheme

This work of Hans Lundmark follows the domain coloring approach by Frank Farris. In his web article he shows example images generated with the image editor GIMP and the MathMap Plug-In [17]. The author uses textures in the domain set (left image in fig. 12 and 13) to enable reading of information from the mapped codomain set image. Fig. 13 shows a coloring scheme that visualizes argument by color (from black over red to yellow) and magnitudes by shade (gray-scale mask with brightness from 0 to 1 equal to the fractional part of $\log_2|z|$). The center image in fig. 13 shows $f(z)=z^2$, the right image shows $f(z)=(z+2)^2\,(z-1-2i)\,(z+i)$.

Newton's method, Julia and Mandelbrot sets, and complex coloring [18]

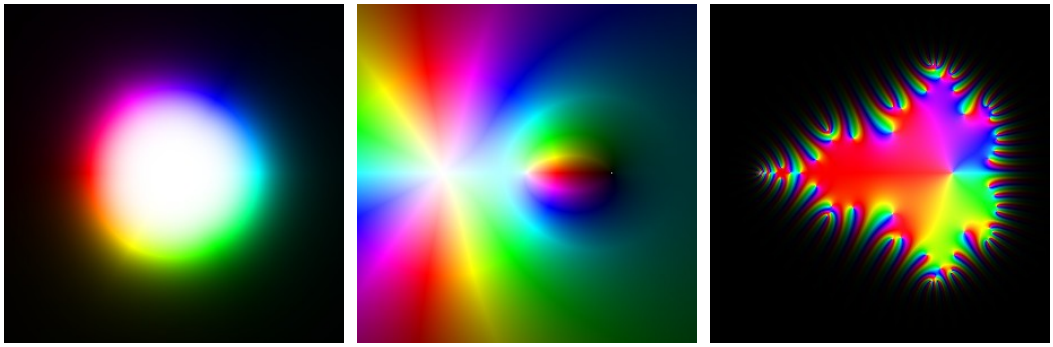


Figure 14: Martin Pergler's domain coloring images

This work of Martin Pergler follows Frank Farris and Hans Lundmark with their domain coloring method. He generated images with Matlab, the Matlab files are available for download. The left image of fig. 14 shows the color in the domain set where he adjusted the rate at which colors collapse into black or white, which he explains as being important to see zeros and poles. The center image shows $(z+1)^2z/(z-1)$, the right image shows the 7^{th} iteration of z^2+c (which creates the Mandelbrot set).

Functions of one Complex Variable - Visualization and Graphical Interpretation [19]

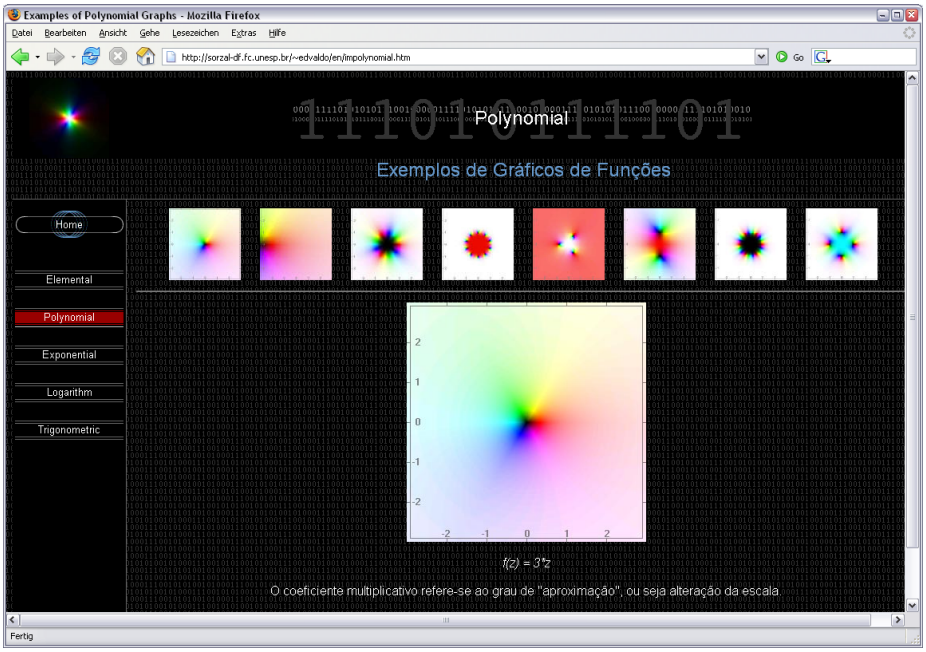


Figure 15: Functions of one Complex Variable web site

This web site by Aguinaldo Robinson de Souza shows well categorized images and animations of important complex functions. The highlight of this work is the catalogue-like character of its representation.

2.2 *Augmented Reality*

Azuma [20] defines Augmented Reality (AR) as a variation of Virtual Reality (VR). As opposed to a complete immersion into a synthetic environment in VR, where the user cannot see the surrounding real world, in AR users still can see the real world, but supplemented with virtual objects through image overlay or composition. An appearance of coexistence of real and virtual objects in the same space would be the ideal result. Milgram et al. [21] writes about a continuum between reality and virtuality with AR being right in the middle. The basic properties of AR are defined by Azuma as:

- Combines real and virtual
- Interactive in real time
- Registered in 3D

As soon as multiple users access a shared space populated by virtual objects superimposed onto reality we speak about a *collaborative AR* environment. "This approach is particularly powerful for educational purposes when users are co-located and can use natural means of communication (speech, gestures etc.), but can also be mixed successfully with immersive VR or remote collaboration." [22].

By running the application implemented in this project inside the Studierstube framework, the manifold features of collaboration in shared augmented reality workspace therein can be taken advantage of.

2.2.1 Science education with AR and VR

AR has been chosen for this project as a technology to build upon, because it has been show in numerous projects to be helpful for instructive and educational applications as a more sophisticated visualization technique than pure display of virtual objects.

Construct3D [22]
This software has been implemented as an educational augmented reality (AR) solution for multiple users (e.g. a teacher and a student) working with geometric primitives (points, lines, bodies) in a shared virtual environment. It allows to intersect objects, generate tangential objects, experiment with boolean operations and other tasks of constructive geometry.

It is technically (being implemented in the Studierstube framework) as well as conceptually a good role example for an instructive mathematical AR application and thereby strongly related this project. As confirmed in the user evaluation results of Construct3D, an important key point for an educational environment is the possibility of social interaction among users in real space while working with virtual objects.

The left image of fig. 16 shows an image composition that demonstrates collaborative construction in Construct3D. The right image shows a typical scene of constructive geometry with points that are bound to the surface of other objects, an intersection curve and projection planes.

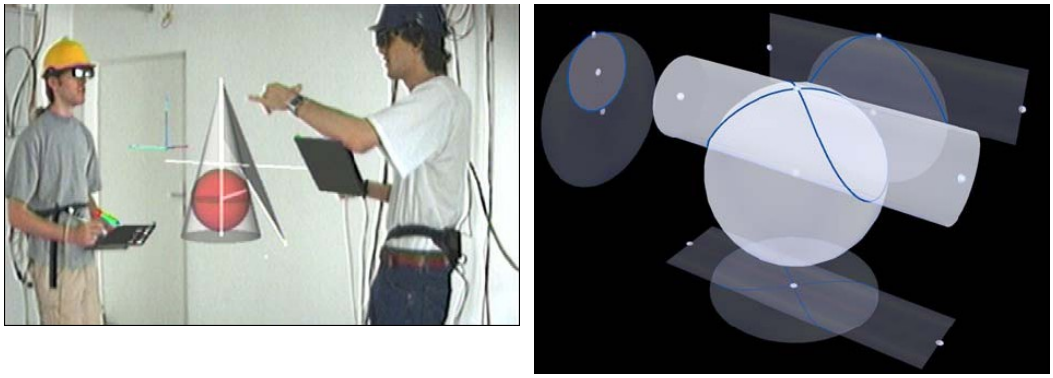


Figure 16: constructive geometry in Construct3D

Cybermath [23]
This is an advanced avatar-based shared virtual environment software. It is designed to allow exploring and teaching mathematics with teaching and learning users being either co-present or physically separated. A desktop setup variant of it exists as well. It implemented a number of mathematical experiments in a virtual space resembling a museum environment, where different dynamic objects are exhibited, for example surfaces that can be interactively manipulated by changing properties or entering equations.

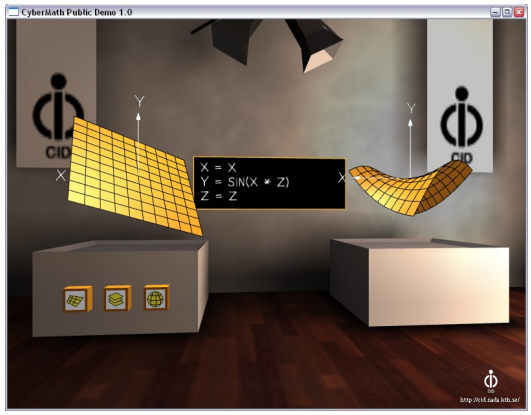


Figure 17: cybermath desktop version screenshot

ScienceSpace [24]
As a collection of three immersive virtual worlds on kinematics and dynamics of one-dimensional motion (NewtonWorld), electrostatic forces (MaxwellWorld) and molecular structures (PaulingWorld) this is another related work in the field of science education in VR.

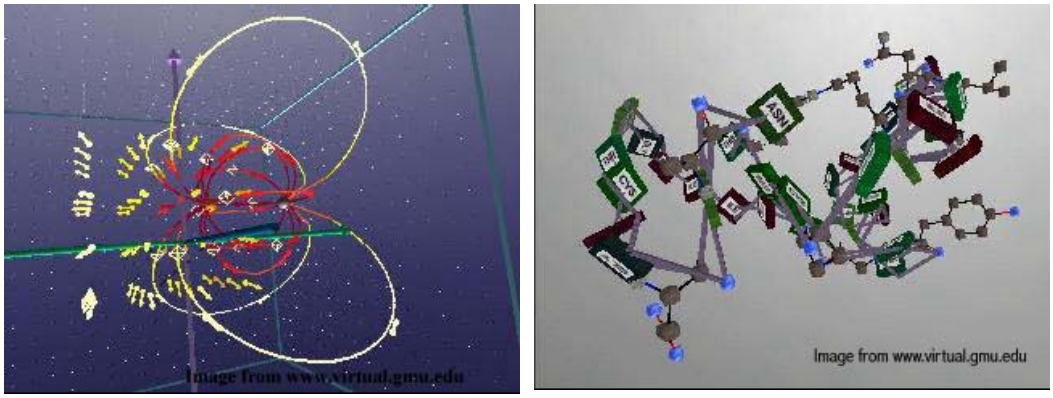


Figure 18: Two views from MaxwellWorld and PaulingWorld of ScienceSpace

VRMath [25]
This recent prototype of a virtual reality learning environment was designed to help children to learn 3D geometry concepts and processes. It allows not only manipulation of objects, but also programming of object creation in a Logo-like programming language.

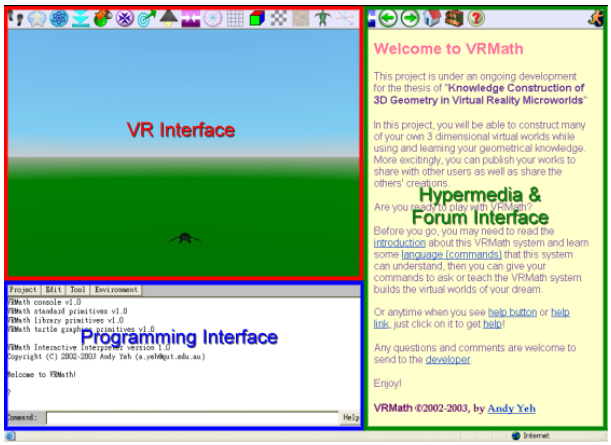


Figure 19: annotated screenshot of VRMath

2.3 *Visualization in Scene Graph APIs*

As this work is built upon the Studierstube framework (see section 3.2) which again is built upon the Inventor scene graph API (see section 3.1), in this section related work dealing with visualization in scene graphs is presented. The actual benefit of this integration will be explained later (section 4.1.1) at the topic of the visualization pipeline.

MeshViz for Mercury Open Inventor
This extension for Mercury (formerly TGS) Open Inventor provides many methods to visualize generic data sets in a scene graph. With the help of the provided scene graph nodes, the functionality of traditional plotting software (as

presented in the survey in appendix A.3) can be combined with the possibilities of rendering a dynamic 3D scene in real time using a scene graph API.

MeshViz takes given raw data (that can be updated dynamically) as input and generated meshes, vector fields, streamlines, many different chart types and more as visual output.

Cash Flow - 3D Flow Data Visualization Framework

Cash Flow is technically another role model for the complex function graph visualization in AR from the aspect, that it implements a complete visualization pipeline (compare fig. 20 and section 4.1.1) in Coin3D. Cash Flow [26] is a framework of scene graph nodes that is specialized on flow data visualization in Coin3D. The important point is, that the parameters of the whole visualization pipeline are interfaced by the scene graph API and thereby interactively controllable. Despite the similarities of its design to the application described in this thesis, the implementation was solved differently in the details, mostly because of the different characteristics of the data at the beginning of the data flow.

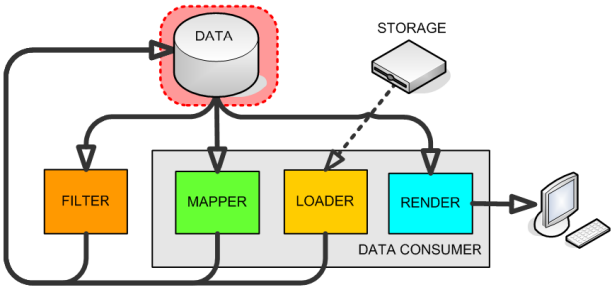


Figure 20: Visualization Pipeline in a Scene Graph API: Data Flow Model of Cash Flow

3 Technological Foundations

3.1 Inventor and Coin3D

Open Inventor (OIV) is an 3D graphics API designed by SGI (originally as "IRIS Inventor") as a library on a higher level than OpenGL that provides scene graph functionality. The key point of working with scene graphs is to operate with object descriptions (their shape, size, coloring, surface texture, location in 3D space) rather than with drawing primitives. OIV is implemented object orientated in C++ and uses OpenGL for rendering. It has been released under an open source licence in 2000 by SGI, but has not been further developed by this company since then. Coin3D ("Coin Open Inventor") is a clone library of this API, written from scratch and is still actively maintained. Coin3D has been chosen as the technical foundation for this project.

The architecture supports an event driven programming style, i.e. typically the application reacts to events from the framework by means of callback functions (this is called *sensors* in Inventor).

Despite its origins from about 15 years ago, the design of Inventor still can compete with more recent developments, it uses many sophisticated concepts (e.g. so-called lazy evaluation) that allow high performance real time behaviour.

Using Inventor for the complex function graph visualization framework implemented in this project means, that new classes where derived in C++ from the base classed provided by the Coin3D API. Therefore it is possible to include these visualization objects into a scene in an application that supports Coin3D.

In the next chapters (especially in the section 4.2 on implementation) some basic knowledge of the API is required. See the excellent book "The Inventor Mentor"[27] on this topic. When in the following a class name from the Inventor API is mentioned it is easy to recognize by its So... prefix (that stands for scene object).

3.1.1 Nodes

Every scene graph object in Inventor is referred to as a node. Its base class SoNode provides functionality to aggregate object data in so-called *fields*. Fields are more powerful than ordinary member variables, because they allow callback functions (sensors) to be attached to them and to synchronize automatically with other fields (*field connections*).

Nodes support serialization to and from an ASCII based text format (*.iv files), a feature that is used in this project to input a scene graph by such a file.

Nodes include member functions that respond to different kinds of scene graph traversal (called *actions*). The most obvious action for a scene graph is the GLRenderAction, which calls the appropriate OpenGL functions to draw itself. But other actions are supported as well, for example the WriteAction to write itself to a scene graph file or the SearchAction to find a node with certain properties in the scene graph.

The directed acyclic graph (DAG) structure of the scene graph is achieved via *group nodes*, which collect a number of nodes (*children*) and forward traversal function calls to these children recursively (depth first, then left to right).

3.2 Studierstube

Studierstube [28] is a framework of Inventor nodes, that extend the scene graph API by objects needed for collaborative AR applications. These extensions include interaction based on 3D tracking events, a workspace with multiple 3D windows, rendering and output modes for common VR/AR output devices (stereoscopic viewing etc.), distributing applications, managing multiple users in a single setup and an interface to a generic tracking layer called OpenTracker [29].

In the Studierstube framework each user can have his/her own 3D pointing device (a pen tracked in 6 DOF) with a button and move around draggable objects just like the familiar drag-and-drop with the mouse in 2D.

For this project the developed scene graph objects for the visualization have been put into a Studierstube 3D window inside a Studierstube application. Configuration files for multiple users have been set up to work together interacting with the mathematical visualization.

4 The Complex Function Graph Visualization Node Kit for Inventor

During this master thesis project a software framework has been implemented, that makes the visualization of complex function graphs easily possible within the graphics framework Inventor (see section 3.1) and the AR/VR framework Studierstube (section 3.2).

Extending the Inventor scene graph library is done by implementing new nodes (see section 3.1.1) or collections of nodes, so-called node kits. While section 4.1 explains the design, the ideas behind the structure of the developed node kit, section 4.2 presents the implementation with all of its details, including all information necessary to use the framework.

As this chapter keeps its focus on the solution inside the scene graph, some words on the general workflow for using these nodes have to be placed at this point. In its executable version the developed application will be started using the Studierstube workspace executable and providing a scene description file as a parameter. This scene description is a scripted graph of nodes, in our case including some of the new developed nodes explained in the following.

4.1 The Framework's Design

4.1.1 A Visualization Task

The overall design goal for the implemented framework was to fulfill a visualization task, which is commonly broken down into the steps of the so-called visualization pipeline [30]:

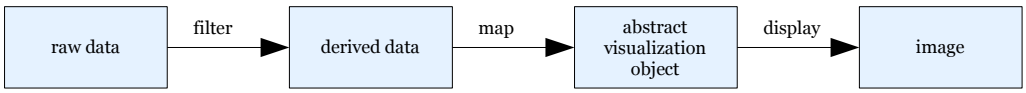


Figure 21: visualization pipeline

In our case of graph visualization the raw data is an infinite set of numbers. The raw data are all numbers of the domain set and the codomain set (these numbers make up the mathematical concept which is called a graph, explained in section 4.1.2).

In the framework developed in this project, the first transition in the pipeline, namely the data filtering step, is rather a data generation step. This is

because the raw data is not a set of numbers given explicitly, but an infinite set given implicitly by a data generation method (evaluating a mathematical function). So the user filters, or better said, selects a subset of data (the domain set to be evaluated) from an infinite set (the complete set of complex numbers $\mathbb{C}$).

The second transition in the pipeline, often called the "visual mapping", generates scene graph objects from the derived data. Finally the scene graph is rendered by the help of a scene graph library.

Most importantly, the described implementation allows an interactive control of all steps of the pipeline including the data generation. The corresponding image is rendered in real time. A classification of visualization scenarios shown in [31] points out, that such a type of visualization ("interactive control") is the most advanced one compared to "passive visualization" (only control of display parameters, the last step in the pipeline) and "interactive visualization" (control filtering, mapping and display for a static set of data).

4.1.2 Function Graph Specific Visualization Task

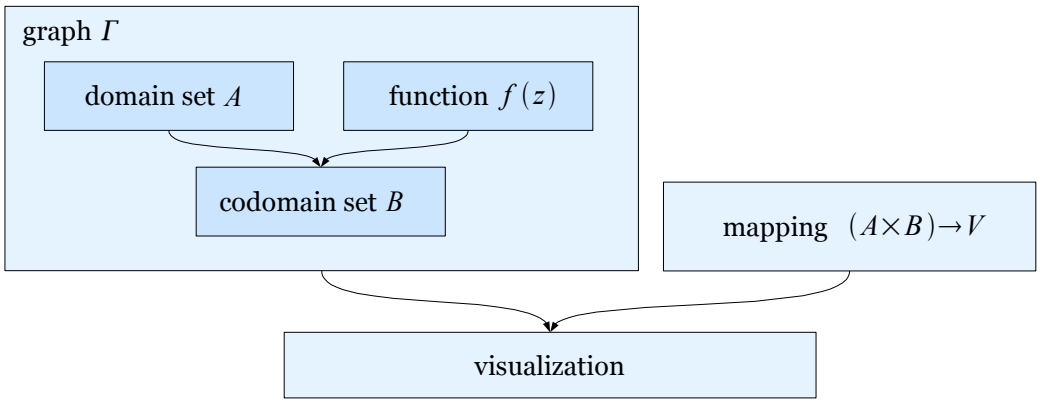


Figure 22: graph visualization concept

The concept of visualizing complex function graphs (filtering and mapping part of the visualization pipeline) has been broken down into the following elements:

- define the graph $\Gamma_f = \{(z, f(z)) \in A \times B \mid z \in A\}$:
  - input a complex function $f(z)$
  - input a finite set of domain values (set $A \subset \mathbb{C}$)
  - this leads to the automatic calculation of all codomain values (set $B \subset \mathbb{C}$)
- define a mapping from the the graph space ($A \times B$) to a visualization space V with dimensions that define the position, shape, color and animation of a geometric visualization object
- manipulate the function, the domain set and the graph $\rightarrow$ visualization mapping in real time to explore properties of the function

4.1.3 Dimensionality Issues

An interesting issue of visualizing complex function graphs is, that $A \times B$ with $A \subset \mathbb{C}$ and $B \subset \mathbb{C}$ has 4 dimensions: a number in $\mathbb{C}$ can be seen as a number in $\mathbb{R}^2$, the so-called complex plane; so $\mathbb{C} \times \mathbb{C}$ can be seen as $\mathbb{R}^4$. This is regarding to visualization much more challenging than a graph of a real valued function in $A \times B$ with $A \subset \mathbb{R}$ and $B \subset \mathbb{R}$ which can easily be visualized as a curve on a plane ($\mathbb{R} \times \mathbb{R} \equiv \mathbb{R}^2$).

In order to gain insight into graph features in all these 4 dimensions the framework that has been implemented offers about 30 dimensions in the visualization space V:

- 3 dimensions of location in space
  - multiplied by 4 due to implementation of an extension of parallel coordinates: one "point" $g \in \Gamma$ is represented as a line connecting a set of maximal 4 points in space
- 6 dimensions of affine transformations of objects
  - 3 DOF rotation
  - 3 DOF scaling
- 4 dimensions of shading
  - 3 dimensional color space, defined by HSV or RGB color model. If color is defined in both models, the colors are blended.
  - 1 dimension through transparency
- 7 dimensions of time-varying visualization
  - frequency of an oscillation between two affine transformations
  - amplitudes of this oscillation in 6 dimensions
    - 3 DOF rotation animation
    - 3 DOF scaling animation

This oversupply of visualization dimensions might seem to counteract the goal of gaining insight into the graph. But the important point, when using all these possibilities to reach the goal of insight, is to follow three rules:

1) use as many visualization dimensions as needed.
2) use as few visualization dimensions as possible.
3) find the best visualization dimension for each graph dimension (there is no optimal mapping that's best for all functions and all setups)

Every visualization dimension has its own strengths, distinguishable in measures (of how good they can be overseen by the user) as:

- range of values
- cyclicity ($[0, r] + k * r, \; k \in \mathbb{Z}$), one sided infinity ($[0, \infty]$) or two sided infinity ($[-\infty, \infty]$)
- resolution
- well combination with other dimensions without interdependencies

4.1.4 Domain Set Issues

Section 4.1.1 explained, that the data filtering step in the visualization pipeline is rather an selection step, it deals with selecting the domain set to be evaluated, a finite subset of $\mathbb{C}$.

Visualizing the complex function for a single domain value is trivial. The user gives the domain value, the function and the $\Gamma \to V$ mapping (see section 4.2.5) as input, the framework calculates the codomain value, the graph space vector $g \in \Gamma$ is mapped to the visualization space vector $v \in V$ and an object is generated with the location, color, transformation and animated transformation specified in v .

But more likely the user will be interested in the graph of a set of domain values. Existing graph visualization solutions show that sets of domain values are usually not drawn as a set of points but rather as a curve or a mesh connecting these points. This has a number of reasons:

- the connecting geometry shows an approximation of the values between the evaluated samples
- it clarifies adjacency relations: in a sampled curve each point (except the end points) has 2 neighbours; a connecting line acts as if each point had "pointers" to its neighbours
- readability: lines are easier to see than single points
- impression of curvature: the human eye is sensitive to curvature (2^{nd} derivation), which might carry interesting information about the function

Connecting adjacent points of a subset has one disadvantage however: if there is a discontinuity between two adjacent samples, a connection will result in a misleading representation of this section.

In the described framework the user can define one- and two-dimensional arrays of domain values (referred to as "1D subset" and "2D subset" in the following) . The generated geometric object will be drawn as a curve or mesh instead of single points. The user defines the domain set as:

- "domain set" = set of "domain subsets"
- "domain subset" = "single point" or "1D subset" or "2D subset"
- the "single point" subset is called "0D subset" for class name consistency
- "1D subset" = sampled "line" or "circle"
- "2D subset" = sampled "rectangular grid" or "polar grid"

By implementing regularly sampled lines, circles, rectangular grids and polar grids in the presented framework the user only has to give the range and the resolution as input to obtain any number of domain values.

4.2 Implementation

Fig. 23 shows the objects and data involved in the data flow in the implementation of the complex function graph visualization framework for the Inventor scene graph API. This figure shall in addition serve as a reference to the reader to locate each of the objects and implemented scene graph nodes, that are described in all of section 4.2 in detail, in the framework's architecture.

4.2.1 Overview of Implemented Classes

All entities shown in fig. 22 are encapsulated in a node kit named **SoComplex-FunctionGraphVisualization**. To organize the scene graph structure more closely to the conceptual entities, substructures of this visualization framework have be implemented as further node kits:

- a node kit named **SoComplexFunctionEvaluator** which describes the graph Γ (SoComplexFunctionGraphVisualization contains a part named *functionEvaluator* of this type; it represents the left box in fig. 22)
- a node kit named **SoMappingGraphPropertyToVizFeature** which describes the graph → visualization mapping (SoComplexFunctionGraph-Visualization contains a list of parts named *mappingList* of this type; this list represents the right box in fig. 22). Detail on this can be found in 4.2.5.

To conclude this as short as possible: a “graph visualization” consists of a “function evaluator” and a “mapping list”.

Now all elements of fig. 22 (concept level) can be brought into the context of fig. 23 (implementation level). An instance of SoComplexFunctionEvaluator contains the domain set, the function, and in dependency of them, the codomain set. The domain set is stored in a number of **SoComplexDomainSubSet** objects, the function in a field of the SoComplexFunctionEvaluator object and the codomain set in an internal array of complex numbers of the same object.

We could follow these objects also from the top to the bottom of this data flow graph: first the user has to define the domain set in a number of SoComplexDomainSubSet objects. In a next stage the domain set is used as the input for a SoComplexFunctionEvaluator object that combines the domain set with a function to a codomain set. In a third stage the combination of domain set and codomain set (called “graph”) is taken from the function evaluator into a visualization object that combines the graph with a description of the visual mapping to a graphical object.

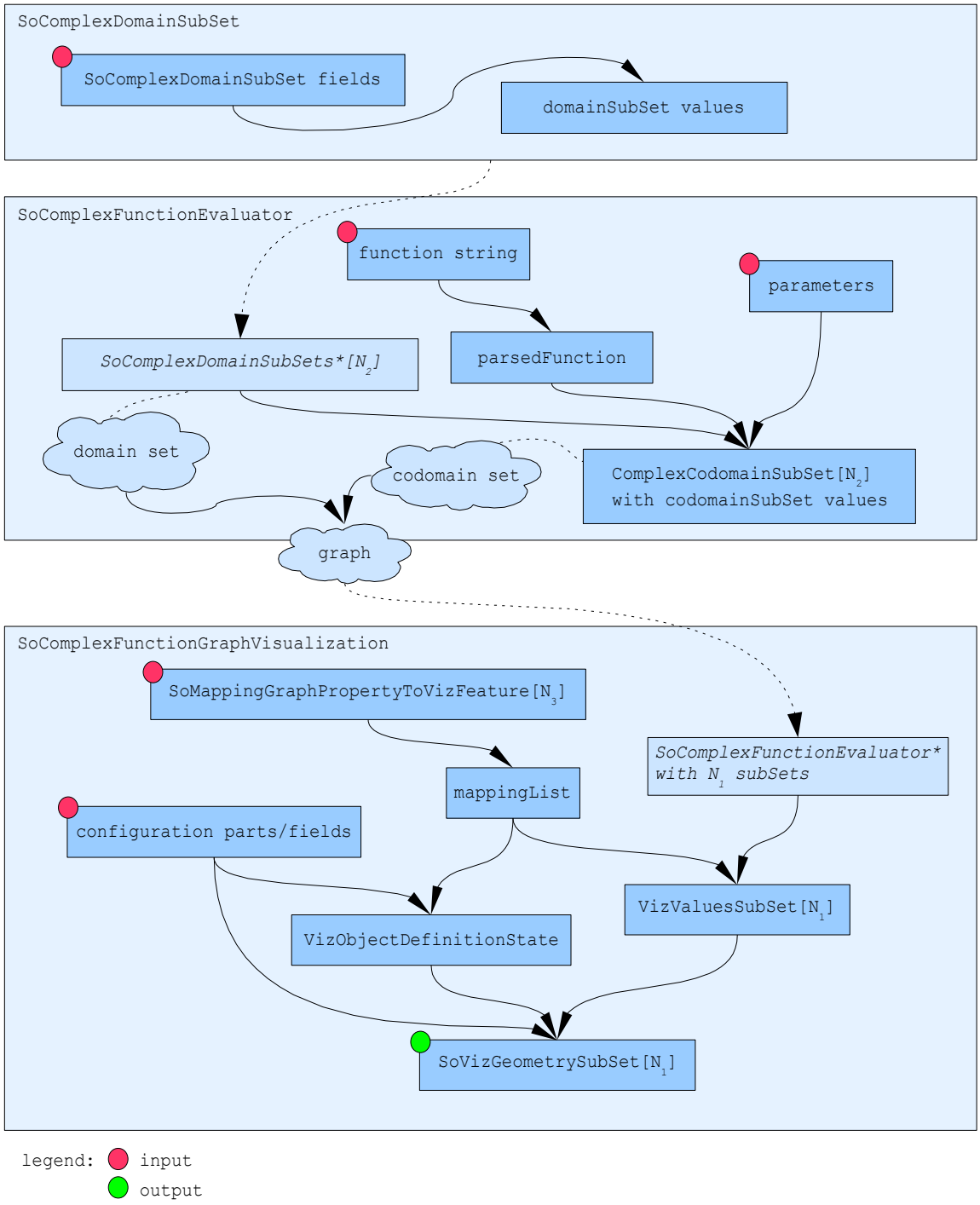


Figure 23: how graph visualization geometry is generated from input data

4.2.1.1 SoComplexFunctionGraphVisualization Internals

Fig. 23 shows, that there is even more behind this concept. It shows in the data flow inside SoComplexFunctionGraphVisualization, that there is an internal entity called **VizValuesSubSet** between (a) the two aforementioned parts *functionEvaluator* and *mappingList* in the upper part and (b) the

SoVizGeometrySubSet on the bottom end of the flow graph. More precisely this entity is an array of objects of the class VizValuesSubSet, that's why it's written VizValuesSubSet[N_1]. N_1 is the number of subsets of the domain set A of the *functionEvaluator*. VizValuesSubSet[N_1] represents the "visualization values set", a set of vectors in visualization space, one vector for each member of the domain set A.

But still VizValuesSubSet[N_1] is nothing more than a set of numbers. These numbers are input of another internal entity of type **SoVizGeometrySubSet**[N_1], again an array of size N_1. This class, the "visualization geometry set", is finally the scene graph node which responds to the render action by inserting nodes derived from SoShape and optionally other nodes that affect the traversal state (as for example SoMaterial or SoTransformation nodes) into the scene.

And yet another concept was introduced in the implementation: because the VizValuesSubSet[N_1] as described later in this chapter doesn't define the SoVizGeometrySubSet[N_1] unambiguously, another class named **VizObjectDefinitionState** has been implemented (SoComplexFunctionGraphVisualization internally stores one instance of this class) and additional parts and fields have been included into the SoComplexFunctionGraphVisualization node kit (they are called "configuration parts/fields" in the following). Section 4.2.6 explains this in more detail.

This concludes an overview of the internal structure of the SoComplexFunctionGraphVisualization node kit and should have explained how graph visualization geometry is generated using a given *functionEvaluator* and *mappingList*. What is missing is some more detail on SoComplexFunctionEvaluator which represents the graph Γ.

4.2.1.2 SoComplexFunctionEvaluator Internals

The function evaluator node kit hosts three entities accessible by the user: (1) the domain set, (2) the function and (3) function parameters. Internally it additionally contains an object named *parsedFunction* and the codomain set. The *parsedFunction* contains a method which can evaluate one member z of the domain set A and returns the associated member $f(z)$ of the codomain set B.

For details on the manifold possibilities of defining the domain set see section 4.1.4.

The function itself is given as a SoSFString field of SoComplexFunctionEvaluator named *function*. Any time *function* is updated, *parsedFunction* runs its parsing method. Section 4.2.3 explains the function input in detail.

The function parameters are pairs of SoSFFloat fields of the function evaluator, which define the complex number constants $p_1, p_2, p_3, p_4 \in \mathbb{C}$. These parameters can be used in the function. The evaluation method of parsedFunction will use their current values without having to run its parsing method when these parameters change. See section 4.2.4 on this.
SoComplexFunctionEvaluator detects changes in any of its inputs and automatically invokes the calculation of all values of the codomain set using (1)

the values of the domain set A stored in SoComplexDomainSubSets*[N_2], (2) the function parameters and (3) the evaluation method of *parsedFunction* and stores these values in ComplexCodomainSubSet[N_2] which represents the codomain set B. Providing all values from A and B SoComplexFunctionEvaluator fully describes the graph that we want to visualize.

4.2.2 Domain Set Input

In the SoComplexFunctionGraphVisualization node kit the domain set is kept inside its *functionEvaluator* part. The functionEvaluator is a node kit of type SoComplexFunctionEvaluator and has a NodeKitListPart named *domainSubSetList*. NodeKitListParts are Inventor nodes similar to SoGroup nodes, but they restrict the type of the children that can be added to them. The list part *domainSubSetList* is restricted to the type SoComplexDomainSubSet and its subclasses. SoComplexDomainSubSet itself is an abstract class, 5 non-abstract subclasses have been implemented, they are explained in detail later in this section:

(1) SoComplexDomainSubSet_0D (subsection 4.2.2.1)
(2) SoComplexDomainSubSet_1D_RegularLine (subsection 4.2.2.2)
(3) SoComplexDomainSubSet_1D_RegularCircle (subsection 4.2.2.3)
(4) SoComplexDomainSubSet_2D_RegularRectangular (subsection 4.2.2.4)
(5) SoComplexDomainSubSet_2D_RegularPolar (subsection 4.2.2.5)

That means for an actual instance of the graph visualization (see fig. 23):

- *domainSubSetList* hosts any number of domain subsets
- *functionEvaluator* internally keeps a corresponding list of codomain subsets
- the SoComplexFunctionGraphVisualization instance, which hosts the *functionEvaluator,* keeps a corresponding list of visualization values subsets and a corresponding list of visualization geometry subsets.

To conclude this: in the data flow of the graph visualization, subsets are defined by the user in the domain set. Such a subset definition automatically invokes the instantiation of 4 subsets: the domain subset, the codomain subset, the visualization values subset and the visualization geometry subset. Each of the 4 corresponding subsets have the same number of elements.

The perhaps most sophisticated detail about this implementation is, that shared instancing is supported throughout all the framework. In fig. 23 two objects (“SoComplexFunctionEvaluator*” and “SoComplexDomainSubSets*[N_2]”, i.e. the *functionEvaluator* and the *domainSubSetList*) are colored light blue and have dotted lines connecting them to another box. This illustration and the * in their name (as an allusion to C++'s pointer notation) should show, that the hosting objects only keep pointers to the function evaluator or to the domain subsets. So there is no one-to-one relationship between the 4 aforementioned corresponding subsets. E.g. two SoComplexFunctionGraphVisualization instances might use the same SoComplexFunctionEvaluator instance, which means that for each domain subset instance there is *one* corresponding codomain subset instance in the function evaluator but *two* different

visualization values subset instances and *two* different visualization geometry subset instances (one in each graph visualization object).

As another example three SoComplexFunctionEvaluator instances might use the same SoComplexDomainSubSet. In this case there would be *one* domain subset instance and *three* codomain subset instances. The number of visualization values subset instances and visualization geometry subset instances again depends on how many graph visualization objects use each function evaluator instance.

Because of this indirect connection of subsets two different variables for the number of subsets have been used in fig. 23: N_1 and N_2. N_1 is the number of subsets for the actually used *functionEvaluator* in SoComplexFunctionGraphVisualization whereas N_2 is the variable for the number of subsets in any instance of SoComplexFunctionEvaluator.

The domain subsets are defined as Inventor nodes as described in the subsections below:

4.2.2.1 SoComplexDomainSubSet_0D

This node is the most simple of all subsets, it always has exactly one member. The value of this single member is defined by SoComplexDomainSubSet_0D's float fields *real* and *imaginary*.

In the Inventor file format of this node looks as follows, the values shown are the default values:

```
SoComplexDomainSubSet_0D {
  real                0.0
  imaginary           0.0
}
```

4.2.2.2 SoComplexDomainSubSet_1D_RegularLine

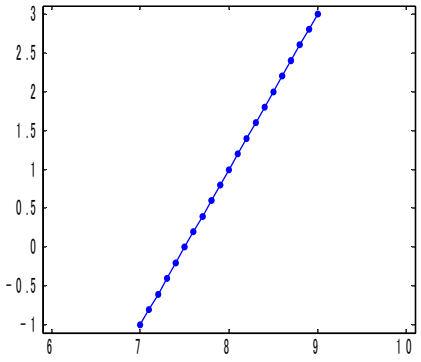


Figure 24: example of a sampled line domain subset

```
SoComplexDomainSubSet_1D_RegularLine {
 realStart   7.0
 imagStart  -1.0
 realEnd     9.0
 imagEnd     3.0
 nrSegments  20
}
```

The node SoComplexDomainSubSet_1D_RegularLine defines a regularly sampled line in the complex plane using the float fields *realStart*, *imagStart*, which define the value of the line's start point, the float fields *realEnd*, *imagEnd*, which define the value of the line's end point and the int field *nrSegments*, which defines the resolution of the samples. Fig. 24 shows an example of the domain values that would be calculated internally from the given field values. It results in a line from 7-1i to 9+3i with 21 samples. *nrSegments* defines the number of

samples minus 1. This input method has been chosen because the user rather sees and thinks in a number of segments, than in a number of vertices.

The default values of this node are:

```
SoComplexDomainSubSet_1D_RegularLine {
  realStart          -1.0
  imagStart           0.0
  realEnd             1.0
  imagEnd             0.0
  nrSegments          10
}
```

4.2.2.3 SoComplexDomainSubSet_1D_RegularCircle

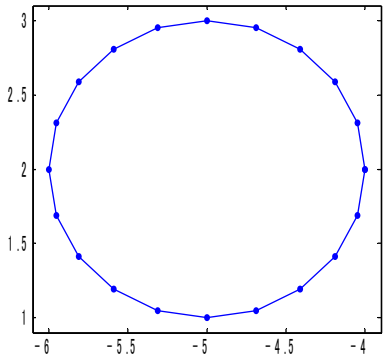


Figure 25: example of a sampled circle domain subset

```
SoComplexDomainSubSet_1D_RegularCircle {
  centerPointReal  -5.0
  centerPointImag   2.0
  radius            1.0
  nrSegments        20
}
```

The node SoComplexDomainSubSet_1D_RegularCircle defines a regularly sampled circle in the complex plane using the float fields *centerPointReal*, *centerPointImag*, which define the value of the circle's center, the float field *radius*, which defines the circle's radius and the int field *nrSegments*, which defines the resolution as in ..._RegularLine (see subsection above), but here the number of segments is equal to the number of vertices because the curve's start and end points coincide. Fig. 25 shows an example with the field values stated next to it, which results in a circle around -5+2i using radius 1 with 20 samples.

Default values are:

```
SoComplexDomainSubSet_1D_RegularCircle {
  centerPointReal     0.0
  centerPointImag     0.0
  radius              1.0
  nrSegments          10
}
```

4.2.2.4 SoComplexDomainSubSet_2D_RegularRectangular

The node SoComplexDomainSubSet_2D_RegularRectangular defines a grid of values in the complex plane sampled regularly in Cartesian coordinates using the float fields *realStart*, *realEnd*, *imagStart* and *imagEnd* which define a range in the real axis and a range in the imaginary axis and the int fields *realNrCells*, *imagNrCells* which specify the resolution, again as the number of segments instead of the number of vertices. Fig. 26 shows an example that results in a grid between -2-2i and 2+2i with 9*17 samples (which shows 8*16 cells).

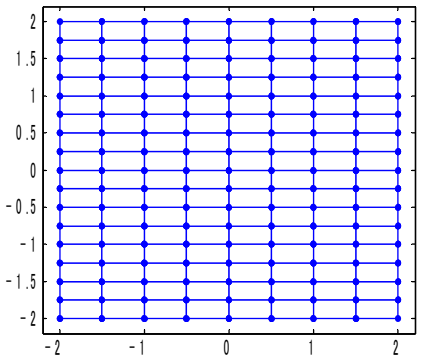


Figure 26: example of a sampled Cartesian grid domain subset

```
SoComplexDomainSubSet_2D_RegularRectangular
{
  realStart          -2.0
  realEnd             2.0
  realNrCells         8
  imagStart          -2.0
  imagEnd             2.0
  imagNrCells         16
}
```

The default values of this node are:

```
SoComplexDomainSubSet_2D_RegularRectangular {
  realStart          -1.0
  realEnd             1.0
  realNrCells         10
  imagStart          -1.0
  imagEnd             1.0
  imagNrCells         10
}
```

4.2.2.5 SoComplexDomainSubSet_2D_RegularPolar

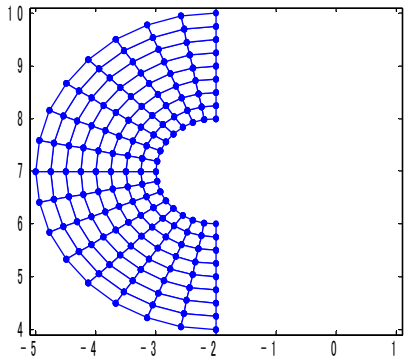


Figure 27: example of a sampled polar grid domain subset

```
SoComplexDomainSubSet_2D_RegularPolar {
 angleStart          1.5708    # 0.5*PI
 angleEnd            4,7124    # 1.5*PI
 angleNrSegments     16
 magnitudeStart      1.0
 magnitudeEnd        3.0
 magnitudeNrSegments 8
 centerPointReal    -2.0
 centerPointImag     7.0
}
```

The node SoComplexDomainSubSet_2D_RegularPolar defines a range of values in the complex plane, sampled regularly in polar coordinates using the float fields *angleStart*, *angleEnd*, *magnitudeStart*, *magnitudeEnd*, which define a set of complex numbers located like a disc section around a center given by the float fields *centerPointReal*, *centerPointImag* and subdivided into a number of segments given by the int fields *angleNrSegments*, *magnitudeNrSegments*.
Fig. 27 shows an example that results in a polar grid around -2+7i between radius 1 and 3 and between angle 90° and 270° with 17*9 samples (which shows 16*8 cells).

This nodes default values:

```
SoComplexDomainSubSet_2D_RegularPolar {
  angleStart          0.0
  angleEnd            6.28318530718
  angleNrSegments     10
  magnitudeStart      0.001
  magnitudeEnd        5.0
  magnitudeNrSegments 10
```

```
    centerPointReal     0.0
    centerPointImag     0.0
  }
```

4.2.2.6 domainSubSetList input

To input a domain set in Inventor file format using these domain subsets explained above would look like this example:

```
SoComplexFunctionEvaluator {
  domainSubSetList NodeKitListPart {
    containerNode Separator {

      SoComplexDomainSubSet_0D {
        # fields ...
      }
      SoComplexDomainSubSet_1D_RegularLine {
      } # no fields given -> using default values
      SoComplexDomainSubSet_1D_RegularLine {
        imagStart           1.0
        imagEnd             1.0
      } # fields partially given ->
        # using default values for the other fields
      SoComplexDomainSubSet_2D_RegularRectangular {
        # fields ...
      }

    } # end of containerNode
  } # end of domainSubSetList
  # function etc. ...
} # end of SoComplexFunctionEvaluator
```

4.2.3 Function Input

Section 4.2.1.2 already mentioned, that the *function* is a SoSFString field of SoComplexFunctionEvaluator. This field is observed by a field sensor which invokes a parsing method of a parser class instance (named *parsedFunction*) hosted by the *functionEvaluator* object.

A parser has been implemented, that wraps all functions and operators offered by the C++ Standard Template Library (STL) which take complex numbers as operands. The grammar for the string to be parsed is explained by this extended Backus-Naur form (EBNF):

```
expr      = stm;
stm       = const | domVal | param |
            oneOpStm "(" stm ")" | twoOpStm "(" stm "," stm ")" ;
            (* the first three symbols are terminating *)
            (* the last two symbols are operations *)
const     = "(" realNr "," realNr ")" ;
realNr    = ? stringWithoutCommaAndBraces ? ;   (* terminating *)
            (* must be parsable with c++ standard library as real
               number *)
domVal    = "z" ;                                (* terminating *)
param     = "p1" | "p2" | "p3" | "p4" ;          (* terminating *)
oneOpStm  = "abs" | "arg" | "conj" | "cos" | "cosh" | "exp" |
            "imag" | "log" | "log10" | "norm" | "real" | "sin" |
            "sinh" | "sqrt" | "tan" | "tanh" | "neg" ;
```

```
twoOpStm  = "polar" | "pow" | "mul" | "div" | "add" | "sub" ;
```

`expr` stands for *expression*, `stm` for *statement*, `const` for *constant*, `realNr` for *real number*, `domVal` for *domain value*, `param` for *parameter*, `oneOpStm` for *one-operator statement*, `twoOpStm` for *two-operator statement*.

As an example the function $f(z)=z^2+c$ would be written as the string `add(pow(z,(2,0)),p1)`, which defines the right hand side of the equation, the left hand side is always $f(z)$. Parsing this example would result in the following tree of symbols:

```
expr                          add(pow(z,(2,0)),p1)
    stm                           add(pow(z,(2,0)),p1)
        twoOpStm                      add
        "("                           (
        stm                           pow(z,(2,0))
            twoOpStm                      pow
            "("                           (
            stm                           z
                domVal                        z
            ","                           ,
            stm                           (2,0)
                const(2,0)                    (2,0)
            ")"                           )
        ","                           ,
        stm                           p1
            param                         p1
        ")"                           )
```

This representation shows the EBNF symbols on the left side and their content on the right side, the indentation displays the hierarchy.

The implemented grammar is kept very simple, e.g. binary operators like “op1+op2” have been wrapped as a function with 2 operands like “add(op1, op1)”. Nevertheless all STL complex number function are accessible using this grammar.

A preprocessor has been included into the parser, it removes all white-spaces in the string. This enables the user to enter the function string in a more readable style.

In the Inventor file format the function input looks like this, the shown value $f(z)=z$ is the default value:

```
SoComplexFunctionEvaluator {
  domainSubSetList { ... }
  function         "z"
}
```

4.2.4 Parameter Input

If the user wants to explore a function like $f(z)=z^2+c$ with a parameter $c\in\mathbb{C}$, with the *parsedFunction* object alone, this variable number would have to be inserted as characters into the *function* string and *parsedFunction* would have to parse all the function again and again.

In order to improve performance in this case, the following 8 SoSFFloat fields have been added to the SoComplexFunctionEvaluator node class (the shown values are the default values):

```
SoComplexFunctionEvaluator {
  domainSubSetList { ... }
  function          ...

  parameter1Real    0.0
  parameter1Imag    0.0
  parameter2Real    0.0
  parameter2Imag    0.0
  parameter3Real    0.0
  parameter3Imag    0.0
  parameter4Real    0.0
  parameter4Imag    0.0

}
```

These fields define $p_1, p_2, p_3, p_4 \in \mathbb{C}$. In the grammar explained in section 4.2.3 we again find the symbols `p1`, `p2`, `p3`, `p4`. The *parsedFunction* object stores these 4 complex number parameters, they are updated by the help of SoComplexFunctionEvaluator's field sensors. When changes occurred to *parsedFunction,* all the codomain set has to be reevaluated. SoComplexFunctionEvaluator takes care, that the graph data is up-to-date.

4.2.5 Mapping Input

The visual mapping, the mapping of vectors from graph space $g \in \Gamma$ to vectors of the visualization values set $v \in V$, can be thought of a mapping matrix in $\Gamma \times V$ (see table 1 shown below to get an overview). Each element of this matrix is set to 1 or 0 if the user wants to map a specific component of a source vector g (column of the matrix) to a specific component of a target vector v (row of the matrix) or not. Vector components in this context are real numbers, so the complex numbers in g are represented by two real components each and coordinates or HSV color values in v are represented by triplets of real components. Multiple components of a source vector g can be mapped to the same component of the target vector v, which results in the sum of the selected g-components being mapped to the selected v-component. If nothing is mapped to a v-component it will be zero.

Table 1 can be used as an useful help to plan the setup of a mapping matrix for a specific graph visualization task. It shows the graph space vector on top and the visualization space vector on the left. The cells of the table could be seen as check boxes, where the user can turn on and off a mapping from one component on the top to another component on the left. By default all fields of this matrix are zero, i.e. turned off.

Before explaining more details on the mapping itself, all components of the vectors g and v have to be explained:

	DOMAIN_REAL	DOMAIN_IMAGINARY	DOMAIN_MAGNITUDE	DOMAIN_ARGUMENT	CODOMAIN_REAL	CODOMAIN_IMAGINARY	CODOMAIN_MAGNITUDE	CODOMAIN_ARGUMENT	CONSTANT
X1									
Y1									
Z1									
X2									
Y2									
Z2									
X3									
Y3									
Z3									
X4									
Y4									
Z4									
COLOR_H									
COLOR_S									
COLOR_V									
COLOR_R									
COLOR_G									
COLOR_B									
COLOR_ALPHA									
X1_ROTATION_X									
X1_ROTATION_Y									
X1_ROTATION_Z									
X1_SCALE_X									
X1_SCALE_Y									
X1_SCALE_Z									
ANIM_FREQUENCY									
X1_ROTATION_X_ANIM									
X1_ROTATION_Y_ANIM									
X1_ROTATION_Z_ANIM									
X1_SCALE_X_ANIM									
X1_SCALE_Y_ANIM									
X1_SCALE_Z_ANIM									

Table 1: the mapping matrix from graph space to visualization space

4.2.5.1 Graph Space Components (Source Vector)

The graph space is defined by the Cartesian product of the domain set and the codomain set (see section 4.1.2). So vectors $g \in \Gamma$ would have 4 components in $\mathbb{R}$: the domain value's real part, its imaginary part, the codomain value's real part and its imaginary part. The identifiers to be used are:

- DOMAIN_REAL
- DOMAIN_IMAGINARY
- CODOMAIN_REAL
- CODOMAIN_IMAGINARY

First experiments during prototype implementation showed, that very often mappings from the polar coordinates of graph values are of interest, so 4 redundant components in $\mathbb{R}$ have been included into the mapping matrix: the domain value's magnitude, its argument, the codomain value's magnitude and its argument. They relate to the real part r and the imaginary part i as $magnitude = \sqrt{r^2 + i^2}$ and $argument = \arctan(\frac{i}{r})$. The component identifiers are:

- DOMAIN_MAGNITUDE
- DOMAIN_ARGUMENT
- CODOMAIN_MAGNITUDE
- CODOMAIN_ARGUMENT

Finally even one more component has been found to be missing in the graph data set: constant values. They can be used to offset the graph in any of the visualization dimensions. For some components of v, as for example the coordinates in space, the same result could be easily achieved by an transformation node before the SoComplexFunctionGraphVisualization kit, but other components as color or animation would be hard to control from outside the visualization kit. Its identifier has been named:

- CONSTANT_ZERO

4.2.5.2 Visualization Space Components (Target Vector)

Section 4.1.3 already mentioned, that this framework offers manifold visualization targets, almost everything a scene graph library as Inventor has to offer: location, affine transformations, shading and animation. A combination of 32 real value components makes up the visualization values vector (in the "visualization space") and is explained in the following.

The first group of target components are the coordinates. They define the location of single points, but as well the shape of the complete graph visualization object in two ways:

1. evaluating one- and two-dimensional arrays of domain set points, as explained in section 4.1.3, results in curve/mesh objects with their shape given by the coordinates
2. the visualization values set vector provides not one, but four separate space coordinate components. This enables techniques as parallel coordinates and others shown in section 4.2.6. With additional configurations, as explained there, each vector of the graph set would result in up to four points in space, connected by a line. Such a line is common presentation technique for higher dimensional vectors, it lets conclusions to be drawn from its shape.

The identifiers to use these components are:

- X1
- X2
- X3
- X4
- Y1
- Y2
- Y3
- Y4
- Z1
- Z2
- Z3
- Z4

A further group of target vector components uses shading properties of the resulting object. The implementation provides components for the diffuseColor property and the transparency property of objects at each point of the set or vertices of curves/meshes that visualize 1D/2D subsets. Many other shading properties would be possible in future implementations, especially when high level shading language scene graph nodes will be used. But currently only one color vector can be defined, three components exist for the red, green and blue color channel in the RGB color space, an additional set of three channels hue, saturation and value (HSV) provides a more convenient color space alternative to set this color. Using both RGB and HSV color vectors will result in a blend of both colors. Those component's identifier are:

- COLOR_H
- COLOR_S
- COLOR_V
- COLOR_R
- COLOR_G
- COLOR_B
- COLOR_ALPHA

Affine transformations are the next group of components in the visualization values vector. They affect the rotation (in 3 DOF → 3 components) and scale (as well 3 DOF → 3 components) of objects located at the coordinates explained above. This has to be seen in connection with a feature explained in section 4.2.6: custom objects, that will be used instead of the very small SoSphere objects, which represent points in space by default. An useful example would be little arrows for each vector of the graph set, where the argument of a domain/codomain set value could be mapped to a rotation of the arrow and the magnitude could be mapped to a scale transformation in the direction that affects the length of the arrow. If the coordinate components are used for curve/mesh objects (graphs of 1D/2D subsets), affine transformations of the vertices have no effect, because they have no rotation or scale, only a position. And if multiple of the up to four coordinate triplets are used, the affine transformation mappings only affect the first one (namely X1/Y1/Z1). That's why the component's identifiers are called POINT1OBJECT_... :

- POINT1OBJECT_ROTATION_X
- POINT1OBJECT_ROTATION_Y
- POINT1OBJECT_ROTATION_Z
- POINT1OBJECT_SCALE_X
- POINT1OBJECT_SCALE_Y
- POINT1OBJECT_SCALE_Z

Finally a group of target vector components enables mapping of data to time. Many solutions for this would have been possible, an oscillation between two affine transformations has been chosen as a simple, but useful example. Seven parameters have been implemented as target vector components, one of them controls the frequency of this oscillation, three components define a 3 DOF rotation vector and another triplet of components defines a scale vector in 3 DOF. The rotation and scale vectors make up a second affine transformation next to the one already defined for the static visualization. A custom object at each point of the set will oscillate its rotation and scale according to these values. As with the

static transformations, only the first coordinate triplet (X1/Y1/Z1) will be affected. The identifiers for the target vector components are:

- ANIM_FREQUENCY
- POINT1OBJECT_ROTATION_X_ANIM
- POINT1OBJECT_ROTATION_Y_ANIM
- POINT1OBJECT_ROTATION_Z_ANIM
- POINT1OBJECT_SCALE_X_ANIM
- POINT1OBJECT_SCALE_Y_ANIM
- POINT1OBJECT_SCALE_Z_ANIM

4.2.5.3 SoMappingGraphPropertyToVizFeature with scale and offset

To enable fields of the mapping matrix in the node kit SoComplexFunctionGraphVisualization, a list part named *mappingList* has been included into this node kit. This list part is restricted to children of type SoMappingGraphPropertyToVizFeature. One such node defines one field of the mapping matrix. This node has a SoSFEnum field named *source* (the enumeration values are the identifiers shown in subsection 4.2.5.1) and another SoSFEnum field named *target* (using the enumeration identifiers from subsection 4.2.5.2).

One limitation showed up in an early prototype: the range of the source values might not fit to the range of the target values. An obvious example is, that the range of (CO)DOMAIN_ARGUMENT is $[-\pi, \pi]$ whereas the range of COLOR_H is $[0,1]$. Because of its cyclicity the hue channel would have been an ideal match for visualizing angles, but the range adaption had to be solved. To achieve this two additional SoSFFloat fields have been put into SoMappingGraphPropertyToVizFeature: *scale* and *offset*. Their values allow a linear transformation of the values during the mapping step, the relation of a target value *t* to a source value *s* will be: $t = s * scale + offset$. This extents the interpretation of table 1: each cell contains not only an on/off value, if it's set to be “on”, it also contains a scale and an offset value.

In the Inventor file format a single mapping looks like this, the shown values are the default values:

```
SoMappingGraphPropertyToVizFeature {
  source              DOMAIN_REAL
  target              X1
  offset              0.0
  scale               1.0
}
```

All mappings, in form of a list look like this:

```
SoComplexFunctionGraphVisualization {
  functionEvaluator ...

  mappingList NodeKitListPart {
    containerNode Separator {
      SoMappingGraphPropertyToVizFeature {
        source ...
        target ...
      }
```

```
        SoMappingGraphPropertyToVizFeature {
          source ...
          target ...
        }
        SoMappingGraphPropertyToVizFeature {
          source ...
          target ...
        }
        ...
      } # containerNode
    } # mappingList

  }
```

4.2.6 GraphVisualization Configuration Input

Up this point all Inventor nodes and fields have been explained, that define the graph (in the SoComplexFunctionEvaluator named *functionEvaluator*) and the visual mapping (in a list of SoMappingGraphPropertyToVizFeature named *mappingList*). According to the visualization pipeline this would be everything needed to generate the output and in fact the user can produce sufficient function graph representations. But still a number of visualization parameter are missing in the *functionEvaluator* and the *mappingList*. These parameters have been included into the node kit SoComplexFunctionGraphVisualization, they are referred to as "configuration parts/fields" and they define:

- custom objects, that will be used as "points" at the coordinates
- line and arc connectors, that define how multiple coordinate vectors from the visualization values set (X1/Y1/Z1, ..., X4/Y4/Z4) will be represented by the scene graph
- array dissolvers, that define if 1D/2D subsets should really be drawn as curves/meshes, or as a set of points/lines instead

All these parameters are optional and not needed in many cases.

4.2.6.1 Custom Point Objects

By default a point at a coordinate from the visualization values set is rendered via a very small sphere:

```
Separator {
  Scale  {
    scaleFactor 0.01 0.01 0.01
  }
  Sphere { }
}
```

In case there are mappings to the affine transformation components, user defined objects will be required, especially arrows to show directions. For this purpose SoComplexFunctionGraphVisualization has 4 optional parts:

- point1Object
- point2Object
- point3Object
- point4Object

These parts are of type SoSeparator, so they can contain arbitrary scene graph models. The content of point1Object will be shown at the (X1/Y1/Z1) coordinate of each element of the visualization values set (but only if a mapping in the mappingList to X1/Y1/Z1 is defined; otherwise if there is no mapping defined, no objects will be rendered). In the same way point2Object will be rendered at each (X2/Y2/Z2), point3Object at (X3/Y3/Z3) and point4Object at (X4/Y4/Z4).

The custom point object part could also contain an empty SoSeperator:

```
point1Object Separator { }
```

This would result in nothing to be drawn at the coordinate even if a mapping to this coordinate is defined. This is mostly only interesting when line/arc connectors (see subsection 4.2.6.2) are used.

A typical content of the custom object part would be the following:

```
point1Object Separator {
  RotationXYZ {
    angle -1.5708
    axis  Z
  }
  Cone {
    bottomRadius 0.02
    height 0.1
  }
}
```

This is a small Cone in positive x direction, the most simple arrow for arrow plots.

4.2.6.2 Line and Arc Connectors

By default if mappings to all 4 possible coordinate targets exist, the graph of a single point set would be 4 points (small spheres or custom point objects) in space; the graph of a 1D subset would be 4 curves and the graph of a 2D subset would be 4 meshes.

But the multiple coordinates where designed to be used as well like parallel coordinates: a set of points connected by a line representing a “point” in a higher dimensional space. To achieve this 6 parts have been added to SoComplexFunctionGraphVisualization, that can switch on and off a line between each possible pair:

- point 1 and point 2
- point 1 and point 3
- point 1 and point 4
- point 2 and point 3
- point 2 and point 4
- point 3 and point 4

For each pair an own SoDrawStyle (line color, thickness and line pattern) can be defined. If 6 boolean fields that switch the lines on and off and 6 more parts for the draw styles would have been included into the node kit, an unnecessary redundancy would have emerged. To avoid this, only the draw style parts where created as optional parts, if they are defined, a line will be generated, otherwise not. That's why the parts were named:

- useLineP1ToP2WithDrawStyle
- useLineP1ToP3WithDrawStyle
- useLineP1ToP4WithDrawStyle
- useLineP2ToP3WithDrawStyle
- useLineP2ToP4WithDrawStyle
- useLineP3ToP4WithDrawStyle

An example for a line to be drawn between each visualization vector's point 1 and point 3 with a thick dashed line would look like:

```
SoComplexFunctionGraphVisualization {
  functionEvaluator ...
  mappingList ...

  useLineP1ToP3WithDrawStyle DrawStyle {
    lineWidth   2.5
    linePattern 0xff00
  }
}
```

Another coordinate connector has been implemented: An arc (a quarter ellipse) starting at point 1, ending at point 2 and having its ellipse center at point 3. Such an arc can visualize, that point 1 and point 3 have a similar relationship as point 2 and point 3; for example point 1 is at distance d above point 3 and point 2 is at distance d left of point 3 → an arc would show, that point 1 is "projected down" to point 2. As with the line connectors explained above, this is a part of SoComplexFunctionGraphVisualization of type SoDrawStyle and has the name:

- useArcP1ToP2AroundP3WithDrawStyle

4.2.6.3 Array Dissolvers

By default 1D subsets are drawn as curves and 2D subsets as meshes. But to the vertices of a curve or a mesh themselves no rotation and scaling could be applied. What the user wants in this case is custom point objects to be drawn at each point of the subset, as if each point was a single point set. That's why two boolean fields have be added to SoComplexFunctionGraphVisualization:

- curveAsPointSet
- surfaceAsPointSet

They are set to FALSE by default, but if set to TRUE 1D/2D subsets are rendered as a set of points.

A related functions has the boolean field

- surfaceAsLineSet

which makes a 2D subset with M*N vertices treated as M+N 1D subsets, the resulting line set provides in some cases a better view (occlusions removed) and sometimes a better impression of curvature.

4.2.7 VizObjectDefinitionState

All user input possibilities have been listed in the preceding sections. In order to make the necessary user input as simple and clear as possible, some default behaviour has been implemented into SoComplexFunctionGraphVisualization. To understand the graph visualization framework completely, this behaviour is described in the following. To organize the default behaviour a state vector has

been created for the visualization node kit. It is called VizObjectDefinitionState and has these members (all of them are boolean variables):

- mappingToPoint1Defined
- mappingToPoint2Defined
- mappingToPoint3Defined
- mappingToPoint4Defined
- mappingToAnyPointDefined
- mappingToRGBDefined
- mappingToHSVDefined
- mappingToTransparencyDefined
- mappingToAnyColorPropertyDefined
- mappingToRotationXDefined
- mappingToRotationYDefined
- mappingToRotationZDefined
- mappingToScaleDefined
- mappingToAnimationFrequencyDefined
- mappingToRotationXAnimationDefined
- mappingToRotationYAnimationDefined
- mappingToRotationZAnimationDefined
- mappingToScaleAnimationDefined
- point1ObjectDefined
- point2ObjectDefined
- point3ObjectDefined
- point4ObjectDefined
- useArcP1ToP2AroundP3
- useLineP1ToP2
- useLineP1ToP3
- useLineP1ToP4
- useLineP2ToP3
- useLineP2ToP4
- useLineP3ToP4
- useAnyLine
- curveAsPointSet
- surfaceAsLineSet
- surfaceAsPointSet

This state collects the 32 visualization values vector components into 18 groups (their names 'mappingTo...Defined' should be self-explanatory), they are automatically kept in sync with the mapping matrix in *mappingList*. Additionally the state mirrors the configuration parts/fields.

When a subset is added to the domain set, a subset is immediately and automatically added to the visualization geometry set. During this setup the following algorithm builds up the scene graph objects depending on the state:

```
if mappingToAnyColorPropertyDefined
  material = new SoMaterial

SoSeparator pointDefaultObject
  pointDefaultObject.add SoScale(0.01)
  pointDefaultObject.add SoSphere
if point1ObjectDefined
    point1Object = parentGraphViz.point1Object
  else
    point1Object = pointDefaultObject
if point2ObjectDefined
    point2Object = parentGraphViz.point2Object
  else
    point2Object = pointDefaultObject
if point3ObjectDefined
    point3Object = parentGraphViz.point3Object
  else
    point3Object = pointDefaultObject
if point4ObjectDefined
    point4Object = parentGraphViz.point4Object
  else
    point4Object = pointDefaultObject
if mappingToPoint1Defined
  point1Translation = new SoTranslation
  if mappingToRotationXDefined
```

```
    if mappingToRotationXAnimationDefined
      point1RotationXPendulum = new SoPendulum
    else
      point1RotationX = new SoRotationXYZ with axis=X
  if mappingToRotationYDefined
    if mappingToRotationYAnimationDefined
      point1RotationYPendulum = new SoPendulum
    else
      point1RotationY = new SoRotationXYZ with axis=Y
  if mappingToRotationZDefined
    if mappingToRotationZAnimationDefined
      point1RotationZPendulum = new SoPendulum
    else
      point1RotationZ = new SoRotationXYZ with axis=Z
  if mappingToScaleDefined
    if mappingToScaleAnimationDefined
      point1ScaleWobble = new SoWobble // scale counterpart to SoPendulum
    else
      point1Scale = SoScale
  SoSeparator point1Sep
  add point1Sep
  if mappingToAnyColorPropertyDefined
    point1Sep.add material
  point1Sep.add point1Translation
  if mappingToRotationXDefined
    if mappingToRotationXAnimationDefined
      point1Sep.add point1RotationXPendulum
    else
      point1Sep.add point1RotationX
  if mappingToRotationYDefined
    if mappingToRotationYAnimationDefined
      point1Sep.add point1RotationYPendulum
    else
      point1Sep.add point1RotationY
  if mappingToRotationZDefined
    if mappingToRotationZAnimationDefined
      point1Sep.add point1RotationZPendulum
    else
      point1Sep.add point1RotationZ
  if mappingToScaleDefined
    if mappingToScaleAnimationDefined
      point1Sep.add point1ScaleWobble
    else
      point1Sep.add point1Scale
  point1Sep.add point1Object
if mappingToPoint2Defined
  ... see mappingToPoint1Defined, but no rotation, scale or animation
if mappingToPoint3Defined
  ... see mappingToPoint1Defined, but no rotation, scale or animation
if mappingToPoint4Defined
  ... see mappingToPoint1Defined, but no rotation, scale or animation

if useArcP1ToP2AroundP3
  add SoSeparator arcSep
  arcSep.add new SoLightModel with model=BASE_COLOR
  if mappingToAnyColorPropertyDefined
    arcSep.add material
  arcSep.add new SoCoordinate3 arcCoordinates
  arcSep.add new SoLineSet arc

if useLineP1ToP2
  add SoSeparator lineP1ToP2Sep
  if mappingToAnyColorPropertyDefined
    lineP1ToP2Sep.add material
  lineP1ToP2Sep.add new SoCoordinate3 lineP1ToP2Coordinates
  lineP1ToP2Sep.add new SoLineSet lineP1ToP2
if useLineP1ToP3
  ... see useLineP1ToP2
if useLineP1ToP4
  ... see useLineP1ToP2
if useLineP2ToP3
  ... see useLineP1ToP2
if useLineP2ToP4
  ... see useLineP1ToP2
if useLineP3ToP4
  ... see useLineP1ToP2
```

As an example of what could be read from the pseudo code above: if according to the VizObjectDefinitionState a mapping to any color property (R, G, B, H, S or V)

is defined, a SoMaterial node is added to the scene graph. In this case any material defined before in the scene graph is overwritten by this material. If no mapping to any color property is defined, the material for the graph objects is taken from the scene graph outside SoComplexFunctionGraphVisualization. This shows the necessity of the visualization objects definition state, because even without any mappings to the colors the visualization values vector contains color values: then these values are zero, i.e. black. Without the state all vertices would be black, but with it we know, that the color is not black, but not mapped, so no SoMaterial is created. For the other default behaviour see the algorithm above or the extensive source code documentation.

For the algorithm to set and update the values for the scene graph nodes generated above see the following pseudo code:

```
if mappingToAnyColorPropertyDefined
  if mappingToHSVDefined && mappingToRGBDefined
    material.diffuseColor =
      vizValuesSubSet->data.[RGB]      / 2 +
      rgb(vizValuesSubSet->data.[HSV]) / 2
  else if mappingToHSVDefined
    material.diffuseColor = rgb(vizValuesSubSet->data.[HSV])
  else if mappingToRGBDefined
    material.diffuseColor = vizValuesSubSet->data.[RGB]

if mappingToPoint1Defined
  if mappingToRotationXDefined
    if mappingToRotationXAnimationDefined
      point1RotationXPendulum.rotation0 = SbRotation(
        SbVec3f(1, 0, 0),
        vizValuesSubSet->data.[POINT1OBJECT_ROTATION_X]
      )
      point1RotationXPendulum.rotation1 = SbRotation(
        SbVec3f(1, 0, 0),
        vizValuesSubSet->data.[POINT1OBJECT_ROTATION_X_ANIM]
      )
      point1RotationXPendulum.speed =
        vizValuesSubSet->data.[ANIM_FREQUENCY]
    else
      point1RotationX.angle =
        vizValuesSubSet->data.[POINT1OBJECT_ROTATION_X]
  if mappingToRotationYDefined
    if mappingToRotationYAnimationDefined
      point1RotationYPendulum.rotation0 = SbRotation(
        SbVec3f(1, 0, 0),
        vizValuesSubSet->data.[POINT1OBJECT_ROTATION_Y]
      )
      point1RotationYPendulum.rotation1 = SbRotation(
        SbVec3f(1, 0, 0),
        vizValuesSubSet->data.[POINT1OBJECT_ROTATION_Y_ANIM]
      )
      point1RotationYPendulum.speed =
        vizValuesSubSet->data.[ANIM_FREQUENCY]
    else
      point1RotationY.angle =
vizValuesSubSet->data.[POINT1OBJECT_ROTATION_Y]
  if mappingToRotationZDefined
    if mappingToRotationZAnimationDefined
      point1RotationZPendulum.rotation0 = SbRotation(
        SbVec3f(1, 0, 0),
        vizValuesSubSet->data.[POINT1OBJECT_ROTATION_Z]
      )
      point1RotationZPendulum.rotation1 = SbRotation(
        SbVec3f(1, 0, 0),
        vizValuesSubSet->data.[POINT1OBJECT_ROTATION_Z_ANIM]
      )
      point1RotationZPendulum.speed =
        vizValuesSubSet->data.[ANIM_FREQUENCY]
    else
      point1RotationZ.angle =
        vizValuesSubSet->data.[POINT1OBJECT_ROTATION_Z]
  if mappingToScaleDefined
    if mappingToScaleAnimationDefined
      point1ScaleWobble.scaleFactor0 =
```

```
        vizValuesSubSet->data.[POINT1OBJECT_SCALE]
      point1ScaleWobble.scaleFactor1 =
        vizValuesSubSet->data.[POINT1OBJECT_SCALE_ANIM]
      point1ScaleWobble.speed = vizValuesSubSet->data.[ANIM_FREQUENCY]
    else
      point1Scale = vizValuesSubSet->data.[POINT1OBJECT_SCALE]
  point1Translation = vizValuesSubSet->data.[X1/Y1/Z1]
if mappingToPoint2Defined
  point2Translation = vizValuesSubSet->data.[X2/Y2/Z2]
if mappingToPoint3Defined
  point3Translation = vizValuesSubSet->data.[X3/Y3/Z3]
if mappingToPoint4Defined
  point4Translation = vizValuesSubSet->data.[X4/Y4/Z4]

if useArcP1ToP2AroundP3
  make arcCoordinatesValues
    from vizValuesSubSet->data.[X1/Y1/Z1 / X2/Y2/Z2 / X3/Y3/Z3]
  arcCoordinates.point.setValues(arcCoordinatesValues)

if useLineP1ToP2
  make lineCoordinatesValues
    from vizValuesSubSet->data.[X1/Y1/Z1 / X2/Y2/Z2]
  lineP1ToP2Coordinates.point.setValues(lineCoordinatesValues)
if useLineP1ToP3
  ... see useLineP1ToP2
if useLineP1ToP4
  ... see useLineP1ToP2
if useLineP2ToP3
  ... see useLineP1ToP2
if useLineP2ToP4
  ... see useLineP1ToP2
if useLineP3ToP4
  ... see useLineP1ToP2
```

5 User Interfaces and Example Setups

As the underlying technology used in this implementation, namely the Studierstube API, supports various input and output devices for many different application environment scenarios, two setups relevant for the complex function graph visualization shall be presented in this chapter.

Following the description of each setup and its discussion, some instructions for the reader who wants to try them him/herself are appended. These instructions relate to the scene graph file, that has to be prepared before starting the application for a specific visualization task. In this file a scene including the SoComplexFunctionGraphVisualization node is described, all user interaction has to be defined therein as well.

5.1 Immersive Setup

The first setup resembles a setup used in the related project Construct3D (see section 2.2.1), where it has been called the "Basic AR Lab Multi-User Setup" (described in section 5.1 of [22]). Here two collaborating users wear semi-transparent HMDs for a shared virtual space and hold fully tracked interaction props in their hands. While Construct3D implements a full-featured menu system on a virtual tablet that is haptically represented and tracked in space by a glass plate in the real world and overlayed by a virtual menu (this is called the PIP, the *personal interaction panel* [32]), the complex function graph visualization framework only needs a tracked pen for interaction. A PIP could be easily integrated, but is no requirement in our case.

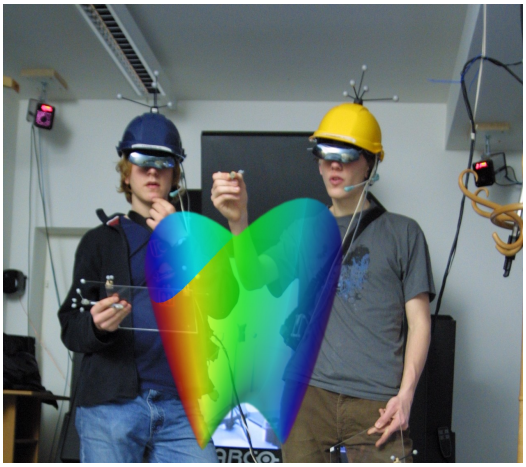


Figure 28: image composition demonstrating the combination of real and virtual space

For rendering of stereoscopic views for both users one dedicated host with two graphic ports is used. Tracking is done by an ARTtrack1 optical tracking system with four cameras that we interface via the OpenTracker API. The Construct3D user studies showed, that while a second host could easily render a 3rd and a 4th view of the scene, the number of users is mainly restricted by the number of cameras of the tracking system. More users cause more occlusions and it has been found, that four camera are sufficient to track two users in very high quality and tracking more than three users would require more of the expensive cameras.

This setup provides a very good basis to perceive spatial objects, not only through the stereoscopic display but much more through the possibility to walk around the objects co-located with the real world.

Interaction with the application is done by dragging around objects that are connected to parameters of the visualization, mostly moving and resizing the subsets of domain values and function parameters on the Complex plane.

5.1.1 Immersive Setup User Instructions

The scene graph file for such a setup would include at least one SoComplexFunctionGraphVisualization node which defines the function, the domain subset and the visual mapping. If this was the only node in the scene, a static visualization could be studied by the observers, the only interaction would be their change of viewpoints while communicating their insights to each other.

A next step to show more information could be to change the scene graph file to let interesting parameters of the visualization being animated. Technically this means to replace a constant field value by an active data source, namely an Inventor type of node referred to as *engine*, for example SoElapsedTime, which continuously changes the field value. In this way a visualization could be observed for a certain range of a parameter instead of just one fix value.

But still the use of engines would not increase the level of interaction. This is easily to be achieved by using the SoDragKit object of the Studierstube API. That means we don't drag a visualization object directly, but a representation of a coordinate (e.g. a small sphere) and make field connections to visualization parameters, for example the *real* and *imaginary* fields of a SoComplexDomainSubSet_0D. This would result in the domain value being located at the coordinate of this small sphere. In this way the user can control the value of a complex number z and observe the graph visualization which is a visual combination of z and $f(z)$. How this visualization looks, depends highly on the visual mapping defined in the visualization kit.

5.2 *Desktop Setup*

This setup has been created to have the least hardware requirements possible. It can be run right from a CDROM on a common PC setup. Software requirements are a 32bit version of Microsoft Windows and a OpenGL compliant graphics driver, input device is a mouse only and the default display is used for a monoscopic view on the scene.

Figure 29: a user working with the desktop setup

The big advantage of this setup is the wide audience that can be reached. As an example all students of a course could do their homework using this software. Another advantage is portability. A teacher could easily bring it on a laptop computer to a lecture and illustrate some of his instructions interactively while showing the screen content on a projection wall.

The selection of the viewpoint without the tracking system works as follows: here we use the functionality of Coin3D and its Win32 GUI binding to control the camera with the mouse. Left click and dragging results in a rotation around the look-at point. Middle click and dragging enables panning the look-at point left/right and up/down. Left+middle click and dragging moves the eye point closer/farer to/from the look-at point.

Because the mouse serves both as camera control and as interaction device, Coin3D implements two modes that can be toggled with the escape key on the keyboard: viewing mode (rotate, pan, dolly the camera) or interaction mode (click and move draggable objects).

5.2.1 Desktop Setup User Instructions

Viewing static function visualizations and animations using field connections to engines without user control, requires in this setup the same preparations as in the immersive setup: write exactly the same scene description into a scene graph file and start the application.

For interaction an equivalent of SoDragKit (see subsection 5.1.1) has to be used in the scene. SoDragKit itself would require keyboard and mouse control via Studierstube's tracking layer OpenTracker. Such OpenTracker modules exist, however, Coin3D features more advanced mouse interaction nodes called *draggers*, which already have a ray picking functionality built-in to enable

intuitive mouse dragging of handles that can be field-connected to scene objects. Coin3D includes even more sophisticated interaction nodes on top of draggers, called manipulators. They can simply replace a transform node in the scene graph and show a bounding box of the affected branch that can be moved around with the mouse. As in the immersive setup, we can again use a representation of a coordinate (like a small sphere) and connect the translation fields of its dragger (or manipulator) to visualization parameters.

6 Usage Examples

To show the strength of the developed application, in this chapter two actual topics from the world of applied complex mathematics are visualized with the described framework.

6.1 Exploring the Process of Fractals Calculation (Iterative Complex Functions)

The Mandelbrot set is the set of complex numbers c, for which the iterative calculation of $f(z)=z^2+c$, starting at $z=0$, does not tend to infinity.

This set has become famous because it shows a very complicated structure arising from a simple definition on the one hand and its aesthetic appeal on the other hand. In this section the calculation process itself for the Mandelbrot set shall be examined with the developed visualization application.

Trying to investigate too many features at once usually leads to an overloaded visualization. So for the function that is examined, different reduced visualizations are proposed that each show their own aspects of the function.

6.1.1 Experiment Nr. 1

The first experiment should simply show a single point $f(z)$ on the complex plane while dragging around z and c. For this purpose we write a scene graph file that contains the following elements (compare fig. 30 for results):

- a light source (has always to be included in the scene)
- a label: a square with the opposite corners (-5, -5, 0) and (5, 5, 0) with a texture that shows a real axis with the range [-5, 5] on the x-axis and an imaginary axis with the range [-5, 5] on the y-axis
  - this represents the Complex plane that we work on
- two draggers for interaction: SoDragKit or SoTabBoxManip objects (depending on the immersive or desktop setup) containing a representation of a point on the plane (a SoSphere with radius 0.3)
  - point z colored yellow, initially placed at (1, 0, 0)
  - point c colored red, initially placed at (0, 0, 0)
  - give them names, that we can use in field connections:
    - DEF DRAG_KIT_1 ... (dragger for z)
    - DEF DRAG_KIT_2 ... (dragger for c)

- a SoComplexFunctionGraphVisualization object for $f(z)$
  - this object has a part named *functionEvaluator* of type SoComplex-FunctionEvaluator, where inside we define the graph by three input elements: the domain set, the function and the parameter
    - the domain set is a part of the *functionEvaluator* with name *domain-SubSetList* and type NodeKitListPart
      - the content of *domainSubSetList* is contained in its part with the name *containerNode* and type SoSeparator
      - inside the *containerNode* we put for our experiment one node of type SoComplexDomainSubSet_0D, which defines the domain set as one single point
        - the two fields of this domain subset *real* and *imaginary* get field connections from the x and y component of DRAG_KIT_1.translation
        - we extract the components of the dragger's vector using an engine, for example SoCalculator, that can output a scalar *oa* from an input vector *A* using the expression "*oa = A[n]*", where n in 0/1/2 for x/y/z
      - as we define only one single domain point, the graph will consist of only two complex numbers: z and $f(z)$
    - *function* is a string-field of functionEvaluator; $f(z)=z^2+c$ is defined in the grammar explained in section 4.2.3 as the string "add(pow(z,(2)),p1)"
    - the parameter p1 is defined by functionEvaluator's fields *parameter1-Real* and *parameter1Imag*
      - they get field-connected to DRAG_KIT_2.translation's x and y components using SoCalculators
  - the visualization object now needs to know how to do the visual mapping of the graph, so we define its part *mappingList* of type NodeKitListPart to contain two mappings (each a SoMappingGraphPropertyToVizFeature node):
    - a mapping from CODOMAIN_REAL to X1 and
    - a mapping from CODOMAIN_IMAGINARY to Y1
  - finally we define the output object to have a blue color using a SoMaterial node in front of the SoComplexFunctionGraphVisualization

In following the SoComplexFunctionGraphVisualization part of the scene described above in words is given in the syntax of the scene graph file:

```
SoComplexFunctionGraphVisualization {
  functionEvaluator SoComplexFunctionEvaluator {
    domainSubSetList NodeKitListPart {
      containerNode Separator {
        SoComplexDomainSubSet_0D {
          real                1.0 =
            Calculator {
              A = USE DRAG_KIT_1.translation
```

```
            expression "oa = A[0]"
          }.oa
        imaginary           0.0 =
          Calculator {
            A = USE DRAG_KIT_1.translation
            expression "oa = A[1]"
          }.oa
      } # SoComplexDomainSubSet_0D
    } # containerNode
  } # domainSubSetList
  function            "add(pow(z,(2)),p1)"
  parameter1Real      0.0 =
          Calculator {
            A = USE DRAG_KIT_2.translation
            expression "oa = A[0]"
          }.oa
  parameter1Imag      0.0 =
          Calculator {
            A = USE DRAG_KIT_2.translation
            expression "oa = A[1]"
          }.oa
} # SoComplexFunctionEvaluator
mappingList NodeKitListPart {
  containerNode Separator {
    SoMappingGraphPropertyToVizFeature {
      source              CODOMAIN_REAL
      target              X1
    } # SoMappingGraphPropertyToVizFeature
    SoMappingGraphPropertyToVizFeature {
      source              CODOMAIN_IMAGINARY
      target              Y1
    } # SoMappingGraphPropertyToVizFeature
  } # containerNode
} # mappingList
} # SoComplexFunctionGraphVisualization
```

Fig. 30 shows results of this experiment as screenshots from a desktop setup. The user can drag around the yellow point (z) and observe where the blue point ($f(z)$) is going. One insight that can be won when doing this, is that when the user moves z once around the origin, $f(z)$ moves twice around the origin. Then the user can move around the red point (c) and observe, that $f(z)$ is translated by the same translation. Moving z once around the origin now results in $f(z)$ moving twice around c .

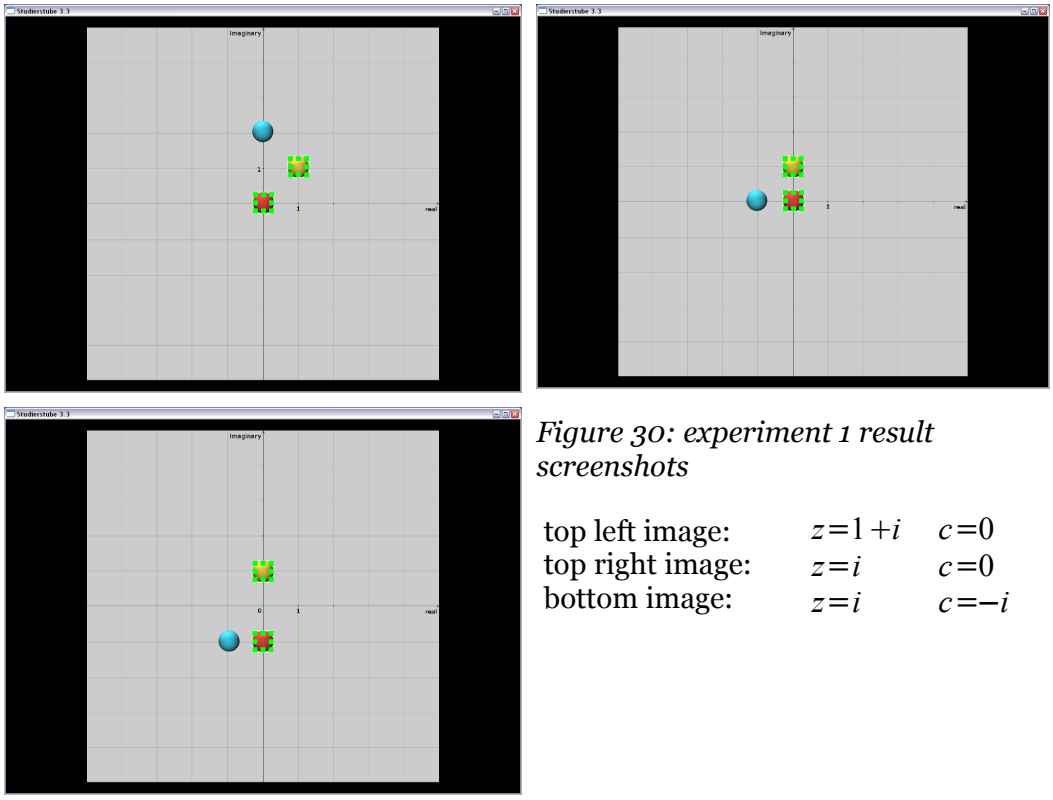


Figure 30: experiment 1 result screenshots

top left image:	$z=1+i$	$c=0$
top right image:	$z=i$	$c=0$
bottom image:	$z=i$	$c=-i$

In an educational context an AR setup could be used very effectively in the following way: In a setup using fully tracked semi-transparent HMDs, the complex plane of the scene above could be co-located with a table top surface in the real world. The label could be removed from the virtual scene and instead a large piece of drawing paper with the labeled axes could be used at the same location. Still the red and the yellow point could be dragged around freely, but this time with a tracked pen. A restriction of the location of the points to the plane (using engines in the scene graph) would be helpful. Now the user could use a real yellow drawing pen to draw a shape (e.g. a circle) into the Complex plane, yellow because this is the color for the domain. Then he/she could use the tracked pen to move the virtual yellow point over the real yellow shape. With the other hand or by the help of another user, using a real blue pen (blue as the color for the codomain) the user could draw along the trajectory of the blue point. In this way the behaviour of the function could be examined thoroughly in how it transforms arbitrary shapes. This applies to a constant value of c, i.e. the red point would have to stay in position.

To explore the function for different values of the parameter c, other sheets of paper could be used. By holding two sheets behind each other into the sunlight, the functions for different parameter values could be compared.

Having examined the behavior of the non-iterative evaluation of the function, now iterations could be followed. That means, the user places the yellow point (z) into the origin and marks the resulting position of the blue point ($f(z)$) with a pencil, possibly writing the iteration number (1) next to it. Then he/she would move the yellow point to mark nr. 1, look for the position of the

blue point and mark iteration number 2 and so on. Now the following fundamental insight to understanding the Mandelbrot set can be gained: sometimes the iterations tend towards infinity and sometimes they keep moving around the proximity of the origin.

To speed up the experiment for two users, two different draggable yellow points for two different domain values (with their according codomain values being generated in blue) could be used in the virtual scene. Then each user could hold a yellow point continuously on his tracked pen tip and when the first user starts with the first iteration with his yellow point at the origin, the second user could already place her/her yellow point exactly at the resulting blue point (second iteration). Then the first user could immediately continue with the third iteration by placing his/her yellow point at the blue point from the second iteration and so on. In very short time the tendency of the iteration could be found. If the iteration does not tend towards infinity, the location of the read point (c) can be marked, that it belong to the Mandelbrot set and the same procedure could be repeated for another value of c .

6.1.2 Experiment Nr. 2

In the first experiment we used a very simple function visualization: observing a single point in the domain set and its counterpart in the codomain set, both in the same plane. In experiment nr. 2 we want to see more information at once by visualizing not one point but a grid of N*M points from the domain set. Additionally we will use the 3rd coordinate component to generate spatial objects. This will require from the user a higher degree of abstraction to "read" the information from the representation, but once he/she learned to interpret the spatial objects generated from the function, more interdependencies can be seen and understood.

In the experiment we want to observe the real part and the imaginary part of $f(z)$ in dependence to z . Therefore we plot two surfaces (one for the real part, the other for the imaginary) over the Complex plane. So we define the scene as follows:

- light source and label identical to experiment nr. 1
- one dragger object holding one point on the Complex plane: this one represents the parameter c of our function
- two objects of type SoComplexFunctionGraphVisualization for the surfaces:
  - the first one defines the *functionEvaluator* to include:
    - the *domainSubSetList* with one polar grid (SoComplexDomainSubSet_2D_RegularPolar), a disc shaped set of N*M points around the origin with the radius of 2
    - the same *function* as in experiment nr. 1: "add(pow(z,(2)),p1)"
    - the same *parameter1Real/...Imag*, field-connected to DRAG_KIT_1. translation
    - the *functionEvaluator* gets a name via "DEF GRAPH_1_POLAR_SURFACE"

- the second SoComplexFunctionGraphVisualization uses a shared instance of the same *functionEvaluator* via “USE GRAPH_1_POLAR_SURFACE “
- the first SoComplexFunctionGraphVisualization instance gets a *mapping-List* that defines a mapping to the components of a coordinate:
  - DOMAIN_REAL to X1
  - DOMAIN_IMAGINARY to Z1 with scale = -1
  - CODOMAIN_REAL to Y1
  - the negation of Z1 converts the right-handed coordinate system of Studierstube to a left-handed coordinate system common in the field of mathematics and geometry
- the second SoComplexFunctionGraphVisualization's *mappingList* defines another coordinate:
  - DOMAIN_REAL mapped to X1
  - DOMAIN_IMAGINARY to Z1 with scale = -1
  - CODOMAIN_IMAGINARY to Y1
- finally we put a green Material node in front of the real surface and a blue Material in front the imaginary surface

This scene looks as shown in fig. 31. As the user moves around the red point, the graph changes. It can be observed, that the green (real) and blue (imaginary) parts only move up and down and show no other transformations in dependency of the red point (*c*).

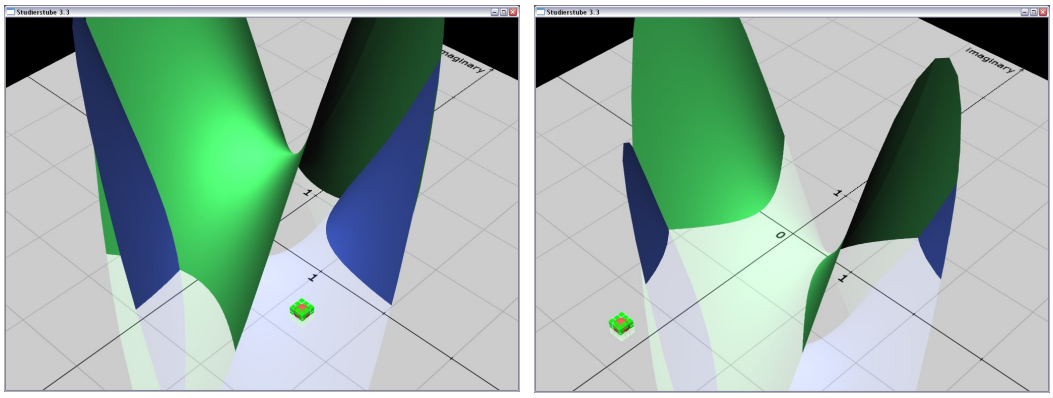


Figure 31: Experiment Nr. 2 screenshots

Now some improvements to the scene are proposed. First, the surfaces occlude each other in many parts, important information might be hidden from some viewpoints. So we could present them as well as grids of lines (see fig. 32). This is done by simply setting the field *surfaceAsLineSet* to *TRUE* in both visualization objects.

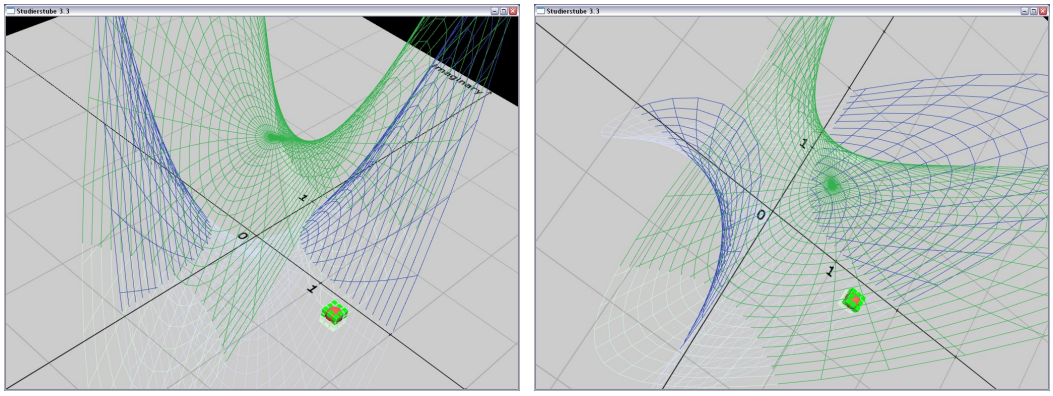


Figure 32: Experiment Nr. 2 screenshots

But as the figure above shows, it is too difficult for the resulting grids to tell which surface is in front of the other. Some trade-off between grids and shaded surfaces would be helpful, so the following solution is proposed: define the two SoComplexFunctionGraphVisualization objects fully shaded (as in fig. 31), but now with a Material node where *diffuseColor* is set to black and *transparency* is set to 0.8. Then define a shared instance of each SoComplexFunctionGraphVisualization, preceded by a Material node defining the color as green or blue respectively with no transparency and a DrawStyle node with *style* set to "LINES" and *lineWidth* set to 2.0. This results in the images shown in fig. 33, which are clearly better readable than in fig. 32.

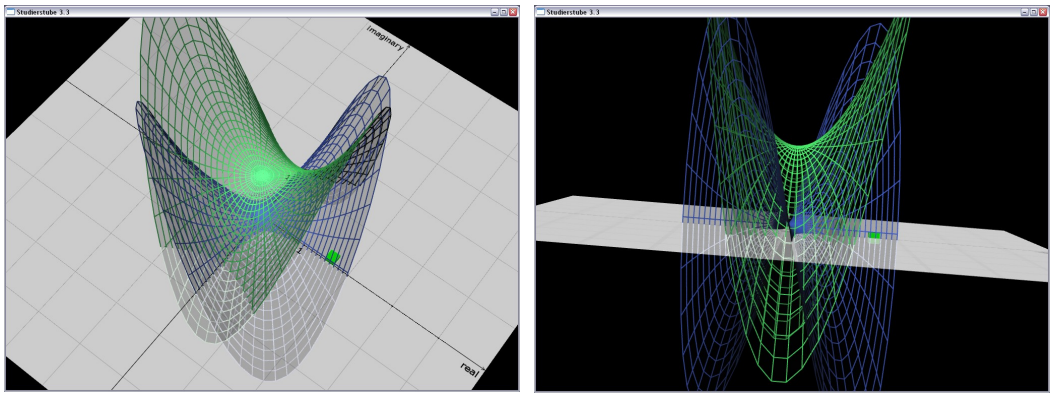


Figure 33: Experiment Nr. 2 screenshots

To understand the visualization of separate surfaces for the real and imaginary part, the user is required to imagine a line normal to the Complex plane, intersecting the plane at a value z and read the intersections of this line with the surfaces as the real and imaginary part of a complex number $f(z)$. To help users to imagine this concept, a visualization of these intersections is proposed as a further extension of experiment nr. 2. Here we extent our scene by a dragger for a single domain value z (identical to the yellow point in experiment nr. 1) and two further SoComplexFunctionGraphVisualization nodes. The first one shows the real part of $f(z)$ as a green point above or below the yellow point, at that position where it lies directly on the green surface. The green point and the yellow point will also get connected by a dotted line. The second new visualization node does the same for the imaginary part of $f(z)$ with a blue point (see fig. 34).

In the syntax of the scene graph file this means: We add a new draggable yellow sphere and give it the name DRAG_KIT_2. For the green point a SoComplexFunctionGraphVisualization node is added that defines a new *functionEvaluator* because the *functionEvaluator* that is already defined for the surfaces contains the right function with the right parameters, but another domain set. The new *functionEvaluator* contains a *domainSubSetList* with a single domain subset of type SoComplexDomainSubSet_0D which is field-connected to DRAG_KIT_2.translation. The *function* and *parameter1Real/...Imag* fields should not be redefined to avoid inconsistencies, but get field-connections from the corresponding fields of the *functionEvaluator* of the surface graph. Then we define the *mappingList* of the visualization node. There we setup five mappings:

- three mappings define coordinate nr. 1 (the location for the green point):
  - DOMAIN_REAL is mapped to X1
  - DOMAIN_IMAGINARY to -Z1
  - CODOMAIN_REAL to Y1
- two mappings define the 2nd coordinate at the domain value on the ground plane which will serve as the end point of the connection line between the yellow and the green point
  - DOMAIN_REAL is mapped to X2
  - DOMAIN_IMAGINARY to -Z2
  - nothing is mapped to Y2, i.e. it will always be zero

Now the visualization node of the green point is almost finished, we only need to define the field *point2Object* as an empty SoSeparator, because otherwise a second green point would be rendered at the same location as the yellow point. We don't want that, we just need the second coordinate as the end point of the connection line. Then, by defining the optional part *useLineP1ToP2WithDrawStyle* as a DrawStyle with *linePattern* "0x00ff" for a dotted line between coordinate 1 and coordinate 2 the real part of the visualization is complete.

Fortunately the imaginary part, the second visualization node that generates the blue point and its connection line, is defined much shorter now. It uses the same *functionEvaluator* as the green point by a shared instance. Then a *mappingList* has to be declared that is almost the same as explained above, only that CODOMAIN_IMAGINARY is mapped to Y1 instead of CODOMAIN_REAL. The empty SoSeparator for point2Object and the useLineP1ToP2WithDrawStyle are the same.

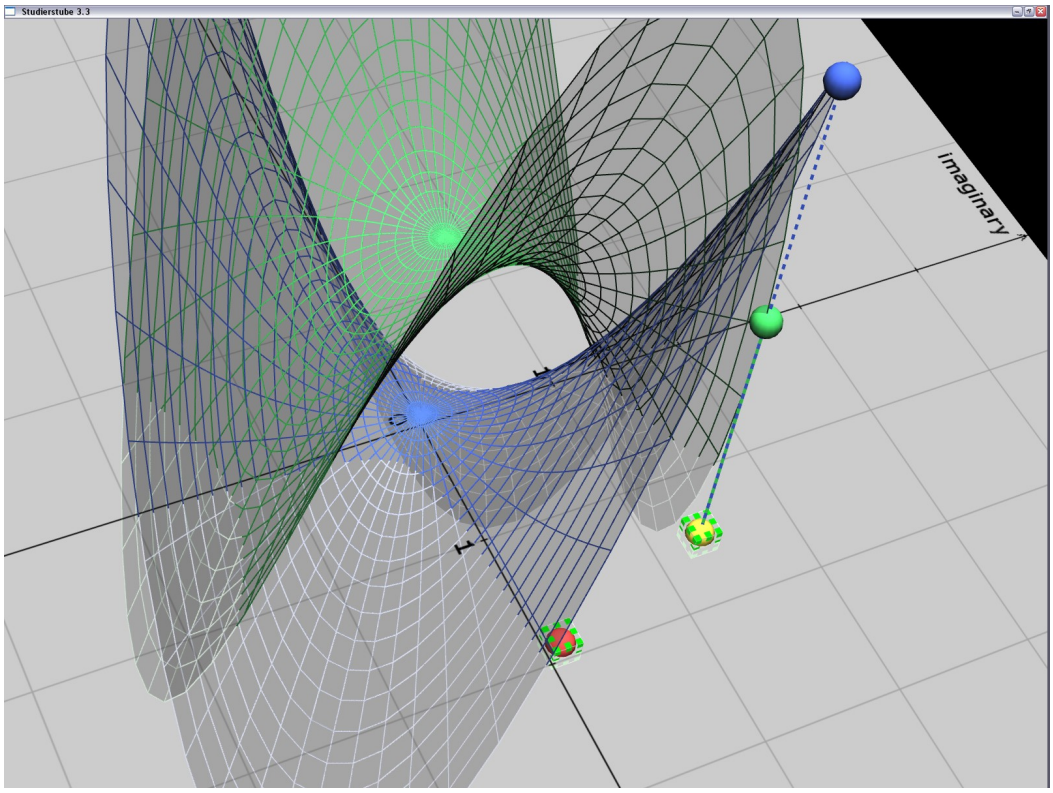


Figure 34: Experiment Nr. 2 with intersection points screenshot

The last extension proposal for experiment nr. 2 is to show the green and the blue point (real and imaginary part of $f(z)$) not only above z (i.e. on the y axis), but as well on the ground plane using the same axes as z (the x axis for the real part and the - z axis for the imaginary part). In order to show the identity of these two representations of $f(z)$ connection lines and arcs are used as shown in the following.

In our scene graph this extension again requires adding two new SoComplexFunctionGraphVisualization nodes. The first one will show copies of the green point, i.e. the real part of $f(z)$ in two new positions: above the origin and in the direction of the x axis from the origin (fig. 35). The other one will show two copies of the blue point, above the origin and in the direction of the negative z axis.

Fig. 35 shows the resulting scene with the surfaces and the blue points switched off, so only c (red), z (yellow) and the different representations of $f(z)$ (green) are left. The point annotated as point 1 is the same that we already had in fig. 34, it is $f(z)$ above z. Point 2 is defined in the new visualization node by mapping CODOMAIN_REAL to Y1 (X1 and Z1 stay at zero). It symbolizes, that the first green point is moved horizontally to the origin. Then we generate the point annotated in the figure below as point 3 by mapping CODOMAIN_REAL to X2 (Y2 and Z2 stay at zero). An arc around the origin shall symbolize, that point 2 is "projected down" to point 3, or better said, moved to another axis. For this arc as well as for the lines connecting point 2 to the origin and point 3 to the origin we need the coordinate of the origin as one of the 4 coordinate vectors in the visualization values set (see the explanation in section 4.2.1.1). But no mapping to

the origin in our *mappingList* is necessary, we just map nothing to X3, Y3 and Z3, then their values stay at zero. Now we use the optional configuration parts of the visualization node kit to generate the arc (*useArcP1ToP2AroundP3WithDrawStyle*) and the lines (*useLineP1ToP3WithDrawStyle* and *useLineP2ToP3WithDrawStyle*). For the connecting line between the points called 1 and 2 in the figure below we need the coordinate of the first point again in the new *mappingList*, so we map DOMAIN_REAL to X4, DOMAIN_IMAGINARY to - Z4 and CODOMAIN_REAL to Y4 and enable the line by the part *useLineP1ToP4WithDrawStyle*.

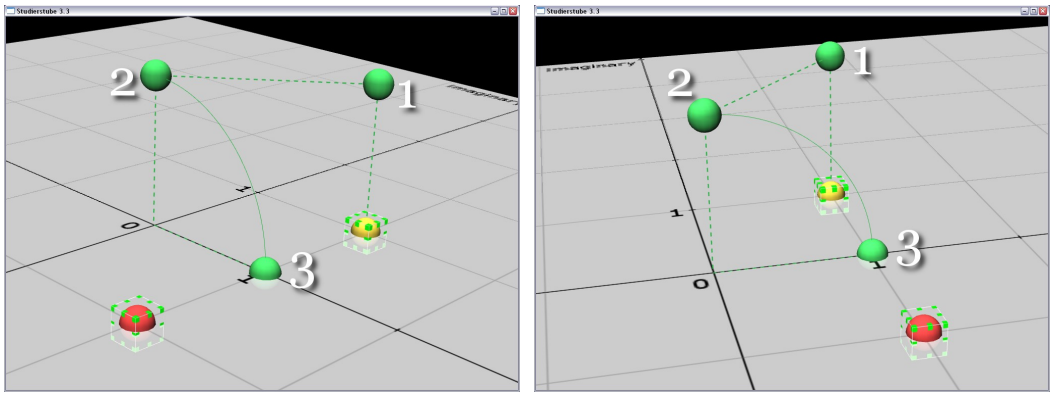


Figure 35: Experiment Nr. 2 with different representations of f(z), annotated screenshot

The visualization node for the blue point is defined analogously.

If the user now interacts with the visualization of the last extension of experiment nr. 2, already a very high load of graphical objects is present. In an AR setup, where he/she can move around the object and after taking some time to familiarize with the scene and learning the interrelations of the parameters, the last extension can convey deep insight into the process of the iterative calculation of $f(z)=z^2+c$.

6.2 Z-Tranform and Transfer Functions of Digital Filters

Section 1.2.3 already mentioned, that a visualization of a specific complex function, namely the transfer function of a digital filter would be helpful to understand how to adjust the filter parameters for a desired frequency domain behaviour of a filter. In this section an example for such a visualization with the developed framework is presented.

First we have to look at the mathematical background: we parameterize the digital signal filter (for our experiment this will be a finite impulse response filter) by defining its filter kernel (also called impulse response), a sequence of numbers that can be convolved with the signal to apply the effect [2]. A simple example for such a kernel might be $h(n)$ with $h(0)=0.5$, $h(1)=0.33$, $h(2)=0.17$ and $h(n)=0$ where $n\notin\{0,1,2\}$. The transfer function (the so called

Z-transform of $h(n)$) which is defined as $H(z)=\sum_{n=-\infty}^{\infty} h(n)\cdot z^{-n}$ would evaluate in our example as $H(z)=h(0)*z^{0}+h(1)*z^{-1}+h(2)*z^{-2} = 0.5+0.33*z^{-1}+0.17*z^{-2}$. This is the first function we want to observe in our experiment.

Again a scene graph has to be built. As in section 6.1 we include a light source and a textured square with labeled axes. First we only have a look at the magnitudes of our function $H(z)$. For this purpose a surface plot suffices. This is very simple to define in one SoComplexFunctionGraphVisualization node (see fig. 36):

- its *functionEvaluator* contains a polar grid of values within the proximity of 5 units around the origin
- its *function* string would be
  `"add(add(p1,mul(p2,pow(z,(-1)))),mul(p3,pow(z,(-2))))"`
- the three parameters have to be given only with their real part: *parameter1-Real* set to 0.5, *parameter2Real* to 0.33 and *parameter3Real* to 0.17

The function string above represents $p_1+p_2*z^{-1}+p_3*z^{-2}$. In order to understand its syntax with its many parentheses, a stepwise approach of writing the string "outside-to-inside" is recommended:

(1) the term consists of two additions: `"add(add(_,_),_)"`
(2) more precisely, additions of p1 and 2 products:
  `"add(add(p1,mul(_)),mul(_))"`
(3) these products multiply each a parameter with a power:
  `"add(add(p1,mul(p2,pow(_))),mul(p3,pow(_)))"`
(4) these powers take z to the power of -1 and -2 respectively, which makes the function string complete:
  `"add(add(p1,mul(p2,pow(z,(-1)))),mul(p3,pow(z,(-2))))"`

The visualization node is finished by setting the *mappingList*: DOMAIN_REAL is mapped to X1, DOMAIN_IMAGINARY to – Z1 and CODOMAIN_MAGNITUDE to Y1. The label would now be occluded by large parts of the surface. To read it anyway we put a Material node with the field *transparency* set to 0.1 in front of the visualization node, which enables a slight transparency.

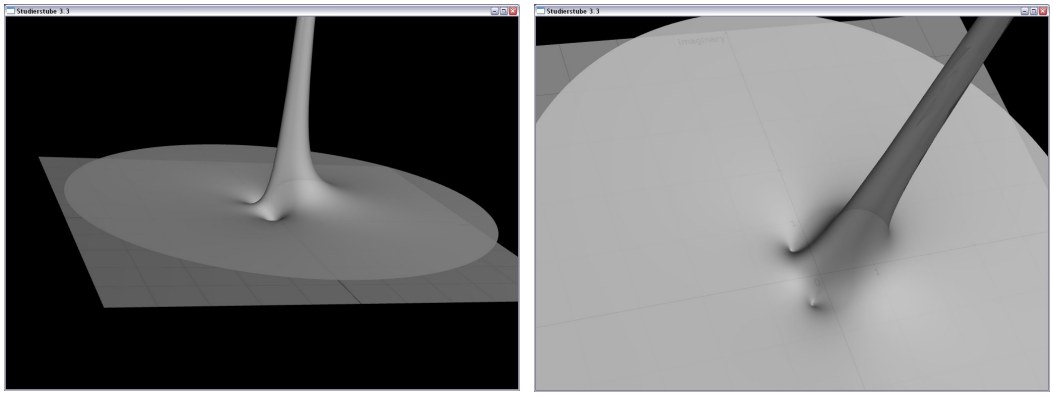


Figure 36: transfer function example screenshots

The transfer function clearly shows two zeros and one pole above the origin, for more insight we again extend our experiment as with the Mandelbrot set example of section 6.1.

To read the frequency response of the filter we need to evaluate $H(e^{j\Omega})$ instead of $H(z)$. The term $e^{j\Omega}$ is a complex number cycling around the unit circle with the parameter Ω, which relates to the frequency of the signal as $\Omega = \omega T = 2\pi \frac{f}{f_s}$, where f_s is the sampling frequency. We illustrate this by a visualization node almost identical to the one shown above, but with a different domain set: the points of the unit circle. Therefore we use a SoComplexDomainSubSet_1D_RegularCircle node.

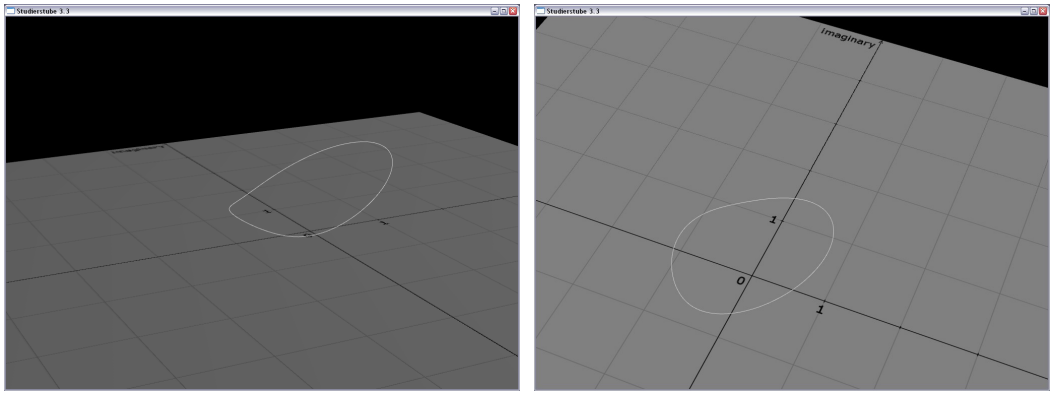


Figure 37: frequency response curve over the unit circle, screenshots

To read this curve the user has to know, that the frequencies whose response magnitude can be read from this curve are located around the circle as follows: 0 Hz is at 0° (above $1+0i$), $\frac{1}{4} f_s$ is at 90° (above $0+1i$) and so on until f_s is at 360° co-located with 0 Hz. In the image shown above for our example filter kernel we can read, that the low frequencies will have a higher magnitude than the frequencies around $\frac{1}{2} f_s$ which will be damped. So $h(n)$ is a kind of a so-called “low pass filter”.

Displaying both visualizations (surface and curve) combined will show the interrelation of zero and poles of the transfer function and the frequency response curve (fig. 38). The corresponding scene graph file contains the two visualization nodes explained above, they can use shared instancing for the *mappingList* and field connections for the *function* and the parameters, just the *domainSubSetList* has to be redefined. To better distinguish both objects from each other, different colors should be applied.

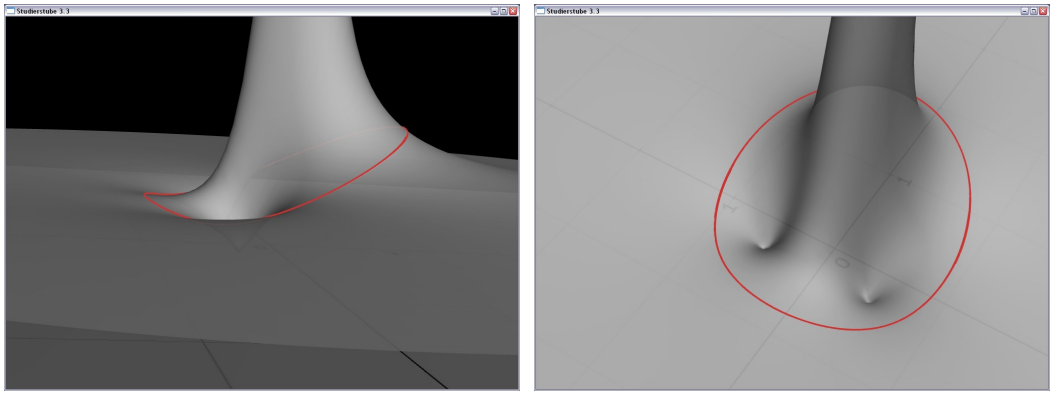


Figure 38: combined view of transfer function and frequency response curve, screenshots

Now the engineer who wants to design a filter needs to adjust the filter kernel interactively and observe the visual results. To achieve this in an AR setup we could use the personal interaction panel [32] provided in the Studierstube framework as a hand-held input device with draggable virtual controller elements, e.g. sliders as shown in fig. 39.

Such sliders provide numbers (for our example the filter kernel values $h(0)$, $h(1)$ and $h(2)$ in the range $[-1,1]$), that can be field-connected to our function parameters. See the Studierstube documentation [33] on how to setup the PIP (the example shown in fig. 39 required setting the *pen* and *pip* parts of the SoUserKit and the part *contextKit.templatePipSheet* of the SoApplicationKit). The example images below show parameter settings for a high pass filter on the left side and a low pass filter on the right side.

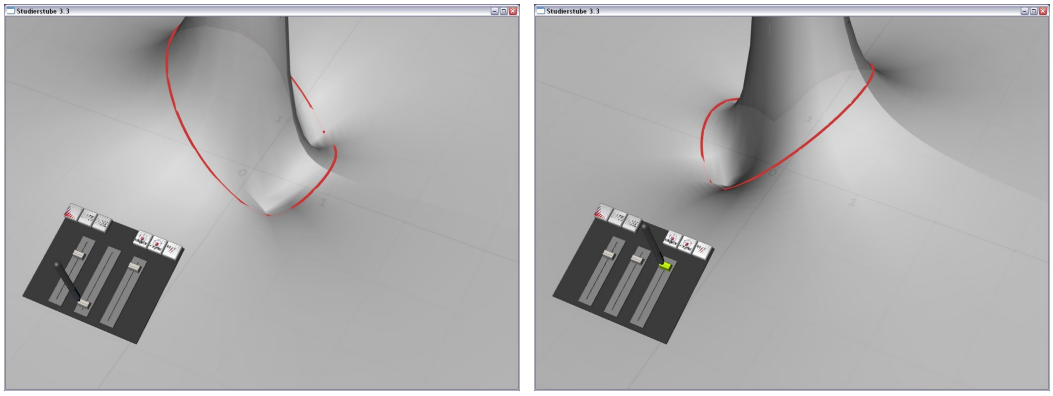


Figure 39: interaction with the PIP (same viewpoint), screenshot

As a last extension, to observe how the phase of a signal is affected in dependency of the frequency, arrows that display the arguments around the frequency response curve shall be added (fig. 40).

The corresponding visualization node will have a SoComplexDomainSubSet_1D_RegularCircle domain set just like the frequency response curve, but with a smaller number of samples around the circle. It shares the function and the parameters from the existing nodes and its *mappingList* will have the same mappings to X1, Y1 and Z1, but further mappings from CODOMAIN_ARGUMENT to POINT1OBJECT_ROTATION_Y and also from CODOMAIN_ARGUMENT to COLOR_H. The *point1Object* part now has to include an arrow object, a

narrow Cone rotated to point into the positive x axis will suffice. Attention has to be paid to not forget to set the *curveAsPointSet* field to TRUE, because no affine transformation mappings (as POINT1OBJECT_ROTATION_Y) will be applied to the vertices of a curve object, the domain set has to be treated as a number of single points.

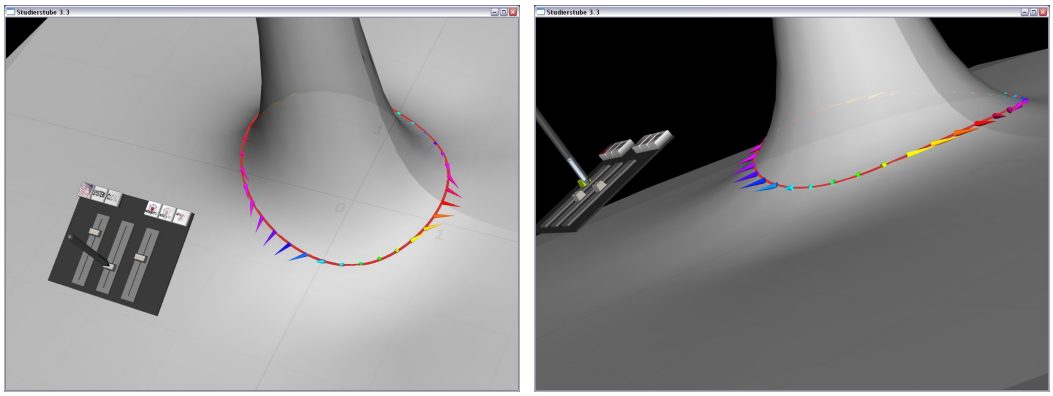


Figure 40: transfer function visualization extended by arrow plots, screenshot

To draw a conclusion from this visual experiment: Through an appropriate setup of a scene for the complex function graph visualization application, the matter of transfer functions and frequency response of digital finite impulse response filters in dependency of their filter kernel can be examined, the interrelations can be understood and filters with a desired behaviour can be designed by interacting with the parameters and observing the results in real time.

7 Conclusion

7.1 Findings

In this work an extensive survey of existing complex function graph visualization applications and related work has been done that found different solutions for visual mappings from an abstract set of numbers to a graphical representation that conveys important features of the complex function. Also concepts of using augmented reality technology to integrate virtual objects into real world environments for educational purposes were introduced in this context. Especially when multiple users share an augmented space, the benefits of dynamically created virtual objects combined with the natural means of human communication and interaction serve very well in a learning and teaching scenario.

A software framework has been developed, that integrates a freely configurable visualization pipeline in a scene graph library. All steps of a complex function graph visualization from the evaluation of the function for user defined domain subsets over an arbitrary visual mapping of the abstract data to manifold properties of a graphical object (shape and position, shading and even animation) over configuration of additional visual clues, like connection lines that visualize dependencies between specific points in space, to the rendering can be setup in the scene graph. As the data generation is also done in real time the user has interactive control over a non-static set of data, which enables him/her to explore mathematical interrelations of higher complexity.

The feature of mapping function graph properties to time dependent visualization properties is a novelty not found in the existing applications. As each visual mapping target has its own strengths and weaknesses, the parameters animation frequency and amplitude can cause high attention by the eye of the observer but don't allow as exact conclusion to the actual value as for example distance measures.

A concept of a data flow in multiple stages that allows user input in each stage is proposed, that allows a high level of control at an minimal level of necessary recomputation of data.

Two environment setups (an immersive one and a desktop setup) were examined and their advantages have been highlighted. Whereas the immersive setup provides an incomparable spatial impression intensified by the possibility to walk around the objects co-located with the real world, the desktop setup allows immediate access to the framework's features with minimal hardware requirements that suggest usage in classroom and homework scenarios.

Two comprehensive usage examples on actual topics of applied complex mathematics showed that scenes with complex function graph visualization objects can be setup very versatilely. They also showed that presenting too many features at once leads to a visual overload, so a step-by-step enhancement of reduced scenes that only show selected aspects of the function is proposed.

It has been shown, that the process of calculating the Mandelbrot set and the relation of the kernel of a digital finite impulse response filter and its transfer function (including the frequency response) can be presented purely graphically even without requiring the observer to be confident in complex calculus.

7.2 Future Work

One type of setup found in a number of related applications has not yet been followed in this project: a web application. It might be interesting and it is possible with the underlying APIs to set up a script on a server with a web page interface where a user could enter a complex function and visualization parameters into a dialog that invokes the generation of a VRML file which he/she can investigate from a web browser plug-in.

In the current version of the application, when changing the resolutions of grids and curves at high speed some stability issues arise. This bug has been tracked and located but it is connected to limitations of the Coin3D API. Fixing it would require a larger redesign of the framework from the current event based data updates using Sensor objects to a managed data flow organized by a custom scheduler.

As the software resulting from this thesis project could be of interest to a larger audience, writing a handbook that would require less previous knowledge in setting up a scene graph would increase its possibilities for propagation. A release version of the binaries with a short documentation and many example files is already finished.

Appendix A: A Survey on Function Plotters

A plethora of function plotter software solutions has been developed over the last decades, some of these tools are already installed on many computers, available online or sold in software shops. Reference [34] gives an overview about current plotting software and the following subsections show a number of examples.

Different classifications of these programs would be possible, some criteria to distinguish them are:

- if programs plot only curves in 2D, or if they also allow graphs as 3D meshes
- if the applications don't only visualize functions, but generic data sets as well
  - on the other hand some applications don't even have a function evaluator included, they have to be fed with the graph data from another program
- if color can be used per curve or per vertex
- if display is interactive or just generation of static pictures
- if parameters can be manipulated in real time; with a slider or a text field
- if vector plots are possible
- supported operating systems
- licence type, price and support availability
- comprehensiveness of documentation
- source code availability
- localization (language of user interface and documentation)
- if it is an executable file or if a scripting language interpreter (like python) is required
- if it is a web based application and in this case if it is
  - server sided (generate an image for a function and a set of parameters) or
  - client sided (java/flash applet)
- file import/export formats

Besides the dedicated function plotter software, so-called computer algebra systems (CAS) and numerical analysis applications exist that offer a large set of mathematics and computation functions but usually include plotting features as well.

In this context also spreadsheet applications have to be mentioned, despite that they are the least specialized applications for function plotting in this survey, they are still often referenced because of their incomparable spreading on a majority of desktop PCs. They offer another way to generate data through evaluation of a function and plot a graph of this data.

Inside each section the lists are ordered by the number of references to them found during this survey.

A.1 Function Plotter Applications

- gnuplot [35]
  - a command-line driven interactive data and function plotting utility
  - developed since 1986

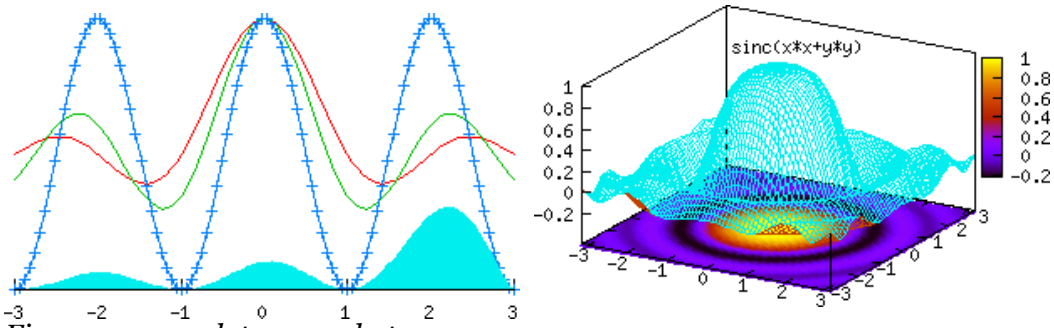


Figure 41: gnuplot screenshots

- LabPlot [36]
  - a data analysis and visualisation program
  - with an extensive parser for creating 2d, 3d functions
  - a majority of its features is related to generic data analysis

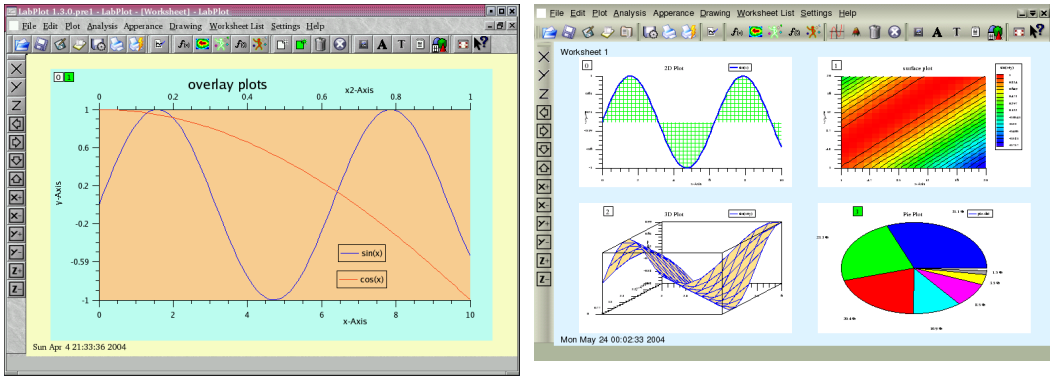


Figure 42: LabPlot screenshots

- Grapher [37], included with Apple OS X 10.4 [38]
  - a program to create 2 and 3 dimensional graphs from real and complex equations as well as from differential equations
  - function parameters can be kept separately from the function
    - a set of graphs can be shown for a range of parameter values
    - an animation can be generated for parameter values being interpolated in a given range at a given speed
  - successor of Curvus Pro X software (bought by Apple Inc. in 2004)

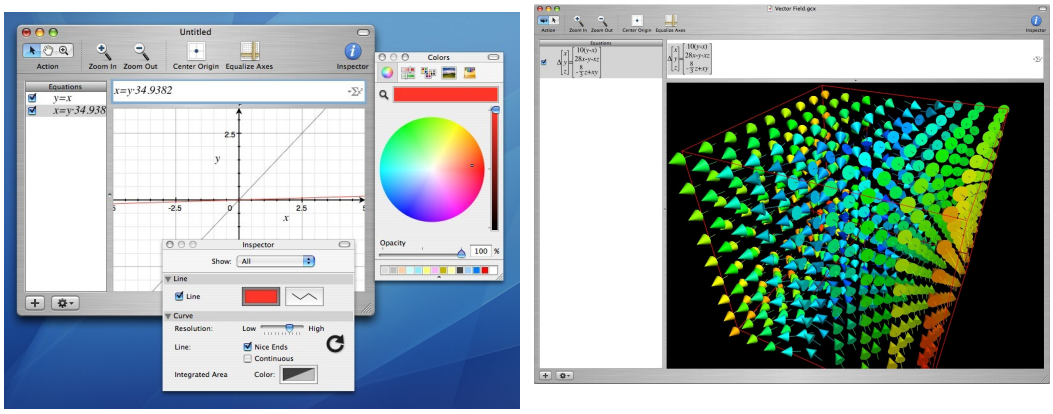


Figure 43: Grapher screenshots

- Curvus Pro [39]
  - plot and evaluate, in 2D and 3D, explicit or implicit curves and graphs, solutions of differential equations, vectorial or scalar fields as well as logical relations
  - predecessor of Grapher

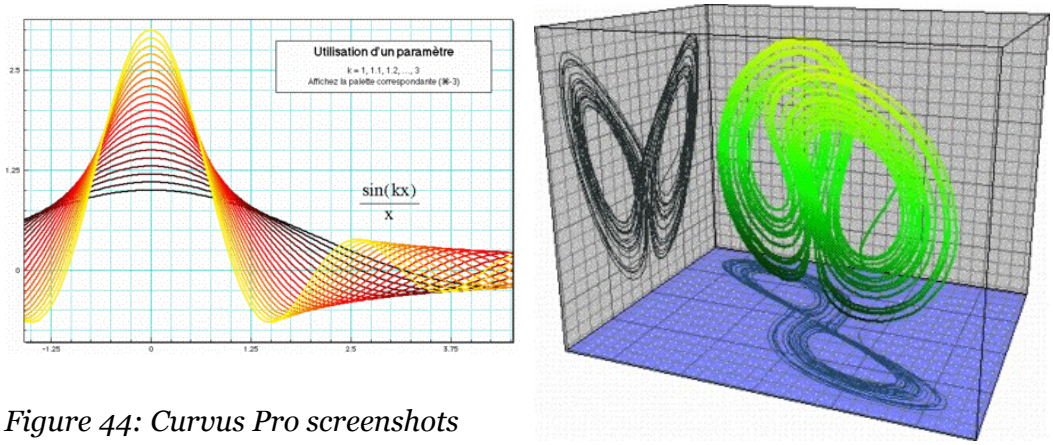


Figure 44: Curvus Pro screenshots

- pro Fit [40]
  - application for data/function analysis, plotting, and curve fitting
  - emphasis on generic data sets

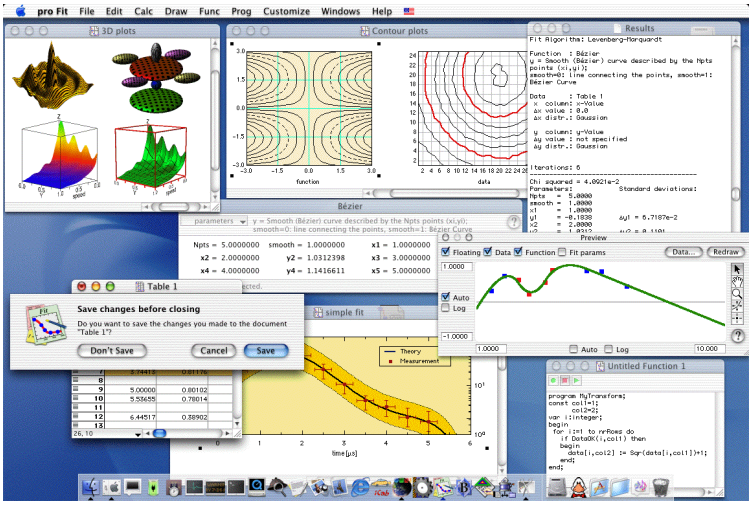


Figure 45: pro Fit screenshot

- SigmaPlot [41]
  - comprehensive program to produce high-quality graphs of functions and data sets
  - emphasizes as well on generic data sets

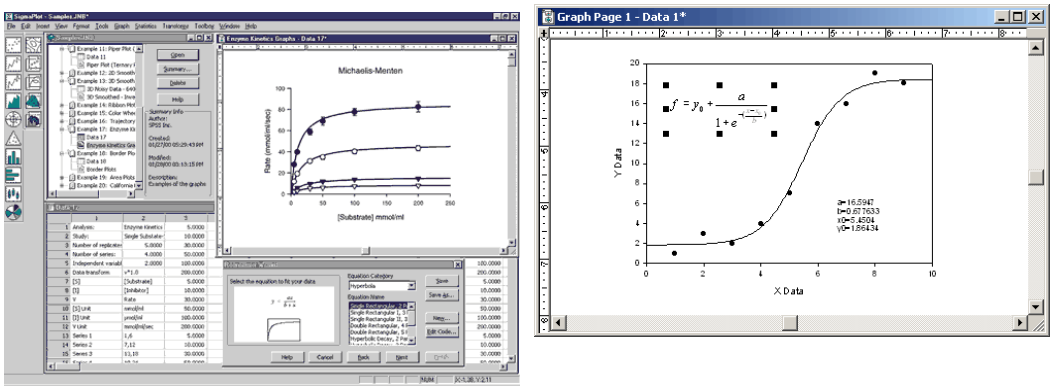
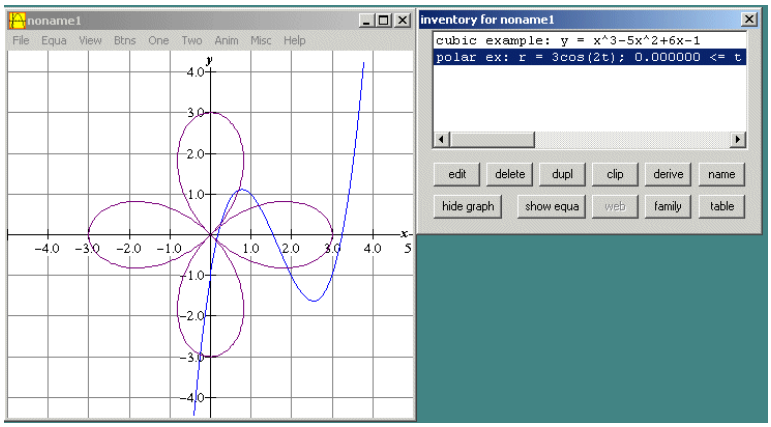


Figure 46: SigmaPlot screenshots

- Winplot [42]
  - general-purpose function plotting utility
  - 2D, 3D and vector plots



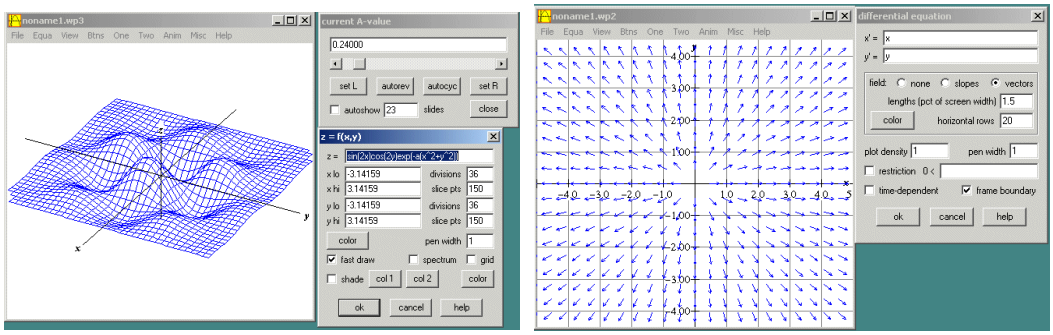


Figure 47: Winplot screenshots

- math4u2 [43]
  - educational mathematics system
  - features parameter value sliders

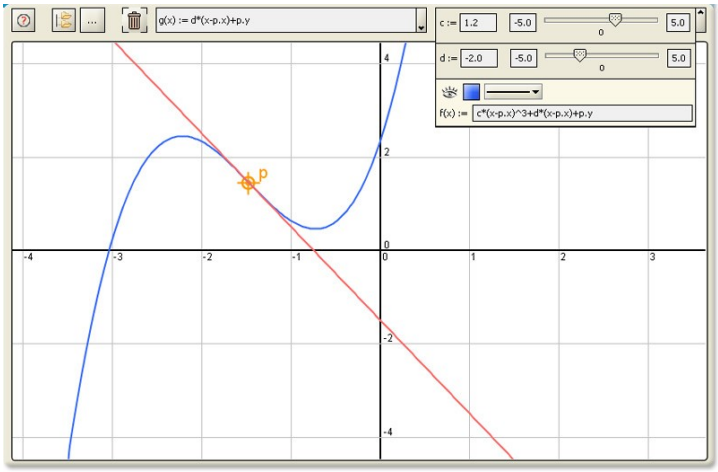


Figure 48: math4u2 screenshot

- KaleidaGraph [44]
  - application for graphing, regression analysis and statistical analysis
  - includes a 2D function plotter

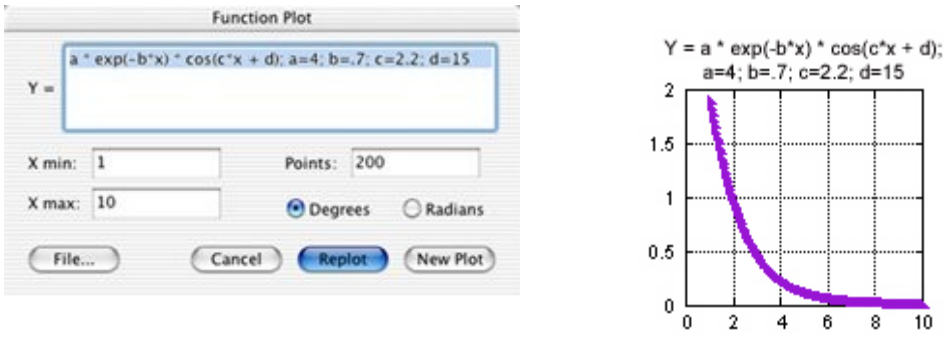


Figure 49: KaleidaGraph screenshot

- DeadLine [45]
  - combines graph plotting with numerical Calculus

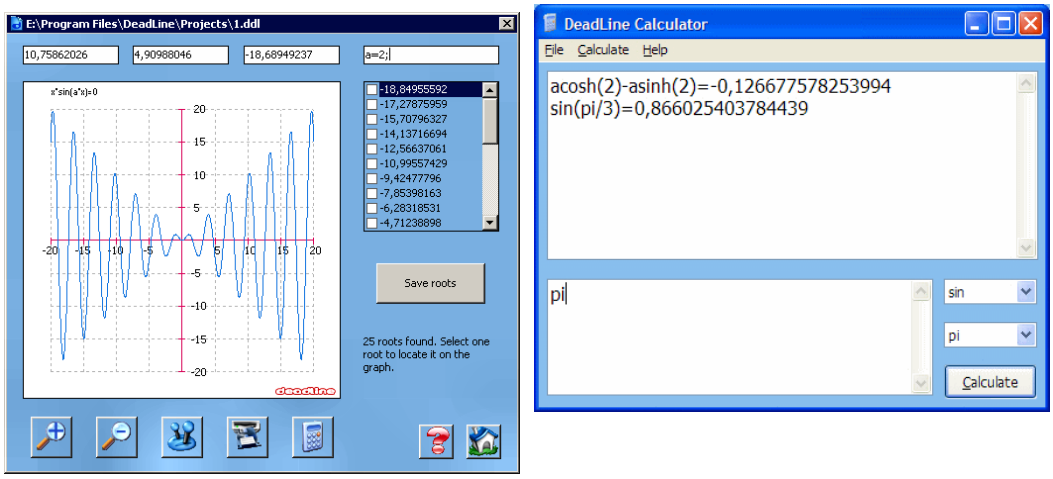


Figure 50: DeadLine screenshots

- Descartes [46]
  - image, data, and function plotter
  - runs on top of the Python programming language

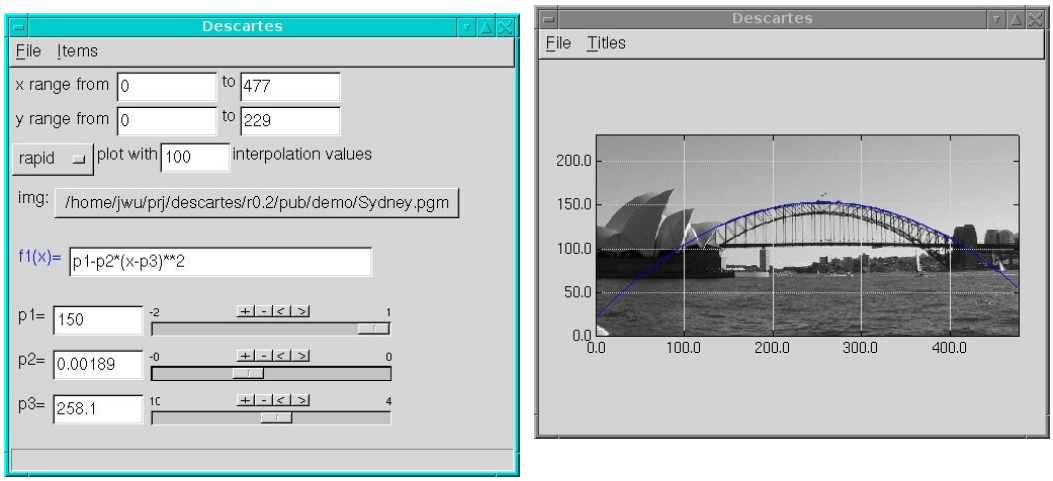


Figure 51: Descartes screenshot

A.2 Web-based Function Plotters

A.2.1 Server Side Plot Image Generators

- MAFA function plotter [47]
  - web interface for function plot images generated by a server sided PHP script
  - produces static images

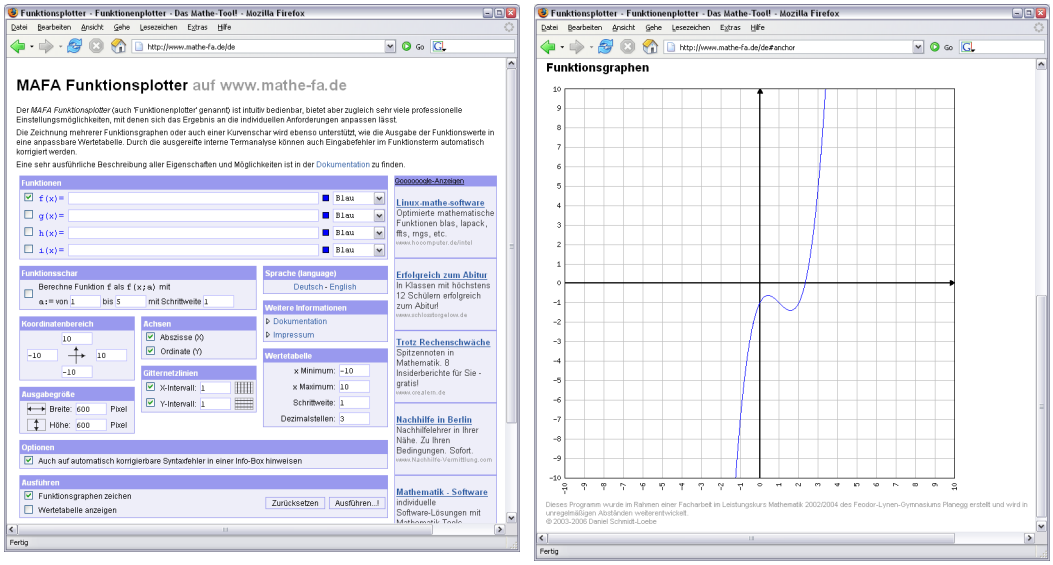


Figure 52: MAFA function plotter web page

- nanoGraph [48]
  - concept similar to MAFA

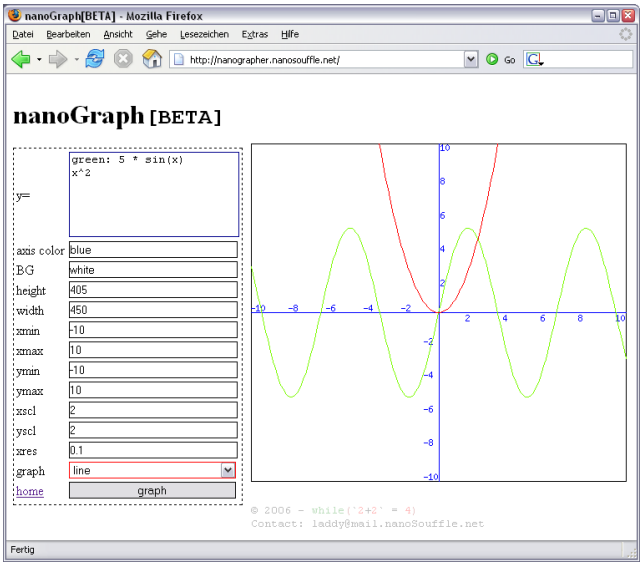


Figure 53: nanoGraph web page

A.2.2 Java/Flash Applets

- 2D function graph plotter by Arndt Brünner [49]
  - java applet to draw function graphs

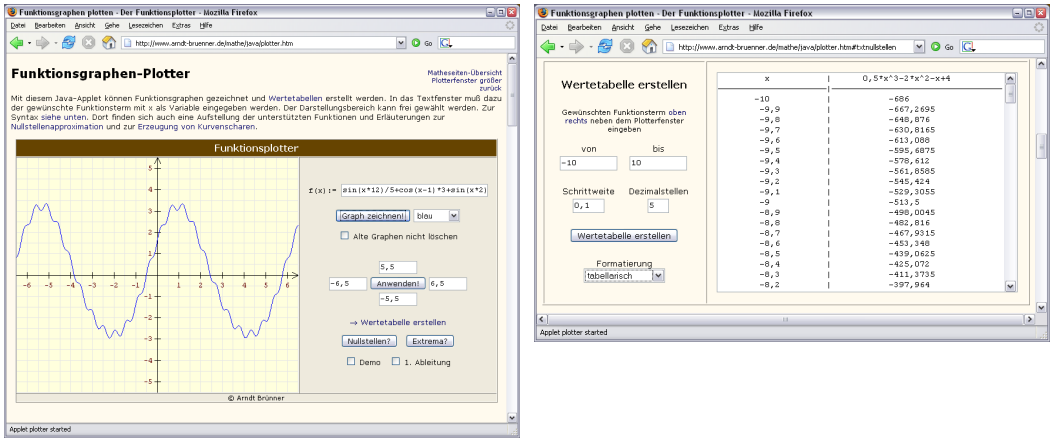


Figure 54: function plotter applet screenshot

- 3D function graph plotter by Arndt Brünner [50]
  - similar to the applet shown above, but here a 3D-plotter for functions where z=f(x,y)

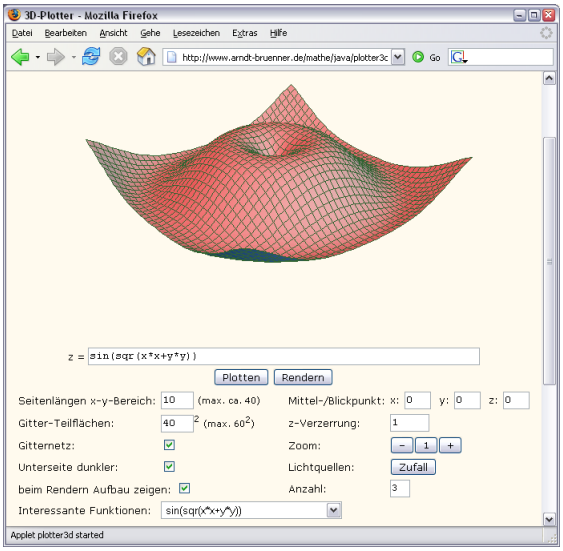


Figure 55: 3D plotter applet screenshot

- mathe-online.at function plotter [51]
  - another java applet to draw 2D function graphs

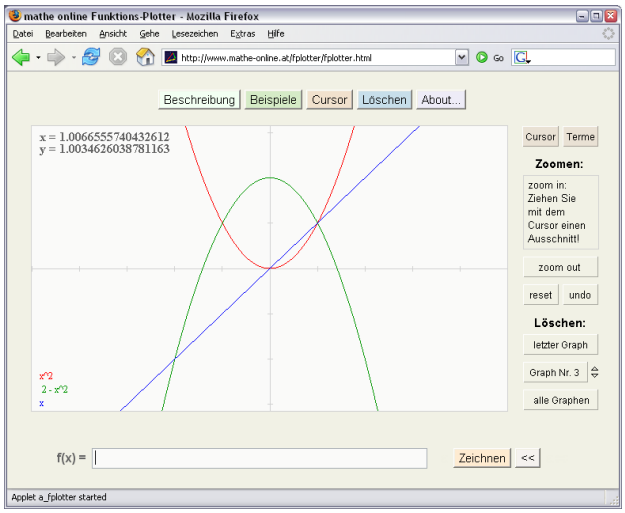


Figure 56: mathe-online.at function plotter applet screenshot

- vibos plotter [52]
  - a 2D function plotter implemented in Macromedia flash

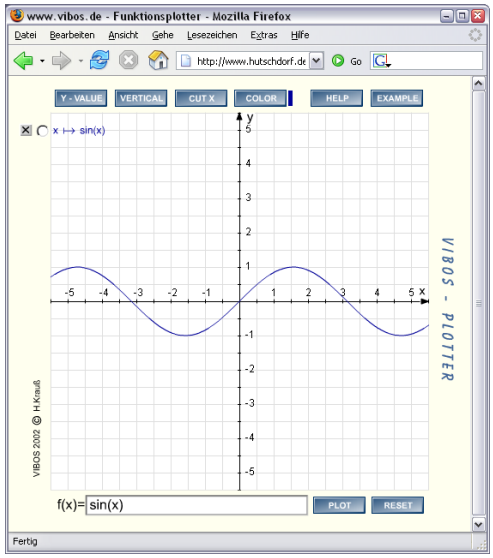


Figure 57: vibos plotter flash applet screenshot

A.3 Generic Data Plotters

- Origin [53]
  - professional graphing and data analysis software for scientists and engineers
  - exhaustive set of generic data set analysis features

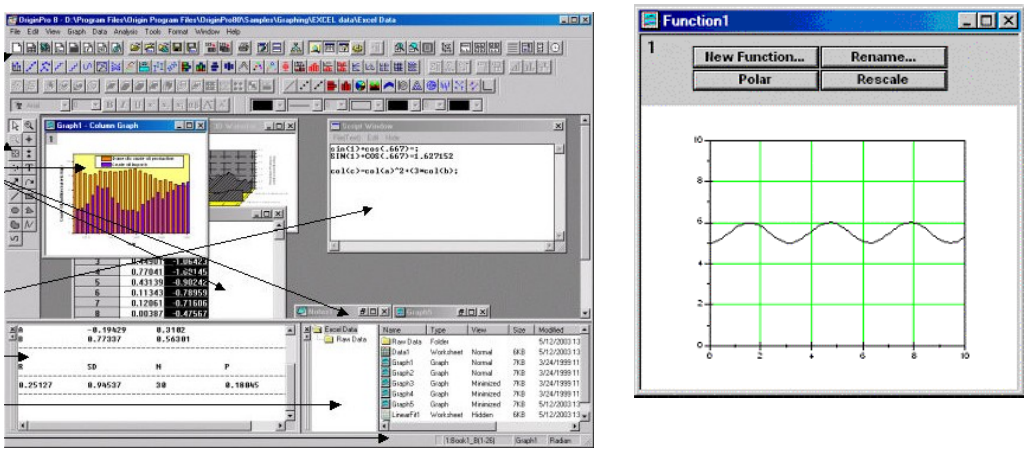


Figure 58: Origin screenshots

- QtiPlot [54]
  - data analysis and scientific plotting
  - aims to be an open source, platform independent clone of Origin

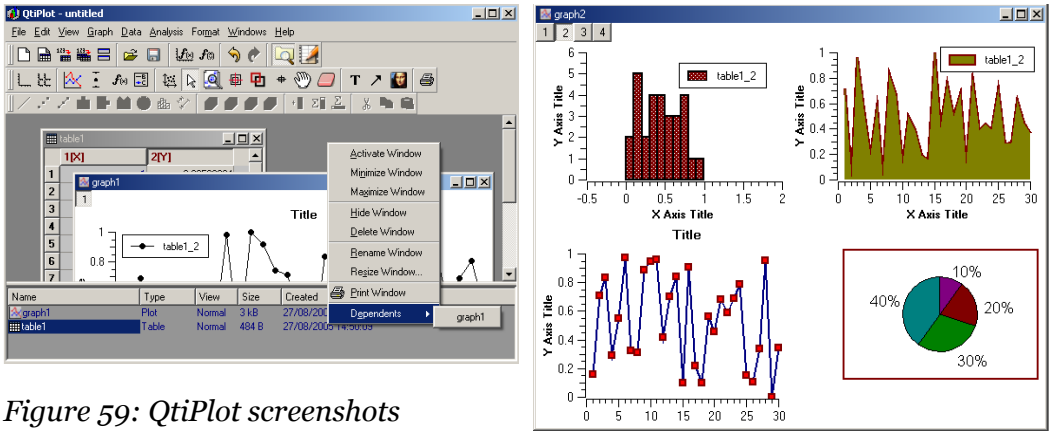


Figure 59: QtiPlot screenshots

- Grace [55]
  - 2D plotting tool

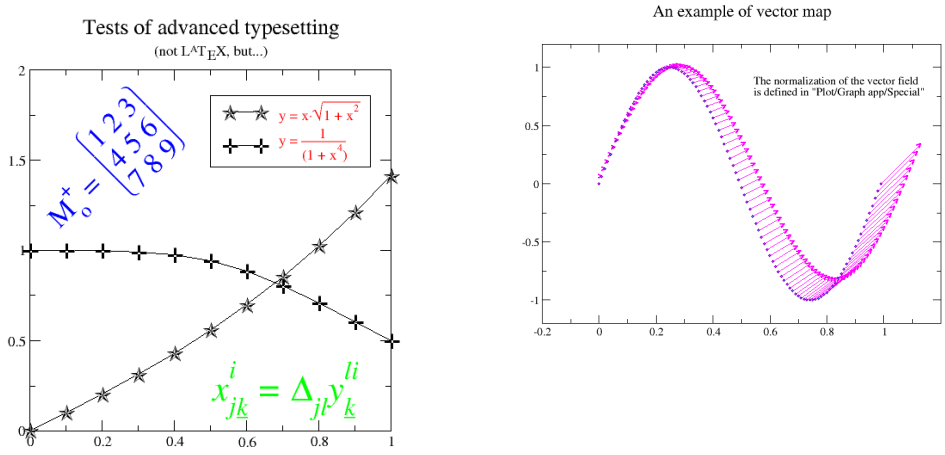


Figure 60: Grace output images

- kst [56]

- plotting and data viewing program

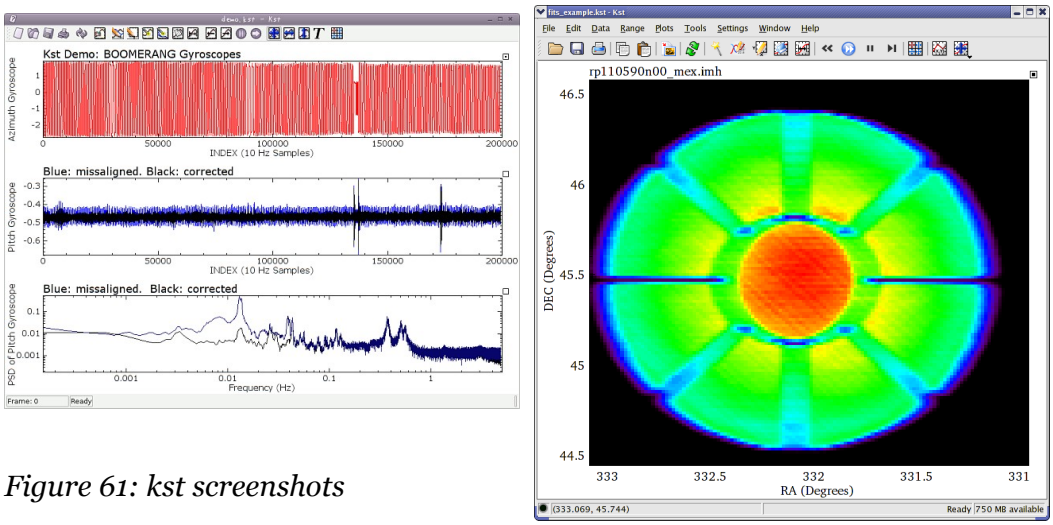


Figure 61: kst screenshots

- KChart [57]
  - charting tool of the KDE KOffice package

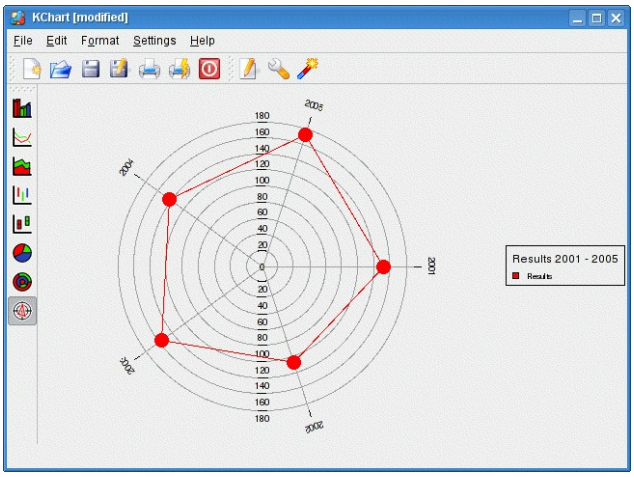


Figure 62: KChart screenshot

- ploticus [58]
  - program for producing plots and charts from data

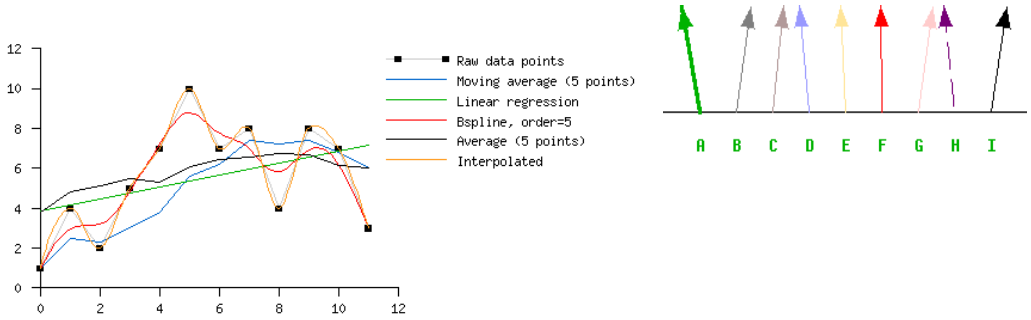


Figure 63: ploticus output images

A.4 Math Packages

A.4.1 Mathematica

Mathematica [59] is a widely-used computer algebra system by Wolfram Research Inc.. Its web site describes it as: “Mathematica seamlessly integrates a numeric and symbolic computational engine, graphics system, programming language, documentation system, and advanced connectivity to other applications.”. The output capabilities of this system include 2D and 3D plotting as well as even sound output.

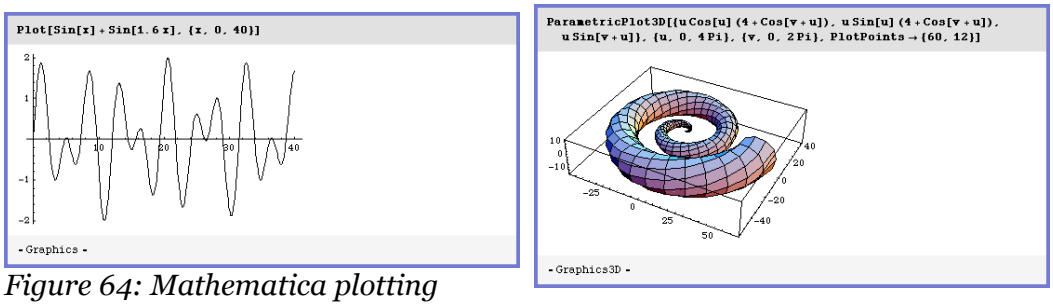


Figure 64: Mathematica plotting screenshots

A.4.2 Matlab

Matlab [60] is a numerical computing environment and programming language. “Created by The MathWorks, MATLAB allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages” [61].

It includes a set of plotting functions that are especially powerful in connection with its programming language.

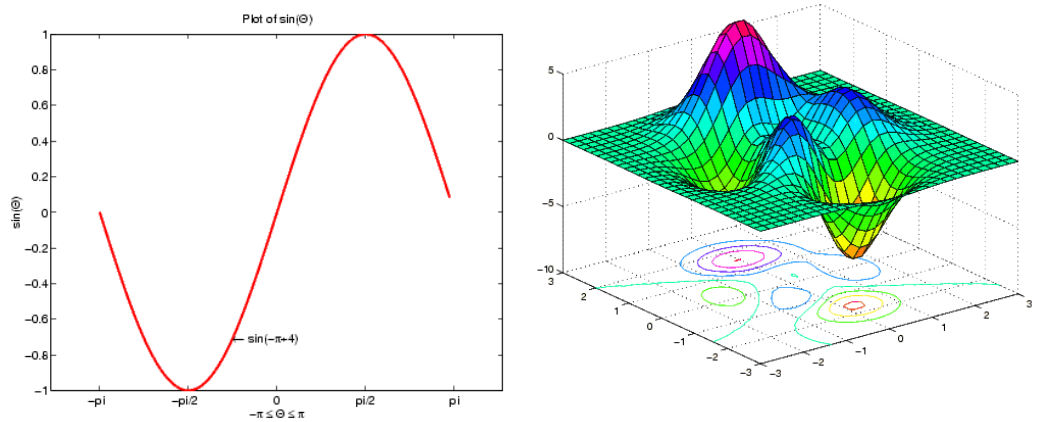


Figure 65: Matlab plotting screenshots

A.5 Spreadsheets

Even this type of applications allow function plotting, but a number of steps to generate the data and to select a proper graphing style (line chart) has to be done manually.

- Microsoft Excel [62]
  - a spreadsheet program that features an intuitive interface and capable calculation and graphing tools

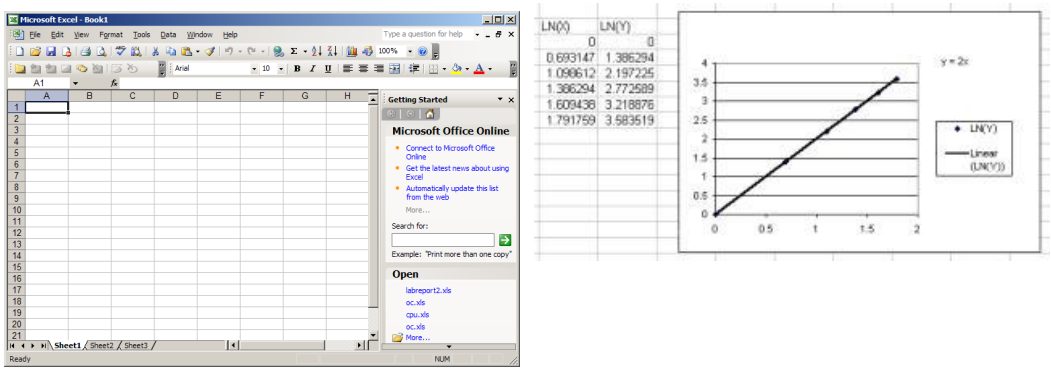


Figure 66: Microsoft Excel screenshots

- OpenOffice.org Calc [63]
  - a spreadsheet similar to Microsoft Excel, with a roughly equivalent range of features

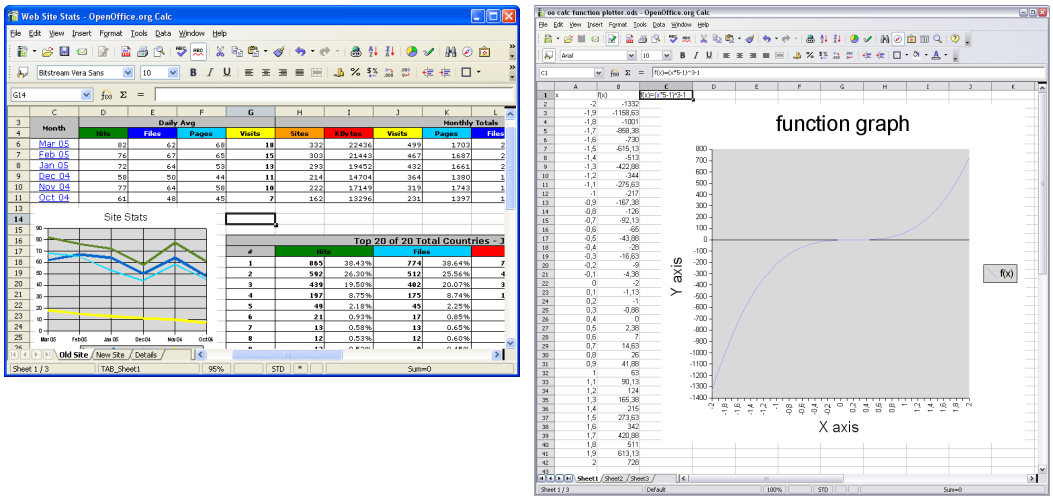


Figure 67: OpenOffice.org Calc screenshots

- Gnumeric [64]
  - a spreadsheet that is part of the GNOME desktop environment

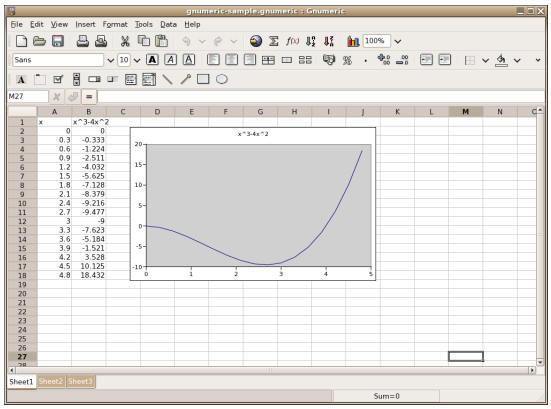


Figure 68: Gnumeric screenshot

List of References

[1] H. Kaufmann and D. Schmalstieg, "Mathematics And Geometry Education With Collaborative Augmented Reality", Computers & Graphics, 27(3), 2003.

[2] U. Zölzer,"Fundamentals of Digital Signal Processing", in "DAFX - Digital Audio Effects", John Wiley & Sons, Ltd. (publisher), pp. 1-30, 2002.

[3] web page, "Lascaux Software", last visit 16.04.2006, http://www.lascauxsoftware.com/

[4] web page (webarchive), "g(z): a tool for visual complex analysis", last visit 16.04.2006, http://web.archive.org/web/20041010184437/http://www.cs.brown.edu/people/dla/ma126/

[5] web page, "Real-Time Zooming Math Engine (rtzme)", last visit 16.04.2006, http://rtzme.sourceforge.net/

[6] web page, "A Complex Function Viewer (Java)", last visit 16.04.2006, http://sunsite.ubc.ca/LivingMathematics/V001N01/UBCExamples/ComplexViewer/complex.html

[7] web page, "Bombelli - a JAVA Complex Function Viewer", last visit 16.04.2006, http://www.dmat.ufpe.br/~ssc/bombelli/

[8] web page, "Complex Function Grapher", last visit 16.04.2006, http://www.math.ksu.edu/~bennett/jomacg/

[9]: George Abdo and Paul Godfrey, "Table of Conformal Mappings Using Continuous Coloring", web publication, last visit 16.04.2006, http://my.fit.edu/~gabdo/

[10]: Douglas N. Arnold, "Graphics for Complex Analysis", web publication, last visit 16.04.2006, http://www.ima.umn.edu/~arnold/complex.html

[11]: Tom Banchoff and Davide Cervone, "Understanding Complex Function Graphs", web publication, last visit 16.04.2006, http://www.geom.umn.edu/~dpvc/CVM/1997/01/ucfg/welcome.html

[12]: Frank A. Farris, "Complex Function Visualization", web publication, last visit 16.04.2006, http://www-acc.scu.edu/~ffarris/complex.html

[13] F. A. Farris, "Review of Visual Complex Analysis by Tristan Needham", American Mathematical Monthly vol. 105, June-July issue, 1998.

[14] T. Needham, "Visual Complex Analysis", Clarendon Press, Oxford, 1997.

[15] web page; Paul Fishback, "Resources for the Teaching of Complex Variables", last visit 16.04.2006, http://faculty.gvsu.edu/fishbacp/complex/complex.htm

[16]: Hans Lundmark, "Visualizing complex analytic functions using domain coloring", web publication, last visit 16.04.2006, http://www.mai.liu.se/~halun/complex/complex.html

[17] web page; M. Probst, "The MathMap GIMP Plug-In", last visit 16.04.2006, http://www.complang.tuwien.ac.at/~schani/mathmap/

[18]: Martin Pergler, "Newton's method, Julia and Mandelbrot sets, and complex coloring", web publication, last visit 16.04.2006, http://users.arczip.com/pergler/mp/documents/ptr/

[19] web page; Aguinaldo Robinson de Souza, "Functions of one Complex Varable - Visualization and Graphical Interpretation", last visit 16.04.2006, http://sorzal-df.fc.unesp.br/~edvaldo/en/index.htm

[20] R. Azuma, "A Survey of Augmented Reality", PRESENCE: Teleoperators and Virtual Environments, vol. 6, pp. 355-385, 1997.

[21] P. Milgram, H. Takemura, A. Utsumi, and F. Kishino, "Augmented reality: a class of displays on the reality-virtuality continuum," Proceedings of Telemanipulator and Telepresence Technologies, pp. 282-292, 1994.

[22] H. Kaufmann, "Geometry Education with Augmented Reality", PhD thesis, http://www.ims.tuwien.ac.at/publication_detail.php?ims_id=135, 2004.

[23] G. Taxén and A. Naeve, "CyberMath: Exploring Open Issues in VR-Based Learning", SIGGRAPH 2001 Conference Abstracts and Applications, pp. 49-51, 2001.

[24] C. Dede, M. C. Salzman, and R. B. Loftin, "ScienceSpace: Virtual Realities for Learning Complex and Abstract Scientific Concepts", Proceedings of IEEE VRAIS '96, pp. 246-252, 1996.

[25] A. Yeh, "VRMath: knowledge construction of 3D geometry in virtual reality microworlds", CHI '04 extended abstracts on Human factors in computing systems, pp. 1061-1062, 2004.

[26] M. Kalkusch, "Cash Flow - A Visualization Framework for 3D Flow Data", diploma thesis, http://www.studierstube.org/stb-thesis.php, 2005.

[27] Josie Wernecke. The Inventor Mentor: Extending Open Inventor. publ: Addison-Wesley, 2nd edition edition, 1993.

[28] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. S. Szalavári, L. M. Encarnacao, M. Gervautz, and W. Purgathofer, "The Studierstube augmented reality project", Presence-Teleoperators and Virtual Environments, vol. 11, pp. 33-54, 2002.

[29] G. Reitmayr and D. Schmalstieg, "An Open Software Architecture for Virtual Reality Interaction," in ACM Symposium on Virtual Reality Software & Technology 2001 (VRST 2001). Banff, Alberta, Canada, 2001.

[30] R.B. Haber and D.A. NcNabb, "Visualisation Idioms: A Conceptual Model for Scientific Visualisation Systems", Visualisation in Scientific Computing, G.M. Nielson and B. Shriver eds., IEEE Computer Society Press, 1990.

[31] H. Schumann and W. Mueller, "Visualisierung", Springer, 2000.

[32] S. Szalavári and M. Gervautz, "The Personal Interaction Panel - A Two-Handed Interface for Augmented Reality," Computer Graphics Forum, vol.16, pp. 335-346, 1997.

[33] web page, "Studierstube Documentation", last visit 16.04.2006, http://www.studierstube.org/doc/stb/

[34] web page (wiki), "List of information graphics software - Plotting", last visit 15.04.2006, http://en.wikipedia.org/wiki/List_of_graphing_software#Plotting

[35] web page, "gnuplot homepage", last visit 15.04.2006, http://www.gnuplot.info/

[36] web page, "LabPlot Homepage", last visit 15.04.2006, http://labplot.sourceforge.net/

[37] web page (wiki), "Grapher", last visit 15.04.2006, http://en.wikipedia.org/wiki/Grapher

[38] web page, "Apple - Mac OS X", last visit 15.04.2006, http://www.apple.com/macosx/

[39] web page, "Curvus Pro", last visit 15.04.2006, http://www.arizona-software.ch/classic/curvuspro/en/

[40] web page, "QuantumSoft and pro Fit: plotting, data analysis and curve fitting for Mac OS", last visit 15.04.2006, http://www.quansoft.com/

[41] web page, "Systat Software Inc. - SigmaPlot", last visit 15.04.2006, http://www.systat.com/products/SigmaPlot/

[42] web page, "Winplot", last visit 15.04.2006, http://math.exeter.edu/rparris/winplot.html

[43] web page, "math4u2", last visit 15.04.2006, http://www.math4u2.de/

[44] web page, "KaleidaGraph - scientific graphing, curve fitting, data analysis software", last visit 15.04.2006, http://www.synergy.com/kgraph.htm

[45] web page, "Solve equations, plot graphs free. DeadLine OnLine educational software", last visit 15.04.2006, http://deadline.3x.ro/

[46] web page, "descartes", last visit 15.04.2006, http://descartes.sourceforge.net/

[47] web page; Daniel Schmidt-Loebe, "Function plotter - Plot your graphs and charts easily!", last visit 16.04.2006, http://www.mathe-fa.de/en

[48] web page, "nanoGraph[BETA]", last visit 16.04.2006, http://nanographer.nanosouffle.net/

[49] web page; A. Brünner, "Funktionsgraphen plotten - Der Funktionsplotter", last visit 16.04.2006, http://www.arndt-bruenner.de/mathe/java/plotter.htm

[50] web page; A. Brünner, "3D-Plotter", last visit 16.04.2006, http://www.arndt-bruenner.de/mathe/java/plotter3d.htm

[51] web page, "mathe online Funktions-Plotter", last visit 16.04.2006, http://www.mathe-online.at/fplotter/fplotter.html

[52] web page; H. Krauß, "www.vibos.de - Funktionsplotter", last visit 16.04.2006, http://www.hutschdorf.de/flash/plotter.htm

[53] web page, "OriginLab - Origin - scientific graphing, data analysis, curve fitting software", last visit 16.04.2006, http://www.originlab.com/

[54] web page, "QtiPlot", last visit 16.04.2006, http://soft.proindependent.com/qtiplot.html

[55] web page, "Grace Home", last visit 16.04.2006, http://plasma-gate.weizmann.ac.il/Grace/

[56] web page, "kst - plots scientific data", last visit 16.04.2006, http://kst.kde.org/

[57] web page, "The KOffice Project - KChart", last visit 16.04.2006, http://www.koffice.org/kchart/

[58] web page, "ploticus", last visit 16.04.2006, http://ploticus.sourceforge.net/

[59] web page, "Mathematica: The Way the World Calculates", last visit 16.04.2006, http://www.wolfram.com/products/mathematica/index.html

[60] web page, "The MathWorks - MATLAB® - The Language of Technical Computing", last visit 16.04.2006, http://www.mathworks.com/products/matlab/

[61] web page (wiki), "MATLAB", last visit 16.04.2006, http://en.wikipedia.org/wiki/Matlab

[62] web page, "Microsoft Office Online: Excel 2003 Home Page", last visit 16.04.2006, http://office.microsoft.com/excel

[63] web page, "Calc", last visit 16.04.2006, http://www.openoffice.org/product2/calc.html

[64] web page, "GNOME Office / Gnumeric", last visit 16.04.2006, http://www.gnome.org/projects/gnumeric/