Visualisierung von dynamischen Aufrufgraphen mit VR-basierten Software-Städten

Bachelorarbeit

Torben Groß
Matrikelnummer: 4223219

19.01.2020

Fachbereich Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. rer.nat. Rainer Koschke
2. Gutachter: Prof. Dr. Gabriel Zachmann

Erklärung

Ich versichere, die Bachelorarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 19.01.2020

.....
(Torben Groß)

Danksagung

Ich bedanke mich bei allen Menschen, die mich beim Verfassen dieser Bachelorarbeit unterstützt haben.

Zuerst möchte ich Prof. Dr. rer.nat. Rainer Koschke dafür danken, dass ich bei ihm meine Bachelorarbeit schreiben durfte und er mir stets bei Problemen mit den richtigen Hilfestellungen geholfen hat. Zusätzlich bedanke ich mich bei M.Sc. Marcel Steinbeck für seine inhaltliche Unterstützung. Ein besonderer Dank gilt meinen Freunden und meiner Familie, die mir immerzu unter die Arme gegriffen und mich in schwierigen Phasen moralisch unterstützt haben. Zu guter Letzt möchte ich allen anonymen Probanden danken, die sich die Zeit und Mühe gemacht haben, an meiner Evaluation teilzunehmen.

Ich werde in dieser Arbeit aus Gründen der Einfachheit das generische Maskulinum verwenden, beziehe mich aber stets ausdrücklich auf beide Geschlechter.

Zusammenfassung

Im Kontext dieser Arbeit ist eine Visualisierung von dynamischen Aufrufgraphen mit Software-Städten entstanden. Diese wurde mithilfe der Unity Engine umgesetzt und lässt sich sowohl am Desktop als auch in Virtual Reality verwenden.

Diese Arbeit stellt verschiedene Datenformate vor und vergleicht diese bezüglich ihrer Tauglichkeit zum Repräsentieren von dynamischen Aufrufgraphen. Ergänzend wird diskutiert, ob diese Datenformate während der dynamischen Analyse zum zügigen Abspeichern neuer Einträge geeignet sind, ohne dabei das Laufzeitverhalten zu stark zu beeinflussen. Es wird sich anschließend auf ein geeignetes Format festgelegt, welches in der folgenden Implementierung verwendet wird.

Zusätzlich befasst sich diese Arbeit mit verschiedenen Methoden, um dynamische Aufrufgraphen mittels dynamischer Analyse zu generieren. Hierbei werden Methoden wie beispielsweise Instrumentierung für Java und C++ beschrieben.

Des weiteren werden unterschiedliche Methoden der Visualisierung von dynamischen Aufrufgraphen mit Software-Städten illustriert. Es werden Vor- und Nachteile abgewägt und anschließend wird sich für die nachfolgende Implementierung auf eine Methode der dynamischen Visualisierung festgelegt.

Es werden die Implementierungsdetails von verschiedenen Methoden der Datenakkumulation präsentiert, um dynamische Aufrufgraphen zu generieren. Zusätzlich wird im Bezug auf die Unreal Engine 4 und die Unity Engine eine eigene Implementierung sowohl für das Einlesen, als auch für die Visualisierung von Aufrufgraphen vorgestellt.

Im Kontext dieser Arbeit wurde eine Evaluation durchgeführt. Es wurde untersucht, ob signifikante Unterschiede bei der Verwendungseffizienz zwischen Desktop und Virtual Reality existieren. Es konnte mit einer Konfidenz von 80% gezeigt werden, dass sich für Virtual Reality die Erwartungswerte des ersten und zweiten Softwaresystems unterscheiden. Dies konnte für die Desktop-Version nicht gezeigt werden. Somit kann vermutet werden, dass Virtual Reality signifikante Vorteile bei der Orientierung innerhalb einer Software-Stadt bietet. An der Evaluation haben insgesamt 10 Probanden teilgenommen. Aufgrund der geringen Probandenzahl ist die Aussagekraft der Studie einzuschränken.

Zu guter Letzt werden mögliche auf diese Arbeit aufbauende Forschungsthemen dargelegt. Interessant zu evaluieren wäre zum einen die Integrierung von Software-Städten in Augmented Reality. Dies könnte die Lücke zwischen der Software-Stadt und dem tatsächlichen Quellcode weiter schließen. Zusätzlich werden weitere Methoden der Visualisierung beschrieben, welche aufbauend auf diese Arbeit evaluiert werden könnten.

INHALTSVERZEICHNIS

KAPITEL 1

Einleitung

Softwaresysteme werden immer größer und komplizierter. Abbildung 1.1 visualisiert die Entwicklung der SLOC (Source lines of code) vom Linux Kernel[1] [2] [3] [4] und vom Betriebssystem Debian[5] [6] [7] im Laufe der Jahre. Das rasante Wachstum der beiden Softwaresysteme ist allerdings keine Ausnahme der Regel. Generell scheint der Trend zu sein, dass Softwaresysteme immer komplexer werden. Doch dieses Wachstum geht nicht ohne daraus resultierenden negativen Nebenwirkungen einher. Es kann gezeigt werden, dass es einen Zusammenhang zwischen der Größe eines Softwaresystems und dessen Verständlichkeit gibt. Tashtoush et al. untersuchten 2014 Korrelationen zwischen verschiedenen Software-Metriken und analysierten im Laufe der Studie fünf öffentlich einsehbare Softwaresysteme. Dabei konnte gezeigt werden, dass eine klare Korrelation zwischen LOC (Lines of code) und der zyklomatischen Komplexität existiert.[8] Je größer ein Softwaresystem also wird, desto schwieriger scheint das Begreifen des Gesamtsystems zu sein.

Um die Verständlichkeit komplexer Systeme zu verbessern, wird aktuell an den unterschiedlichsten Methoden geforscht. Ein interessantes Forschungsgebiet ist hierbei die sogenannte Software-Visualisierung. Dabei werden Softwaresysteme abstrahiert visualisiert, mit dem Ziel, einen Überblick über das gesamte Softwaresystem oder dessen Subsysteme zu vermitteln.

Bei der Software-Visualisierung wurde in den letzten Jahren vermehrt mit Virtual Reality experimentiert. Durch wie Verwendung von VR wird sich eine erhöhte Immersion und eine bessere Orientierung erhofft. Weitergehend kann Virtual Reality mit der von Wettel et al. vorgestellten Stadtmetapher kombiniert werden. Im Kontext der Software-Visualisierung haben Wettel et al. im Jahre 2007 erstmalig Softwaresysteme als Stadt visualisiert. Dabei wurden Klassen als Gebäude und Pakete als Straßen statisch dargestellt.[9]

[1] Wheeler, D. A., *More Than a Gigabuck: Estimating GNU/Linux's Size*, `https://dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`, 2001, Zugriff: 12.01.2020.

[2] Leemhuis, T., *What's new in Linux 2.6.32*, `https://web.archive.org/web/20131219054613/http://www.h-online.com/open/features/What-s-new-in-Linux-2-6-32-872271.html?view=print`, 2009, Zugriff: 12.01.2020.

[3] Kroah-Hartman, G. et al., *Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*, The Linux Foundation, 2012.

[4] Leemhuis, T., *What's new in Linux 3.6*, urlhttps://web.archive.org/web/20131219054847/http://www.h-online.com/open/features/What-s-new-in-Linux-3-6-1714690.html?page=3, 2012, Zugriff: 12.02.2020.

[5] Leemhuis, T., *Linux-Kernel durchbricht die 20-Millionen-Zeilen-Marke*, `https://www.heise.de/newsticker/meldung/Linux-Kernel-durchbricht-die-20-Millionen-Zeilen-Marke-2730780.html`, 2015, Zugriff: 12.01.2020.

[6] González-Barahona, Jesús M. et al., *Counting potatoes: the size of Debian 2.2*, 2008.

[7] Robles, G., *Debian Counting*, 2013.

[8] Tashtoush, Y. et al., *The Correlation among Software Complexity Metrics with Case Study*, `https://arxiv.org/ftp/arxiv/papers/1408/1408.4523.pdf`, International Journal of Advanced Computer Research, Vol.4, Nr. 2, 2014, S. 417 f., Zugriff: 13.01.2020.

[9] Wettel, R. et al., *CodeCity: 3D Visualization of Large-Scale Software*, 2008, S. 3.

Gegenstand der Forschung ist allerdings nicht ausschließlich das Verstehen der statischen Softwarearchitektur, sondern zusätzlich des dynamischen Laufzeitverhaltens. Bezüglich dessen ergibt sich die interessante Fragestellung, ob Virtual Reality und Software-Städte zu einer dynamischen Visualisierung kombiniert werden können. Diese Arbeit wird sich genau mit dieser Frage auseinander setzen und eine auf Software-Städte aufbauende dynamische Visualisierung von Aufrufgraphen präsentieren. Zusätzlich wird im Sinne dieser Visualisierung Virtual Reality auf mögliche Vorteile gegenüber der klassischen Anwendung am Desktop verglichen.

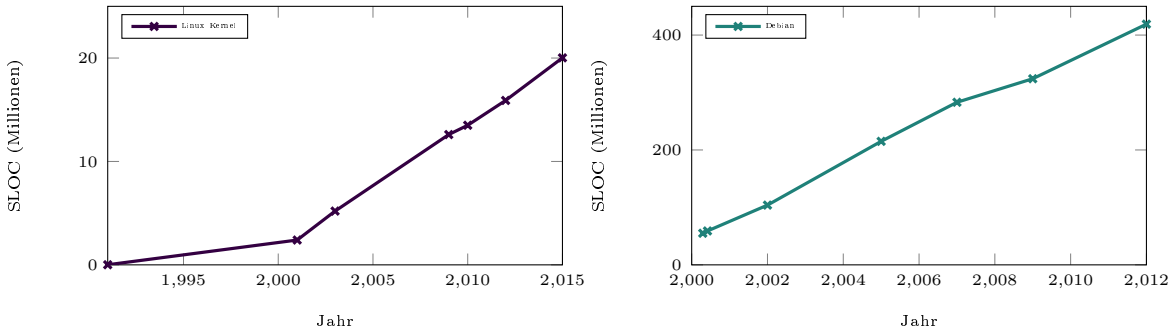


Abbildung 1.1: SLOC vom Linux Kernel und von Debian

1.1 Aufbau

Kapitel 2 wird zunächst einige für diese Arbeit relevante Grundlagen vermitteln. In Kapitel 3 werden daraufhin verschiedene Verfahren für die dynamische Analyse und die Visualisierung derer Ergebnisse vorgestellt. Hierbei werden auf Vor- und Nachteile abgewägt und einige Designentscheidungen getroffen. Anschließend wird in Kapitel 4 auf die tatsächlichen Implementierungsdetails der Software eingegangen. Kapitel 5 beschreibt darauf folgend die Evaluation bezüglich Aufbau, Durchführung und Ergebnisse. Zu guter Letzt wird in Kapitel 6 ein Ausblick auf mögliche weitere interessante Fragestellungen gegeben.

KAPITEL 2

Grundlagen

2.1 Virtual Reality

Bei Virtual Reality, oder auch virtuelle Realität, handelt es sich um ein simuliertes Erlebnis, welches durch ein Head-Mounted Display erlebt wird.[1] Oftmals findet Virtual Reality Anwendung in der Spielindustrie, aber auch in anderen Bereichen findet VR immer mehr Anklang. Selbst in der Medizin wird bereits mit VR geforscht.[2]

Laut Steuer müsse zum Definieren von Virtual Reality zunächst der Begriff der *Präsenz* definiert werden. Präsenz ließe sich, so Steuer, als die Erfahrung der physischen Umgebung beschreiben. Hierbei beziehe sich die Erfahrung nicht auf die physische Welt, sondern auf die Wahrnehmung.[3] Zusätzlich definiert Steuer den Begriff der *Telepräsenz* durch einen Vergleich mit dem Begriff der Präsenz. Präsenz sei die natürliche Perzeption der Umgebung, wohingegen Telepräsenz die modellierte Perzeption einer Umgebung beschreibe. Diese Umgebung könne eine animierte, aber nichtexistierende virtuelle Welt sein.[4] Virtual Reality lässt sich also als die Wahrnehmung einer künstlichen und simulierten Welt beschreiben.

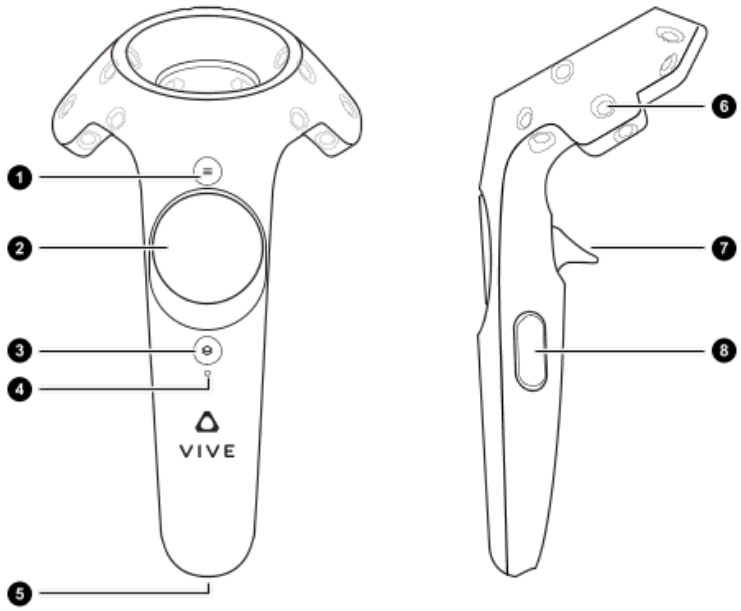
2.1.1 HTC Vive

Die HTC Vive ist ein von HTC und Valve entwickeltes VR-Headset, mit der die virtuelle Realität erlebt werden kann. Für die Evaluation wird die HTC Vive verwenden, aber generell wäre jedes beliebige Head-Mounted Display für die Evaluation tauglich.

2.1.1.1 Controller

Die Controller der HTC Vive sind kabellos und haben eine Vielzahl von verschiedenen Knöpfen, welche für die Steuerung der Software relevant sein werden. Abbildung 2.1 ordnet den verfügbaren Elementen der Controller die jeweiligen Namen zu, wie sie auch auf der offiziellen Website der Hersteller zu finden sind.

[1] Steuer, J., *Defining Virtual Reality: Dinemsions Determining Telepresence*. 1993. S. 5.
[2] Weyhe, D. et al., *Immersive Anatomy Atlas: Empirical Study Investigating the Usability of a Virtual Reality Environment as a Learning Tool for Anatomy*, 2018.
[3] Steuer, J., *Defining Virtual Reality: Dinemsions Determining Telepresence*. 1993. S. 5.
[4] ebd., 6.



1 Menü Taste
2 Trackpad
3 System Taste
4 Statuslampe
5 Micro-USB-Anschluss
6 Verfolgungssensoren
7 Trigger
8 Griff Taste

Abbildung 2.1: Controller der HTC Vive.[5]

2.2 Visualisierung

Schumann et al. definieren 2013 das Ziel der Visualisierung folgendermaßen:

> "Die wissenschaftlich-technische Visualisierung, auch als *Scientific Visualization* bezeichnet, hat sich Aufgabe, geeignete visuelle Repräsentationen einer gegebenen Datenmenge zu erzeugen, um damit eine effektive Auswertung zu ermöglichen."[6]

Laut Schumann et al. soll mithilfe von Visualisierung die Analyse, das Verständnis und die Kommunikation von Modellen, Konzepten und Daten erleichtert werden.

2.2.1 Softwarevisualisierung

Der Begriff der Softwarevisualisierung bezieht sich auf die Visualisierung von Softwaresystemen und ist somit eine Spezialisierung der Visualisierung. Stephan Diehl beschreibt 2007 Softwarevisualisierung als Visualisierung der Artefakte bezüglich Software und dessen Entwicklungsprozess. Zusätzlich zum Programmcode seien hierbei beispielsweise Voraussetzungen, Designdokumente, Änderungen zum Quellcode, aber auch Fehlerreports Teil der Artefakte. Laut Diehl beschäftigen sich Forscher mit der Visualisierung von *Struktur*, *Verhalten* und *Evolution* der Software.

Struktur beziehe sich hierbei auf den statischen Anteil des Systems, also beispielsweise den statischen Aufrufgraph oder die Organisation der Software in Module.

[5] *Über die VIVE Controller*, `https://www.vive.com/de/support/vive/category_howto/about-the-controllers.html`, Zugriff: 28.12.2019.

[6] Schumann, H. et al., *Visualisierung: Grundlagen und allgemeine Methoden*, Springer-Verlag Berlin Heidelberg 2000, 2013, S. 5.

Verhalten beziehe sich auf die Ausführung des Programms. Die Ausführung könne auf einer höheren Abstraktionsebene gesehen werden wie beispielsweise Funktionen, die andere Funktionen aufrufen.[7]

Evolution beziehe sich auf den Entwicklungsprozess des Softwaresystems und betone die Tatsache, dass Software sich über die Zeit verändert.[8]

Im Kontext dieser Arbeit wird eine Visualisierung bezüglich des Verhaltens von Softwaresystemen entwickelt. Diese Visualisierung baut auf eine bereits gegebene Struktur-Visualisierung auf.

2.2.2 Software-Städte

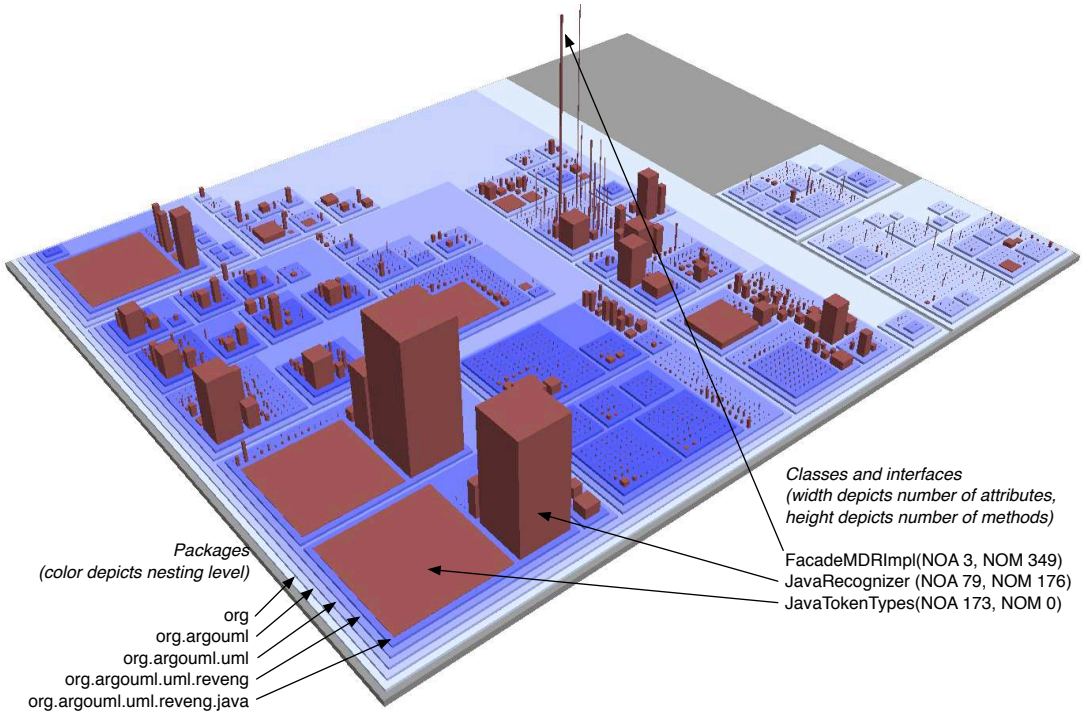


Abbildung 2.2: Überblick der Code City von ArgoUML v.0.24.[9]

Eine Software-Stadt, oder auch Code City ist eine auf der Stadtmetapher von Wettel et al. aufbauende Methode der Softwarevisualisierung. Laut Wettel et al. sei der Hauptgedanke der Stadtmetapher, objektorientierte Konstrukte wie Pakete, Klassen, Methoden, Attribute und Relationen visuell darzustellen. Hierbei, so Wettel et al., werden Klassen als Gebäude repräsentiert. Diese befinden sich in Stadtteilen, welche Pakete repräsentieren.

Die Wahl dieser Metapher habe mehrere Gründe. Zum einen sei eine Stadt eine bekannte Vorstellung mit klarem Konzept. Außerdem sei eine Stadt an sich immer noch recht komplex. So könne sie, ebenso wie auch Software inkrementell verstanden werden. Zu starke Abstraktionen bzw. Vereinfachungen werden somit ausgeschlossen. Weitergehend, so Wettel et al., seien Klassen und Pakete Schlüsselelemente der objektorientierten Softwareentwicklung und

[7] Diehl, S., *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*, 2007, S. 3.
[8] ebd., 4.
[9] Wettel, R. et al., *CodeCity: 3D Visualization of Large-Scale Software*, 2008, S. 3.

somit ein primärer Faktor der Orientierung von Softwareentwicklern.[10]

2.2.3 Gestaltpsychologie

Die Gestaltpsychologie beschäftigt sich mit der grundlegenden menschlichen Fähigkeit, Muster visuell wahrzunehmen. 1923 formulierte Wertheimer, welcher als einer der Hauptbegründer des Begriffs der Gestaltpsychologie gilt, sechs wesentliche Gestaltgesetzte.[11] Im folgenden Abschnitt wird auf diese spezifischer eingegangen.

2.2.3.1 Gestaltgesetze

2016 fasste Udo Frese in der Lehrveranstaltung *Grundlagen der Medieninformatik 1* fünf der sechs Gestaltgesetze folgendermaßen zusammen:

1. Gesetz der Nähe

Das Gesetz der Nähe drückt aus, dass Elemente mit geringem Abstand als zusammengehörig wahrgenommen werden.[12]

2. Gesetz der Ähnlichkeit / Gesetz der Gleichheit

Elemente, die ähnlich sind, werden als zusammengehörig wahrgenommen. Hierbei spielen beispielsweise Farbe, Helligkeit, Größe, aber auch Orientierung oder Form eine Rolle.[13]

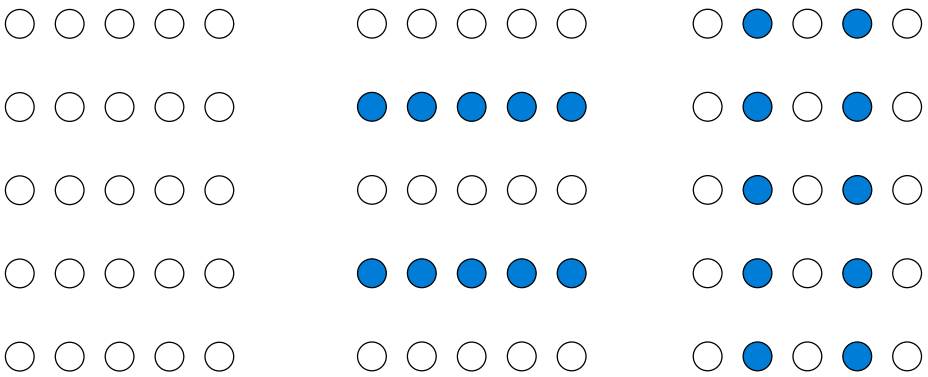


Abbildung 2.3: Gesetz der Nähe und Gesetz der Gleichheit.[14]

3. Gesetz der guten Fortsetzung

Elemente, die räumlich oder zeitlich einheitlich einem Schema folgen, wirken zusammengehörig.[15]

4. Gesetz der Schließung

[10] Wettel, R. et al., *Visualizing Software Systems as Cities*, 2007, S. 2.
[11] Wertheimer, M., *Untersuchung zur Lehre von der Gestalt (II)*, 1923.
[12] Frese, U., *Grundlagen der Medieninformatik 1: Menschliche Wahrnehmung*, 2016, S. 18.
[13] ebd., 19.
[14] ebd., 20.
[15] ebd., 21.

Unvollständige Konturen werden vervollständigt, weshalb auch verdeckte Konturen als Ganzes gesehen werden können.[16]

5. Gesetz der Symmetrie

Symmetrische Zwischenräume werden als Figur, asymmetrische Zwischenräume als Hintergrund erkannt.[17]

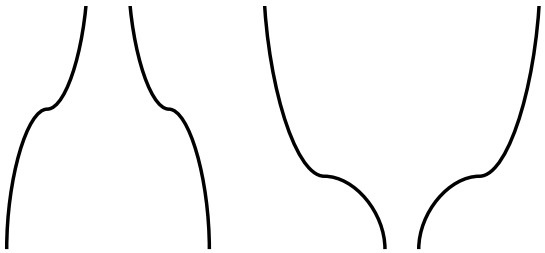


Abbildung 2.4: Gesetz der Symmetrie.[18]

2.3 Software Reengineering

Im Folgenden werden zunächst in Abschnitten 2.3.1 und 2.3.2 die Begriffe *Forward Engineering* und *Reverse Engineering* definiert, um darauf aufbauend in Abschnitt 2.3.3 den Begriff des *Reengineering* definieren zu können. Anschließend geht Abschnitt 2.3.4 auf die dynamische Software-Analyse und Abschnitt 2.3.5 auf Instrumentierung von Softwaresystemen ein.

2.3.1 Forward Engineering

Der Begriff des *Forward Engineering* beschreibe, so Chikofsky et al., den traditionellen Prozess von einer hohen Abstraktionsebene und einem logischen, implementierungsunabhängigem Design ausgehend zur tatsächlichen Implementierung eines Systems.[19]

2.3.2 Reverse Engineering

Chikofsky et al. beschreiben *Reverse Engineering* als den Prozess des Analysieren eines Systems um

- die Komponenten und Relationen des Systems zu identifizieren und
- Repräsentationen des Systems in einer anderen Form oder höheren Abstraktionsebene zu erzeugen.

[16] ebd., 23.
[17] ebd., 24.
[18] ebd.
[19] Chikofsky, E. et al., *Reverse Engineering and Design Recovery: A Taxonomy*, 1990, S. 14 f.

Oftmals werde ein existierendes System analysiert. Dies sei allerdings nicht notwendigerweise immer der Fall. Als Ausgang vom Reverse Engineering könne, so Chikofsky et al., jede beliebige Abstraktionsebene dienen.[20]

2.3.3 Reengineering

Den Begriff des *Reengineering* definieren Chikofsky et al. folgendermaßen:

> "Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."[21]

Reengineering lässt sich also als Begutachtung mit anschließender Modifizierung eines Systems verstehen, mit dem Ziel, dieses in eine neue Form zu bringen.

Laut Chikofsky et al. inkludiere Reengineering generell eine Form des Reverse-Engineering, um eine abstraktere Beschreibung zu erzeugen. Darauf folge für gewöhnlich eine Form von Forward-Engineering oder Restrukturierung. Während des Prozesses werden oftmals Modifikationen aufgrund neuer Anforderungen vorgenommen.[22]

2.3.4 Dynamische Software-Analyse

Bei einer dynamischen Software-Analyse wird ein Softwaresystem während der Ausführung beobachtend analysiert. Laut Safyallah et al. werden dabei Bereiche wie Optimierungen der Performanz, Visualisierung der Softwareausführung und das Bestimmen, welche Codeabschnitte für welche Features verantwortlich sind, abgedeckt. Eine dynamische Analyse habe im Vergleich zur statischen Programmanalyse allerdings einige Herausforderungen. Zum einen generiere die statische Analyse lediglich eine kleine Teilmenge aller möglichen Programmverläufe. Die statische Analyse hingegen generiere stets eine vollständige Menge an Informationen. Zum anderen sei die Gewinnung aussagekräftiger Informationen eine schwierige Aufgabe. Zusätzlich können Konstrukte wie Schleifen oder Rekursionen, so Safyallah et al., die gesamte dynamische Analyse stören.[23]

2.3.5 Instrumentierung

Laut Quante et al. verstehe man unter dem Begriff der *Instrumentierung* das Einfügen von zusätzlichem Code in ein Programm. Dieser zusätzliche Code soll den Programmstatus ausgeben oder Informationen sammeln, welche zu einem bestimmten Punkt notwendig sein werden. Der Vorteil dieser Variante sei, dass diese Methode in jeder Umgebung funktioniert. Allerdings werde Quellcode modifiziert, so dass das Programm nicht mehr dem Ursprünglichen gleiche.

Hierbei seien drei Stufen der Instrumentierung möglich:

- *Quellcode*: Einfügen von zusätzlichen Code direkt in den Quellcode
- *Zwischendarstellung*: Einfügen von zusätzlichen Knoten und Kanten in die Zwischendarstellung

[20] ebd., 15.
[21] ebd.
[22] ebd.
[23] Safyallah, H. et al., *Dynamic Analysis of Software Systems using Execution Pattern Mining*, 14th IEEE International Conference on Program Comprehension, 2006, S. 1.

- *Binärcode*: Einfügen von zusätzlichen Code in den binären Code des ausführbaren Programms[24]

Für diese Arbeit sind im Folgenden die erste und dritte Stufe der Instrumentierung relevant. Die folgende Java-Instrumentierung (siehe Abschnitt 3.3.1) modifiziert Binärcode und bei der C++-Instrumentierung (siehe Abschnitt 3.3.3) wird Quellcode modifiziert.

2.4 GXL - Graph eXchange Language

GXL (Graph eXchange Language) wurde als Standard-Austauschformat für Graphen entwickelt. Bei GXL handelt es sich um einen XML-Dialekt. Insbesondere wurde GXL entwickelt, um die Interoperabilität zwischen Software Reengineering Tools und Komponenten zu ermöglichen.[25]

Im Kontext dieser Arbeit wird mit diesem Format bei der Generierung von dynamischen Aufrufgraphen experimentiert.

2.5 Game Engines

Game Engines sind Software-Pakete, welche als Unterstützung beim Entwickeln von Videospielen dienen. Die hauptsächliche Funktionalität von Game Engines ist das Rendern von graphischen Elementen. Game Engines enthalten allerdings oftmals viele weitere Subsysteme wie eine Physik-Engine inklusive Kollisionsdetektion, Soundunterstützung und Mehrspielerunterstützung, um Entwicklern bei der Arbeit noch besser aushelfen zu können. Sharrod definiert Game Engines 2007 als:

> "a framework comprised of a collection of different tools, utilities, and interfaces that hide the low-level details of the various tasks that make up a video game."[26]

Game Engines unterstützen allerdings nicht nur bei der Entwicklung von Videospielen. Viele graphische Applikationen können von der Verwendung einer Game Engine profitieren. Auch die vielen zusätzlichen Funktionalitäten machen Game Engines oftmals attraktiv für Entwickler aus anderen Anwendungsbereichen. Befasst man sich mit Game Engines, so liest man immer wieder die selben zwei Namen - Unreal Engine 4 und Unity. Da in dieser Arbeit mit beiden Game Engines experimentiert wird, gehen die folgenden Abschnitte auf beide Systeme detaillierter ein.

2.5.1 Unreal Engine 4

Die Unreal Engine ist eine von Epic Games veröffentlichte Game Engine. Die erste Version wurde von Tim Sweeney, dem Gründer von Epic Games entwickelt. 1995 begann Sweeney mit der Entwicklung und veröffentliche die Game Engine 1998 zusammen mit dem Ego-Shooter-Spiel *Unreal*. Seitdem wurden vielzählige Videospiele mithilfe der Unreal Engine entwickelt.

[24] Quante, J. et al., *Dynamic Object Process Graphs*, Journal of Systems and Software, 2008, S. 3.
[25] *Graph eXchange Language*. `http://www.gupro.de/GXL/Introduction/background.html`, Zugriff: 01.01.2020.
[26] Sherrod, A., *Ultimate 3D Game Engine Design & Architecture*, 2007.

Die neuste Version wurde 2014 unter dem Namen der Unreal Engine 4 (kurz UE4) veröffentlicht.[27] [28]

Die UE4 wurde in C++ geschrieben. Die ersten Versionen der UE4 unterstützten keine Skriptsprache. Seit der Version 4.20 ist allerdings Phython[29] als Skriptsprache in die Game Engine integriert. Zusätzlich können Entwickler Code direkt in C++[30] schreiben oder auf die visuelle Programmiersprache der Blueprints[31] zurückgreifen. Mit der UE4 wurde der sogenannte Unreal Engine Marketplace eröffnet. Bei diesem handelt es sich um eine Website, welche es Entwicklern und Künstlern ermöglicht, Assets oder andere UE4-spezifische Produkte zum Verkauf anzubieten.

2.5.2 Unity Engine

Die Unity Engine wurde ursprünglich im Jahre 2005 als eine Mac OS X-exklusive Game Engine angekündigt. Im Laufe der Jahre wurden von Unity allerdings immer mehr und mehr unterschiedliche Plattformen unterstützt, sodass Unity mittlerweile als eine platformübergreifende Game Engine angesehen werden kann.[32]

Wie auch die UE4 unterstützt Unity eine Skriptsprache. Im Gegensatz zur UE4 wird hierbei allerdings C# verwendet.[33] Dafür wird von der aktuellsten stabilen Version (stand 09.01.2020) keine visuelle Programmiersprache wie die Blueprints aus der Unreal Engine 4 unterstützt. Laut der offiziellen Roadmap wird allerdings bereits an einer visuellen Programmiersprache entwickelt.[34] Seit 2010 gibt es den so genannten Unity Asset Store, welcher stark die Funktionalität des Unreal Engine Marketplace ähnelt. Auch hier können Assets oder andere spezifische Produkte zum Verkauf angeboten werden.

[27] Edwards, B., *From The Past To The Future: Tim Sweeney Talks*, `https://www.gamasutra.com/view/feature/4035/from_the_past_to_the_future_tim_.php`, 2009, Zugriff: 09.01.2020.

[28] Horvath, S., *The Imagination Engine: Why Next-Gen Videogames Will Rock Your World*, `https://www.wired.com/2012/05/ff-unreal4/`, 2012, Zugriff: 09.01.2020.

[29] *Scripting the Editor using Phython*, `https://docs.unrealengine.com/en-US/Engine/Editor/ScriptingAndAutomation/Python/index.html`, Zugriff: 09.01.2020.

[30] *Unreal Engine API Reference*, `https://docs.unrealengine.com/en-US/API/index.html`, Zugriff: 09.01.2020.

[31] *Blueprints Visual Scripting*, `https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html`, Zugriff: 09.01.2020.

[32] Axon, S., *Unity at 10: For better - or worse - game development has never been easier*, 2016, Zugriff: 09.01.2020.

[33] *Scripting*, `https://docs.unity3d.com/Manual/ScriptingSection.html`, Zugriff: 09.01.2020.

[34] *Unity Roadmap*, `https://unity3d.com/unity/roadmap`, Kategorie: Research, Zugriff: 09.01.2020.

KAPITEL 3

Entwurf

3.1 Anforderungen der Software

Die Grundaufgabe der zu erstellenden Software ist das Visualisieren von dynamischen Aufrufgraphen. Die dynamische Visualisierung soll auf statischen Software-Städten aufbauen. Um diese Aufgabenstellung erwartungsgemäß zu erfüllen, müssen zunächst einige Anforderungen an das zu entwickelnde Softwaresystem gestellt werden.

Bereits bei der Wahl des Datenformats lassen sich einige Anforderungen feststellen. Zum einen soll die Dateigröße möglichst klein gehalten werden. Dies ist insbesondere relevant, da für eine vollständige Analyse pro Sekunde hunderte oder gar tausende Funktionsaufrufe dokumentiert werden können. Weiterhin ist es wichtig, dass Dateien mit dem gewählten Format in Echtzeit durch weitere Einträge ergänzt werden können, ohne die Laufzeit der analysierten Software übermäßig auszudehnen.

Auch bei der Datenakkumulation sind einige Anforderungen zu formulieren. Die Analyse soll den dynamischen Aufrufgraphen ohne Lücken generieren können. Zusätzlich soll das Laufzeitverhalten der Software möglichst wenig beeinflusst werden.

Bei der Visualisierung ist gewünscht, dass diese besonders leicht zu verstehen ist. Zusätzlich soll immerzu der gesamte Aufrufstapel dargestellt werden. Dabei soll die Richtung des Kontrollflusses immer leicht zu erkennen sein. Um den Programmverlauf besser verfolgen zu können, soll zusätzlich, ähnlich wie bei einem Debugger, schrittweise durch die Software gelaufen werden können. Hierbei soll der Benutzer vor, aber auch zurück gehen können. Jeder Schritt nach vorne oder hinten soll dabei immer direkt zum nächsten bzw. vorherigen Funktionsaufruf springen. Somit unterscheidet sich die erwünschte Funktionalität leicht von der eines Debuggers. Weiterhin soll die Software beliebige Ergebnisse von Software-Analysen visualisieren können. Es ist also gewünscht, dass die Visualisierung weitgehend unabhängig von der analysierten Programmiersprache agiert.

Da zusätzlich eine Evaluation durchgeführt werden soll, sollen die Unterschiede bei der Anwendung zwischen VR und Desktop möglichst gering gehalten werden.

In den folgenden Abschnitten wird die zu entwickelnde Software entworfen. Hierbei werden unterschiedliche Ansätze vorgestellt und anhand der Anforderungen verglichen. Schlussendlich wird sich auf einige der Ansätze festgelegt.

3.2 Wahl des Datenformats

In den folgenden Abschnitten werden verschiedene potentielle Datenformate vorgestellt und auf ihre Tauglichkeit für die Generierung von dynamischen Aufrufgraphen zur Laufzeit un-

tersucht. In Abschnitt 3.2.5 wird ein Fazit gezogen und die die Wahl des Datenformats weiter eingeschränkt.

3.2.1 GXL

Bereits in dem Ausgangsprojekt wurden GXL-Dateien verwendet, um aus den darin gespeicherten Graphen statische Software-Städte zu generieren. Für dynamische Aufrufgraphen kann das GXL-Format ebenfalls verwendet werden. Dabei können die Funktionen als Knoten und die Aufrufe als Kanten repräsentiert werden.

Ein Vorteil von GXL ist, dass sehr einfach neue Funktionsaufrufe hinzugefügt werden können. Es müssen lediglich zwei neue Knoten (`node`) für die Quelle und das Ziel des Aufrufs und eine neue Kante (`edge`), die beide verbindet, angehängt werden. Auch können hierbei unterschiedliche Attribute (`attr`) wie beispielsweise der Funktionensname bei Knoten oder die Funktionslaufzeit bei Kanten festgehalten werden. Ein weiterer großer Vorteil ist weite Verbreitung des Formats. Dies hat zur Folge, dass viele Bibliotheken das Format bereits unterstützen und somit ohne große Anpassungen verwendet werden können.

Jeder Eintrag in der Datei nimmt allerdings einige Zeilen an Platz ein und enthält Schlüsselwörter unnötig häufig, weshalb die Dateigröße für große Aufrufgraphen zügig anwächst. Zusätzlich kann gefordert sein, dass innerhalb der Datei stets alle Knoten vor den Kanten aufgelistet werden müssen. Dies hätte zur Folge, dass beim Einfügen eines neues Knotens ein großer inhaltlicher Teil der Datei nach hinten verschoben werden muss. Probleme können zusätzlich entstehen, wenn gewünscht ist, dass jede Funktion maximal einmal als Knoten hinterlegt wird. Dafür müsste sich also entweder gemerkt werden, welche Funktion bereits hinterlegt wurde oder die GXL im Anschluss an die dynamische Analyse gefiltert werden. Das Einfügen neuer Elemente erweist sich durch einige Einschränkungen also doch als aufwendiger, als vorher angenommen. All diese Nachteile schränken die Tauglichkeit des Formats für eine dynamische Analyse ein.

Listing 3.1: Beispiel eines Aufrufgraphen im GXL-Format

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
3  <gxl xmlns:xlink="http://www.w3.org/1999/xlink">
4    <graph id="Example" edgeids="true">
5      <node id="N1">
6          <type xlink:href="Method"/>
7          <attr name="Linkage.Name">
8              <string>R example.cpp:main(i,c**)_c->i</string>
9          </attr>
10     </node>
11     <node id="N2">
12         <type xlink:href="Method"/>
13         <attr name="Linkage.Name">
14             <string>R example.cpp:Foo:foo()->i</string>
15         </attr>
16     </node>
17     <node id="N3">
18         <type xlink:href="Method"/>
19         <attr name="Linkage.Name">
20             <string>R example.cpp:Boo:bar()->i</string>
21         </attr>
22     </node>
23     <edge id="E1" from="N1" to="N2">
24       <type xlink:href="FunctionCall"/>
25     </edge>
26     <edge id="E2" from="N2" to="N3">
27       <type xlink:href="FunctionCall"/>
28     </edge>
29     <edge id="E3" from="N1" to="N2">
```

```
30        <type xlink:href="FunctionCall"/>
31      </edge>
32      <edge id="E4" from="N1" to="N2">
33        <type xlink:href="FunctionCall"/>
34      </edge>
35    </graph>
36  </gxl>
```

3.2.2 CSV

Seit Visual Studio 2015 lassen sich die Ergebnisse des in die IDE integrierten Profilers als Aufrufstrukturansicht anzeigen.[1] Hier sind essentielle Samplingdaten wie Funktionsname und Tiefe in der Aufrufstruktur, aber auch andere interessante Daten wie Anzahl von inklusiven und exklusiven Samplings enthalten.[2] Nach dem Erstellen einer solchen Ansicht lässt sich diese als CSV exportieren.[3] Die erste Zeile enthält dabei die Kategorien der Ausgabe. Die folgenden Zeilen enthalten jeweils die Daten eines Funktionsaufrufs und sind nach Aufrufszeitpunkt sortiert. Da bei den Daten eines jeden Funktionsaufrufs stets die Tiefe innerhalb der Aufrufstruktur enthalten ist und die Einträge chronologisch sortiert sind, kann aus dieser Liste ohne weiteres ein Aufrufgraph generiert werden.

Ein großer Vorteil dieses Datenformats ist die kompakte Darstellung. Jeder Eintrag nimmt hierbei lediglich eine einzige Zeile ein. Ferner wird in jeder Zeile nur das Minimum an Inhalt gespeichert, da lediglich die Werte der vorher definierten Kategorien und einige notwendige Trennzeichen gespeichert werden. Somit kann der benötigte Speicherplatz recht gering gehalten werden. Da neue Einträge lediglich ans Ende der Datei angehängt werden müssen, ist das Erweitern eines Aufrufgraphen durch neue Einträge leicht umsetzbar. Es können allerdings keine Kategorien entfernt oder neue hinzugefügt werden, falls die CSV durch den Profiler von Visual Studio generiert wird.

3.2.3 DYN: Erste Version

Um die Größe der Datei möglichst gering zu halten, wurde das das DYN-Format entwickelt. Im Folgenden wird der Aufbau des Formats beschrieben.

Zuerst werden alle einzigartigen Attribute aller Knoten und Kanten aufgelistet. Nach dem Schlüsselwort ”attribute” folgt dabei in Anführungsstrichen zuerst der Name und daraufhin der Wert des Attributs. Anschließend werden alle einzigartigen Knoten des Aufrufgraphen jeweils mit dem Schlüsselwort ”node” aufgelistet. Jeder Knoten muss mindestens zwei Attribute besitzen:

- ”Linkage.Name” und
- ”Level”

Ohne diese beiden Attribute kann kein Aufrufgraph generiert werden. Generell werden Attribute nach dem Schlüsselwort ”attributes” anhand einer Folge von auf Attribute verweisende

[1] Hogeson, M. et al., *Call Tree view*, `https://docs.microsoft.com/en-us/visualstudio/profiling/call-tree-view?view=vs-2017`, Zugriff: 26.12.2019.

[2] Hogeson, M. et al., *Call Tree view - sampling data* , `https://docs.microsoft.com/en-us/visualstudio/profiling/call-tree-view-sampling-data?view=vs-2017`, Zugriff: 19.01.2020

[3] Hogeson, M. et al., *Save and export performance tools data*, `https://docs.microsoft.com/en-us/visualstudio/profiling/saving-and-exporting-performance-tools-data?view=vs-2017`, Zugriff: 19.01.2020

Indizes aufgelistet. Hierbei wird mit "0" begonnen. Nach den Knoten folgen nun alle einzigartigen Kanten des Aufrufgraphen. Diese verbinden durch auf Knoten verweisende Indizes zwei Knoten miteinander. Attribute können hierbei wie auch bei Knoten mit dem Schlüsselwort "attributes" und beliebig vielen auf Attribute verweisende Indizes angegeben werden. Sind keine Attribute gegeben, kann das Schlüsselwort "attributes" weggelassen werden. Schlussendlich werden alle Funktionsaufrufe chronologisch aufgelistet. Dabei folgt auf das Schlüsselword "nodes" eine Liste von auf Kanten verweisende Indizes.

Listing 3.2: Beispiel eines Aufrufgraphen im ersten DYN-Format

```
1  attribute "Linkage.Name" "R example.cpp:main(i,c**)_c->i"
2  attribute "Linkage.Name" "R example.cpp:Foo:foo()->i"
3  attribute "Linkage.Name" "R example.cpp:Bar:bar()->i"
4  attribute "Level" "0"
5  attribute "Level" "1"
6  attribute "Level" "2"
7
8  node attributes 0 3
9  node attributes 1 5
10 node attributes 2 4
11
12 edge 0 2
13 edge 2 1
14
15 edges 0 1 0 0
```

Das Format minimiert die Größe eines Aufrufgraphen zum einen dadurch, dass alle Attribute, Knoten und Kanten nur genau einmal aufgelistet werden. Mehrfach vorkommende Funktionsaufrufe werden lediglich durch eine einzige Zahl dargestellt. Weitergehend wird die Größe eines Aufrufgraphen minimiert, falls folgende Kriterien erfüllt sind:

1. Der Aufrufgraph enthält viele Funktionsaufrufe bzw. es folgen viele Indizes auf das Schlüsselwort "nodes"
2. Der Aufrufgraph enthält wenig einzigartige Attribute, Knoten und Kanten

Sind in einem Aufrufgraphen viele Kanten, aber wenig einzigartige Attribute, Knoten und Kanten vorhanden, so enthält die Datei lediglich eine lange Liste von Indizes nach dem "nodes"-Schlüsselwort. Sind allerdings viele einzigartige Attribute, Konten und Kanten enthalten, so kann eine Datei in dem Format extrem stark anwachsen. Das Erstellen einer DYN-Datei sollte zusätzlich nicht zur Analysezeit passieren, da die Performanz der zu analysierenden Software je nach Implementierung mit der Zeit abnehmen kann. Da jedes Attribut, jeder Knoten und jede Kante eindeutig ist, könnte bei der Erstellung eine Menge für jeden der Elementtypen angelegt werden. Über die Zeit könnte dabei die Größe der Mengen stark ansteigen, was das Durchsuchen und Einfügen neuer Elemente verlangsamen würde. Selbst durch Verwendung von Hashtabellen oder anderen Konstrukten kann der Effekt nicht gänzlich eliminiert werden. Umgangen werden kann dieses Problem, indem zuerst alle Funktionsaufrufe aufgelistet und nach der Analyse in das DYN-Format konvertiert werden.

3.2.4 DYN: Zweite Version

Um Dateien im gewählten Format direkt zur Analysezeit generieren zu können, wurde die zweite Version von DYN entwickelt. Dieses ähnelt bis auf minimale Änderungen dem CSV-Format, das aus der Aufrufstrukturansicht von Visual Studio generiert werden kann. Die erste Zeile enthält auch hier die Kategorien der Elemente eines Funktionsaufrufs. Nach den Kategorien folgen alle Funktionsaufrufe in chronologischer Reihenfolge.

Das zweite DYN-Format teilt aufgrund der Ähnlichkeit zu CSV alle Vor- und Nachteile des Formats. Außerdem kann dieses Format ohne große Probleme um beliebige Kategorien erweitert werden. Ein möglicher Nachteil ist, dass es sich hierbei um kein bekanntes und somit kein weit verbreitetes Format handelt. Da im Kontext dieser Arbeit allerdings keine zusätzlichen Bibliotheken verwenden werden, stellt dies in diesem Zusammenhang kein schwerwiegendes Problem dar.

Listing 3.3: Beispiel eines Aufrufgraphen im zweiten DYN-Format

```
1 "Level" "Linkage.Name"
2 "0" "R example.cpp:main(i,c**)_c->i"
3 "1" "R example.cpp:Foo:foo()->i"
4 "2" "R example.cpp:Bar:bar()->i"
5 "1" "R example.cpp:Foo:foo()->i"
6 "1" "R example.cpp:Foo:foo()->i"
```

3.2.5 Fazit

Es wurden für alle beschrieben Formate mögliche Vor- und Nachteile genannt. Für die dynamische Analyse eignen sich GXL und die erste Version des DYN-Formats weniger gut. GXL nimmt zu viel Speicherplatz ein und das erste DYN-Format lässt sich nicht wie gewünscht zur Laufzeit generieren. Die exportierte CSV der Aufrufstrukturansicht von Visual Studio und die zweite Variante des DYN-Formats eignen dahingegen besonders für die Verarbeitung von Aufrufdaten zur Laufzeit. Da der Visual Studio Profiler die Kategorien einschränkt und das zweite DYN-Format minimal leichter zu verarbeiten ist als das CSV-Format, wird im Folgenden die zweite Variante des DYN-Formats verwendet, um dynamische Aufrufgraphen zu speichern.

3.3 Datenakkumulation

Im Folgenden wird auf verschiedene Methoden der dynamischen Softwareanalyse eingegangen, um Laufzeitdaten zu akkumulieren. Es wird hierbei primär auf dynamische Analysen von Java und C++ eingegangen.

3.3.1 Java Instrumentierung

Bereits kompilierte Java-Klassen können mithilfe der Java Instrumentierungs API nachträglich instrumentiert werden. Hierfür muss zunächst ein Java Agent definiert werden. Ein solcher Agent ist eine spezielle JAR-Datei, mithilfe dessen Bytecode in bereits kompilierten Java-Klassen einfügen werden kann. Dies kann statisch oder dynamisch geschehen. Beim statischen Laden eines Java Agenten wird der Bytecode direkt beim Start der Applikation modifiziert.[4] Im Folgenden wird lediglich auf die statische Variante eingehen, da diese für den Kontext dieser Arbeit zuverlässig direkt von Beginn des Java-Programms dynamische Analysedaten akkumulieren kann. In Abschnitt 4.1.1 wird genauer auf eine konkrete Implementierung eingegangen.

[4] *Java. Interface Instrumentation.* `https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html`, Zugriff: 27.12.2019.

3.3.2 Visual Studio

Um eine dynamische Analyse von C++-Programmen durchzuführen, kann sich Profiler von Visual Studio zunutze gemacht werden. Seit Visual Studio 2015 lassen sich die Ergebnisse des Profilers wie in Abschnitt 3.2.2 beschrieben exportieren. Visual Studio instrumentiert hierbei nicht den Code, sondern prüft mehrfach pro Sekunde, welche Funktion aktuell läuft. Diese Variante kann lediglich in einer unoptimierten Umgebung sinnvolle Ergebnisse liefern, da vermehrte Compileroptimierungen den Quellcode zu stark abändern. Außerdem sind die exportierten Kategorien invariabel. In Kapitel 4.1.2 wird auf weitere Probleme und mögliche Lösungen dieser Variante eingehen.

3.3.3 Manuelle Instrumentierung in C++

C++ Programme können manuell instrumentiert werden. Hierfür könnte ein Makro erstellt werden, welcher zu Beginn jeder Funktion eingefügt wird. Der Makro erstellt automatisch ein Objekt, welches bei der Destruktion, also nach Beendigung des Funktionsaufrufs eine Aktion ausführen kann. Zusätzlich wäre es möglich, den Makro vollautomatisch in jeder Funktion des zu analysierenden Softwaresystems einzufügen. In dieser Arbeit wird der Einfachheit halber C++-Code allerdings manuell instrumentiert. Auf die gewählte Implementierung wird in Abschnitt 4.1.3 eingegangen.

3.4 Visualisierung

Damit der Benutzer sofort erkennen kann, welche Funktionen aktuell miteinander interagieren, sollen die entsprechenden Gebäude der Software-Stadt deutlich sichtbar hervorgehoben und visuell miteinander verbunden werden. Dies soll mithilfe der in Abschnitt 2.2.3 beschriebenen Gestaltgesetze realisiert werden. Im Folgenden wird in Abschnitt 3.4.1 zuerst auf die Visualisierung der Knoten und in Abschnitt 3.4.2 daraufhin auf die Visualisierung der Kanten eingegangen.

3.4.1 Knoten

Zuerst sollen alle Knoten, also alle Gebäude, die derzeit in einen Funktionsaufruf verwickelt sind, hervorgehoben werden. Dies soll zum einen durch Formveränderung geschehen. Die Größe der Gebäude soll sich in einer Schleife sinus- bzw. kosinusförmig "pulsierend" verändern.

Zusätzlich soll die Farbe der Gebäude synchron mit deren Größe anhand eines Farbverlaufs variiert werden. Stéfan van der Walt und Nathaniel Smith veröffentlichten 2018 das Viridis Paket als eine matplotlib-Bibliothek für Python. In dieser sind fünf Farbskalen (siehe Abbildung 3.1) enthalten, welche sich für den Farbverlauf aus mehreren Gründen besonders gut eignen. Zum einen, so Rudis et al., spannen die Farben der Paletten so weit wie möglich, damit Unterschiede leicht zu erkennen sind. Zusätzlich werden die Paletten als uniform wahrgenommen. Das bedeutet, dass nahe beieinander liegende Farben ähnlich und Farben, die weit voneinander entfernt sind unterschiedlich wirken. Dieser Effekt ist bei gleichem Abstand auf der gesamten Palette konsistent. Zu guter Letzt seien die Farbpaletten bei vielen der am weitesten verbreiteten Arten der Farbblindheit besonders robust.[5]

[5] Rudis, B. et al., *The viridis color palettes*, `https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html`, 2018, Zugriff: 28.12.2019.

Es wird von den fünf Farbskalen im Folgenden die Viridis Farbpalette verwendet. Dies hat den einfachen Grund, dass diese bereits bei anderen Elementen der vorhandenen Software verwendet wird und somit das Endergebnis einheitlicher wirken lässt. Dabei wird die am hellsten wahrgenommene Farbe der Farbpalette zu dem Zeitpunkt verwendet, an dem die Blöcke ihre maximale Größe erreicht haben. Dies soll das natürliche Verhalten von sich ausdehnenden Objekten widergespiegelt werden, welche mit der Ausdehnung oftmals heller werden.

Da sich die aktuell aktiven Gebäude synchron bewegen und farblich verändern, greift an dieser Stelle das Gesetz der guten Fortsetzung. Alle aktiven Gebäude wirken also als zusammengehörig. Durch die identische Einfärbung der Gebäude wird zusätzlich das Gesetz der Ähnlichkeit beachtet, was den Effekt der Zusammengehörigkeit weiter verstärkt.

Abschnitt 4.3.2.1 geht auf eine mögliche Implementierung in der Unity Engine ein.

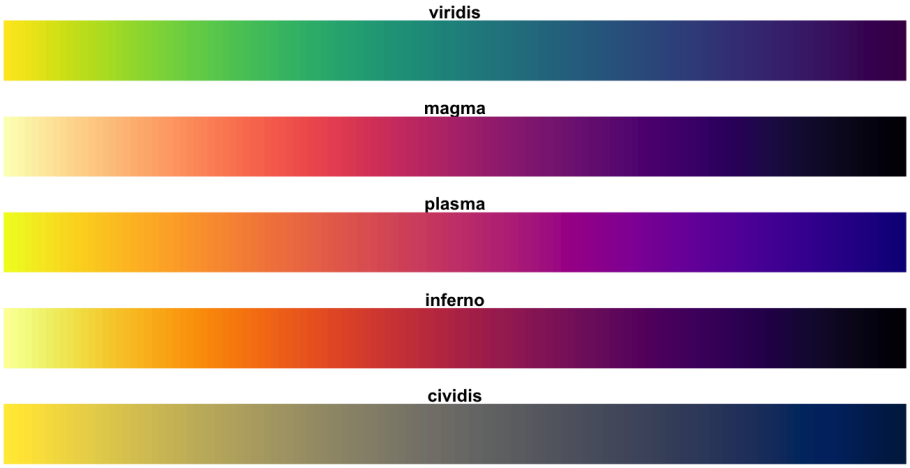


Abbildung 3.1: Die Farbpaletten des Viridis Pakets von R.[6]

3.4.2 Kanten

Um erkennen zu können, zwischen welchen Gebäuden aktuell Funktionsaufrufe laufen, müssen die jeweiligen Gebäude auf irgendeine Art miteinander verbunden werden. Hierbei sind zwei Dinge zu beachten. Zum einen muss klar und deutlich zu erkennen sein, welche Gebäude miteinander verbunden sind. Zum anderen muss aber auch die Richtung des Aufrufs schnell ersichtlich werden. Im Folgenden wird auf einige potentielle Methoden der Visualisierung eingegangen.

3.4.2.1 Partikelsysteme

Reeves beschreibt 1983 ein Partikelsystem als eine Menge von Partikeln, welche gemeinsam ein Objekt bilden.[7] Dieses habe, so Reeves, eine irreguläre und komplexe Oberfläche.[8] In das System werden nach und nach Partikel generiert, durch das System bewegt und verändert und zu guter Letzt aus dem System entfernt. Hierbei werden immer die folgenden Schritte durchgeführt:

[6] ebd.
[7] William T. Reeves, *Particle Systemsm: Technique for Modeling a Class of Fuzzy Objects*, ACM Transactions on Graphics. Vol. 2, 1983, S. 92.
[8] ebd., 91.

1. Neue Partikel werde in das System generiert.
2. Jedem Partikel werden individuelle Attribute zugewiesen.
3. Alle Partikel, welche länger als ihre definierte Lebensdauer gelebt haben, werden gelöscht.
4. Alle übrigen Partikel werden anhand ihrer dynamischen Attribute bewegt und transformiert.
5. Ein Bild der lebendigen Partikel wird gerendert.[9]

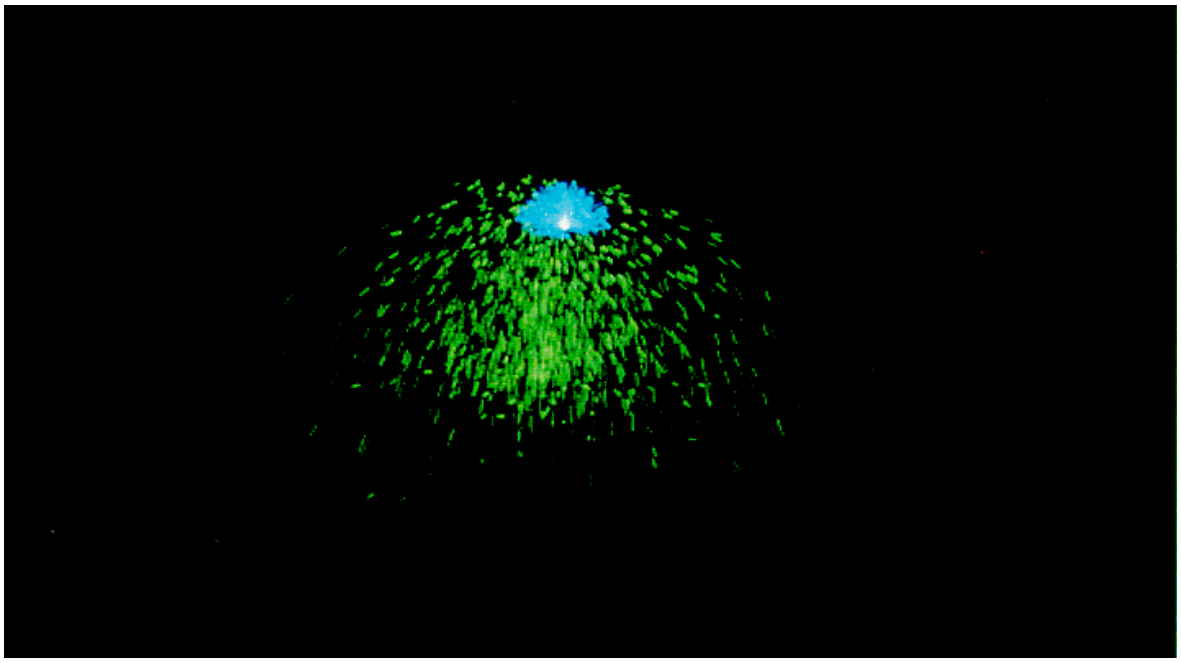


Abbildung 3.2: Grünes und blaues Feuerwerk als Partikelsystem.[10]

Auch Funktionsaufrufe können durch Verwendung eines Partikelsystems visualisiert werden. Viele Game Engines wie beispielsweise die Unreal Engine 4 oder auch die Unity Engine unterstützen Partikelsysteme und bieten mächtige Tools für die Generierung solcher Systeme an. Abbildung 3.3 zeigt den Editor für Partikelsysteme in der Unreal Engine 4. Das hier dargestellt Partikelsystem hat einen Start- und Endpunkt, die auf die Quelle bzw. das Ziel eines Funktionsaufrufs gesetzt werden können. So ließen sich Gebäude visuell miteinander verbinden. Die Richtung kann ebenfalls durch Partikelsysteme dargestellt werden. Hierbei können Partikel generiert werden, welche sich von der Quelle ausgehend in Richtung des Ziels des Funktionsaufrufs bewegen.

Ein signifikanter Nachteil von Partikelsystemen im Kontext dieser Arbeit ist die generelle Idee, dass Partikel eine festgelegte Lebensspanne haben. Dies wird problematisch, sobald einige der Abstände von Start zu Ziel zwischen Funktionsaufrufen inkonsistent sind. Dies hätte zufolge, dass für jeden Funktionsaufruf ein separates Partikelsystem definiert werden müsste. Zusätzlich werden durch Partikelsysteme oftmals irreguläre Objekte dargestellt. Möchte man allerdings eher simple Visualisierungen wählen, sind Partikelsysteme oftmals ungeeignet.

[9] ebd., 92ff.
[10] ebd., 104.

Trotz dieser Einschränkungen wurde im Kontext dieser Arbeit mit Partikelsystemen für die Visualisierung von dynamischen Aufrufgraphen experimentiert. Abschnitt 4.3.1.1 geht daher auf eine mögliche Implementierung in der Unreal Engine 4 an.

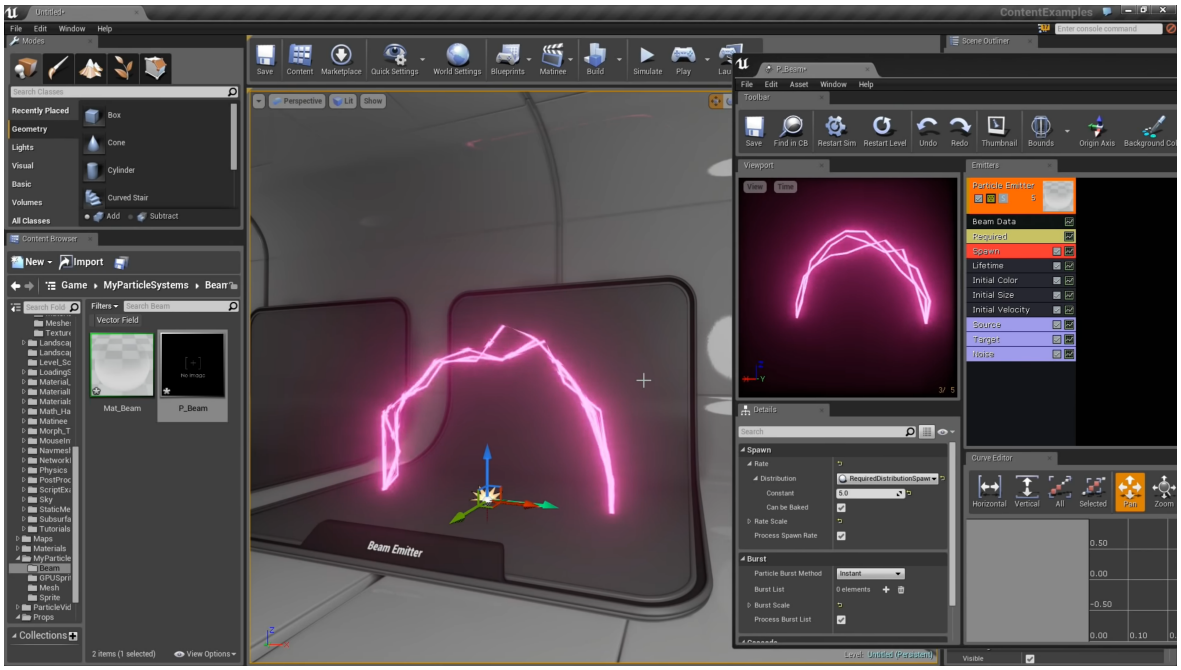


Abbildung 3.3: Ausschnitt aus einer auf dem YouTube-Kanal der Unreal Engine veröffentlichten Anleitung zu Partikelsystemen.[11]

3.4.2.2 Kugeln

Die Visualisierung von Kanten kann auch ohne die Verwendung von Partikelsystemen vonstatten gehen. Ein potentieller Ansatz wäre, Kugeln von den jeweiligen Quellengebäude zum Zielgebäude "fliegen" zu lassen. Abbildung 3.4 zeigt, wie eine solche Visualisierung aussehen könnte.

Hier gilt das Gesetz der Nähe, da die Kugeln, wie in der Abbildung zu erkennen ist, nah beieinander sind und somit als zusammengehörig wirken. Auch gilt durch die identische Form der Kugeln das Gesetz der Ähnlichkeit und durch das identische Bewegungsschema das Gesetz der guten Fortsetzung. Ebenso wie auch die Gebäude können die Kugeln mit einem Farbverlauf eingefärbt werden. Die Kugeln können ihre Farbe je nach Entfernung vom Quell- bzw. Zielgebäude verändern und somit einen weiteren Indikator für die Richtung der Funktion vermitteln. Um Einheitlichkeit zu bewahren kann auch hier die selbe Farbpalette verwendet werden, die auch für die Gebäude genutzt wurde.

Die Nachteile der Partikelsysteme greifen in dieser Form der Visualisierung nicht, weshalb dieses Schema für die finale Implementierung verwendet wird. Abschnitt 4.3.2.2 beschreibt die Implementierungsdetails unter Verwendung der Unity Engine.

[11] *Intro to Cascade: Creating a Beam Emitter | 07 | v4.2 Tutorial Series | Unreal Engine*, `https://www.youtube.com/watch?v=ywd3lFOuMV8`, ca. 11:50, Zugriff: 06.01.2020.

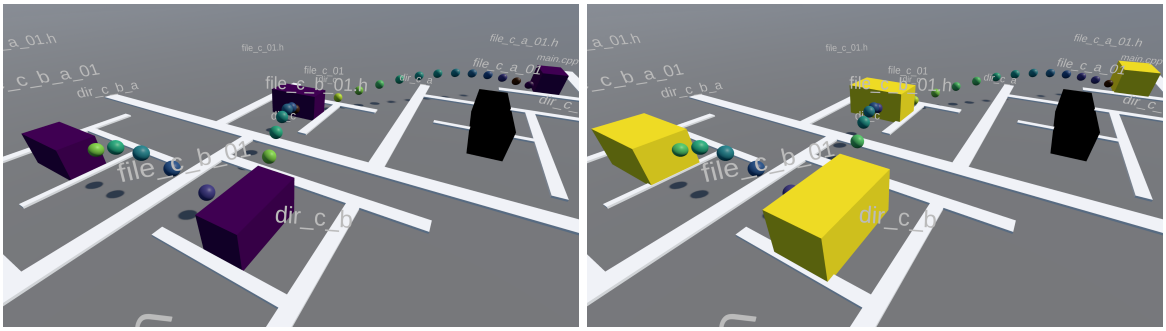


Abbildung 3.4: Ausschnitte aus dem Animations-Loop der aktiven Gebäude

3.5 Interaktion

3.5.1 Fortbewegung

Auf dem Desktop kann man sich durch das gedrückt halten der folgenden Tasten fortbewegen:

Taste	Aktion
W	Forwarts bewegen
A	Nach links bewegen
S	Rückwärts bewegen
D	Nach rechts bewegen
Leertaste	Nach oben bewegen
Linke Steuerung-Taste	Nach unten bewegen
Linke Shift-Taste	Beschleunigen

Tabelle 3.1: Tastenbelegung für die Fortbewegung im Desktop-Modus

In VR kann sich durch Halten des Triggers (siehe Abbildung 2.1) des rechten Controllers fortbewegt werden. Hierbei bewegt man sich immer genau in die Richtung, in die der Controller ausgerichtet ist. Zusätzlich lässt sich dabei die Geschwindigkeit durch die Druckstärke beeinflussen.

3.5.2 Simulation

Eine der Anforderungen an die Software ist die konsistente Bedienung zwischen Desktop und VR, um die Evaluation zu vereinfachen. Dies wird ermöglicht, indem die Simulation durch Tasten beeinflusst werden kann. Ein Menü wird im Kontext dieser Arbeit nicht entwickelt.

Am Desktop kann die Simulation durch das Drücken der Enter-Taste ein- oder ausgeschaltet werden. Ist die Simulation eingeschaltet, so kann durch das Drücken der Plus- und Minus-Taste im Programmverlauf vor- und zurück navigiert werden. In VR kann die Simulation durch das Drücken der Trigger-Taste vom linken Controller gestartet oder gestoppt werden. Auch hier kann im Programmverlauf erst vor- und zurückgegangen werden, sobald die Simulation gestartet ist. Um zum nächsten Funktionsaufruf zu springen, muss auf dem Trackpad des linken Controllers der rechte Bereich gedrückt werden. Um zu dem Vorherigen zu springen, muss der linke Bereich des linken Trackpads gedrückt werden.

KAPITEL 4

Implementierung

4.1 Datenakkumulation

4.1.1 Java Instrumentierung

Im Folgenden wird auf die Implementierung der Java Instrumentierung eingegangen. Hierbei erklärt Abschnitt 4.1.1.1 zunächst die generelle Funktionsweise des `Instrumentation`-Interfaces. Daraufhin beschreibt Abschnitt 4.1.1.2 die gewählte Implementierung für die dynamische Analyse.

4.1.1.1 Instrumentation-Interface

Wie bereits in Abschnitt 3.3.1 angedeutet, muss zunächst ein Java Agent[1] erstellt werden. Dieser muss, um statisch geladen werden zu können eine `public static void premain(String, Instrumentation)`-Methode definieren. Diese wird aufgerufen, sofern beim Starten einer JVM der entsprechende Java Agent angegeben wird.

Fürs Instrumentieren von Java-Klassen muss zusätzlich eine konkrete Implementierung des `ClassFileTransformer`[2]-Interfaces definiert werden. Diese muss stets die Methode `public byte[] transform(ClassLoader, String, Class<?>, ProtectionDomain, byte[])` definieren. In dieser kann wie in Abbildung 4.1 beispielhaft angedeutet der Quellcode modifiziert werden. Der modifizierte Quellcode wird daraufhin als `byte[]` zurückgegeben. Soll eine bestimmte Klasse unverändert bleiben, kann alternativ auch der bereits übergebene Buffer der unveränderten Klasse (siehe `classfileBuffer` in Abbildung 4.1)[3] [4] zurückgegeben werden.

Listing 4.1: Beispiel einer Implementierung des `ClassFileTransformer`-Interfaces

```
1  public class MyTransformer implements ClassFileTransformer {
2
3      @Override
4      public byte[] transform(
5          ClassLoader loader,
6          String className,
7          Class<?> classBeingRedefined,
8          ProtectionDomain protectionDomain,
9          byte[] classfileBuffer) {
10
11         try {
```

[1] *Interface Instrumentation*, `https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html`, Zugriff: 27.12.2019.

[2] *Interface ClassFileTransformer*, `https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/ClassFileTransformer.html`, Zugriff: 02.01.2020.

[3] ebd.

[4] *Class CtMethod*, `https://www.javassist.org/html/javassist/CtMethod.html`, Zugriff: 02.01.2020.

```
12              ClassPool classPool = ClassPool.getDefault();
13              CtClass ctClass = classPool.get(className);
14              CtMethod method = ctClass.getDeclaredMethod("foo");
15              method.insertBefore("System.out.println(\"Hello World!\n\")");
16              byte[] byteCode = ctClass.toBytecode();
17              ctClass.detach();
18              return byteCode();
19          } catch (Exception e) {
20              return classfileBuffer;
21          }
22
23      }
24
25  }
```

Anschließend muss ein Objekt für alle gewünschten `ClassFileTransformer`-Klassen instantiiert und mittels `addTransformer([...])` dem `Instrumentation`-Objekt übergeben werden. Daraufhin muss lediglich nur noch `retransformClasses(Class<?>...)` mit allen zu instrumentierenden Klassen auf dem `Instrumentation`-Objekt aufgerufen werden, um die Instrumentierung zu starten. Abbildung 4.2 zeigt wie eine solche `premain`-Methode aussehen kann.

Listing 4.2: Beispiel der `premain`-Methode eines Java Agenten.[5]

```
1   public static void premain(
2       String agentArgs,
3       Instrumentation inst) {
4
5       MyTransformer transformer = new MyTransformer();
6       inst.addTransformer(transformer, true);
7       Class c = [...];
8       inst.retransformClasses(c);
9
10  }
```

Ist der Agent nun erstellt, so lässt er sich wie in Abbildung 4.3 dargestellt statisch an die JVM anhängen.

Listing 4.3: Java Agent "agent.jar" statisch an Applikation "application.jar" anhängen.[6]

```
1  java -javaagent:agent.jar -jar application.jar
```

4.1.1.2 Implementierung der Java Instrumentierung

Für die Implementierung wurde zunächst eine einfache Stack-Klasse (siehe Abbildung 4.4) definiert. Diese enthält eine `push`- und eine `pop`-Methode sowie einen internen Zähler (`counter`).

Durch Aufruf der `push`-Methode wird zunächst der aktuelle Wert des internen Zählers und der übergebene Name der Funktion in eine Datei geschrieben. Hierbei wird das in Abschnitt 3.2.4 beschriebene Format verwendet. Der Zählerwert ist stets die aktuelle Tiefe des Aufrufstacks. Somit wird dieser nach dem Erstellen des neuen Eintrags um Eins inkrementiert. Der Aufruf der `pop`-Methode dekrementiert den Wert des internen Zählers wieder um Eins.

Listing 4.4: Vereinfachte Stack-Klasse

```
1  public class Stack {
2
3      private static int counter;
```

[5] *Interface Instrumentation*, `https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html`, Zugriff: 27.12.2019.

[6] *Package java.lang.instrument*, `https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html`, Zugriff: 15.01.2020.

```
4
5       public static void push(string linkageName) {
6           [...] // create entry in file
7           counter++;
8       }
9
10      public static void pop() {
11          counter--;
12      }
13
14  }
```

Um mithilfe dieser Klasse einen Aufrufgraphen generieren zu können, muss die `push`-Methode zu Beginn und die `pop`-Methode am Ende von jeder Funktion aufgerufen werden. Hierfür bietet die `CtMethod`[7]-Klasse folgende Funktionalität:

public void insertBefore(java.lang.String src) throws CannotCompileException[8]

Diese Methode fügt Bytecode (`src`) am Anfang der Methode ein.

public void insertAfter(java.lang.String src, boolean asFinally) throws CannotCompileException[9]

Diese Methode fügt Bytecode (`src`) am Ende des Methodenkörpers ein. Der Bytecode wird direkt vor jedem `return`-Statement eingesetzt. Wird eine `Exception` geworfen, so wird der Bytecode nur genau dann ausgeführt, wenn `true` bei `asFinally` übergeben wird. `asFinally` muss für die korrekte Generierung von dynamischen Aufrufgraphen also immer `true` sein, damit beim Verlassen der Methode aufgrund einer Exception der Zähler des Stacks wie erwartet dekrementiert wird.

Es muss nun nur noch der Name der aktuellen Methode bestimmt und der Code mittels der eben beschriebenen Methoden instrumentiert werden. Abbildung 4.5 zeigt, wie dies vonstatten gehen kann.

Listing 4.5: Aufrufen der `CtMethod`-Methoden

```
1  insertBefore("
2      String className = this.getClass().getName();\n
3      String methodName = new Object(){}.getClass().getEnclosingMethod().getName();\n
4      String linkageName = className + \".\" + methodName;\n
5      Stack.push(linkageName);
6  ");
7  insertAfter("
8      Stack.pop();
9  ");
```

Um nun einen funktionierenden Java Agenten zu erzeugen, fehlen noch einige Kleinigkeiten, die aufgrund der Einfachheit lediglich kurz erwähnt werden. Zum einen muss jede instrumentierte Klasse das `Stack`-Paket importieren.[10] Zusätzlich muss eine Manifest-Datei definiert werden, in der die Position der `premain`-Funktion festgelegt werden muss. Wird der Java

[7] *Class CtMethod*, `https://www.javassist.org/html/javassist/CtMethod.html`, Zugriff: 02.01.2020.

[8] *Class CtBehavior: insertBefore*, `https://www.javassist.org/html/javassist/CtBehavior.html#insertBefore(java.lang.String)`, Zugriff: 15.01.2020.

[9] *Class CtBehavior: insertAfter*, `https://www.javassist.org/html/javassist/CtBehavior.html#insertAfter(java.lang.String,boolean)`, Zugriff: 15.01.2020.

[10] *Class ClassPool: importPackage*, `http://www.javassist.org/html/javassist/ClassPool.html#importPackage(java.lang.String)`, Zugriff: 15.01.2020.

Agent nun gebaut und wie in Abbildung 4.3 gestartet, so wird der Code automatisch Instrumentiert und der Aufrufgraph in eine Ausgabedatei geschrieben.

4.1.2 Visual Studio Aufrufstruktur zu DYN Version 1

Bei der dynamischen Analyse von C++-Programmen wurde mit dem Visual Studio Profiler experimentiert. Wie bereits in Abschnitt 3.3.2 erklärt, lassen sich mithilfe des Visual Studio Profilers eine Aufrufstruktur als CSV exportieren. Um die Ergebnisse weiter bearbeiten zu können, wurde zunächst ein Programm geschrieben, welches eine solche CSV einlesen und aus dem darin enthaltenen dynamischen Aufrufgraphen eine baumartige Struktur aufbauen kann. Diese kann nun in ein beliebiges Format konvertiert werden. Da zu diesem Zeitpunkt noch mit unterschiedlichen Formaten experimentiert wurde, wurde der Baum zunächst in die erste Version vom DYN-Format konvertiert. Im Anhang ist ein Beispiel für vor (siehe Abbildung 1) und nach (siehe Abbildung 2) der Konvertierung enthalten.

Beim Experimentieren wurden allerdings einige Probleme dieser Methode festgestellt. Beim Analysieren einer kleinen Beispielsoftware fiel auf, dass die generierte CSV oftmals Samples anderer, parallel laufender Programme enthält. Das Ergebnis muss also gefiltert werden, bevor es für die Visualisierung verwendbar ist. Beim Filtern eines minimalen Beispiels fiel zusätzlich auf, dass die CSV keine Samples der eigentlich zu analysierenden Software enthielt. Ist die Laufzeit zu kurz, so schien der Profiler keine Samples der eigentlich relevanten Software zu speichern. Zum Testen wurde daraufhin die Laufzeit der Software mithilfe eines `for`-Loop künstlich ausgedehnt. Nun fiel auf, dass die Ergebnisse lückenhaft sind. Abbildung 4.6 zeigt das gefilterte Ergebnis des künstlich erweiterten Beispiels.

Listing 4.6: Beispiel eines Aufrufgraphen im CSV-Format

```
1  0,"example.exe",56,0,"100,00","0,00",""
2  2,"mainCRTStartup",52,0,"92,86","0,00","example.exe",
3  3,"__scrt_common_main",52,0,"92,86","0,00","example.exe",
4  4,"__scrt_common_main_seh",52,0,"92,86","0,00","example.exe",
5  5,"invoke_main",51,0,"91,07","0,00","example.exe",
6  6,"main",51,5,"91,07","8,93","example.exe",
7  7,"dir_a::file_a_01::function_a_01",39,10,"69,64","17,86","example.exe",
8  8,"dir_a_a::file_a_a_01::function_a_a_01",19,7,"33,93","12,50","example.exe",
9  9,"dir_a_b::file_a_b_01::function_a_b_01",12,9,"21,43","16,07","example.exe",
10 10,"dir_a_b::file_a_b_02::function_a_b_02",2,2,"3,57","3,57","example.exe",
11 10,"@ILT+410",1,1,"1,79","1,79","example.exe",
12 8,"dir_a_b::file_a_b_01::function_a_b_01",7,6,"12,50","10,71","example.exe",
13 9,"dir_a_b::file_a_b_02::function_a_b_02",1,1,"1,79","1,79","example.exe",
14 8,"@ILT+410",3,3,"5,36","5,36","example.exe",
15 7,"dir_b::file_b_01::function_b_01",6,5,"10,71","8,93","example.exe",
16 8,"dir_b::file_b_02::function_b_02",1,1,"1,79","1,79","example.exe",
17 7,"dir_c::file_c_01::function_c_01",1,1,"1,79","1,79","example.exe",
```

Wäre die CSV komplett, so müssten mindestens 10.000.001 Zeilen, also für jeden der Funktionsaufrufe genau eine, enthalten sein. Es wurden allerdings nur 17 Funktionsaufrufe gespeichert. Die ungefilterte CSV für dieses Beispiel enthielt genau 1.879 Funktionsaufrufe. Es werden also noch zusätzlich unnötig viele Daten angehäuft, die für die eigentliche Analyse irrelevant sind. Da diese Variante der Datenakkumulation viele Anforderungen nicht ausreichend erfüllt, wird sie im Folgenden nicht weiter verwendet.

4.1.3 Manuelle Instrumentierung

Wie bereits in Abschnitt 3.3.3 erwähnt, kann zur manuellen Instrumentierung ein Makro erstellt werden. Die aufs wesentliche reduzierte Implementierung dieser Variante ist in Ab-

bildung 4.7 dargestellt. Die Abbildung ist stark vereinfacht und enthält Elemente wie die Modifizierung des Ergebnisses von `__FUNCSIG__` nicht.

Listing 4.7: Vereinfachte Instrumentor-Header-Datei

```
namespace inst {

    struct FunctionData
    {
        string name_;
        [...]
    };

    struct Function
    {
        Function(const FunctionData& function_data);
        ~Function();
        [...]
    };

    struct CallStack
    {
        CallStack();
        void push(const FunctionData& function_data);
        FunctionData pop();
        [...]
    };

    #define FUNCTION() ::inst::Function function##__LINE__({ __FUNCSIG__, [...] });

}
```

Listing 4.8: Beispiel für eine instrumentierte Funktion

```
// include instrumentor.h

public void foo()
{
    FUNCTION()
    [...]
}
```

Jede Funktion, welche von der Analyse betrachtet werden soll, muss als erstes Element den `FUNCTION()`-Makro enthalten (siehe Abbildung 4.8). Durch das Einfügen des Makros wird automatisch ein `Function`-Objekt erzeugt. Dabei wird zuerst ein Eintrag in die Ausgabedatei geschrieben und anschließend die aktuellen Funktionsdaten dem Stack hinzugefügt. Der Eintrag enthält hierbei immer den vollständigen Namen der Funktion und die Anzahl der Elemente im Stack. Die Anzahl entspricht dabei immer der Tiefe im Aufrufgraphen. Ist das Ende einer instrumentierten Funktion erreicht, so wird das Funktionsobjekt zerstört. Die Funktionsdaten werden somit wieder vom Aufrufstack entfernt.

Führt man eine instrumentierte Software aus, so wird automatisch eine Ausgabedatei erstellt und jeder instrumentierte Funktionsaufruf dokumentiert. Abbildung 4.9 zeigt die Ausgabedatei nach dem Ausführen einer instrumentierten Beispiel-Software.

Listing 4.9: Beispiel eines Aufrufgraphen im zweiten DYN-Format

```
"Level" "Linkage.Name"
"0" "int main(int,char **)"
"1" "void dir_a::file_a_01::function_a_01(void)"
"2" "void dir_a_a::file_a_a_01::function_a_a_01(void)"
"3" "void dir_a_b::file_a_b_01::function_a_b_01(void)"
"4" "void dir_a_b::file_a_b_02::function_a_b_02(void)"
"3" "void dir_a_b::file_a_b_02::function_a_b_02(void)"
"2" "void dir_a_b::file_a_b_01::function_a_b_01(void)"
"3" "void dir_a_b::file_a_b_02::function_a_b_02(void)"
```

```
10 "2" "void dir_a_b::file_a_b_02::function_a_b_02(void)"
11 "1" "void dir_b::file_b_01::function_b_01(void)"
12 "2" "void dir_b::file_b_02::function_b_02(void)"
13 "1" "void dir_c::file_c_01::function_c_01(void)"
```

4.2 Einlesen von Aufrufgraphen im zweiten DYN-Format

Um einen Funktionsaufruf visuell Darstellen zu können, muss der dynamische Aufrufgraph zunächst einmal eingelesen werden. Im Kontext dieser Arbeit wurde sowohl mit der Unreal Engine 4, als auch mit der der Unity Engine experimentiert, weshalb im Folgenden auf die Implementierungen in beiden Game Engines eingegangen wird. Zuerst beschreibt Abschnitt 4.2.1 die Implementierung in der UE4. Anschließend geht Abschnitt 4.2.2 auf die Implementierung in Unity ein.

4.2.1 Unreal Engine 4

In den Abschnitten 4.2.1.1 bis 4.2.1.4 wird das grundsätzliche Verfahren beim Erstellen von eigenen Formaten in UE4 erklärt. Daraufhin beschreibt Abschnitt 4.2.1.5 die auf das Verfahren aufbauende Implementierung.

4.2.1.1 UFactory

UE4 ermöglicht es, Dateien mit beliebigen Endungen einzulesen. Um die UE4 um ein nicht nativ unterstütztes Format erweitern zu können, muss zunächst die Klasse `UFactory` überschrieben werden. Diese enthält Funktionen mit dem Präfix "FactoryCreate", welche überschrieben werden können. Abbildungen 4.10 und 4.11 zeigen vereinfacht, wie eine solche Klasse aussehen kann.

Im Konstruktor werden die von der Factory unterstützten Formate angegeben. Hier sind auch weitere Anpassungen möglich. Wird nun eine Datei mit diesem Format in den Asset-Bereich der Engine gezogen, so wird automatisch die in den Abbildungen definierte Funktion aufgerufen. Hier kann nun ein Objekt generiert und zurückgegeben werden.

Listing 4.10: Beispielhafte Implementierung von `UFactory` (h)

```
1  UCLASS()
2  class UMyFactory : public UFactory
3  {
4      GENERATED_UCLASS_BODY()
5
6      UMyFactory(const class FObjectInitializer& ObjectInitializer);
7      virtual UObject* FactoryCreateText(
8          UClass* Class,
9          UObject* Parent,
10         FName Name,
11         EObjectFlags Flags,
12         UObject* Context,
13         const TCHAR* Type,
14         const TCHAR*& Buffer,
15         const TCHAR* BufferEnd,
16         FFeedbackContext* Warn) override;
17 };
```

Listing 4.11: Beispielhafte Implementierung von `UFactory` (cpp)

```
1  UMyFactory::UMyFactory(const FObjectInitializer& ObjectInitializer)
```

```
2       : Super(ObjectInitializer)
3   {
4       SupportedClass = UMyObject::StaticClass();
5       Formats.Add([...]);
6   }
7
8   UObject* UMyFactory::FactoryCreateText(
9       UClass* Class,
10      UObject* Parent,
11      FName Name,
12      EObjectFlags Flags,
13      UObject* Context,
14      const TCHAR* Type,
15      const TCHAR*& Buffer,
16      const TCHAR* BufferEnd,
17      FFeedbackContext* Warn)
18  {
19      UMyObject* object = NewObject<UMyObject>([...]);
20      [...]
21      return object;
22  }
```

4.2.1.2 UActorFactory

Aktuell ist es lediglich möglich, Dateien als Asset in die Engine zu ziehen. Die `UActorFactory` ermöglicht es nun, aus einem solchen Asset ein tatsächliches Objekt im Spiel zu erzeugen. Abbildungen 4.12 und 4.13 zeigen simplifiziert, wie eine solche Klasse definiert wird.

Auch hier können im Konstruktor weitere Anpassungen vorgenommen werden. In der Abbildung wird beispielhaft der Anzeigename und die Klasse eines neuen `AActor`-Objektes definiert. Die Funktion `PostSpawnActor` wird immer genau dann aufgerufen, wenn ein `AActor`, also ein tatsächliches Objekt im Spiel, aus einem eigenen Asset erstellt wurde. Der `AActor` kann hier nach Bedarf initialisiert werden.

Listing 4.12: Beispielhafte Implementierung von `UActorFactory` (h)

```
1   UCLASS()
2   class UMyActorFactory : public UActorFactory
3   {
4       GENERATED_UCLASS_BODY()
5
6       UMyActorFactory(const class FObjectInitializer& ObjectInitializer);
7
8       virtual void PostSpawnActor(
9           UObject* Asset,
10          AActor* NewActor) override;
11
12      virtual bool CanCreateActorFrom(
13          const FAssetData& AssetData,
14          FText& OutErrorMsg) override;
15  };
```

Listing 4.13: Beispielhafte Implementierung von `UActorFactory` (cpp)

```
1   UMyActorFactory::UMyActorFactory(const FObjectInitializer& ObjectInitializer)
2       : Super(ObjectInitializer)
3   {
4       DisplayName = [...]
5       NewActorClass = AMyActor::StaticClass();
6   }
7
8   void UMyActorFactory::PostSpawnActor(
9       UObject* Asset,
10      AActor* NewActor)
11  {
```

```
12      Super::PostSpawnActor(Asset, NewActor);
13      if (UMyObject* myObject = Cast<UMyObject>(Asset))
14      {
15          [...]
16      }
17  }
18
19  bool UMyActorFactory::CanCreateActorFrom(
20      const FAssetData& AssetData,
21      FText& OutErrorMsg)
22  {
23      return (AssetData.IsValid() &&
24              AssetData.GetClass()->IsChildOf(UMyObject::StaticClass()));
25  }
```

4.2.1.3 IDetailCustomization

Zu guter Letzt können im Editorfenster weitere Funktionalitäten bereitgestellt werden. Hierfür kann die Klasse `IDetailCustomization`, wie in Abbildung 4.14 beispielhaft gezeigt wird, definiert werden. Mithilfe dieser können im Editor nun einfache Informationen, aber auch von Knöpfe oder andere UI-Elemente dargestellt werden.

Listing 4.14: Beispielhafte Implementierung von `IDetailCustomization` (h)

```
1  class FMyDetailCustomization : public IDetailCustomization
2  {
3  public:
4      virtual void CustomizeDetails(IDetailLayoutBuilder& DetailBuilder) override;
5  };
```

4.2.1.4 Weitere Anpassungen

Es sind noch weitere Anpassungen wie zum Beispiel das Erstellen eines Symbols für das benutzerdefinierte Format möglich.[11] Im Weiteren wird allerdings nicht genauer darauf eingegangen, da die beschriebene Funktionalität für den Zweck dieser Arbeit ausreichend ist.

4.2.1.5 Implementierung

Bei der gewählten Implementierung konnte grundlegend dem im vorherigen Kapiteln beschriebenen Vorgang gefolgt werden. Die Architektur ähnelt allerdings an einigen Stellen stark der gewählten Architektur in der Unity Engine, weshalb im Folgenden oftmals auf spätere Abschitte verwiesen wird.

Zuerst wurde eine DYNFactory (siehe Abbildung 4.15) erstellt, um aus Dateien im DYN-Format `UObject`-Objekte (in diesem Fall `UDYNObject`) erstellen zu können. Um ggf. entstehende Probleme mit der Serialisierungstiefe zu vermeiden, wurde an dieser Stelle wie in Abschnitt 4.2.2.1 vorgegangen. Der eigentliche dynamische Aufrufgraph wird erst zu Beginn des Spiels generiert.

Listing 4.15: Ausschnitt aus `DYNFactory`.cpp

```
1  [...]
2  UObject* UDYNFactory::FactoryCreateText(
3      [...]
4      const TCHAR*& Buffer,
```

[11] *FSlateStyleSet*, `https://docs.unrealengine.com/en-US/API/Runtime/SlateCore/Styling/FSlateStyleSet/index.html`, Zugriff: 15.01.2020

```
5       const TCHAR* BufferEnd,
6       [...])
7   {
8       UDYNObject* object = NewObject<UDYNObject>([...]);
9       FString FileContent(BufferEnd - Buffer, Buffer);
10      object->InitSerializable(FileContent);
11      return object;
12  }
13  [...]
```

In der `UDYNActorFactory` wird bei einem Aufruf der `FactoryCreateText`-Funktion lediglich das `UDYNObject` als Datenfeld im `UDYNActor` gespeichert. Wie bereits erwähnt wird der dynamische Aufrufgraph aufgebaut, sobald das Spiel gestartet wird. Hier wird generell wie in Abschnitt 4.2.2.2 vorgegangen.

4.2.2 Unity Engine

Das Einlesen von Dateien wurde in der Unity Engine generell simpler implementiert. Es wird im Gegensatz zur Unreal Engine 4 kein neues Format eingerichtet, sondern lediglich der Inhalt der Datei eingelesen. Die Abschnitte 4.2.2.1 und 4.2.2.2 beschreiben die weitergehende Implementierung zum Parsen des Dateiinhalts genauer.

4.2.2.1 Vor Beginn der Laufzeit

Abbildung 4.1 zeigt die generelle Klassenstruktur vom DYN-Parser als UML-Diagramm. Ist im `DYNParser` der Dateiname (`filename`) gesetzt, so kann mit Aufruf der `Load()`-Methode die Datei im DYN-Format eingelesen werden. Hierbei werden die Zeilen der Datei zuerst mithilfe von `Tokenize(string)` in Tokens getrennt. Daraufhin wird, je nachdem ob es sich bei der Zeile um Kategorien oder Funktionsaufrufe handelt, `Categories(string[])` bzw. `FunctionCall(string[])` mit den entsprechenden Tokens als Parameter aufgerufen. Diese werden nun vom `CallTreeReader` überschrieben und generieren sukzessive den `CallTree`.

Da die Serialisierungstiefe in Unity eingeschränkt ist, können sowohl `predecessor` als auch `successor` nicht beim Generieren der Stadt, also vor der tatsächlichen Laufzeit, initialisiert werden. Dies würde die Tiefe der Serialisierung überschreiten. Daher werden im Aufrufgraphen beim Aufruf von `Load()` lediglich die Kategorien (`categories`) vollständig erstellt. Bei den Funktionsaufrufen (`functionCalls`) werden zu diesem Zeitpunkt nur die Kategorien und Attribute gespeichert.

Zusätzlich zum Aufrufgraphen wird ein `GameObject` mit einem `Runtime`-Skript erstellt. Dieses ist relevant für jegliche Laufzeitaktivität.

4.2.2.2 Zu Beginn der Laufzeit

Das `Runtime`-Skript (Abbildung 4.2) ist zwar hauptsächlich für die Visualisierung zur Laufzeit verantwortlich, generiert aber auch zu Beginn der Laufzeit Elemente des Aufrufgraphen, die aufgrund von Limitierungen der Serialisierung nicht vor Beginn der Laufzeit generiert werden konnten.

Durch den Aufruf von `GenerateTree()` wird im Aufrufgraphen zunächst durch alle Funktionsaufrufe iteriert und die jeweiligen Vorgänger und Nachfolger der Knoten gesetzt. Diese sind bei der Visualisierung relevant, da nicht ausschließlich der aktuelle Aufruf, sondern stets der gesamte Aufrufstapel angezeigt werden soll.

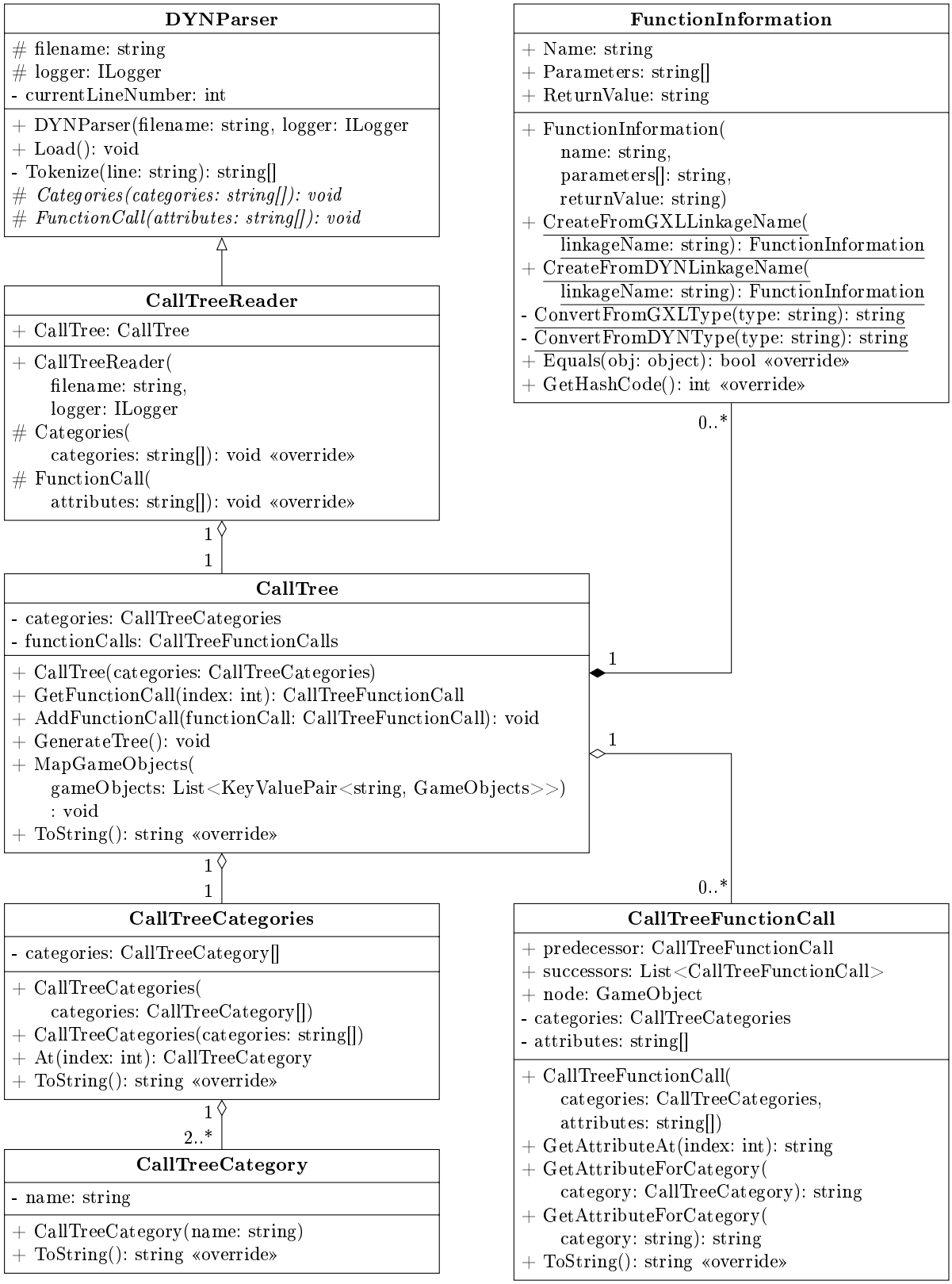


Abbildung 4.1: UML-Klassendiagramm des DYN-Parsers

Mit einem Aufruf der Methode `MapGameObjects(List<KeyValuePair<string, GameObject>> gameObjects)` werden anschließend die Gebäude auf die entsprechenden Funktionsaufrufe abgebildet. Der Schlüssel von `gameObjects` ist hierbei der vollständige Name der vom Gebäude visualisierten Funktion. Der Wert ist das entsprechende `GameObject`. Es wird hierbei zunächst für jedes Element von `gameObjects`, aber auch für jeden der im Aufrufgraphen gespeicherten Funktionsaufrufe ein `FunctionInformation`-Objekt generiert. Dieses enthält alle relevanten Eigenschaften eines Funktionsaufrufs. Durch Vergleichen dieser Objekte können so die Gebäude auf die Funktionsaufrufe abgebildet werden.

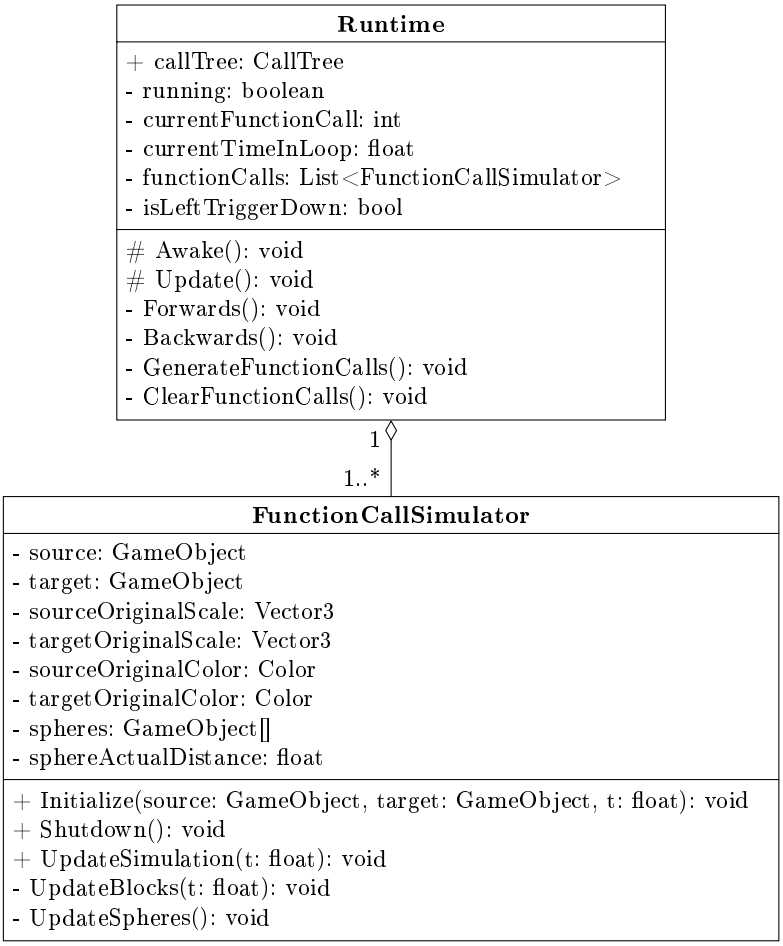


Abbildung 4.2: UML-Klassendiagramm des Laufzeitanteils

4.3 Visualisierung

Im Kontext dieser Arbeit wurde sowohl in der Unreal Engine 4, als auch in der Unity Engine mit verschiedenen Visualisierungen experimentiert. Da sich diese Visualisierungen in beiden Game Engines unterscheiden, werden die Implementierungen separat beschrieben. Abschnitt 4.3.1 geht zuerst auf die Implementierung in UE4 und Abschnitt 4.3.2 anschließend auf die Implementierung in Unity ein.

4.3.1 Unreal Engine 4

Da die finale Software auf der Unity Engine aufbaut, wurden die Experimente zur Visualisierung in UE4 auf die Repräsentation von Kanten beschränkt. Hierbei wurden für die Visualisierung von Funktionsaufrufen Partikelsysteme verwendet.

4.3.1.1 Kanten mit Partikelsystemen

Bei der Erstellung des Partikelsystems wurde an vielen Stellen einem von der Unreal Engine veröffentlichten Video[12] gefolgt. Für das Partikelsystem wurde zuerst ein neues Material angelegt. Dieses beschreibt die visuellen Eigenschaften der Partikel.

Daraufhin wurde ein neues Partikelsystem erzeugt. Dem darin enthaltenen Emitter wurde zuerst das neue Material zugewiesen. Daraufhin wurde dem Emitter ein sogenanntes *Beam Type Data* hinzugefügt. Dieses weise daraufhin, dass der Emitter des Partikelsystems ein Start- und Endpunkt besitzt.[13] Da Funktionsaufrufe immer eine Quelle und ein Ziel haben, ist dies erforderlich. Um den Start- und Endpunkt der Partikel nun festlegen zu können, muss dem Emitter zusätzlich ein *Source* und ein *Target* hinzugefügt werden. Hier kann nun je nach gewählter *Source Method* der Start- und Endpunkt der Partikel einem `AActor` folgen, aber auch direkt über Blueprint festgelegt werden.

An dieser Stelle können Gebäude bereits durch das Partikelsystem miteinander verbunden werden. Um zusätzlich die Aufrufrichtung andeuten zu können, muss allerdings eine weitere Einstellung vorgenommen werden. Im *Beam Data* des Emitters muss die Geschwindigkeit eines Partikels über das Parameter *Speed* erhöht werden. Dies führt dazu, dass sich die Partikel automatisch mit der festgelegten Geschwindigkeit vom Startpunkt in Richtung des Endpunktes bewegen. Werden nun noch einige visuelle Einstellungen vorgenommen, ist das Partikelsystem einsatzbereit.

4.3.2 Unity Engine

Wie bereits in Abschnitt 4.2.2.2 angedeutet, ist das in Abbildung 4.2 dargestellte `Runtime`-Skript hauptsächlich für die Visualisierung verantwortlich. In diesem Skript wird stets der Index des aktuell visualisierten Funktionsaufrufs gespeichert. Mit den in Abschnitt 3.5 beschrieben Interaktion kann dieser Index verändert werden. Daraufhin wird immer automatisch `ClearFunctionCalls()` und `GenerateFunctionCalls()` aufgerufen, um die Visualisierung zu aktualisieren. `ClearFunctionCalls()` löscht dabei zuerst alle bisherigen `FunctionCallSimulator`-Objekte und `GenerateFunctionCalls()` generiert jene für den aktualisierten Index.

Ein `FunctionCallSimulator` ist nun für die tatsächliche Visualisierung verantwortlich. Dieser enthält zum einen die Quelle und das Ziel des jeweiligen Funktionsaufrufs, speichert aber auch die ursprüngliche Farbe und Skalierung. Dies ermöglicht es, die ursprünglichen Werte nach Beendigung der Simulation wiederherstellen zu können.

[12] *Intro to Cascade: Creating a Beam Emitter | 07 | v4.2 Tutorial Series | Unreal Engine*, `https://www.youtube.com/watch?v=ywd3lFOuMV8`, Zugriff: 06.01.2020

[13] *Beam Type Data*, `https://docs.unrealengine.com/en-US/Engine/Rendering/ParticleSystems/Reference/TypeData/Beam/index.html`, Zugriff: 16.01.2020

4.3.2.1 Knoten

Die Knoten werden wie in Abschnitt 3.4.1 beschrieben visualisiert. Die Gebäude werden also zum einen skaliert. Hierbei wird die gespeicherte Ursprungsskalierung anhand einer zeitabhängigen Kosinuskurve folgendermaßen angepasst:

Seien L_t die Dauer eines Schleifendurchgangs in Sekunden, Δ_t der aktuelle Zeitpunkt im Intervall $[0, L_t)$ und $\Delta_{S_{max}}$ die maximale Änderung der Blockgröße. Dann ist

$$t = \frac{\Delta_t}{L_t} \tag{4.1}$$

und somit ist die Skalierung eines Blocks S zum Zeitpunkt Δ_t definiert als:

$$S = \frac{\Delta_{S_{max}}}{2} \cdot (1 + cos(2 \cdot \pi \cdot t + \pi)) \tag{4.2}$$

Die Farbe eines Blocks wird ebenfalls wie im Entwurf beschrieben visualisiert. Ließe sich jede Farbe der Viridis-Palette durch einen Index im Intervall $[0, 1]$ bestimmen, so ist der Index einer Farbe eines Blocks C zum Zeitpunkt Δ_t definiert als:

$$C = \frac{1}{2} \cdot (1 + cos(2 \cdot \pi \cdot t + \pi)) \tag{4.3}$$

4.3.2.2 Kanten mit Kugeln

Bei der Visualisierung fliegen, wie bereits in Abschnitt 3.4.2 beschrieben, Kugeln von der Quelle zum Ziel des Funktionsaufrufs. Die Anzahl der Kugeln wird Anhand des Abstands zwischen der Quelle und dem Ziel bestimmt. Damit die Kugeln nicht asynchron werden können, wird zuerst immer nur die Position der ersten Kugel angepasst und die anderen Kugeln anhand dieser neu positioniert. Die Berechnung der ersten Kugel funktioniert folgendermaßen:

Zuerst wird die Kugel auf die XZ-Ebene, also den Boden projiziert. Anhand der vergangenen Zeit seit dem letzten Frame und einer festgelegten Geschwindigkeit wird die Kugel nun Richtung Ziel translatiert. Überschreitet sie hierbei das Ziel, so wird sie um die Distanz zwischen Quelle und Ziel zurückbewegt, damit sie sich stets zwischen Quelle und Ziel befindet.

Nachdem die erste Kugel auf der XZ-Ebene platziert wurde, werden anhand dieser alle weiteren Kugeln mit gleichem Abstand voneinander sukzessive auf der XZ-Ebene platziert. Dabei wird lineare Interpolation verwendet, um die jeweiligen Abstände zur ersten Kugel zu bestimmen. Überschreitet eine Kugel das Ziel, wird diese an dieser Stelle ebenfalls um die Distanz zwischen Quelle und Ziel zurückbewegt.

Im Folgenden werden nun die Farben der Kugeln bestimmt. Ähnlich wie bereits bei der Visualisierung der Knoten (siehe Abschnitt 4.3.2.1) wird die Farbe anhand eines des Wertes t im Intervall $[0, 1]$ gewählt. t ist genau 0, wenn sich die Kugel auf der XZ-Ebene auf der Position der Quelle und 1, wenn sie sich in der XZ-Ebene auf der Position des Ziel befindet. Die Farbe kann hier also einfach linear Interpoliert werden.

Zu guter Letzt soll die vorher beschriebene bogenartige Fluglinie der Kugeln erreicht werden. Hierfür wird - ebenfalls mithilfe von linearer Interpolation - die Höhe anhand einer Sinuskurve

angepasst. Sei H_{max} die maximale Höhe der Kurve. Dann lässt sich die Höhe einer Kugel H mithilfe eines Wertes t aus dem Intervall $[0, 1]$ definieren als:

$$H = H_{max} \cdot sin(t * \pi) \tag{4.4}$$

Das Ergebnis dieser Berechnungen ist die in Abbildung 3.4 dargestellte Visualisierung.

KAPITEL 5

Evaluation

Im Kontext dieser Arbeit soll die Anwendung am Desktop mit der Anwendung in Virtual Reality verglichen und evaluiert werden.

5.1 Planung

Zu Beginn der Studie wird die gewählte Visualisierung anhand einer Grafik vorgestellt, um die Probanden mit der Visualisierung vertraut zu machen. Es wird hierbei eine statische Grafik (siehe Abbilding 5.1) verwendet, damit sich die Probanden nicht vorab mit der Steuerung am Desktop bzw. VR auseinandersetzen können.

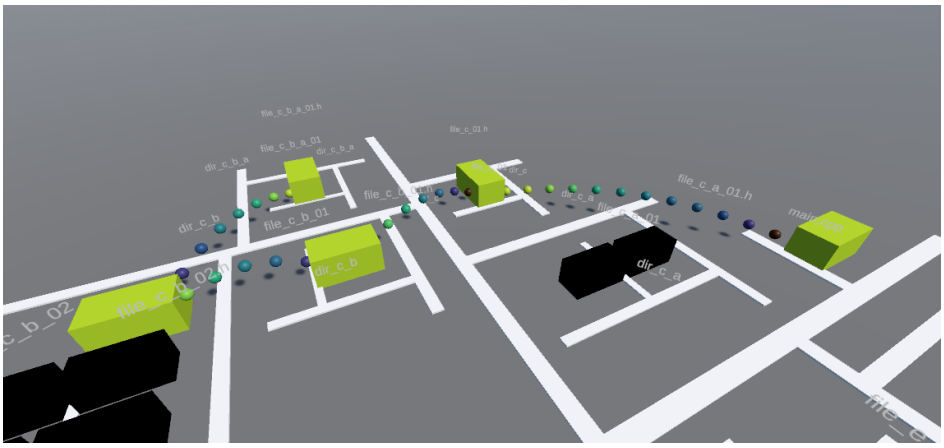


Abbildung 5.1: Ausschnitt der gezeigten Grafik

Die eigentliche Evaluation beinhaltet die Visualisierung zweier Softwaresysteme, in denen die Probanden jeweils die selbe Fragestellung beantworten sollen. Hierbei wird intendiert, dass die Fragestellung beim ersten Softwaresystem etwas leichter zu beantworten ist, um den Probanden einen leichten Start in die Evaluation zu geben. Dies soll durch die Größe der Softwaresysteme realisiert werden. Das erste Softwaresystem enthält acht Dateien, acht Funktionen und zwölf Funktionsaufrufe. Das Zweite hingegen enthält 17 Dateien, 33 Funktionen und 22 Funktionsaufrufe und ist somit deutlich größer als das Erste. Folgende Fragestellung soll von den Probanden beantwortet werden:

> Welche Funktion wird während des dynamischen Programmablaufs am häufigsten aufgerufen?

Gemessen wird die benötigte Zeit fürs Beantworten der Fragestellung für das jeweilige Soft-

waresystem. Zusätzlich wird evaluiert, wie häufig die korrekten Ergebnisse gefunden werden.

Eine Hälfte der Probanden beantwortet die Frage für das erste Softwaresystem am Desktop und für das Zweite in Virtual Reality. Die zweite Hälfte hingegen beantwortet die Frage für das erste Softwaresystem in VR und für das Zweite am Desktop. Dies ermöglicht es, die Unterschiede in Korrektheit und benötigter Zeit leichter miteinander vergleichen zu können.

5.2 Durchführung

An der Studie nahmen zehn Probanden teil. Bei allen der Probanden handelt es sich um Studierende in einem Informatik-zentrierten Studiengang mit Erfahrungen im Bereich der Softwareentwicklung. Zusätzlich hatten alle Probanden bereits vor der Studie Erfahrungen mit Virtual Reality. Durchgeführt wurde die Studie in einem Raum der Universität mit VR-Ausstattung. Jeder Proband nahm separat an der Studie teil und keiner der Probanden hat vor Beginn der Studie Einblicke in die gewählte Visualisierung oder Steuerung bekommen, um die Ergebnisse nicht zu verfälschen.

Die einführende Erklärung folgte für gewöhnlich dem selben Schema. Zuerst wurde dem Probanden das generelle Prinzip von Software-Städten beschrieben. Daraufhin wurde dem Probanden erklärt, um was es sich bei einer dynamischen Analyse handelt. Zu guter Letzt wurde dem Probanden gewählte Visualisierung beschrieben. Währenddessen wurde einige Male nachgefragt, ob vom Probanden alles verstanden wurde, um eventuelle Missverständnisse vor Beginn der Testdurchläufe beseitigen zu können.

Zu Beginn der Testdurchläufe wurde zunächst die jeweilige Steuerung erklärt. Daraufhin sollte sich der Proband mit der Steuerung im jeweiligen System vertraut machen. Sobald der Proband sicher in der Software-Stadt navigieren konnte, wurde die dynamische Visualisierung gestartet und somit wurde begonnen, die Zeit zu stoppen. Nachdem der Proband seine finale Antwort gab, wurde die Stoppuhr beendet. Dem Proband wurde während der Studie nicht verraten, ob die gegebenen Antworten korrekt waren.

5.3 Ergebnisse

Die Tabellen 5.1 und 5.2 stellen die Ergebnisse der Evaluation dar. Im Folgenden werden zuerst die Testdurchläufe miteinander verglichen und anschließend die jeweiligen Testdurchläufe einzeln evaluiert.

Teilnehmer ID	System	Korrekte Antwort	Benötigte Zeit in Minuten
1	Desktop	√	00:42:00
2	VR	√	01:40:40
3	Desktop	√	00:35:46
4	VR	√	03:34:78
5	Desktop	√	01:36:76
6	VR	√	01:30:80
7	Desktop	√	02:22:98
8	VR	√	00:38:33
9	Desktop	√	01:20:12
10	VR	×	01:54:99

Tabelle 5.1: Ergebnisse vom ersten Softwaresystem

Teilnehmer ID	System	Korrekte Antwort	Benötigte Zeit in Minuten
1	VR	√	00:24:49
2	Desktop	√	00:34:07
3	VR	√	01:36:62
4	Desktop	√	00:57:56
5	VR	√	00:40:27
6	Desktop	√	00:51:17
7	VR	√	00:37:42
8	Desktop	√	00:23:88
9	VR	√	01:20:07
10	Desktop	√	03:16:69

Tabelle 5.2: Ergebnisse vom zweiten Softwaresystem

Im Folgenden werden lediglich die Zeiten vergleichend evaluiert. Die Korrektheit der Antworten kann nicht sinnvoll ausgewertet werden, da diese nicht normalverteilt sind. Wie in Tabelle 5.2 zu erkennen ist, wurde die Aufgabe von allen Probanden korrekt erfüllt. Daher ist die Standardabweichung jeder möglichen Teilmenge immer genau null. Es gibt also keine normalverteilte Teilmenge der zweiten Testdurchgangs, die sinnvoll mit einer Teilmenge des ersten Testdurchlaufs verglichen werden kann. Da zusätzlich im ersten Testdurchlauf nur eine inkorrekte Antwort gegeben wurde, kann auch der erste Durchgang nicht in zwei Teilmengen separiert werden, ohne dass eine der beiden Teilmengen eine Standardabweichung von null hat. Die Aufgabenstellung war also entweder zu einfach oder es haben nicht genügend Probanden an der Evaluation teilgenommen, um die Korrektheit der gegebenen Antworten zuverlässig auswerten zu können.

5.3.1 Evaluation zwischen Testdurchläufen

Um ein besseres Verständnis davon zu bekommen, ob es signifikante Differenzen zwischen dem ersten und dem zweiten Testdurchlauf gab, werden zuerst die benötigten Zeiten beider Softwaresysteme miteinander verglichen. Hierbei wird mit einem generellen Vergleich aller Stichproben begonnen, ohne dabei Unterscheidungen zwischen Desktop und Virtual Reality vorzunehmen.

Um die Ergebnisse der Evaluation miteinander vergleichen zu können, wird zum einen der abhängige t-Test[1] verwenden. Mit diesem können Differenzen bezüglich des Mittelwerts zwischen zwei Stichproben untersucht werden. Der abhängige t-Test setzt voraus, dass zwischen den beiden Stichproben eine bijektive Abbildung existiert. Weiterhin wird vorausgesetzt, dass die vorliegenden Daten normalverteilt sind.

Zuerst muss eine Nullhypothese H_0 aufgestellt werden, welche im Folgenden widerlegt werden möchte. Im Fall dieser Studie ist dies generell die Hypothese $H_0 : \mu_0 = \mu_1$, also die Hypothese, dass zwei Erwartungswerte sich gleichen. Daraufhin muss zwischen allen n Paaren jeweils die Differenz bestimmt werden. Von diesen ausgehend werden anschließend das arithmetische Mittel $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ und die Standardabweichung $\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x}^2}{n}}$ ermittelt. Damit folgt dann die t-Statistik $t = \sqrt{n} \cdot \frac{\bar{x}}{\sigma}$. Gegeben ein Signifikanzniveau und ein Freiheitsgrad $(n-1)$ kann nun in Tabelle 1 abgelesen werden, ob $|t|$ größer ist als der Wert in der Tabelle. Ist dies der Fall, kann die Nullhypothese verworfen werden.

[1] Gosset, W. S., *Student: The Probable Error of a Mean*, Biometrika. 6, Nr. 1, 1908, S. 1 - 25.

Zusätzlich dazu wird der Zweistichproben-t-Test[2] verwendet. Dieser ermöglicht den Vergleich zweier voneinander unabhängigen Stichproben. Die t-Statistik lässt sich hierbei als

$$t = \frac{\bar{x}_0 - \bar{x}_1}{\sqrt{\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}}}| \tag{5.1}$$

definieren, wobei $\bar{x}_0$ bzw. $\bar{x}_1$ als das arithmetische Mittel, σ_0 bzw. σ_1 als die Standardabweichung und n_0 und n_1 als die Größe der ersten bzw. zweiten Stichprobe gilt. Der restliche Vorgang gleicht dem des abhängigen t-Tests.

5.3.1.1 Genereller Vergleich

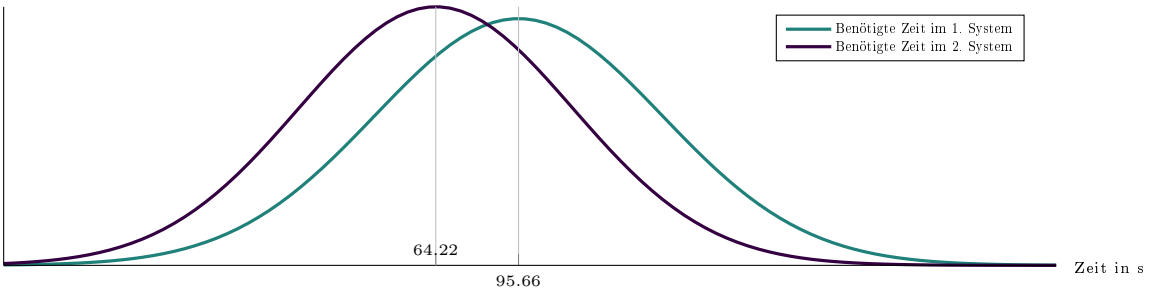


Abbildung 5.2: Differenz der benötigten Zeiten zwischen dem ersten und dem zweiten Softwaresystem

Abbildung 5.2 visualisiert vergleichend die Normalverteilungen der Zeiten beider Testdurchläufe. Die Abbildung lässt vermuten, dass das zweite Testbeispiel für die Probanden leichter zu lösen war. Dies soll im Folgenden mithilfe des t-Tests untersucht werden.

Sei $H_0 : \mu_1 = \mu_2$, wobei μ_1 und μ_2 die Erwartungswerte für die benötigte Zeit im ersten bzw. zweiten Softwaresystem sind. H_0 soll mit einer Konfidenz von 80% abgelehnt werden. Hierfür müssen zunächst die Differenzen in benötigter Zeit zwischen den Softwaresystemen bestimmt werden. Für jede Zeit t_{1_i} des ersten und t_{2_j} des zweiten Testdurchlaufs, zwischen denen eine bijektive Abbildung existiert, wird die Differenz als $\Delta_{t_{1,2}} = t_{1_i} - t_{2_j}$ definiert. Die Tabelle 5.3 zeigt die Zeitdifferenz pro Teilnehmer zwischen dem ersten und zweiten Softwaresystem.

Teilnehmer ID	Differenz in Sekunden
1	17.51
2	66.33
3	-61.16
4	157.22
5	56.51
6	39.63
7	105.56
8	14.45
9	0.05
10	-81.70

Tabelle 5.3: Zeitreduktion der Probanden vom ersten zum zweiten Testdurchlauf

[2] ebd.

Von den Differenzen lässt sich dann das arithmetische Mittel $\bar{x}_{\Delta 1,2} = 31,44$, die Standardabweichung $\sigma_{\Delta 1,2} \approx 71,55$ und schließlich die t-Statistik $t_{\Delta 1,2} = \sqrt{n_{\Delta 1,2}} \cdot \frac{\bar{x}_{\Delta 1,2}}{\sigma_{\Delta 1,2}} \approx 1,39$ ermitteln. Da $n_{\Delta 1,2} = 10$, gibt es $n_{\Delta 1,2} - 1 = 9$ Freiheitsgrade. Aus der in Tabelle 1 gegebenen t-Verteilung lässt sich für ein Signifikanzniveau von $0,9$ und einem Freiheitsgrad von 9 der kritische Wert $1,383$ ablesen. Da $t_{\Delta 1,2}$ in diesem Fall größer ist als dieser kritische Wert, kann die Nullhypothese verworfen werden und somit sind die Erwartungswerte beider Teilergebnisse mit einer Konfidenz von 80% unterschiedlich.

5.3.1.2 Desktop

Ich möchte nun evaluieren, ob sich die Zeiten beim Verwenden von Desktop zwischen dem ersten und dem zweiten Softwaresystem signifikant unterschieden haben. Da jeder Proband nur eine Aufgabe am Desktop gelöst hat und somit keine bijektive Abbildung existiert, wird im Folgenden der Zweistichproben-t-Test verwendet.

Sei $H_0 : \mu_{1_D} = \mu_{2_D}$ die Nullhypothese, wobei μ_{1_D} und μ_{2_D} die Erwartungswerte für die benötigte Zeit am Desktop im ersten bzw. zweiten Softwaresystem sind. H_0 soll mit einer Konfidenz von 80% abgelehnt werden.

$\bar{x}_{1_D} = 79,468$ und $\bar{x}_{2_D} = 72,674$ sind die arithmetischen Mittel, $\sigma_{1_D} \approx 43,802$ und $\sigma_{2_D} \approx 70,608$ die Standardabweichungen und $n_{1_D} = 5$ und $n_{2_D} = 5$ die Anzahlen der Stichproben am Desktop im ersten und zweiten Softwaresystem. Daraus folgt dann die t-Statistik:

$$t_{D_{1,2}} = \frac{\mu_{1_D} - \mu_{2_D}}{\sqrt{\frac{\sigma_{1_D}^2}{n_{1_D}} + \frac{\sigma_{2_D}^2}{n_{2_D}}}} \approx 0,18 \tag{5.2}$$

In der t-Verteilung der Tabelle 1 findet man unter dem Freiheitsgrad $n - 1 = 4$ und dem Signifikanzniveau von $0,9$ den kritischen Wert von $1,533$. Da allerdings $|t_{D_{1,2}}| \approx 0,18$ und somit $1,533 \not< |t_{D_{1,2}}|$ gilt, kann die Nullhypothese nicht verworfen werden.

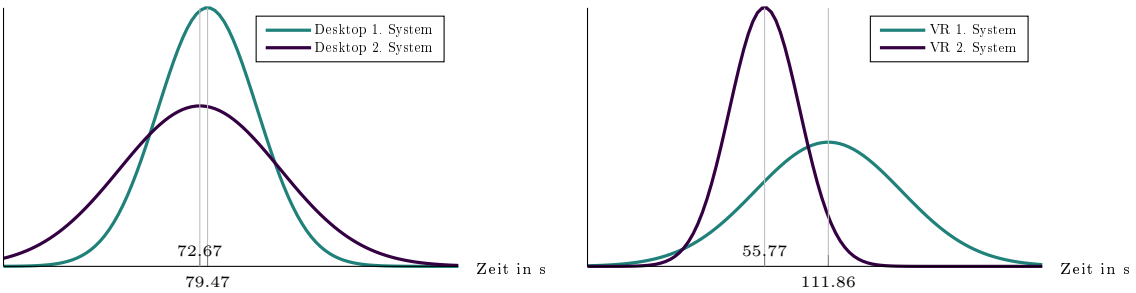


Abbildung 5.3: Vergleich der benötigten Zeiten zwischen dem ersten und dem zweiten Softwaresystem am Desktop bzw. in VR

5.3.1.3 Virtual Reality

Ähnlich wie auch die Desktop-Variante werden im Folgenden die in Virtual Reality benötigten Zeiten des ersten und des zweiten Softwaresystem mittels des Zweistichproben-t-Tests miteinander verglichen.

Sei $H_0 : \mu_{1_{VR}} = \mu_{2_{VR}}$ die Nullhypothese, wobei $\mu_{1_{VR}}$ und $\mu_{2_{VR}}$ analog zu oben im Bezug auf VR definiert sind. H_0 soll ebenfalls mit einer Konfidenz von 80% abgelehnt werden.

$\bar{x}_{1_{VR}} = 111,86$ und $\bar{x}_{2_{VR}} = 55,774$ sind die arithmetischen Mittel, $\sigma_{1_{VR}} \approx 64,389$ und $\sigma_{2_{VR}} \approx 30,881$ die Standardabweichungen und $n_{1_{VR}} = 5$ und $n_{2_{VR}} = 5$ die Anzahlen der Stichproben in VR im ersten und im zweiten Softwaresystem. Daraus folgt dann die t-Statistik:

$$t_{VR_{1,2}} = \frac{\mu_{1_{VR}} - \mu_{2_{VR}}}{\sqrt{\frac{\sigma^2_{1_{VR}}}{n_{1_{VR}}} + \frac{\sigma^2_{2_{VR}}}{n_{2_{VR}}}}} \approx 1,76 \tag{5.3}$$

Wie bereits zuvor ist der kritische Wert $1,533$. Da $|t_{VR_{1,2}}| \approx 1,76$ und somit $1,533 < |t_{VR_{1,2}}|$ gilt, kann die Nullhypothese verworfen werden und somit sind die Erwartungswerte beider Teilergebnisse mit einer Konfidenz von 80% unterschiedlich.

5.3.2 Evaluation innerhalb von Testdurchläufen

Um mögliche Unterschiede innerhalb einzelner Testdurchläufe zu detektieren, sollen die Testdurchläufe im Folgenden einzeln evaluiert werden. Es wird jeweils Desktop mit Virtual Reality verglichen.

5.3.2.1 Erster Testdurchlauf

Im Folgenden sollen die benötigten Zeiten im ersten Testdurchlauf der Desktop-Version mit der VR-Version vergleichend evaluiert werden. Für die Evaluation wird der Zweistichproben-t-Test verwendet.

Sei $H_0 : \mu_{1_D} = \mu_{1_{VR}}$ die Nullhypothese, wobei μ_{1_D} und $\mu_{1_{VR}}$ die Erwartungswerte für die benötigte Zeit des ersten Softwaresystem am Desktop bzw. in VR sind. H_0 soll mit einer Konfidenz von 80% abgelehnt werden.

$\bar{x}_{1_D} = 79,468$ und $\bar{x}_{1_{VR}} = 111,86$ sind die arithmetischen Mittel, $\sigma_{1_D} \approx 43,802$ und $\sigma_{1_{VR}} \approx 64,389$ die Standardabweichungen und $n_{1_D} = 5$ und $n_{1_{VR}} = 5$ die Anzahl der Stichproben im ersten Softwaresystem am Desktop und in VR. Daraus folgt dann die t-Statistik:

$$t_{1_{D,VR}} = \frac{\bar{x}_{1_D} - \bar{x}_{1_{VR}}}{\sqrt{\frac{\sigma^2_{1_D}}{n_{1_D}} + \frac{\sigma^2_{1_{VR}}}{n_{1_{VR}}}}} \approx -0,93 \tag{5.4}$$

Aus der Tabelle der t-Verteilung lässt sich für ein Signifikanzniveau von $0,9$ und einem Freiheitsgrad von $n - 1 = 4$ der kritische Wert $1,533$ ablesen. Da allerdings $|t_{1_{D,VR}}| = 0,93$ und somit $1,533 \not< |t_{1_{D,VR}}|$ gilt, lässt sich die Nullhypothese nicht verwerfen.

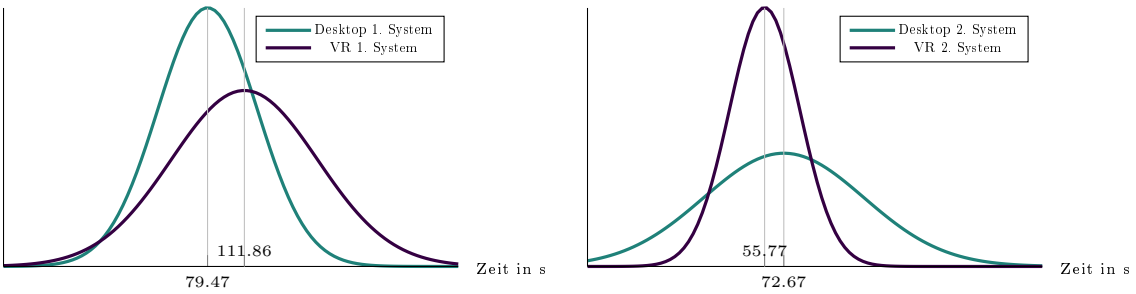


Abbildung 5.4: Vergleich der benötigten Zeiten zwischen Desktop und VR im ersten bzw. zweiten Softwaresystem

5.3.2.2 Zweiter Testdurchlauf

Sei $H_0 : \mu_{2_D} = \mu_{2_{VR}}$ die Nullhypothese, wobei μ_{2_D} und $\mu_{2_{VR}}$ analog zu oben fürs zweite Softwaresystem definiert sind. H_0 soll ebenfalls mit einer Konfidenz von 80% abgelehnt werden.

$\bar{x}_{2_D} = 72,674$ und $\bar{x}_{2_{VR}} = 55,774$ sind die arithmetischen Mittel, $\sigma_{2_D} \approx 70,608$ und $\sigma_{2_{VR}} \approx 30,881$ die Standardabweichungen und $n_{2_D} = 5$ und $n_{2_{VR}} = 5$ die Anzahlen der Stichproben im zweiten Softwaresystem am Desktop und in VR. Daraus folgt dann die t-Statistik:

$$t_{2_{D,VR}} = \frac{\mu_{2_D} - \mu_{2_{VR}}}{\sqrt{\frac{\sigma^2_{2_D}}{n_{2_D}} + \frac{\sigma^2_{2_{VR}}}{n_{2_{VR}}}}} \approx 0,49 \tag{5.5}$$

Auch hier kann die Nullhypothese mit dem selben kritischen Wert von $1,533$ wegen $1,533 \not< |t_{2_{D,VR}}| = 0,49$ nicht verworfen werden.

5.3.3 Fazit

Es konnte mit einer Konfidenz von 80% gezeigt werden, dass sich er Erwartungswert für die benötigte Zeit vom ersten zum zweiten Softwaresystem verkleinert. Es kann also angenommen werden, dass den Probanden das zweite Beispiel leichter gefallen ist. Dies könnte verschiedene Gründe gehabt haben. Zum einen kann die Visualisierung durch einen Lerneffekt vom ersten zum zweiten Softwaresystem leichter fallen. Zu anderen ist es allerdings auch möglich, dass das zweite Beispiel einfach generell simpler war.

Zusätzlich konnte mit einer Konfidenz von 80% gezeigt werden, dass sich er Erwartungswert für die benötigte Zeit in Virtual Reality vom ersten zum zweiten Softwaresystem reduziert. Für den Desktop konnte dies nicht gezeigt werden. Dies könnte zum einen bedeuten, dass die benötigte Zeit mit mehr Erfahrung mit der gewählten Visualisierung in Virtual Reality stärker abnimmt, als am Desktop. Um dies bestätigen zu können, müssen allerdings noch weitere Untersuchungen vorgenommen werden. Zusätzlich spielt möglicherweise die Größe einer Software-Stadt eine signifikante Rolle. Die zweite Software-Stadt war erheblich größer als die erste, weshalb diese im Allgemeinen möglicherweise als unübersichtlicher wahrgenommen wurde. Hier wäre es möglich, dass Virtual Reality eine bessere Orientierung und Übersichtlichkeit über die Software-Stadt bietet und somit eine schnellere Lösungszeit unterstützt. Auch dies ist lediglich eine Vermutung, welche aufbauend auf diese Arbeit weiter untersucht werden kann.

KAPITEL 6

Ausblick

Im Laufe dieser Arbeit haben sich zusätzlich viele interessante Fragestellungen ergeben, auf die in den folgenden Abschnitten detaillierter eingegangen wird.

6.1 Automatisierte Instrumentierung in C++

Im Kontext dieser Arbeit wurde für C++ eine manuelle Instrumentierung gewählt. Möchte man allerdings komplexere Softwaresysteme analysieren, so wäre eine manuelle Instrumentierung äußerst unpraktikabel. Aufbauend auf diese Arbeit könnte ein automatisches System zur Instrumentierung entwickelt werden, um die dynamische Analyse von C++-Programmen zu vereinfachen.

6.2 Augmented Reality

In dieser Arbeit wurde lediglich Desktop mit Virtual Reality verglichen. Zusätzlich dazu wäre eine Evaluation von Augmented Reality interessant. Hier könnte der Entwickler beispielsweise seine eigene Software-Stadt auf seinen Schreibtisch stellen und in Echtzeit die virtuelle Software-Stadt mit der tatsächlichen Software am PC miteinander vergleichen. Dies könnte für den Entwickler das Verständnis der Abbildung vom Softwaresystem auf die Software-Stadt vereinfachen.

Abbildung 6.1: Konzept einer Software-Stadt in Augmented Reality.[1]

6.3 Visualisierung

Auch bei der Visualisierung ergeben sich weitere interessante Fragestellungen, die aufbauend auf diese Arbeit evaluiert werden könnten.

6.3.1 Zusätzliche Metriken

Aktuell werden bei der Visualisierung lediglich die Quelle und das Ziel der Funktionsaufrufe miteinander verbunden und zusätzlich die Aufrufrichtung angedeutet. Allerdings sind oftmals viele weitere Metriken relevant für den Entwickler.

Eine für viele Entwickler relevante Metrik ist die Aufrufdauer, um langsame Prozeduren finden zu können. Diese könnte beispielsweise durch die Gebäudefarbe visualisiert werden. Je länger eine Funktion läuft, desto saturierter könnte die jeweilige Gebäudefarbe werden. Auf eine ähnliche weise ließe sich ebenfalls die Aufrufhäufigkeit visualisieren. Hierbei könnte jeder Funktionsaufruf den roten Farbwert des entsprechenden Gebäudes erhöhen. Wird eine Funktion nicht aufgerufen, so könnte sich der rote Farbwert langsam wieder reduzieren. Zusätzlich ließe sich die Aufrufhäufigkeit als statische Heatmap darstellen. Hierbei könnte der Untergrund der Stadt je nach Aufrufhäufigkeit der darauf befindlichen Gebäude eingefärbt werden.

Hat der Entwickler einen interessanten Teil der Stadt gefunden und möchte genauere Werte für die jeweiligen Metriken erhalten, könnten diese zusätzlich in einem Menü dargestellt werden.

6.3.2 Beendete Prozeduren

Ein Teilnehmer der Studie hatte den Vorschlag, beendete Prozeduren visuell hervorzuheben. Häufig kommt es vor, dass von einem Schritt zum nächsten eine Unterprozedur beendet und von der überliegenden Funktion ein neuer Funktionsaufruf getätigt wird. Aktuell wird dies allerdings in keinster Weise visuell angedeutet. Abbildung 6.2 stellt dar, wie beendete Prozeduren visualisiert werden könnten. In der ersten Abbildung ruft die Funktion aus ”file_c_01” eine Funktion aus ”file_c_a_01” auf. Die zweite Abbildung zeigt den Folgeschritt. Hier wurde die Unterprozedur bereits beendet und die Funktion in ’file_c_b_01” aufgerufen. Das entsprechende Gebäude wurde rot hervorgehoben, um deutlicher zu visualisieren, dass die Prozedur beendet wurde. Aber auch andere Visualisierungstechniken wären denkbar.

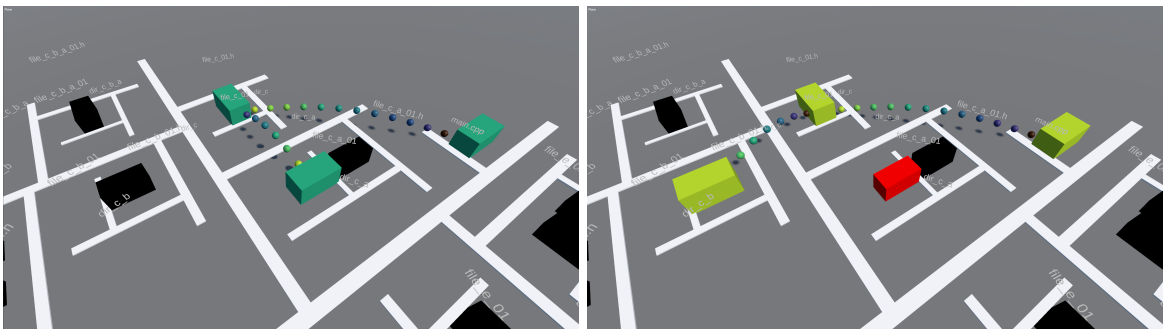


Abbildung 6.2: Mögliche Visualisierung von beendeten Prozeduren

[1] Roman Bozhko, *Clean minimalist office*, `https://unsplash.com/photos/PypjzKTUqLo`, 2017, Zugriff: 19.01.2020

6.3.3 Bundled Edges

Danny Holten beschreibt 2006 die Visualisierungsmethode der Bundled Edges.[2] Die Idee hierbei ist, dass Kanten sich gegenseitig "anziehen". Abbildung 6.3 zeigt, wie eine solche Bündelung im Zweidimensionalen aussehen kann.

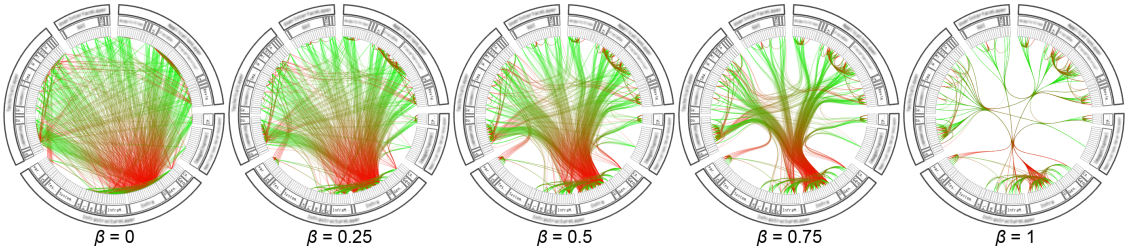


Abbildung 6.3: Bundled Edges mit unterschiedlich starken Bündelungen.[3]

In dieser Arbeit wurden bezüglich der XZ-Ebene lediglich direkt Pfade zwischen Gebäuden gewählt. Aufbauend auf diese Arbeit könnte allerdings zusätzlich mit gebündelten Kanten experimentiert werden.

[2] Danny Holten, *Hierarchicel Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data*, IEEE Transactions on Visualization and Computer Graphics, Vol. 12, No. 5, 2006.
[3] ebd., 6.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

LITERATURVERZEICHNIS

[1] *Beam Type Data*. https://docs.unrealengine.com/en-US/Engine/Rendering/ParticleSystems/Reference/TypeData/Beam/index.html. Zugriff: 16.01.2020.

[2] *Über die VIVE Controller*. https://www.vive.com/de/support/vive/category_howto/about-the-controllers.html. Zugriff: 28.12.2019.

[3] *Blueprints Visual Scripting*. https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html. Zugriff: 09.01.2020.

[4] *Class ClassPool*. http://www.javassist.org/html/javassist/ClassPool.html. Zugriff: 15.01.2020.

[5] *Class CtMethod*. https://www.javassist.org/html/javassist/CtMethod.html. Zugriff: 02.01.2020.

[6] *FSlateStyleSet*. https://docs.unrealengine.com/en-US/API/Runtime/SlateCore/Styling/FSlateStyleSet/index.html. Zugriff: 15.01.2020.

[7] *Graph eXchange Language*. http://www.gupro.de/GXL/Introduction/background.html. Zugriff: 01.01.2020.

[8] *Interface ClassFileTransformer*. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/ClassFileTransformer.html. Zugriff: 02.01.2020.

[9] *Interface Instrumentation*. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html. Zugriff: 27.12.2019.

[10] *Intro to Cascade: Creating a Beam Emitter | 07 | v4.2 Tutorial Series | Unreal Engine*. https://www.youtube.com/watch?v=ywd3lFOuMV8. Zugriff: 06.01.2020.

[11] *Package java.lang.instrument*. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html. Zugriff: 15.01.2020.

[12] *Scripting*. https://docs.unity3d.com/Manual/ScriptingSection.html. Zugriff: 09.01.2020.

[13] *Scripting the Editor using Python*. https://docs.unrealengine.com/en-US/Engine/Editor/ScriptingAndAutomation/Python/index.html. Zugriff: 09.01.2020.

[14] *Table of Critical Values for Student's t Distributions*. Uni Konstanz. Zugriff 11.01.2020.

[15] *Unity Roadmap*. https://unity3d.com/unity/roadmap. Zugriff: 09.01.2020.

[16] *Unreal Engine API Reference*. https://docs.unrealengine.com/en-US/API/index.html. Zugriff: 09.01.2020.

[17] Axon, S.: *Unity at 10: For better - or worse - game development has never been easier*. https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/, 2016. Zugriff: 09.01.2020.

[18] Bozhko, R.: *Clean minimalist office*. https://unsplash.com/photos/PypjzKTUqLo, 2017. Zugriff: 19.01.2020.

[19] Chikofsky, E. J. und J. H. C. II: *Reverse Engineering and Design Recovery: A Taxonomy*. 1990.

[20] Diehl, S.: *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. 2007.

[21] Edwards, B.: *From The Past To The Future: Tim Sweeney Talks*. https://www.gamasutra.com/view/feature/4035/from_the_past_to_the_future_tim_.php, 2009. Zugriff: 09.01.2020.

[22] Frese, U.: *Grundlagen der Medieninformatik 1: Menschliche Wahrnehmung*. 2016.

[23] González-Barahona, J. M., M. A. O. Pérez, P. de las Heras Quirós, J. C. González und V. M. Olivera: *Counting potatoes: the size of Debian 2.2*, 2008. Zugriff: 12.01.2020.

[24] Gosset, W. S.: *Student: The Probable Error of a Mean*. Biometrika. 6, Nr. 1, 1908.

[25] Hogenson, G., S. Cai, G. Warren, M. Jones, M. McGee, N. Schonning, J. Perazzo, S. Shetty, M. Blome und C. Caserio: *Save and export performance tools data*. https://docs.microsoft.com/en-us/visualstudio/profiling/saving-and-exporting-performance-tools-data?view=vs-2017. Zugriff: 19.01.2020.

[26] Hogenson, G., S. Cai, G. Warren, M. Jones, M. McGee, S. Shetty, N. Schonning und M. Blome: *Call Tree view*. https://docs.microsoft.com/en-us/visualstudio/profiling/call-tree-view?view=vs-2017. Zugriff: 26.12.2019.

[27] Hogenson, G., S. Cai, G. Warren, M. Jones, M. McGee, S. Shetty, N. Schonning und M. Blome: *Call Tree view - sampling data*. https://docs.microsoft.com/en-us/visualstudio/profiling/call-tree-view-sampling-data?view=vs-2017. Zugriff: 19.01.2020.

[28] Holten, D.: *Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data*. IEEE Transactions on Visualization and Computer Graphics, Vol. 12, Nr. 5, 2006.

[29] Horvath, S.: *The Imagination Engine: Why Next-Gen Videogames Will Rock Your World*. https://www.wired.com/2012/05/ff-unreal4/, 2012. Zugriff: 09.01.2020.

[30] Kroah-Hartman, G., J. Corbet und A. McPherson: *Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*. 2012. Zugriff: 12.01.2020.

[31] Leemhuis, T.: *What's new in Linux 2.6.32*. https://web.archive.org/web/20131219054613/http://www.h-online.com/open/features/What-s-new-in-Linux-2-6-32-872271.html?view=print, 2009. Zugriff: 12.01.2020.

[32] Leemhuis, T.: *What's new in Linux 3.6*. https://web.archive.org/web/20131219054847/http://www.h-online.com/open/features/What-s-new-in-Linux-3-6-1714690.html?page=3, 2012. Zugriff: 12.01.2020.

[33] Leemhuis, T.: *Linux-Kernel durchbricht die 20-Millionen-Zeilen-Marke*. https://www.heise.de/newsticker/meldung/Linux-Kernel-durchbricht-die-20-Millionen-Zeilen-Marke-2730780.html, 2015. Zugriff: 12.01.2020.

[34] Quante, J. und R. Koschke: *Dynamic Object Process Graphs*. Journal of Systems and Software, 2008.

[35] Reeves, W. T.: *Particle Systemsm: Technique for Modeling a Class of Fuzzy Objects*. ACM Transactions on Graphics, Vol. 2, 1983.

[36] Robles, G.: *Debian Counting*, 2013.

[37] Rudis, B., N. Ross und S. Garnier: *The viridis color palettes*. https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html, 2018. Zugriff: 28.12.2019.

[38] Safyallah, H. und K. Sartipi: *Dynamic Analysis of Software Systems using Execution Pattern Mining*. 14th IEEE International Conference on Program Comprehension, 2006.

[39] Schumann, H. und W. Müller: *Visualisierung: Grundlagen und allgemeine Methoden*. Springer-Verlag Berlin Heidelberg 2000, 2013.

[40] Sherrod, A.: *Ultimate 3D Game Engine Design & Architecture*, 2007.

[41] Steuer, J.: *Defining Virtual Reality: Dimensions Determining Telepresence*. 1993.

[42] Tashtoush, Y., M. Al-Maolegi und B. Arkok: *The Correlation among Software Complexity Metrics with Case Study*. International Journal of Advanced Computer Research, Vol. 4, Nr. 3, 2014.

[43] Wertheimer, M.: *Untersuchung zur Lehre von der Gestalt (II)*. 1923.

[44] Wettel, R. und M. Lanza: *Visualizing Software Systems as Cities*, 2007.

[45] Wettel, R. und M. Lanza: *CodeCity: 3D Visualization of Large-Scale Software*, 2008.

[46] Weyhe, D., V. Uslar, F. Weyhe, M. Kaluschke und G. Zachman: *Immersive Anatomy Atlas: Empirical Study Investigating the Usability of a Virtual Reality Environment as a Learning Tool for Anatomy*. 2018.

[47] Wheeler, D. A.: *More Than a Gigabuck: Estimating GNU/Linux's Size*. https://dwheeler.com/sloc/redhat71-v1/redhat71sloc.html, 2001. Zugriff: 12.01.2020.

Zusätzliche Anhänge

Da die originalen Varianten der folgenden Dokumente gemeinsam weit über 10.000 Zeilen beinhalten, folgen ausschließlich die verkürzten Varianten. An Stellen mit `[...]` wurde Inhalt entfernt.

Listing 1: Verkürzte CSV vom Export der Visual Studio Aufrufstrukturansicht

```
Level,Function Name,Inclusive Samples,Exclusive Samples,Inclusive Samples %,Exclusive Samples %,
   Module Name,
0,"Creek.exe",2.958,0,"100,00","0,00","",
0,"[ntdll.dll]",2.958,0,"100,00","0,00","ntdll.dll",
1,"[kernel32.dll]",2.949,0,"99,70","0,00","kernel32.dll",
2,"mainCRTStartup",2.350,0,"79,45","0,00","Creek.exe",
3,"__scrt_common_main",2.350,0,"79,45","0,00","Creek.exe",
4,"__scrt_common_main_seh",2.350,0,"79,45","0,00","Creek.exe",
5,"invoke_main",2.348,0,"79,38","0,00","Creek.exe",
6,"main",2.348,0,"79,38","0,00","Creek.exe",
7,"std::make_shared<TestLayer>",1.424,0,"48,14","0,00","Creek.exe",
8,"std::_Ref_count_obj<TestLayer>::_Ref_count_obj<TestLayer><>",1.424,0,"48,14","0,00","Creek.exe",
[...]
12,"[ntdll.dll]",1,0,"0,03","0,00","ntdll.dll",
13,"[ntdll.dll]",1,0,"0,03","0,00","ntdll.dll",
14,"[ntdll.dll]",1,0,"0,03","0,00","ntdll.dll",
15,"[ntdll.dll]",1,1,"0,03","0,03","ntdll.dll",
5,"[KernelBase.dll]",1,0,"0,03","0,00","KernelBase.dll",
6,"[KernelBase.dll]",1,0,"0,03","0,00","KernelBase.dll",
7,"[KernelBase.dll]",1,0,"0,03","0,00","KernelBase.dll",
8,"[KernelBase.dll]",1,0,"0,03","0,00","KernelBase.dll",
9,"[KernelBase.dll]",1,0,"0,03","0,00","KernelBase.dll",
10,"[ntdll.dll]",1,1,"0,03","0,03","ntdll.dll",
```

Listing 2: Verkürztes Ergebnis der Konvertierung von CSV aus Abbildung 1 zum ersten DYN-Format

```
attribute "Module Name" ""
attribute "Module Name" "ntdll.dll"
attribute "Module Name" "kernel32.dll"
attribute "Module Name" "Creek.exe"
attribute "Module Name" "ucrtbased.dll"
attribute "Module Name" "KernelBase.dll"
attribute "Module Name" "vcruntime140d.dll"
attribute "Module Name" "opengl32.dll"
attribute "Module Name" "nvoglv64.dll"
attribute "Module Name" "msvcp140d.dll"
[...]
attribute "Module Name" "cfgmgr32.dll"
attribute "Module Name" "wintrust.dll"
attribute "Module Name" "advapi32.dll"
attribute "Module Name" "SHCore.dll"
attribute "Module Name" "DXCore.dll"
attribute "Module Name" "dwmapi.dll"
attribute "Module Name" "wtsapi32.dll"
attribute "Module Name" "InputHost.dll"
attribute "Module Name" "combase.dll"
attribute "Module Name" "ucrtbase.dll"
attribute "Function Name" "Creek.exe"
attribute "Function Name" "[ntdll.dll]"
attribute "Function Name" "[kernel32.dll]"
attribute "Function Name" "mainCRTStartup"
attribute "Function Name" "__scrt_common_main"
attribute "Function Name" "__scrt_common_main_seh"
attribute "Function Name" "invoke_main"
attribute "Function Name" "main"
attribute "Function Name" "std::make_shared<TestLayer>"
attribute "Function Name" "std::_Ref_count_obj<TestLayer>::_Ref_count_obj<TestLayer><>"
[...]
attribute "Function Name" "_glfwPlatformDestroyWindow"
attribute "Function Name" "Molly::RenderContext::Shutdown"
attribute "Function Name" "Molly::OpenGL_RenderContext::ShutdownImpl"
attribute "Function Name" "glfwTerminate"
attribute "Function Name" "[DXCore.dll]"
attribute "Function Name" "[dwmapi.dll]"
attribute "Function Name" "[wtsapi32.dll]"
attribute "Function Name" "[InputHost.dll]"
attribute "Function Name" "[combase.dll]"
attribute "Function Name" "[ucrtbase.dll]"
attribute "Level" "0"
```

```
attribute "Level" "1"
attribute "Level" "2"
attribute "Level" "3"
attribute "Level" "4"
attribute "Level" "5"
attribute "Level" "6"
attribute "Level" "7"
attribute "Level" "8"
attribute "Level" "9"
[..]
attribute "Level" "92"
attribute "Level" "93"
attribute "Level" "94"
attribute "Level" "95"
attribute "Level" "96"
attribute "Level" "97"
attribute "Level" "98"
attribute "Level" "99"
attribute "Level" "100"
attribute "Level" "101"
attribute "Exclusive Samples" "0"
attribute "Exclusive Samples" "57"
attribute "Exclusive Samples" "306"
attribute "Exclusive Samples" "171"
attribute "Exclusive Samples" "32"
attribute "Exclusive Samples" "6"
attribute "Exclusive Samples" "9"
attribute "Exclusive Samples" "5"
attribute "Exclusive Samples" "3"
attribute "Exclusive Samples" "7"
[...]
attribute "Exclusive Samples" "28"
attribute "Exclusive Samples" "24"
attribute "Exclusive Samples" "15"
attribute "Exclusive Samples" "14"
attribute "Exclusive Samples" "16"
attribute "Exclusive Samples" "12"
attribute "Exclusive Samples" "18"
attribute "Exclusive Samples" "33"
attribute "Exclusive Samples" "153"
attribute "Exclusive Samples" "88"
attribute "Exclusive Samples %" "0.000000"
attribute "Exclusive Samples %" "1.930000"
attribute "Exclusive Samples %" "10.340000"
attribute "Exclusive Samples %" "5.780000"
attribute "Exclusive Samples %" "1.080000"
attribute "Exclusive Samples %" "0.200000"
attribute "Exclusive Samples %" "0.300000"
attribute "Exclusive Samples %" "0.170000"
attribute "Exclusive Samples %" "0.100000"
attribute "Exclusive Samples %" "0.240000"
[...]
attribute "Exclusive Samples %" "0.950000"
attribute "Exclusive Samples %" "0.810000"
attribute "Exclusive Samples %" "0.510000"
attribute "Exclusive Samples %" "0.470000"
attribute "Exclusive Samples %" "0.540000"
attribute "Exclusive Samples %" "0.410000"
attribute "Exclusive Samples %" "0.610000"
attribute "Exclusive Samples %" "1.120000"
attribute "Exclusive Samples %" "5.170000"
attribute "Exclusive Samples %" "2.970000"
attribute "Inclusive Samples" "2958"
attribute "Inclusive Samples" "2949"
attribute "Inclusive Samples" "2350"
attribute "Inclusive Samples" "2348"
attribute "Inclusive Samples" "1424"
attribute "Inclusive Samples" "921"
attribute "Inclusive Samples" "904"
attribute "Inclusive Samples" "886"
attribute "Inclusive Samples" "885"
attribute "Inclusive Samples" "882"
[...]
attribute "Inclusive Samples" "88"
attribute "Inclusive Samples" "84"
attribute "Inclusive Samples" "72"
attribute "Inclusive Samples" "71"
attribute "Inclusive Samples" "80"
attribute "Inclusive Samples" "70"
attribute "Inclusive Samples" "60"
attribute "Inclusive Samples" "46"
attribute "Inclusive Samples" "162"
attribute "Inclusive Samples" "153"
attribute "Inclusive Samples %" "100.000000"
attribute "Inclusive Samples %" "99.699997"
attribute "Inclusive Samples %" "79.449997"
attribute "Inclusive Samples %" "79.379997"
attribute "Inclusive Samples %" "48.139999"
attribute "Inclusive Samples %" "31.139999"
attribute "Inclusive Samples %" "30.559999"
attribute "Inclusive Samples %" "29.950001"
attribute "Inclusive Samples %" "29.920000"
attribute "Inclusive Samples %" "29.820000"
[...]
attribute "Inclusive Samples %" "2.970000"
attribute "Inclusive Samples %" "2.840000"
```

```
attribute "Inclusive Samples %" "2.430000"
attribute "Inclusive Samples %" "2.400000"
attribute "Inclusive Samples %" "2.700000"
attribute "Inclusive Samples %" "2.370000"
attribute "Inclusive Samples %" "2.030000"
attribute "Inclusive Samples %" "1.560000"
attribute "Inclusive Samples %" "5.480000"
attribute "Inclusive Samples %" "5.170000"

node attributes 0 35 833 935 974 1013 1137
node attributes 1 36 833 935 974 1013 1137
node attributes 2 37 834 935 974 1014 1138
node attributes 3 38 835 935 974 1015 1139
node attributes 3 39 836 935 974 1015 1139
node attributes 3 40 837 935 974 1015 1139
node attributes 3 41 838 935 974 1016 1140
node attributes 3 42 839 935 974 1016 1140
node attributes 3 43 840 935 974 1017 1141
node attributes 3 44 841 935 974 1017 1141
[...]
node attributes 9 138 841 935 974 1034 1158
node attributes 4 74 842 935 974 1034 1158
node attributes 9 138 843 935 974 1034 1158
node attributes 5 80 844 935 974 1034 1158
node attributes 1 36 848 945 984 1034 1158
node attributes 5 80 838 935 974 1034 1158
node attributes 5 80 839 935 974 1034 1158
node attributes 5 80 840 935 974 1034 1158
node attributes 5 80 841 935 974 1034 1158
node attributes 5 80 842 935 974 1034 1158

edge 1 2
edge 2 3
edge 3 4
edge 4 5
edge 5 6
edge 6 7
edge 7 8
edge 8 9
edge 9 10
edge 10 11
[...]
edge 2838 2839
edge 2839 2840
edge 2840 2715
edge 1476 2841
edge 2809 2842
edge 2842 2843
edge 2843 2844
edge 2844 2845
edge 2845 2846
edge 2846 2805

edges 0 1 2 3 4 5 6 7 8 9 [...] 3164 3016 3084 3165 3166 3167 3168 3169 3170 3171
```

TABLE of CRITICAL VALUES for STUDENT'S *t* DISTRIBUTIONS

Column headings denote probabilities (α) ***above*** tabulated values.

d.f.	0.40	0.25	0.10	0.05	0.04	0.025	0.02	0.01	0.005	0.0025	0.001	0.0005
1	0.325	1.000	3.078	6.314	7.916	12.706	15.894	31.821	63.656	127.321	318.289	636.578
2	0.289	0.816	1.886	2.920	3.320	4.303	4.849	6.965	9.925	14.089	22.328	31.600
3	0.277	0.765	1.638	2.353	2.605	3.182	3.482	4.541	5.841	7.453	10.214	12.924
4	0.271	0.741	1.533	2.132	2.333	2.776	2.999	3.747	4.604	5.598	7.173	8.610
5	0.267	0.727	1.476	2.015	2.191	2.571	2.757	3.365	4.032	4.773	5.894	6.869
6	0.265	0.718	1.440	1.943	2.104	2.447	2.612	3.143	3.707	4.317	5.208	5.959
7	0.263	0.711	1.415	1.895	2.046	2.365	2.517	2.998	3.499	4.029	4.785	5.408
8	0.262	0.706	1.397	1.860	2.004	2.306	2.449	2.896	3.355	3.833	4.501	5.041
9	0.261	0.703	1.383	1.833	1.973	2.262	2.398	2.821	3.250	3.690	4.297	4.781
10	0.260	0.700	1.372	1.812	1.948	2.228	2.359	2.764	3.169	3.581	4.144	4.587
11	0.260	0.697	1.363	1.796	1.928	2.201	2.328	2.718	3.106	3.497	4.025	4.437
12	0.259	0.695	1.356	1.782	1.912	2.179	2.303	2.681	3.055	3.428	3.930	4.318
13	0.259	0.694	1.350	1.771	1.899	2.160	2.282	2.650	3.012	3.372	3.852	4.221
14	0.258	0.692	1.345	1.761	1.887	2.145	2.264	2.624	2.977	3.326	3.787	4.140
15	0.258	0.691	1.341	1.753	1.878	2.131	2.249	2.602	2.947	3.286	3.733	4.073
16	0.258	0.690	1.337	1.746	1.869	2.120	2.235	2.583	2.921	3.252	3.686	4.015
17	0.257	0.689	1.333	1.740	1.862	2.110	2.224	2.567	2.898	3.222	3.646	3.965
18	0.257	0.688	1.330	1.734	1.855	2.101	2.214	2.552	2.878	3.197	3.610	3.922
19	0.257	0.688	1.328	1.729	1.850	2.093	2.205	2.539	2.861	3.174	3.579	3.883
20	0.257	0.687	1.325	1.725	1.844	2.086	2.197	2.528	2.845	3.153	3.552	3.850
21	0.257	0.686	1.323	1.721	1.840	2.080	2.189	2.518	2.831	3.135	3.527	3.819
22	0.256	0.686	1.321	1.717	1.835	2.074	2.183	2.508	2.819	3.119	3.505	3.792
23	0.256	0.685	1.319	1.714	1.832	2.069	2.177	2.500	2.807	3.104	3.485	3.768
24	0.256	0.685	1.318	1.711	1.828	2.064	2.172	2.492	2.797	3.091	3.467	3.745
25	0.256	0.684	1.316	1.708	1.825	2.060	2.167	2.485	2.787	3.078	3.450	3.725
26	0.256	0.684	1.315	1.706	1.822	2.056	2.162	2.479	2.779	3.067	3.435	3.707
27	0.256	0.684	1.314	1.703	1.819	2.052	2.158	2.473	2.771	3.057	3.421	3.689
28	0.256	0.683	1.313	1.701	1.817	2.048	2.154	2.467	2.763	3.047	3.408	3.674
29	0.256	0.683	1.311	1.699	1.814	2.045	2.150	2.462	2.756	3.038	3.396	3.660
30	0.256	0.683	1.310	1.697	1.812	2.042	2.147	2.457	2.750	3.030	3.385	3.646
31	0.256	0.682	1.309	1.696	1.810	2.040	2.144	2.453	2.744	3.022	3.375	3.633
32	0.255	0.682	1.309	1.694	1.808	2.037	2.141	2.449	2.738	3.015	3.365	3.622
33	0.255	0.682	1.308	1.692	1.806	2.035	2.138	2.445	2.733	3.008	3.356	3.611
34	0.255	0.682	1.307	1.691	1.805	2.032	2.136	2.441	2.728	3.002	3.348	3.601
35	0.255	0.682	1.306	1.690	1.803	2.030	2.133	2.438	2.724	2.996	3.340	3.591
36	0.255	0.681	1.306	1.688	1.802	2.028	2.131	2.434	2.719	2.990	3.333	3.582
37	0.255	0.681	1.305	1.687	1.800	2.026	2.129	2.431	2.715	2.985	3.326	3.574
38	0.255	0.681	1.304	1.686	1.799	2.024	2.127	2.429	2.712	2.980	3.319	3.566
39	0.255	0.681	1.304	1.685	1.798	2.023	2.125	2.426	2.708	2.976	3.313	3.558
40	0.255	0.681	1.303	1.684	1.796	2.021	2.123	2.423	2.704	2.971	3.307	3.551
60	0.254	0.679	1.296	1.671	1.781	2.000	2.099	2.390	2.660	2.915	3.232	3.460
80	0.254	0.678	1.292	1.664	1.773	1.990	2.088	2.374	2.639	2.887	3.195	3.416
100	0.254	0.677	1.290	1.660	1.769	1.984	2.081	2.364	2.626	2.871	3.174	3.390
120	0.254	0.677	1.289	1.658	1.766	1.980	2.076	2.358	2.617	2.860	3.160	3.373
140	0.254	0.676	1.288	1.656	1.763	1.977	2.073	2.353	2.611	2.852	3.149	3.361
160	0.254	0.676	1.287	1.654	1.762	1.975	2.071	2.350	2.607	2.847	3.142	3.352
180	0.254	0.676	1.286	1.653	1.761	1.973	2.069	2.347	2.603	2.842	3.136	3.345
200	0.254	0.676	1.286	1.653	1.760	1.972	2.067	2.345	2.601	2.838	3.131	3.340
250	0.254	0.675	1.285	1.651	1.758	1.969	2.065	2.341	2.596	2.832	3.123	3.330
inf	0.253	0.674	1.282	1.645	1.751	1.960	2.054	2.326	2.576	2.807	3.090	3.290

Tabelle 1: Tabelle der t-Verteilung.[6]

[6] *Table of Critical Values for Student's t Distributions*, `http://www.math.uni-konstanz.de/~buerkel/Student-t-Tabelle.pdf`, Uni Konstanz, Zugriff: 11.01.2020