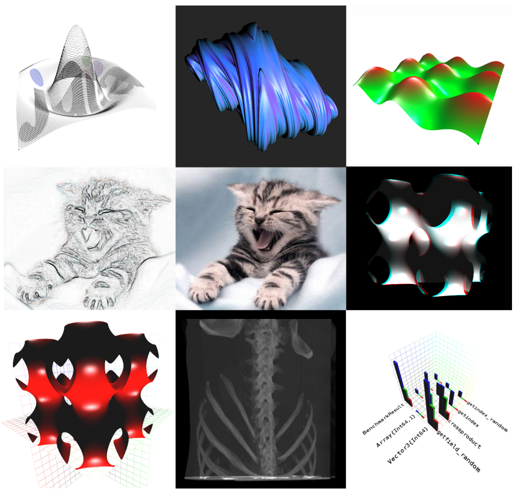
UNIVERSITÄT OSNABRÜCK

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 introduces Romeo, a prototype for an interactive scripting environment. Romeo allows the user to execute Julia code and visualize and interact with all variables of the script. Special care has been taken to make all components easy to use and extendable. A crucial choice was the programming language, which needs to be both fast and high-level. Julia is a novel programming language for scientific computing which promises to match the speed of C, while offering a concise coding style. This makes Julia a great fit for implementing a scientific visualization library, as speed is crucial for smoothly animated visualizations. The visualization library, named GLVisualize, was implemented separately from Romeo, to make it usable for similar projects. GLVisualize greatest accomplishment is to offer a simple way to animate all data via Signals while offering state of the art performance. It also offers Graphical User Interface (GUI) elements, including text fields, sliders and color pickers. All libraries are implemented in Julia and the Open Graphics Library (OpenGL). This allows Romeo users to achieve top performance even for large datasets while being able to stay in a high-level language for all tasks.

Acknowledgements

I like to thank Prof. Dr.-Ing. Elke Pulvermüller for being a great and patient supervisor for this project. She helped me to get through the messy process of fitting my personal project into the form of a Bachelor Thesis. I also have to thank Apl. Prof. Dr. Kai-Christoph Hamborg for going over the usability part of this thesis. My greatest respect goes out to the creators of Julia and LLVM. These two excellent projects build the foundation for this thesis and without them, it would miss its most crucial foundation. Finally, I have to thank Valentin Churavy, Felix Schüler and Steve Kelly for the tiring task of proofreading this thesis.

II Table of Contents

III List of Figures

IV List of Tables

V Listing

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
CPU	Central Processing Unit
GLSL	OpenGL Shading Language
OpenCL	Open Compute Language
WebCL	Web Computing Language
OpenGL	Open Graphics Library
VTK	Visualization Toolkit
AST	Abstract Syntax Tree
CUDA	Compute Unified Device Architecture
API	Application Program Interface
APIs	Application Program Interfaces
JIT	Just in Time
FFI	Foreign Function Interface
SPIR-V	Standard Portable Intermediate Representation - Vulkan
WebGL	Web Graphics Library

1 Introduction

This Bachelor Thesis is about Romeo, an interactive scripting environment for scientific computing. Other libraries have been developed to implement Romeo in a modular way, fostering reusability. The most important library is GLVisulize, which is a generic 3D visualization library for dynamic data. The ultimate goal is to make scientific computing more accessible to the user.

For this, the focus has been put on usability, applied to all the different interfaces, ranging from Application Program Interfaces (APIs) to graphical user interfaces. The programming language Julia was chosen for implementing all libraries, as it offers a unique combination of speed, high-level programming style and a lot of functionality for scientific computing.

GLVisualize offers a default visualization for every data type. It also offers GUI elements and editable text fields, which forms the basis for editing scripts and visualizations in Romeo. In this way, Romeo offers a primitive form of visual debugging. Due to the generic nature of GLVisualize, these interactive features can be used in other programs independent from Romeo.

An introduction to the general field of research and its challenges and problems relevant to this thesis will be given. This chapter will conclude with a solution to the previously found problems, how to measure the success and give an outlook on the structure of the entire Bachelor Thesis.

1.1 Scientific Computing

Scientific computing is programming needed for scientific research. It is a very broad field involving a lot of different challenges. In some areas such as particle physics, most problems are computationally intensive and can only be solved with the help of supercomputers. In other areas, like robotics, it is important to be efficient because the algorithms are running on embedded systems with limited resources. In other areas, speed is not that important, but it may be that the algorithm itself is very difficult to comprehend. The more comprehensible an algorithm can be written in a programming language, the easier it will be to implement the algorithm without errors. In any case, programming itself is secondary to the researcher. It can be expected that most researcher only have rudimentary programming skills. Even if they are experienced programmers, it is often inefficient to implement complex algorithms while also dealing with a lot of programming problems. These problems especially arise, when dealing with low-level programming in order to reach optimal speed.

So things like manual memory management and difficult design patterns with a lot of unnecessary code should be avoided in scientific computing. This has led to the rise of high-level programming languages and tools specifically tailored to scientific computing[7]. 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 a simple syntax for linear algebra and statistical code while taking away difficult tasks like memory management and multithreading. They come with a rich standard library so most research can be done without loading any additional module, which makes them great tools for rapid prototyping. Currently, these languages suffer from poor performance introduced by the high level of abstraction. As a consequence, a lot of the performance critical parts are written in another language like C/C++ or Fortran. This is referred to as the **Two Language Problem**[5].

This leads us straight to the field of research and the problems that are addressed in this thesis.

1.2 Field of Research and Problem

In a slow high-level programming language like Matlab, one needs to switch to a fast multi-purpose language, as soon as high-performance is needed. In this case, one is losing 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 algorithm has been confirmed to work, it gets rewritten in a fast language.

This increases development costs and makes it harder to improve the algorithm in the future.

One of the first languages promising to solve this dilemma for scientific computing is Julia[6]. It is designed to be high-level and optimized for the work of scientific computing while approaching the speed of statically compiled languages like C.

This thesis is about bringing usability and performance together in the realms of scientific computing and 3D visualization. These two demands are opposing concepts. The first is about bringing tasks into a form which is best understandable to humans, and the other is about transforming a task to make it perform well on a specific computer architecture. These two tasks could not be more different. For humans, data and algorithms become understandable if they are high-level and represented visually, auditorial or tactile together with immediate feedback. It is the task of making problems accessible to a human, who has evolved 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 it cares about, whether a human understands this or not.

To close this gap, compilers 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 have no education in computer science. Brat Victor writes excellent essays about making programming easier and more understandable to the novel user and is one of the most influential researchers in this field. He demonstrates, how visualizations, interaction and well chosen abstraction can make programming easier[61][59]. The field of scientific computing would benefit greatly from integrating Bret Victors ideas.

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[7]. Barriers in this field make it hard for novel users to have a pleasant start. A few barriers are for example difficult installation processes of specialized software, working on datasets without visualizations and interaction and cumbersome Application Program Interface (API)s. This calls for a visualization library offering the interaction and well-designed APIs. Most state of the art visualization libraries use C++ at their performance critical core, are closed source or they are implemented in a slow language, introducing unavoidable performance bottlenecks.

Using C++ introduces complexity and performance bottlenecks when interfacing with other languages. This is especially true for languages that produce assembly which is not binary compatible to C++. The next problem occurs, when the library does not offer the needed functionality and the programmer has to step in and extend the library. If it is closed source or one has to switch the language for that, this will either be not possible or introduce additional work. It is nice to have a library in which one can get results very quickly even though it does not offer high performance. But it is very frustrating when one needs to switch to another library for more serious projects, as previously learned concepts and work have to be discarded. So it is desirable to have a library which scales well from small projects to big projects.

Finally, one often does not have GUI elements. Even if there are GUI elements, they might come from a different package (possibly written in yet another language) or they are complicated to use. All in all, this makes it hard for researchers to visualize and interact with their data and to create solutions which are tailored to the research 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*

Visualization libraries play an integral role in scientific computing. As pointed out by Bret Victor, the computer should be used much more to give us insights into algorithms and forming connections between the lines of code and what they actually represent[60]. A good visualization can be the primer to understanding problems better or bringing across novel theories. Consider the following function: $f(x,y,z) = \sin(\frac{x}{15}) + \sin(\frac{y}{15}) + \sin(\frac{z}{15})$. It describes a 3D volume mathematically. This is a simple function, but it is already not that easy to interpret. In Figure 1 different visualizations of f are shown. If one 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 opens the gates of scientific computing to novel users.

Also, debugging problems can be easier with a good visualization. If one writes an algorithm that calculate normals of faces on a mesh, the errors in the math becomes obvious when visualizing the normals. Without a proper visualization, the output of the algorithm is really hard to comprehend since it is just a huge array of numbers. Performance bottlenecks in a call-graph can be seen easily if the graph is color-coded for the execution times of a call. Making these tasks enjoyable can help to get an easy start in scientific computing.

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 via built-in, or user defined, visualization routines. An example would be a materials researcher, who is investigating different 3D shapes and materials and their reaction to pressure. The researcher would need to read in the 3D object they want 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. Also, while doing all this, they may want to modify the script that calculates the pressure. There are quite a few libraries out there offering this. The unique benefits of GLVisualize will be discussed in the second chapter.

1.3 Contribution

The main contribution of this thesis to the field of scientific computing is writing GLVisualize in Julia, which offers the following advantages.

Julia is a high-level language and effort has been put into creating a concise architecture. This allows for short development cycles and easily extendable libraries.

Romeo targets researchers that want to write everything in one language. 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. This makes it easy to create visualization pipelines in which every routine is as fast as it can get. On top of that Romeo uses modern OpenGL, which allows to achieve fast, hardware accelerated 3D renderings. Also, the researcher can extend the library in the same language they are already working in. In the case of Julia and Romeo, this is especially easy, as every project involved is open source and directly accessible. This allows for the flexibility and transparency which is needed for big projects.

Another feature that sets GLVisualize apart from other libraries is, that is build for animations from the beginning.

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

This thesis will continue with a chapter **Background**, which gives the reader an overview over Julia and similar languages with their respective 3D visualization libraries. After this, the requirements for Romeo will be laid out and how to measure if the requirements were met. Then the technologies used will be explained, which builds the basis for discussing the implementation. In the chapter **Results and Discussion**, one will find if the requirements were met and how Romeo compares to similar software. This chapter leads straight to the conclusion which will summarize this thesis.

2 Background

In this chapter, a short overview over the current state of the art for visualization in the field of scientific computing is given. The chapter will start out with a look at Julia.

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 one of the main goals. So Julia plays a crucial role in this thesis.

Julia was published first in 2012 which makes it a very new language. As of July 2015 it is at version 0.3.10 and 0.4 is in development. Following semantic versioning, this means Julia is still in an early release phase with the core features and names anticipated to change[53]. Julia is a multi-paradigm language for scientific computing and uses the compiler infrastructure LLVM to generate fast assembly code.

Some of its most important features are multiple dispatch, a dynamic type system, macros, good performance and an interface to C/C++, R, Java 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 numerical problems. 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 run-time. A special form of Just in Time (JIT) compilation is used to achieve this. 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[33]:

- C-like performance
- native C interface
- macros similar to Lisp
- mathematical notations like Matlab
- as 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 built-in types. This allows Julia to implement all arithmetic types in Julia itself, allowing anyone to extend and rewrite them without the need to modify the compiler. Julia claims to take an approach at scientific computing which is more modern than other programming languages. They justify this by pointing out the performance characteristics of Julia and the possibilities that come from this[48][7]. It allows to write even performance critical libraries in Julia. This means Julia does not need to call out to C for performance and because of this most of the standard library is implemented in Julia. In contrast, Matlab, Python, and R need to implement any performance-critical code in another language, which has led to a programming pattern which is best known by the name of vectorization[7]. This pattern has evolved, as the vector operations that are built into the language are much faster than vector operation written in the language itself. This means, if one needs performance, the code needs to be rewritten to only use functions which are implemented in another, faster language. A deeper analysis of this problem can be found in the first Julia paper[7].

All in all, 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[33] have been achieved.

2.1.2 IJulia

IJulia[35] 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 allows to write formatted documentation together with data, inlined plots, and executable program snippets. Formulas can be written in latex and they will get rendered and inlined nicely into an IJulia Notebook. By implementing a language backend for Julia, all this functionality is accessible from Julia. See Figure 16 for an example notebook. Figure 17 gives an idea of the workflow in IPython.

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 the Web Graphics Library (WebGL). The current integration is just prototypical and limited to simple 3D meshes up to now[34][57].

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 ease 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 is currently developed by the company Mathworks, which was founded in order to keep improving Matlab.

Matlab offers a broad range of functionality, including matrix manipulation, plotting of functions and data, the creation of user interfaces and interfacing with a range of languages like C/C++, Java, Fortran and Python. Matlab is known for having print ready visualization tools deeply integrated into the standard library. Also, when relying on vectorized code it can be very fast while programs written in pure Matlab code tend to be slow. Matlab itself is written in C, C++, Java and MATLAB. It is a proprietary software with a pricing of around 2000 €[50], which can be extended via free, open source and proprietary modules like Simulink.

Romeo intends to lay out the groundwork to provide a similar deep integration of visualizations in Julia. It is quite far away in terms of functionality, but it builds upon a more modern architecture. Romeo is using modern OpenGL and Julia intends to solve one of Matlab's biggest problems, namely the need for vectorization. Finally, what might be the main advantage of Romeo, GLVisualize and Julia is that they are open-source, making them much more accessible and easier to extend than Matlab.

2.1.4 Mayavi and VTK

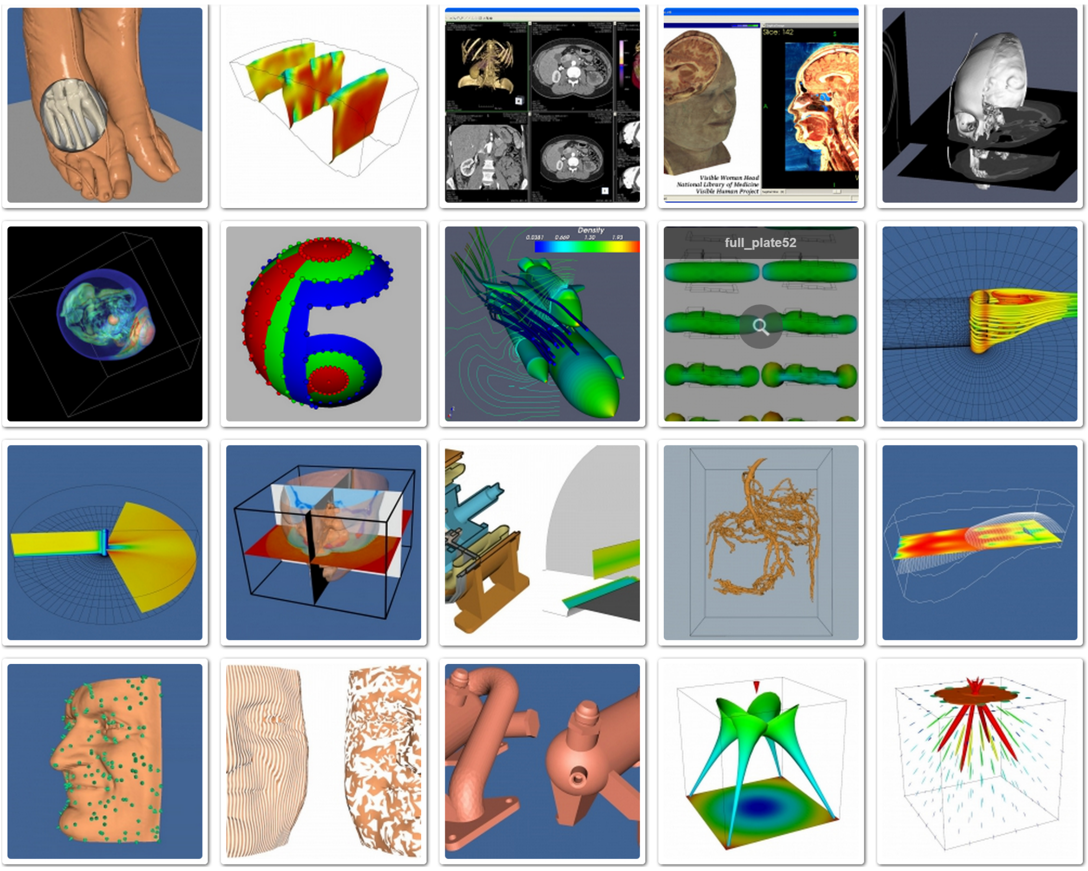


Figure 2: *Different visualizations done with VTK.*

Mayavi[55] is probably one of the biggest open source libraries for interactive 3D visualizations in Python. It is written almost completely 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 one can see some of the visualization taken from the VTK gallery[37].

Mayavi shares some of its goals with Romeo, namely[22]

- 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. The scripting part is very similar to the goals of Romeo. 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 10 and 12. The biggest difference is, that Romeo is implemented in a scientific programming language, while Mayavi's 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, which is the case for some python distributions, 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.

2.1.5 Vispy

Vispy is yet another interactive 3D visualization library. It is from the goals and development status the closest to Romeo. These include[62]:

- 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 being able to achieve state of the art performance. These goals are very similar to GLVisualize's, with the only difference being that GLVisualize is completely implemented in Julia while Vispy is implemented in Python. So the biggest differences between GLVisualize and Vispy will be found in the performance and the concrete feature set.

3 Requirements

All building blocks in this thesis are developed with the purpose in mind to give the user the ability 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 were made in order to achieve state of the art speed.

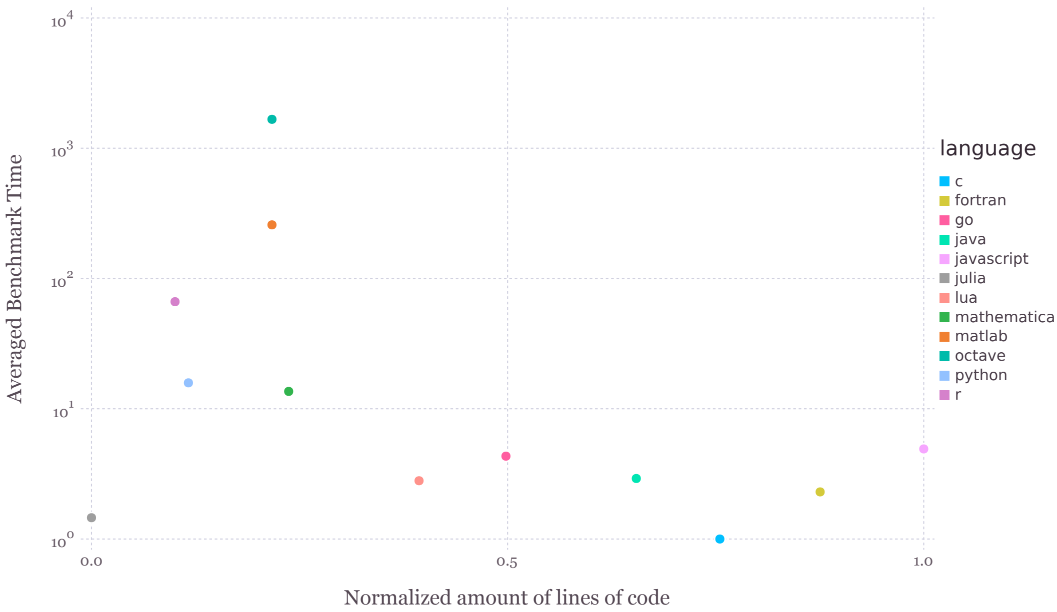


Figure 3: *Languages speed relative to C (averaged benchmark results), plotted against the length of the needed code. The source can be found on Github[14][39].*

3.1 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 cannot 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 3. The code length is an ambiguous measure for conciseness, but if the code is similarly refactored and implements the same algorithm, 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. This is one of the reasons why it has been chosen as the programming language for this project.

There are a lot of libraries out there which can be used as a basis for implementing a 3D visualization library. If the previous demands are taken 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 pointed out in Figure 3, 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 essentially the same functionality. Only OpenGL is truly cross-platform, which is why OpenGL has been chosen. To access OpenGL from within Julia, a wrapper had to be written. This leaves us with one binary dependency not written in Julia, namely the video driver, which implements OpenGL.

Measurement of success is pretty straightforward, 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 technology 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.

3.2 Extensibility

Extensibility is an important factor in scientific computing, as a lot of flexibility is needed when exploring new horizons. It is not only that but also a great factor determining the growth of a library. 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 requires 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. 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, difficult compilation procedures, missing documentation and cryptic highly optimized code. Furthermore, 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 and 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.

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.

3.3 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. The cleanest abstraction for animations and events are signals. Signals represent 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. Reactive[27] was chosen for this as it offers the nice abstraction of signals. It will be discussed in more detail in chapter 4.3.

3.4 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. The lowest relevant abstraction is the programming language. The next level of abstraction is the general architecture of the modular components, which has been discussed previously. This section specifies the API design choices made.

The first API is the OpenGL layer. The chosen philosophy was to make the wrapper for native libraries as thin and reusable as possible and a one-to-one mapping of the underlying C-library. This guarantees reusability for others, who want to work with the low-level library. Also, they might disagree with some higher-level abstractions and prefer to write their own. The developed library is called ModernGL[17].

Over this sits an abstraction layer needed to simplify the work with OpenGL, namely GLAbstraction[19]. Based on this abstraction, GLVisualize[20] was implemented, which renders the OpenGL accelerated visualizations.

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 one can create different types for the same data without losing performance. One can create a type representing a unit like meters which has the same performance footprint as a native floating point type, but the visualization function can specialize to this with multiple dispatch. This way, a single function, e.g. *visualize*, can create a default visualization for every data type. Instead of manually passing additional information to the visualization function most information is already encoded into the type. Together with the event system introduced in 4.3, it is possible to edit and visualize rich data with a simple interface. This makes it possible to visualize variables without further user interaction, which is perfect for visual debugging. For a more fine-grained control of the visualization, more information is needed.

Addition information can be passed via key word arguments. An optional argument to the function is a style type, which controls the chosen defaults and the type of the visualization. So for a float matrix, the default visualization may be a height-field, but passing the style `text` to the function might change the default to render a textual representation of the matrix. This makes it easy to extend the visualize function, as the user just has to overload the function with a custom style together with optional key word arguments. Finally, there are also graphical user interfaces developed for this thesis. As optimizing them is out of the scope of this thesis, they are kept very simple.

The measurement of success is again relatively difficult to evaluate. Best would be to make a user poll to get actual feedback from people that use the software. This is a pretty demanding task, so instead the interface will just be evaluated analytically.

4 Used Technologies

4.1 The Julia Programming Language

The basic introduction to Julia has already been given in the Background chapter. This chapter is focused on how to write programs with Julia and if it satisfies the requirements. Most influential language constructs 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 types[6]. Composite types, which are comparable to C's structs and can be mutable or immutable, parametric composite types, bits types, abstract and parametric abstract types. While the first three are all concrete types, abstract types cannot be instantiated 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. One can build type hierarchies like this:

```
abstract Number
abstract FloatingPoint{Size} <: Number # inherit from Number
bitstype 32 Float32 <: FloatingPoint{32} # inherit from a parametric
    abstract type
type Complex{T} <: Number
     real::T
     img::T
end
```

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

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

What will happen at runtime is, that Julia compiles a method specialized on the arguments which result in overloading the function with a concretely typed method.

In the case of *foo*, it is initially overloaded with two methods. Now, if one calls *foo* with one *Float32* argument, a new method will be added at run time, specialized to the *Float32* argument. Like this, if the function does not access nonconstant 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 more of a functional oriented language. But there are also ways to use Julia in a more object-oriented style. 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 the function via the arguments

Another crucial feature is the very simple, overhead free 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[38]. This is perfect for integrating low-level libraries like OpenGL and the Open Compute Language (OpenCL).

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. This is why a performance analysis is done to prove that Julia is a good choice for a high-performance visualization library.

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 is 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 Julia's performance.

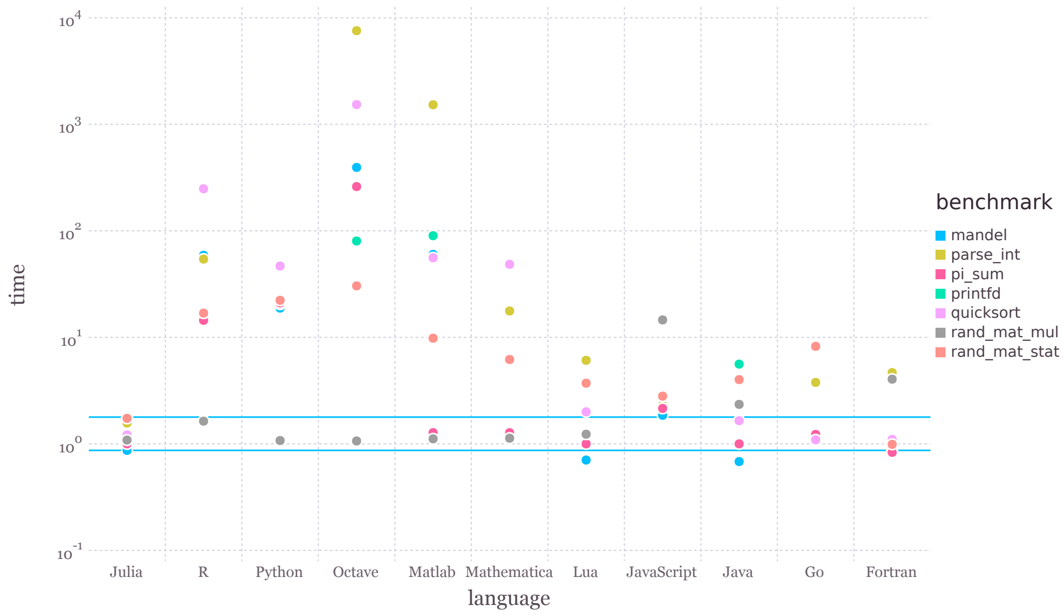


Figure 4: *Julia's performance compared to other languages, taken from Julia's micro bench suite [39]. Smaller is better, C performance = 1.0.*

In the first benchmark from Figure 4, 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 and the performance of a language is a moving target.

So it is not guaranteed, that there is no bias favoring Julia in these benchmarks. Another benchmark was found, which compares C++, Julia, and F#. It was created by Palladium Consulting[10] which should not have any interest in favoring any of the languages. They compare the performance of C++, Julia and F# for an IBM/370 floating point to IEEE floating point conversion algorithm. The results[26] have been, that F# comes out last with 748.275 ms, then 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 cannot 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 solvers, 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 an FEM solver written in Julia compared to some existing libraries. [54]*

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 incremental garbage collector. As seen in the benchmarks, it does not necessarily introduce big slowdowns. But there are issues, where garbage collection introduces a significant slowdown[52]. Analyzing 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 tell how big the actual differences will be. Another big difference is between different compiler technologies. Julia uses LLVM[45] for compilation. In order to further understand Julia's potential, a further look at LLVM is taken in the next section.

4.1.1 LLVM

LLVM is a compiler infrastructure, which has front ends for different languages and compiles to different platforms like x86, ARM, OpenCL (via SPIR) and the Compute Unified Device Architecture (CUDA) (via NVPTX). A language designer must create an Abstract Syntax Tree (AST) which LLVM can convert into LLVM's Intermediate Representation (IR). This IR forms a standardized basis for any optimization step. Every language that can be converted to LLVM IR can be combined at this level. LLVM itself offers many optimizations, but also third party optimization can be integrated. This yields superior language interfacing for LLVM based languages, as inlining and other optimizations can be done across the boundary of one language. This is why Julia can look forward to having one of the best C++ Foreign Function Interface (FFI)[23].

LLVM's concept is effective, as it makes it possible to accumulate state of the art optimizations in one place, making them accessible to many languages. Because of the many back-ends, languages that use LLVM can run on many architectures. While Julia does not support all back-ends yet support will be added in the future. LLVM is also used by Clang[9], the C/C++ front end for LLVM rivaling gcc, it is used by Apple's programming language Swift[3] and for Mozilla's language Rust[40]. This makes LLVM a solid basis for a programming language, as these are highly successful projects guaranteeing LLVM further prospering. To see where LLVM stands considering performance, one can compare it with gcc, which is one of the most successful open source compilers for C++. If C++ code that is compiled with gcc is much faster than the same code compiled with LLVM, it can be expected that the gcc version will also be faster than a comparable Julia program. In order to investigate the impact of this, a benchmark series written by Phoronix will be analyzed. They benchmarked gcc 4.92 against LLVM 3.5 and LLVM 3.5 against LLVM 3.6. Here is a summary of their exhaustive benchmarks:

Statistic	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. [4][41][43]*

The full tables can be found in the appendix under the table 15 and 17. 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 one considers that LLVM is much younger than gcc. With big companies like Apple[66], Google[66], Mozilla[40], AMD[42], NVidia[56] and Microsoft[44] being invested in LLVM, it is to be expected that LLVM will stay competitive, which means Julia should, in theory, stay competitive as well.

4.2 OpenGL

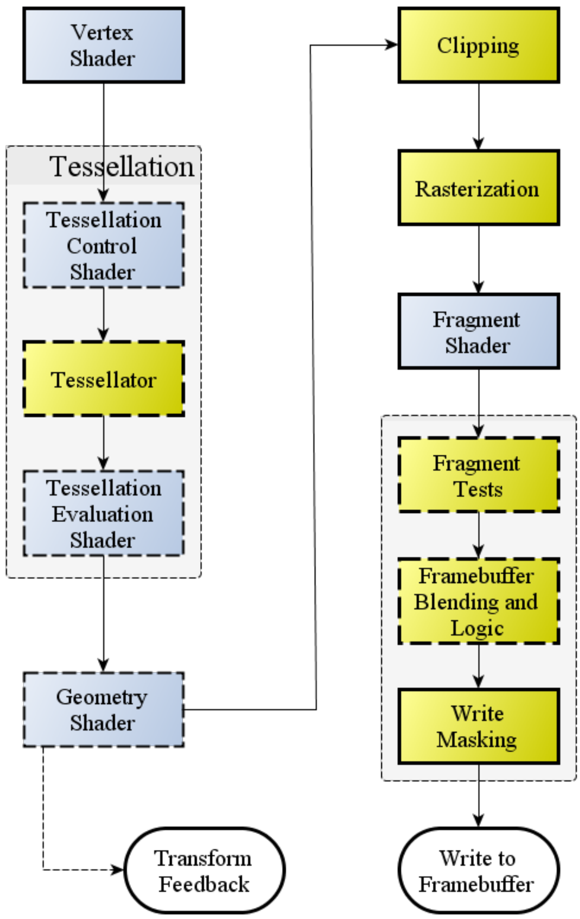


Figure 5: *Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages. Arrows show the flow of data[64]*

OpenGL is a low-level graphics API worked out by the Khronos Group[29] and 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 target, trading between modernity and platform support. For Romeo, it was decided to support OpenGL 3.1 as the lowest bound, as it is sufficiently available while still having most of the modern features. All features that help to call less OpenGL functions can be considered as modern, as they take away the load from the CPU. The modern features used in this thesis include instance rendering, vertex arrays, and shader programs written in OpenGL Shading Language (GLSL).

In Figure 5 the basic architecture of an OpenGL program pipeline is shown. 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.

In order to have a working OpenGL rendering pipeline at least a vertex shader and a fragment shader needs to be written. All shaders are compiled and linked into a program object, which can be executed on the GPU. Shaders are written in GLSL, which is a C dialect specialized for vector operations. Data gets loaded into the shaders 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 usually used to apply geometric and perspective transformations and calculating the light. In newer APIs general compute operations are available, making it possible to create more flexible shader stages. Displaying objects can only be achieved by rasterizing pixels to the framebuffer via the fragment shader. The framebuffer can be sent directly to the monitor. Framebuffers can contain multiple render targets, which are the buffers the fragment shader can write to. The write operation is heavily restricted. The fragment shader can only write to the location calculated by the vertex shader and simultaneously reads from the framebuffer are not possible. This restriction exists to allow for the massive parallel execution model that OpenGL uses to speed up rendering times.

The usual set of render targets includes a depth channel, stencil buffer and, of course, the color buffer. The depth channel is usually used to discard all fragments that are behind another fragment (known as depth test) while the stencil buffer can be used to discard arbitrary fragments based on the stencil mask. Custom render targets can be created in newer OpenGL versions which can be written to via the fragment shader. Here is a simple minimal example for a program rendering some vertex data with a flat color to the screen.

```
//Vertex Shader
in vec3 vertex; // vertex fed into the shader via a buffer
uniform mat4 projection; // Projection matrix
uniform mat4 view; // View matrix, setting rotation and translation of
    the camera
```

```
void main()
{
    gl_position = projection*view*vec4(vertex, 1); // apply
        transformations to vertex and output to fragment shader
}
//Fragment shader
out vec4 framebuffer_color; //color render target, which will get
    written to the display
void main()
{
    framebuffer_color = vec4(1,0,0,1); // write a red pixel at
        gl_position from the vertex shader.
}
```

All visualization code is written in OpenGL shaders, which are compiled and executed via GLAbstraction.

4.3 Reactive

Reactive[27] is a functional event system designed for event-driven programming. It implements Elm's[11] 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.

```
a = Input(40)         # an integer signal.
b = Input(2)          # an integer signal.
c = lift(+, a,b)      # creates a new signal with the callback plus. Equal
      to c = a+b
lift(println, c)      # executes println, every time that c is updated.
push!(a, 20)          # updates a, resulting in c being 22
#prints: 22
push!(b, 5)           # updates b, resulting in c being 25
#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 original 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 this 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. Reactive is used in all parts of the library. It builds the basis for the camera code, the widgets and any value that needs to be animated is realized via a signal.

4.4 GLFW

GLFW[25] 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. Every operating system exposes this functionality in a slightly different way, making it very hard to create a window and to retrieve window events. So a library like GLFW is crucial if one does not want to waste a lot of time just to implement all the corner cases for all the different operating systems out there. In addition, GLFW exposes low-level features like the operating systems OpenGL context handle. This can be used for creating advanced contexts that share the memory with another context. Romeo does not use this feature yet, but it makes GLFW a future-proof choice.

5 Implementation

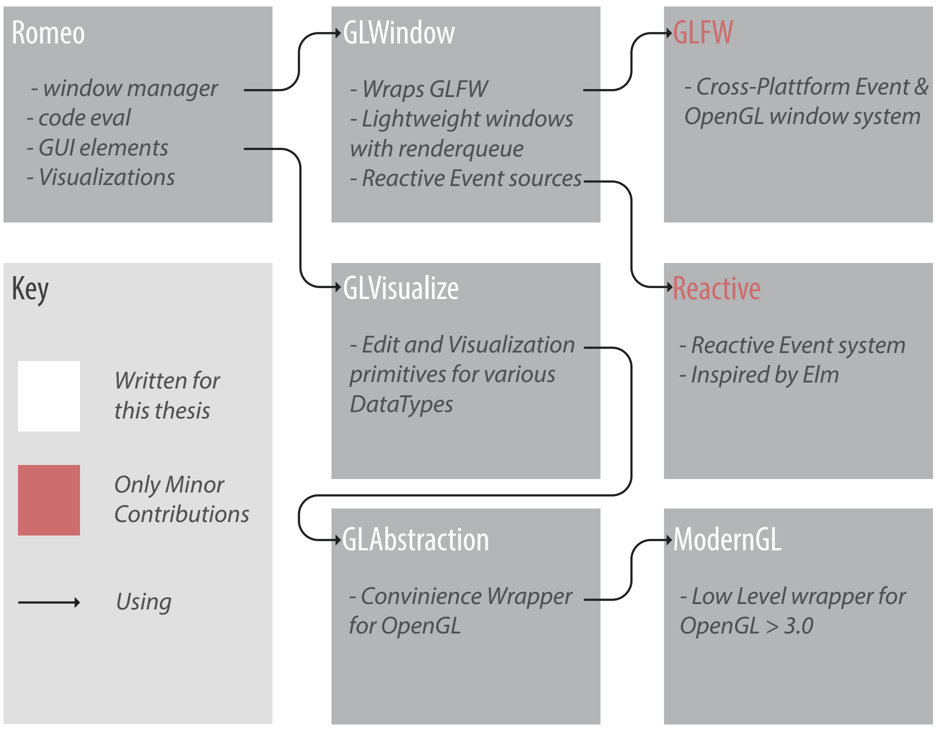


Figure 6: *Main modules used in Romeo and their relation (simplified).*

This chapter is about the implementation of Romeo and GLVisualize. 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 desired behavior. Romeo relies on a multitude of packages, which step for step abstract away the underlying low-level code that is used for the window creation and rendering. As already pointed out in the requirements, special care has been taken to make all modules self-sufficient. Every single package can be used for other applications, which allows for higher flexibility and a broader user base. ModernGL can be used by any OpenGL project which strives to build a low-level OpenGL program. GLWindow and GLAbstraction can be used to build slightly higher level OpenGL visualizations. GLVisualize can be used to build other interactive visualization packages besides Romeo.

As shown in Figure 6, Romeo uses GLVisualize for creating GUI elements and the visualizations. Romeo's code evaluation is done via Julia built-in functions and text fields are created with GLVisualize. Windows are managed by GLWindow. GLWindow creates an OpenGL window with the help of GLFW and converts all window events into Reactive's signals. It also offers a very simple render queue, for rendering graphics attached to a window. Signals are not only used as the event sources but are also the main abstraction for time-varying values in GLVisualize. 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 for ModernGL. ModernGL implements the OpenGL function loading and exposes all the function and Enums definitions from OpenGL with version higher than 3.0.

5.1 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 pointers are only available after an OpenGL context was created and may only be valid for this context[51]. 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 of ModernGL has undergone some changes over time. Starting with a very simple solution, there have been pull requests to improve it. The current approach in ModernGL master was not written by myself, but by the Github user aaalexandrov[1]. Before aaalexandrov's approach, the fastest approach would have used a pretty new Julia feature, named generated 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. When the generated function gets called the pointer can be queried and gets inlined into the just in time compiled function.

Generated functions only work with the newest Julia build, which is why aaalexandrov's approach was favored.

5.2 GLAbstraction

GLAbstraction[19] 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. It also makes it easy to create them from Julia data types like arrays and images. Additionally, it implements convenience functions to load shader code and to provide 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 transformations and camera code. Building upon that, it defines a signal based perspective and orthographic camera.

Signals are an important part of the infrastructure not only for the camera but for everything that changes. As an example, the function that creates an OpenGL program also takes signals of shader source code. With every source code update the OpenGL program gets recompiled, making it possible to interactively develop OpenGL shader. The signal can come from file updates, or from some text editing widget inside Julia.

One of the main data types is the *RenderObject*. It combines uniforms, OpenGL buffers, textures, and programs. One can call *render(::RenderObject)*, which executes the program with the given uniforms and buffers loaded into the program. Creating visualizations with GLAbstraction turns into writing the shader and then combining it with the needed data in the form of OpenGL types. When the program includes a fragment stage, calling *render* results in the object being displayed on the screen. But it is also possible to use compute shaders as *RenderObjects*, which then just write into the supplied buffers without displaying anything.

A lot of OpenGL functions are bothersome to use from within Julia, as the output has to be pre-allocated and gets then passed to the function. These functions have been overloaded in GLAbstraction to return the value instead. With only ModernGL, the code would typically look like this:

```
result = Ref{GLint}(-1) #using Julia's reference type
glGetShaderiv(shaderID, GL_COMPILE_STATUS, result)
result[] # dereferencing
```

With GLAbstraction this becomes possible:

```
result = glGetShaderiv(shaderID, GL_COMPILE_STATUS)
```

Similarly, OpenGL does not overload functions, which means that a function that does the same for different types has a base name combined with different suffixes to indicate the type. The function with the most methods is *glUniform*, which has around 34 methods for all the different uniform types. It is used to set the value of an OpenGL program uniform. With GLAbstraction, *glUniform* has been overloaded for all different types, including arrays and signals. So one can simply call *glUniform* on the type without the need of looking up the correct suffix. This has been achieved with a combination of overloading and generated functions. The generated function puts together the right function name and inlines it into the *glUniform* method specialized to the argument type.

A lot more of these simplifications have been done in order to leverage the usage of OpenGL with Julia. This guarantees that the code in GLVisualize is very short and concise, making it easy to extend and maintain.

5.3 GLWindow

GLWindow[16] 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.

5.4 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, Reactive does not allow rearranging the event tree. One cannot create subtrees in advance and then fuse them together at run time. This led to the design choice, that signals are sampled from a render loop. This is a sub-optimal approach, as sampling artifacts can occur, and frames are rendered even if the scene does not change. In the future, it will be desirable to work around this and bring the elegant functional approach from Reactive to OpenGL.

5.5 GLVisualize

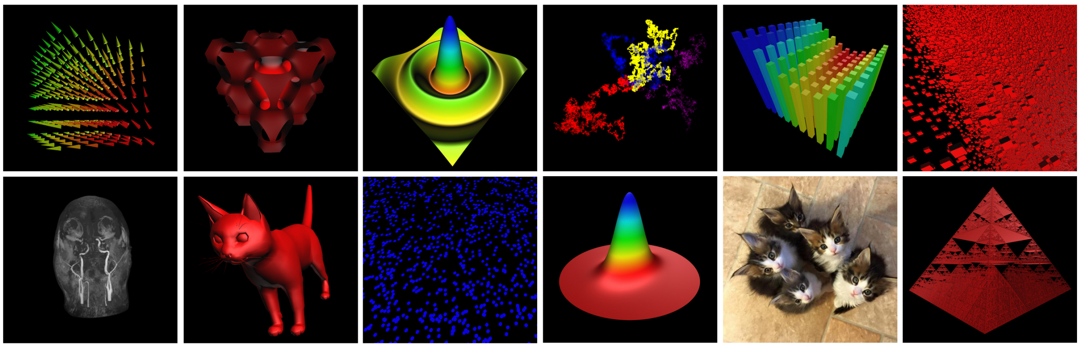


Figure 7: *Different visualizations rendered with GLVisualize.*

GLVisualize[20] implements the visualization functionality. 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, which is made possible via multiple dispatch. 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 few transformations of the data as possible and directly transferring the data 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:

```
Dict{Symbol, Any}      = visualization_defaults(data::Union(Signal{T}, T), style::Style) # returns a dictionary of default parameters
RenderObject           = visualize(data::Union(Signal{T}, T), style=Style{:default}; parameters...) # returns an object which can be directly rendered
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 by passing signals. These interactions can be purely programmatically, or via the widgets from the edit function. The visualize methods that take Julia objects are transforming them into GPU objects, which will then get transferred to the actual visualize method. This method takes GPU objects as its arguments, making it easy to visualize objects which already exist on the GPU, or which are shared between visualizations.

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

5.5.1 Mesh primitive Rendering

The rendering of meshes in OpenGL is pretty straight forward with a normal vertex and fragment stage. Vertex, Normal and UV data is supplied via Vertex Arrays and the perspective transformations via uniform matrices. In the fragment shader, a Blinn-Phong lighting model is applied.

5.5.2 Particle Rendering

Most of the visualizations in GLVisualize are realized via instancing a mesh primitive. So the bar plot is nothing else than a cube placed in a grid, with scaling information that get applied to every individual cube. The surface plot is a quad or any other 2D mesh spaced across a grid while the vertexes are projected onto a height field. The vector field is a mesh placed on a 3D grid inside a cube while the rotations from the direction vector get applied to each individual mesh. Even text rendering functions in the same way. The difference is, that the particle not only holds position information but also indexes into a texture atlas in which renderings of the glyphs are cached. So when rendering the text particles, the exact scale and image of the glyph is queried, which will then be used to render a quad with the image of the particular glyph to the screen. The texture atlas approach was chosen, because rendering a high-quality vector graphic is very time-consuming, especially if the description of the font is only available as a Bézier spline. A more detailed description can be found in chapter 5.5.3. All particles are rendered via OpenGL's instanced rendering API, which allows to render millions of particles with only one draw call and very little memory usage, as the geometry of the particle just needs to be uploaded one time. For every individual particle, additional information like color, position, scale and so forth can be queried from within the fragment or vertex shader stage. This additional information can be stored in uniform buffers, uniform arrays or textures. Textures and texture buffers have been chosen for this thesis, as they offer the greatest support among devices and are easy to use. In the future, other approaches can be implemented, gaining more performance or flexibility.

5.5.3 Vector Graphics Rendering

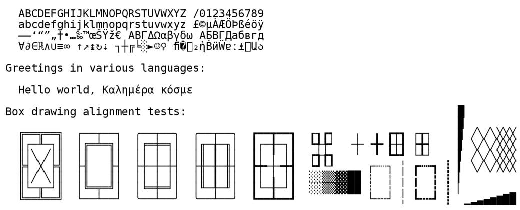


Figure 8: *GLVisualize's fast UTF8 rendering with the help of FreeType, a texture atlas, and 2D particles.*

Vector graphics are difficult to render, as they are not a good fit for the GPU. Long stretched, curved and thin lines introduce several problems for the GPU[47]. Another problem is that splines used in vector graphics are usually supplied as Bézier curves, which are very demanding to rasterize. Besides from that, anti-aliasing of thin lines introduces problems. With post-processing anti-aliasing techniques, lines which are thinner than one pixel will introduce artifacts, as OpenGL primitives smaller than a pixel will get discarded by the OpenGL pipeline. So additional care needs to be taken in order to assure, that primitives are always thicker than one pixel, or multi-sampling techniques have to be used. This is only a very short summary of the problems, which is only given to illustrate that this is not a problem that can be solved in the scope of this thesis. Instead, FreeType[46] is used for rendering fonts. As FreeType is relatively slow, these renderings get cached in a texture atlas from which they can get queried and rendered to the screen in large numbers. This results in high quality and fast renderings. This comes at the cost of higher space requirements and resolution dependence. So when zooming into the vector graphics, either a new version has to be rendered or one gets pixelated results. In the future, distance fields can be used to reduce the resolution dependence[28].

5.5.4 Volume Rendering

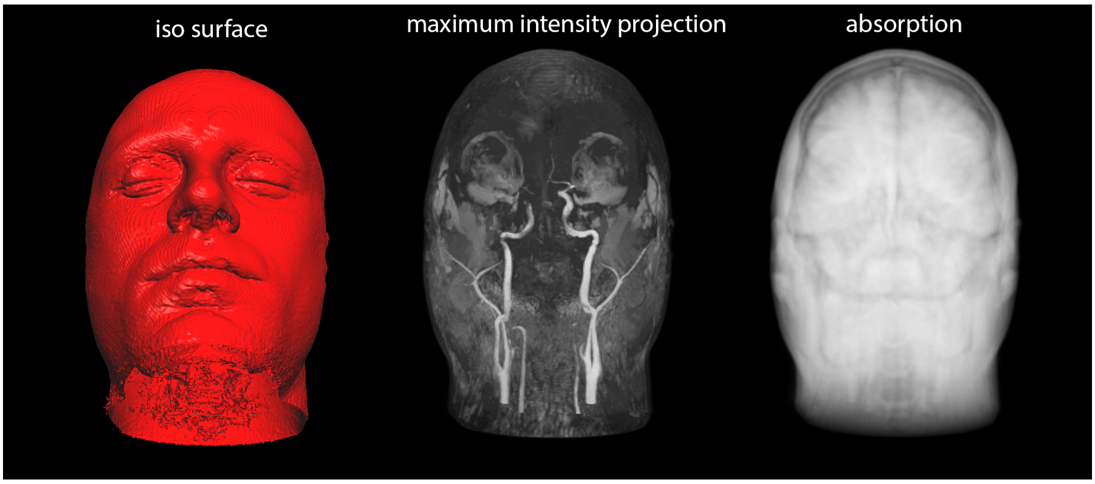


Figure 9: *Different visualizations rendered with GLVisualize. As can be seen, every method misses critical features of the volume.*

Volumes in 3D computer graphics are usually represented as voxels, which can be understood as 3D pixels. They represent values on a 3D grid. The name stems from the marriage of volume and pixel. There are sparse and dense storage options for voxels and different information can be represented, like speed, rotation, color, intensities and so forth. Displaying volumes is not a trivial task. Not only from a computational point of view, but it is also difficult to make a visualization which shows the inside of a volume without hiding the surface. This is illustrated in Figure 9. This is why different forms of visualizations and interactivity are needed to get a good representation of a volume. GLVisualize is able to render iso-surfaces, maximum intensity projections and an absorption based visualization form. On top of this, the particle systems can be used to give further insights into the volume by rendering particles for each voxel. This is especially helpful for sparse volumes.

For the Volume rendering, two techniques are being used. Marching tetrahedra and ray casting[49]. Marching tetrahedra is an algorithm that can be used to extract a mesh representation of an iso-surface from a volume. Ray marching is the process of shooting rays from the camera origin through the object. With every step inside the volume, values are sampled and depending on the combination of these values, different visualization forms can be realized. When stopped at a certain value, iso-surfaces can be rendered. If only the maximum is kept, it will yield a maximum intensity projection. When all values are combined via an absorption function, the volume will be displayed as if it is made of some translucent material like smoke.

The ray-casting rendering method was explicitly implemented for this thesis on the GPU.

For the marching tetrahedra algorithm, there was already a Julia implementation available in the package Meshes.jl[63]. The resulting mesh can be displayed with GLVisualizes mesh rendering capabilities. Both techniques have their advantages and disadvantages. While marching tetrahedra is relatively slow, it can be used to generate a mesh which is very fast to display. Ray casting on the other hands allows for a wide range of visualization forms and is fast enough to calculate it for every frame. The downside is, that the calculation cannot be cached camera position independent. But it allows to interactively change the iso values and all the other parameters, making it ideal to explore a volume.

5.6 Scene Graph

The scene graph is not a specialized data structure, but rather a list of objects, which can be directly rendered with OpenGL. Functionality from Reactive, GLAbstraction, GLVisualize and GLWindow are involved in this. GLVisualize creates a renderable object with GLAbstraction, which will get pushed into a render queue in the screen from GLWindow. Everything that moves is handled via signals from Reactive. Asynchronously updating the signals and iterating over the render object list is achieved with Julia's simple asynchronous API. It is not truly multi-threaded, but rather creates a producer-consumer structure. It could be multithreaded, but this is more difficult to implement as OpenGL is not thread safe. This is an extremely simple form of a scene graph, which does not allow to perform any optimizations. Optimizations usually include sorting in order to reduce OpenGL state changes and culling of invisible objects. As GLVisualize produces OpenGL code which relies on very few calls to OpenGL, the first optimization is currently not as important. But culling can make a large difference for big 3D scenes, as usually only a fraction needs to be rendered. As this is a more involved process, which would preferably be done completely on the GPU, this was not in the scope of this thesis.

5.7 3D Picking

3D picking is the process of selecting a 3D object from a 2D projection like it is the case when one selects a 3D object on the screen with the mouse. It forms the basis for any mouse interaction with objects displayed on the screen. The two most usual approaches are ray picking and color picking. For the ray picking approach, a ray is sent from the 2D position of the camera plane into the 3D scene and is tested for intersection with every object in the scene. In contrast, color picking works with an extra render target, which stores an object id for every pixel, which gets written together with the color pass inside the fragment shader. Color picking has been chosen for this thesis, as it is far easier to implement. Ray picking must be implemented on the GPU, best with some data structures optimized to do fast ray intersections. Otherwise, it will be very slow. If done on the CPU, the 3D objects need to be kept in video memory and CPU memory, further introducing complexity and bottlenecks.

For color picking, two framebuffer render targets need to be created. One for the color channel and one to represent the object id plus an additional number to store contextual information. The additional number is usually used for the instance index, which can be used to e.g. infer what text glyph is selected. When rendering, the fragment shader does not only write the color into the render target but if the color is opaque also the current object id. This way transparency aware 3D picking can be achieved without an extra processing step. The advantage of this methods is that one does not need an extra pass over the geometry. The disadvantage is higher space requirements and that OpenGL's native anti-aliasing does not work well together with the additional render target. Also, the OpenGL pipeline has to be flushed in order to read from the framebuffer. The anti-aliasing problem can be solved by implementing an extra anti-aliasing post processing step. A very simple FXAA algorithm has been implemented, which does not give perfect results but is fast and was easy to implement. The algorithm was taken from Matt DesLauriers[21] and was adjusted to fit into GLVisualize's pipeline.

5.8 Romeo

So far Romeo[13] 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 text 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, one 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 one types in a simple variable, the variable will be visualized. But one can also search and transform a variable via simple Julia terms. In Figure 18 a screen shot of Romeo can be seen. On the left is a window for editing scripts. The middle is used to visualize bound variables, in this case, the variable *barplot* is visualized. On the bottom of the middle, the variables can be selected via a text field. The text field allows to execute code, so one can do things like filtering an array which then will display the filtered array. On the right all variables of the visualization can be edited interactively. If one clicks on the color circle, for example, a color chooser will pop up which will let one chose the color. The numbers are sliders, so when one clicks and drags them, the value can be adjusted. While changing the values, the changes will be immediately displayed.

6 Results and Discussion

In this chapter, it will be discussed if the goals of the thesis have been reached. For that purpose, the methods presented in the Requirement chapter will be used. We will start by analyzing the performance. Afterward, the extensibility of the software will be discussed. The chapter finishes with a short analysis of the usability of the introduced APIs.

6.1 Performance Analysis

In order to analyze how close this thesis comes to achieve the desired performance, we need to look at some benchmarks. If not stated otherwise, benchmarks were 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 Visual Studio Express 2013 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[14].

6.1.1 ModernGL

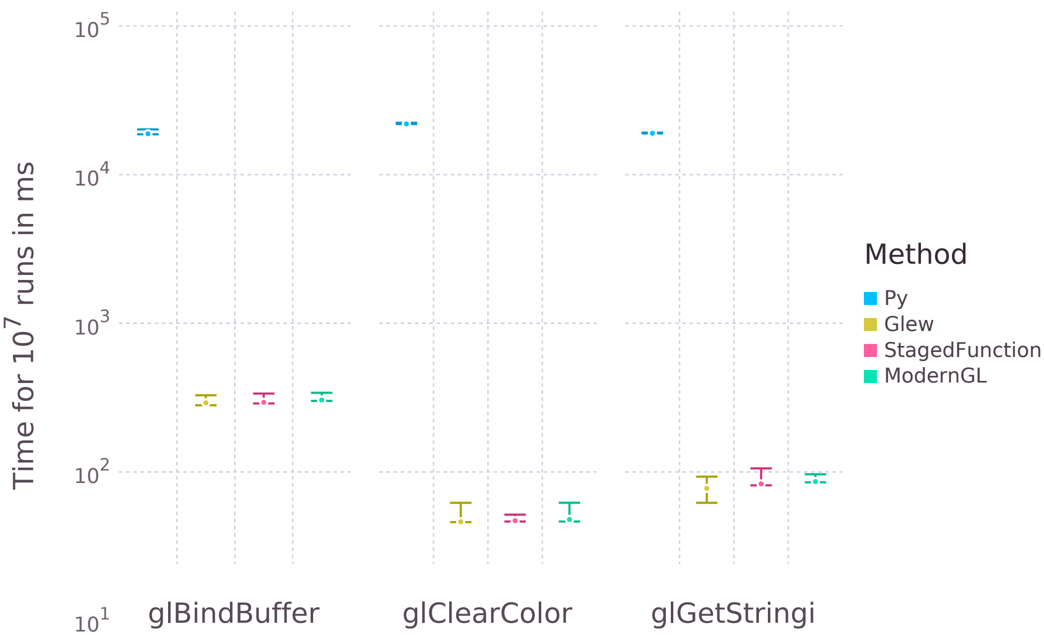


Figure 10: *Different performance of OpenGL wrappers. The time for* 10^7 *calls was measured 100 times for each function.*

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. The execution time of the

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)*

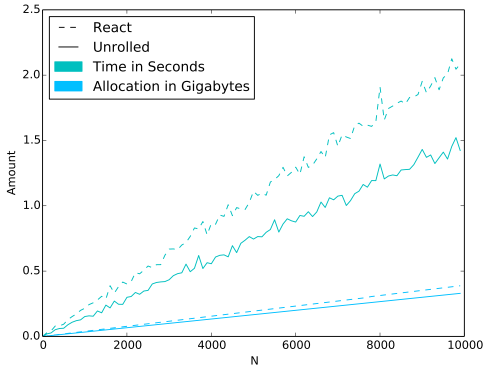


Figure 11: *Complicated graph, fast math*

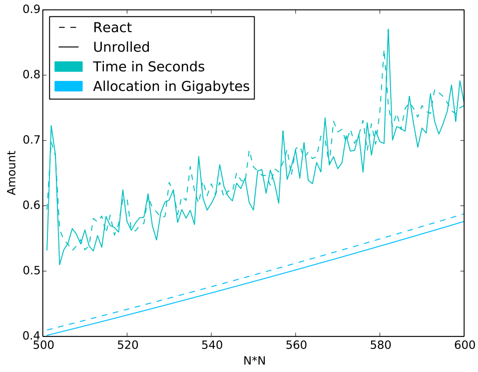


Figure 12: *High memory, simple event graph*

loop got measured. The results are plotted in Figure 10. ModernGL does pretty well compared to C++. Python comes out very slow, with being up to 470 times slower in the case of glClearColor.

In contrast, Julia offers nearly the same speed as calling an 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 researched that thoroughly. From this benchmark can be concluded, that Julia offers a solid basis for an OpenGL wrapper library, which is a crucial prerequisite for writing a visualization library.

6.1.2 Reactive

It is relatively hard to benchmark the used event system in real world scenarios. To compare Romeo's performance with different event systems, one would have to reimplement Romeo for every benchmark. 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 15, small mathematical operations with a complex event graph have 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 adds some overhead.

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

6.1.3 3D Rendering Benchmark

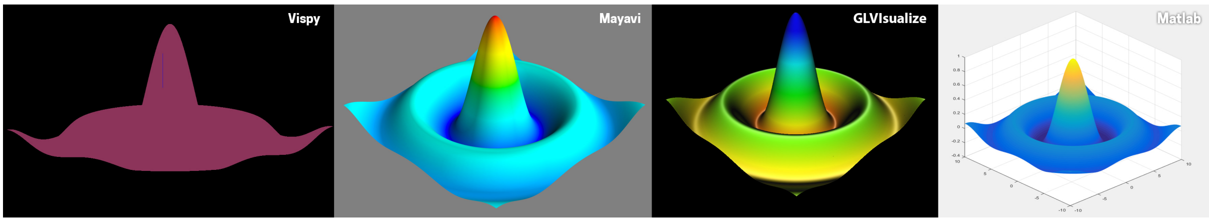


Figure 13: *Different visualizations of the same surface*

The biggest problem with benchmarking the 3D rendering speed is, that there is no library which will allow one to exactly reproduce similar conditions and measures. Additionally, without extensive knowledge of the library, it is difficult to foresee what gets benchmarked. Vispy shows, why it is difficult to rely on measurements like the frame rate. When measuring the framerate, it will show very low frame rates, as it only creates a new frame whenever the camera changes. Romeo on the other hand, has a fixed render loop, which renders as many frames as possible, leading to an entirely 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 to perceive 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 anymore. The last recorded pleasant threshold is then the result of the Benchmark.

The first benchmark is an animated and a 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. Unit is the dimension of the Matrix (N) for which rendering speed was still acceptable.*

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 that the normals were calculated on the CPU resulting in a major slow down[24]. The same can be expected from 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.

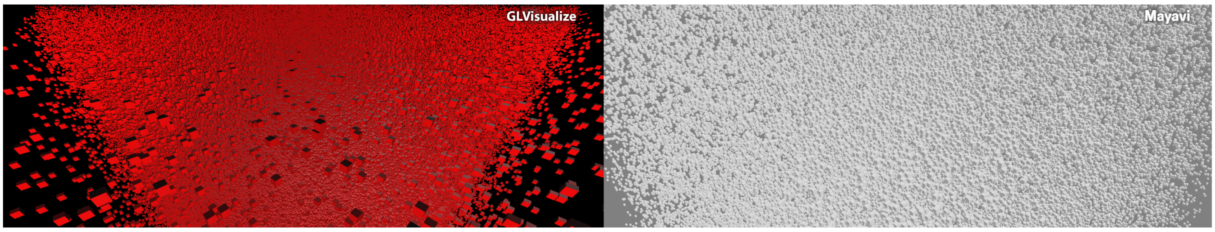


Figure 14: *Rendered particles*

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 a particle primitives and Vispy only had an example, where one needs to write your own shader, which cannot be seen as a serious option. This is a benchmark for easy to use and high-level plotting libraries. We want to find out how well one can solve a problem with the tools that 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 GLVisualize uses OpenGL's native instance rendering. It cannot get much faster than that. The next level of optimizations can be culling, which would only give an advantage at certain zoom levels.

6.1.4 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 it is hard to reach optimal performance for interactive graphics with IJulia. First of all, IJulia uses ZMQ to bridge the web interface with the Julia kernel. ZMQ[30] 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 cannot compete with the native performance of staying inside one language. This is not very important as long as there is not a lot of communication between Julia and the IPython kernel. This changes dramatically 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 OpenGL. While the execution speed of OpenGL can be expected to be similar, there are a lot of new techniques missing, which could speed up rendering. Also, interoperability with OpenCL and CUDA backends ranges from difficult to impossible. This makes it impossible to further accelerate massively parallel tasks.

A benchmark was created to get a sense of what the current state of 3D rendering in IJulia is. It is between Romeo and Compose3D[57], which was the only library found to be able to display 3D models created with Julia directly inside 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 in which one can easily create and display interactive 3D graphics in the IJulia or IPython notebook. This benchmark creates a Sierpinski gasket and Compose3D displays it in the IJulia notebook while Romeo displays it natively in a window.

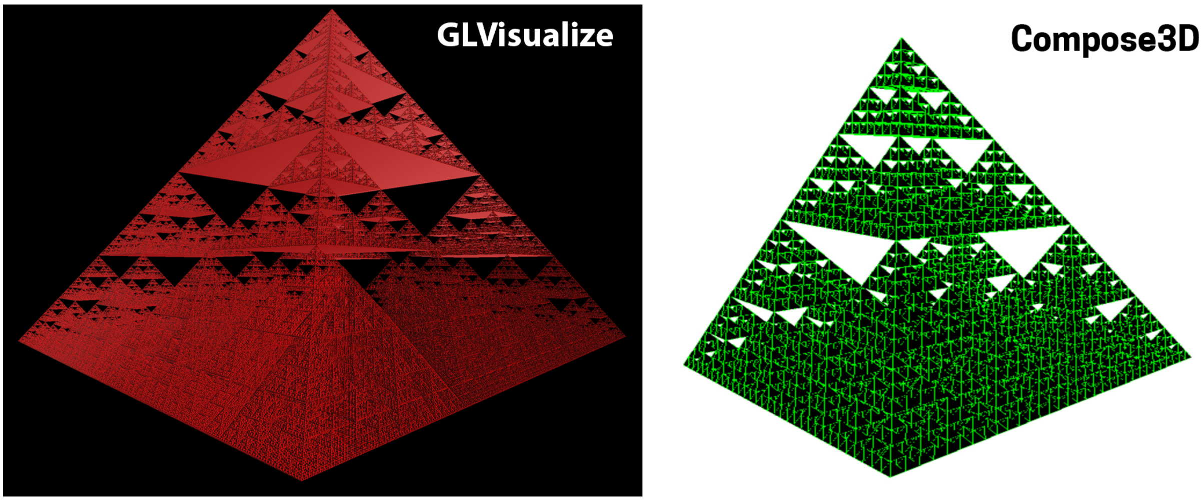


Figure 15: *Sierpinski pyramid in 3D*

Library	**Still**
Compose3D	15625
Romeo	1953125
Speed up	125x

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

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 reusability has already proven itself. Most of the created modules are used independently by different people. GLVisualize is used at least for two other packages besides Romeo. One is GLPlot[18], a scientific plotting package for Julia and Osmosis[15], which is a chat client using GLVisualize for rich media display. See Figure 19, to get an idea of how Osmosis looks like.

GLVisualize is also used by a 3D printing startup[2] to do animated renderings of distance fields. They use it for stereolithographic 3D printing. The distance field renderings get projected into an ultraviolet light curable resin, hardening only specific parts of the polymer.

ModernGL and GLAbstraction are also used by other projects as can be seen in the

Github usage statistics. This indicates, that the abstraction and modularity are well designed so that all the modules can be used alone.

GLWindow makes an exception, as it has been used only indirectly through the other packages. This can mean three things. Firstly, 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.

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. This definitely needs to improve in the future to fully unfold the potential of 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 cannot be written in Julia. But thanks to all involved packages being mainly written in Julia, it is possible to dive into the code base and extend it where needed without switching the language and difficult compile processes. 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 the mentioned software will be analyzed in more detail.

The language usage statistics and infrastructure supplied 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 extendibility of a software. Using different languages for a project can make sense if the project has different domains where domain specific languages give an advantage. As this means that one needs to be fluent in all used languages, this still introduces complexity, but it is at least justified complexity. Together with the total number of lines of code these statistics can give a sense for the complexity of the software, which directly influences the ease at which the library can be extended and rewritten. This is a crucial feature for a 3D visualization library, as it makes it easier to stay state of the art in a field that changes in a rapid pace.

6.2.1 Romeo and GLVisualize

In GLVisualize, every customization can be done from within Julia. Everything like native OpenGL context handles and OpenGL memory objects is directly accessible. One

Language	files	blank	comment	code
Julia	101	1231	281	5499
GLSL	28	273	114	1090
SUM:	132	1504	395	6641

Table 7: *Repositories written for this thesis, including GLVisualize, GLAbstraction, and GLWindow*

can implement even advanced features like OpenCL context sharing with GLVisualize[58]. This enables the users to do e.g. physics simulations in OpenCL on the GPU and visualize the data directly without going through the Central Processing Unit (CPU) which would introduce severe bottlenecks. This is made especially easy, as all visualization functions are defined on the GPU memory objects. So if one already has a GPU array from some other function, it can be directly visualized without additional memory transfers. But even in the worst case, if the whole architecture is not well fitted for the task, reimplementing large parts would not be difficult as there are only 6641 lines of code which are mostly written in Julia as can be seen in table 7. If it comes even worse and the language itself has to be modified for more advanced features, this will pose no problem as Julia is open source. This allows to implement even advanced features like running Julia code directly on the GPU[8]

6.2.2 IJulia

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 8: *]*
Technologies used in IJulia. Statistics are taken from Github

IJulia is written in Julia and relies on ZMQ and IPython. ZMQ is written in C++ to make it as fast as possible. IPython uses multiple JavaScript rendering backends like Three.js and D3. The graphical notebooks are running inside a browser, which adds another complex technology stack. One the plus side, most technologies used are open source, which, in theory, would allow for customizations of the entire stack.

IPython is written in Python, so using IPython from Julia forces the implementation to already use two similar languages. JavaScript, CSS, and HTML are only needed to layout graphics inside the browser. CSS and HTML can be seen as a domain-specific languages for creating GUIs, so while introducing yet another language, they bring the value of easy GUI creation. C++ offers similar functionality to JavaScript and Python, so it can be assumed it was used for speed and interoperability with other C++ libraries. So IJulia builds upon a very heterogeneous technology stack, making it difficult to add advanced features. From inside Julia, there are no handles to the actual rendering process. These are written mainly in JavaScript and Python. Further rendering extensions which do not rely on combining functions from other libraries are only available by emitting HTML and Javascript code from Julia which then get executed in the browser. While modern Browsers seem to offer context sharing between WebGL and the Web Computing Language (WebCL) (the counterparts for OpenGL and OpenCL), it must be done from within JavaScript.

All in all, from within Julia there are little options to implement completely new, advanced rendering features.

6.2.3 Mayavi and VTK

Language	files	blank	comment	code
Python	541	15430	24414	47673
C	1	202	354	1913
make	1	15	4	91
Cython	1	25	45	72
Fortran 77	1	21	28	60
HTML	1	10	0	54
YAML	1	0	5	31
SUM:	547	15703	24850	49894

Table 9: *Mayavi language statistik*

Mayavi itself is mostly written in Python, which amounts to an impressive 49,894 lines of code. This is quite a lot, considering that this does not include rendering code, which is done in VTK. As can be seen in Table 12, VTK amounts to a total of 2,812,005 lines of code written in approximately 27 languages. So the main problem for extending VTK and Mayavi's core will be its complexity. If one is willing to switch to C++, VTK will allow to implement any advanced feature. From inside Mayavi and Python, one will have problems with accessing the underlying rendering core of VTK and one will also run into performance problems with Python. But if performance is not a problem, one can extend Mayavi by writing filters and new sources[31].

6.2.4 Matlab

Matlab is closed source, which makes the core of Matlab impossible to extend by the user. So a lot of the mentioned extensions are not possible to begin with. To extend functionality which needs to modify the language or the core library one will have to ask

Mathworks to do that. This is why Matlab relies on a plug-in architecture which enables developers to write closed or open source plug-ins for Matlab. Simple packages can be written in Matlab itself and imported into a project. This is not an option for performance critical plug-ins, though, as the pure Matlab language is too slow. So something like a 3D visualization library should not be written in pure Matlab code. For that, a mex plug-in is best fitted, which is Matlab's way of letting one write C/C++ plug-ins. It is a very simple process of including a Matlab header file into the C/C++ source, which can then be compiled with Matlab. The header file holds types, which can be used to make the C/C++ functions work with Matlab's native types. It also holds the *mexFunction* macro, which needs to be used in order to make the function callable from Matlab. So it seems to be fairly easy to integrate C/C++ code, but it also defies the purpose of writing in a scientific programming language. Also, Matlab's visualization core itself is not extendable. This means that for an advanced task, one needs to start from scratch or call out to some third party library.

6.3 Usability Analysis

Doing a broad user survey was out of scope for this thesis. This is why the usability study was 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. The main user-facing API, namely GLVisualize's will be analyzed for this thesis. The named aspects will be analyzed together with feedback from Github.

6.3.1 GLVisualize

GLVisulize has a very simple API, as it offers only four functions: *renderloop*, *visualize*, *visualize_defaults* and *edit*. There are also the functions *bounce* and *loop*, which offer 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 if this simple interface still allows people to create the visualization they want. At closer inspection one can see, that visualize is overloaded for 67 Julia types, with each of these methods having a set of keyword arguments which enable further customization. These can introduce drastic changes. The particle visualization, for example, can take any mesh as a primitive. This allows a level of customization, which was not possible 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 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 one 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 cannot 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 cannot be done easily or the format is not intuitive for the programmer. To be fair, this problem exists for every kind of visualization API. In the end the only difference is, how easy it is to do the data transformations. Let us 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 one works with mesh data, one will often handle the face and vertex list in isolation. So an API that offers a function like `visualize_mesh(x::VertexList, y::FaceList)` will be more straightforward for the programmer. This is actually the standard way of displaying a mesh in most scientific plotting packages. It usually goes under the name `patch` and it has become one of the most occurring questions on Github, even though that there are usage examples for displaying a mesh. So diverting from standard plotting interfaces might be the biggest usability issue for GLVisualize. This is not necessarily an issue, though. It is quite easy to offer a compatibility package, which defines functions like `visualize_mesh(facelist, vertexlist) = 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.

The functionality of GLVisualize is easy to explore. There are only three functions to be remembered. From there it is easy to find out how they work. First, the user can just call visualize on your data and see if the default is already sufficient. If not, `visualize_defaults` can be called on the data, to find out what kind of default parameters there are, which can then be tweaked. As an alternative, one can edit every parameter of the visualization on the spot via the widgets. If that is not enough, a list of the data types that can be visualized can be queried with Julia in builds.

7 Conclusion

In this work, a fast 3D visualization library for scientific computing has been presented. Building upon this, Romeo was implemented, a scripting environment specialized on interactive visualizations. The focus for all packages is not only on speed but also usability and reusability. This is why some drastic design choices were made. One of the most important decision was to write the library in Julia. This is a risky decision as Julia is a very young language, which is still in the alpha release phase. Because of this, there are not as many IDE's readily available, the language is still in flux, debugging is harder and the packages are not that mature and stable as in older languages. Besides these problems, Julia turned out to be surprisingly stable and well suited for the task. Using Julia brought the huge advantage of writing in a high-level language without sacrificing speed. That the advantages of Julia play out can be seen in the analysis chapter. It turns out that the developed packages offer state of the art speed while being easy to use and extendable. This is mostly thanks to Julia and a modular design, which made it possible to write the whole visualization library with very little code, making it easy to overlook and reducing complexity. A simple interface for creating animated and interactive visualizations has been implemented by relying on signals for all data involved in the visualization. It makes it very easy to animate every parameter of the visualization while keeping a concise code base. Together with the edit widgets, GLVisualize enables users to change all parameters of a visualization interactively. This is used to implement Romeo to create an interactive scripting environment which enables people to execute scripts and visualize and interact with the variables. The fact that it was possible to also implement other software packages like the chat client Osmosis along Romeo with only little effort are proof that the chosen abstractions and interfaces function well. To enable more people to extend Romeo or create other libraries building upon GLVisualize a lot more work has to be done. One important first step is to improve the documentation. Also, the dependencies are difficult to install on some platforms. This indicates that a lot more testing and bug fixing needs to be done in order to create a pleasant user experience. The rendering speed turned out to be very competitive compared to other established packages, but there is still more room for improvement. More complicated benchmarks with complex scenes have not been benchmarked and it will be interesting to see if there will be large performance drops due to the simple scene graph that does not do any higher level optimizations yet. As multithreading is not yet implemented, severe issues turn up when a task is computational heavy. This scenario will currently result in a freeze of the whole application. This is definitely not acceptable for a serious visualization library.

All in all, it has been shown that Julia and Romeo build a steady foundation for scientific projects with the need for highly performant and interactive 3D graphics.

7.1 Future Work

As already mentioned, there is a lot to do in order to make all packages more pleasant to use. The scene graph is very simple, there is no real multithreading support, the anti-aliasing is sub-optimal, a lot of important features are missing and the OpenGL version used is just mildly modern, compared to the features released recently. Staying state of the art will be a big challenge, especially while keeping downward compatibility for platforms that do not support the newest feature set. One of the most exciting possibilities though is to make parts of Julia compile to the GPU directly. This becomes possible, as the Khronos Group recently released the Standard Portable Intermediate Representation - Vulkan (SPIR-V), an intermediate representation which OpenGL and OpenCL can compile to in the future. The Khronos Group plans to release a converter from LLVM IR to SPIR-V, which would make it possible to program OpenGL and OpenCL alike functionality in any LLVM based language[36]. This means, that all the code that is currently written in GLSL could be rewritten in Julia. On top of that, some functionality that is currently written in Julia and runs on the CPU could run on the GPU in the future. This includes functionality which would profit from highly parallel execution, like ray picking, calculating normals and culling objects. This would further unify the platform, improve speed and will lower the complexity of the library. Such a unique feature is currently not implemented in any other language and could be a game changer for high-performance visualization libraries. With possibilities like these, one can look forward to a bright future for high-performance graphics in Julia.

8 References

[1] Aaalexandrov. Aaalexandrov's github page. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpYD5CyV`.

[2] AddSub. Tools to make your ideas a reality! Accessed: 09/07/2015 : `http://www.webcitation.org/6Zu0xE0rq`.

[3] Apple. A modern programming language that is safe, fast, and interactive. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpOHva47`.

[4] Open Benchmarking. Llvm clang 3.6 compiler tests. Accessed: 04/15/2015 : `http://www.webcitation.org/6XoVc1S0y`.

[5] Jeff Bezanson, Jiahao Chen, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Array operators using multiple dispatch: a design methodology for array implementations in dynamic languages. *CoRR*, abs/1407.3845, 2014.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014.

[7] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[8] Valentin Churavy. Compiling julia to opencl spir instead of opencl c. Accessed: 17/07/2015 : `http://www.webcitation.org/6a5ns8KAQ`.

[9] Clang. clang: a c language family frontend for llvm. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpNGZVlR`.

[10] Palladium Consulting. Scientific and business computing for technology leaders. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpIgxe5w`.

[11] Evan Czaplicki. Elm. Accessed: 05/22/2015 : `http://www.webcitation.org/6YiVPsavP`.

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

[13] Simon Danisch. 3d visualizations and editing in julia + opengl. Accessed: 21/07/2015 : `http://www.webcitation.org/6aBwT305q`.

[14] Simon Danisch. Benchmarks. Accessed: 14/07/2015 : `http://www.webcitation.org/6a1bBOJhY`.

[15] Simon Danisch. Chat client for julia. Accessed: 08/07/2015 : `http://www.webcitation.org/6ZrApnEoT`.

[16] Simon Danisch. Create a window with an opengl context. Accessed: 21/07/2015 : `http://www.webcitation.org/6aBwI9wtk`.

[17] Simon Danisch. Opengl 3+ bindings for julia. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpHoUgQl`.

[18] Simon Danisch. Plotting for julia with opengl. Accessed: 08/07/2015 : `http://www.webcitation.org/6ZrAkkm6j`.

[19] Simon Danisch. Utility package for moderngl. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpHzv8m5`.

[20] Simon Danisch. Visualization library written in julia and opengl. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpI9ZS6G`.

[21] Matt DesLauriers. A webgl implementation of fast approximate anti-aliasing. Accessed: 05/31/2015 : `http://www.webcitation.org/6ZpGVIT54`.

[22] Enthought. Mayavi. Accessed: 05/16/2015 : `http://www.webcitation.org/6YZNh5oEC`.

[23] Keno Fischer. The julia c++ foreign function interface (ffi). Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpIgxe5w`.

[24] Github. surface slow and buggy 892. Accessed: 05/13/2015 : `http://www.webcitation.org/6YVE1zeNb`.

[25] GLFW. Free, open source, portable framework for opengl application development. Accessed: 05/22/2015 : `http://www.webcitation.org/6YiVA6PY7`.

[26] Sebastion Good. Little performance explorations. Accessed: 04/14/2015 : `http://www.webcitation.org/6Xmtrox4u`.

[27] Shashi Gowda. Reactive. Accessed: 05/22/2015 : `http://www.webcitation.org/6YiVI2SUy`.

[28] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 CoWurses*, SIGGRAPH '07, pages 9–18, New York, NY, USA, 2007. ACM.

[29] Khronos Group. The khronos group - connecting software to silicon. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpP5nDGX`.

[30] iMatix Corporation. Zmq. Accessed: 14/07/2015 : http://www.webcitation.org/6a1fkmLVK.

[31] Enthought Inc. Extending mayavi with customizations. Accessed: 17/07/2015 : http://www.webcitation.org/6a5rroRG9.

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

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

[34] JuliaGeometry. Generation and manipulation of triangular meshes. Accessed: 07/07/2015 : http://www.webcitation.org/6ZpGGB4uX.

[35] JuliaLang. Ijulia. Accessed: 20/07/2015 : http://www.webcitation.org/6a9zURFZO.

[36] John Kessenich and Boaz Ouriel. *SPIR-V Specification (Provisional)*. Khronos Group, 1 edition, 2015.

[37] Kitware. Vtk gallery. Accessed: 04/15/2015 : http://www.webcitation.org/6XodgQgdm.

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

[39] Julia Language. benchmark times. Accessed: 04/13/2015 : http://www.webcitation.org/6XlX8Wwft.

[40] Rust Language. Rust. Accessed: 04/26/2015 : http://www.webcitation.org/6YoVz6hnx.

[41] Michael Larabel. Compiler intel broadwell linux tests. Accessed: 04/15/2015 : http://www.webcitation.org/6XoVX2L1r.

[42] Michael Larabel. Future work on the amd gpu llvm back-end. Accessed: 04/26/2015 : http://www.webcitation.org/6YoW7fDyt.

[43] Michael Larabel. Intel broadwell: Gcc 4.9 vs. llvm clang 3.5 compiler benchmarks. Accessed: 04/15/2015 : http://www.webcitation.org/6XoW1HeDq.

[44] Michael Larabel. Microsoft announces an llvm-based compiler for .net. Accessed: 04/15/2015 : http://www.webcitation.org/6YOdC964v.

[45] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[46] Werner Lemberg. Freetype is a freely available software library to render fonts. Accessed: 09/07/2015 : `http://www.webcitation.org/6ZtqPHjCA`.

[47] Eivind Lyngsnes Liland. Path rasterizer for openvg, 2007.

[48] M. Lubin and I. Dunning. Computing in Operations Research using Julia. *ArXiv e-prints*, December 2013.

[49] Ricardo Marques, Luís Paulo Santos, Peter Leskovsky, and Céline Paloc. Gpu ray casting. 2009.

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

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

[52] Tanmay Mohapatra. reduce gc load in readdlm. Accessed: 04/15/2015 : `http://www.webcitation.org/6YOc9Qv1T`.

[53] Tom Preston-Werner. Semantic versioning 2.0.0. Accessed: 19/07/2015 : `http://www.webcitation.org/6a9OHQVwH`.

[54] Amuthan Arunkumar Ramabathiran. Finite element programming in julia. Accessed: 04/14/2015 : `http://www.webcitation.org/6XmvHthh5`.

[55] Prabhu Ramachandran and Gaël Varoquaux. Mayavi: a package for 3d visualization of scientific data. *CoRR*, abs/1010.4891, 2010.

[56] The Register. Nvidia ditches homegrown c/c++ compiler for llvm. Accessed: 04/26/2015 : `http://www.webcitation.org/6YoWHmFJF`.

[57] Rohit Varkey Thankachan. A library to try and extend compose.jl to 3d. Accessed: 07/07/2015 : `http://www.webcitation.org/6ZpGc9yrS`.

[58] Simon Danisch Valentin Churavy. A quaterion julia set implementation in julia running on the gpu, with opencl and opengl. Accessed: 17/07/2015 : `http://www.webcitation.org/6a5nyzlUN`.

[59] Bret Victor. Up and down the ladder of abstraction, October 2011. Accessed: 19/07/2015 : `http://www.webcitation.org/6a8vTlioO`.

[60] Bret Victor. Inventing on principle, January 2012.

[61] Bret Victor. Learnable programming, September 2012. Accessed: 19/07/2015 : http://www.webcitation.org/6a8vQRDTy.

[62] Vispy. A python library for interactive scientific visualization. Accessed: 05/16/2015 : http://www.webcitation.org/6YZPFZODy.

[63] Tracy Wadleigh. Iso surfaces. Accessed: 05/24/2015 : http://www.webcitation.org/6Yltap0ze.

[64] OpenGL Wiki. Rendering pipeline overview. Accessed: 04/10/2015 : http://www.webcitation.org/6Xgd8fedi.

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

[66] Wired. The one last thread holding apple and google together. Accessed: 04/15/2015 : http://www.webcitation.org/6YOdawS5T.

Appendix

A IJulia

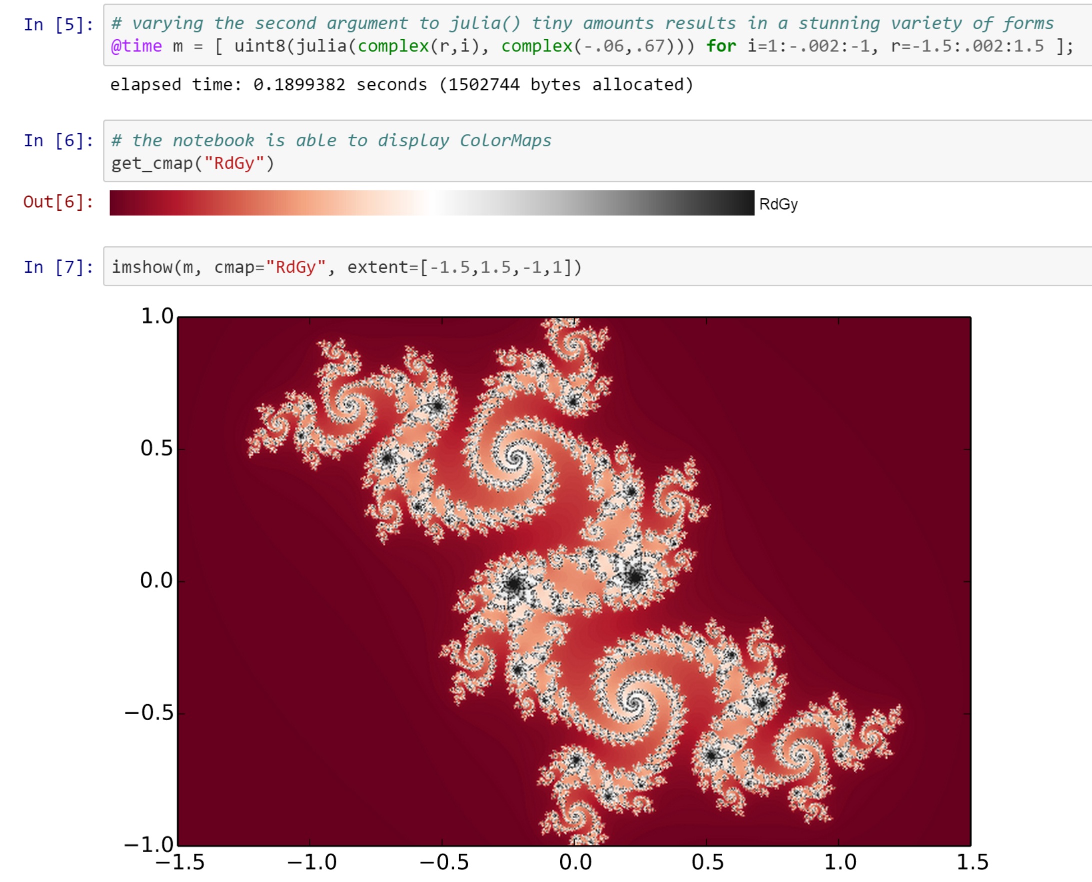


Figure 16: *Example of an IJulia Notebooks. Screenshot taken from [32]*

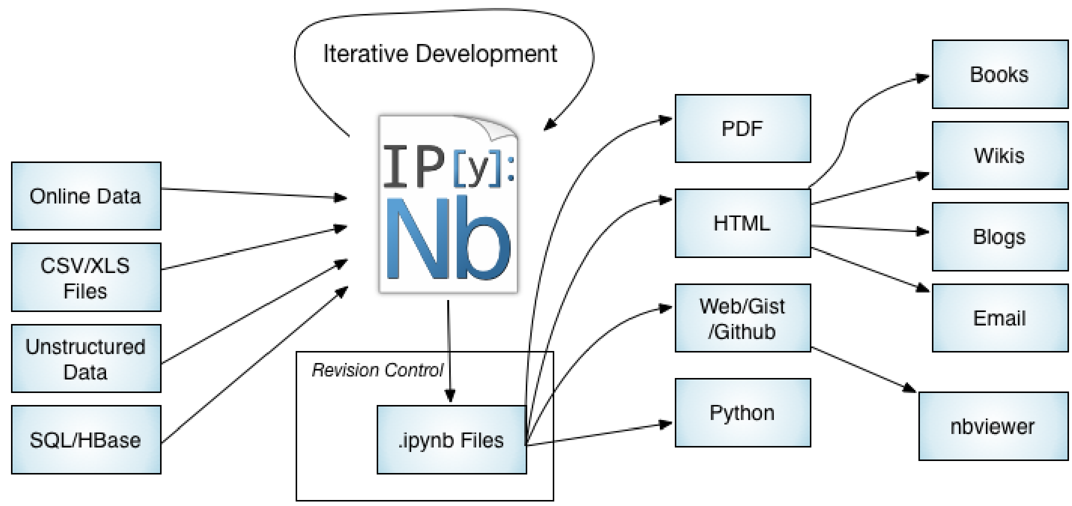


Figure 17: *Workflow of IPython Notebooks. Graphic from Wikipedia [65]*

B Language Statistics

All language statistics have been made with cloc [12] the current master of the github repositories.

Table 10: *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 12: *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

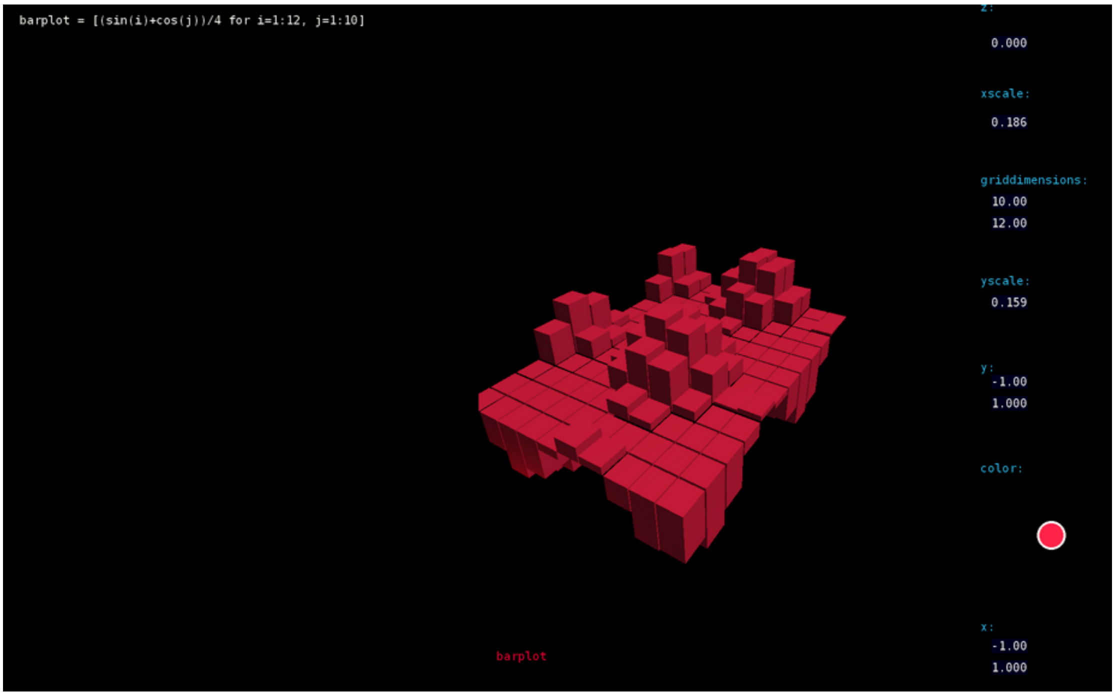


Figure 18: *Screenshot of the prototype. Left: evaluated script, middle: visualization of the variable barplot, right: GUI for editing the parameters of the visualization*

D Osmosis's GUI

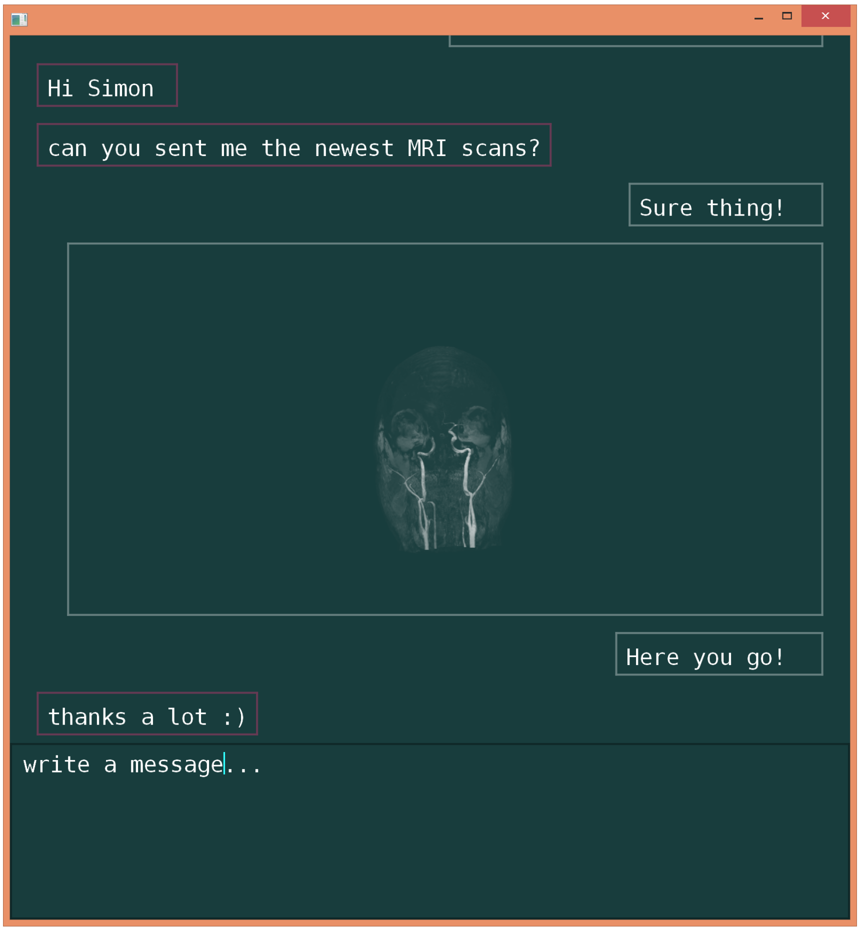


Figure 19: *Screenshot of Osmosis. All visualizations are editable and interactive*

E Benchmark

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 15: *LLVM 3.5 compared to LLVM 3.6 in the Phoronix benchmark test suite[4]*

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 17: *gcc 4.9.2 compared to LLVM 3.5 in the Phoronix benchmark test suite[41]*

Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

Date: ....

(Signature)