UNIVERSITÄT OSNABRÜCK

**Faculty of
Cognitive Science**

# Bachelor Thesis

Romeo: Scripting Environment with interactive Visualizations

**Author:**    Simon Danisch
          sdanisch@email.de

**Supervisor:**   Prof. Dr.-Ing. Elke Pulvermüller

**Co-Reader:**   Apl. Prof. Dr. Kai-Christoph Hamborg

**Filing Date:**   01.02.2014

# I Abstract

This bachelor thesis is about writing a simple scripting environment for scientific computing, with focus on visualizations and interaction. Focus on visualization means that every variable can be inspected and visualized at runtime, ranging from a textual representation to complex 3D scenes. Interaction is achieved by offering simple GUI elements for all parts of the program and the visualizations. All libraries are implemented in Julia and modern OpenGL, to offer high performance, opening the world to scientists who have to work with large datasets. Julia is a novel high-level programming language for scientific computing, promising to match C speed, making it the optimal match for this project. -This section needs more work, and should probably be written in the end

# II Table of Contents

# III List of Figures

# IV List of Tables

# V Listing-Verzeichnis

# VI List of Abbreviations

| | |
|---|---|
| **GUI** | Graphical User Interface |
| **LLVM** | Low Level Virtual Machine |
| **IR** | Intermediate Representation |
| **gcc** | GNU Compiler Collection |
| **Matlab** | Matrix Laboratory |
| **REPL** | Read Eval Print Loop |
| **GPU** | Graphics Processing Unit |
| **GLSL** | OpenGL Shading Language |

# 1 Introduction

This Bachelor Thesis is about writing a fast and interactive 3D visualization environment for scientific computing. The focus is on usability, applied to all the different interfaces, ranging from abstract API interfaces to graphical user interfaces. The ultimate goal is to make scientific computing more accessible to the user. As Graphical User Interface (GUI) elements and editable text fields are supplied, one can also write and execute scripts, and immediately visualize all bound variables of the script and edit them via simple GUI elements like sliders. With this it is possible to implement rudimentary interactive programming or visual debugging, further helping the user to understand his algorithms.

The introduction is structured in the following way. First, an introduction to the general field of research and its challenges is given. From these challenges, the problems relevant to this thesis will be extracted. Finally this chapter will conclude with a solution to the problem, how to measure the success and give an outlook on the structure of the entire Bachelor Thesis.

## 1.1 Scientific Computing

Non professional programmers, using programming as a tool to solve research tasks. What they want: Easy to use Low overhead for small scripts Fast Fast and extensible libraries, especially for linear algebra Rich standard library

## 1.2 Contribution

The contribution of this thesis is an interactive visualization library tied to a fast scientific computing language, while entirely written in the same language.

## 1.3 Field of Research and Problem



Figure 1: different visualizations of $f(x, y, z) = \sin(\frac{x}{15}) + \sin(\frac{y}{15}) + \sin(\frac{z}{15})$, visualized with Romeo. From left to right: Isosurface with isovalue=0.76, Isosurface with isovalue=0.37, maximum value projection

The general research field is making the capabilities of computers more accessible and understandable. This is a very broad definition and there are many different ways of making it easier to use a computer. One of the first big steps was to move from coding in binary to assembly. Many more steps have followed, for example introducing graphical user interfaces, novel input devices like the mouse, understandable visualizations and so forth. All these advances have made computers usable even for people who don't have an education in computer science. In this bachelor thesis the field is scientific computing, which still has quite a lot of barriers for novel users. Scientific computing is usually about implementing mathematical equations, complex algorithms and manipulating and analyzing data. As it is difficult to offer easy to use graphical interfaces for this kind of work, most research is done in some specialized, high-level scientific computing language. As most high-level languages are relatively slow, but for a lot of algorithms state of the art performance is required, this has led to a dual system. Prototyping in a high-level language, and then redoing the work in a fast low-level language. That this is not the perfect work flow is immediately visible, and a lot of research has been put into making high-level languages faster. These efforts slowly pay off and there is a whole new range of languages, that claim to be easy to work with while being as fast as it can get. This is a relatively recent trend and hasn't fully arrived in scientific computing yet, as most languages still have their core implemented in another fast language, which makes it hard to extend them for non professional users. This is especially true for high performance visualization libraries, which mostly use C++ at their performance critical core. To leverage the extensibility of these libraries, this bachelor thesis implements a visualization library in a fast high level language. Visualizations where chosen as they are a crucial building blocks for many fields in scientific computing.

Consider the following function $f(x, y, z) = \sin(\frac{x}{15}) + \sin(\frac{y}{15}) + \sin(\frac{z}{15})$, which describes a 3D volume mathematically. This is a simple function, which is already not that easy to interpret. In figure 1, you can see different visualizations of f. Especially for more complex functions, visualizing might be the only way to get a deeper understanding of the values that a formula or algorithm produces. This deeper understanding is crucial for identifying problems in the underlying math, or extending the algorithm. Additionally, widgets and simple GUIs are indispensable, giving scientist an easy way to interact with their data and algorithms. This helps to further understand the dynamics of the data and quickly spot mistakes.

In summary, the software in this thesis (Romeo) focuses on research which involves writing short scripts, while playing around with some parameters and visualizing the results. An example would be a material researcher, who is investigating different 3D shapes and

materials and their reaction to pressure. The researcher would need to read in the 3D object he wants to analyze, have an easy way to tweak the material parameters and it would be preferable to get instant feedback on how the pressure waves propagate through the object.

## 1.4 Outlook

# 2 Background

## 2.1 Related Work

### 2.1.1 The Julia Programming Language

Bringing Julia's ease of use and speed to a dynamic visualization library is the declared goal, which makes Julia a crucial building block for this thesis. Julia was published in 2012, which makes it a very new language. Julia is a multi paradigma language for scientific computing. The focus on scientific computing means, that Julia's standard library is equipped with a lot of functions, data structures and specialized syntax for implementing complex math. It promises to approach C speed, while being dynamically typed language which is easy to use. Julia was created to integrate a lot of advantages from other languages into one language. These include [3]

- C like performance

- macros like in Lisp

- mathematical notations like Matlab

- good at general purpose programming as python

- easy for statistics as R.

This makes it the perfect language for a fast scientific visualization library.

### 2.1.2 IJulia

IJulia is the Julia language backend for IPython. IPython is a software stack, which was created to allow for interactive computing in Python. It offers an interactive shell to execute python scripts, GUI toolkits, tab completion and rich media visualizations. It comes with a web based notebook, which enables you to write formated documentations together with data, inlined plots and executable program sniplets. You can also formulate mathematical formulas in latex, which will get rendered and inlined nicely into an IJulia Notebook. See figure 4 for an example.

IJulia offers a very similar feature set compared to Romeo, but it has a different focus. The notebook is completely web based, concentrates on 2D visualizations and interactivity is mostly limited to the programming and not the graphics. 3D graphics are possible via Three.js, which is a powerfull 3D visualization library based on WebGL. The integration is just prototypical and limited to simple 3D meshes up to date.

### 2.1.3 Matlab

Matrix Laboratory (Matlab) is a numerical computing environment that comes with its own programming language. It was created in 1984 by Cleve Moler. He designed it to leverage the effort of accessing LINPACK and EISPACK for his students. Since then it grew to be a widely used tool for scientific computing, in all areas ranging from teaching to actual engineering uses in companies. It offers a broad range of functionality, including matrix manipulation, plotting of functions and data, creation of user interfaces and interfacing with a range of languages like C/C++, Javam Fortran and Python.

Matlab itself is written in C, C++, Java and MATLAB. It's proprietary software ranging around 2000 € [5], which can be extended via free, open source and proprietary modules like Simulink.

Romeo intends to lay out the ground work to provide something remotely similar together with Julia. It is quite far away in terms of functionality, but it builds upon a more modern architecture, which intends to solve some problems that have accumulated for Matlab. While Julia

### 2.1.4 Paraview and VTK

Paraview is one of the largest and well established scientific visualization libraries for 3D. It is very fast and has a huge amount of visualization capabilities. It shares many of its goals with Romeo, namely [4]

- Develop an open-source, multi-platform visualization application.

- Support distributed computation models to process large data sets.

- Create an open, flexible, and intuitive user interface.

- Develop an extensible architecture based on open standards.

It is a very big project. This amounts to a total of 3.642.105 lines of code written in 29 languages. The statistics can be found in table 1 and 3.

# 3 Design

All building blocks in this thesis are developed with the purpose in mind to give the user the possibility to visualize and interact with complex 2D and 3D data, while being able to easily extend the library. To enable this kind of functionality, a lot of parts of the infrastructure need to work seamlessly together. Certain design choices had to be made to guarantee this. As speed is the most constraining factor, this chapter will start by introducing the design choices that had to be made in order to achieve state of the art speed.
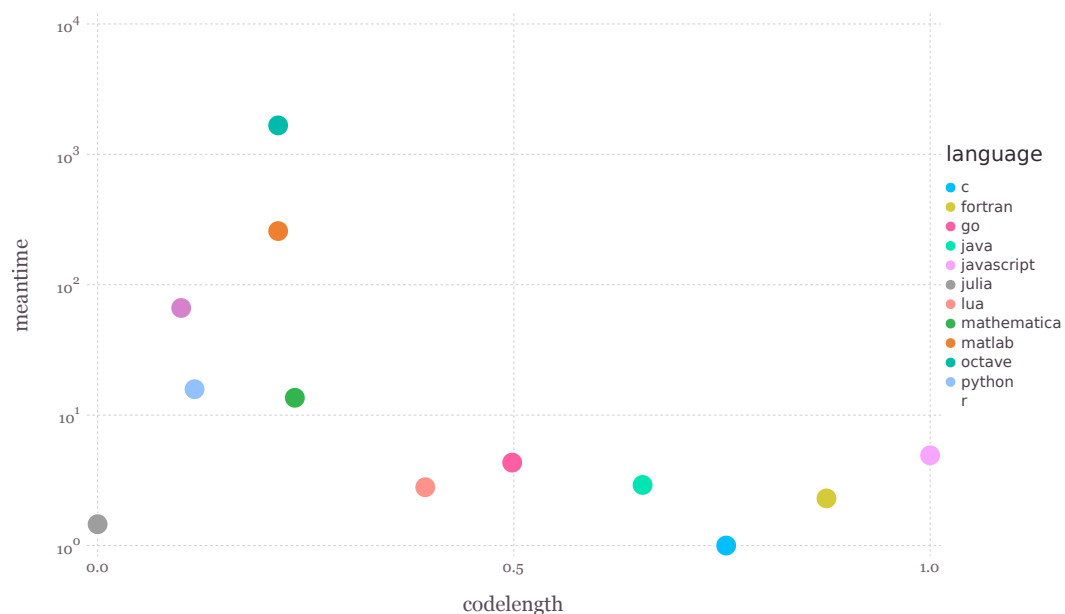


Figure 2: Languages speed relative to C (averaged benchmark results), plotted against the length of the needed code (Source in Appendix)

### 3.0.5 Speed

Speed is mainly a usability factor. It's a factor, that can make a software unusable, or render it unproductive. Because of this, speed has taken a high priority in this thesis. As general coding productivity is also a concern, this thesis is set on using a high level language. Historically, these two demands can't be both satisfied. How to achieve state of the art speed with a high level language is an ongoing research and basically the holy grail of language design. Luckily, there is a new programming language namely Julia building upon the compiler infrastructure Low Level Virtual Machine (LLVM), promising a concise, high-level programming style, while approaching C-performance. This is well illustrated in figure 2. Code length is an ambiguous measure for conciseness, but if the

code is similarly refactored it is a good indicator of how many lines of code need to be written to achieve the same goal. LLVM is an impressive compiler infrastructure, which has front ends for different languages and back-ends for different chip architectures. A language designer has the task, to emit LLVM Intermediate Representation (IR), which than gets just in time compiled and optimized to the architecture resulting in fast machine code. LLVM's concept is effective, as you can accumulate state of the art optimizations in one place, making them accessible to many languages, while being able to compile to different platforms. There are x86, ARM, OpenCL and CUDA back ends. While Julia doesn't support them all, it will hopefully be possible in the future. LLVM is also used by Clang, the C/C++ front end for LLVM rivaling GNU Compiler Collection (gcc) and it is used by Apple's programming language Swift. This makes LLVM a solid basis for a programming language, as these are highly successful projects guaranteeing LLVM further prospering of the technology.

To get high performant 3D graphics rendering, there are on the first sight a lot of options. If you start to take the previous demands into account, the options shrink down considerably, though. The visualization library should be implemented in one high level language, which can be used for scientific computing and has state of the art speed. At this point, there are close to zero libraries left. As you can see in figure 2, Matlab, Python and R disqualify, as they are too slow. JavaScript, Java, Go and Lua are missing a scientific background and the others are too low level for the described goals. This leaves only Julia, but in Julia there weren't any 3D libraries available, which means that one has to start from scratch. There are only a couple of GPU accelerated low-level libraries available, namely Khrono's OpenGL, Microsoft's DirectX, Apple's Metal and AMD's Mantel, which are offering basically the same functionality. As only OpenGL is truly cross-platform, this leaves only OpenGL as an option. So for the purpose of high speed visualizations, OpenGL was wrapped with a high-level interface written in Julia. This leaves us with one binary dependency not written in Julia, namely the video driver, which implements OpenGL.

Measurement of success is pretty straight forward, but the devil is in the detail. It's easy to benchmark the code, but quite difficult to find a baseline, as one either has to implement the whole software with the alternative technologies, or one has to find similar software. This thesis will follow a hybrid strategy, comparing some simple implementations with different technologies and choose some rivaling state of the art libraries as a baseline.

### 3.0.6 Extensibility

Extensibility is an important factor, which can decide, if a library is fit for scientific computing or not. It's not only that, but also a great factor determining growth of a

software, as the more extensible the software is, the higher the probability that someone else contributes to it. In order to write extensible software, we first have to clarify what extensibility is. Extensible foremost needs, that the code is accessible. There are different levels of accessibility. The lowest level is closed source, where people purposely make the code inaccessible. While this is obvious, it is just a special case of not understanding the underlying language. Just shipping binaries without open sourcing the code, means that the source is only accessible in a language which is extremely hard to understand, namely the machine code of the binary. So another example for inaccessibility is to write in a language that is difficult to understand. Other barriers are obfuscated language constructs, missing documentations and cryptic highly optimized code. Further more the design of the library in the whole is an important factor for extensibility. It's not only important, that all parts are understandable, but also, that every independent unit in the code solves only one problem. If this is guaranteed, re-usability in different contexts becomes much simpler. This allows for a broader user base, which in turn results in higher contributions and bug reports. Short concise code is also important, as it will take considerably less time to rewrite something, as the amount of code that has to be touched is shorter and less time is spend on understanding and rewriting the code.

So the code written for this thesis should be open source, modular, written in a high level language and concise.

This is pretty difficult to measure as these are either binary choices, which are either followed or not, or higher level concepts like writing concise code, which can be a matter of taste. To get an idea of the effectiveness of my strategy, usage patterns from Github will be analyzed.

### 3.0.7 Event System

For interaction events have to be handled. The chosen event system is named Reactive, which also reflects its design principle. Reactive programming is an event driven approach, using signals as the abstraction. Signals are values, which change over time, which can be transformed via functions, yielding a new signal. Here is an example to clarify the notion of signals:

```
1  a = Input(40) # an integer signal.
2  b = Input(2) # an integer signal.
3  c = lift(+, a,b) # creates a new signal with the value 42
4  push!(a, 20) # updates a, resulting in c being 22
```

The event system was chosen for two reasons: First, because it simply was the only available event system for Julia at that time. Secondly, it naturally represents change

over time, which is a perfect fit for animations.

### 3.0.8 Interfaces

Working with a computer means working with interfaces to a computer, which in the end simply jiggles around with zeros and ones. There is a huge hierarchy of abstractions involved, to make this process of binary juggling manageable to the human. We already dealt with the lowest relevant abstraction: the choice of programming language, which forms our first interface to the computer. The next level of abstraction is the general architecture of the modules, which has been discussed previously. This chapter is about the API design choices that have been made. The first API is the OpenGL layer. The philosophy is to make the wrapper for native libraries as thin and reusable as possible and an one to one mapping of the library itself. This guarantees re-usability for others, as they might be used to work only with the low-level library and they might disagree with any higher-level abstraction. Over this sits an abstraction layer needed to simplify the work with OpenGL. With this abstraction, the actual visualization library is implemented. APIs for visualization libraries are very difficult to realize, as there are endless ways of visualizing the same data. The design choice here was to use Julia's rich type system, to better describe the data. Julia makes this possible, as you can name the same data differently, without loosing performance. So you can actually have a unit like meter represented as a native floating point type and have the visualization specialize to this. Like this you can have a single function e.g. *visualize*, that does creates a default visualization for a lot of common data. It is parameterizable and can actually be overloaded for different styles. So the signature looks like this in the end:

```
1 visualization{data} = visualize(data, style=default, parameters=default)
```

The same principle is used for editing data, so there is also:

```
1 visualization{data}, signal{data} = edit(data, style=default, parameters
    =default)
```

Together with the event system which consists of signals, it is possible to edit and visualize rich data over a simple interface, which is perfect for visual debugging, as it is always the same function call applied to the data and no further user interaction is needed. It is also easy to extend, as the user just has to overload the function, with a custom style and or parameters. Finally, there are also graphical user interfaces developed for this thesis. As also optimizing them is out of the scope of this thesis, they are kept very simple. The measurement of success is again relatively difficult to do. (I need to think this over)

# 4 Technologies Used

## 4.1 The Julia Programming Language

The basic introduction of Julia has already been given in the Background chapter. This chapter is focused on how to write programs with Julia. Most influential language construct are it's hierarchical type system and multiple dispatch. Multiple dispatch is basically runtime function overloading. To better understand multiple dispatch, one has to be familiar with Julia's type system. The type system builds uppon four basic components. Composite types, which are comparable to C structs, parametric composite types, bitstypes, abstract and parametric abstract types. While the first three are all concrete types, abstract types can be used to build a type hierarchy. Every concrete type can inherit from one abstract type, while abstract types can also inherit from abstract types. So you can build type hierarchies like this:

```
1  abstract Number
2  abstract FloatingPoint{Size} <: Number # inherit from Number
3  bitstype 32 Float32 <: FloatingPoint{32} # inherit from a parametric
       abstract type
4  type Complex{T} <: Number
5      real::T
6      img::T
7  end
```

With this type hierarchy you can overload functions with abstract, concrete or untyped arguments.

```
1  foo{T}(x::Number, y::Complex{T}) = println("some number: ", x, " some
       complex Number: ", y) # shorthand function defintion
2  function foo(x, y::Float32)
3      println("overloading foo with a new signature")
4  end
```

This gives Julia a more functional feel, although that you can mimic object oriented programming fairly well. The functions are also easy to pass around. Functions can be bound to variables and can then be called like normal functions. One of the most crucial features is the very simple, overheadless C-Interface. Thanks to the binary compatibility of LLVM's emited assembly, a C function call to a shared library inside Julia has the same overhead as it would be from inside C[**?**]. This is perfect for dealing with low-level libraries like OpenCL and OpenGL.

## 4.2 OpenGL

OpenGL is a low-level graphics API implemented by the video card vendor via their video driver. As such it doesn't offer much abstraction over the actual Graphics Processing Unit (GPU), but instead offers high flexibility and performance. OpenGL 1.0 was released in 1992 and the current version is 4.5 with 5.0 aka Vulkan just around the corner. Not all video drivers implement the newest OpenGL standards. As a result, one has to decide to which OpenGL version to program against, allowing a trade-off between modernity and platform support. For Romeo, it was decided to support OpenGL 3.3 as the lowest bound, as it is sufficiently available, while still having most of the modern features. The features include instance rendering, vertex arrays and modern OpenGL Shading Language (GLSL) shader.

## 4.3 Reactive

is an functional event system based on ELM's language principles.

## 4.4 GLFW

Simple OpenGL context and window creation library written in C.
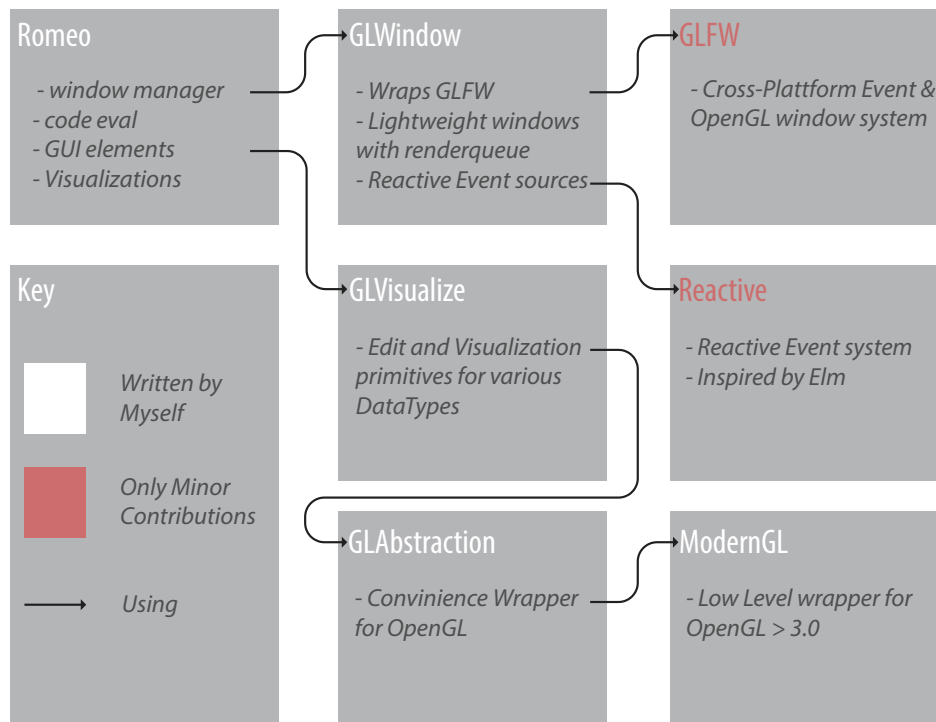
# 5 Implementation



Figure 3: Main modules used in Romeo and their relation (simplified)

This chapter is about the implementation of Romeo. The Romeo package itself is small and just defines the high-level functionality of the editor. This includes window layout and connecting all the different event sources to create the wanted behavior. To do this, Romeo relies on a multitude of packages, which step for step abstract away the underlying low-level code that is used to do the window creation and rendering. GLVisualize is the main package offering the rendering functionality and the editor widgets like text fields and sliders. For rendering GLVisualize relies on GLAbstraction, which defines a high-level interface to OpenGL. OpenGL function loading is done by ModernGL, which keeps all the function and Enums definitions from OpenGL with version higher than 3.0. The event management is handled by Reactive.

## 5.1 Reactive

Reactive is a functional event system designed for event driven programming. It implements Elm's signal based event system in Julia. Signals are values that change over time. These signals can be transformed via arbitrary functions which creates a new signal. This simple principle leads too a surprisingly simple yet effective way of programming event based applications.

```
1  a = Input(40) # an integer signal.
2  b = Input(2) # an integer signal.
3  c = lift(+, a,b) # creates a new signal with the callback plus. Equal to
      c = a+b
4  lift(println, c) # executes println, every time that c is updated.
5  push!(a, 20) # updates a, resulting in c being 22
6  #prints: 22
7  push!(b, 5) # updates a, resulting in c being 22
8  #prints: 25
```

Visualizing dynamic data makes the event system a crucial component of the library. The event system was challenging to integrate for several reasons. First of all Reactive is a functional event system, while OpenGL relies heavily on global states, which are two perpendicular concepts. Also, it doesn't allow to rearrange the event tree. In other words if you have two signal branches, there is no way to fuse them together. Finally, it doesn't reuse memory for signals. This means, if you have a signal for a large array every lift operation will allocate new memory for the array. Working around these shortcomings reduced the ease of use of the overall API. It has lead to two design choices which are sub optimal. First, the event system is decoupled from the render loop, to set the global OpenGL states appropriately. This means, that the structure is as follows:

```
1  #instead of:
2  a = Input(data)
3  b = lift(visualize, a)
4  c = lift(render_opengl, b)
5  # it is now:
6  while window.open
7    render(b.value)
8  end
```

The sub optimal performance of Reactive for large data has let to the following design:

```
1  # instead of
2  a = Input(large_array)
3  b = lift(some_computation, a, some_timed_signal)
4  c = lift(visualize, b)
5  # it is now:
6  a = visualize(large_array)
7  lift(some_timed_signal) do time
8    b = some_computation(large_array, time)
9    update!(a, b) # write directly to the \ac{GPU} object
10  end
```

Romeo: Scripting Environment with interactive Visualizations

## 5.2 ModernGL

OpenGL is implemented by the video card vendor and is shipped via the video driver, which comes in the form of a C-Library. The challenge is, to load the function pointer system and vendor independently. Also one further complication is, that depending on the platform, function pointer are only available after an OpenGL context was created and may only be valid for this context. [6] This problem is solved, by initializing a function pointer cache with null and as soon as the function is called the first time the real pointer gets loaded. This is suboptimal, as the pointer cannot be inlined and has to be checked for null. In the newest version of Julia, this can be implemented even more efficiently with staged functions. Staged functions can be thought of as a runtime macro. At the first call of the function code can be generated, which then will get compiled in time and replaces the function definition. This makes the an OpenGL function call nearly twice as fast. Like this, even C can be outperformed in terms of speed, as C doesn't have just in time compilation capabilities, so the function pointers can not be inlined like this. [benchmark pending, probably better in discussion though?]

## 5.3 GLAbstraction

GLAbstraction is the abstraction layer over ModernGL. It wraps OpenGL primitives like Buffers and Textures in Julia objects and hooks them up to Julia's garbage collector. Additionally, it implements convenient functions to load shader code and it makes it easy to feed the shader with the correct data types. Besides wrapping OpenGL, it also offers the linear algebra needed for the various 3D transformation and camera code. On top of that, it defines a perspective and orthographic camera type which can be used with signals to manipulate the parameters of the camera.

## 5.4 GLWindow

GLWindow is a lightweight wrapper around GLFW. GLFW is a C-Library, that offers cross-platform OpenGL window and context creation and event handling. GLWindows is the abstraction layer, that builds upon a Julia wrapper of GLFW. It mainly offers a screen type, which contains signals for all the different GLFW events (Mouse, Keyboard, etc...). It also offers a hierarchical structure for nesting screens in each other. All the screen areas are signals, which resize the screen area when they change. Like this, you can dynamically have the screen sizes depend on each other and react to resizing the window.

## 5.5 GLVisualize

GLVisualize implements the main functionality of this library. It offers rendering of different primitives. GLVisiualize is designed with two intentions in mind: supplying a very simple interface consisting of just two functions and transport the data with as little conversions and copies to the GPU. This allows to manipulate the data directly on the GPU, which is the fastest way to update dynamic data. The interface to create visualizations is very simple and only consists of two functions:

```
1  visualization          = visualize(data::T, style=Style{:default};
       parameters...)
2  visualization, signal = edit(       data::T, style=Style{:default};
       parameters...)
```

With this, the following data can be visualized:

- Text (Vector of Glyphs)

- Height fields with different primitives (Matrix of height values)

- 3D bar plots (Matrix of height values)

- Images (Matrix of color values)

- Videos (Vector of Images)

- Volumes (3D Array of intensities)

- Particles (Vector of Points)

- Vector Fields (3D array of directional Vectors)

- Colors (Single Color values)

These can be integrated into the same scene. For all of these, it is possible to change their values interactively. The edit function is making it easier to edit the values of the data. It calls the visualize function to render the data type and then registers appropriate events to update the data. Take a look at the text edit function. It first uploads the text to video memory and sets up the functionality to visualize it, and than updates the text data on the GPU according to the cursor position and keyboard input.

Up to now, there is only an edit function available for text fields, colors, numbers, vectors and matrices.

### 5.5.1 Romeo

So far Romeo just consists of one file with 500 lines of code. It just defines some simple text field, a search field, and a visualize and edit window. The texts gets evaluated as Julia code as soon as it changes. Like this, the text field acts like a very simple Read Eval Print Loop (REPL). Via the search field, you can execute simple Julia statements and the results will be displayed in the visualize window, while all parameters can be edited via the edit window. This means, if you type in a simple variable, the variable will be visualized. But you can also search and transform a variable via simple Julia terms.

# 6 Results and Discussion

## 6.1 Performance Analysis

benchmarks, benchmarks, benchmarks

## 6.2 Extendability Analysis

The modular design of Romeo has proven to be very effective and the goal of reusability has already proven itself. Most of the created modules are already used independantly by different people. GLVisualize is used by myself for two packages, namely GLPLot, a scientific plotting package for Julia and a for a prototype of a file explorer. It got forked by several users to create their own dynamic visualization packages. The same applies for ModernGL and GLAbstraction. Most other used packages are at least used by one other project. This indicates, that the abstraction and modularity is well designed, so that all the modules can function on their own.

The only exception is GLWindow, which has been used just indirectly through my other packages. This can mean three things. First, it is badly abstracted and doesn't cleanly wrap one use case. Secondly, it can be, that the use case is not entirely clear to other people, which would not be a big surprice considering the minimal amount of documentation for GLWindow. And finally, considering the small group of people deveoping graphics for Julia, it could be that they simply don't need the lower level functionality of GLWindow and instead rely on my other packages that use GLWindow.

This kind of modularity guarantees a broad user and developper base, which in turn results in rich functionality and stability. From further analyzing the github repository written by me, one can find out that there is a lack of documentation. This hinders people from contributing and using the packages, but could not been

### 6.2.1 IJulia

So IJUlia is written in:

using ZMQ(C++), D3, Three.js JavaScript 62.4 HTML 26.4 Python 6.9, C++ 1.9 C 1.3, GLSL 0.6 D3 JavaScript 95.6 CSS 4.3 IPython Python 78.5 JavaScript 15.1 HTML 5.0 Other 1.4

### 6.2.2 Paraview and VTK

This amounts to a total of 3.642.105 lines of code written in 29 languages.

## 6.3 Usability Analysis

# 7 Conclusion

## 7.1 Future Work

# 8  References

[1] Al Danial. Cloc. Accessed: 04/01/2015 : `http://www.webcitation.org/6XT73jFkv`.

[2] IPython. Ijulia notebook. Accessed: 03/23/2015, Archived by WebCite®: `http://www.webcitation.org/6XFFIHQee`.

[3] Viral Shah Alan Edelman Jeff Bezanson, Stefan Karpinski. Why we created julia. Accessed: 04/01/2015 : `http://www.webcitation.org/6XT6wqYAf`.

[4] Kitware. Paraview overview. Accessed: 03/23/2015, Archived by WebCite®: `http://www.webcitation.org/6XFPu1Wz9`.

[5] Julia Lang. Calling c and fortran code. Accessed: 04/02/2015 : `http://www.webcitation.org/6XUueZYLw`.

[6] MathWorks. Matlab pricing. Accessed: 03/22/2015, Archived by WebCite®: `http://www.webcitation.org/6XEFJ0PBE`.

[7] Microsoft. Wglgetprodaddress documentation. Accessed: 02/27/2015, Archived by WebCite®: `http://www.webcitation.org/6WemKehYL`.

[8] Wikipedia. Ipyton. Accessed: 03/23/2015, Archived by WebCite®: `http://www.webcitation.org/6XFEB9BB3`.

# Appendix

## A IJulia

```
In [5]:  # varying the second argument to julia() tiny amounts results in a stunning variety of forms
         @time m = [ uint8(julia(complex(r,i), complex(-.06,.67))) for i=1:-.002:-1, r=-1.5:.002:1.5 ];

         elapsed time: 0.1899382 seconds (1502744 bytes allocated)

In [6]:  # the notebook is able to display ColorMaps
         get_cmap("RdGy")

Out[6]:                                                              RdGy

In [7]:  imshow(m, cmap="RdGy", extent=[-1.5,1.5,-1,1])
```
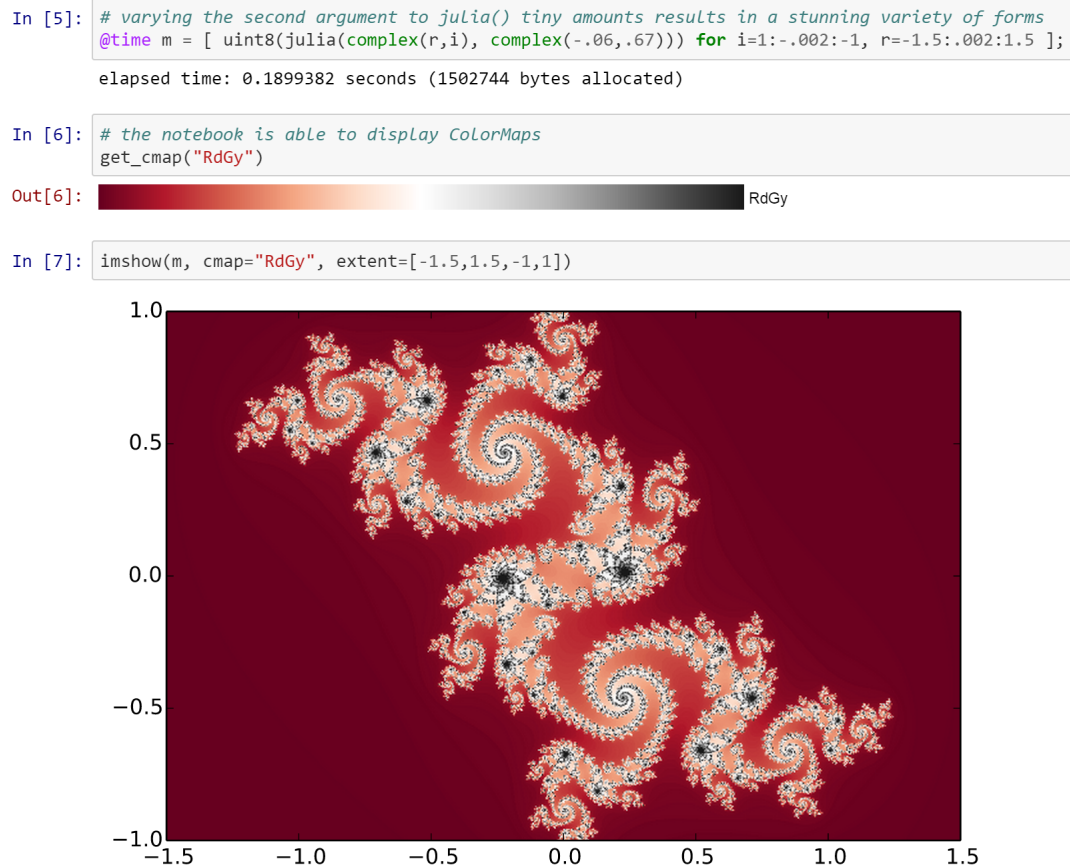
Figure 4: Example of an IJulia Notebooks. Screenshot taken from [2]
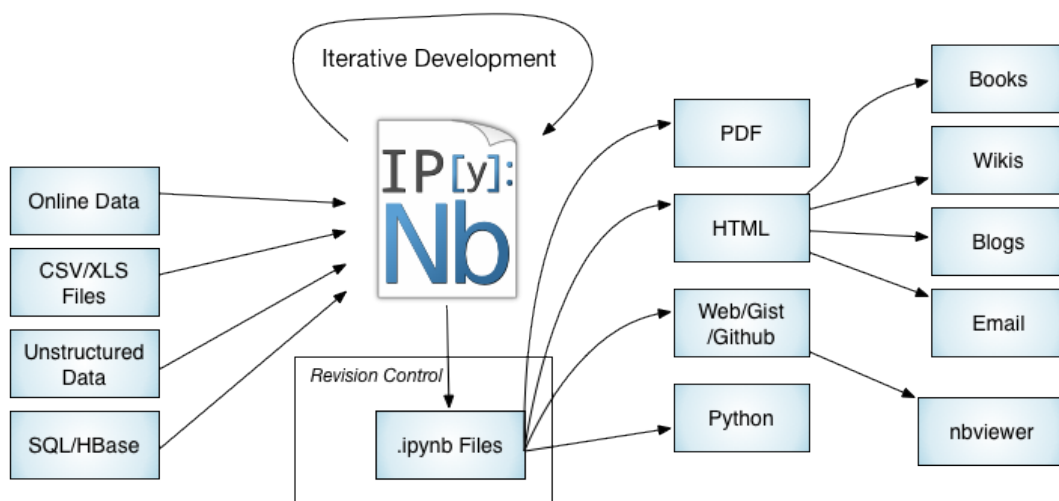
Figure 5: Workflow of IPython Notebooks. Graphic from Wikipedia [7]

# B Language Statistics

All language statistics have been made with cloc [1] the current master of the github repositories.

Table 1: Paraview, language statistic

| Language | files | blank | comments | code |
|---|---|---|---|---|
| C++ | 2037 | 70003 | 86594 | 391121 |
| C/C++ Header | 1937 | 48345 | 93434 | 141581 |
| C | 273 | 35843 | 17101 | 135937 |
| XML | 275 | 1930 | 3521 | 59030 |
| Fortran 77 | 67 | 28 | 18039 | 39116 |
| Python | 209 | 5883 | 8719 | 21935 |
| CMake | 443 | 3705 | 6185 | 20025 |
| Javascript | 20 | 1285 | 1847 | 7982 |
| CSS | 23 | 750 | 251 | 4827 |
| HTML | 26 | 240 | 1692 | 2328 |
| JSON | 13 | 2 | 0 | 2162 |
| yacc | 1 | 207 | 138 | 881 |
| Bourne Again Shell | 19 | 186 | 347 | 799 |
| make | 8 | 248 | 90 | 734 |
| Bourne Shell | 18 | 158 | 116 | 708 |
| XSLT | 3 | 46 | 17 | 388 |
| CUDA | 1 | 58 | 184 | 318 |
| Pascal | 2 | 69 | 102 | 228 |
| SUM: | 5375 | 168986 | 238377 | 830100 |

Table 3: VTK, language statistic

| Language | files | blank | comment | code |
|---|---|---|---|---|
| C++ | 3845 | 203851 | 179827 | 1278279 |
| C | 1103 | 130996 | 289623 | 707122 |
| C/C++ Header | 3489 | 103162 | 246368 | 382728 |
| Python | 1681 | 88983 | 121122 | 258787 |
| Tcl/Tk | 573 | 11052 | 7830 | 48213 |
| CMake | 739 | 4715 | 7424 | 35956 |
| Javascript | 47 | 6941 | 6747 | 33098 |
| CSS | 33 | 1476 | 323 | 18100 |
| XML | 10 | 17 | 36 | 8337 |
| Objective C++ | 20 | 1210 | 1372 | 5601 |
| m4 | 3 | 660 | 83 | 4922 |
| yacc | 3 | 726 | 570 | 4852 |
| HTML | 25 | 553 | 531 | 4313 |
| Java | 50 | 912 | 1192 | 4239 |
| Cython | 20 | 848 | 1625 | 3484 |
| Perl | 11 | 939 | 950 | 3119 |
| JSON | 3 | 5 | 0 | 2658 |
| Windows Resource File | 21 | 333 | 380 | 1835 |
| lex | 3 | 215 | 162 | 1510 |
| DTD | 3 | 435 | 477 | 1335 |
| Assembly | 13 | 202 | 0 | 936 |
| Bourne Again Shell | 16 | 191 | 333 | 866 |
| CUDA | 6 | 113 | 77 | 740 |
| Bourne Shell | 15 | 64 | 122 | 380 |
| make | 5 | 54 | 187 | 170 |
| IDL | 1 | 0 | 0 | 150 |
| Windows Module Definition | 3 | 3 | 0 | 142 |
| JavaServer Faces | 3 | 26 | 0 | 88 |
| Objective C | 2 | 13 | 18 | 17 |
| SUM: | 11749 | 558698 | 867379 | 2812005 |

## C  Romeo's GUI



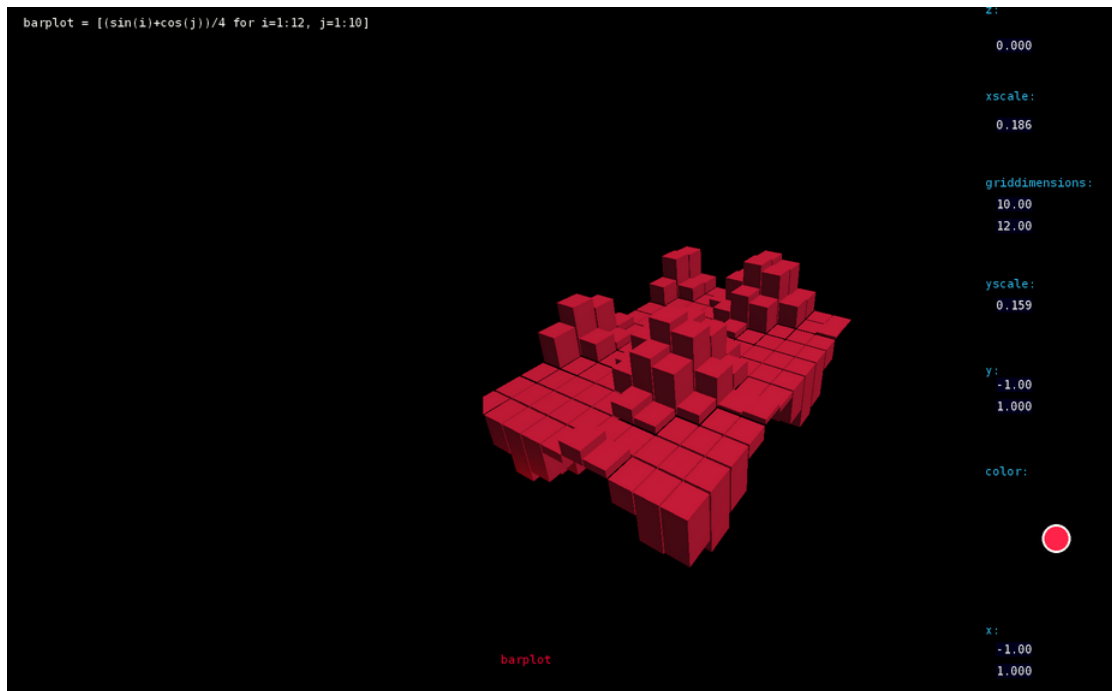Figure 6: Screenshot of the prototype. Left: evaluated script, middle: visualization of the variable barplot, right: GUI for editing the parameters of the visualization

## D  Benchmark

Table 5: FE Implementation comparison

| implementations | Language | Speed in Seconds |
| --- | --- | --- |
| JFinEALE | Julia | 9.6 |
| Comsol 4.4 with PARDISO | Java | 16 |
| Comsol 4.4 with MUMPS | Java | 22 |
| Comsol 4.4 with SPOOLES | Java | 37 |
| FinEALE | Matlab | 810 |

# Official Statement

I hereby guarantee, that I wrote this thesis and didn't use any other sources and utilities than mentioned.

Date:                                    .......................................................
                                              (Signature)