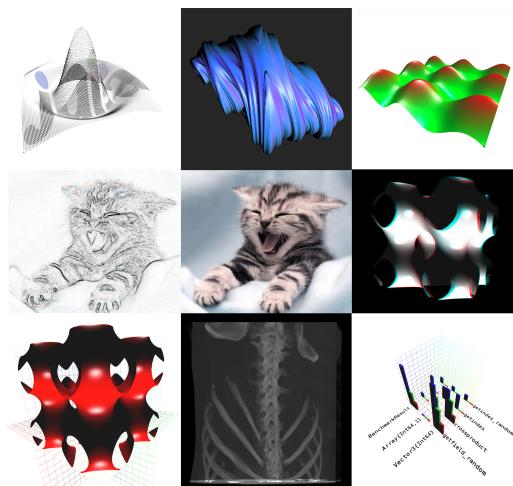




**Faculty of
Cognitive Science**

Bachelor Thesis

Romeo: An Interactive 3D Visualization Library for Julia



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

I Abstract	I
II Table of Contents	II
III List of Figures	IV
IV List of Tables	V
V Listing-Verzeichnis	V
VI List of Abbreviations	VI
1 Introduction	1
1.1 Scientific Computing	1
1.2 Field of Research and Problem	2
1.3 Contribution	4
1.4 Outlook	4
2 Background	5
2.1 Related Work	5
2.1.1 The Julia Programming Language	5
2.1.2 Low Level Virtual Machine (LLVM)	6
2.1.3 IJulia	6
2.1.4 Matlab	7
2.1.5 Mayavi and VTK	8
2.1.6 Vispy	9
3 Design	10
3.0.7 Speed	10
3.0.8 Extensibility	11
3.0.9 Event System	12
3.0.10 Interfaces	12
4 Used Technologies	13
4.1 The Julia Programming Language	13
4.2 Open Graphics Language (OpenGL)	15
4.3 Reactive	16
4.4 GLFW	17
5 Implementation	18
5.1 Event System	18
5.2 ModernGL	19
5.3 GLAbstraction	19
5.4 GLWindow	19
5.5 GLVisualize	20
5.5.1 Romeo	21

6 Results and Discussion	22
6.1 Performance Analysis	22
6.1.1 Julia	22
6.1.2 ModernGL	25
6.1.3 Reactive	26
6.1.4 3D Rendering Benchmark	27
6.1.5 IJulia	29
6.2 Extensibility Analysis	29
6.2.1 IJulia	30
6.2.2 Mayavi and VTK	31
6.2.3 Matlab	31
6.3 Usability Analysis	31
7 Conclusion	33
7.1 Future Work	33
8 References	34
Appendix	II
A IJulia	II
B Language Statistics	III
C Romeo's GUI	V
D Benchmark	V

III List of Figures

Abb. 1	Volume Visualization	2
Abb. 2	VTK Capabilities	8
Abb. 3	Volume Visualization	10
Abb. 4	OpenGL	15
Abb. 5	Architecture	18
Abb. 6	Visualisations	20
Abb. 7	Julia Performance	22
Abb. 8	OpenGL Wrapper	25
Abb. 9	Reactive 1	26
Abb. 10	Reactive 2	26
Abb. 11	Benchmark	27
Abb. 12	Particles	28
Abb. 13	Sierpinsky	28
Abb. 14	IJulia Notebook Example	II
Abb. 15	IPython Notebook Workflow	II
Abb. 16	Prototype	V

IV List of Tables

Tab. 1	FEM Benchmark	23
Tab. 2	gcc vs llvm summary	24
Tab. 3	OGL Relative Speed	25
Tab. 4	3D Benchmark	27
Tab. 5	3D Benchmark	28
Tab. 6	3D Benchmark	29
Tab. 7	IJulia Stack	31
Tab. 8	Paraview, language statistic	III
Tab. 10	VTK, language statistic	IV
Tab. 12	FE Comparison	V
Tab. 14	LLVM 3.5 compared to LLVM 3.6	VI
Tab. 16	GNU Compiler Collection (gcc) 4.9.2 compared to LLVM 3.5	VII

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
OpenCL	Open Compute Language
OpenGL	Open Graphics Language
VTK	Visualization Toolkit
AST	Abstract Syntax Tree
CUDA	Compute Unified Device Architecture

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. Using these widgets, all bound variables can be visualized and some of them can be edited interactively. This can be used as a basis for 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

Scientific computing is the area of computing that evolves around all kind of scientific research. It is a very broad field involving a lot of different challenges. In some areas like particle physics, the problems are computationally so demanding, that they can only be solved with the help of super computers. In other areas like robotics, it is important to be efficient, because the algorithms are running on embedded systems with limited resources. In a lot of other areas, speed does not need to be important, but it can be that the algorithm in itself is very difficult to comprehend. So the more comprehensible an algorithm can be written down in a programming language, the easier it will be to implement the algorithm without errors. Above all, programming itself is secondary to the research goal. This means that it can be expected that a researcher just has rudimentary programming skills. Even if he is a professional programmer, he wants to put as little time as possible into solving pure programming problems.

So things like manual memory management and difficult design patterns with a lot of boilerplates are to be avoided in scientific computing. This has lead to the rise of programming languages and tools specifically tailored to scientific computing. The most prominent examples include Mathematica, R, and Matlab. Python could be in this list as well, but the scientific computing part is only realized by third party libraries, while Python itself is a multi purpose language. The others all aim to provide simple syntax for linear algebra and statistical code, while taking away programmatically difficult tasks like memory management. Also, they come with a rich standard library, which means most

research can be done without loading any additional module, which makes them great tools for rapid prototyping. At the current state, the speed of these languages suffer from the high level of abstraction. In order to cater to the fields of scientific computing which is in need of highly performant code, a lot of the core is written in another language like C/C++ and Fortran. This poses a problem in itself. As soon as a researcher needs to do something out of the ordinary in a performant way, he needs to switch to a fast multi purpose language. So in the end he is loosing all the advantages of the high level scientific computing language. A pattern which has evolved out of this dilemma is to prototype in a nice high level language and as soon as the algorithms has been confirmed to work, to rewrite it in a fast low level language. One of the first language promising to solve this dilemma for scientific computing is Julia. It is supposed to be high level and optimized for the work of scientific computing while approaching the speed of statically compiled languages like C. This leads us straight to the field of research and problem that gets solved in this thesis.

1.2 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*

This thesis is about bringing performance and usability together in the realms of scientific computing and 3D visualizations. These two demands are pretty much opposing concepts. One is about bringing tasks into a form of making them best understandable to humans, and the other is about transforming a task to make it fit well to a computer architecture. These two tasks could not be more different. For humans, data and algorithms becomes understandable if they are high level and represented visually, auditorial or tactile together with immidiate feedback. It is the task of making problems accessible to a human, who has evolved his capabilities in order to survive and find food and not to create complex algorithms. Computers on the other hand love to have their registers filled optimally, move memory to smaller and faster caches and dislike random access to memory. That is

all they care about, whether a human understands this or not.

To close this gaps, compiler have been created. They are translators between human understandable languages to machine instructions. This is just the first step and many more are needed to create an enjoyable user experience. These steps range from introducing graphical user interfaces, novel input devices like the mouse, understandable visualizations and so forth. All these advances have made computers usable for people who do not have an education in computer science. In this 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. Most research is done in some specialized, high-level scientific computing language. Besides the previously discussed performance problems with this approach, the lack of easily usable, extendable and fast visualization libraries also poses a problem. Most state of the art visualization libraries use C++ at their performance critical core, they are not extendable or they are simple toy libraries, which can not be used for projects with higher performance demands.

This is a problem for several reasons. First, it creates a complexity and performance bottlenecks when interfacing with other languages. The next problem occurs, when the library does not offer the needed functionality and the programmer has to step in and extend the library. Finally, you often do not have easily accessable GUI elements, they come from a different package (possibly written in yet another language) or they are complicated to use. This makes it hard for the researcher to visualize and interact with his data. It would be desirable, if interactability, speed and extendability would be the default for any visualization library.

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 . If you can interact with this visualization, by moving through the iso values or coloring certain areas, it will make the function more understandable. This deeper understanding is crucial for identifying problems in the underlying math, extending the function, or explaining it to other people. Making problems more understandable like this further opens the gates of scientific computing to novel users.

In summary, the software in this thesis 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. There are quite a few libraries out there offering this, but none of them is written in a high level language, offers speed, extendability and usability, while being deeply integrated into a scientific programming language.

1.3 Contribution

The main contribution of this thesis is writing Romeo in Julia, which offers the following advantages.

Julia is a high-level language and some effort was also put into creating a concise architecture, so one contribution is, that the development cycles can be very short and the library is easy to extend.

The target group for Julia are researchers, which should be able to write their research completely in Julia. As Julia is fast and the library is also written in Julia, this will enable researchers to stay in the same language for their project. Due to Julia's speed, this makes it easy to create visualisation pipelines in which every routine is as fast as it can be. Also, the researcher can extend the library in the same language he is already working in.

On top of that, the library makes it simple to interact with complex algorithms via widgets and forms a basis for visual debugging. This comes with an ease of use, which would be hard to achieve if the library was not that deeply embedded in Julia.

1.4 Outlook

[short outlook, includes BA structure and some words about the results]

2 Background

In this chapter, a short overview will be given over the current state of the art for visualization inside the field of scientific computing and a short intro to Julia will be given.

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. So Julia plays a crucial role in this thesis. It is the most important previous work, as much as Julia is the main used technology. This chapter gives a short introduction to the Julia Programming Language.

Julia was published in 2012, which makes it a very new language. It is currently at version 0.3.7 stable and 0.4 pre-release. Following common versioning conventions this means Julia is still in an early release phase with the core features and names suspicible to change. Julia is using the compiler infrastructure LLVM to generate fast assembly code.

Julia is a multi paradigm language for scientific computing. Some of its most important features are multiple dispatch, a dynamic type system, macros, good performance and an interface to C and Python. 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 like linear algebra and statistics. It promises to approach C speed, while being a dynamic language which is easy to use. This is made possible by the compile process which can be described as statically compiled at runtime. Julia uses a garbage collector, taking the task of memory management away from the programmer. There are quite a few things Julia promises to the developer which includes the following items[8]:

- C like performance
- native C interface
- macros like in Lisp
- mathematical notations like Matlab
- good at general purpose programming as Python
- easy for statistics as R.

Another interesting feature of Julia is, that all user defined types are as fast and compact as build in types. This allowes Julia to write all arithmetic types in Julia itself, allowing

anyone to extend and rewrite them without diving into the compiler. This means that you can write your own floating point type, with the same performance and characteristics as the build in floating point type.

This makes Julia a very desirable scientific computing language, which promises to be also great for a visualization library. As part of this thesis, it will be investigated if Julia's claims have been achieved.

2.1.2 LLVM

LLVM is an compiler infrastructure, which has front ends for different languages and compiles to different platforms like x86, ARM, Open Compute Language (OpenCL) and Compute Unified Device Architecture (CUDA). A language designer must create an Abstract Syntax Tree (AST) which than LLVM can convert into LLVM Intermediate Representation (IR). This IR can than be heavily optimized. Every language that can be converted to LLVM IR can be combined at this level, going through the same optimizations in the end. This yields superiour language interfacing, as inlining and other optimizations can be done over the boundary of one language.

LLVM's concept is effective, as you can accumulate state of the art optimizations in one place, making them accessible to many languages. Because of the many backends, the language can run on many architectures. While Julia does not 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 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.

2.1.3 IJulia

IJulia is the Julia language back-end 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 snippets. You can also formulate mathematical formulas in latex, which will get rendered and inlined nicely into an IJulia Notebook. See figure 14 for an example.

IJulia has some similar goals 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 powerful 3D visualization library based on WebGL. The current integration is just prototypical and limited to simple 3D meshes up to now.

2.1.4 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++, Java Fortran and Python.

Matlab itself is written in C, C++, Java and MATLAB. It's proprietary software with a pricing of around 2000 €[15], 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.

2.1.5 Mayavi and VTK

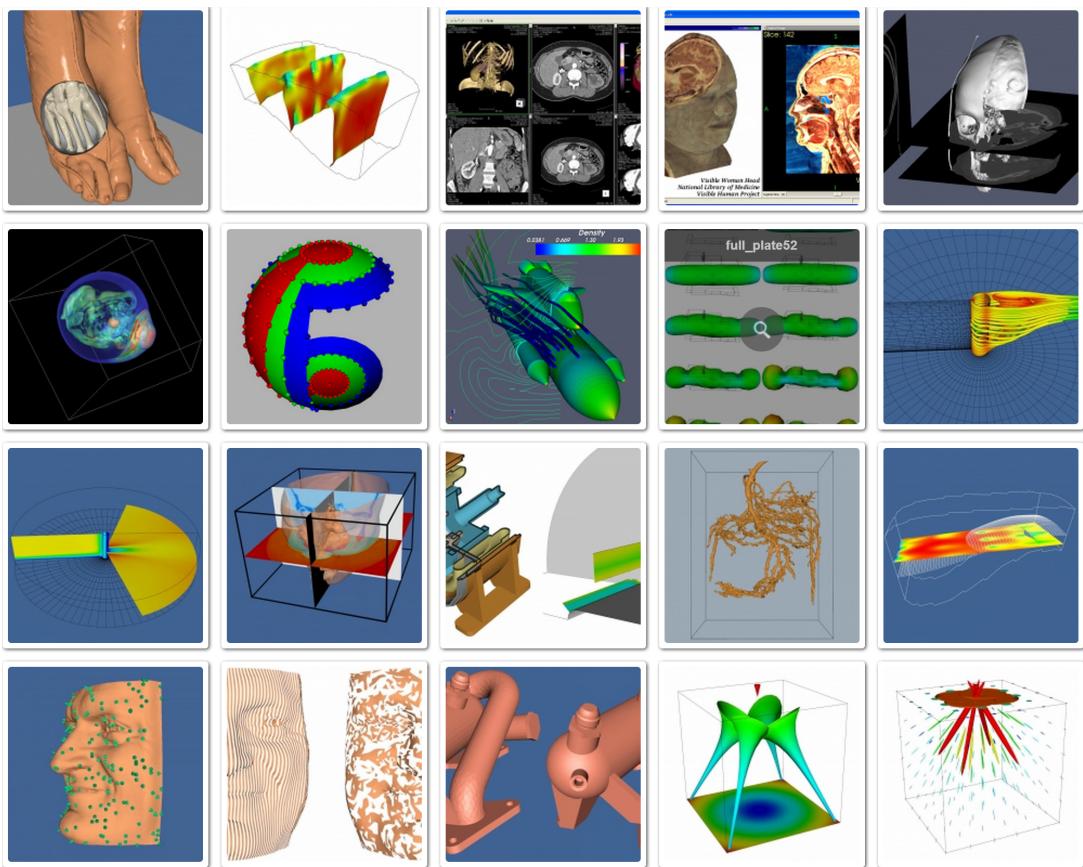


Figure 2: *Different visualizations done with VTK.*

Mayavi is probably one of the biggest, open source library for interactive 3D visualizations. It is written 99.9% in python, but relies on Visualization Toolkit (VTK) for rendering. VTK is one of the most advanced scientific visualization library, with a huge amount of visualization types. In figure 2 you can see some of the visualization taken from the VTK gallery[9].

Mayavi shares some of its goals with Romeo, namely[3]

- An (optional) rich user interface with dialogs to interact with all data and objects in the visualization.
- A simple and clean scripting interface in Python, including one-liners, or an object-oriented programming interface. Mayavi integrates seamlessly with numpy and scipy for 3D plotting and can even be used in IPython interactively, similarly to Matplotlib.
- The power of the VTK toolkit, harnessed through these interfaces, without forcing you to learn it.

Obviously, the python part is not a shared goal, but creating an interactive 3D visualization library deeply embedded into a language is. Mayavi together with VTK is a very big project and in this sense not really comparable to Romeo. It amounts to a total of 3.642.105 lines of code written in 29 languages. The statistics can be found in table 8 and 10. The biggest difference is, that Romeo is implemented in a scientific programming language, while Mayavis core uses VTK which is mainly implemented in C++. This has two big implications. Firstly, if the language does not have native C++ compatible data types and an overhead less C++ interface, shipping a large stream of data to VTK becomes slow. Secondly, one must know C++ to extend VTK. This makes it difficult to create customized visualizations.

In contrast, Romeo is implemented in one language, making these tasks very simple and efficient.

2.1.6 Vispy

Vispy is yet another interactive 3D visualization library. It is from the goals and development status the closest to Romeo. These include[4]:

- High-quality interactive scientific plots with millions of points.
- Direct visualization of real-time data.
- Fast interactive visualization of 3D models (meshes, volume rendering).
- OpenGL visualization demos.
- Scientific GUIs with fast, scalable visualization widgets (Qt or IPython notebook with WebGL).

It is a fairly new library, promising to use modern OpenGL and state of the art performance. This is very similar to Romeo's goals, with the only difference being that Romeo is implemented in Julia while Vispy is implemented in Python. So the biggest differentiation between Romeo and Vispy will be the performance and the concrete feature set.

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.

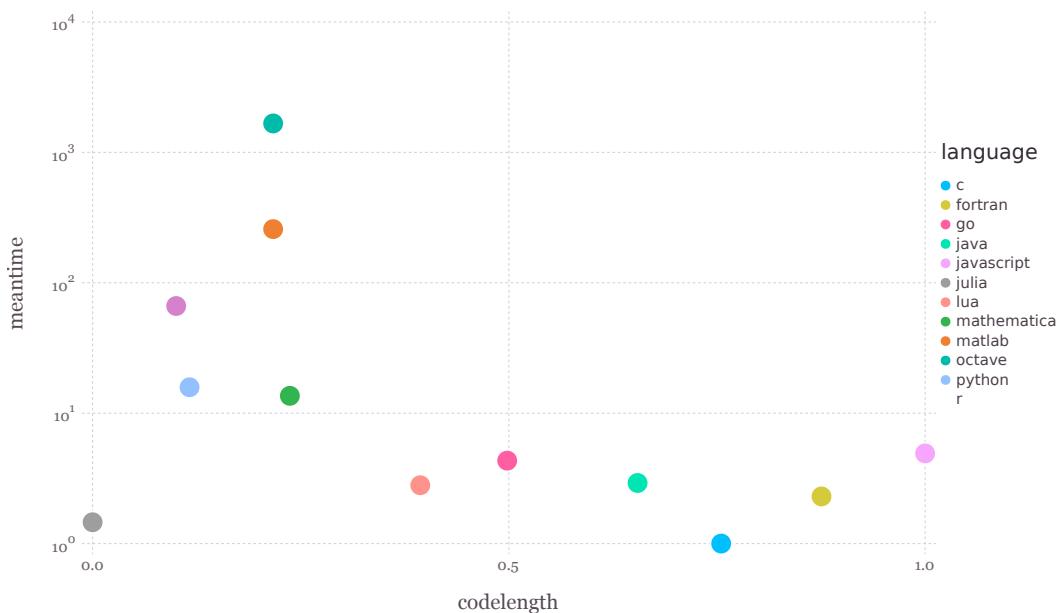


Figure 3: *Languages speed relative to C (averaged benchmark results), plotted against the length of the needed code (Source in Appendix).*

3.0.7 Speed

Speed is mainly a usability factor. It is 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 not be satisfied at the same time. 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. Julia promises to do exactly this, which is illustrated in figure 7. 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 are needed to achieve the same goal. From this figure we can conclude, that Julia at least comes close to its promises, which is why it has been chosen as the programming language.

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. 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 7, 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 were no 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 Khronos'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 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 is easy to benchmark the code, but quite difficult to find a baseline, as one either has to implement the whole software with an 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.

3.0.8 Extensibility

Extensibility is an important factor, which can decide, if a library is fit for scientific computing or not. It is not only that, but also a great factor determining growth of a software, as the more extensible the software is, the higher is 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 is not only important, that all parts are understandable, but also, that every independent unit in the code solves only one problem. This guarantees that one can quickly exchange it, or use it somewhere else where the same problem needs to be solved. 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 spent on understanding and rewriting the code.

So the code written for this thesis will 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 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 the strategy, usage patterns and feedback from Github will be analyzed.

3.0.9 Event System

The event system is a crucial part of the library, as the proclaimed goal is to visualize dynamic, animated data. This means, there are hard demands for usability and speed on the event system. The chosen event system has an immediate influence on how to handle animations. This leads to the design choice of using signals. Signals are a very good abstraction for values that change over time. If well implemented, it makes it natural to reason about time, without the need of managing unrelated structures and callback code.

3.0.10 Interfaces

Working with a computer means working with interfaces to a computer, which in the end simply juggles 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 underlying library. This guarantees re-usability for others, as they might be used to work only with the low-level library or they might disagree with some higher-level abstraction and prefer to write their own.

Over this sits an abstraction layer needed to simplify the work with OpenGL. With this abstraction, the actual visualization library is implemented.

API! (**API!**)s 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 create different types for the same data, without loosing performance. So you can have a unit like meters 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 create a default visualization for different data types. Instead of manually passing additional information to the visualization function, it is coded in the type itself. 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 optional key word arguments. 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 Used Technologies

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 its hierarchical type system and multiple dispatch. Multiple dispatch is in its core function overloading at runtime. To better understand multiple dispatch, one has to be familiar with Julia's type system. The type system builds upon four basic components. Composite types, which are comparable to C-Structs, parametric composite types, bits types, abstract and parametric abstract types. While the first three are all concrete types, abstract types can not be instanciated but are used to build a type hierarchy. Every concrete type can only inherit from one abstract type, while abstract types can also inherit from abstract types. Bit types are just immutable, stack allocated memory chunks, usable for implementing numbers. 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}(y::Complex{T}, y::Float32) = println("some number: ", x, " some
      complex Number: ", y) # shorthand function definition
2 function foo(x)
3     println("overloading foo with a new unspecific signature")
4 end

```

What will happen at runtime is, that Julia compiles a method specialized on the arguments which results in overloading the function with the concretely typed arguments. To illustrate this let us look at the example. Initially, `foo` will be overloaded with two methods. Now, if you call `foo` with one `Float32` argument, a new method will be added at run time specialized to `Float32`. Like this, if the function does not access non constant global values, all types inside the function will be known at call time. This allows Julia to statically compile the function body, while propagating the type information down the call-tree.

With multiple dispatch, Julia is definitely a functional oriented language. But there are also ways to give Julia a more object oriented feel. Functions are first-class, so they are easy to pass around. They can be bound to variables and can then be called like normal functions via the variable name. This implies that functions can also be bound to objects. There is no self reference available in Julia, so the object still needs to be passed to function via the function arguments

Another of the most crucial features is the very simple, overhead less C-Interface. Thanks to the binary compatibility of LLVM's emitted assembly, a C function call to a shared library inside Julia has the same overhead as it would be from inside C[10]. This is perfect for dealing with low-level libraries like OpenGL and OpenCL.

4.2 OpenGL

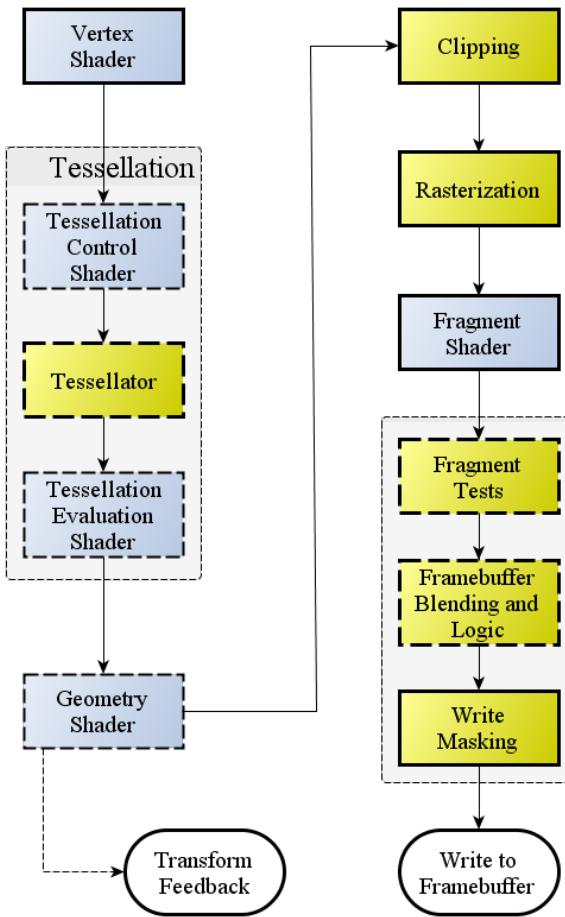


Figure 4: Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages. [19]

OpenGL is a low-level graphics API implemented by the video card vendor via the video driver. As such it does not 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. A critical element when developing OpenGL applications is, that not all video drivers implement the newest OpenGL standards.

As a result, one has to decide which OpenGL version to program against, trading 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.

In figure 4 you can see the basic architecture of an OpenGL program pipeline. As the description states, the blue boxes are programmable shaders, while the dotted boxes are

optional parts of the pipeline. The yellow boxes describe stages which are not directly accessible. They are part of the global OpenGL state, which can be set via OpenGL commands.

So in order to have a functioning OpenGL rendering pipeline one just needs to write a vertex shader and a fragment shader. All shaders are compiled and linked into a program object, which then can be executed on the GPU. Shaders are written in a C dialect specialized for vector operations. You feed shaders with data via buffers, textures and uniforms. Buffers are 1D arrays, textures 1D/2D/3D arrays with both having their own memory, while uniforms live in the program object.

The different shaders are used to apply geometric, perspective transforms and calculating the light. In newer APIs general compute operations are available, making it possible to create more flexible shaders. Finally, the fragment shader rasterizes the data to the screen. Here is a simple minimal example for a program rendering some vertex data with a flat color to the screen.

```

1 //Vertex Shader
2 in vec3 vertex; // vertex fed into the shader via a buffer
3 uniform mat4 projection; // Projection matrix
4 uniform mat4 view; // View matrix, setting rotation and translation of
                  the camera
5 void main()
6 {
7     gl_position = projection*view*vec4(vertex, 1); // apply
          transformations to vertex and output to fragment shader
8 }
9 //Fragment shader
10 out vec4 framebuffer_color; //output from fragment shader, which will
          get written into the display framebuffer
11 void main()
12 {
13     framebuffer_color = vec4(1,0,0,1); // write a red pixel at
          gl_position from the vertex shader.
14 }
```

4.3 Reactive

Reactive is a functional event system designed for event driven programming. It implements Elm's signal based event system in Julia. Signals can be transformed via arbitrary functions which in turn create a new signal. This simple principle leads to 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(ln 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

```

Lifting a signal creates a callback, which gets called whenever the signal changes. There are more operations than lifting, like folding, merging, filtering and so on. With this, one can build up a complex tree of operators which will get applied to the origin signal. For the concrete case of Reactive, every signal carries around a list of children and parents. Each signal has a rank, in order to build up a sorted heap with these information. So every time a signal is updated, the heap can be traversed and the functions get applied in the right order, updating all the values of the children.

4.4 GLFW

GLFW is a cross platform OpenGL context and window creation library written in C. GLFW allows to register callbacks for a multitude of events like keyboard, mouse and window events. This, together with a wrapper library for Julia makes GLFW perfect for doing the window creation. In addition, GLFW exposes low level features like the operating systems context handle. This can be used for creating advanced contexts that share memory with another context. Romeo does not use this feature yet, but it makes GLFW a future proof choice.

5 Implementation

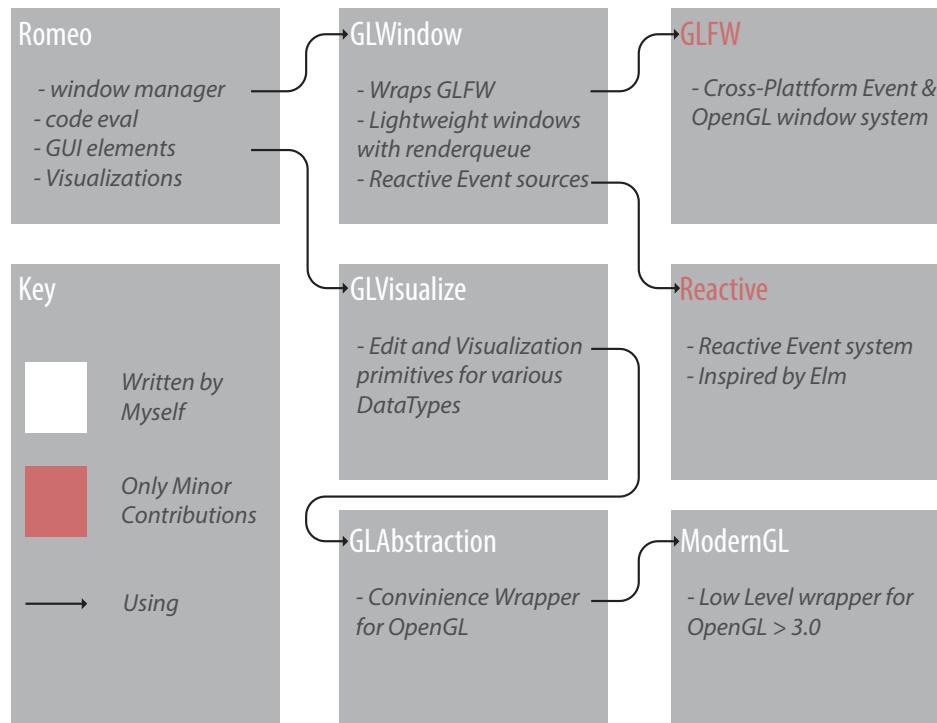


Figure 5: Main modules used in Romeo and their relation (simplified).

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 Event System

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 does not allow to rearrange the event tree. In other words, you can not create sub trees in advance and then fuse them together at run time.

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 pointers system and vendor independent. 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. [16] 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.

The OpenGL function loader from ModernGL has undergone some changes over the time. Starting with a very simple solution, there have been pull requests to include better methods for the function loading. The current approach in ModernGL master was not written by myself, but by the Github user aaalexandrov. Before aaalexandrov's approach, the fastest approach would have used a pretty new Julia feature, named staged functions. It should in principle yield the best performance as it compiles a specialized version of the function when it gets called for the first time. This is perfect for OpenGL function loading, as the pointer to the function can only be queried after an OpenGL context has been created. When the staged function gets called the pointer can be queried and gets inlined into the just in time compiled function.

Staged functions only work with the newest Julia build, which is why aaalexandrov's approach was used in the end.

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 supplying an abstraction layer over OpenGL, it also offers the linear algebra needed for the various 3D transformation and camera code. Building up on that, it defines a signal based perspective and orthographic camera type.

5.4 GLWindow

GLWindow is a lightweight wrapper around GLFW. It mainly offers a screen type, which contains signals for all the different GLFW events. It also offers a hierarchical structure for nesting screens. All the screen areas are signals, which makes it easy to change the screen area. This makes it simple to implement windows that react to changing the size of the windows or resized objects.

5.5 GLVisualize

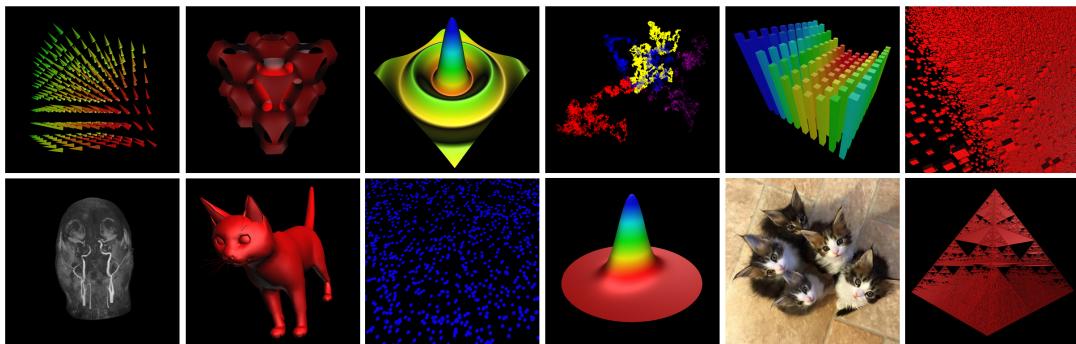


Figure 6: *Different visualizations rendered with GLVisualize.*

GLVisualize implements the main functionality of this library. Its structure is quite simple. It relies as much as it can on common Julia data types and creates specialized visualizations for them via dispatch. So instead of offering differently named functions for different visualizations, there is just one function with lots of methods for different types. This has two advantages. First, it makes it very easy to use for visual debugging, as any value can be displayed immediately without any user interaction. Secondly, the user does not have to remember or lookup the function name, as long as there is a default visualization for the type he is working with. The next design goal was to make this fit for dynamic data, which resulted in relying on as little transformation of the data as possible and directly transferring it to the GPU. Depending on the complexity of the visualization, this means the visualization can be updated with as little overhead as possible.

The interface to create visualizations is very simple and only consists of three functions:

```

1 Dict{Symbol, Any}      = visualization_defaults(data::Union(Signal{T}, T),
                                                 style::Style) # returns a dictionary of parameters
2 RenderObject          = visualize(data::Union(Signal{T}, T), style=Style{:
                           default}; parameters...) # returns an object which can be directly
                           rendered
3 RenderObject, Signal = edit(data::Union(Signal{T}, T), style=Style{:
                           default}; parameters...) # returns an RenderObject and signal which
                           outputs the changed values

```

With this simple interface, 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)

All of these can be integrated into the same scene and it is possible to change their parameters interactively. These interactions can be purely programmatically, or via the widgets from the edit function. 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 then updates the text data on the GPU according to the cursor position and keyboard input.

Up to now, there is an edit function available for strings, 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

This chapter supplies some benchmarks, to analyze how close this thesis comes to achieve the wanted performance, which should be on eye level with C.

If not stated otherwise, benchmarks are written for this thesis and executed on an Intel Core i5-4200U with an HD 4400 graphic and 8GB of RAM. Julia 0.4 has been used, C++ code has been compiled with VS13 and for python the anaconda distribution with Python 2.7 was used. Benchmarks were run on an idle computer with as little background processes running as possible. The sources of the benchmarks can be found on Github.

6.1.1 Julia

Julia's performance is crucial for this thesis. If Julia does not perform close to C it would weaken the whole argument of writing the visualization library in Julia. It is a very tedious task to write representative benchmarks for a programming language. The only way out is to rely on a multitude of sources and try to find analytical arguments. In this thesis, Julia's own benchmark suite will be used in addition to some real world benchmarks. In addition, the general compiler structure of Julia will be analyzed to find indicators for the limits of Julias performance.

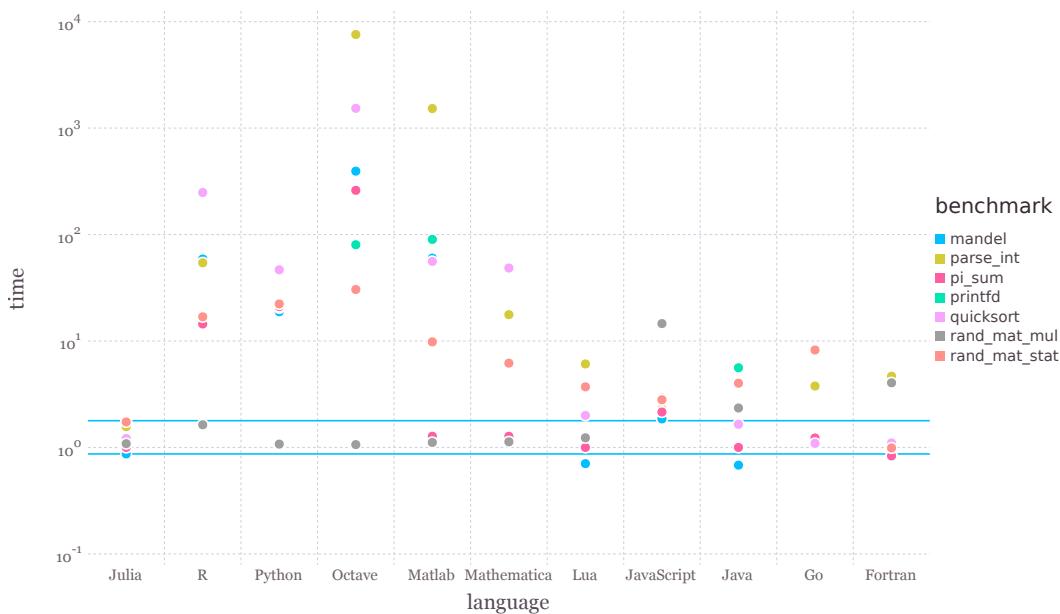


Figure 7: *Julia's performance compared to other languages, taken from Julia's micro bench suite [11]. Smaller is better, C performance = 1.0.*

In the first benchmark from figure 7, we can see that Julia stays well within the range of C Speed. Actually, it even comes second to C-speed with no other language being that close. This is a very promising first look at Julia, but it should be noted, that these benchmarks are written by the Julia core team. So it is not guaranteed, that there is no bias favoring Julia in these benchmarks. There is another benchmark comparing C++, Julia and F#, which was created by Palladium Consulting which should not have any interest in favoring one of the languages. They compare the performance of C++, Julia and F# for an IBM/370 floating point to IEEE floating point conversion algorithm. This is part of a blog series[6] written by Palladium Consulting. F# comes out last with 748.275 ms, than Julia with 483.769 ms and finally C++ with 463.474 ms. At the citation time, the Author had updated the C++ version to achieve 388.668 ms. It looks like the author was only working on making the C++ version faster, so it can not be said that the other versions could not have been made faster too.

The last Julia benchmark is more real world oriented. It is comparing Finite Element solver, which is an often used algorithm in material research and therefore represents a relevant use case for Julia.

N	Julia	FEniCS(Python + C++)	FreeFem++(C++)
121	0.99	0.67	0.01
2601	1.07	0.76	0.05
10201	1.37	1.00	0.23
40401	2.63	2.09	1.05
123201	6.29	5.88	4.03
251001	12.28	12.16	9.09

Table 1: *Performance of a FEM solver written in Julia compared to some existing libraries.* [18]

These are remarkable results, considering that the author states it was not a big effort to achieve this. After all, the other libraries are established FEM solvers written in C++, which should not be easy to compete with.

This list could go on, but it is more constructive to find out Julia's limits analytically. As already mentioned, Julia is statically compiled at runtime. This means, as long as all types can be inferred at runtime, Julia will have in the most cases identical performance to C++. The biggest remaining difference in this case is the garbage collection. Julia 0.3 has a mark and sweep garbage collector, while Julia 0.4 has an iterative mark and sweep garbage collector. As seen in the benchmarks, it does not necessarily introduce big slowdowns. But there are issues, where garbage collection introduces a significant slow down[17]. Analysing this further is not in the scope of this thesis, though. But it can be said that Julia's garbage collector is very young and only the future will show how big

the actual differences will be.

Another big difference is the difference in between different compiler technologies. LLVM's biggest rival is gcc. If C++ code that is compiled with gcc is much faster than the same code compiled with LLVM, the gcc version will also be faster as a comparable Julia program. In order to investigate the impact of this, one last benchmark will be analyzed. This is a summary of a series of articles posted on Phoronix, which benchmarked gcc 4.92 against LLVM 3.5 and LLVM 3.5 against LLVM 3.6:

Method	gcc vs LLVM 3.5	LLVM 3.5 vs LLVM 3.6
mean	0.99	0.99
median	0.97	1.00
maximum	1.48	1.10
minimum	0.39	0.88

Table 2: *Summary of the Phoronix benchmark. Unit is speedup of LLVM, bigger is better.* [1]/[12]/[13]

The full tables can be found in the appendix under the table 14 and 16. The results suggest, that LLVM is well in the range of gcc, even though that there can be big differences between the two. These are promising results, especially if you consider that LLVM is much younger than gcc. With big companies like Apple, Google[21] and Microsoft[14] being invested in LLVM, it is to be expected that LLVM will stay competitive, which means Julia should in theory stay competitive as well.

6.1.2 ModernGL

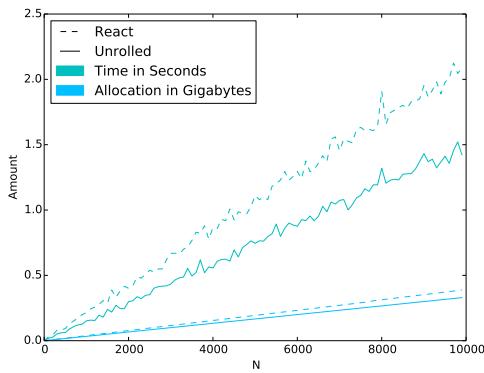
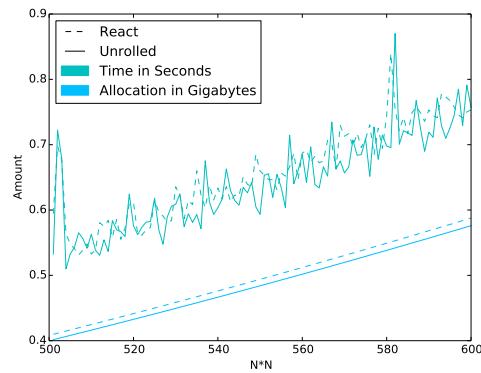


Figure 8: Different performance of OpenGL wrappers. The time for 10^7 calls was measured 100 times for each function.

Function	Python	Staged Function	ModernGL
glBindBuffer	64.43	1.00	1.04
glClearColor	474.72	1.02	1.04
glStringi	244.44	1.07	1.1

Table 3: Performance relative to C++ with Glew (slowdown, bigger is worse)

In this chapter, ModernGL, GLEW and PyOpenGL will get benchmarked. The procedure was, to call an OpenGL function 10^7 times in a tight loop. Execution time of the loop gets measured. The results are plotted in figure 8. ModernGL seems to do pretty well compared to C++ and python does very badly, with being up to 470 times slower in the case of glClearColor. Julia in contrast offers nearly the same speed as calling OpenGL functions from C++ as can be seen in the table 6. As all the OpenGL wrappers are pretty mature by now and bind to the same C library (the video driver), this should mainly be a C function call benchmark. Python performs badly here, but it must be noted that there are a lot of different Python distributions and some promise to have better C interoperability. As this benchmarks goal is to show that Julia's ccall interface is comparable to a C function call from inside C++, the python options have not been

Figure 9: *Complicated graph, simple calculation*Figure 10: *High memory, simple event graph*

researched that thoroughly. From this benchmark can be concluded, that Julia offers a solid basis for an OpenGL wrapper library.

6.1.3 Reactive

It is relatively hard to benchmark the used event system in real world scenarios as it is hard to find a baseline. One would have to rewrite Romeo with another Event system. Using other visualization libraries as a baseline is also difficult, as it is hard to isolate the performance of the event system. This is why we will compare an event graph from Reactive with its unrolled version. For the unrolled version the functions from the callback graph have been executed in the same order as the event graph would have without introducing any event system related overhead. This way we can measure the overhead introduced by the event system. Two code samples have been benchmarked, one simulating the operation needed for the camera and the other simulates animating a large array. The first has low memory usage with a more complex event graph. The second has a straight forward event graph, but it must pass on a large array and needs to execute the callbacks on the array.

As can be seen in figure 13, small operations with a complex event graph has some noticeable overhead. Reactive is in this scenario about 1.45 times slower than the optimal version. This does not come as a surprise as sorting and managing the graph structure obviously adds some overhead.

The second scenario looks much better for Reactive. The performance difference is neglectable, making Reactive a good choice for creating signals with high memory throughput.

6.1.4 3D Rendering Benchmark

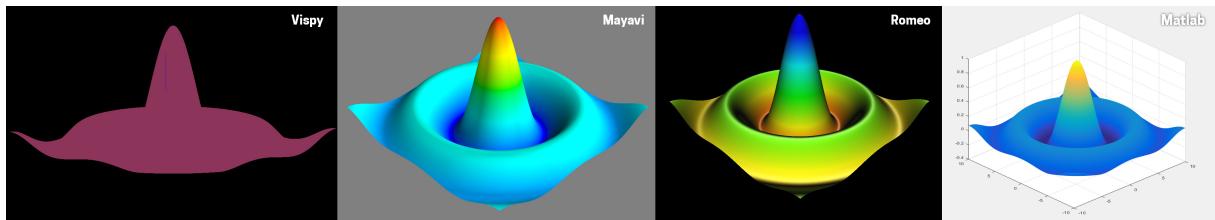


Figure 11: *Different visualizations of the same surface*

The biggest problem with benchmarking the 3D rendering speed is, that there is no library which will allow to exactly reproduce similar conditions and measures. Additionally, without extensive knowledge of the library, it is difficult to foresee what gets benchmarked. As an example of why it is difficult to measure the frame rate we can look at Vispy. When you enable to measure the frame rate, it will show very low frame rates, as it only creates a new frame on demand. On the other side Romeo has a fixed render loop, which renders as much frames as possible, leading to totally different amount of rendered frames per second. This is why it was decided, to use the threshold at which a similar 3D scene is still conceived as enjoyable and interactive. Usually the minimal amount of frames per second for perceiving movements as smooth is around 25. So the benchmark was executed in the way, that the number regulating the complexity of the 3D scene was increased until one could not move the camera without stutters. The recorded threshold is than the result of the Benchmark.

First benchmark is an animated and still 3D surface plot. The libraries offering this functionality where Vispy, Mayavi and Matlab.

Library	Still	Animated
Vispy	300	80
Mayavi	800	150
Matlab	800	450
Romeo	900	600
Speed up Vispy	9x	56x
Speed up Mayavi	1.26x	16x
Speed up Matlab	1.26x	1.7x

Table 4: *3D surface created from a NxN matrix.*

Vispy had some issues, as the camera was never really smooth for the surface example. Also the normals were missing and there was no option to colorize the surface depending on the height. It was decided to use the threshold of going from a little stutter to unpleasant stutters, making vispy not completely fail this benchmark. For Vispy it was found out, that the normals where calculated on the CPU resulting in a major slow down[5]. The

same can be expected for Mayavi, but Mayavi seems to be faster at calculating the normals. There is not much information available on how Matlab renders their visualization, as it is closed source.

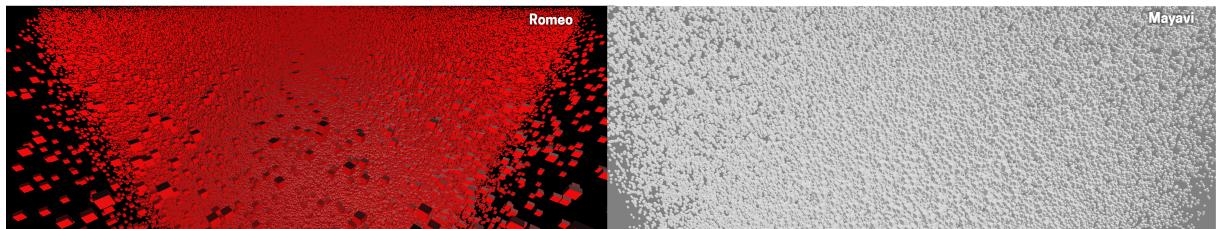


Figure 12: *Rendered particles*

The next benchmark is only between Romeo and Mayavi, as the other libraries did not offer a comparable solution. Matlab does not allow to use cubes as particle primitives and Vispy only had an example, where you needed to write your own shader, which can not be seen as a serious option. This is a benchmark for easy to use and high level plotting libraries. It is always possible to write an optimal version yourself in some framework, but what really interesting is, how well you can solve a problem with the tools the library has readily available.

Library	Still	Animated
Mayavi	90000	2500
Romeo	1000000	40000
Speed up	11x	16x

Table 5: *Maximum number of particles that could be displayed without stutter.*

Romeo is an order of magnitude faster in this specific benchmark. This is most likely due to the fact that Romeo uses OpenGL's native instance rendering.

6.1.5 IJulia

It was not possible to compare IJulia directly with Romeo, as the feature set for plotting is too different.

But there are certain factors, which indicate, that is hard to reach optimal performance with IJulia. First of all, IJulia uses ZMQ to bridge the web interface with the Julia kernel. ZMQ is a messaging system using different sockets for communication like inproc, IPC, TCP, TIPC and multicas. While it is very fast at its task of sending messages, it can not compete with the native performance of staying inside one language. This is not very important as long as there does not have to be much communication between Julia and the IPython kernel. This changes drastically for animations, where big memory chunks have to be streamed to the rendering engine of the browser. It can be expected, that

this will always be a weakness of IJulia. On the other hand, GPU accelerated rendering in a web browser is also limited. It relies on WebGL, which offers only a subset of the OpenGL's functionality. So while the execution speed of OpenGL can be expected to be similar, there are a lot of new techniques missing, which can speed up rendering.

To investigate this another benchmark has been created. It is between Romeo and Compse3D, which was the only library found to be able to display 3D models created with Julia directly from the IJulia notebook. This benchmark is not entirely fair, as Compose3D is just a very rough prototype so far. But there seems to be no other library with which you can easily create and display interactive 3D graphics in the IJulia or IPython notebook. This benchmark creates a sierpinsky gasket and Compose3D displays it in the IJulia notebook while Romeo displays it natively in a window.

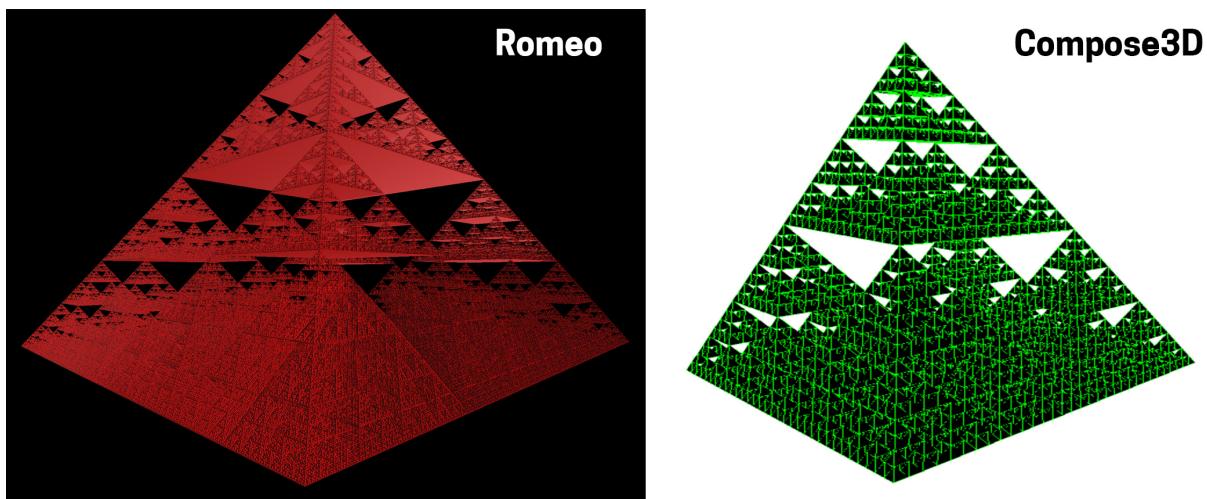


Figure 13: *Sierpinsky pyramid in 3D*

Library	Still
Compose3D	15625
Romeo	1953125
Speed up	125x

Table 6: *Maximum number of pyramids that could be displayed without stutter.*

Again, Romeo is an order of magnitude faster. This can change in the future when Compose3D matures. But one needs to notice, that Romeo utilizes OpenGL's instancing to gain this speed. Native instancing is not yet available in WebGL, which means that this optimization will not be available for the IPython notebook in the near future.

6.2 Extensibility Analysis

The modular design of Romeo has proven to be effective and the goal of re-usability has already proven itself. Most of the created modules are used independently by different people. GLVisualize is used by myself for two packages, namely GLPlot, a scientific plotting package for Julia and 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 the other packages. This can mean three things. First, it is badly abstracted and does not 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 surprise considering the minimal amount of documentation for GLWindow. And finally, considering the small group of people developing graphics for Julia, it could be that they simply do not need the lower level functionality of GLWindow and instead rely on the other written packages that use GLWindow.

Modularity guarantees a broad user and developer base, which in turn results in rich functionality and stability. From further analyzing the Github repository written for this thesis, one can find out that there is a general lack of documentation. This hinders people from contributing and using the packages.

The implementation in just one language has been achieved by choice. There are only a few exceptions, like the kernel code for OpenGL shaders, which currently can not be written in Julia. Julia programmers that use Romeo can extend Romeo with Julia and immediately see their results without complicated compilations. This together with the speed is one of the main achievements compared to other libraries offering similar functionality, like IJulia, Mayavi and Matlab. To further proof this point I will analyze the mentioned software in more detail. The language usage statistics and necessary tools needed in order to extend the software will be the main focus of the analysis. One needs to note, that the usage statistic of languages is just a weak indicator for the extendability of a software. Using different languages for one project can make sense if the project has different domains where domain specific languages give an advantage. As this means one needs to know all used languages, this still introduces complexity, but it is at least justified complexity. This chapter will only discuss the complexity introduced by languages, which are only needed for compatibility with other libraries or because the main language is too slow. This is something, which should ideally be avoided.

6.2.1 IJulia

IJulia is written in Julia and relies on ZMQ(C++) and IPython. IPython uses multiple JavaScript rendering back ends like Three.js and D3. [more to come]

Software	languages used
IPython	Python 78.5% JavaScript:15.1% HTML 5.0% Other 1.4%
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%

Table 7: *J*
Technologies used in IJulia. Statistics taken from Github

6.2.2 Mayavi and VTK

Table 8 in the appendix shows an extensive summary of the used languages in the Paraview repository. It amounts to a total of 3.642.105 lines of code written in 29 languages. [more to come]

6.2.3 Matlab

Matlab is closed source, which makes the core of Matlab impossible to extend by the user. This is why Matlab relays on a plug-in architecture, which enables developers to write closed or open source plug-ins for Matlab. [Analysis of the plug-in Architecture] [more to come]

6.3 Usability Analysis

Doing a broad user survey or similar methods was out of scope for this thesis. As a result from this, the usability study has to be done analytically. There are different aspects which can be analyzed. For example, how many function names need to be remembered, how easy they are memorized, if they expose the wanted functionality and how difficult it is to look up unknown functionality. For this thesis, I will analyze the two main packages, namely GLVisualize and GLAbstraction. The named aspects will be analyzed and in addition feedback from Github will be used.

GLVisualize has a very simple API, as it offers only five functions: visualize, visualizedefaults, edit and renderloop. There are also the functions bounce and loop, which offers a simplification for creating periodic signals. These functions might get moved into Reactive, though. So for GLVisualize, only very few function names have to be remembered. The question is, does this simple interface still allow people to create the visualization they want. At closer inspection one can see, that visualize is overloaded 67 times, with each of

these methods having a set of keyword arguments which enables further customization. These can introduce drastic changes. The particle visualization for example can take any mesh as a primitive. This enables a customization, which was not possible in such an easy way in the other examined packages. Also, most of the functions take either a data type, or a signal of that data type. This makes it very intuitive to animate your data. In contrast, in order to setup the animation for the other packages, it took quite some time to find out how to update values of an existing visualization. This is acceptable, as it might take quite some time to find out that Romeo uses signals and how to work with signals. But when this is found out, Romeo functions in the same, consistent way. Signals add a fourth dimension to any parameter or data you would like to visualize, making the usage principle consistent across the different visualizations.

For the other packages though, one needs to find out the names of the data for every visualization type in order to access and update them. Some attributes can not be animated, making the API even less consistent.

So for Romeo one can achieve anything by bringing the data into the right format. Problems arise, if this can not be done easily or the format is not intuitive for the programmer. To be fair, you will have this problem with every kind of visualization API. The difference is in the end, how easy it is to do the data transformations. Lets examine an example, where GLVisualize often will not allow to directly call `visualize` on the data. There is only a method for visualizing a Mesh, but not for a vertex list plus a face list. If you work with mesh data, you will often handle the face and vertex list isolated. So an API that offers a function like `visualize_mesh(x::VertexList, y::FaceList)` will be more straightforward for a programmer. Especially, as this is the standard way of displaying a mesh in most scientific plotting packages. This has become one of the most occurring questions on github, even though that there are usage examples for displaying a mesh.

But this functionality can be added in a simple way for GLVisualize by defining `visualize_mesh(facevertexlist) = visualize(Mesh(facelist, vertexlist))`. As GLVisulize should stay as close to the principle of only having one function name, this should be moved to other interface only packages, though. [more to come]

7 Conclusion

7.1 Future Work

8 References

- [1] Open Benchmarking. Llvm clang 3.6 compiler tests. Accessed: 04/15/2015 : <http://www.webcitation.org/6XoVc1S0y>.
- [2] Al Danial. Cloc. Accessed: 04/01/2015 : <http://www.webcitation.org/6XT73jFkv>.
- [3] Enthought. Mayavi. Accessed: 05/16/2015 : <http://www.webcitation.org/6YZNh5oEC>.
- [4] Enthought. Vispy. Accessed: 05/16/2015 : <http://www.webcitation.org/6YZPFZ0Dy>.
- [5] Github. surface slow and buggy 892. Accessed: 05/13/2015 : <http://www.webcitation.org/6YVE1zeNb>.
- [6] Sebastian Good. Little performance explorations. Accessed: 04/14/2015 : <http://www.webcitation.org/6Xmtrox4u>.
- [7] IPython. IJulia notebook. Accessed: 03/23/2015, Archived by WebCite®: <http://www.webcitation.org/6XFFIHQee>.
- [8] Viral Shah Alan Edelman Jeff Bezanson, Stefan Karpinski. Why we created julia. Accessed: 04/01/2015 : <http://www.webcitation.org/6XT6wqYAf>.
- [9] Kitware. Vtk gallery. Accessed: 04/15/2015 : <http://www.webcitation.org/6XodgQgdm>.
- [10] Julia Lang. Calling c and fortran code. Accessed: 04/02/2015 : <http://www.webcitation.org/6XUueZYLw>.
- [11] Julia Language. benchmark times. Accessed: 04/13/2015 : <http://www.webcitation.org/6X1X8Wwft>.
- [12] Michael Larabel. Compiler intel broadwell linux tests. Accessed: 04/15/2015 : <http://www.webcitation.org/6XoVX2L1r>.
- [13] Michael Larabel. Intel broadwell: Gcc 4.9 vs. llvm clang 3.5 compiler benchmarks. Accessed: 04/15/2015 : <http://www.webcitation.org/6XoW1HeDq>.
- [14] Michael Larabel. Microsoft announces an llvm-based compiler for .net. Accessed: 04/15/2015 : <http://www.webcitation.org/6Y0dC964v>.
- [15] MathWorks. Matlab pricing. Accessed: 03/22/2015, Archived by WebCite®: <http://www.webcitation.org/6XEFJ0PBE>.

- [16] Microsoft. Wglgetprodaddress documentation. Accessed: 02/27/2015, Archived by WebCite®: <http://www.webcitation.org/6WemKehYL>.
- [17] Tanmay Mohapatra. reduce gc load in readdlm. Accessed: 04/15/2015 : <http://www.webcitation.org/6Y0c9Qv1T>.
- [18] Amuthan Arunkumar Ramabathiran. Finite element programming in julia. Accessed: 04/14/2015 : <http://www.webcitation.org/6XmvHthh5>.
- [19] OpenGL Wiki. Rendering pipeline overview. Accessed: 04/10/2015 : <http://www.webcitation.org/6Xgd8fedi>.
- [20] Wikipedia. Ipyton. Accessed: 03/23/2015, Archived by WebCite®: <http://www.webcitation.org/6XFEB9BB3>.
- [21] Wired. The one last thread holding apple and google together. Accessed: 04/15/2015 : <http://www.webcitation.org/6Y0dawS5T>.

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])
```

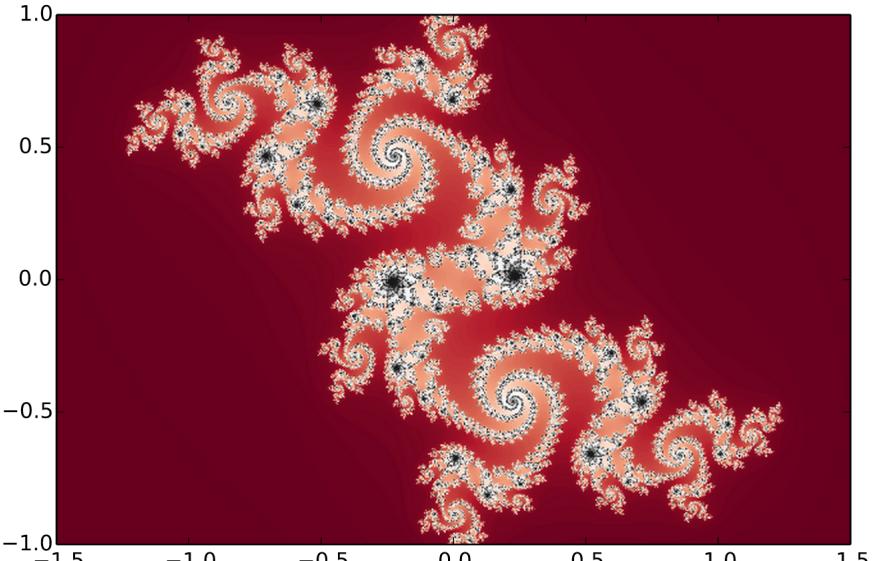


Figure 14: Example of an IJulia Notebooks. Screenshot taken from [7]

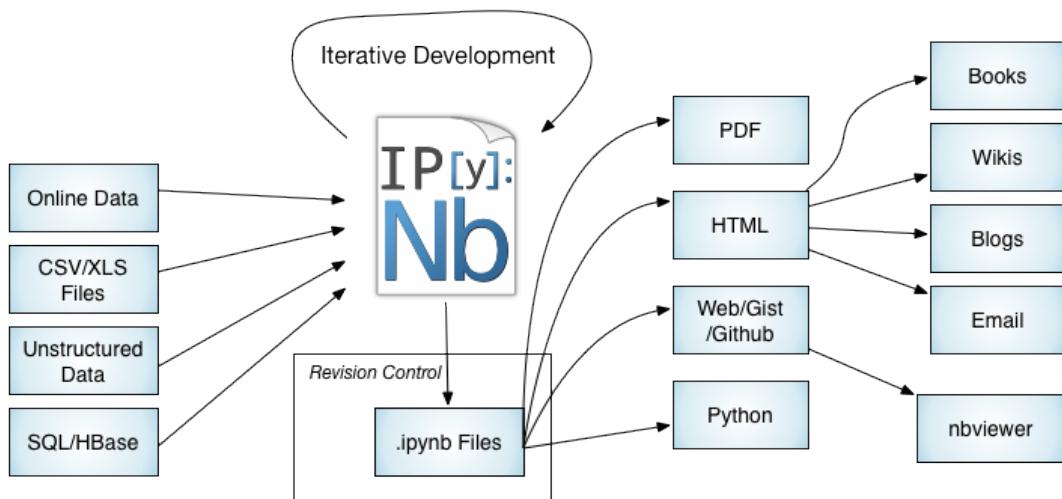


Figure 15: Workflow of IPython Notebooks. Graphic from Wikipedia [20]

B Language Statistics

All language statistics have been made with cloc [2] the current master of the github repositories.

Table 8: *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 10: *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

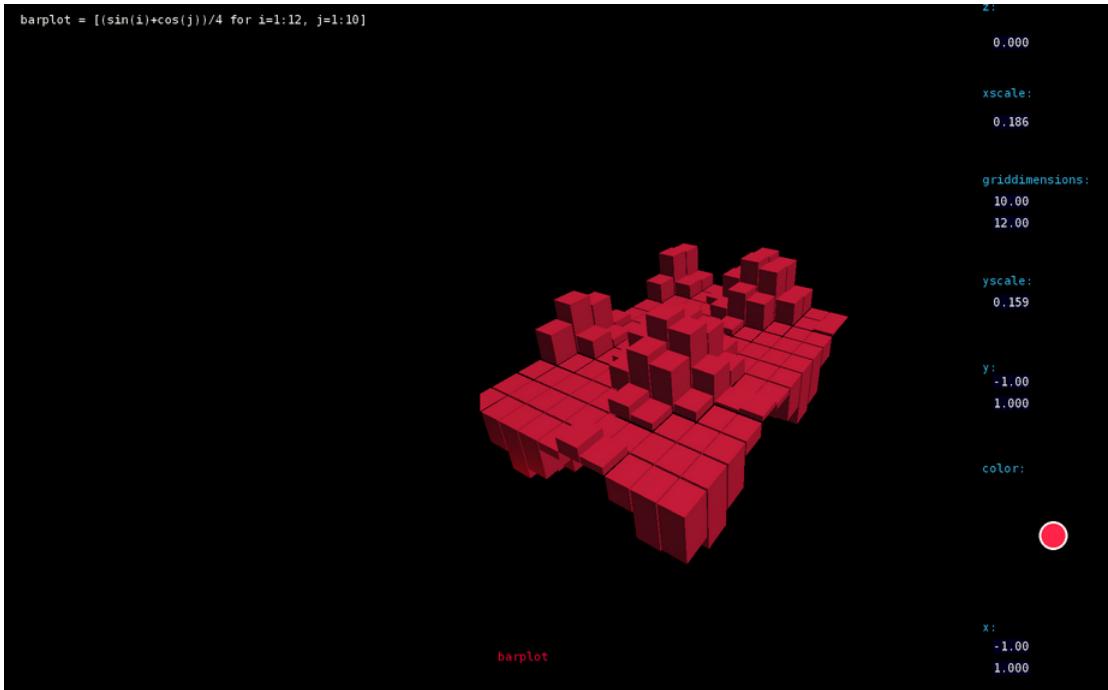


Figure 16: Screenshot of the prototype. Left: evaluated script, middle: visualization of the variable barplot, right: GUI for editing the parameters of the visualization

D Benchmark

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

Table 12: FE Implementation comparison

Benchmark	LLVM35	LLVM36	Difference
Rodinia	265.34	289.43	0.92
Rodinia	118.52	118.42	1.00
FFTW	6034.26	5961.52	0.99
FFTW	5988.02	5969.68	1.00
FFTW	4373.76	4349.26	0.99
FFTW	4405.26	4376.60	0.99
Timed HMMer Search	11.43	12.20	0.94
Timed MAFFT Alignment	4.69	4.69	1.00
SciMark	2008.04	1939.20	0.97
SciMark	556.37	537.07	0.97
SciMark	355.18	362.56	1.02
SciMark	2790.01	2452.42	0.88
SciMark	4843.14	4834.33	1.00
SciMark	1495.53	1509.64	1.01
John The Ripper	936.00	984.00	1.05
John The Ripper	5219000.00	5204000.00	1.00
John The Ripper	14767.00	14779.00	1.00
Himeno Benchmark	1572.74	1574.91	1.00
Timed Apache Compilation	23.56	25.34	0.93
Timed ImageMagick Compilation	19.13	20.67	0.93
C-Ray	12.13	12.73	0.95
Smallpt	148.00	145.00	1.02
Stockfish	3775.00	3812.00	0.99
Bullet Physics Engine	3.43	3.40	1.01
Bullet Physics Engine	5.85	5.95	0.98
Bullet Physics Engine	6.38	6.51	0.98
Bullet Physics Engine	5.99	5.68	1.05
Bullet Physics Engine	3.77	3.84	0.98
Bullet Physics Engine	1.25	1.23	1.02
Bullet Physics Engine	1.47	1.46	1.01
FLAC Audio Encoding	7.17	7.28	0.98
LAME MP3 Encoding	15.99	15.43	1.04
Hierarchical INTegration	240423016.30	264346632.10	1.10
Apache Benchmark	17643.10	19412.76	1.10

Table 14: LLVM 3.5 compared to LLVM 3.6 in the Phoronix benchmark test suite[1]

Benchmark	GCC492	LLVM35	Difference
Timed MAFFT Alignment	11.39	12.88	0.88
Timed MrBayes Analysis	25.44	26.28	0.97
SciMark	1179.35	1497.26	1.27
SciMark	564.44	603.62	1.07
SciMark	263.97	279.26	1.06
SciMark	1957.09	2070.94	1.06
SciMark	2046.08	2953.00	1.44
SciMark	1065.17	1579.54	1.48
John The Ripper	2382.00	926.00	0.39
John The Ripper	4296333.00	4925333.00	1.15
Himeno Benchmark	1618.11	1459.12	0.90
ebizzy	18192.00	17879.00	0.98
Timed Apache Compilation	55.65	38.61	1.44
Timed PHP Compilation	60.94	44.14	1.38
C-Ray	48.40	73.93	0.65
Smallpt	64.00	148.00	0.43
Stockfish	3933.00	4108.00	0.96
Bullet Physics Engine	5.75	6.08	0.95
Bullet Physics Engine	6.65	7.45	0.89
Bullet Physics Engine	5.76	6.59	0.87
Bullet Physics Engine	3.79	3.89	0.97
Bullet Physics Engine	1.23	1.31	0.94
Bullet Physics Engine	1.46	1.57	0.93
FLAC Audio Encoding	6.87	9.12	0.75
LAME MP3 Encoding	12.82	12.88	1.00
Hierarchical INTegration	206580965.69	237608504.18	1.15
Apache Benchmark	15429.86	15499.88	1.00
FFTW	6283.32	5443.12	0.87
FFTW	6053.00	5447.48	0.90
FFTW	4504.06	4260.74	0.95
FFTW	4091.20	4070.66	0.99

Table 16: *gcc 4.9.2 compared to LLVM 3.5 in the Phoronix benchmark test suite[12]*

Official Statement

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

Date:

(Signature)