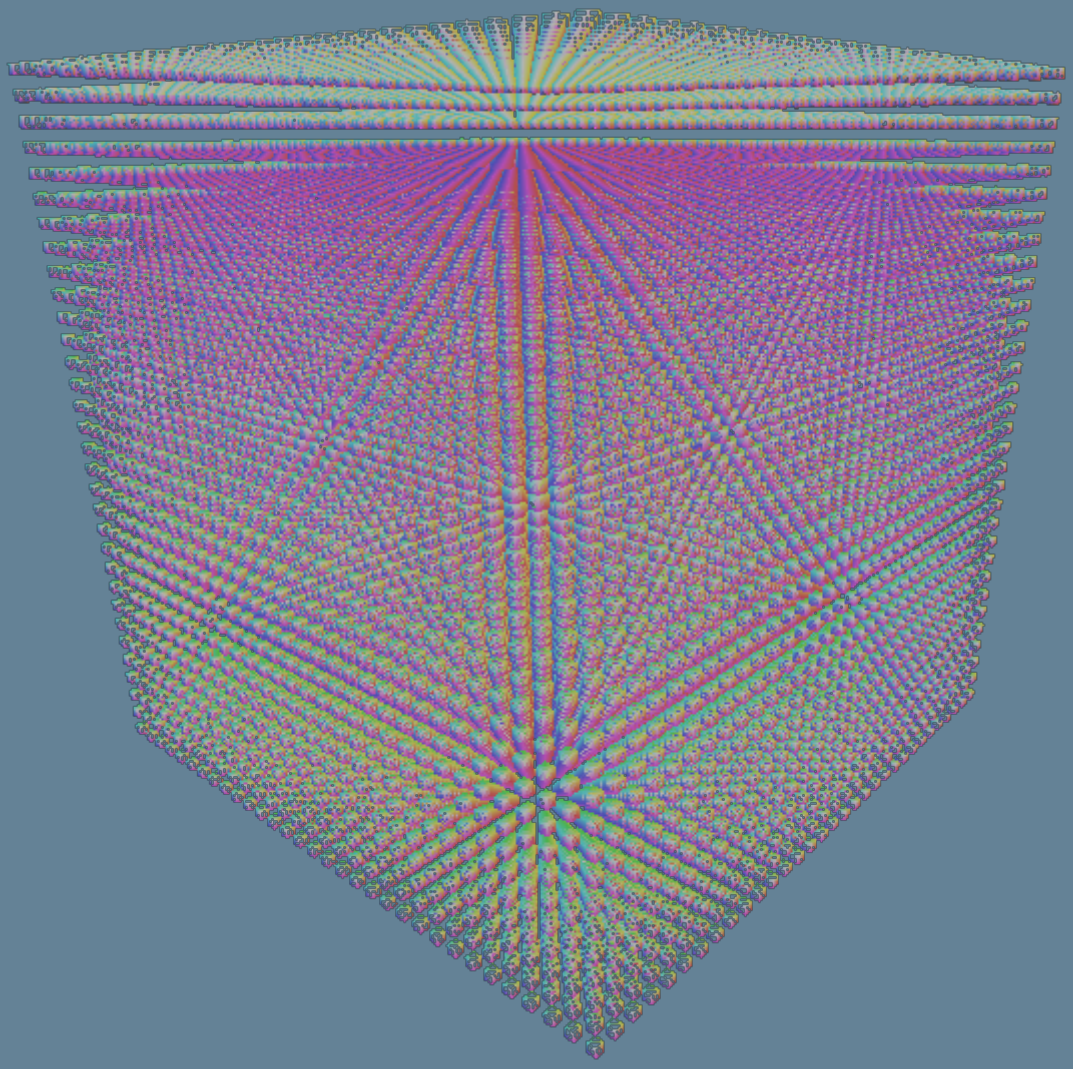


Julia Grafik Engine

Independent Coursework 2

Mario Link (s0536176)

Inhaltsverzeichnis

Julia Grafik Engine

Mario Link

16. April 2018

1 Einleitung

Das Dokument beschreibt die Idee der Entwicklung einer Grafik Engine in der Sprache Julia. Das Ziel ist es zu zeigen ob Julia für Entwicklung von Grafik Engines geeignet ist. Ein weiteres Ziel ist einen Vergleich zu anderen Sprachen aufzustellen und welche Vor- und Nachteile sich mit dieser Sprache ergibt.

Problemstellung/Fragestellung

Ist die Entwicklung einer Grafik Engine in Julia einfacher als in anderen Sprachen wie z.B. Python?

Weitere Fragen

- Welche Vor-/Nachteile ergeben sich bei der Verwendung von Julia statt anderen Sprachen wie z.B. Python? Gibt es Performance unterschiede?
- Was sind der Vor-/Nachteile von JIT?
- Sind Design Pattern in Julia umsetzbar?
- Wie sieht Parallelisierung, Garbage Collection und Unit Tests in Julia aus?
- Wie gut sind Julia Projekte Skalierbar? (Blick in Zukunft: Weiterentwicklung zur Game Engine)

Ziel

Demonstrationssystem (Grafik Engine) in Julia entwickeln, Arbeitsaufwand und Performance bewerten.

2 Basics

2.1 Was ist Julia?

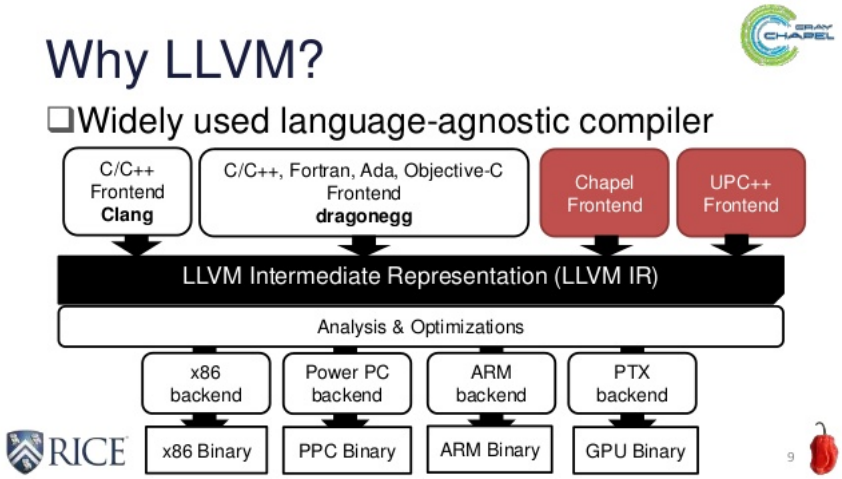
Julia ist eine Hoch-Level, hoch-performante dynamische Sprache für numerisches Rechnen. Es stellt einen hochentwickelten Compiler, verteilte parallelisierte Ausführung, numersiche Genauigkeit und eine umpfangreiche mathematische Funktionsbinliothek zur Verfügung. Julia's Basis-Bibliothek wurde größtenteils selbst in Julia geschrieben. Julia integriert zudem optimierten (bestmöglichen) Code von Open Source C und Fortran Bibliotheken für lineare Algebra, Generierung von Zufallsnummern, Singal und String Verarbeitung. Zusätzlich bietet die Julia Entwickler Community eine Reihe von externen Packeten (Modulen) durch den eingebauten Julia Packetmanager an. IJulia ist eine Kollaboration zwischen Jupyter und Julia Communities und stellt ein mächtiges Browser-basiertes grafisches Notebook mit Schnittstelle zu Julia zur Verfügung. Julia's Stärken zeigen sich vorallem durch den auf LLVM-basierten hoch-performanten just-in-time (JIT) Compiler, der kombiniert mit dem Julia Sprach-Design eine C nahe Performance ermöglicht. Julia Programme sind durch den multiplen Dispatch organisiert, der den Einsatz für verschiedene Kombinationen von Argumenttypen für überladete Funktionen unterstützt. Die Sprache kann als dynamische Bibliothek (shared library) gebaut werden. Einfache Aufrufe zu externen Funktionen in C und Fortran aus dynamischen Bibliotheken sind ebendso möglich, ohne das Wrapper Code geschrieben oder existierener Code recompiliert werden muss. Julia ist Open Source (Core Bibliothek enthält MIT Lizenz). Weitere Lizenzen wie GPL, LGPL, and BSD werden von anderen Bibliotheken genutzt.
(siehe https://julialang.org/#free-open-source-and-library-friendly)

Zusammenfassung der Features:

- Multiples Dispatch
- Dynamisches Typ-System
- Gute performance (nahe C)
- Built-in package manager
- Lisp-like Macros und andere Vorteile der Metaprogrammierung
- Call Python Funktionen (PyCall)
- Call C Funktionen (direkt, keine Wrapper)
- Mächtige shell-ähnliche Fähigkeiten für das Managen von Prozessen
- Designed für Paralellität und Cloud Computing
- Coroutinen: leichtgewichtetes "grünes" Threading
- Benutzerdefinierte Typen (schnell und kompakt wie eingebaute Basis Typen)
- Automatische und effiziente Code Generierung
- Elegante und erweierbare Konvertierung und Promotionen für numerische und andere Types
- Effizienter Support für Unicode (UTF-8)
- MIT Lizenz: frei und Open Source

2.2 Was ist LLVM?

LLVM (Low Level Virtual Machine)ist eine modulare Compiler-Unterbau-Architektur mit einem virtuellen Befehlssatz, einer virtuellen Maschine, die einen Hauptprozessor virtualisiert. Es besteht aus einer Sammlung von Compiler und Toolchain Technologien für die Entwicklung von Compiler Back- und Frontends. LLVM ist in C++ geschrieben und wird zur Optimierung von Compilerzeit, Linkzeit, Laufzeit und Leerlaufzeit herangezogen.



Quelle: llvmbased-communication-optimizations-for-pgas-programs

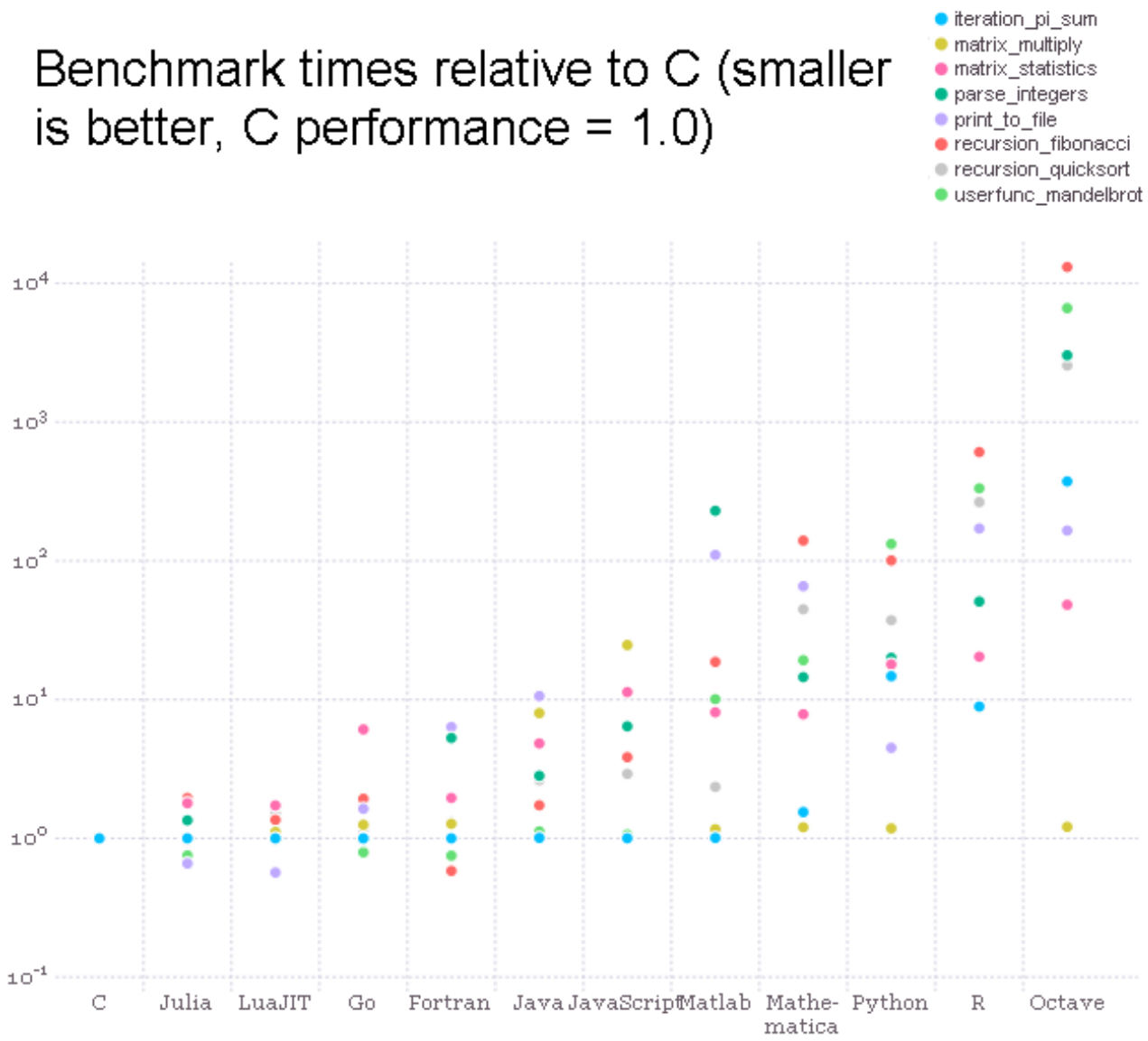


Abb. 1
Vergleich von Micro-benchmarks zwischen verschiedenen Sprachen: C, Fortran, Julia, Python, Matlab/Octave, R, JavaScript, Java, Lua, Mathematica.

Quelle: Micro-benchmarks https://julialang.org/

2.3 Dokumentation

Julias Dokumentation ist auf Seite https://docs.julialang.org/en/stable/ einsehbar.
Ein kleine Hilfe für den schnellen Einstieg bietet die Seite https://learnxinyminutes.com/docs/julia/

2.4 Beispiel Code

Die ausführbare Julia Datei (julia.exe) liefert eine interaktive Befehlsoberfläche (Konsole) genannt REPL (read-eval-print loop). In dieser Konsole kann Julia Code direkt ausgeführt werden. Desweiteren können Julia Skripte angelegt und per julia.exe ausgeführt werden. Die Vorgehensweise:
1. Ein Batch Script run.bat schreiben und den Pfad zur julia.exe und dem Script angeben.

```
@echo off
"C:/Users/\%username\%/AppData/Local/Julia-0.6.0/bin/julia.exe" "main.jl"
pause
```

2. Das Script main.jl anlegen.

```
println("Hello World!")
```

3. Die run.bat starten und das Ergebnis bewundern.

2.5 Julia Box - Julia Code im Web

Julia kann auch im Web ausgeführt werden, siehe https://www.juliabox.com/.

2.6 Design Pattern

Anders als in Java existiert in Julia die Behandlung von Klassen, private und public Definitionen nicht. In Julia existieren Module, Structs, Types und Funktionen. Module könnten als public static Class betrachtet werden. Der Unterschied von Struct zu Type ist, dass Struct Objekte unveränderlich sind.

```
julia> module MyApp end
julia> type MyObject end
julia> struct MyImmutableObject end
```

Die Zugehörigkeit von Funktionen zu Structs/Types lässt sich nur funktional über den Funktionsparameter lösen.

```
julia> function myFunction(this::MyObject, value::Int) end
```

Allerdings kann die Zugehörigkeit durch einen Workround, durch das Speichern von anonymen Funktionen, geregelt werden:

```
julia> type MyObject
   x::Int
   getX::Function # public method
   function MyObject()
      this = new()
      this.x = 0
      this.getX = ()->this.x
      this # return object
   end
end
julia> MyObject().getX()
```

Private Methoden können mit diesem Ansatz gelöst werden:

```
julia> let
   global brotherB
   function brotherA(x="I am private!")
      println("brotherA: ",x)
   end

   function brotherB()
      println("brotherB: I am public!")
      brotherA("Bro... i was hiding!!!")
   end
end
julia> brotherA()
ERROR: UndefVarError: brotherA not defined
julia> brotherB()
brotherB: I am public!
brotherA: Bro... i was hiding!!!
```

Informationen zum Scope können hier gefunden werden: https://docs.julialang.org/en/stable/manual/variables-and-scoping/

2.7 Garbage Collection

Der Garbage Collector erfolgt nach einer gewissen Zeit automatsich. Allerdings kann mit dem Befehl gc() der Garbage Collector sofort ausgeführt werden.

2.8 Unit Tests

Tests können in Julia können z.B. mithilfe des Macro Befehls @test aus dem Modul Base.Test ausgeführt werden (https://docs.julialang.org/en/stable/stdlib/test).

```
julia> using Base.Test
julia> @test 1 == 1
Test Passed
julia> @test 1 == 2
Test Failed
  Expression: 1 == 2
   Evaluated: 1 == 2
ERROR: There was an error during testing
```

2.9 Parallelisierung

Mit Ausführung von ''julia.exe -p 10 myScript.jl'' kann ein Script über mehrere Prozesse (z.B. 10) gestartet werden:

```
@everywhere function write(i)
  Libc.systemsleep(5)
  (myid(),i)
end
list = Future[]
println("Write")
for i=1:nprocs() println("$i"); push!(list, @spawn write(i)) end
println("Read - Wait ~ 5 sec")
for entry in list println(fetch(entry)) end
```

Threads lassen sich bisher nur experimentell mit Umgebungsvariable JULIA_NUM_THREADS=4 (4 Threads) ausführen:

```
a = zeros(10)
Threads.@threads for i = 1:10
   a[i] = Threads.threadid()
end
```

2.10 JIT und Power of Eval

Der JIT Compiler ermöglich die Ausführung der des Codes zur Laufzeit. Demnach werden Beispielsweise include() Anweisungen in Funktionen merhfach ausgeführt.

```
julia> function inlucdeAll()
   include("myCode1.jl")
   include("myCode2.jl")
   include("myCode3.jl")
end
julia> for i:10 #include all code 10 times
   inlucdeAll()
end
```

Allerdings kann es hierbei zu Problemen kommen, wenn in dem includierten Code Struct oder Typen definiert sind. Eine Redefinition dieser Objekte ist nicht möglich, da diese an das momentane Modul festgebunden sind. Erst ein komplettes überschreiben des Moduls ermöglicht eine Redefinition.

```
julia> type MyObject end
julia> type MyObject x::Int end
ERROR: invalid redefinition of constant MyObject
```

Mit der Anweisung eval() lässt sich dynamischen Julia Code generieren. Die Funktion eval benötigt einen Symbol Parameter, um neuen Code zu generieren. Durch die Funktion parse() lässt sich ein Befehl als String Objekt in einen Symbol Objekt umwandeln.

```
julia> function myFunction() end
julia> myFunction() #keine Ausgabe
julia> eval(parse("function myFunction(); println(\"Hi!\"); end"))
julia> myFunction() #Ausgabe "Hi!"
```

2.10.1 Was sind die Vor- und Nachteile von JIT?

Vorteil von JIT ist, dass der Code nicht noch einmal interpretiert werden muss, sondern direkt in Binärcode kompiliert wird. Dadurch wird sofort ein Ergebnis geliefert. JIT hat längere Ausführungszeit als AOT (Ahead of Time) Systeme. Liefert jedoch deutlich besseren Code. Durch die extistieren Systemprofile werden die bestmöglichen Instruktionen für das Zielsystem bei der Kompilerung ausgewählt. AOT liefert eine schnellere Ausführungsgeschwindigkeit als JIT, verbraucht jedoch mehr Arbeits- und Festplattenspeicher und muss alle Möglichen Systemprofile durchgehen (Quelle: https://www.thoughtco.com/definition-of-compiler-958198).

3 Julia vs Python

3.1 Artikel 'Giving up on Julia'

In meiner Recherche bin ich auf den Artikel **Giving up on Julia** (http://www.zverovich.net/2016/05/13/giving-up-on-julia.html) gestoßen. Der Artikel ist 2 Jahre alt und bezieht sich auf Julia 0.4. Er beschreibt Probleme von Julia und zeigt nur wenig Testszenarien als Beweis. Der Author bemängelte die Microbenchmarks auf der Julia Webseite, sowie:

- Performance Probleme bei StartUp Zeit und JIT lags
- Undurchsichtige Syntax und Probleme mit der Zusammenarbeit mit anderen Sprachen
- Schwache Textformatierungshilfen in der Sprache sowie fehlen von guten Unit Test Frameworks
- standardmäßig unsicheres Interface zu nativen Schnittstellen (APIs)
- Unnötig komplizierte Code-Basis und unzureichende Aufmerksamkeit beim BugFixing

Ein Test zeigt ein ''Hello World'' Program und ein weiterer Test einen Schnittstellenaufruf mit ccall(). In den Testszenarien wurde keine direkten Gegenüberstellung von Python und Julia gezeigt.
Desweiteren schlägt er vor Python Funktionen zu optimieren (siehe https://www.ibm.com/developerworks/community/blogs/jfp/entry/Python_Meets_Julia_Micro_Performance)

3.2 Artikel 'Updated Analysis'

Als Gegendarstellung enstand der Artikel **Updated Analysis** (https://tk3369.wordpress.com/2018/02/04/an-updated-analysis-for-the-giving-up-on-julia-blog-post/), der durch direkte Gegenüberstellung zeigt, dass Python gegenüber Julia schlechter abschneidet. Der Author dieses Artikels kritisiert den Author des **Giving up on Julia** Artikels. Dieser habe die Darstellung nur verwendet, um auf seinen Block ''How To Make Python Run As Fast As Julia'' aufmerksam zu machen und nicht das eigentliche Problem der Gegenüberstellung behandelt. Die Gegendarstellung in diesem Artikel zeigt Tests mit der Berechnung von Fibonacci-Zahlen, Quicksort, Mandelbrot Set und Integer Parsing. Desweiteren bezieht er sich auf die Argumente im **Giving up on Julia** Artikel wie Syntax, Sicherheit, Textformatierung beim Printf, Unit Tests und weiteren. In allen Tests schneidet Julia besser ab als Python oder CPython.

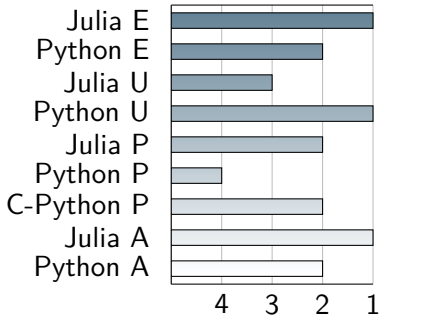
3.3 Artikel 'Observations'

Bei diesem Artikel (https://medium.com/@Jernfrost/python-vs-julia-observations-e61ee667fa95) bezieht sich dieser Author auf die allgemeine Handhabung von Julia und Python. Er vergleicht REPL, die Interaction mit Arrays, die Integration von Shell Befehlen und anderen Eigenschaften der Syntax. Julia schneidet hier nach seiner Meinung sehr gut ab. Der Author erkennt an, dass Python eine deutlich größere Community besitzt als Julia und dadurch einen Vorteil erhält. Er ist jedoch überzeugt, dass Julia einene besseren Einstieg ermöglicht als Python.

3.4 Zusammenfassung

Julia ist eine sehr neue Sprache und enthält viele Vorteile aus anderen Sprachen und weniger deren Nachteile. Julia ist aus dem Kenntnisstand erfahrender Entwickler aus aller Welt enstanden, die früher in Matlab, C++, Java, R, Python und anderen Sprachen programmiert haben. Ich sehe darin ein deutlichen Vorteil, da unterschiedlich Erfahrungswerte hier zusammenkommen.

Im Überblick - die Persöhnliche Einschätzung (Erfahrungswerte): (E = Einfachheit, Syntax, Style; U = Umpfang, Anzahl Bibliotheken; P = Performance zu C; A = investierter Aufwand/Zeit für einen Anwendungsfall)



4 Entwicklung

Für die Entwicklung wurden zwei Programme geschrieben. Das erste repräsentiert die Herangehensweise der Entwicklung einer Grafikengine und das zweite die Herangehensweise eines optimierten Renderalgorithmus für einen speziellen Fall (Das Rendern von vielen Blöcken/Cubes).

4.1 Projekte einrichten

Herunterladen:

1. Julia 0.6: https://julialang.org/downloads/
2. Julia OpenGL: https://github.com/Gilga/JuliaOpenGL
3. Julia Grafik Engine: https://github.com/Gilga/JGE

Installieren:

1. Julia 0.6 Setup ausführen
2. Bei den Projekten die dort enthaltene README.md lesen und Packete nachinstallieren mit:

```
Julia> Pkg.add("Modulname hier einfügen")
```

Die Projekte wurden nur auf diesem System entwickelt und getestet:

- Operating System: Windows 10 Home 64-bit

- Processor: Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz (4 CPUs), 2.0GHz
- Memory: 8192MB RAM
- Graphics Card 1: Intel(R) HD Graphics Family
- Graphics Card 2: NVIDIA GeForce 840M

Es wurde bevorzugt mit der NVIDIA Grafikkarte getestet (für bessere FPS Werte).

4.2 Julia GrafikEngine

Das Projekt ist so aufgeteilt das App bezogene Skripte und Objekte (wie Shader, Texturen, Szenenscript) im Root Verzeichnis des Projektes liegen und die Core Skripte für die Engine im source Ordner. Im source Ordner werden die Julia Versionen unterschieden (0.5, 0.6, usw.). In dem Ordner sollten der Source Code (im src Ordner), sowie Tests (im test Ordner) stehen. In meinem Fall habe ich aus Prioriäts- und Zeitgründen keine konkreten Skripttests geschrieben. Stattdessen wurden die Tests manuell über die Julia Konsole ausgeführt, um Probleme zu beheben. Allerdings existieren allgemeine Test Skripts im zweiten Projekt (JuliaOpenGL Projekt).
Skripte und Ordner:

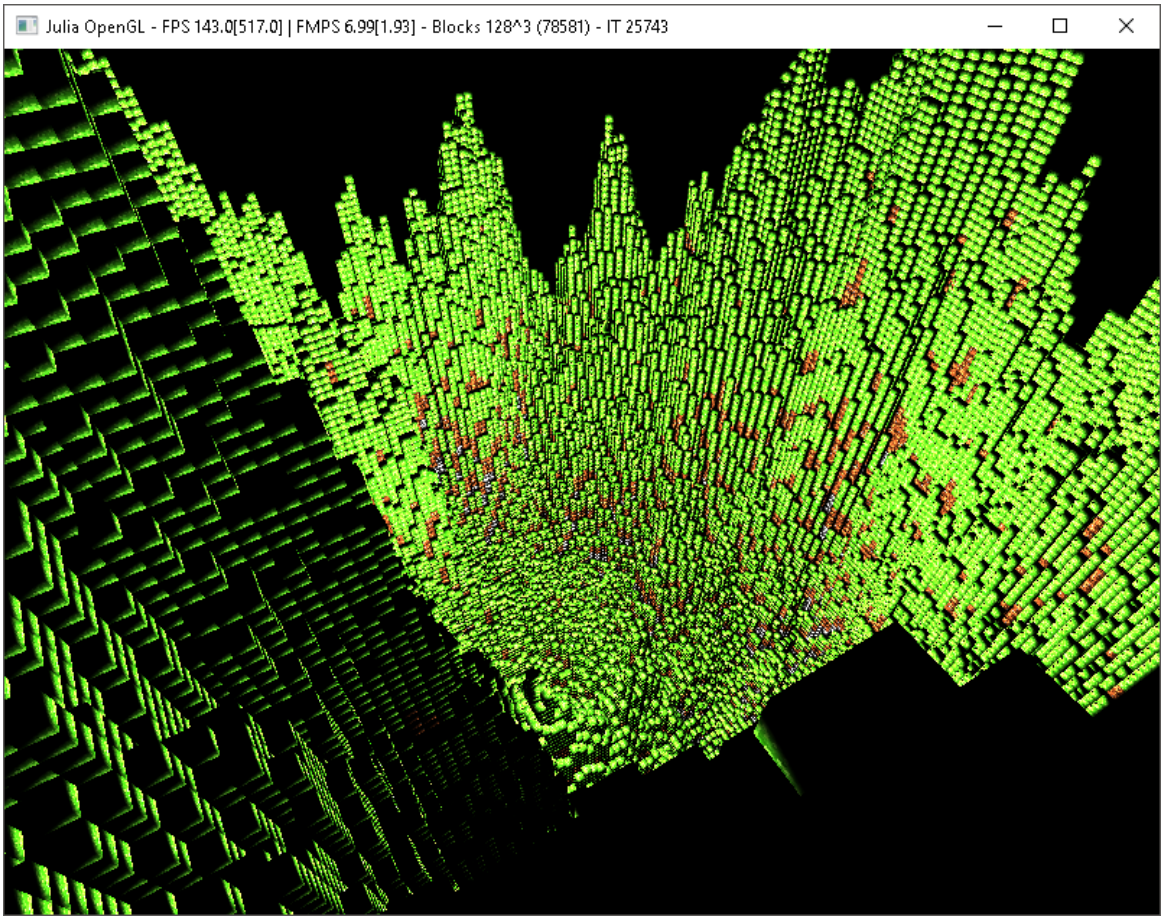
- scripts/input.jl - enthält Szenenbeschreibung
- source/0.6/App.jl - Beschreibt den Aufruf des Programms und initialisiert den Prozess
- source/0.6/CoreExtended.jl - enthält eine Sammlung von erweitere Kernfunktionen für viele Zwecke
- source/0.6/Environment.jl - legt Umgebung fest wie z.B. globale Pfade und Variablen. Ist derzeit leer und nicht von Bedeutung.
- source/0.6/FileManager.jl - Behandelt Lese und Schreib Funktionen für Dateien, sowie ein Event wenn festgelegte Dateien verändert wurden.
- source/0.6/MatrixMath.jl - enthält mathematische Operationen für Matritzen und Vectoren
- source/0.6/LoggerManager.jl - enthält die Vewaltung des Loggings
- source/0.6/JLScriptManager.jl - Verwaltet das Sktiptsystem
- source/0.6/RessourceManager.jl - Verwaltet Ressourcenpfade
- source/0.6/ThreadFunctions.jl - enthält erweiterte Thread Funktionen
- source/0.6/ThreadManager.jl - Verwaltet Threads
- source/0.6/TimeManager.jl - enthält Zeit Funktionen
- source/0.6/WindowManager.jl - Verwaltet das Programmfenster (GLFW)
- source/0.6/JLGEngine.jl - Verwaltet die Zuordnung der Skripts für die Engine
- source/0.6/JLGEngine/LibGL/ - enthält Skripte zu OpenGL, repräsentiert Schnittstelle OpenGL
- source/0.6/JLGEngine/ModelManager - einfache Gruppierung zur Übersicht enthält Skripte für Meshfunktionen wie z.B. das Laden von OBJ Dateien

Mit der GrafikEngine wird eine 3D Szene gebaut die verschiedene Objekte anzeigen kann. Zum Beispiel kann ein Würfel mit einem Mandelbrot Shader angezeigt werden. Ein anderer Würfel trägt eine Textur und ein weiterer Würfel tarnt sich als Kugel (durch einen Shadertrick). Transparente Objekte sind ebenfalls möglich. In der Szene kann der Betrachter die Kamera bewegen und sich umschauen. Die 3D Szene wird im Script **scripts/input.jl** beschrieben. Dort wird die Zuordnung von Objekten und Eigenschaften festgehalten. Bei einem Bearbeiten und Speichern der Datei wird das Programm automatsich die Szene neu Laden und die Veränderungen der Szene übernehmen. Die GrafikEngine ist so aufgebaut, dass für jedes logische Grafikobjekt wie z.B. Shader, Kamera, Mesh, Texture, usw. Manager Module existieren. Dadurch wird die Verwaltung einfacher und es können später optimierte Methoden hinzugefügt werden. Diese Skripte sind im **source/0.6/JLGEngine** Ordner zu finden. Die Renderszene wird durch ein Render Objekt (Renderer) verwaltet. Dieser enthält als Komponenten Eigenschaften (Meshdaten, Transformation, Texturen, usw.) für das jeweilige Objekt, dass gerendert werden soll. Das input.jl Script wird durch das eingebaute Skriptsystem verwaltet. Hierbei wird bei der Aktualisierung der input.jl Datei das zum Script repräsentierende Julia Modul überschrieben. Vor dem Aufruf des Renderprocesses werden Event Methoden (wie z.B. OnStart, OnUpdate) vordefiniert und Events werden an das Modul übergeben, sodass im Script die Behandlung von Events möglich ist. Events vom Programmfenster (GLFW) (z.B. OnKey) werden an das Script mitübergeben.

4.3 JuliaOpenGL

In diesem Projekt sind alle benötigten Skripte im Root Verzeichnis. Die Renderszene ist in der main.jl enthalten. In der Entwicklung wurde erkannt, dass der ursprungliche Ansatz des Grafikengine Konzepts für spezielle Fälle (wie Optimierung) ungeeignet ist. Eine sehr gute Grafikengine kann demnach nur über mehrere Entwicklungsphasen entstehen und wenn solche Spezialfälle mitberücksichtigt werden. In meinem Fall war es das Rendern von 128^3 Blöcken (Cubes). Mit der OpenGL Draw Methode glDrawArrays() lassen sich nur Werte von 1-6 FPS maximal auf meinem System erreichen. Demnach wurde hier ein Optmierungsverfahren geschrieben. Es wird die Methode glDrawElementsInstanced() verwendet, ein Vertex, Fragment und Geometry Shader, sowie zwei Algorithmen (**Frustum Culling** und **Outside Only**). Die zwei Algorithem reduzieren die Anzahl der Blöcke bzw. Vertices. Dadurch entstehen großzügige Werte über 100 FPS bei 128^3 Blöcken. Im Prinzip werden nur die sichtbaren Blöcke gerendert und nicht alle 128^3. Bei glDrawElementsInstanced werden Instanzen einer Vertex Gruppe (z.B. Cube) mehrfach gerendert. Die Transformationsdaten werden über einen Buffer mitgegeben. Durch diese Methode wird ein mehrfach Aufruf einer OpenGL Funktion in einer Schleife (wie z.B. glDrawArrays) vermieden und das verursacht eine Performancesteigerung beim Rendern. Das **Frustum Culling** Verfahren schneidet nur den sichtbaren Teil der Szene aus. Es repräsentiert die Kameraansicht (Viewport) und enthält sechs Ebenen (Oben, Unten, Rechts, Links, Vorne und Hinten). Nur innerhalb dieser Ansicht werden Objekte gerendert. Bei dem **Outside Only** Verfahren werden versteckte Blöcke, die von anderen Blöcken umgeben sind, nicht gerendert. Der

Geometry Shader ist standardmäßig beim mehrfachen Rendern schlechter als wenn Objekte mit festgelegten Strukturen (Vertices) gerendert werden. Allerdings ist er durch die Kombination von **Frustum Culling** und **Outside Only** Verfahren sehr stark, da nur noch Punktdaten an OpenGL übergeben werden müssen und erst bei Bedarf der Block bzw. eine Seite des Blocks erstellt wird. Dadurch lassen sich noch weniger Strukturen (Vertices) rendern.



4.4 Warum wurde nicht GLAbstraction.jl verwendet?

Die Bibliothek GLAbstraction.jl (https://github.com/JuliaGL/GLAbstraction.jl) war für meinen Ansatz eher ungeeignet, da sich die Bibliothek als Framework präsentiert und ich dadurch weniger nachvollziehen kann was eigentlich passiert. Da ich bereits eine GrafikEngine in C++ geschrieben und Kenntnisse mit nativen OpenGL Funktionen habe, wäre die Verwendung von GLAbstraction.jl für mich mit noch mehr Arbeit verbunden. Desweiteren bedient sich die Bibliothek momentan noch auf einer veralteten Bibliothek FixedSizeArrays.jl und dem Reactive.jl, was bei meinen Tests noch unzuverlässige Ergebnisse lieferte.

4.5 Optimierung

Es gibt die Möglichkeit mit Julia den GCC compiler auszuführen, um C-Code zu kompilieren. Der JuliaOptimizer im JuliaOpenGL Projekt wurde mit Visual Studio angelegt. Hier kann auch der Visual Stduio C++ Compiler verwendet werden, um optimierten Code zu schreiben. In Julia können über DLL-Schnittstellen C-Funktionen direkt angesprochen werden.

4.6 Ausführbare Dateien

Mithilfe meines Build Skriptes lassen sich ausführbare Dateien erstellen: https://github.com/Gilga/BuildExecutable.jl

Das BuildScript sollte im gleichen Verzeichnis sein wo sich die zwei Projekte befinden. Per Batchdatei (build-(version).bat) lassen sich ausführbare Dateien generieren. In der Batch-Datei sollte der Pfad zu Julia richtig sein. Nach der Ausführung sollte ein build/(Julia-Version)/(Projectname)/ Ordner existieren und dort befindet sich die kompielierte ausführbare Datei.

5 Probleme

Ein großer Nachteil ist, dass es standardmäßig keine klare Möglichkeit gibt Julia Skripte in ausführbaren Dateien (executables) umzuwandeln. Demnach hab ich eine Fremdbibliothek (BuildScript) verwendet und diese für meine Verhältnisse angepasst (Github forked). Design Schwierigkeiten hatte ich am Anfang, da es keine Klassen und Vererbung gibt. Verebung lässt sich über komponentenbasiertes Design lösen. Fehlermeldung können schwer nachvollziehbar werden, wenn eval verwendet wird, aber das ist auch in anderen Sprachen ein bekanntes Problem. Wenn viel Code geschrieben wird, kann es passieren, dass unnützer Code entsteht und nicht aufgerufen wird. Das ist in Julia ein noch größeren Problem als in anderen Sprachen, da evtl. dieser Code sogar typ-falsch oder nicht extistent sein kann. Erst bei einem Aufruf kann Julia Probleme feststellen, daher sind automatiesierte Tests und Dokumentation sehr wichtig.

Ein ganz anderes Problem ist in meinem GrafikEngine Projekt das Skriptsystem. Derzeit ist es möglich im Script auf all Julia Funktionen zurückzugreifen. Das hat den Nachteil, dass das ganze Programm zweckentfremdet und somit als Schadsoftware umgeschrieben werden kann. Als mögliche Lösung könnten die zu verwenden Julia Funktionen eingeschränkt werden.

6 Zusammenfassung

Anhand dieses Projektes konnte ich zeigen, dass trotz der Größe sich das Projekt in zwei Projekte aufspalten und entwickeln lies. Ich habe keine jahrelangen Erfahrungswerte mit Julia, dennoch lernte ich die Sprache sehr schnell kennen. Ich würde Julia als ernsthaften Konkurrenten gegenüber Python ansehen, allerdings Bedarf es an etwas noch mehr Entwicklungzeit und Wachstum der Community, damit mehr Bibliotheken entsehen und weniger neu geschrieben werden muss, wie es auch in meinem Fall war. Die Möglichkeit Julia Skripte in ausführbaren Dateien (executables) umzuwandeln, sollte ein Bestandteil des Core Systems sein. Bisher ist es nur über externe Libraries (wie z.B. meiner eigenen) möglich. Desweiteren gibt es in vielen Modulen noch Optimierungsbedarf, um die Ausführungszeit von Julia Code zu reduzieren. Im allgemeinen kann ich Julia für kleine bis mittelgroße Forschungs- und Entwicklungsprojekte empfehlen.

6.1 Wie gut sind Julia Projekte Skalierbar?

Für sehr große Projekte sehe ich noch Schwierigkeiten im Punkt Verlässichkeit. Neue Julia Versionen könnten mit den bisherigen Frembibliotheken aus dem Julia Modul Verzeichnis der Community nicht mehr funktionieren, das hängt mit der schnellen Entwicklung von Julia und der Community zusammen (derzeit gibt es Version 0.6). Wer später von Version 0.6 auf Version 1.0 umsteigen will kann möglicherweise ein neues Projekt beginnen. Für kleine und mittelgroße Projekte sehe ich bei Julia starke Vorteile gegenüber anderen Sprachen wie Python, Java oder sogar C++. Das hängt vorallem mit den Vorteilen der einfachen Syntax, Möglichkeit der Metaprogramming und den bereits bestehenden Bibliothelen für wie z.B. Images.jl (zum Anzeigen von Bilder), Plots.jl (für Diagramme) und anderen zusammen. Julias findet vor allem Anwendung in Forschungsprojekten. Für weitere Details siehe Punkt Basics - Was ist Julia? - Zusammenfassung der Features.

Dieses Dokument wurde mit LateX erstellt von Mario Link (s0536176).

Berlin, 16. April 2018