UNIVERSITÄT OSNABRÜCK

Faculty of
Cognitive Science

Bachelor Thesis

Romeo: Scripting Environment with interactive Visualizations

Author: Simon Danisch
sdanisch@email.de

Supervisor: Prof. Dr.-Ing. Elke Pulvermüller

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

Filing Date: 01.02.2014

I Abstract

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

II Table of Contents

I Abstract **I**

II Table of Contents **II**

III List of Figures **IV**

IV List of Tables **V**

V Listing-Verzeichnis **V**

VI List of Abbreviations **VI**

1 Introduction **1**
1.1 Field of Research and Problem . . . 1
1.2 Problem Solutions and Measurements of Success . . . 3
1.2.1 Visualizations and Interaction . . . 3
1.2.2 Extensibility . . . 3
1.2.3 Speed . . . 4
1.3 Outlook . . . 5
1.4 Used Technologies . . . 5
1.4.1 Julia . . . 5
1.4.2 OpenGL . . . 5
1.5 Similar Work . . . 5
1.5.1 Other Languages . . . 5
1.5.2 Ipython Notebook . . . 5
1.5.3 Matlab . . . 5
1.5.4 Mathematica . . . 5
1.5.5 Other Graphic acceleration APIs . . . 5

2 Background **6**

3 Architecture **7**
3.1 Event System . . . 7
3.2 Low Level . . . 8
3.2.1 ModernGL . . . 8
3.2.2 GLAbstraction . . . 8
3.2.3 GLFW . . . 8
3.2.4 GLWindow . . . 8
3.3 High Level . . . 8
3.3.1 GLVisualize . . . 8
3.3.2 Romeo . . . 8

4 Results and Analysis **8**
4.1 Performance Analysis . . . 8
4.2 Extendability Analysis . . . 8

4.3 Usability Analysis 8

5 Conclusion **8**
5.1 Discussion 8
5.1.1 Performance 8
5.1.2 Extensability 8
5.1.3 Usability 8
5.2 Future Work 8

Appendix **I**

A GUI **I**

B Benchmark **I**

III List of Figures

Abb. 1 Volume Visualization 1
Abb. 2 Volume Visualization 4
Abb. 3 Architecture 7
Abb. 4 Prototype I

IV List of Tables

Tab. 1 FE Implementation comparison I

V Listing-Verzeichnis

VI List of Abbreviations

LLVM	Low Level Virtual Machine
IR	Intermediate Representation

1 Introduction

This Bachelor Thesis is about writing a fast and interactive visualization environment suitable for scientific computing. As GUI elements and editable text fields are supplied, one can also write and execute scripts, and immediately visualize all bound variables of the script and edit them via simple GUI elements like sliders. The focus is on creating a modular library, that is written in a fast high-level language, making the library easy to extend. 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 Field of Research and Problem

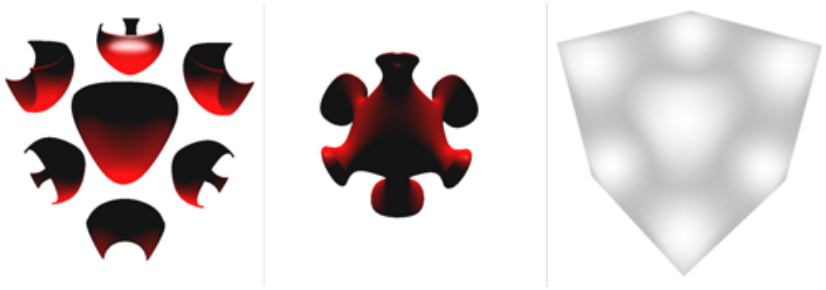

Figure 1: different visualizations of $f(x, y, z) = \sin(\frac{x}{15}) + \sin(\frac{y}{15}) + \sin(\frac{z}{15})$, visualized with Romeo. From left to right: Isosurface, isovalue=0.76, Isosurface, isovalue=0.37, maximum value projection

It's not immediately visible, but the general research field is making computer science more accessible and understandable. Visualization and interaction do make opaque processes more accessible, by giving the user the ability to look at data and algorithms from different perspectives. One example can be this function $f(x, y, z) = \sin(\frac{x}{15}) + \sin(\frac{y}{15}) + \sin(\frac{z}{15})$, which describes a 3D volume. 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 an algorithm produces. It is also very helpful to debug the actual math and not just the program. For example by visualizing the rotation from a rotation matrix, immediately makes it visible if the wanted effect was achieved. This is especially relevant for users, who don't have a broad background in computer science, or programmers implementing complicated state of the art algorithms. These users can be found especially in scientific computing, which is why this thesis focuses on scientific computing.

More precisely, it focuses on research which involves writing short scripts, while visualizing

the results is helpful. 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.

Several demands by the researcher makes it challenging to offer software for this area. Lets look at one of the challenges at a time:

- **Visualizations**

  Visualizations are a key element to the understanding and access of complicated algorithms. In some domains, problems become only manageable by visualizing them. The problem with that is, that a visualization which successfully captures the gist of the data is pretty hard. To leverage this problem, the visualization API should be intuitive to use and very flexible, to allow the researcher to build highly customized visualizations for his problem. Also, research is getting published, together with visualizations explaining the results. As they represent the research to the public, they should be as understandable as possible and preferably look good. Offering a creative, interactive work flow can make this challenge considerably easier.

- **Money and Time is constrained**

  This means the research has to conclude quickly and most likely, it is not an option to employ a person or even a company to solve sub problems. From this we can deduce three preferences: The used libraries should be accessible to the researcher, because when something doesn't fit his demands, he most likely needs to resolve it himself due to constrained resources. If code needs to be written to solve this, it should be in an easy to understand high-level language, otherwise the sub problem can considerably slow down the research.

- **Speed**

  Speed can be both seen as a usability or a time/money constraint. Time and money constraints become clear, if the computation times are very big. If one library is 60 times slower than the other, it might not matter if the task only takes 10^{-5} s. But it does matter if the fast computation takes a day, leaving the slow library with a computation time of 60 days. This would mean for researchers, that they either have to buy more powerful computers, or deal with the lost time. But also for small numbers in the second range, a difference of 20 times slower can have a big impact on the researchers productivity. If the computation is repetitive, like trying out

all the different materials, this has an immediate influence on how many material combinations the researcher can try out in one day. More subtle is the influence of speed on a simple task like changing the color. If the color slider stutters, it is completely possible to change the color, but it will be a frustrating user experience. This is why the whole system should be designed to offer top notch speed, to not run into these kind of issues.

1.2 Problem Solutions and Measurements of Success

1.2.1 Visualizations and Interaction

Having visualizations are the main purpose of the written software in this thesis.

1.2.2 Extensibility

As previously deduced, 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, this 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 other examples for inaccessibility are writing 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 more simple. 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 moved is shorter and less time is spend on understanding the code.

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

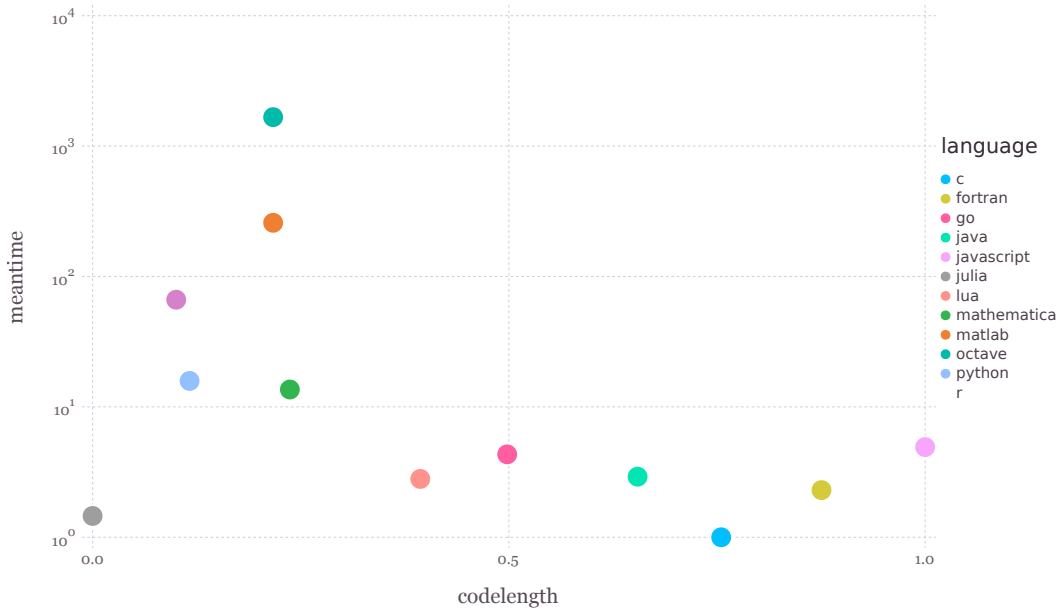

Figure 2: Languages speed relative to C (averaged benchmark results), plotted against the length of the needed code

1.2.3 Speed

How to achieve state of the art speed with a high level language is an on going research and basically the holy grail of language design. Luckily, there is Julia building upon Low Level Virtual Machine (LLVM), promising a concise, high-level programming style, while approaching C-performance. This is well illustrated in figure 2. LLVM is a nice compiler infrastructure, which has frontents for different languages and back-ends for different chip architectures. A language designer has the task, to emit LLVM Intermediate Representation (IR), which than gets just in time compiled and optimized to the architecture by LLVM. This concept is very nice, as you can accumulate state of the art optimizations in one place, making them accessible to many languages, while being able to compile to different platforms. There are x86, ARM, OpenCL and CUDA backends. While Julia doesn't support them all, it will hopefully be possible in the future. LLVM is also used by Clang, the C/C++ frontend for LLVM rivaling gcc and for Apple's programming language Swift. This makes LLVM is a very solid basis for a programming language, as these are highly successful projects giving LLVM a lot of drive(source!?). See table 1, for a little benchmark.

To get high performant 3D graphics rendering, there are on the first sight a lot of options. If you start to take the previously demands into account, the options shrink down. It should be implemented in one high level language, which can be used for scientific

computing and has state of the art speed. At this point, there are close to zero libraries left. As you can see in figure 2, Matlab, Python and R disqualify, as they are too slow. JavaScript, Java, Go and Lua are missing a scientific background and the others are too low level for the described goals. This leaves only Julia, but in Julia there weren't any 3D libraries available, which means that one has to start from scratch. There are only a couple of GPU accelerated low-level libraries available, namely OpenGL, DirectX and Mantel, which are relatively comparable. This leaves one library, if you additionally introduce the constraint of cross-platform compatibility. 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.

1.3 Outlook

1.4 Used Technologies

1.4.1 Julia

1.4.2 OpenGL

1.5 Similar Work

1.5.1 Other Languages

1.5.2 Ipython Notebook

1.5.3 Matlab

1.5.4 Mathematica

1.5.5 Other Graphic acceleration APIs

2 Background

3 Architecture

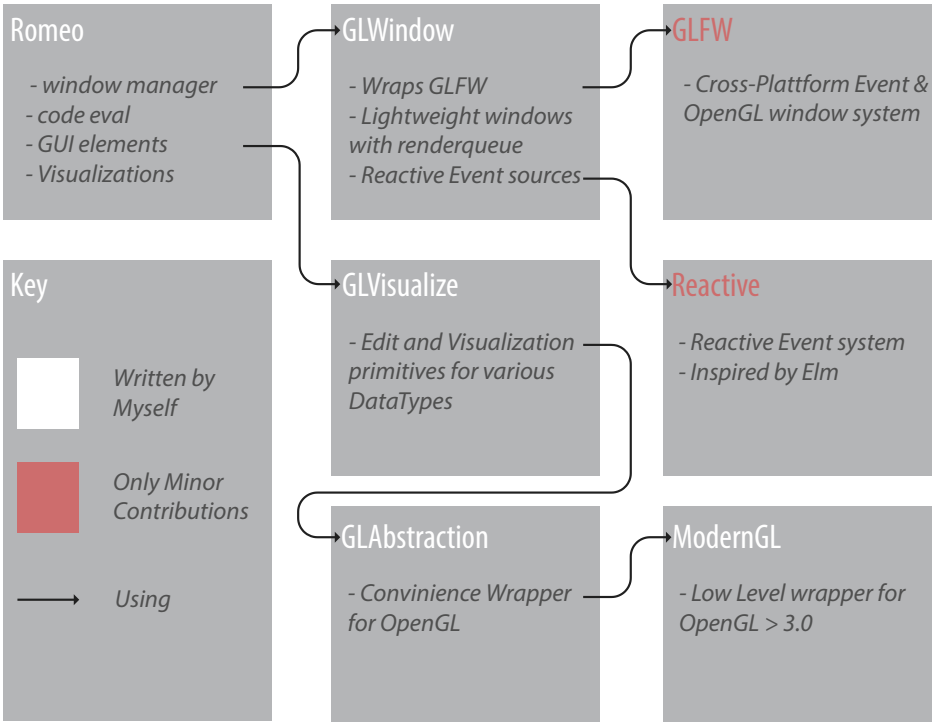

Figure 3: Main modules used in Romeo and their relation (simplified)

This chapter is about the architecture of Romeo. Romeo itself 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 nice 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, which is a reactive event system written in Julia.

3.1 Event System

Holding everything together, since 1890.

3.2 Low Level

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

3.2.2 GLAbstraction

3.2.3 GLFW

3.2.4 GLWindow

3.3 High Level

3.3.1 GLVisualize

GLVisualize implements the main functionality.

3.3.2 Romeo

4 Results and Analysis

4.1 Performance Analysis

4.2 Extendability Analysis

4.3 Usability Analysis

5 Conclusion

5.1 Discussion

5.1.1 Performance

5.1.2 Extensability

5.1.3 Usability

5.2 Future Work

Appendix

A GUI

A nice Appendix.

Screenshot

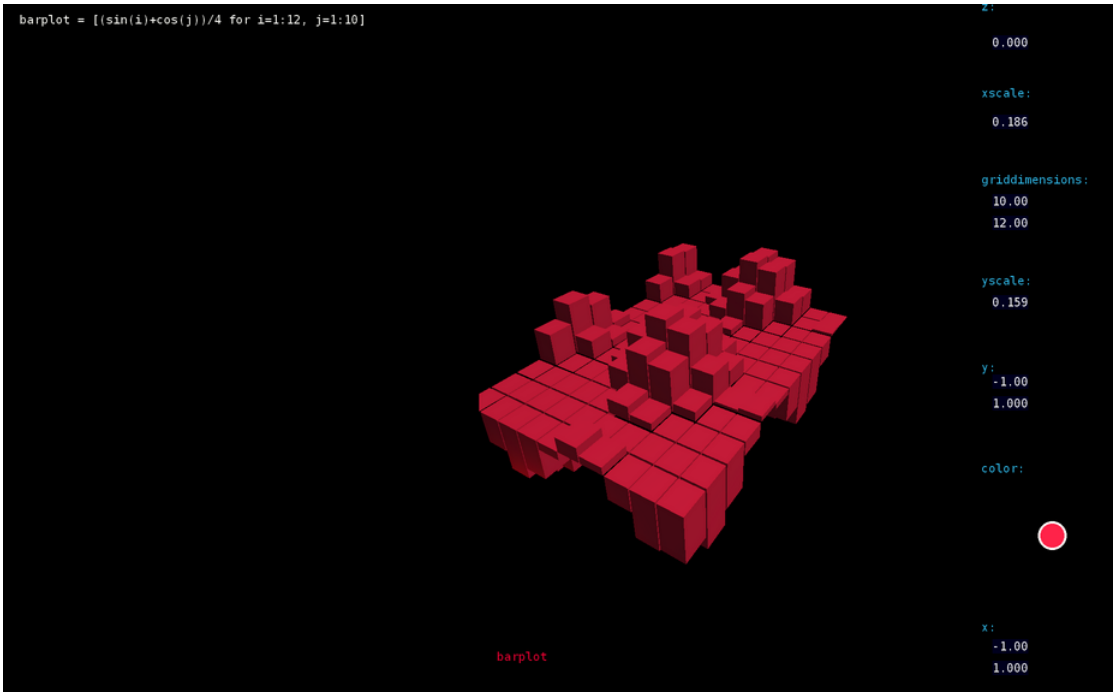

Figure 4: Screenshot of the prototype. Left: evaluated script, middle: visualization of the variable barplot, right: GUI for editing the parameters of the visualization

B Benchmark

Table 1: FE Implementation comparison

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

Official Statement

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

Date: .....

(Signature)