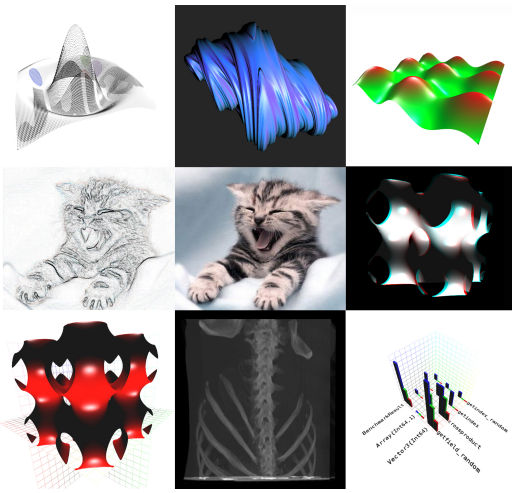
UNIVERSITÄT OSNABRÜCK

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

II Table of Contents

III List of Figures

IV List of Tables

V Listing-Verzeichnis

VI List of Abbreviations

GUI	Graphical User Interface
LLVM	Low Level Virtual Machine
gcc	GNU Compiler Collection
Matlab	Matrix Laboratory
REPL	Read Eval Print Loop
GPU	Graphics Processing Unit
GLSL	OpenGL Shading Language
OpenCL	Open Compute Language
OpenGL	Open Graphics Language

1 Introduction

This Bachelor Thesis is about writing a fast and interactive 3D visualization environment for scientific computing. The focus is on usability, applied to all the different interfaces, ranging from abstract API interfaces to graphical user interfaces. The ultimate goal is to make scientific computing more accessible to the user. As Graphical User Interface (GUI) elements and editable text fields are supplied, one can also write and execute scripts. Using these widgets, all bound variables can be visualized and some of them can be edited interactively. This can be used as a basis for interactive programming or visual debugging, further helping the user to understand his algorithms.

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

1.1 Scientific Computing

Scientific computing is the area of computing, that evolves around all kind of scientific research. It is a very broad field involving a lot of different challenges. In some areas like particle physics, the problems are computationally so demanding, that they can only be solved with the help of super computers. In other areas like robotics, it is important to be fast, while running on embedded systems with small resources. In mathematics, speed does not need to be important, but it can be that the algorithm in itself is very difficult to comprehend. So the more comprehendable it can be written down in a programming language, the easier it will be to actually implement the algorithm. Above all, programming itself is often secondary to the research goal. This means, that it can be expected that the researcher just has rudimentary programming skills and that he does want to put as little time as possible into solving programming problems. So things like manual memory management and difficult design patterns with a lot of boilerplates are to be avoided in scientific computing. This has lead to the rise of programming languages and tools specifically tailored to scientific computing. The most prominent examples include Mathematica, R and Matlab. They all aim to provide simple syntax for linear algebra stastical code, while taking away programatically difficult tasks like memory management. Also, they come with a rich standard library, which means most research can be done without loading any addtional module, which makes them great tools for rapid prototyping. At the current state, the speed of these languages suffer from the high level of abstraction. In order to cater to scientific research which is in need of highly performant code, a lot of

the core is written in another language like C/C++ and Fortran. This leads us straight to the contributation of this thesis.

1.2 Contribution

The goal is to make it easier to understand complex algorithms and data. For that a fast visualization library is needed, which is optimized for dynamic data. It needs to be simple to use, as it would be no use if all your time is spend on visualizing the problem while this time could be used for understanding the problem better. Visualising data that changes over time is very sensitive to latency. If you can not calculate every value in time for a single frame, you either need to skip frames or delay them. This leads to stutters, which, depending on the size, can be unpleasant or completely corrupt the work-flow. Which makes it important, that every routine that is used in the visualization pipeline is as fast as possible. This is only possible, if everything is written in a fast language with as little conversations and memory transfers as possible. If the link between two languages can not be guaranteed to be fast, from this also follows, that everything should be written in one language. This has been achieved by writing Romeo in Julia. As the chosen language is also a high-level language and some effort was put into creating a concise architecture, a further contributation will be, that the development cycles can be very short and the library is easy to extend. Another contribution is the interactivity of the visualizations via simple widgets, which allow you to change the apperance of large datasets without any stutters. Also due to deep language integration and a design that gives every object a default visualization, it can be used to visualy debug your code or interactively write your program while updating your visualization.

1.3 Field of Research and Problem

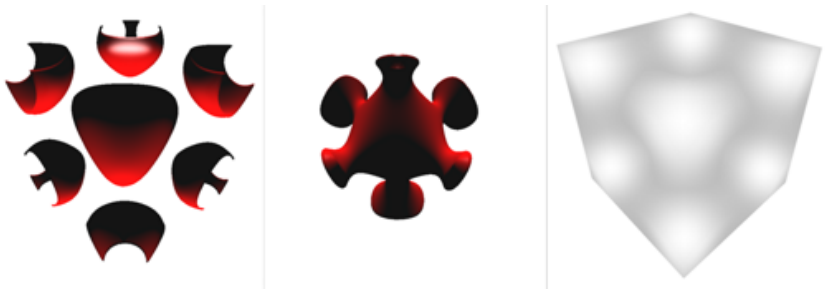


Figure 1: different visualizations of $f(x, y, z) = \sin(\frac{x}{15}) + \sin(\frac{y}{15}) + \sin(\frac{z}{15})$, visualized with Romeo. From left to right: Isosurface with isovalue=0.76, Isosurface with isovalue=0.37, maximum value projection

It is not easy to pin down the correct research field of this thesis. The general research

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

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

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

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

1.4 Outlook

2 Background

2.1 Related Work

2.1.1 The Julia Programming Language

Bringing Julia's ease of use and speed to a dynamic visualization library is the declared goal. So Julia plays a crucial role in this thesis. It is the most important previous work, as much as Julia is the main used technology. This chapter gives a short introduction to the Julia Programming Language.

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

Julia is a multi paradigma language for scientific computing. The focus on scientific computing means, that Julia's standard library is equipped with a lot of functions, data structures and specialized syntax for implementing complex math like linear algebra and statistics. It promises to approach C speed, while being dynamic language which is easy to use. This is made possible by the compile process which can be described as statically compiled at runtime. Julia uses a garbage collector, taking the memory management away from the programmer. There are quite a few things Julia promises to the developer described in the article [5]. These include:

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

These are the reasons why Julia is well suited for implementing an interactive scientific visualization library. Interactive visualizations have very hard demands on performance. You need around 60 frames per second to feel comfortable, so there are only 0.016 seconds available for computing a single frame. There should be no stutters and interfacing C-libraries should be simple and fast in order to communicate with the video driver. This is the one side, but the other is equally important: You can do the scientific programming in the same language as you do the visualization. Like this, Julia's native data types can

be used without serialization and copies and the library can be extended by the Julia programmer. Extending the library is supposed to be a lot easier than in other languages, thanks to Julia's concise and high-level coding style.

2.1.2 IJulia

IJulia is the Julia language back-end for IPython. IPython is a software stack, which was created to allow for interactive computing in Python. It offers an interactive shell to execute python scripts, GUI toolkits, tab completion and rich media visualizations. It comes with a web based notebook, which enables you to write formated documentations together with data, inlined plots and executable program snippets. You can also formulate mathematical formulas in latex, which will get rendered and inlined nicely into an IJulia Notebook. See figure 10 for an example.

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

2.1.3 Matlab

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

Matlab itself is written in C, C++, Java and MATLAB. It's proprietary software with a pricing of around 2000 € [12], which can be extended via free, open source and proprietary modules like Simulink.

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

2.1.4 Paraview and VTK

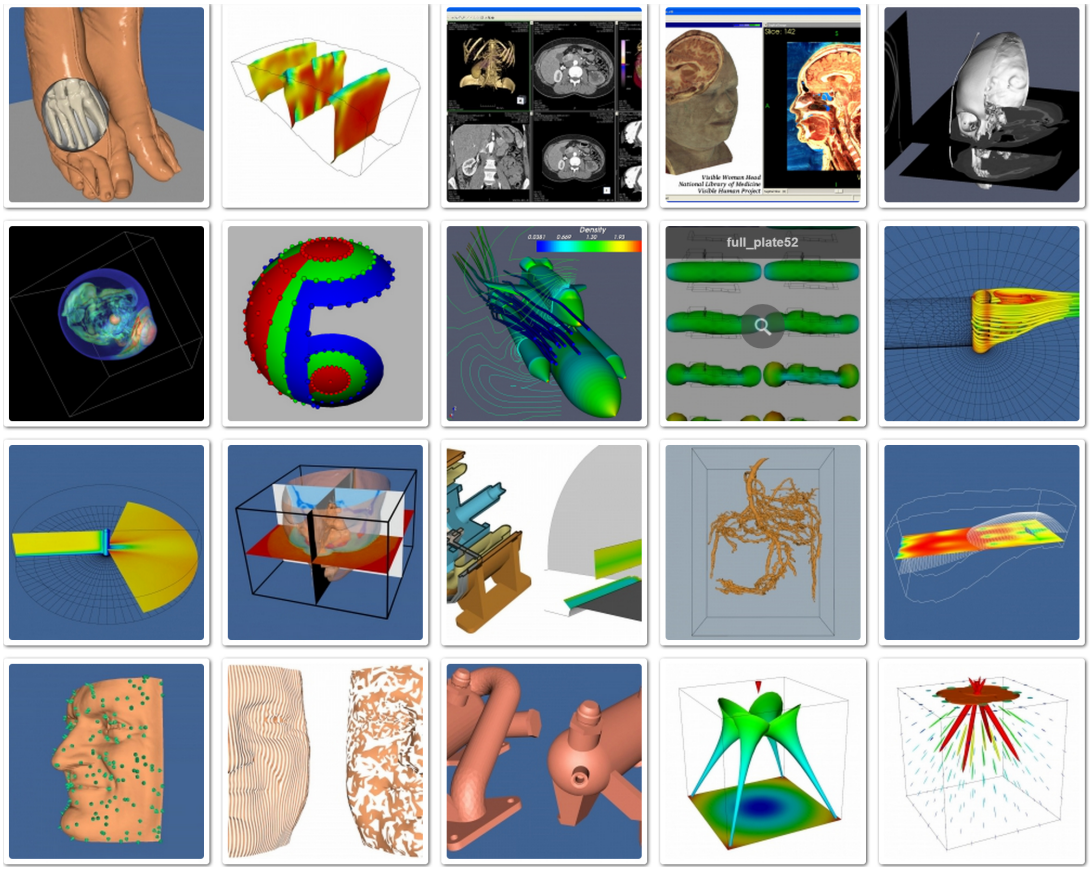


Figure 2: Different visualizations done with VTK.

VTK is probably the most advanced scientific visualization library, with a huge amount of visualization types. In figure 2 you can see some of the visualization forms taken from the VTK gallery[7].

It shares many of its goals with Romeo, namely [6]

- Develop an open-source, multi-platform visualization application.
- Support distributed computation models to process large data sets.
- Create an open, flexible, and intuitive user interface.
- Develop an extensible architecture based on open standards.

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 5 and 7. The biggest difference is, that Romeo is implemented in a scientific programming language, while VTK is mainly implemented in C++. This has two big implications: Firstly, if the language doesn't have native C++ compatible data types and an overhead less C++ interface, shipping a large stream of data to VTK becomes

slow. Secondly, one must know C++ to extend VTK. This makes it difficult to create customized visualizations.

3 Design

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

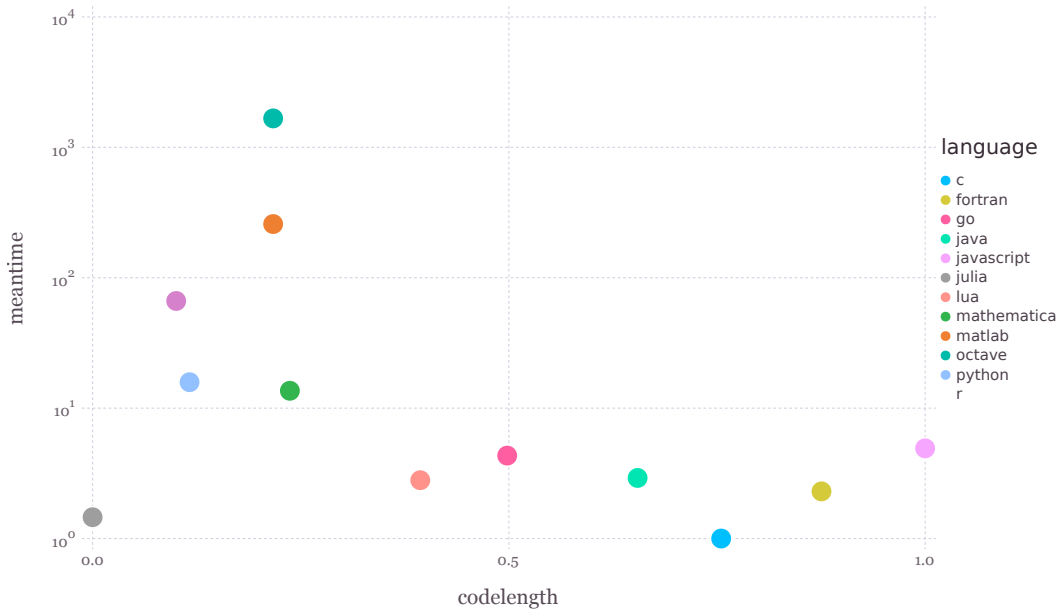


Figure 3: Languages speed relative to C (averaged benchmark results), plotted against the length of the needed code (Source in Appendix).

3.0.5 Speed

Speed is mainly a usability factor. It's a factor, that can make a software unusable, or render it unproductive. Because of this, speed has taken a high priority in this thesis. As general coding productivity is also a concern, this thesis is set on using a high level language. Historically, these two demands can't be satisfied both. How to achieve state of the art speed with a high level language is an ongoing research and basically the holy grail of language design. Luckily, there is a new programming language namely Julia building upon the compiler infrastructure LLVM, promising a concise, high-level programming style, while approaching C-performance. This is well illustrated in figure 6. Code length is an ambiguous measure for conciseness, but if the code is similarly refactored it is a

good indicator of how many lines of code are needed to achieve the same goal. LLVM is an impressive compiler infrastructure, which has front ends for different languages and back-ends for different chip architectures.

LLVM's concept is effective, as you can accumulate state of the art optimizations in one place, making them accessible to many languages, while being able to compile to different platforms. There are x86, ARM, Open Compute Language (OpenCL) and CUDA back ends. While Julia doesn't support them all, it will hopefully be possible in the future. LLVM is also used by Clang, the C/C++ front end for LLVM rivaling gcc and it is used by Apple's programming language Swift. This makes LLVM a solid basis for a programming language, as these are highly successful projects guaranteeing LLVM further prospering of the technology.

To get high performant 3D graphics rendering, there are on the first sight a lot of options. If you start to take the previous demands into account, the options shrink down considerably, though. The visualization library should be implemented in one high level language, which can be used for scientific computing and has state of the art speed. At this point, there are close to zero libraries left. As you can see in figure 6, Matlab, Python and R disqualify, as they are too slow. JavaScript, Java, Go and Lua are missing a scientific background and the others are too low level for the described goals. This leaves only Julia, but in Julia there weren't any 3D libraries available, which means that one has to start from scratch. There are only a couple of GPU accelerated low-level libraries available, namely Khrono's OpenGL, Microsoft's DirectX, Apple's Metal and AMD's Mantel, which are offering basically the same functionality. As only OpenGL is truly cross-platform, this leaves OpenGL as an option. So for the purpose of high speed visualizations, OpenGL was wrapped with a high-level interface written in Julia. This leaves us with one binary dependency not written in Julia, namely the video driver, which implements OpenGL.

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

3.0.6 Extensibility

Extensibility is an important factor, which can decide, if a library is fit for scientific computing or not. It's not only that, but also a great factor determining growth of a software, as the more extensible the software is, the higher the probability that someone else contributes to it. In order to write extensible software, we first have to clarify what

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

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

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

3.0.7 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 immidiate influence on how to handle animations. This lead to the design choice of using signals. Signals are a very good abstraction for values that change over time. If well implemented, it makes it natural to reason about time, without the need of managing unrelated structures and callback code.

3.0.8 Interfaces

Working with a computer means working with interfaces to a computer, which in the end simply jiggles around with zeros and ones. There is a huge hierarchy of abstractions involved, to make this process of binary juggling manageable to the human. We already dealt with the lowest relevant abstraction: the choice of programming language, which forms our first interface to the computer. The next level of abstraction is the general

architecture of the modules, which has been discussed previously. This chapter is about the API design choices that have been made. The first API is the OpenGL layer. The philosophy is to make the wrapper for native libraries as thin and reusable as possible and an one to one mapping of the library itself. This guarantees re-usability for others, as they might be used to work only with the low-level library or they might disagree with some higher-level abstraction and prefer to write their own. Over this sits an abstraction layer needed to simplify the work with OpenGL. With this abstraction, the actual visualization library is implemented. APIs for visualization libraries are very difficult to realize, as there are endless ways of visualizing the same data. The design choice here was to use Julia's rich type system, to better describe the data. Julia makes this possible, as you can create different types for the same data, without loosing performance. So you can have a unit like meters represented as a native floating point type and have the visualization specialize to this. Like this you can have a single function e.g. *visualize*, that does create a default visualization for different data types. So instead of manually passing additional information to the visualization function, it is instead coded in the type itself. Together with the event system which consists of signals, it is possible to edit and visualize rich data over a simple interface, which is perfect for visual debugging, as it is always the same function call applied to the data and no further user interaction is needed. It is also easy to extend, as the user just has to overload the function, with a custom style and optional key word arguments. Finally, there are also graphical user interfaces developed for this thesis. As also optimizing them is out of the scope of this thesis, they are kept very simple. The measurement of success is again relatively difficult to do. (I need to think this over)

4 Used Technologies

4.1 The Julia Programming Language

The basic introduction of Julia has already been given in the Background chapter. This chapter is focused on how to write programs with Julia. Most influential language construct are it's hierarchical type system and multiple dispatch. Multiple dispatch is in its core dynamic function overloading at runtime. To better understand multiple dispatch, one has to be familiar with Julia's type system. The type system builds upon four basic components. Composite types, which are comparable to C-Structs, parametric composite types, bits types, abstract and parametric abstract types. While the first three are all concrete types, abstract types can be used to build a type hierarchy. Every concrete type can inherit from one abstract type, while abstract types can also inherit from abstract types. Bit types are just immutable, stack allocated memory chunks, usable for implementing

numbers. You 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 you 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 the function specialized on its arguments which results in overloading the function with the specific arguments. So in the previous example, foo will originally be overloaded with two methods. Now, if you call foo with one Float32 argument, a new method will be added specialized to Float32. Like this, if the function does not access non constant global values, all types inside the function will be known at call time. This allows Julia to statically compile the function body, while propagating the type information down the call-tree.

With multiple dispatch, Julia is definitely a functional oriented language. But there are also ways to give Julia a more object oriented feel. Functions are 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. But the object still needs to be passed to function via the arguments, so there is no self reference available in Julia.

4.2 OpenGL

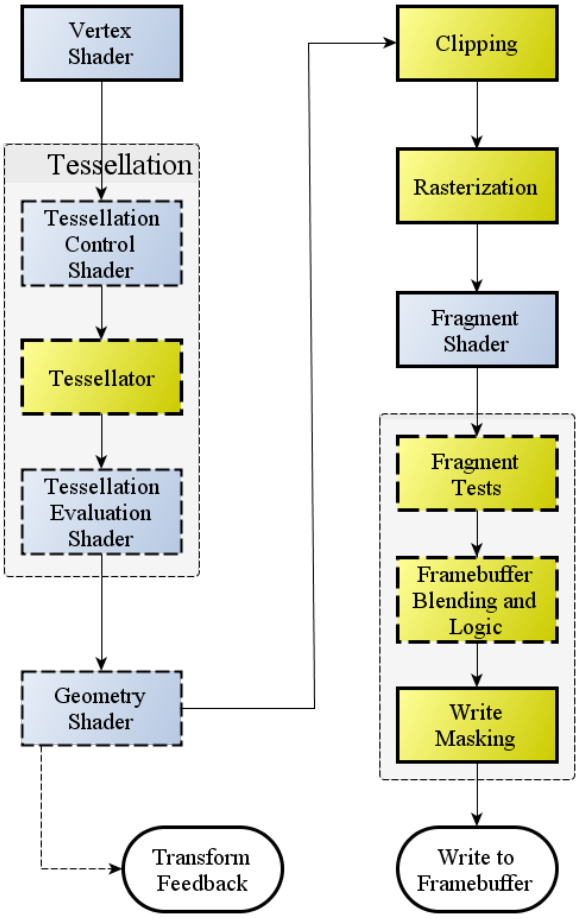


Figure 4: Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages. [16]

OpenGL is a low-level graphics API implemented by the video card vendor via the video driver. As such it doesn't offer much abstraction over the actual Graphics Processing Unit (GPU), but instead offers high flexibility and performance. OpenGL 1.0 was released in 1992 and the current version is 4.5. A critical element when developing OpenGL applications is, that not all video drivers implement the newest OpenGL standards.

As a result, one has to decide which OpenGL version to program against, trading between modernity and platform support. For Romeo, it was decided to support OpenGL 3.3 as the lowest bound, as it is sufficiently available, while still having most of the modern features. The features include instance rendering, vertex arrays and modern OpenGL Shading Language (GLSL) shader.

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

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

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

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

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

4.3 Reactive

Reactive is a functional event system designed for event driven programming. It implements Elm's signal based event system in Julia. Signals can be transformed via arbitrary functions which in turn create a new signal. This simple principle leads too a surprisingly simple yet effective way of programming event based applications.

```
1  a = Input(40)        # an integer signal.
```

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

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

4.4 GLFW

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

5 Implementation

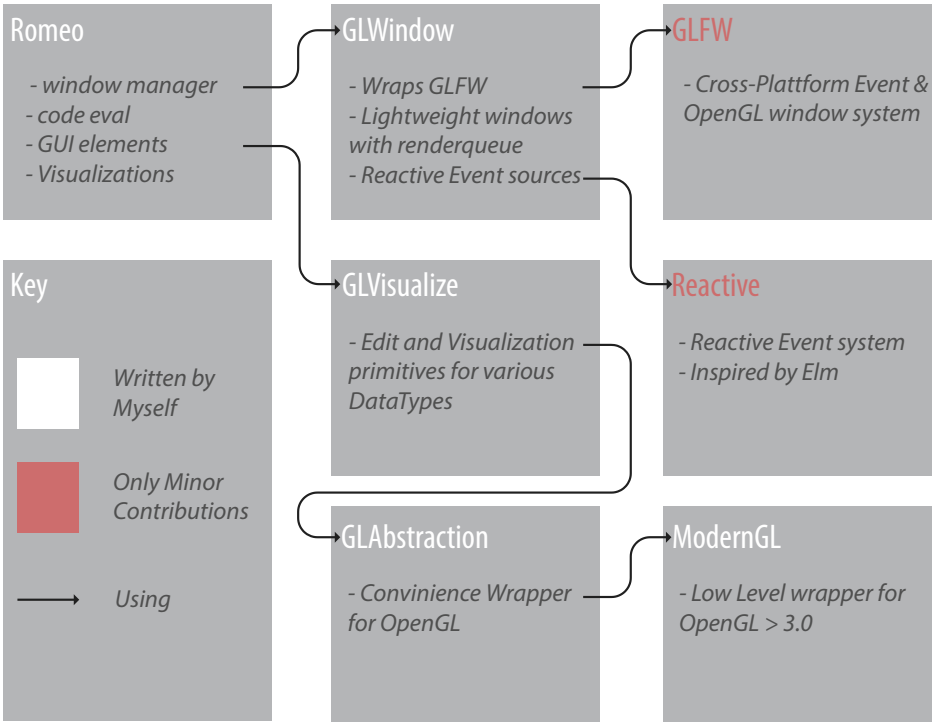


Figure 5: Main modules used in Romeo and their relation (simplified).

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

5.1 Event System

The event system was challenging to integrate for several reasons. First of all Reactive is a functional event system, while OpenGL relies heavily on global states, which are two perpendicular concepts. Also, it doesn't allow to rearrange the event tree. In other words, you can not create sub trees in advance and then fuse them together at run time.

5.2 ModernGL

OpenGL is implemented by the video card vendor and is shipped via the video driver, which comes in the form of a C-Library. The challenge is, to load the function pointers system and vendor independent. Also one further complication is, that depending on the platform, function pointer are only available after an OpenGL context was created and may only be valid for this context. [13] 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.

5.3 GLAbstraction

GLAbstraction is the abstraction layer over ModernGL. It wraps OpenGL primitives like Buffers and Textures in Julia objects and hooks them up to Julia's garbage collector. Additionally, it implements convenient functions to load shader code and it makes it easy to feed the shader with the correct data types. Besides supplying an abstraction layer over OpenGL, it also offers the linear algebra needed for the various 3D transformation and camera code. Building up on that, it defines a signal based perspective and orthographic camera type.

5.4 GLWindow

GLWindow is a lightweight wrapper around GLFW. It mainly offers a screen type, which contains signals for all the different GLFW events (Mouse, Keyboard, etc...). It also offers a hierarchical structure for nesting screens. All the screen areas are signals, which result in the resizing the screen area when they change. This makes it simple to implement windows that react to changing screen areas or resized objects.

5.5 GLVisualize

GLVisualize implements the main functionality of this library. It 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 a lot of methods for different types. This has two advantages. First, it makes it very easy to use for visual debugging, as any value can be displayed immediately without any user interaction. Secondly, the user doesn't have to remember or lookup the function name, as long as there is a default visualization for the type he is working with. The next design goal was to make this fit for dynamic data, which resulted in relying on as little transformation of the data as possible and directly

transferring it to the GPU. Depending on the complexity of the visualization, this means the visualization can be updated with as little overhead as possible.

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

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

With this simple interface, the following data can be visualized:

- Text (Vector of Glyphs)
- Height fields with different primitives (Matrix of height values)
- 3D bar plots (Matrix of height values)
- Images (Matrix of color values)
- Videos (Vector of Images)
- Volumes (3D Array of intensities)
- Particles (Vector of Points)
- Vector Fields (3D array of directional Vectors)
- Colors (Single Color values)

All of these can be integrated into the same scene and it is possible to change their parameters interactively. These interactions can be purely programmatically, or via the widgets from the edit function. It calls the visualize function to render the data type and then registers appropriate events to update the data. Take a look at the text edit function. It first uploads the text to video memory and sets up the functionality to visualize it, and than updates the text data on the GPU according to the cursor position and keyboard input.

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

5.5.1 Romeo

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

6 Results and Discussion

6.1 Performance Analysis

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

If not stated otherwise, benchmarks are written for this thesis and executed on an Intel Core i5-4200U with an HD 4400 graphic and 8GB of RAM. Benchmarks were run on an idle computer with as little background processes running as possible. The sources of the benchmarks can be found on Github.

6.1.1 Julia

Julia's performance is crucial for this thesis. If Julia doesn't perform close to C it would weaken the whole argument of writing the visualization library in Julia. It is a very tedious task to write representative benchmarks for a language. The only way out is to rely on a multitude of sources and try to find analytical arguments. In this thesis, Julia's own benchmark suite will be used in addition to some real world benchmarks from the Julia-Mailing list, where users ported code from some other language to Julia. In addition, the general compiler structure of Julia will be analyzed to find indicators for Julia's overall performance.

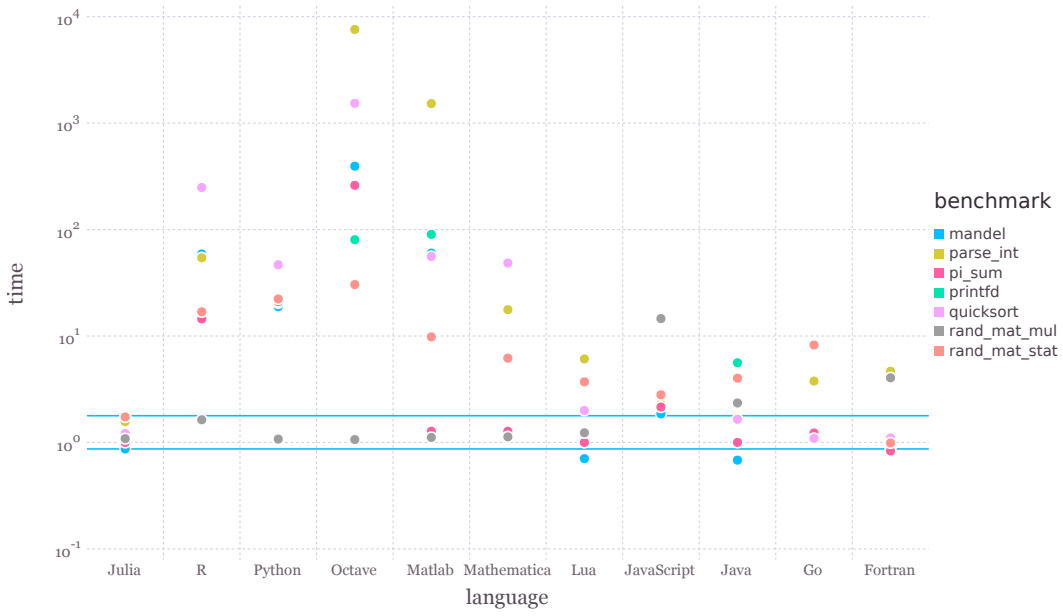


Figure 6: Julia's performance compared to other languages, taken from Julia's micro bench suite [8]. Smaller is better, C performance = 1.0.

In the first benchmark from figure 6, we can see that Julia stays well within the range of C Speed. Actually, it even comes second to C-speed with no other language being that close. This is a very promising first look at Julia, but it should be noted, that these benchmarks are written by the Julia core team. So it is not guaranteed, that there is not a bias favoring Julia in these Benchmarks. There is another benchmark comparing C++, Julia and F, which was created by Palladium Consulting which should not have any interest in favoring one of the languages. They compare in a blog post series [3] the performance of C++, Julia and F# for an IBM/370 floating point to IEEE floating point conversion algorithm. F# comes out last with 748.275 ms, than Julia with 483.769 ms and finally C++ with 463.474 ms. At the citation time, the Author had updated the C++ version to achieve 388.668 ms. This will be ignored, as it can not be said, if the other versions could not have been made faster with some more work as well.

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

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

Table 1: Performance of a FEM solver written in Julia compared to some existing libraries. [15]

These are remarkable results, considering the small effort needed to program this in Julia, while being able to compete with established FEM solvers written in C++.

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, Julia will have in the most cases identical performance to C++. The biggest remaining difference in this case is the garbage collection. Julia 0.3 has a mark and sweep garbage collector, while Julia 0.4 has an iterative mark and sweep garbage collector. As seen in the benchmarks, it does not necessarily introduce big slowdowns. But there are issues, where garbage collection introduces a significant slow down[14]. I won't go much into detail here. Julia's garbage collector is very young and only the future will show how big the actual differences will be.

Another big difference is the difference in between different compiler technologies. LLVM's

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

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

Table 2: Summary of the Phoronix benchmark. Unit is speedup of LLVM, bigger is better. [1][9][10]

The full tables can be found in the appendix under the table 11 and 13. The results suggest, that LLVM is well in the range of gcc, even though that there can be big differences between the two. This is quite comforting, especially if you consider, that LLVM is much younger than gcc. With big companies like Apple, Google[18] and Microsoft[11] being invested in LLVM, it is to be expected that LLVM will stay competitive, which means Julia should in theory stay competitive as well.

6.1.2 ModernGL

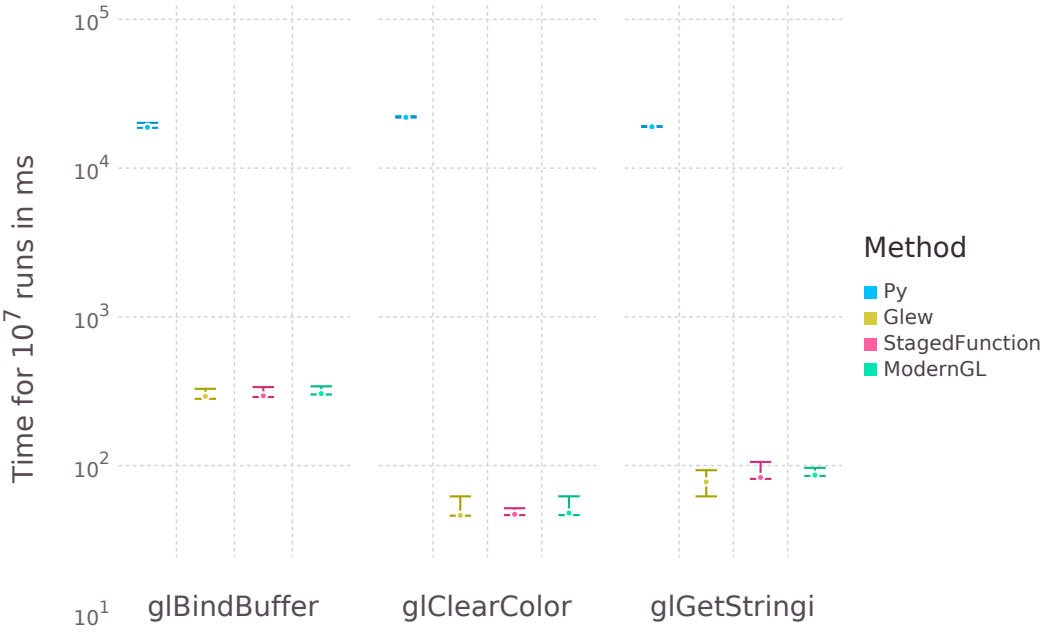


Figure 7: Different performance of OpenGL wrappers. The time for 10^7 calls was measured 100 times for each function.

Function	Python	Julia StagedFunction
glBindBuffer	64.43	1.00
glClearColor	474.72	1.02
glStringi	244.44	1.07

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

The OpenGL function loader from ModernGL has undergone some changes over the time. Starting with a very simple solution, there have been pull requests to include better methods for the functional loading. The current approach was not written by myself, but by the Github user aaalexandrov. Before aaalexandrov's approach, the fastest approach written for this thesis would have used a pretty new Julia feature, named staged functions. They should in principle yield the best performance as it directly inlines the function pointer when called the first time. Staged functions only work with the newest Julia build, which is why aaalexandrov's approach is favorable.

ModernGL gets benchmarked against Glew (with C++), PyOpenGL (with Python). From ModernGL itself, aaalexandrov's approach and the staged function approach gets benchmarked. These libraries where chosen by popularity, which was determined via

Google search rank, development status and ranking on Github. Like this, it should be guaranteed that they are the representative wrappers for the language.

The results can be seen in figure 7. ModernGL seems to do pretty well compared to C++ and python does very badly, with being roughly 400 times slower for glClearColor. Julia in contrast offers the same speed as C++ as can be seen in table 3. As all the OpenGL wrappers are pretty mature by now and bind to the same C library, this should mainly be a C function call benchmark. Python does very bad here, but it must be noted that there are a lot of different Python distributions, where 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.

6.1.3 Reactive

It is relatively hard to benchmark the used event system in real world scenarious as it's hard to find a baseline. One would have to rewrite Romeo with another Event system. Using other visulization 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 in the event graph, 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.

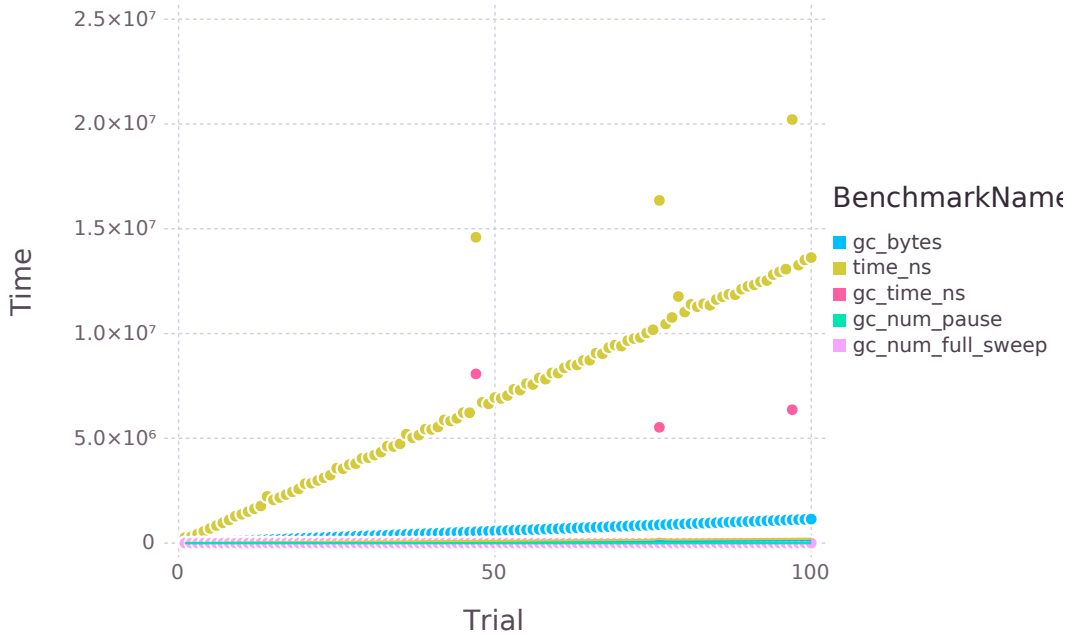


Figure 8: Reactives performance for complicated graph, simple calculation

As can be seen in figure 8, small operations with a complex event graph is bad scenario for Reactive. This doesn't come as a surprise as sorting and managing the graph structure is a much more complicated operation then adding two float32 values.

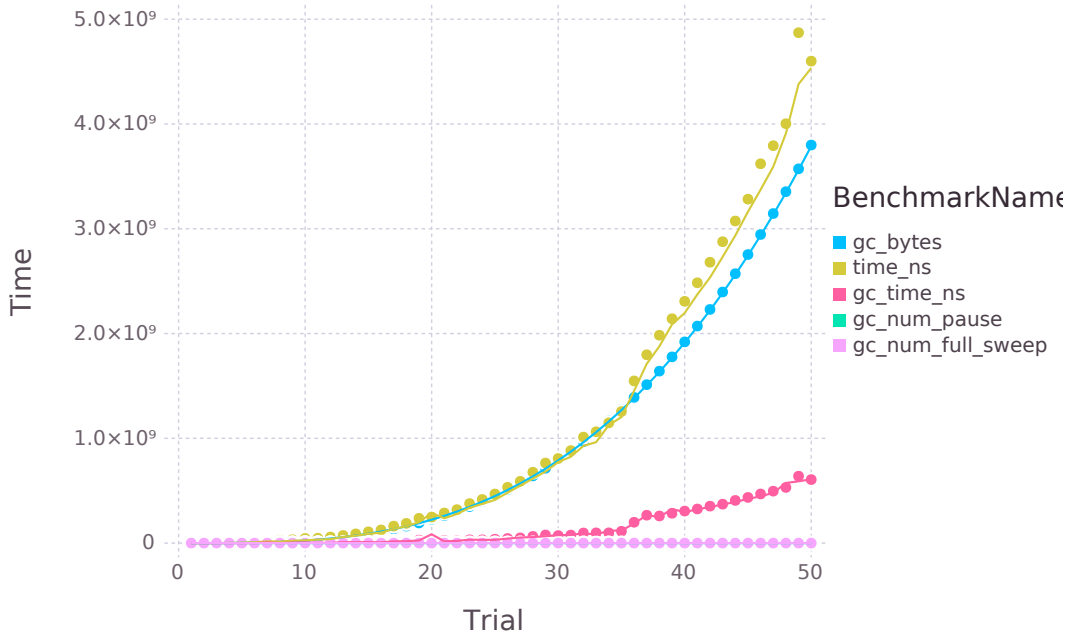


Figure 9: Reactives performance for high memory, simple event graph

The second version looks much better for Reactive. The performance difference is neglectable, making Reactive a good choice in this scenario.

6.1.4 IJulia

First of all, IJulia uses ZMQ to bridge the web interface with the Julia instance. ZMQ is a messaging system using different sockets for communication like inproc, IPC, TCP, TIPC and multicas. While it is very fast at it's task of sending messages, it can't compete with the native performance of staying inside one language. This is not very important as long as there doesn't have to be much communication between Julia and the IPython kernel. This changes drastically for animations, where big memory chunks have to be streamed.

6.2 Extensibility Analysis

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

The only exception is GLWindow, which has been used just indirectly through the other packages. This can mean three things. First, it is badly abstracted and doesn't cleanly wrap one use case. Secondly, it can be that the use case is not entirely clear to other people, which would not be a big 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 don't need the lower level functionality of GLWindow and instead rely on my other packages that use GLWindow.

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

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

6.2.1 IJulia

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

Software	**languages used**
IPython	Python 78.5% JavaScript:15.1% HTML 5.0% Other 1.4%
Three.js:	JavaScript 62.4% HTML 26.4% Python 6.9% C++ 1.9% C 1.3% GLSL 0.6%
D3:	JavaScript 95.6% CSS 4.3%

Table 4:]
Technologies used in IJulia. Statistics taken from Github

6.2.2 Paraview and VTK

Table 5 in the appendix shows an extensive summary of the used languages in the Paraview repository. It amounts to a total of 3.642.105 lines of code written in 29 languages. [more to come]

6.2.3 Matlab

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

6.3 Usability Analysis

[Consistency and ease of use of programming API in GLVisualize+GLAbstraction and Romeo. Short comment about the GUI]

7 Conclusion

7.1 Future Work

8 References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix

A IJulia

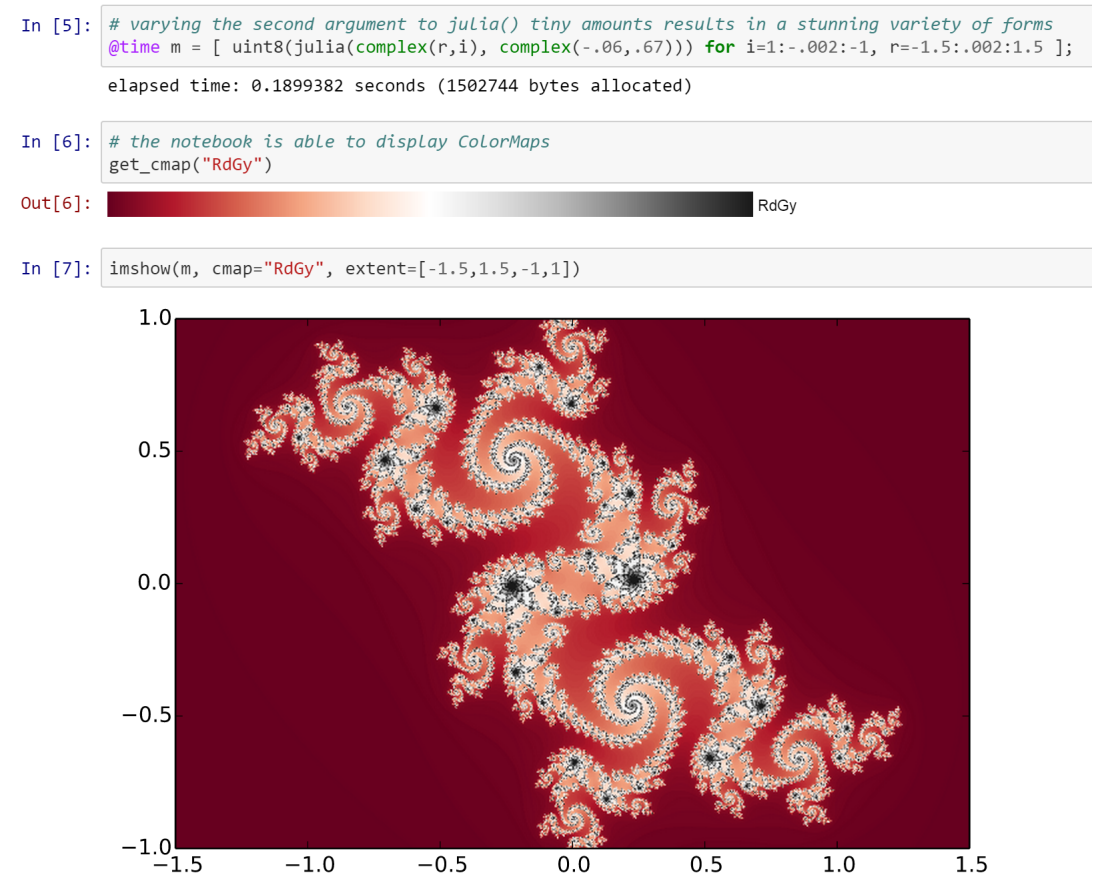


Figure 10: Example of an IJulia Notebooks. Screenshot taken from [4]

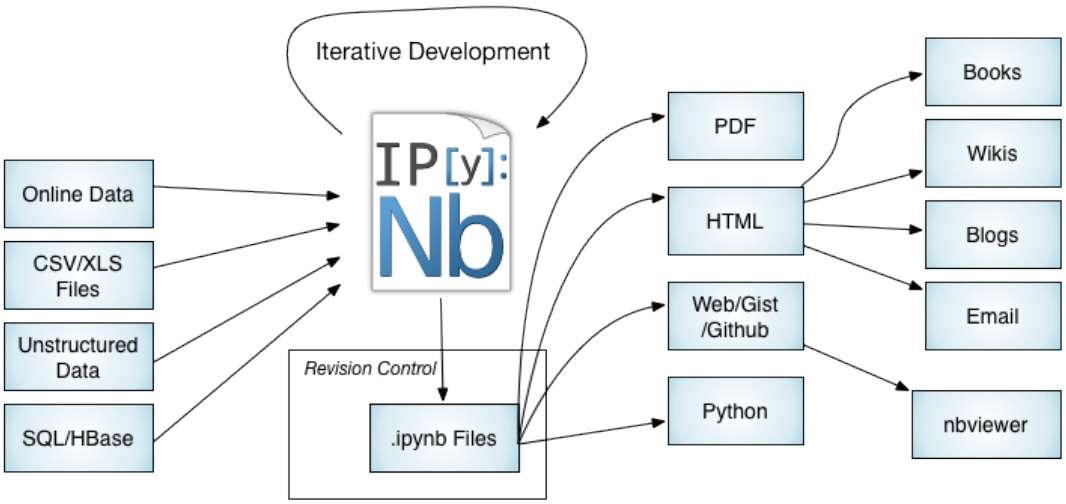


Figure 11: Workflow of IPython Notebooks. Graphic from Wikipedia [17]

B Language Statistics

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

Table 5: 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 7: 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

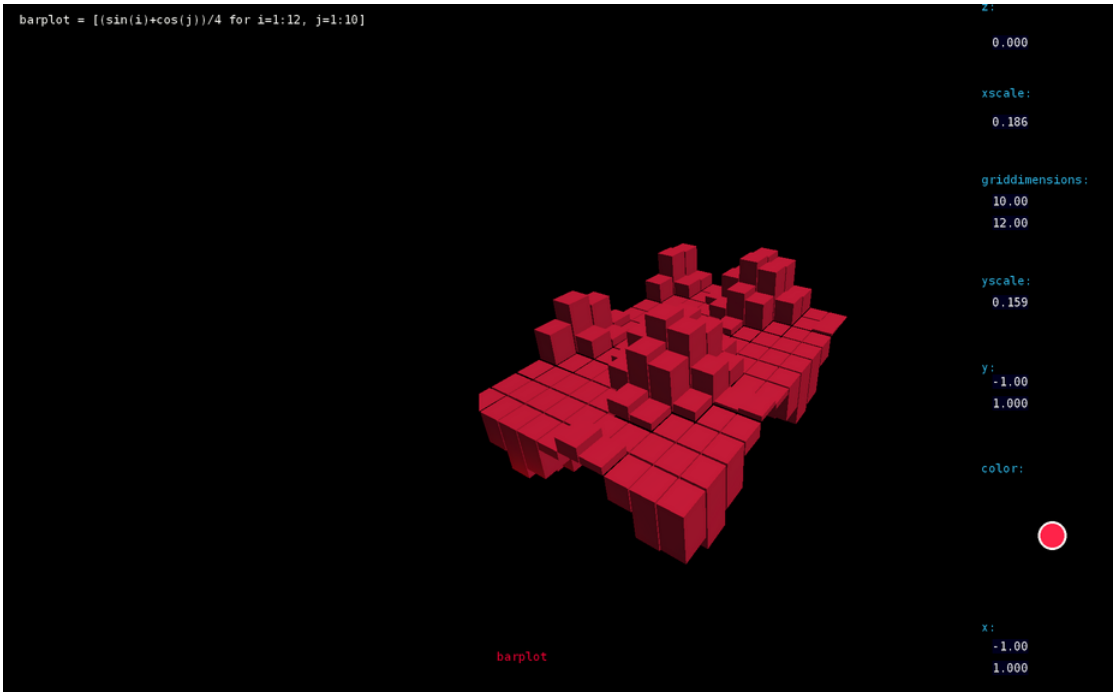


Figure 12: Screenshot of the prototype. Left: evaluated script, middle: visualization of the variable barplot, right: GUI for editing the parameters of the visualization

D Benchmark

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

Table 9: FE Implementation comparison

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

Table 11: LLVM 3.5 compared to LLVM 3.6 in the Phoronix benchmark test suite[1]

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

Table 13: gcc 4.9.2 compared to LLVM 3.5 in the Phoronix benchmark test suite[9]

Official Statement

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

Date: .....

(Signature)