HOCHSCHULE
MITTWEIDA
University of Applied Sciences

MASTERARBEIT

Herr
Kai Philipp Müller, B.Sc.

Hindernisse und Vorteile der Quellcode-Modernisierung einer 3D-Engine

Modernisierung und Messung am Beispiel der Object-Oriented Graphics Rendering Engine

Mittweida, August 2022

Fakultät **Angewandte Computer- und Biowissenschaften**

MASTERARBEIT

Hindernisse und Vorteile der Quellcode-Modernisierung einer 3D-Engine

Modernisierung und Messung am Beispiel der Object-Oriented Graphics Rendering Engine

Autor:
Kai Philipp Müller

Studiengang:
Medieninformatik und Interaktives Entertainment

Seminargruppe:
MI19w1-M

Erstprüfer:
Prof. Dr. rer. nat. habil. Thomas Haenselmann

Zweitprüfer:
Dr. rer. nat. Rico Beier-Grunwald

Einreichung:
Mittweida, 17.08.2022

Verteidigung/Bewertung:
Mittweida, 2022

Faculty of **Applied Computer Sciences and Biosciences**

MASTER THESIS

Obstacles and Benefits of Modernizing the Source Code of a 3D-Engine

Modernization and Measuring using the Example of the Object-Oriented Graphics Rendering Engine

Author:
Kai Philipp Müller

Course of Study:
Computer Science of Media and Interactive Entertainment

Seminar Group:
MI19w1-M

First Examiner:
Prof. Dr. rer. nat. habil. Thomas Haenselmann

Second Examiner:
Dr. rer. nat. Rico Beier-Grunwald

Submission:
Mittweida, 17.08.2022

Defense/Evaluation:
Mittweida, 2022

Bibliografische Beschreibung:

Müller, Kai Philipp:
Hindernisse und Vorteile der Quellcode-Modernisierung einer 3D-Engine, *Modernisierung und Messung am Beispiel der Object-Oriented Graphics Rendering Engine*. – 2022. – 75 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Masterarbeit, 2022.

Referat:

Neue Versionen einer Programmiersprache eröffnen neue Möglichkeiten, komplexe Zusammenhänge auszudrücken. So ermöglichte auch C++20 in dem dieser Arbeit vorausgegangenen Forschungsmodul eine Alternative zu Vererbung mit virtuellen Funktionen, welche sich in Microbenchmarks als performanter erwies. Eine Messung in einem vollwertigen Software-Projekt erfordert jedoch zunächst eine Modernisierung dessen. So ist u.a. die 3D-Engine OGRE lediglich auf dem Stand von C++11. Es stellt sich die Frage, ob der Arbeitsaufwand und die Risiken, die mit einer Modernisierung und anschließender Umsetzung der Alternative verbunden sind, letztlich zu rechtfertigen wären. Zumindest für den ersten Schritt kann dies auch unabhängig vom zweiten bestimmt werden. Detaillierte Beschreibungen und Microbenchmarks zu neuen Features können zwar oft gefunden werden, jedoch sind die exakten Hindernisse im konkreten Fall schwer einzuschätzen und die tatsächlichen Vorteile für ein vollwertiges Software-Projekt sind aus bloßen Microbenchmarks nicht direkt abzuleiten. Die vorliegende Arbeit beschreibt die mit der Umsetzung neuer Features verbundenen Hindernisse in der 3D-Engine OGRE. Anhand inkrementeller Messungen wird schließlich entschieden, welche Modernisierungen an und für sich lohnenswert sind und von welchen abzuraten ist.

Abstract:

New versions of a programming language enable new possibilities to express complex issues. During the research module preceding this document, C++20 enabled an alternative to inheritance with virtual functions, which proofed to outperform the latter in microbenchmarks. To measure the alternative within a fully fledged software project however, its modernization is required first and foremost. Among others, the 3D-Engine OGRE is only making use of C++11. This begs the question, whether the effort and risks tied to a modernization with subsequent implementation of the aforementioned alternative would pay off in the end. At the very least, the first step could be evaluated independently from the second. Detailed descriptions and microbenchmarks of new features may often times exist, yet it remains difficult to accurately gauge the obstacles in a concrete case and it's not possible to derive the effective benefits in a fully fledged software project from microbenchmarks alone. The document at hand describes the obstacles that arose while implementing new features into the 3D-Engine OGRE. Based on incremental measurements it will then be determined which modernizations are worthwhile and which should be advised against.

Inhaltsverzeichnis

Abbildungs- und Tabellenverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Die Programmiersprache C++, welche 1983 veröffentlicht wurde, befindet sich in stetiger Weiterentwicklung. Im Dezember 2020 wurde mit ISO/IEC 14882:2020 der Standard C++20 abgeschlossen[53], womit nun bereits an C++23 gearbeitet wird[63]. Neue Versionen bringen neue Möglichkeiten in der Sprache Syntax, Plattform-Unabhängigkeit, Ausführungsgeschwindigkeit, Speicherverbrauch, Sicherheit und Kompilationsdauer eines Programms zu verbessern.

Im Rahmen des *Forschungsmoduls*, welches Teil des Masterstudiengangs *Medieninformatik und interaktives Entertainment* an der Hochschule Mittweida ist, wurde mit Hilfe einiger dieser neuen Möglichkeiten die *Archetypische Rekomposition* (ATR) entworfen[102]. Dies ist eine Technik welche alternativ zu Vererbung mit virtuellen Funktionen angewendet werden kann. Im Vergleich mit weiteren Alternativen konnte sich ATR in den Kategorien Performance und Speicherverbrauch zusammen mit einer von Hand optimierten Variante von allen anderen absetzen. In der Tat generierte ATR trotz hohem Abstraktionsniveau ein Assembly, welches bis auf Symbol-Namen identisch zu dem der von Hand optimierten Variante war. In den jeweiligen Demo-Programmen konnte dieselbe Aufgabe in beiden Fällen doppelt so schnell und mit 20% weniger Arbeitsspeicher gelöst werden als bei der Variante mit virtuellen Funktionen. Das hohe Abstraktionsniveau von ATR führte allerdings in der ursprünglichen Version zum Verzehnfachen der Kompilationsdauer. Weitere Kritikpunkte umfassten die neue Syntax von ATR sowie die resultierenden Symbol-Namen in einem Debugger. Letztlich handelte es sich bei den Demo-Programmen um kaum mehr als Microbenchmarks. Bei diesen sind die Ergebnisse nicht ohne Weiteres auf vollwertige Programme übertragbar[2, S. 90]. Insofern steht offen, wie groß die Vorteile bei einer Echtzeit-Anwendung ausfallen würden.

Nun ist selbst für eine testweise Integration von ATR in eine Echtzeit-Anwendung jedoch erforderlich, dass sie sich des aktuellen C++ Standards bedient. Vorausgesetzte Features der Sprache wären sonst nämlich nicht verfügbar. Zum Beginn dieser Arbeit konnte jedoch keine entsprechend geeignete Anwendung gefunden werden, obwohl dies technisch bereits möglich gewesen wäre. Selbst wenn ATR in der Theorie für deutliche Verbesserungen in Echtzeit-Anwendungen sorgen würde, könnte dies ohne die Grundlage des aktuellen C++ Standards nicht genutzt werden. Für eine solche Anwendung besteht also zu allererst die Notwendigkeit einer Modernisierung des Quellcodes auf den neuesten Standard.

> *„The purpose of modernizing code is to simplify adding new functionality, to ease maintenance, and to increase performance[...]. There are risks implied by every change and costs [...] implied by having an outdated code base. The cost reductions must outweigh the risks.“*[130, Appendix B]

Der damit verbundene Arbeitsaufwand ist jedoch unbekannt und das Risiko besteht, dass die letztendlichen Vorteile von ATR in der finalen Anwendung zu gering ausfallen. Es gilt folglich zu klären, mit welchen Hindernissen bei der Modernisierung gerechnet werden muss. Dass eine Modernisierung neue Möglichkeiten eröffnet steht außer Frage. Wie bedeutend diese jedoch sind und ob sie den Arbeitsaufwand rechtfertigen können, soll in dieser Arbeit behandelt werden.

1.2 Zielsetzung

Das primäre Ziel der vorliegenden Arbeit ist es zu ermitteln, wie sich verschiedene C++ Modernisierungen in eine Echtzeit-Anwendung umsetzen lassen. Damit wird auch eine Grundlage geschaffen, die in weiterer Arbeit eine Integration und Messung von ATR ermöglicht. Zudem wird in dieser Arbeit ein Referenzpunkt für ATR geschaffen in Bezug auf Hindernisse und Vorteile der Umsetzung. Voraussetzung für jegliche Modernisierung ist die Auswahl einer bereits existierende Echtzeit-Anwendung. Sie sollte dabei auf Vererbung mit virtuellen Funktionen aufbauen, da ebendies in späterer Arbeit durch ATR zu ersetzen ist. Der dieser Arbeit zugrundeliegende Studiengang *Medieninformatik und interaktives Entertainment* legt die Verwendung einer 3D-Engine nahe.

In dieser 3D-Engine sind verschiedene Modernisierungen durchzuführen, deren Risiken durch die positiven Auswirkungen auf Syntax, Plattform-Unabhängigkeit, Ausführungsgeschwindigkeit, Speicherverbrauch, Sicherheit und/oder Kompilationsdauer übertrumpft werden sollen. Die Modernisierung einer kompletten 3D-Engine ist jedoch kein triviales Unterfangen, jede einzelne Code-Zeile auf potentielle Verbesserungen zu untersuchen ist unpraktikabel. Automatisierte Umwandlungen, Text-Ersetzungen basierend auf regulären Ausdrücken sowie stichprobenartige Änderungen versprechen schnell große Teile des Quellcodes auf den aktuellen Standard zu heben. Die während des Prozesses entstehenden Hindernisse sind dabei zu verallgemeinern, sodass sie beispielhaft für die Modernisierung anderer Programme zu Rate gezogen werden können. Die gemessenen Unterschiede wiederum helfen bei der Entscheidung, welche Modernisierungen detailliertere Umsetzungen rechtfertigen.

Es erfolgt zunächst eine Einschränkung des zu modernisierenden Quellcodes auf eine minimale Untermenge. Dadurch wird erreicht, dass sich wiederholende Modernisierungen nur in absolut notwendiger Anzahl durchgeführt werden müssen. Der Quellcode soll schließlich zu einer lauffähigen Anwendung kompiliert werden, welche mit Hilfe der 3D-Engine eine sich ändernde dreidimensionale Welt darstellt. Weggelassen werden dabei explizit:

- Komponenten der 3D-Engine, welche nicht in der Anwendung benötigt werden.
- Kompatibilität mit mehr als einem Betriebssystem.
- Kompatibilität mit mehr als einem Compiler.

Mit dieser minimalen Untermenge soll darauf die Modernisierung erfolgen. Sie umfasst die Kompatibilität mit den neuesten Features bis einschließlich C++23 sowie deren Integration.

1.3 Auswahl der 3D-Engine

Für die notwendige 3D-Engine fiel die Wahl auf die *Object-Oriented Graphics Rendering Engine* (OGRE). Mit *Object-Oriented* impliziert sie bereits im Namen die Verwendung von Vererbung und virtuellen Funktionen. Darüber hinaus sprechen noch folgende Kriterien für OGRE:

1. **Freizügige Lizenz:** OGRE ist unter der *MIT-License* veröffentlicht. Dadurch wird der gesamte Quellcode öffentlich bereitgestellt und darf beliebig verändert werden. Viele andere 3D-Engines bedienen sich restriktiverer Lizenzen oder sind gar nicht öffentlich zugänglich.
2. **Größtenteils C++:** OGRE ist, bis auf wenige Ausnahmen in C, vollständig in C++ geschrieben. Andere 3D-Engines verwenden oft Script-Sprachen wie *Lua* oder bestehen zu größeren Teilen aus C. Beides verringert den potenziellen Anteil von modernisierbarem C++ Code.

3. **Verwendung von CMake:** OGRE verwendet CMake als build script generator. Zum Zeitpunkt des Schreibens haben nur wenige build script Generatoren Unterstützung für die notwendigen C++20 Module, auch nicht CMake[4, S. 644]. Jedoch konnte in CMake bereits vor dieser Arbeit eine Übergangslösung gefunden werden.
4. **Aktiv angewendet:** OGRE dient mehreren bereits veröffentlichten Spielen als Grundlage[117, D. README]. Dadurch sind Messungen in OGRE auf reale Anwendungen übertragbar und nicht nur experimenteller Natur.
5. **Integrierte Beispiele:** OGRE kommt mit mehreren Beispiel-Anwendungen (Samples), welche die Funktionalität demonstrieren. Eine davon kann für die Messungen herangezogen werden, ohne dass zusätzlicher Aufwand entsteht.

1.4 Wissenschafliche Einordnung

Obwohl C++20 nun schon vor über einem Jahr fertiggestellt wurde, können laut einer Umfrage auf isocpp.org gerade mal 22,85% der Entwickler in 2022 dies in Gänze in ihrer Arbeit anwenden, 50,40% sogar noch überhaupt nicht[54, § Q11]. Mit einer Umfragedauer von einer Woche ist diese dabei weniger repräsentativ für den durchschnittlichen C++ Entwickler als für C++ Enthusiasten, welche sich zumindest wöchentlich auf dem aktuellen Stand halten. Insofern liegt die tatsächliche Prozentzahl der Entwickler, welche die neuesten Features verwenden, eher noch darunter. Selbst C++11, ein Standard welcher schon seit über 10 Jahren existiert, wird noch nicht universell eingesetzt. In seinem Konferenz-Vortrag *„Why you should move your legacy code to smart pointers“* nennt Sébastien Gonzalve drei Befürchtungen, warum es länger dauert bis ein neuer Standard von Entwicklern akzeptiert wird:

1. **Verluste oder Verwirrung:** Wenn der Code bereits wie gewünscht funktioniert kann jede Änderung dazu führen, dass Funktionalitäten verloren gehen. Auch wird nicht jeder Entwickler mit den neuen Features vertraut sein. Jedoch ist fraglich, ob der alte Code wirklich 100% korrekt ist.
2. **Zusätzliche Arbeit:** Eine Modernisierung beansprucht Arbeitszeit, die eigentlich für die Entwicklung neuer Software-Komponenten geplant ist. Es wird angenommen, dass sich der Aufwand nicht lohnt. Eine Modernisierung kann Code allerdings so vereinfachen, dass mehrere Zeilen ohne Weiteres einfach gelöscht werden können. Dies reduziert den Wartungsaufwand.
3. **Versagen:** Die Modernisierung an einer Stelle kann zunächst zu Compiler-Fehlern an vielen anderen Stellen führen, welche ohne das richtige Wissen schwierig zu beheben sind. Dieses neue Wissen kann man sich jedoch aneignen.

Es folgen Modernisierungs-Beispiele, die sich mit *Smart Pointern* befassen. Dabei wird aufgezeigt, dass diese für mehr Sicherheit sorgen und dass entstehende Fehlermeldungen sogar nützlich sein können[40, M. 4:30ff].

Um die Korrektheit eines Programms vor und nach einer Modernisierung zu testen gibt es neben Frameworks wie *GoogleTest*[15] auch unterstützende Tools wie der *AddressSanitizer* von Clang[73]. Um den Arbeitsaufwand und die Risiken zu minimieren, kann der Modernisierungs-Prozess in Teilen automatisiert werden. Hierfür können existierende Programme wie Clang-Tidy dienen.[1] Schließlich kann das Wissen über moderne Features beispielsweise über Lehrbücher angeeignet werden.

[1] In *Large-scale semi-automated migration of legacy C/C++ test code* wurde für die Modernisierung eigens ein Meta-Programm geschrieben. Dabei geht es allerdings um die Migration zu einer moderneren Bibliothek statt um die Nutzung modernerer Features in existierendem Quellcode[120].

Lehrbücher wie *Beginning C++20*[52], *Modern C++ for Absolute Beginners*[29], *Exploring C++20*, *FORSCHUNG mit modernem C++*[41] oder *Expert C++*[46] dienen als Einführung bzw. Vertiefung in die Programmierung mit C++ allgemein, inklusive Features aus C++17 bzw. C++20. *Clean C++20*[118] oder *C++ High Performance*[2] gehen auf die Anwendung moderner Features in Bezug auf Design- oder Performance-Prinzipien ein. *Effective Modern C++*[99] legt einen Schwerpunkt auf die Modernisierung existierenden Codes, ist dabei jedoch auf C++14 beschränkt. Das gleiche gilt für *Embracing Modern C++ Safely*[66]. Darin werden neue Features aus C++11 und C++14 im Detail behandelt. Sie werden dabei in drei Kategorien eingeteilt: Sichere, bedingt sichere und unsichere Features. Dazu werden ihre jeweiligen Anwendungsfälle beschrieben und Tücken aufgezeigt. Stellenweise sind die Vorteile mit Messungen aus Microbenchmarks belegt. Der Fokus liegt eher auf dem Schreiben neuen Quellcodes mit modernen Techniken als die Modernisierung existierenden Quellcodes.

Trotz Erscheinungsdatum in 2020 sind einige Werke zu C++20 bereits veraltet. So wird in *C++: Das umfassende Handbuch* behauptet, dass mit C++20 `char const[]` als Template-Parameter verwendet werden kann, wodurch String-Literale in diesem Kontext wie Zahlen verwendet werden können[149, S. 570]. Korrekt ist lediglich, dass seit C++20 String-Literale ein Template-Argument sein können, sofern der entsprechende Template-Parameter ein Klassen-Typ ist, welcher implizit aus einem String-Literal konstruierbar ist. Parameter des Typs `char const[]` verfallen zu `char const*` [126, § 3.2, § 6.2]. *C++20 Recipes* nimmt Bezug auf C++20 auf dem Stand von Ende 2019. So sind die Vergleichs-Kategorien *strong equality* und *weak equality* aufgeführt[12, S. 143f], welche jedoch wieder aus C++20 entfernt wurden[115]. Die zuverlässigste Quelle für die neuesten Features sind insofern der aktuelle C++ Standard selbst[63], sowie diejenigen Artikel über Vorschläge, die in ersteren aufgenommen sind. Diese erklären dazu auch oft die Motivation hinter den neuen Features sowie Design-Entscheidungen, Umsetzbarkeit und genaue Formulierung.

Über die meisten Modernisierungen lässt sich mit den genannten Werken schnell ein Überblick verschaffen, welcher ausreicht, sie in neuen Software-Komponenten anzuwenden. Es wird jedoch nicht aufgeführt, welche Hindernisse der Modernisierung existierenden Quellcodes im Wege stehen. Auch bleibt offen, wie hoch der Preis in Bezug auf Kompilationsdauer und Ausführungsgeschwindigkeit in einem vollwertigen Projekt ist. Es gibt zudem Änderungen, die zwar Performance und Speicher-Vorteile versprechen, wie groß diese jedoch im konkreten Fall ausfallen ist unklar. Dieser Frage will sich die vorliegende Arbeit widmen. Der Schwerpunkt soll somit nicht auf den Feinheiten der Features sondern auf deren Integration am Beispiel von OGRE und inkrementellen Messungen der Änderungen liegen. Entsprechende Features werden dennoch zusammenfassend erklärt. Dies schließt natürlicherweise Features aus, welche bereits integriert sind oder keinen Anwendungsfall in OGRE haben. Dabei vertritt der Autor die Hypothese, dass die Hindernisse und Einbußen vertretbar sind und die Vorteile sowie die eröffneten Möglichkeiten einer Modernisierung klar überwiegen.

2 Ausgangssituation

2.1 Programmversionen

Die Grundlage für die Modernisierung ist eine Fork von OGRE in Version 13.3.3 vom 14. März 2022[117, C. 1ffe04]. Andere Versionen werden in der vorliegenden Arbeit nicht berücksichtigt. Als Betriebssystem für die Entwicklung und alle Messungen diente Kubuntu 22.04. Zum Zeitpunkt des Schreibens gewährt dieses eine gute Systemstabilität und erlaubt Zugriff auf aktuelle Programmversionen. Dazu gehört auch LLVM mit dem Compiler Clang in der Version 14.0.0. Darüberhinaus wurde auch eine Vorabversion von Clang 15 vom 07. Juli 2022 verwendet[68, C. 9a0471].

Version 14 wird dabei in den ersten Schritten wegen seiner Stabilität verwendet, in späteren Schritten ist jedoch Version 15 notwendig geworden. Diese Version unterstützt nämlich erstmals Modul-Partitionen, aber auch eine verbesserte Umsetzung der Header `<ranges>` und `<format>`. Zudem wurde es in dieser Version möglich, den neuen C++20 Header `<source_location>` zu verwenden, auch wenn dieser noch kein offizieller Teil von Clang 15 ist. Für dessen Verwendung wurden in Clang 15 die notwendigen Dateien aus dem noch ausstehenden Update eingefügt[61]. Da die Umstellung der Compiler-Version ebenfalls Teil einer Projekt-Modernisierung ist, sind im Folgenden auch die Unterschiede der beiden Versionen festgehalten. Für diese Arbeit wurde Clang statt dem standardmäßigen GCC gewählt, da verwendete Tools wie *include-what-you-use* und Clang-Tidy ebenfalls auf Clang zugreifen und auftretende Fehler gleicher Natur sind[42, D. README]. Als Standard-Bibliothek dient dabei libc++. Für die spätere Integration von Modulen war dies wichtig, da Clang nur unter der Verwendung von libc++ Header der Standard-Bibliothek als importierbare Header-Units erkennt.

2.2 Inkompatibilitäten

Die Umstellung der Standard-Bibliothek erfordert, dass alle verlinkten Bibliotheken von Drittanbietern ebenfalls mit libc++ kompiliert werden. Sonst kann es zu ABI Inkompatibilitäten kommen, welche folgende Form annehmen:

```
error: undefined symbol:
std::__cxx11::basic_string<char, std::char_traits<char>,
        std::allocator<char> >::reserve(unsigned long)
>>> referenced by gtest-all.cc.o:
```

Die Klasse `string` und ihre Member-Funktionen unterscheiden sich in ihrem vollständigen Symbol-Namen von libc++ zu libstdc++. Dies ist auf eine Änderung an der `string` ABI zurückzuführen, welche mit C++11 eingeführt wurde[91]. Dabei war libc++ nicht betroffen, da diese Bibliothek direkt mit C++11 entstand[77].[2] Wurde also zunächst mit libstdc++ kompiliert, ist ein anderer Symbol-Name in die kompilierte Datei eingebettet. Der Linker schließlich findet die zugehörige Definition nicht, da diese in der anderen Bibliothek einen anderen Namen trägt.

[2] Die Quelle nennt C++0x und nicht C++11. C++0x ist jedoch gleichbedeutend mit C++11, da der Standard ursprünglich vor 2010 erscheinen sollte[131].

2.3 Konfigurationen

Das OGRE-Projekt umfasst mehrere Bibliotheken, von welchen einige sowohl statisch als auch dynamisch geladen werden können. Standardmäßig werden dynamische Bibliotheken bevorzugt, für die Analyse des Programms zur Laufzeit sind statische Bibliotheken jedoch günstiger. Tools wie valgrind oder AddressSanitizer funktionieren besser mit statischen Bibliotheken, weswegen im Folgenden dynamische Bibliotheken vermieden werden. Obwohl OGRE dies offiziell unterstützt, schlägt ein Test bei der Verwendung statischer Bibliotheken fehl:

```
Could not load dynamic library lib/RenderSystem_GL.
System Error: lib/RenderSystem_GL.so.13.3:
cannot open shared object file: No such file or directory in
DynLib::load at OgreMain/src/OgreDynLib.cpp (line 136)"
```

Hier wird versucht, eine dynamische Bibliothek zu laden, welche durch eine statische ersetzt wurde. Deaktiviert man das dynamische Laden in dieser Konfiguration komplett, tritt dieser Fehler nicht mehr auf. Das Auftreten von Fehlern in bestimmten Konfigurationen im noch unveränderten Projekt bestätigt die Entscheidung, zunächst eine Reduktion auf eine geringere Auswahl von Konfigurationen durchzuführen. Somit kann das Augenmerk auf der Modernisierung liegen und die Arbeitszeit muss nicht dem Beheben von existierenden Fehlern eingeräumt werden.

2.4 Tests

Teil des OGRE Projektes sind zwei Test-Programme. Das Erstellen beider Test-Programme muss jedoch erst im Projekt aktiviert werden, bevor sie durchgeführt werden können. Das erste, *Test_Ogre*, überprüft die Kern-Funktionalitäten der Engine mittels *GoogleTest*[15]. Beispiele hierfür sind das Verhalten der Kamera oder das Laden von Ressourcen. Die CMake-Dateien von OGRE erledigen Download und Konfiguration von *GoogleTest* automatisch. GCC 11, der Standard-Compiler von Kubuntu 22.04[140], gibt mit der verwendeten *GoogleTest*-Version 1.10.0 Fehler aus[15, l. 3219]. Von daher wurde an dieser Stelle stattdessen *GoogleTest*-1.11.0 verwendet.[3]

Das zweite Test-Programm nennt sich *TestContext*. Dieses rendert einzelne Bilder verschiedener Scenes und vergleicht diese auf hinreichende Ähnlichkeit mit den Bildern aus dem vorherigen Durchlauf. Damit wird sichergestellt, dass keine gravierenden Unterschiede beim Rendering entstehen. Die Definition von hinreichender Ähnlichkeit für *TestContext* war dabei auf 99,9% festgelegt. Nun kann es durch die Abhängigkeit an die aktuelle Ausführungsgeschwindigkeit des jeweiligen Rechners dazu kommen, dass ohne jegliche Änderungen am Quellcode die gerenderten Bilder aufeinander folgender Test-Durchläufe zu stark voneinander abweichen. Um die Integrität der Testfälle zu bewahren, wurde stattdessen 90% Ähnlichkeit festgelegt.

Ein Testfall schlägt bereits ohne jegliche Veränderung am Projekt mit folgender Meldung fehl:

```
Named constants have not been initialised, perhaps a compile
error in _findNamedConstantDefinition at
OgreMain/src/OgreGpuProgramParams.cpp (line 1391)
```

[3] Die gleiche Änderung wurde am 12. April 2022 auch im OGRE Repository durchgeführt[117, C. 269530d].

Es konnte keine Möglichkeit gefunden werden, dies zu beheben, weswegen dieser Test deaktiviert wurde.

2.5 Dateinamen

Die Namenskonvention für C++ Dateien in OGRE sieht die Endung `.h` für Header-Dateien und `.cpp` für Implementierungs-Dateien vor[117, D. CodingStandards]. Dies ist konform zu den *C++ Core Guidelines*[130, § SF.1]. Im Verlauf dieser Arbeit hat sich jedoch erwiesen, dass die Endung `.h` die Unterscheidung zwischen den zu modernisierenden C++ Headern und den ebenfalls im Projekt existierenden C Headern sowie Shader-Headern, welche unberührt bleiben sollen, verkompliziert. Clang-Tidy, ein Tool welches bei der Modernisierung unterstützend eingesetzt wurde, verfügt über eine Option `header-filter` um den Suchbereich einzuschränken[74]. Um diese Möglichkeit effektiv nutzen zu können wurden die Endung für alle C++ Header in `.hpp` umbenannt.

Für die in Abschnitten 3.2 und 10.3 beschriebenen Tools ist es zudem hilfreich, wenn Header und die zugehörigen Implementierungs-Dateien bis auf die Endung den gleichen Namen haben. Dies vereinfacht ihre Zuordnung. In wenigen Fällen war es also notwendig, bestimmte inkonsistente Namen abzuändern oder Dateien in einen anderen Ordner zu verschieben.

2.6 Memory-Leaks

Mit Hilfe der Option `fsanitize=address` kann Clang ein Programm mit dem sog. *AddressSanitizer* so modifizieren, dass es Fehler in der Speicherverwaltung erkennt und mit einer Fehlermeldung abbricht[73]. Insbesondere werden hiermit Memory-Leaks identifiziert. Für eine möglichst genaue Messung des Speicherverbrauchs ist es notwendig, dass es im Programm zu keinen solchen Memory-Leaks kommt. Auch ist es jenseits von Messungen wichtig, diese zu beheben, denn:

> *„Even a slow growth in resources will, over time, exhaust the availability of those resources. This is particularly important for long-running programs, but is an essential piece of responsible programming behavior.“*[130, § P.8]

Unglücklicherweise kamen durch die Analyse in OGRE mehrere Memory-Leaks zu Tage. An insgesamt neun verschiedenen Stellen konnten diese durch die Verwendung von `unique_ptr` aus C++11 behoben werden. Das zugrundeliegende Problem ist, dass für reservierte Speicherblöcke kein expliziter Besitzer existiert. Der Besitzer ist dabei verantwortlich, dass der Speicherblock wieder freigegeben wird[2, S. 210]. Ohne diesen expliziten Besitzer kann es leicht passieren, dass vergessen wird den Speicher wieder freizugeben, was schließlich zu Memory-Leaks führt. Ein solcher expliziter Besitzer, welcher den Speicher automatisch freigibt, ist `unique_ptr`. Dieses Feature wird in Abschnitt 9.6 näher behandelt. Da das Beheben der Memory-Leaks für eine genaue Messung jedoch essentiell ist, wurden die Unterschiede, welche durch das Beheben der Memory-Leaks entstehen, nicht gemessen. Weitere Speicher-Probleme umfassten das Aufrufen einer inkorrekten Überladung einer Funktion sowie das Fehlen eines notwendigen Aufrufs an die Funktion `SDL_Quit` der Verwendeten Drittanbieter-Bibliothek SDL[67, D. SDL.h]. Auch im verwendeten Grafikkarten-Treiber konnten mit AddressSanitizer mehrere Memory-Leaks entdeckt werden. Da dies jedoch bei jeder Ausführung konstant auf 807 entdeckte Memory-Leaks mit etwas über 15 kByte hinausläuft, können sie für die Messungen außer Acht gelassen werden. Nicht zuletzt auch, weil sie nicht Bestandteil von OGRE sind, wurden sie also nicht behoben.

3 Minimales Subset

3.1 Allgemein

Um den Aufwand der Migration zu minimieren, sowie die Fehler, welche durch Änderungen des Projektes resultieren, auf einen möglichst kleinen Bereich zu beschränken, empfiehlt es sich, nicht unmittelbar notwendige Komponenten aus dem Projekt zu entfernen. Suchmuster, die auf alle Dateien im Projekt angewendet werden, liefern so nur Treffer aus tatsächlich relevanten Dateien zurück. Weiterhin können Änderungen an nicht im Programm verwendeten Dateien auch nicht auf Korrektheit überprüft werden, wodurch das Risiko besteht Fehler einzubauen. Auch werden bei der semantischen Analyse des Quellcodes falsch positive Warnmeldungen bezüglich fehlender Plattform-spezifischer Eigenschaften und Dateien vermieden. Letztlich können Messungen der Kompilationsdauer nicht durch Dateien verfälscht werden, welche nicht mehr existieren.

Dank der Versionsverwaltung mit *git* besteht immer die Möglichkeit, entfernte Komponenten bei Bedarf wiederherzustellen, sofern die Änderungen in separaten Commits durchgeführt wurden. Hierfür reicht der Befehl `git revert <commit>` aus[39, B. git-revert].

3.2 Entfernen nicht benötigter Komponenten

Da lediglich das Sample *New Instancing* betrachtet werden soll, können zunächst alle anderen Samples entfernt werden. Gleiches gilt für alle PlugIns, welche nicht für die Ausführung von *New Instancing* notwendig sind. Interessanterweise führt dies zunächst zu Fehlermeldungen während der Laufzeit, welche folgende Form annehmen:

```
Error: ScriptCompiler - invalid parameters in
emitted_emitter.particle(14): Cannot find emitter type 'Box'
```

Sie entstehen dadurch, dass OGRE prinzipiell alle Ressourcen lädt, welche im jeweilig konfigurierten Verzeichnis zu finden sind, selbst wenn diese ein nicht geladenes PlugIn voraussetzen, wie in diesem Fall *ParticleFX*. Da diese Dateien jedoch ebenfalls nicht für *New Instancing* benötigt werden, können auch sie mitsamt der Referenz auf den nun leeren Ordern in der Ressourcen-Konfiguration problemlos entfernt werden. Für PlugIns, welche von *New Instancing* nicht benötigt werden, können immer noch Tests existieren, die auf diesen aufbauen. Diese gilt es folglich ebenfalls zu entfernen.

Weiterhin liegen dem OGRE-Projekt mehrere Tools bei, welche zum Teil in C++ geschrieben sind. Diese sind für die Entwicklung von Spielen mit OGRE nützlich, beispielsweise um Ressourcen-Dateien unterschiedlicher Formate zu konvertieren. Für *New Instancing* haben diese jedoch keine Relevanz und wurden aus dem Projekt entfernt.

Mehrere Render-Systeme stehen in OGRE zur Auswahl. Dabei wird jeweils nur eines benötigt um die Bilder der Scene zu rendern. Entscheidend für die Auswahl des Render-Systems *OpenGL* für die Weiterentwicklung war die Eigenschaft, mit dem Tool valgrind am besten kompatibel zu sein.

3.3 Entfernen einzelner Dateien

Das Entfernen der Samples und PlugIns hat zur Folge, dass weitere Quellcode-Dateien im Projekt nicht mehr benötigt werden. Ein primitiver Ansatz, diese zu finden, besteht darin, einzelne Dateien zu löschen und zu überprüfen, ob das Programm weiterhin kompiliert. Dies kann bei kleineren Projekten schon ausreichen, bei einem Projekt der Größe von OGRE ist dies jedoch schon nicht mehr effizient. Stattdessen lässt sich der Suchprozess automatisieren. Für jede kompilierte Quellcode-Datei werden zunächst die Dateien ermittelt, von denen sie abhängig ist. Dadurch wird diese Datei nur dann neu kompiliert, wenn sie selbst oder eine ihrer Abhängigkeiten verändert wurde[108, § C/C++]. Beim Erstellen des ausführbaren Programms kann mit dem Build System *Ninja* die Option `-d keepdepfile` angegeben werden. Dadurch werden die Abhängigkeiten jeder kompilierten Quellcode-Datei in je einer Datei gespeichert. Dabei folgen sie diesem Muster:

```
Samples/Browser/CMakeFiles/SampleBrowser.dir/src/main.cpp.o: \
 Samples/Browser/src/main.cpp \
 ...
 /usr/include/time.h \
 Samples/Browser/include/SampleBrowser.h \
 ...
```

Diese Abhängigkeits-Dateien können darauf automatisiert ausgelesen werden um alle notwendigen Header-Dateien im Projekt zu ermitteln. Quellcode-Dateien, für die keine Abhängigkeits-Datei existiert und die auch nicht in den Abhängigkeiten aller kompilierten Dateien auftauchen, werden folglich für das Erstellen des Programms nicht benötigt. Darüber hinaus kann man, ausgehend von einer Datei, diejenigen Dateien bestimmen, von denen sie transitiv abhängt. Dazu sucht man für jeden Header in der Liste der Abhängigkeiten eine zugehörige Implementierung und für jede Implementierung wieder rekursiv die Header und deren Implementierungen. Alle im Projekt befindlichen Dateien, die so nicht gefunden werden, sind Kandidaten, aus dem Projekt entfernt zu werden. Ein Programm, welches mit dieser Methode alle nicht verwendeten Dateien heraussucht, liegt dieser Arbeit bei.

Eine manuelle Überprüfung der Dateien ist dennoch notwendig. So können diese über `extern`-Deklarationen eingebunden sein, wichtige Nebeneffekte in der *statischen Initialisierung* haben oder Implementierungen von Headern können aufgrund eines unkonventionellen Namens nicht gefunden werden. In den meisten Fällen kann davon ausgegangen werden, dass die Implementierungs-Datei den gleichen Namen trägt wie die zugehörigen Header-Datei. Sie befinden sich entweder im selben Ordner oder die Header-Datei befindet sich in einem `include` Ordner und die Implementierungs-Datei in einem `src` Ordner. Darüber hinaus gibt es in der Regel entweder keine oder genau eine Implementierungs-Datei pro Header. In Einzelfällen kommt es in OGRE jedoch vor, dass zusammengehörige Dateien unterschiedliche Namen tragen, sich in unterschiedlichen Ordnern befinden oder dass es mehr als eine Implementierungs-Datei für einen Header gibt.

Die aufgelisteten Abhängigkeiten schließen jedoch transitiv alle eingeschlossenen Header ein, selbst wenn diese nicht zwangsweise notwendig sind. Ein Tool, welches hier unterstützend eingesetzt werden kann, ist *include-what-you-use* (IWYU). Wie sich aus dem Namen bereits ableiten lässt, soll es dabei helfen, dass jede Quellcode-Datei nur diejenigen Header einschließt, welche auch tatsächlich benötigt werden. IWYU beschreibt seinen Hauptzweck folgendermaßen:

> *„When every file includes what it uses, then it is possible to edit any file and remove unused headers, without fear of accidentally breaking the upwards dependencies of that file.“*[42, D. README]

Das Entfernen nicht benötigter Header verkürzt die Kompilationsdauer und minimiert die Abhängigkeiten. Es besteht zudem die Option, Header durch Forwärts-Deklarationen zu ersetzen um die Kompilationsdauer weiter zu reduzieren und noch mehr Abhängigkeiten zu beseitigen. Wie in Abschnitt 10.4 ersichtlich wird, können diese getarnten Abhängigkeiten die Migration zu Modulen erschweren. Insbesondere kann die One Definition Rule verletzt werden. IWYU bietet zwar die Option `no_fwd_decls` an, um Forwärts-Deklarationen zu vermeiden, jedoch ist dies problematisch, wenn sie tatsächlich notwendig sind um zyklische Abhängigkeiten zu umgehen. Ersetzt man diese dann durch die jeweiligen Header-Includes, können sich zwei Header gegenseitig einschließen. Dies führt zu Fehlermeldungen, in denen bemängelt wird, dass ein Typ nicht bekannt ist, obwohl der zugehörige Header eingeschlossen wurde. IWYU hat diese Option standardmäßig deaktiviert[42, D. WhyIWYUIsDifficult] und sie wurde auch bei der Modernisierung nicht verwendet.

Eine Technik zur Verkürzung der Kompilationsdauer, welche in OGRE Anwendung findet und so weit verbreitet ist, dass sie schon durch CMake 3.16 unterstützt wird, sind die sog. präkompilierten Header[14, B. target_precompile_headers]. Diese sind jedoch nicht Teil des C++ Standards und werden in C++20 durch Module obsolet. In der Tat verwendet Clang für präkompilierte Header und Module die gleiche Herangehensweise[79, § Design Philosophy]. Hierbei werden eine oder mehrere Header-Dateien, welche im Projekt oft eingeschlossen werden, vorkompiliert, wodurch ihre Kompilation beim Einschluss in andere Dateien entfällt. Da der Einschluss von präkompilierten Headern dazu führt, dass Abhängigkeiten entstehen, die nicht notwendig sind, wurde die Option in OGRE deaktiviert, nicht zuletzt auch deswegen, da diese in Kapitel 10 durch Module modernisiert werden.

Unterstützung für *include-what-you-use* ist in CMake über spezifische Optionen bereits eingebaut[14, B. LANG_INCLUDE_WHAT_YOU_USE]. Ein verkürztes Beispiel für eine typische Ausgabe von IWYU beim Kompilieren des Programms sieht dabei wie folgt aus:

```
OgreMain/include/OgreArchive.h should add these lines:
#include "OgreSharedPtr.h"              // for SharedPtr
#include <vector>                       // for vector
namespace Ogre { class Archive; }

OgreMain/include/OgreArchive.h should remove these lines:
- #include "OgreDataStream.h"  // lines 32-32
```

Solche Ausgaben können dann durch `fix_includes.py` dazu verwendeten werden, notwendige Header einzufügen und gegebenenfalls Forwärts-Deklarationen durch Header zu ersetzen. Nur mit der Option `nosafe_headers` werden auch nicht benötigte Header entfernt. IWYU ist dabei nicht perfekt[42, D. WhyIWYUIsDifficult]. Es können mehrere Durchläufe sowie manuelle Korrekturen notwendig sein. Besondere Probleme stellen der bedingte Einschluss von Header sowie die Verwendung von Macros dar. Eine Möglichkeit, dies zu umgehen, ist die Auflösung dieser vor der Verwendung von IWYU, was in Abschnitt 9.2 beschrieben wird.

4 Testaufbau

4.1 Test-Scene

Der Name des Samples *New Instancing* kommt von seiner Eigenschaft, viele identische 3D-Modelle in je 50 Reihen und Spalten zu instanziieren und in eine Scene zu setzen. Ihre Orientierung und Haltung sind dabei pseudo-zufällig mit Hilfe von `mt19937` generiert. Auf diese Weise wird gewährleistet, dass die Startbedingungen bei jedem Programmdurchlauf identisch sind. Es beseht zudem die Möglichkeit, die Anzahl der Reihen und Spalten nachträglich zu erhöhen und die 3D-Modelle zu animieren oder zu bewegen. Um konsistente Testergebnisse zu erzielen, wird der Startwert für die Anzahl der Spalten und Reihen direkt auf das Maximum 100 gesetzt und Animationen und Bewegungen gleich zu Programmstart aktiviert. Es befinden sich also 10'000 animierte und bewegte 3D-Modelle in einer Scene. Somit soll erreicht werden, dass die CPU, auf welcher der C++ Code ausgeführt wird, der die Modelle einzeln animiert, den Flaschenhals des Programms darstellt und nicht etwa die GPU. Die Scene wird im Vollbild-Modus auf 1920 mal 1080 Pixeln gerendert. Die Benutzeroberfläche, welche zum nachträglichen justieren der Szene diente, wurde zudem entfernt. Eine Aufnahme der Scene ist in Abbildung 4.1 zu sehen.

Abbildung 4.1: Die gerenderte Scene aus NewInstancing

4.2 Kompilationsdauer

In der Entwicklung jedes C++ Programms ist es für Testzwecke notwendig, dieses immer wieder neu zu kompilieren. Entsprechend wichtig ist es, dass die Kompilationsdauer kurz ausfällt. Kapitel 3 ging bereits auf Forwärts-Deklarationen und präkompilierte Header und ihre positive Wirkung auf die Kompilationsdauer ein. Insbesondere die mit C++20 eingeführten Module versprechen, dass modernisierte Programme schneller kompilieren. Allerdings kann auch jede andere Modernisierung

dazu führen, dass sich die Kompilationsdauer verändert. Sei es, dass der Compiler neue Programm-Strukturen effizienter parsen kann oder dass neue Abstraktionen eingeführt werden, deren Auflösung den Compiler länger beschäftigen. Aus diesem Grund wird im Folgenden für jede Modernisierung ihre Auswirkungen auf die Kompilationsdauer festgehalten. Ein Auswertungs-Skript misst die durchschnittliche Kompilationsdauer über jeweils fünf Durchläufe mittels des Linux-Kommandos *time*. Die Messung findet für die Konfigurationen Debug und Release statt. Zum Zeitpunkt vor jeder Modernisierung wurden dabei folgende Zeiten gemessen:

Konfiguration	Debug	Release
Mit präkompilierten Headern	39,4 s	35,2 s
Ohne präkompilierte Header	62,2 s	56,9 s

Tabelle 4.1: Initiale Kompilationsdauer in Debug und Release Konfigurationen

4.3 Berechnungsdauer pro Frame

In OGRE ist eine optionale Möglichkeit zum Profiling eingebaut. Wird diese zur Konfigurationszeit aktiviert, misst OGRE die Zeit für einzelne Berechnungsschritte während des Renderns eines Frames. Entscheidend und übertragbar auf jede andere Engine ist dabei die gesamte Berechnungsdauer für einen einzelnen Frame, die Frame Time. Darum wird im Folgenden der Fokus auf diese Zeit beschränkt. Es werden über die gesamte Programmausführung die Metriken minimale Frame Time (Min), maximale Frame Time (Max) und durchschnittliche Frame Time (Avg) berechnet und in Millisekunden ausgegeben. Dem wurde nachträglich auch die Standard-Abweichung (StdDev) hinzugefügt.

OGRE ermittelt die Frame Time mit Hilfe der C Funktion `clock`. Diese gibt die vom Programm verwendente Prozessor-Zeit zurück, wodurch Einflüsse durch parallel laufende Programme minimiert werden. Um vom Rückgabe-Typ `clock_t` auf Sekunden zu kommen, gibt es zudem das Macro `CLOCKS_PER_SEC`[88, § 7.27.2.1]. OGRE rechnete hier die `clock_t` Werte in Millisekunden des Typs `float` um. Beim Beenden des Programms erfolgte dann mit kumulativer Aufrechnung der Frame Times die Berechnung des Durchschnitts. Nicht nur hat der dabei verwendete Typ eine geringe Präzision, beim Errechnen des Durchschnitts – und nun auch der Standard-Abweichung mit kumulativen Quadraten der Frame Times – mit Gleitkomma-Zahlen kann es zu zusätzlicher Ungenauigkeit kommen:

> *„Many (or all) of the correctly compute digits will cancel, leaving a computed S with a possibly unacceptable relative error.“* [13, § 1.2]

Die einfachste Methode, dieses Problem zu umgehen besteht darin, ganze Zahlen – also `clock_t` – statt Gleitkomma-Zahlen zu verwenden und erst bei der letzten Division durch die Anzahl der Frames einen hinreichend großen Gleitkomma-Typ (`long double`) zu nehmen.

Für die Messung der Laufzeit-Performance soll in jeder Konfiguration die gleiche Arbeit erledigt werden um den bestmöglichen Vergleich zu erzielen. *New Instancing* wird jeweils so lange ausgeführt, bis eine konstante Anzahl Frames gerendert wurde. Für ein Konfidenzniveau von 99% und einer Fehlerspanne von 5% wurde diese Zahl dabei auf 666 festgelegt. Dadurch wurde eine Balance zwischen Test-Genauigkeit und Test-Dauer getroffen.

Die 3D-Modelle bewegen sich mit konstanter Geschwindigkeit pro Sekunde. Dies entspricht dem regulären Verhalten innerhalb einer Simulation und führt dazu, dass die Frame Time keinen Einfluss auf die Bewegungsgeschwindigkeit der 3D-Modelle hat. Damit die Messung der Frame Time mit verschiedenen Konfigurationen vergleichbarer sind, hat sich Folgendes erwiesen: Die Ergebnisse eines Durchlaufs sind stabiler, wenn sich die 3D-Modelle mit konstanter Geschwindigkeit pro Frame bewegen. Das liegt daran, dass die Bewegungen eines 3D-Modells ein dahinterstehendes Modell in anderer Weise verdecken können, was zu anderen Berechnungen und somit zu stärker abweichenden Frame Times führt.

Ein Vergleich über fünf verschiedene Durchläufe, jeweils in der Release-Konfiguration vor modernisierenden Änderungen, ist in Tabelle 4.2 dargestellt. In der ersten Variante liegt der jeweilige Durchschnitt zwischen 30,442 ms und 30,799 ms. Ein Wert weicht also bis zu einer viertel Millisekunde vom Durchschnitt 30,682 ms ab. In der zweiten Variante liegt der jeweilige Durchschnitt zwischen 31,316 ms und 31,381 ms, was etwas weniger als eine zwanzigstel Millisekunde vom Durchschnitt 31,353 ms abweicht. Aus diesem Grund werden Messungen im Folgenden mit konstanter Geschwindigkeit pro Frame mit dem Durchschnitt aus fünf Durchläufen angegeben. Dabei werden nur diejenigen Messungen als relevant erachtet, bei denen die Differenz zur vorherigen Messung eine zehntel Millisekunde übersteigt.

	Konstant pro Sekunde				Konstant pro Frame			
#	Min	Max	Avg	σ	Min	Max	Avg	σ
1	27,322	46,027	30,707	2,317	27,898	46,538	31,381	2,343
2	27,254	45,614	30,774	2,311	27,901	46,552	31,316	2,345
3	27,067	46,657	30,442	2,312	27,790	46,485	31,351	2,367
4	27,263	45,741	30,689	2,323	27,820	46,536	31,345	2,362
5	27,145	45,632	30,799	2,304	27,660	46,859	31,371	2,342
Avg	27,210	45,934	30,682	2,313	27,814	46,594	31,353	2,352
σ	0,102	0,437	0,142	0,007	0,099	0,150	0,025	0,012

Tabelle 4.2: Initiale Berechungsdauer bei konstanter Geschwindigkeit pro Sekunde und pro Frame

Da die durchzuführenden Modernisierungen sich kumulativ auf das Laufzeit-Verhalten auswirken, wird lediglich die Differenz der durchschnittlichen Frame Times zum Stand direkt vor der jeweiligen Modernisierungen angegeben. Dadurch wird der tatsächliche Gewinn ersichtlich und die Reihenfolge der Modernisierungen bleibt größtenteils unerheblich. Um die Ursache der jeweiligen Differenzen zu ermitteln, sind besonders Änderungen am Assembly (siehe Abschnitt 4.4) interessant.

In spezifischen Fällen wird aber auch valgrind mit dem Tool *cachegrind* herangezogen. Dabei werden Speicherzugriffe auf Instruktionen und Daten in unterschiedlichen Cache-Ebenen gemessen. Ausschlaggebend sind dabei vor Allem die Fehlzugriffe auf die letzte Cache-Ebene vor dem Zugriff auf den Arbeitsspeicher[141, § 5.1]. Zugriffe auf den Arbeitsspeicher sind nämlich bedeutend langsamer als Zugriffe auf den Cache-Speicher. Somit führen viele Fehlzugriffe im Cache zu vielen Zugriffen auf den Arbeitsspeicher und so zu einer Verlangsamung des Programms. Aber auch Fehlzugriffe auf anderen Cache-Ebenen haben zur Folge, dass auf die jeweils nächste und langsamere Cache-Ebene zugegriffen werden muss.

4.4 Assembly

Eine Modernisierung ändert den Quellcode. Aber nicht in jedem Fall wird dadurch auch das resultierende Assembly verändert. So kann es vorkommen, dass es sich lediglich um verbesserte Syntax für bereits existierende Features handelt. Dies ist durch identische Assemblies zu belegen. Es gibt auch Modernisierungen, welche das Programm beschleunigen. Hier sind entsprechende Änderungen im Assembly zu ermitteln. Darüber hinaus kann es Modernisierungen geben, welche zwar das Assembly erheblich verändern, jedoch kaum Einfluss auf das Laufzeitverhalten des Programms haben. Diese Fälle sind ebenfalls festzuhalten.

Eine mit Clang kompilierte Objekt-Datei kann mit Hilfe von `llvm-objdump` zurück in Assembly verwandelt werden[76, B. llvm-objdump]. In einem eigens dafür geschriebenen Script geschieht dies mit den Optionen `demangle`, `disassemble`, `no-leading-addr` und `no-show-raw-insn`. Diese vereinfachen das Vergleichen und Zuordnen zum ursprünglichen Quellcode der Instruktionen, sofern die Objekt-Datei mit Debug-Symbolen kompiliert wurde. Darüber hinaus werden alle Adressen in den Assembly-Dateien zu `0x` gekürzt, da eine Verschiebung sonst zu sehr vielen falsch positiven Differenzen führen würde. Die dabei entstehenden 278 Dateien mit Assembly-Instruktionen werden pro Konfiguration in einem Verzeichnis abgelegt. Daraufhin können alle Dateien auf Unterschiede untersucht werden. In folgenden Kapiteln wird dabei jedoch nicht auf jede Differenz eingegangen, da dies zu umfangreich wäre. Auch werden lediglich die Unterschiede der Release-Konfiguration betrachtet, da sich die Assemblies der Debug-Konfiguration, welche bedeutend weniger Optimierungen durchführt, erheblich unterscheiden.

4.5 Speicherverbrauch

Eine genaue Messung des Speicherverbrauchs ist in OGRE nicht integriert. Hier bot sich zunächst die Verwendung des externen Tools *Memcheck* von valgrind an. Dieses erlaubt die Ausführung eines Programms auf einem synthetischen Prozessor. Dadurch verlangsamt sich die Geschwindigkeit zwar um einen Faktor von 10 bis 50, dafür kann unter anderem jedoch der komplette Speicherverbrauch festgehalten und bei Programmende ausgegeben werden[141, § 2.1]. Die Verwendung von valgrind lieferte jedoch selbst nach Anpassungen am Programm immer wieder Abweichungen, welche zwischen ein paar Bytes und mehreren Megabytes betrugen. Aus diesem Grund wurde diese Herangehensweise nicht weiter verfolgt.

Stattdessen können mithilfe der globalen Operatoren `new` und `delete` Speicherreservierungen für diagnostische Zwecke abgefangen werden[12, S. 332ff]. Die notwendige Anzahl Bytes wird dann in einer Summe festgehalten. Der Speicher, welcher durch die C Funktionen `malloc` und `free` verwaltet wird, wird dabei nicht mitgerechnet. Da diese in verwendeten Drittanbieter-Bibliotheken aufgerufen werden, ergibt es durchaus Sinn, sie für die Evaluation von OGRE außer Acht zu lassen.

Für einen aussagekräftigen Vergleich des Speicherverbrauchs des Programms jeweils vor und nach einer Modernisierung ist es notwendig, dass Programmdurchläufe desselben Stands näherungsweise den gleichen Speicher verbrauchen. Nun wird jedoch pro Frame in OGRE mehrmals dynamisch Speicher reserviert. Das heißt, dass für vergleichbare Werte immer dieselbe Anzahl Frames verglichen werden muss.

Ohne Änderungen schwankt selbst hier der Speicherverbrauch von Durchlauf zu Durchlauf. Die wohl größte Abweichung kommt durch die Verarbeitung von Eingabe-Events zustande. Dies lässt sich dadurch eliminieren, dass während der Ausführung des Programms keinerlei Eingaben oder Maus-Bewegungen getätigt werden und dass keine Events für Controller abgefragt werden. Leichte Abweichungen beim Laden des Shader-Caches, bei der Anzeige des Ladebalkens zu Programmstart oder der Anzeige der Frame Times des Profiling verursachen zudem inkonsistenten Speicherverbrauch. Darum wurden diese Funktionalitäten aus dem Test-Programm entfernt. Da es dennoch zu Abweichungen von mehreren Kilobytes kommen kann, erfolgt die Angabe des Speicherverbrauchs zu Programmstart stets bis auf zehn Kilobytes gerundet. Bei der Auswertung des Speicherverbrauchs werden im Folgenden jeweils drei Zustände gemessen:

1. **Der Speicherverbrauch nach 0 Frames:** Dieser steht repräsentativ für den Speicherverbrauch während des Programmstarts. Vor den Modernisierungen betrug dieser Wert 172,09 MB.
2. **Der Speicherverbrauch nach 20 Frames:** In den ersten Durchläufen des Game Loops kommt es an einigen Stellen zu erstmaligen Reservierungen, die sich dann aber nicht mehr oder nur noch selten wiederholen. Um näher an den tatsächlichen Speicherverbrauch pro Frame zu kommen, kann dieser Wert subtrahiert werden. In diesen 20 Frames kommen vor den Modernisierungen 31,10 MB hinzu. Dies beläuft sich auf etwa 1,56 MB pro Frame.
3. **Der Speicherverbrauch nach 666 Frames:** Nachdem sich die initiale Speicher-Reservierung stabilisiert hat, kann über die nächsten Frames der durchschnittliche Speicherverbrauch pro Frame errechnet werden. Dieser gibt einen Anhaltspunkt darüber, wie sich der Speicherverbrauch im restlichen Verlauf entwickelt hätte. Zum Vergleichspunkt vor den Modernisierungen beträgt dies zusätzliche 303,96 MB. Pro Frame sind das etwa 0,47 MB.

4.6 Startdauer

Neben der Ausführungsgeschwindigkeit spielt auch die Startdauer des Programms eine Rolle. Schließlich möchte der Nutzer nicht lange nach dem Aufruf des Programms warten. Insofern wird neben dem Speicherverbrauch bis Frame 0 auch die Zeit gemessen, welche bis dahin benötigt wurde. Hierfür dient wieder das Linux-Kommando *time*. Dabei wird allerdings nicht nur die Startdauer gemessen, sondern auch die Zeit, die das Programm zum Beenden braucht. Jedoch sind beim Beenden keine neuen Ressourcen zu laden oder komplexe Initialisierungen durchzuführen. Von daher reflektieren gemessene Veränderungen die Differenzen der Startdauer. Wie in Abschnitt 4.3 wird aus je fünf Durchläufen der Durchschnitt ermittelt. Für den Stand vor jeder Modernisierung ergeben sich folgende Werte:

Messung #1	Messung #2	Messung #3	Messung #4	Messung #5	**Durchschnitt**
1,091 s	1,042 s	1,053 s	1,075 s	1,037 s	**1,060 s**

Tabelle 4.3: Initiale Startdauer

5 Versionsupgrade

5.1 Allgemein

Seit C++11 wurden drei C++ Standards fertiggestellt: C++14, C++17 und C++20. Zudem befindet sich C++23 in der Entwicklung. Jeder Standard bietet unterschiedliche neue Möglichkeiten. Zum Teil wird dabei ein Aspekt der Sprache – beispielsweise Schleifen – in mehreren Standards erweitert. Damit Modernisierungen eines bestimmten Aspekts zusammen durchgeführt werden können, ist es von daher notwendig, zunächst den neuesten C++ Standard zu aktivieren und erst anschließend einzelne Aspekte zu modernisieren. Dafür sind moderne Compiler gefragt. Während für manche der durchgeführten Modernisierungen noch Clang 14 ausreichte, wurde später eine Vorabversion von Clang 15 notwendig. Selbst diese setzt den Standard C++20 noch nicht vollständig um. Es ist bei einer Modernisierung also ratsam, immer auf dem neuesten Stand zu bleiben.

5.2 C++11

Obwohl OGRE in der verwendeten Version 13.3.3 bereits auf C++11 eingestellt ist, ist dieser Standard erst seit Version 1.11, welche am 29. April 2018 erschienen ist, vorausgesetzt[117, D. 1.11-Notes].[4] Insofern nutzt OGRE nur zum Teil Features aus C++11, weswegen die Modernisierung im Folgenden auch C++11 einschließt. Um auch wirklich C++ in seiner *standardisierten* Version zu verwenden, wurden die nicht standardisierten präkompilierten Header deaktiviert. Zu den wichtigsten Ergänzungen zur Sprache in C++11, welche in folgenden Kapiteln keinen dedizierten Abschnitt haben, gehören:

- **Perfect Forwarding:** Perfect Forwarding hängt stark mit den *move semantics* aus Abschnitt 8.5 zusammen. Es geht darum, Argumente von einer Funktion an eine andere weiterzureichen, ohne dass die Kategorie der Argument-Werte verloren geht. Insbesondere müssen temporäre Werte auch als temporäre Werte weitergereicht werden. Möglich wird dies mit Template-Parametern der Form `T&&`. Wird `T` hergeleitet, so resultieren Argumente der Typen `A` in `A&&`, `A&` in `A&` und `A&&` in `A&&`[49, § References to Rerences, § Template Argument Deduction with A&&].
- **Lambda Ausdrücke:** Ein Lambda-Ausdruck ist eine verkürzte Schreibweise für die Erzeugung eines Objektes mit einem `operator()`. Er beginnt immer mit eckigen Klammern. Darin können sichtbare Variablen angegeben werden, die innerhalb des Ausdrucks verwendbar sind. Sie können von dem entstehenden Objekt wahlweise kopiert oder referenziert werden. In optionalen runden Klammern folgen die Parameter analog zu einer Funktion und in geschweiften Klammer schließlich der Funktionskörper[143]. Ein gültiger Lambda-Ausdruck kann also die Form `[]{}` haben.
- **Variadische Templates:** Templates mit einer unbestimmten Anzahl an Parametern konnten vor C++11 nicht direkt ausgedrückt werden. Umwege umfassten schwer wartbaren und sich wiederholenden Quellcode, insbesondere mit vielen Macros[43, § 1]. Variadische Templates ermöglichen ein sog. Parameter-Pack mit ... vor dem Parameter-Namen des Templates für eine beliebige Anzahl an Argumenten. Im darauf folgenden Kontext kann dieses Parameter-Pack mit ... nach dem Parameter-Namen wieder in die jeweiligen Argumente zerlegt werden. Anwendung findet diese Technik in vielen Hilfs-Funktionen der Standard-Bibliothek.

[4] Version 13 folgt dabei auf Version 1.12, welche der Nachfolger von 1.11 ist. Von 1.12 auf 13 fand eine Änderung des Versionsschemas statt[117, D. 13-Notes].

5.3 C++14

OGRE mit C++14 zu kompilieren erfordert lediglich die Änderung einer CMake Variablen von 11 auf 14[14, B. CXX_STANDARD]. Dabei konnten keine Inkompatibilitäten festgestellt werden. Ein Blick in die generierten Assembly-Dateien zeigt jedoch, dass sich dabei sieben Translations-Einheiten verändert haben. Die Änderungen bestehen überwiegend aus Neuanordnungen desselben Codes, mit C++14 werden aber auch einige Abschnitte weg optimiert. Allerdings wurde die Assembly-Datei für `OgreStringInterface` mit über 200 Änderungen fast vollständig umgewandelt. Die darin definierte Klasse `StringInterface` besteht u.a. aus Unterobjekten der Typen `string`, `vector` und `map`. An `string` wurde in C++14 nur wenig verändert. Einige Member-Funktionen erhielten ein neues Standard-Argument um Fehler-Anfälligkeit zu reduzieren[45], bei anderen Member-Funktionen wurde die Exception-Spezifikation angepasst[48][110][119]. Sowohl `vector`, `map` als auch alle anderen Standard-Container erhielten ein optionales Allocator-Argument im Constructor[147]. Als assoziativer Container erhielt `map` die Möglichkeit des heterogenen Vergleichs[71]. Dies verhindert, dass z.B. für eine `map` mit `string` als Schlüssel-Typ jedes Mal ein neuer `string` erstellt wird, wenn ein Argument vom Typ `char*` als Schlüssel verwendet wird. Stattdessen werden die Schlüssel direkt verglichen, ggf. mit `less<void>`[69]. Diese Möglichkeit ist jedoch nur dann aktiv, wenn ein entsprechender Vergleichs-Typ explizit angegeben wird. Weitere wichtige Neuerungen sind in den folgenden Kapiteln behandelt.

5.4 C++17

Die Umstellung auf C++17 erfolgt analog zu C++14 und ohne weitere Inkompatibilitäten. Dabei bleiben die generierten Assembly-Dateien identisch. Obwohl die in C++17 eingeführte *guaranteed copy elision* unnötige Kopien vermeiden soll, spiegelt sich dies nicht im Assembly wider. Da diese Änderung jedoch nur darin besteht, dass *copy elision* zu einer Garantie verschärft wurde, ist davon auszugehen, dass Clang diese Optimierung bereits mit den vorherigen Versionen durchgeführt hat. In der Tat wurde vor C++17 bemängelt, dass diese Optimierung in der Praxis zwar durchgeführt wurde, aber die dadurch eigentlich wegfallenden Anforderungen an einen kopierten Typen nach wie vor erhalten bleiben[123]. Es ist nun also in gewissen Fällen nicht mehr notwendig, dass ein Typ kopierbar ist.

Eine weitere Neuerungen soll an dieser Stelle hervorgehoben werden, der in folgenden Kapiteln kein eigener Abschnitt gewidmet ist: Falt-Ausdrücke. Die Argumente, welche seit C++11 an ein Variadisches Template übergeben werden können, sind seit C++17 *faltbar*. Das heißt, für einen binärer Operator wie +, dass der Ausdruck `(... + arg)` alle Argumente aufsummiert. Es gibt vier verschiedene Muster, je nach dem, ob von links nach rechts oder von rechts nach links ausgewertet werden soll und ob es einen initialen Wert gibt [136, § Proposal].

5.5 C++20

Mit C++20 generiert Clang folgende neue (gekürzte) Warnung:

```
ISO C++20 considers use of overloaded operator '=='
(with operand types 'Token' and 'Token') to be ambiguous
despite there being a unique best viable function
```

Die Ursache für diese Warnung ist, dass nur ein Parameter der angegebene Vergleichs-Funktion `const` qualifiziert ist – bei einem Vergleich sollten dies stets beide Parameter sein[130, § F.16, § Con.2, § Con.3]. Relevant wird diese Eigenschaft allerdings erst mit den in C++20 eingeführten synthetischen umgeschriebenen Vergleichs-Operatoren[112]. Dadurch wird aus einem Vergleichs-Operator ein zweiter mit umgekehrten Parametern generiert. Da es in diesem Fall nun zu zwei Operatoren der Formen

```
bool operator == (Token       &, Token const&)
bool operator == (Token const&, Token       &)
```

käme, ist die Auswahl nicht mehr eindeutig, obwohl es sich um ein und dieselbe Funktion handelt[63, § C.2.7]. Die Lösung ist trivial: Die Funktion muss mit `const` qualifiziert werden. Insofern hilft C++20 bereits dabei, problematischen Code zu entdecken. Neben dem synthetischen Umschreiben der Vergleich-Operatoren gibt es zudem nun den neuen Operator <=>, aus welchem die Operatoren <, >, <= und >= synthetisiert werden können[135, § 2.3]. Diese Neuerung findet bereits in der Standard-Bibliothek Anwendung[11][114][129], womit sie auch die existierenden Assembly-Dateien beeinflusst. In der Tat sind die meisten Assembly-Änderungen in insgesamt 31 Dateien in unmittelbarer Nähe oder in Zusammenhang mit einem Vergleich, auch sind Definitionen einzelner Operatoren durch eine Definition des Operators <=> ersetzt worden.

Weitere Änderungen stehen in Zusammenhang mit einem neuen Rückgabe-Typ in der Funktion `remove` von `list`[63, § C.2.11][103] und der Ersetzung der seit C++17 veralteten `destroy` Funktion von `allocator` durch die in C++17 eingeführte Funktion `destroy_at`[37][63, § C.2.15][93][94, § D.9].

Auch wenn sich in folgenden Kapitel kein dedizierter Abschnitt mit Concepts beschäftigt, so sind sie dennoch ein wichtiger Bestandteil von C++20. Sie erlauben es, Einschränkungen auf Templates anzugeben. Beispielsweise kann so bei überladenen Funktionen besser angegeben werden, welche Überladung zu wählen ist. Concepts sind dabei wie logische Terme aufgebaut. Verschiedene atomare Einschränkungen werden mit Konjunktionen und Disjunktionen zusammengefügt[137, § 17.10.1]. Nur wenn der entstehende Term für das Concept erfüllt ist, kann auch ein zugehöriges Template verwendet werden. Ein primitives Beispiel sieht dabei wie folgt aus:

```
template<typename T>
concept GreaterOne = sizeof(T) > 1;

template<typename T>
requires GreaterOne<T>
void F(T);

F(1);    // F wird aufgerufen, da sizeof(int) > 1
F('c');  // Keine passende Funktion gefunden, da sizeof(char) == 1
```

Stehen Templates mit verschiedenen Concepts im Konflikt zueinander, wird überprüft, ob ein Term einen anderen impliziert. Concepts, welche ein anderes implizieren, werden diesem vorgezogen: Sie sind stärker eingeschränkt. Das bedeutet, dass auf Concepts eine partielle Ordnung definiert ist[137, § 17.10.4].

5.6 C++23

Wie auch beim Übergang von C++14 auf C++17 bleiben die Assembly-Dateien von C++20 auf C++23 identisch. Da Clang in der verwendeten Version 14 jedoch C++23 nur in Teilen umsetzt[75], können die Unterschiede in späteren Versionen ansteigen. Nichtsdestotrotz kommen in folgenden Kapiteln ein paar Features aus C++23 bereits zum Einsatz.

5.7 Clang 15

Für die Verwendung der aktuellsten Features ist selbstverständlich auch eine aktuelle Version eines Compilers erforderlich. So war es auch im Verlauf dieser Arbeit notwendig von Clang 14 auf Clang 15 zu wechseln. Idealerweise könnte ohne Weiteres eine Version durch die neuere ersetzt werden. Da dies in diesem Fall jedoch nicht zutraf, werden im Folgenden die dabei auftretenden Probleme festgehalten. Gleich bei der erstmaligen Kompilation fällt auf, dass das Template `binary_function` aus der Standard-Bibliothek nicht mehr existiert und die Kompilation fehlschlägt[63, § C.3.12]. Es wurde offiziell schon mit C++17 entfernt, da Perfect Forwarding in C++11 es obsolet machte[72, § D.8]. In der Tat ist die Verwendung dieses Templates in OGRE überflüssig.[5]

Das nächste Problem kann festgestellt werden, wenn das Programm zum ersten mal mit Address-Sanitizer ausgeführt wird. So kommt es in der Funktion `deallocate` von `AlignedMemory` zu einem Speicherzugriffsfehler. Die Ursache hierfür liegt allerdings in dem Template `AlignedAllocator` oder besser gesagt in den Member-Funktionen, die es von `allocator` erbt. Seit C++23 ist eine davon die Funktion `allocate_at_least`. Bei Speicherreservierungen kann es vorkommen, dass diese in Blöcken bestimmter Größe reserviert werden, welche gegebenenfalls größer sind als eigentlich notwendig. Wird daraufhin mehr Speicher benötigt kommt es zu einer erneuten Speicherreservierung, auch wenn eigentlich der noch freie Speicher im Block ausreichen würde. Mit `allocate_at_least` wird neben einem Pointer zum Speicherblock auch dessen vollständige Größe zurückgegeben. Damit können gegebenenfalls unnötige Speicherreservierungen eingespart werden[148].

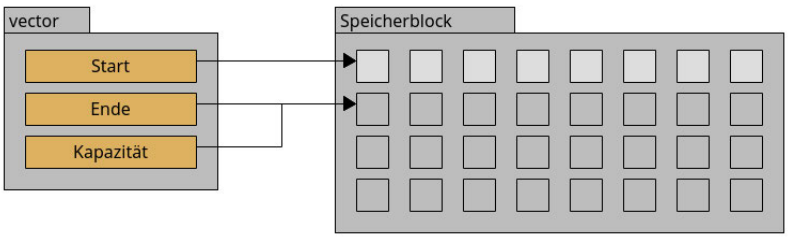

Abbildung 5.1: Vector mit ausgeschöpfter Kapazität ohne allocate_at_least

Abbildung 5.1 zeigt einen `vector`, in dem für acht Objekte Speicher reserviert wurde. Seine Kapazität ist erreicht. Das Hinzufügen eines neuen Objektes führt dazu, dass ein neuer Speicherblock reserviert wird, obwohl im alten noch Platz wäre. Zudem müssten die acht existierenden Element verschoben werden. In Abbildung 5.2 hingegen wird ein `vector` gezeigt, der seinen Speicher mit

[5] Wäre die Definition von `binary_function` dennoch notwendig, könnte dieses Template in libc++ mit dem Macro `_LIBCPP_ENABLE_CXX17_REMOVED_UNARY_BINARY_FUNCTION` wiederhergestellt werden.

`allocate_at_least` reserviert hat. Obwohl nur für acht Objekte Speicher reserviert werden musste, stimmt die Kapazität mit der tatsächlichen Größe des Speicherblocks überein. Das Hinzufügen eines neuen Objektes benötigt nun keinen neuen Speicherblock und auch kein Verschieben der existierenden Objekte.

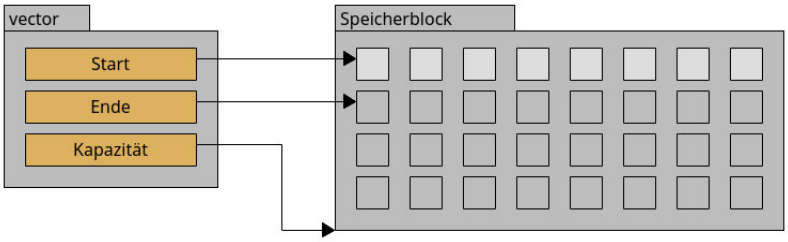

Abbildung 5.2: Vector mit freier Kapazität durch allocate_at_least

Mit Clang 15 findet dies u.a. Anwendung in `vector`, wo nun ein Aufruf an `allocate_at_least` statt an `allocate` geschieht. Für den benutzerdefinierten Allocator `AlignedAllocator` bedeutet das ein Aufruf and die geerbte Funktion von `allocator` statt wie bisher an die benutzerdefinierte. Letztere hatte dabei zum Zweck, reservierte Speicher-Adressen auszurichten, sodass diese mit einer gewissen Anzahl Nullen enden. Die Korrektur findet nun nicht mehr in `allocate` statt, in `deallocate` wird jedoch davon ausgegangen, wodurch es zum beobachteten Fehler kommt. Zum einen kann dies behoben werden, indem `AlignedAllocator` ebenfalls `allocate_at_least` implementiert. Zum anderen gibt es seit C++17 Speicherreservierungen mit Ausrichtung. Dabei wird an den `new`-Operator ein zusätzliches Argument des Typs `align_val_t` übergeben[105]:

```
void* allocate(size_t size, size_t alignment)
{
  return new(std::align_val_t{alignment})
             unsigned char[size];
}
```

6 Verbesserung der Syntax

6.1 Allgemein

Gewisse Code-Muster wiederholen sich in vielen Projekten. Für Programmierer bedeutet diese Wiederholung Schreibarbeit, Fehleranfälligkeit und erhöhte Lesedauer. Es gibt einige Modernisierungen, welche nur zum Ziel haben, dass häufig vorkommende Code-Muster in verkürzter Form geschrieben werden können. Die folgenden Abschnitte behandeln eine Auswahl solcher Modernisierungen, die auch in OGRE Anwendung fanden.

6.2 Überflüssige Parameter des Typs void

An vielen Stellen verwendet OGRE für Funktionen ohne Parameter den Typen `void`:

```
int Function(void);
```

Obwohl dies eine gültige Konvention ist, wird sie als nicht mehr zeitgemäß angesehen[130, § NL.25]. Stattdessen kann der Raum zwischen den Klammern einfach leer gelassen werden. Clang-Tidy kann diesen Prozess für das gesamte Projekt automatisch durchführen[74, B. redundant-void-arg]. Die Modernisierungen hat dabei keine Auswirkungen auf das Assembly.

6.3 Initialisierung des Rückgabewerts

Die mit C++11 eingeführte *list-initialization* ermöglicht die Verkürzung eines Rückgabewertes durch die Auslassung des Typs[97]. Die Funktion

```
Vector2 Function(uint32 x, uint32 y) { return Vector2(x, y); }
```

kann mit Clang-Tidy automatisch zu

```
Vector2 Function(uint32 x, uint32 y) { return {x, y}; }
```

umgewandelt werden[74, B. return-braced-init-list]. In OGRE führt dies zunächst zu Fehlermeldungen folgender Form:

```
non-constant-expression cannot be narrowed from type
'Ogre::uint32' (aka 'unsigned int') to 'int' in initializer list
```

`Vector2` ist dabei definiert als `Vector<2, int>`. Dieser Typ setzt sich aus zwei `int` zusammen. Durch die Konvertierung ändert sich der darstellbare Zahlenbereich. Nur noch die untere Hälfte der positiven Zahlen kann repräsentiert werden, alle Zahlen darüber werden negativ. Insofern ist dies eine nützliche Fehlermeldung. Es besteht sowohl die Option, die Konvertierung explizit zu verlangen (z.B. mit `static_cast`), oder aber man ändert den Rückgabe-Typ zu `Vector<2, uint32>` und vermeidet die Umwandlung. In diesem Fall wurde sich für letzteres entschieden. Insgesamt bleibt das Assembly durch diese Umwandlung unverändert.

6.4 Auto

Die Deklaration einer Variablen erfordert in C++ die Angabe ihres Typs. Dieser kann dabei sowohl besonders lang als auch gänzlich unbekannt sein:

```
template<typename T>
void FindAndCall(T const& t, std::vector<T> const& v)
{
  typename std::vector<T>::const_iterator // sehr lang
    it = std::find(v.begin(), v.end(), t);
  ??? result = (*it)(); // unbekannt
}
```

Beide Probleme sollen mit dem in C++11 abgeänderten Schlüsselwort `auto` angegangen werden. Es kann anstelle eines konkreten Typs eingesetzt werden. Der Typ der Variablen wird dann mit den gleichen Regeln hergeleitet, die auch für die Argumente von Template-Funktionen dienen (wie `t` im obigen Beispiel)[55, § 1.1, § 4]. Seit C++14 ist es sogar möglich, mit `auto` als Parameter einen generischen Lambda-Ausdruck ohne `template` Schlüsselwort zu erstellen[142], seit C++20 gilt dies für jede Funktion[146]. Mit C++23 kann `auto` letztlich für expliziten Typ-Verfall genutzt werden[151]:

```
void FindAndCall(auto const& t, // hergeleitet
  std::vector<decltype(auto(t))> const& v) // nicht hergeleitet
{
  auto it = std::find(v.begin(), v.end(), t); // sehr kurz
  auto result = (*it)(); // hergeleitet
}
```

Der verkürzte Code ist dabei einfacher zu lesen und verringert die Fehleranfälligkeit[130, § ES.11]. Insbesondere wenn sich später in der Entwicklung Rückgabe-Typen verändern, passen sich Variablen mit `auto` automatisch an. Die explizite Angabe des Typs kann zu impliziten Konvertierungen führen, die sich gegebenenfalls auf die Ausführungszeit auswirken. Ist der Name des Typs einer Variablen besonders lang oder im selben Ausdruck mehrmals vorhanden ersetzt Clang-Tidy den Typ mit `auto`[74, B. auto].

Bei der Anwendung in OGRE konnte festgestellt werden, dass dies zu Fehlern führt, wenn in einer Deklaration zwei Initialisierer mit unterschiedliche Typen existieren, beispielsweise `iterator` und `const_iterator`. Die Verwendung von `auto` für mehrere Initialisierer ist dabei zwar vorgesehen[9], es kommt mit `auto` jedoch zu einem Konflikt in der Herleitung des Typen beider Initialisierer. Wo vorher der explizite Typ für beide Deklarationen nämlich `const_iterator` war, fand eine implizite Konvertierung statt, welche nun nicht mehr möglich ist. Dies kann behoben werden, indem man die Initialisierer auf einzelne Deklarationen aufteilt, jeder mit seinem eigenen `auto` Schlüsselwort. Dies entspricht der Empfehlung nur eine Variable pro Deklaration einzuführen[130, § ES.10]. Die automatisierte Änderung führt dabei in OGRE lediglich zur Verschiebung einer Funktions-Definition in einer Assembly-Datei.

6.5 Trailing Return-Type

Seit C++11 gibt es neben der herkömmlichen Syntax für die Funktions-Deklaration

```
int Add(int a, int b);
```

noch eine weitere, bei welcher der Rückgabe-Typ hinten angestellt ist:

```
auto Add(int a, int b) -> int;
```

Der Hauptzweck hierfür ist die Verwendung des Operators `decltype` mit den Parametern der Funktion. Die Deklaration

```
template<typename T, typename U>
auto Add(T a, U b) -> decltype(a + b);
```

leitet den Rückgabe-Typ aus generischen Argumenten her. Wo dieser Typ entweder `T` oder `U` ist, wäre die herkömmliche Syntax noch ausreichend. Ist dies jedoch ein dritter Typ `R`, so kann dies nicht mehr ausgedrückt werden[56, § 2.1]. Dazu kommt die Leserlichkeit bei einem komplexeren Rückgabe-Typen. Hier ist es bei

```
int (*GetAddFunction())(int, int);
```

schwieriger, direkt zu sehen, wo sich die Parameter der Funktion befinden als es bei

```
auto GetAddFunction() -> auto(*)(int, int) -> int;
```

der Fall wäre[56, § 3]. Und das obwohl der Typ ein und derselbe ist. Für den Fall, dass diese Syntax für die spätere Weiterentwicklung gebraucht wird, es es ratsam, die Syntax einheitlich im gesamten Projekt zu verwenden. Diese Umstellung kann mit Clang-Tidy automatisch vollzogen werden[74, B. trailing-return-type]. Sollten in der Deklaration jedoch Macros vorkommen, wie z.B. in OGRE `OGRE_NODISCARD` oder `OGRE_FORCE_INLINE`, schlägt die Umstellung fehl oder ordnet die Macros inkorrekt an. Insofern ist zu empfehlen, diese Macros im Vorfeld aufzulösen. Mit C++14 wird die explizite Angabe des Rückgabe-Typs sogar optional und kann aus der Rückgabe-Anweisung hergeleitet werden, sofern diese einsehbar ist[98]. Besonders interessant ist dies für Funktionen mit einer `if constexpr` Bedingung, was seit C++17 möglich ist[86]. Jede Verzweigung darf dabei einen anderen Rückgabe-Typ haben, der schließlich hergeleitet werden muss. Dieses Feature fand jedoch noch keine Anwendung in OGRE.

6.6 Using Alias

Das mit C++11 eingeführte Schlüsselwort `using` ist eine modernisierte Form des aus `C` bekannten `typedef`. Es dient dazu, ein Alias für einen Typen zu deklarieren. `using` hat dabei den Vorteil, dass der neue Name des Alias getrennt vom Typen ist. Bei bei `typedef` ist er darin verschachtelt und gegebenenfalls schlechter auf Anhieb zu erkennen[130, § T.43]:

```
typedef int (*Alias)(int);      // alt
using Alias = int(*)(int);      // neu
```

Die Haupt-Funktionalität einer Alias-Deklaration besteht jedoch darin, einen alternativen Namen für ein Template anzulegen, was mit `typedef` nicht möglich ist[31]. Dies verkürzt Ausdrücke, welche auf einen in einem Template geschachtelten Typen verweisen oder bei einem Template mit mehreren Argumenten einige vorgeben. Dies findet auch in der Standard-Bibliothek Anwendung[63, § 21.3.3]:

```
template<class T>
using add_pointer_t = typename add_pointer<T>::type;
template<bool B>
using bool_constant = integral_constant<bool, B>;
```

Letzteres konnte in OGRE allerdings noch keine Anwendung finden. Für zukünftige Änderungen könnte dies jedoch der Fall werden. Mit Clang-Tidy lässt sich aber der erste Verwendungszweck automatisiert umsetzen, wobei `typedef` Deklarationen durch `using` ersetzt werden[74, B. using]. Allerdings konnte dabei festgestellt werden, dass Aliase zu Arrays nicht umgewandelt werden. Wie zu erwarten hat dies keine Auswirkungen auf die Assembly-Dateien.

6.7 Member Initialisierer

Um die nicht-statischen Daten-Member eines Objektes einheitlich zu initialisieren war es vor C++11 noch notwendig, in jedem einzelnen Konstruktor dieselben Werte anzugeben. Soll der Standard-Wert geändert werden erfordert dies eine Änderung in jedem einzelnen Konstruktor. Insbesondere wenn die Konstruktor-Definitionen über mehrere Dateien verteilt sind, ist es schnell passiert, dass ein Konstruktor nicht angepasst wird. Dies kann mit Member-Initialisierern umgangen werden[128]. Auch ist es seit C++11 möglich, einen Konstruktor von einem anderen aufzurufen, wodurch identischer Code nur in einem Konstruktor vorkommen muss. So könnte die Definition

```
struct S
{
  int i;
  float f;
  S() : i(1), f(2.4f) {}                 // #1
  S(int j) : i(j), f(j * 2.4f) {}        // #2
  S(float g) : i(1), f(g) {}             // #3
  S(int j, float g) : i(j), f(j * g) {} // #4
};
```

zu

```
struct S
{
  int i{ 1 }; // immer gleich
  float f;
  S() : S(2.4f) {}                       // #1 delegiert an #3
  S(int j) : S(j, 2.4f) {}               // #2 delegiert an #4
  S(float g) : f(g) {}                   // #3 Standard Init. von i
  S(int j, float g) : i(j), f(j * g) {} // #4
};
```

verkürzt werden. Dies kommt der Wartbarkeit zugute und vereinfacht das Lernen von C++ mit Vorkenntnissen aus Sprachen, in welchen dies bereits möglich ist[132]. Hinzu kommt, dass ein Member-Initialisierer die einzige Methode ist, ein Aggregat (siehe Abschnitt 6.8) mit Standard-Werten zu versehen, da in diesem keine Konstruktor-Deklarationen existieren dürfen[63, § 9.4.2].

Clang-Tidy bietet eine automatisierte Umwandlung zu Member-Initialisierern an[74, B. default-member-init]. Dabei kann es jedoch vorkommen, dass in einem Header, welcher vorher lediglich eine Forwärts-Deklaration benötigt hatte, nun die komplette Definition eines Typs benötigt wird um diesen zu initialisieren. Da Member-Initialisierer, anders als Konstruktoren, eine Initialisierung mit geschweiften Klammern voraussetzen, kommt es zudem zu ungültigen impliziten Konvertierungen, beispielsweise von `-1` in einen Typ ohne Vorzeichen. Hier schafft ein `static_cast` Abhilfe.

6.8 Aggregate

Aggregate erlauben es, die Unterobjekte eines Objektes mit geschweiften Klammern zu initialisieren. In Kombination mit den Member-Initialisierern entfällt die Notwendigkeit von einfacheren Konstruktoren. Damit ein Typ sich dies zu Nutze machen kann, muss dieser mehrere Anforderungen erfüllen, welche sich im Laufe der C++ Standardversionen verändert haben. Folgende Anforderungen spiegeln den aktuellen Stand zum Zeitpunkt des Schreibens wieder: Ein Typ ist ein Aggregat, wenn dieser entweder ein Array oder eine Klasse ist, welche weder benutzerdefinierte Konstruktoren, vererbte Konstruktoren, nicht öffentliche Daten-Member beziehungsweise Basis-Klassen oder virtuelle Funktionen bzw. Basis-Klassen aufweist[63, § 9.4.2].

Sind keine virtuellen Funktionen oder besonderen Invarianten im Konstruktor notwendig, so ist es möglich, bestimmte Konstruktoren für einen Typen redundant zu machen. Hierfür müssen alle Daten-Member und Basis-Klassen öffentlich gemacht werden. Zudem sind alle Konstruktoren, welche nicht nur die Daten-Member initialisieren, zu ersetzen, beispielsweise durch statische Member-Funktionen. Ist dann die Reihenfolge der Konstruktor-Parameter identisch zur Reihenfolge der nicht-statischen Daten-Member und werden diese mit denselben Werten initialisiert wie in einem etwaigen Standard-Konstruktor, können schließlich alle Konstruktoren entfernt werden und es entsteht ein Aggregat.

Seit C++20 benötigen diese weder explizite Konvertierungen der Argumente noch geschweifte Klammern zur Initialisierung. In manchen Fällen ist ersteres durchaus gewünscht. Letzteres adressiert ein Problem, welches mit Funktionen wie `make_unique` oder `emplace` auftrat. Diese verwenden intern nämlich runde Klammern, womit sie inkompatibel mit Aggregaten sind[145]. Da die verwendete Version von Clang dies leider noch nicht unterstützt[75], war es bei der Umwandlung geeigneter Typen in Aggregate notwendig, Initialisierungen mit runden Klammern in geschweifte umzuändern, sowie explizite Konvertierungen einzufügen. Entsprechende Vorkommen von `make_unique` oder `emplace` mussten auf eine explizite Kopie zurückgreifen. Darüber hinaus benötigt die Initialisierung von Aggregaten mit geschweiften Klammern innerhalb einer Macro-Funktion zusätzlich ein Paar runde Klammern, damit die Kommas nicht als Trennzeichen der Macro-Argumente interpretiert werden.[66, S. 248]. Dies war bei den mit *GoogleTest* geschriebenen Tests notwendig. Insofern wäre eine Reduktion der Konstruktoren mit einem Compiler, welcher C++20 vollständig unterstützt, einfacher. Fehler traten in OGRE dort auf, wo sich die Parameter-Reihenfolge im Konstruktor von der der Unterobjekte unterschied. Eine entsprechende Neuanordnung letzterer verschaffte dem Abhilfe. Es ist also mit besonderer Vorsicht auf die Reihenfolge zu achten.

6.9 Nested Namespace

Schon vor C++11 konnte ein `namespace` dazu verwendet werden, Deklarationen logisch abzugrenzen und Namenskollisionen zu vermeiden. Seit C++11 können diese mit `inline` definiert werden. Dies assoziiert den `namespace` mit dem jeweils äußeren `namespace`. Dadurch werden in diesem alle Member des inneren `namespace` verwendbar, als wären sie auch dessen Member. Dies ist günstig um verschiedene Versionen zu unterstützen[96]. Es bildet die Grundlage der in Abschnitt 2.2 festgestellten Inkompatibilitäten, da sich die tatsächlichen Symbol-Namen unterscheiden, weil jeweils der Name des `inline namespace` in der Syntax weggelassen werden kann. Für OGRE konnte dafür jedoch keine Anwendung gefunden werden. Anders verhält es sich mit geschachtelten `namespace` Definitionen. Seit C++17 können diese verkürzt dargestellt werden. Clang-Tidy kann

```
namespace Ogre
{
  namespace RTShader
  {
    // ...
  }
}
```

in

```
namespace Ogre::RTShader
{
  // ...
}
```

umwandeln[74, B. nested-namespace]. Beide Code-Abschnitte sind dabei äquivalent[59] und führen zum gleichen Assembly. Seit C++20 wäre dies auch mit `inline namespace` möglich, indem nach `::` ein `inline` folgt[95].

6.10 Unverarbeitete String-Literale

Bestimmte Sonderzeichen wie Anführungszeichen müssen mit sog. Escape-Sequenzen codiert werden, damit sie in einem String-Literal vorkommen können[63, § 5.13.3]. Sind diese in dem String besonders häufig vorzufinden, ist die notwendige Zeichenkette oft kompliziert und schwer zu lesen[4, S. 101]. Um dem Abhilfe zu schaffen gibt es seit C++11 die Möglichkeit, String-Literale in einer Form ohne Escape-Sequenzen anzugeben[18]. Hierfür wird das Präfix `R„(` mit Suffix `)"` benötigt. Diese können durch eine Zeichenkette mit bis zu 16 Zeichen zwischen Anführungszeichen und Klammer angepasst werden, sollte der String-Literal selbst `)"` enthalten[63, § 5.13.5]. Zum Beispiel:

```
char const* str = R"ABC(R"()" ist ein leerer String.)ABC"
```

Clang-Tidy kann dabei helfen, String-Literale mit solchen Sonderzeichen zu finden und in unverarbeitete String-Literale umzuwandeln[74, B. raw-string-literal]. Im Fall von OGRE betrifft dies lediglich eine Datei, bei der das Assembly dadurch unverändert bleibt.

6.11 Vergleichs-Operatoren

Um Objekte in C++ zu vergleichen stehen mit ==, !=, <, <=, > und >= insgesamt 6 Operatoren bereit. Für benutzerdefinierte Typen heißt das jedoch, dass 6 Funktionen definiert werden müssen. Soll zudem ein Typ mit einem anderen verglichen werden, sind sogar 18 Funktionen zu definieren[4, S. 536]. Dies ist nicht nur Schreibarbeit, es ist auch anfällig für logische Fehler und führt in manchen Fällen sogar zu suboptimaler Performance[135, § 2.2.2, § 2.6]. Seit C++20 können diese jedoch aus den Operatoren == und <=> synthetisiert werden[112]. Sie können zudem mit `default` definiert sein, was bedeutet, dass die Daten-Member nacheinander lexikographisch verglichen werden. Auch definiert ein `default` <=> Operator implizit einen `default` == Operator. Für bestehenden Code heißt das, dass ein vorhandener == Operator einen != Operator obsolet macht. Die verbleibenden vier Operatoren, welche aller Wahrscheinlichkeit nach ähnlich aufgebaut sind, können zu einem <=> Operator zusammengeführt werden. Dies geschah auch für die Standard-Bibliothek[114]. Ein Ausdruck `a < b` wird dann in `(a <=> b) < 0` umgewandelt, analog für die anderen Vergleichs-Operatoren.

Wie in Abschnitt 5.5 bereits beschrieben führt allein schon die Umstellung auf C++20 zur Verwendung dieser Modernisierung wegen den Änderungen in der Standard-Bibliothek. Aber auch für den Code von OGRE heißt das, dass an vielen Stellen Definitionen des != Operators entfernt werden und Definitionen anderer Vergleichs-Operatoren umgeschrieben werden können. Bis auf wenige Ausnahmen ist dies auch ohne große Schwierigkeit zu machen. So existiert in OGRE ein > Operator, welcher seine Argumente mit dem [] Operator von `map` modifiziert[63, § 24.4.4.3]. Es ist unklar, ob dies gewollt ist oder ob es sich um einen Fehler handelt, da ein Vergleich im Normalfall seine Argumente nicht verändert. Diese Funktion wurde aus diesem Grund nicht abgeändert.

Es ist bei einer schrittweisen Umstellung darauf zu achten, zunächst die Typen zu verändern, welche in anderen als Daten-Member Einsatz finden, damit falls möglich `default` für die Operatoren als Definition ausreicht. In libc++ befindet sich derzeit noch keine Umsetzung des <=> Operators für `string`. Diese ist jedoch für die implizite Generierung notwendig. Sie konnte jedoch manuell mit Hilfe der Member-Funktion `compare` umgesetzt werden[63, § 23.4.3.8.4]. Alternativ ist es auch möglich, aus einem == und einem < Operator eines Daten-Members einen <=> Operator zu synthetisieren[113], davon wurde allerdings keine Verwendung gemacht. Die Umstellung berührt letztlich 47 Assembly-Dateien. Dabei werden zum Teil einige Funktionen zu einer zusammengeführt, sei es der == oder <=> Operator.

6.12 Schleifen und Ranges

Iteratoren gibt es in C++ schon seit längerer Zeit. Der wohl einfachste Iterator ist ein Pointer zu einem Element eines Arrays. Aber auch Standard-Container definieren eigene Iterator-Typen. Sie werden vor allem als Argumente für Standard-Algorithmen verwendet, wo ein `begin`-Iterator mit einem `end`-Iterator übergeben wird:

```
std::algorithm(container.begin(), container.end(), ...);
```

In diesen Funktionen oder auch außerhalb werden sie dann für Schleifen verwendet:

```
for (auto it = container.begin(); it != container.end(); ++it)
{
  auto& element = *it;
  // ...
}
```

Diese Syntax wurde als zu umständlich angesehen, die notwendige mehrmalige Wiederholung des Iterators `it` ist zudem anfällig für Fehler. Seit C++11 versteht man unter einem Iterator-Paar `begin` und `end` eine *Range*. So kann jedes Objekt, das die zwei entsprechenden Funktionen anbietet, mit der neuen Syntax

```
for (auto& element : container)
{
  // ...
}
```

verwendet werden. Gegebenenfalls ist dies nicht nur kompakter zu schreiben sondern auch schneller in der Ausführung[109]. Seit C++17 ist es mit diesem Konstrukt zudem möglich, dass sich die Typen von `begin` und `end` unterscheiden, sofern diese noch vergleichbar sind[106]. Dies ebnet den Weg für Vereinfachungen und Optimierungen, welche schließlich in C++20 mit Alternativen zu den alt-bekannten Algorithmen genutzt werden. Mit Hilfe von Concepts ist es nun möglich, diese in vereinfachter Form aufzurufen[107]:

```
std::ranges::algorithm(container, ...);
```

Die Concepts stellen dabei sicher, dass es sich bei dem übergebenen Objekt auch tatsächlich um eine Range mit `begin` und `end` Funktionen handelt. Die geringere Anzahl an Argumenten macht den Code leichter verständlich. Auch sind `begin` und `end`, welche fast immer zusammengehören, nicht mehr voneinander getrennt. Es wird unwahrscheinlicher, dass es zu Verwechslungen der Positionen der Argumente kommt[130, § I.23, § I.24]. Diese Umwandlung folgt einem recht simplen Muster und konnte von daher mit Hilfe von regulären Ausdrücken über das gesamte Projekt angewandt werden.

In OGRE gibt es mehrere tausende Schleifen. Sie alle auf einmal von Hand zu modernisieren ist unpraktikabel. Clang-Tidy ermöglichst zumindest eine teilweise automatisierte Umwandlung jener Schleifen, welche dem oben genannten Muster entsprechen. Schleifen, in denen der zugrundeliegende Container verändert wird, werden dabei nicht konvertiert. Dies könnte nämlich zu ungültigen Iteratoren im nächsten Schleifendurchlauf führen[74, B. loop-convert]. Unglücklicherweise gibt es noch weitere Schleifen, welche zwar semantisch identisch sind, syntaktisch jedoch zu sehr abweichen um von Clang-Tidy erkannt zu werden. Hierzu zählen vor allem Fälle, in denen der Iterator der Schleife außerhalb deklariert oder sogar initialisiert ist. Auch kommen in OGRE oft Schleifen vor, in denen neben dem Iterator noch eine weitere Variable pro Schleifendurchlauf inkrementiert wird. Reguläre Ausdrücke können diese Muster erkennen und schließlich in eine Form bringen, die Clang-Tidy vereinfachen kann. Die nach diesem Schritt übrig gebliebenen Schleifen werden schließlich manuell umgewandelt.

Es ist zudem seit C++20 möglich, einen Initialisierer vor den Range-Ausdruck zu stellen, dessen Variable nur innerhalb der Schleife zugänglich ist. Dies ist vor Allem dann interessant, wenn über eine Range eines temporären Objektes iteriert wird, welches andernfalls sofort zerstört werden würde[65, § Discussion]:

```
auto f() -> Object;
for (auto obj = f();
      auto& element : obj.container)
{
  // ...
}
```

Es ist aber auch nützlich, wenn noch eine weitere Variable nur innerhalb der Schleife benötigt wird:

```
for (int i = 0;
      auto& element : container)
{
  ++i;
}
```

Gleiches gilt seit C++23 auch für Typ-Aliase[87], dafür konnte in OGRE jedoch keine Verwendung gefunden werden.

Letztlich gibt es in OGRE auch Iterator-Paare, welche nicht direkt von einem Container stammen, sondern nur einen Teil dessen enthalten sollen. Für kontinuierlich im Speicher abgelegte Objekte ermöglicht `span` seit C++20 das Erstellen eines Views auf eine Untermenge des Containers. Dabei ist neben der Angabe des `begin`-Iterators entweder ein `end`-Iterator oder die Länge der Range notwendig[80, § Construction]:

```
std::span skipFirst{container.begin() + 1, container.end()};
std::span firstFive{container.begin(), 5};
```

`span` ist jedoch nicht einsetzbar für Container wie `map`, deren Elemente nicht kontinuierlich im Speicher liegen. In den Fällen, wo dennoch eine Range in einer Schleife vorkommen soll, wurde an den entsprechenden Stellen in OGRE eine simple lokale Klasse folgender Form verwendet:

```
struct LocalSpan
{
  Iterator Begin;
  Iterator End;
  auto begin() { return Begin; }
  auto end() { return End; }
};
```

6.13 Structured Binding

Ein häufiger Anwendungsfall bei der Iteration über einen assoziativen Container ist der Zugriff auf Schlüssel und Wert:

```
for (auto& pair : container)
{
  auto& key = pair.first;
  auto& value = pair.second;
  // ...
}
```

Dieses Muster wiederholt sich sehr oft. Entsprechend anfällig ist diese Syntax für Fehler. Seit C++17 kann dies jedoch wesentlich kürzer geschrieben werden:

```
for (auto& [key, value] : container)
{
  // ...
}
```

Dies wird *structured binding* genannt. Dabei verweisen die deklarierten Variablen `key` und `value` auf Komponenten des `pair`, welches über den Iterator der `map` zugreifbar ist. Diese neue Syntax ist dabei für drei Fälle anzuwenden:

1. **Arrays fester Größe:** Die deklarierten Variablen verweisen auf die Elemente im Array in aufsteigender Reihenfolge.
2. **Tuple ähnliche Typen:** Die Spezialisierung des Templates `tuple_element` sowie das Bereitstellen einer `get` Funktion, welche einen Index als Template-Argument akzeptiert, ermöglicht die Verwendung von *structured binding* mit beliebigen Typen. Die deklarierten Variablen haben dabei den Typen, welcher über `tuple_element` definiert wurde und werden mit den Rückgabewerten von `get` initialisiert, wobei Indices in aufsteigender Reihenfolge als Template-Argument dienen. Die Angabe der notwendigen Funktionen und Spezialisierungen bedeutet jedoch einiges an zusätzlicher Arbeit, weswegen es sich in manchen Fällen gar nicht erst lohnt[41, S. 165].
3. **Typen mit allen Daten-Membern in derselben Klasse:** Die deklarierten Variablen werden in Deklarations-Reihenfolge an die Daten-Member gebunden.

Neben der Verwendung in Schleifen dient *structured binding* hauptsächlich für mehrere Rückgabewerte pro Funktion. Obwohl lediglich ein Objekt eines Typen von einer Funktion zurückgegeben werden kann, kann dieses direkt in seine Bestandteile zerlegt und an Variablen gebunden werden, was praktisch der Rückgabe mehrerer Werte entspricht[134]. Ein weiterer nennenswerter Anwendungsfall kommt in Kombination mit einem weiteren Feature aus C++17 zu Tage. So dürfen innerhalb einer `if` oder `switch` Bedingung Variablen deklariert werden, damit diese nur innerhalb des folgenden Blocks zugänglich sind[64]. Bei Funktionen, in welchen ein Rückgabewert oft in einer solchen Bedingung abgefragt wird, ist *structured binding* nützlich[4, S. 59]:

```
if(auto [position, wasInserted] = map.try_emplace(key, value);
    wasInserted)
{
  // ...
}
```

Es ist allerdings nicht trivial, alle Stellen in OGRE zu erkennen, wo effektiv mehrere Rückgabewerte verwendet werden. Ein guter Anhaltspunkt ist jedoch die Initialisierung von Variablen mit den Komponenten `first` und `second` eines `pair`. Es ist wahrscheinlich, dass diese mit *structured binding* ersetzt werden können. Somit wurden in OGRE nur an den Stellen *structured binding* angewendet, wo eine solche Zuweisung durch reguläre Ausdrücke gefunden werden konnte. Wählt man die Namen der Variablen gleich, so muss am Rest des Quellcodes nichts geändert werden.

Eine Ausnahme der Regel gibt es jedoch: In manchen Fällen wurde eine explizite Kopie angefertigt, wo die Variablen in *structured binding* lediglich referenzieren. Diese Kopien sind stellenweise beizubehalten, in manchen Fällen können sie aber auch eingespart werden. Zudem ist es notwendig, allen Komponenten des zerlegten Objektes einen Namen zu geben, auch wenn diese im darauf folgenden Kontext nicht verwendet werden. Es ist darüber hinaus nicht möglich, nur eine Variable mit `const` zu deklarieren. Beispielsweise wenn der Schlüssel unverändert bleiben muss, aber der Wert verändert werden soll. Diese Kritikpunkte waren bereits schon im ursprünglichen Vorschlag aufgeführt, sind seitdem jedoch nicht angegangen worden. In OGRE gab es einige Fälle, in denen dies relevant gewesen wäre.

7 Plattform-Unabhängigkeit

7.1 Allgemein

Vor Beginn dieser Arbeit war OGRE für mehrere Plattformen ausgelegt. Sowohl mehrere Betriebssysteme als auch mehrere Compiler wurden unterstützt. Um den zu modernisierenden Quellcode zu minimieren wurde diese Eigenschaft zunächst entfernt. Nun gibt es aber solche Modernisierungen, die Plattform-spezifische Implementierungen in OGRE überflüssig machen. Dadurch sinkt der Wartungsaufwand, da sich der Quellcode an weniger Stellen von Plattform zu Plattform unterscheidet. Stattdessen wird in der Standard-Bibliothek oder dem Compiler selbst in Abhängigkeit der jeweiligen Plattform entschieden, wie die Implementierung auszusehen hat. Sollten zu einem späteren Zeitpunkt wieder mehrere Plattformen unterstützt werden, so muss nicht unbedingt alles wiederhergestellt werden, was vorher entfernt wurde.

7.2 Inline

Das Schlüsselwort `inline` existierte bereits vor C++11. Es dient dazu, dem Compiler mitzuteilen, dass eine Funktion nach Möglichkeit nicht aufgerufen, sondern ihr Code in die aufrufende Funktion kopiert werden soll. Dabei ist dies kein Muss. Garantiert ist jedoch die Eigenschaft, dass die Funktion in jeder Translations-Einheit dieselbe Entität ist[63, § 9.2.8]. Um das Kopieren des Funktions-Codes zu erzwingen verwendet OGRE das Macro `OGRE_FORCE_INLINE`, welches in Abhängigkeit des verwendeten Compilers definiert ist:

```
#if OGRE_COMPILER_MIN_VERSION(OGRE_COMPILER_MSVC, 1200)
 #define OGRE_FORCE_INLINE __forceinline
#elif OGRE_COMPILER_MIN_VERSION(OGRE_COMPILER_GNUC, 340)
 #define OGRE_FORCE_INLINE inline __attribute__((always_inline))
#else
 #define OGRE_FORCE_INLINE __inline
#endif
```

Da sich in Abschnitt 6.5 erwiesen hat, dass dieses Macro für die automatisierte Modernisierung problematisch ist, wurde es mit `inline` ersetzt. Dabei gibt es auch bis einschließlich C++23 keinen besseren standardisierten Ersatz. Zumindest mit Clang ist dies aber auch nicht notwendig, da das entstehende Assembly identisch ist. Es ist davon auszugehen, dass alle so markierten Funktionen nach wie vor in ihre jeweiligen Aufrufer kopiert wurden.

7.3 Deprecated

OGRE verwendet das Macro `OGRE_DEPRECATED` um Entitäten als veraltet (deprecated) zu markieren. Die Plattform-Unabhängigkeit erfolgt dabei analog zu Abschnitt 7.2. *Deprecated* impliziert, dass diese Entitäten in Zukunft entfernt werden sollen, wegen Rückwärts-Kompatibilität jedoch noch eine Weile

erhalten bleiben. Die Modernisierung der Engine stellt natürlicherweise eine gute Gelegenheit dar, veraltete Entitäten zu entfernen. Da diese Gelegenheit im Rahmen der Arbeit um Kapitel 3 bereits genutzt wurde, entfällt an dieser Stelle das Messen der Änderungen.

Nichtsdestotrotz bestünde seit C++14 die Möglichkeit, anstelle eines Macros das standardisierte Attribut `[[deprecated]]` zu verwenden[6]. Dieses ist nicht nur auf allen Plattformen gleich, es besteht zudem die Möglichkeit, die Warnmeldung über die Verwendung einer veralteten Entität anzupassen. Somit verbessert sich nicht nur die Plattform-Unabhängigkeit, sondern auch die Qualität der Warnmeldungen.

7.4 Nodiscard

Benötigt man lediglich die Nebeneffekte einer Funktion, so kann in der Regel der Rückgabewert gefahrlos ignoriert werden. Den Rückgabewert von Funktionen ohne Nebeneffekte zu ignorieren ist jedoch häufig ein unbemerkter Logik-Fehler. Dies wurde schon länger von verschiedenen Compilern anerkannt. Deswegen gibt es auf unterschiedlichen Plattformen bereits unterschiedliche Attribute, die eine Funktion so markieren, dass ignorierte Rückgabewerte eine Warnung erzeugen. OGRE wechselt mit dem Macro `OGRE_NODISCARD` zwischen den Plattform-spezifischen Attributen, welche diese Funktionalität erfüllen. Seit C++17 gibt es mit `[[nodiscard]]` eine standardisierte Version dieses Attributs. Es kann dabei nicht nur Funktionen, sondern auch ganze Typen markieren. Das heißt, dass jede Funktion, deren Rückgabewert einen solchen Typen hat, eine Warnung erzeugt, wenn der Rückgabewert ignoriert wird[139]. Seit C++20 ist es zudem möglich, die jeweilige Warnmeldung mit einem String-Literal anzupassen[89].

Im Rahmen der Modernisierung wurde `OGRE_NODISCARD` durch `[[nodiscard]]` ersetzt. Darüber hinaus konnten mit Clang-Tidy automatisch Funktionen mit `[[nodiscard]]` markiert werden, die sonst keine Nebeneffekte hätten[74, nodiscard]. Die Umstellung führte direkt zu drei Warnmeldungen in OGRE. Sie informieren darüber, dass der Rückgabewert einer Funktion ignoriert wurde. In der Tat ist der gesamte Aufruf an diese Funktion nicht notwendig, wurde aber zu einem vorherigen Stand noch verwendet.[6]

7.5 Noreturn

Das Attribut `[[noreturn]]` gehört zu den ersten standardisierten Attributen, welche mit C++11 eingeführt wurden. Wie es der Name bereits nahe legt, findet es Einsatz bei Funktionen, die niemals zum Aufrufer zurückkehren. Das umfasst Funktionen, die das Programm beenden oder in jedem Fall eine Exception werfen. Mit dieser Information ist es dem Compiler möglich, den Code weiter zu optimieren[83, § 10]. OGRE definiert ein Macro um zwischen verschiedenen Plattform-spezifischen Attributen zu wählen, welche diese Funktionalität erfüllen. Stattdessen kann nun `[[noreturn]]` verwendet werden. Der Ersatz hat dabei keine Auswirkungen auf das Assembly.

[6] Vor Commit 13a18a8 des OGRE Projektes wurde der Rückgabe-Wert noch in einer Variablen aufsummiert. Da die Summe danach nicht mehr verwendet wurde, konnte auch die Aufsummierung und somit später der Funktionsaufruf entfernt werden[117, C. 13a18a8].

7.6 Fallthrough

In den `case` Abschnitten eines `switch`-Blocks wird meistens das Schlüsselwort `break` verwendet um die jeweils folgenden `case` Abschnitte zu überspringen. Das Auslassen eines `break` ist also meistens nicht gewollt und ein Fehler[52, S. 128]. Dementsprechend bieten Compiler Warnungen für diesen Fall an.[7] Dennoch gibt es Fälle, in denen es gewünscht ist, nach einem `case` Abschnitt direkt den darauf folgenden auszuführen. Warnungen wären hier falsch positiv. Um diese Warnungen zu unterdrücken gibt es seit C++17 das `[[fallthrough]]` Attribut[139]. Ein `switch`-Block kann damit wie folgt aussehen:

```
switch(i)
{
  case 0:
    Function1();
    break; // springt aus dem switch Block heraus
  case 1:
    Function2();
    [[fallthrough]]; // keine Warnmeldung
  default:
    Function3();
    break;
}
```

OGRE verwendet hingegen das Macro `OGRE_FALLTHROUGH`. Dieses löst zu Plattform-spezifischen Attributen auf. Das Assembly bleibt durch das Ersetzen mit dem nun standardisierten Attribut unberührt.

7.7 Byteswap

Das Umkehren der Byte-Reihenfolge ist eine Operation, welche oft nur eine Instruktion benötigt. Sie ist vor Allem für das Laden unterschiedlicher Datei-Formate relevant. Es hängt dabei von der jeweiligen Plattform ab, ob und wie die notwendige Instruktion erzeugt werden kann. OGRE bestimmt mithilfe des Präprozessors in den Funktionen `bswap16`, `bswap32` und `bswap64` wie diese Anweisung für die jeweilige Plattform auszusehen hat. Mit C++23 übernimmt dies nun die Standard-Bibliothek. Im Header `<bit>` findet sich dazu nun die Funktion `byteswap`. Dabei handelt es sich um ein Template, welches integrale Typen voraussetzt[101]. Es ist also nur eine Funktion notwendig statt wie in OGRE drei. In der Tat wird diese Funktion zu denselben Instruktionen aufgelöst, womit das Assembly identisch bleibt.

7.8 Filesystem

Einer der fundamentalen Vorteile der Nutzung einer Engine wie OGRE ist, dass das Laden von Ressourcen-Dateien wie Bildern oder 3D-Modellen vom Dateisystem bereits vorgegeben ist. Auf den verschiedenen Plattformen gestaltet sich dieser Zugriff jedoch unterschiedlich. Der Wunsch nach

[7] Warnungen sind kein obligatorischer Teil des C++ Standards. Ein ISO C++ konformer Compiler bräuchte also keine Warnungen zu unterdrücken, dies ist lediglich empfohlen[63, § 9.12.5].

Vereinheitlichung und Portabilität ist groß und schon seit mindestens 2002 existieren Drittanbieter-Bibliotheken, die hierfür eine Plattform-unabhängige Schnittstelle anbieten[17, § Motivation]. Die Aufnahme in die Standard-Bibliothek geschah allerdings erst mit C++17. Das Design der Schnittstelle orientiert sich dabei an POSIX[63, § 31.12.2.2]. OGRE selbst kapselt Zugriffe auf das Dateisystem in wenigen Quellcode-Dateien ab, wo diese in Abhängigkeit von der Plattform schließlich auf die jeweilig spezifische Weise implementiert sind. Dies kann mit dem Header `<filesystem>` entfallen. Darin enthalten sind die wichtigsten Operationen auf Dateisystemen wie zum Beispiel das Erstellen und Löschen von Ordnern. Darüber hinaus können unterschiedliche Pfadangaben mit / bzw. \ einheitlich gehandhabt werden.

Für die meisten von OGRE benötigten Vorgänge gibt es äquivalente Funktionalität in `<filesystem>`. Bei der Umwandlung konnte jedoch festgestellt werden, dass nicht auf alles Plattform-unabhängig zugegriffen werden kann. So sind beispielsweise versteckte Dateien auf Windows mit einem über `<filesystem>` nicht zugreifbaren Attribut gekennzeichnet. Auch existiert nicht die Möglichkeit, alle Dateien, die einem gewissen Muster entsprechen, direkt zu laden. Stattdessen muss dieses Muster selbst interpretiert werden. Dafür gibt es den `directory_iterator` für die Iteration über jeglichen Inhalt in einem bestimmten Ordner, was den Inhalt von Unterordnern nicht mit einschließt. Wahlweise kann aber auch das mit `recursive_directory_iterator` erreicht werden. Glücklicherweise handelt es sich bei den in OGRE verwendeten Mustern lediglich um Präfixe und Suffixe. Da der Anwendungsfall Präfixe oder Suffixe zu vergleichen sehr häufig vorkommt, sind die Funktionen `starts_with` und `ends_with` seit C++20 Teil der String-Schnittstelle[82]. Mit ihnen ist diese Anforderung leicht umsetzbar.

An vielen Stellen, wo vorher `string` für die Angabe eines Dateinamens diente, eignet sich nun `path` besser. Mit Objekten dieses Typs kann direkt auf wichtige Funktionen des Dateisystems zugegriffen werden. Dort, wo nach wie vor `string` notwendig ist, kann die Funktion `native` dienen um auf die interne `string` Repräsentation von `path` zuzugreifen[63, § 31.12.6.5.6]. Unglücklicherweise gibt es keine standardmäßige Formatierung von `path`[63, § 22.14.6.2]. Wo notwendig muss hier also entweder ein Aufruf an `native` hinzugefügt werden oder aber für Aufrufe an `format` eine benutzerdefinierte Spezialisierung von `formatter`. Abschnitt 9.8 beschreibt dies genauer. Letztlich ist noch der Typ des Zeitstempels des Änderungsdatums einer Datei von `time_t` zu `file_time_type` zu ändern. Sind lediglich Vergleiche unterschiedlicher Zeitstempel gefragt, so stellt die Umwandlung kein Problem dar.

8 Ausführungsgeschwindigkeit und Speicherverbrauch

8.1 Allgemein

Immer wieder kommt es vor, dass sich gewisse Muster im Quellcode wiederholen, die auf die gleiche Art und Weise optimiert werden könnten. Zum Teil erfordert dies die Erweiterung der Sprache oder der Standard-Bibliothek. Durch sie können Kopien großer Objekte eingespart oder effizienter mit Speicher und Instruktionen umgegangen werden. Auch ist es möglich Berechnungen, die sonst während der Laufzeit des Programms durchgeführt werden, bereits zur Kompilierzeit fertigzustellen. Ob dies aber auch in jedem Fall wirklich dazu führt, dass letztendlich das Programm schneller ausgeführt wird, ist im Einzelfall zu klären. Folgende Abschnitte behandeln Modernisierungen, die in Aussicht stellten, die Ausführungsgeschwindigkeit in OGRE zu verbessern oder den Speicherverbrauch zu reduzieren.

8.2 Noexcept

Die Behandlung von Exceptions kann mit unter einiges an Instruktionen und Daten im Programm in Anspruch nehmen. Kann der Compiler jedoch beweisen, dass in keinem Fall eine Exception auftritt, können diese Instruktionen und Daten entfallen. Um diese Optimierung zu ermöglichen führt C++11 das Schlüsselwort `noexcept` ein[1]. Dieses ersetzt die zuvor verwendete `throw()` Spezifikation[44], was auch in der Standard-Bibliothek reflektiert ist[38]. Auch kann jede Funktion mit `noexcept` spezifiziert werden, von welcher bekannt ist, dass sie keine Exceptions wirft. Dies ist seit C++17 auch Teil des Funktions-Typen[85].

Diese Umstellung erfordert jedoch eine Inspektion jeder einzelnen Funktion. Auch ist die freizügige Verwendung von `noexcept` nicht zu empfehlen, da dies spätere Änderungen verkomplizieren könnte[99, S. 93]. Zudem führt das unerwartete Werfen einer Exception in einer `noexcept` Funktion zum spontanen Abbruch des Programms, ohne dass der Fehler in irgendeiner Weise behandelt werden kann[66, S. 1124]. Obwohl `noexcept` für weniger Instruktionen sorgt und somit in der Theorie eine bessere Ausführungszeit erwartet wird, stellt dies in der Praxis eher die Ausnahme dar[66, S. 1134-1142]. Von daher ist die Verwendung von `noexcept` eher als Richtlinie für zukünftige Änderungen zu sehen, insbesondere bei kleineren, häufig aufgerufenen Funktionen[130, § F.6].

Nichtsdestotrotz können mittels Suchens und Ersetzens regulärer Ausdrücke viele dieser kleinen Funktionen mit `noexcept` unter moderatem Aufwand modernisiert werden. Hierzu eignen sich Funktionen die lediglich aus einer einzigen `return` Anweisung bestehen, aber auch Funktionen, deren Namen mit `get` oder `is` beginnen, sind Kandidaten für den `noexcept` Spezifikator. Da sich dadurch der Typ der Funktion ändert, ist es wichtig, wie in Abschnitt 9.5 beschrieben, alle überschreibenden Funktionen mit `override` zu kennzeichnen. Andernfalls kann es passieren, dass eine Funktion, welche vorher eine Funktion in der Basis-Klasse überschrieben hat, dies nicht mehr tut. Clang-Tidy bietet zusätzlich noch eine automatisierte Umstellung von `throw()` nach `noexcept` an[74, B. noexcept].

8.3 Emplace

Vor C++11 gab es keine Möglichkeit, Elemente in einen Container hinzuzufügen, ohne diese zuerst zu konstruieren und dann in den Container zu kopieren. Dies kann mit den Funktionen `emplace` und `emplace_back` vermieden werden. Sie nehmen mit variadischen Templates beliebig viele Argumente beliebigen Typs zur Konstruktion eines Elements entgegen und tun dies schließlich in einem vorher reservierten Speicherblock. Die Idee dabei ist, dass kein temporäres Objekt entstehen muss, das letztlich nach dem Einfügen wieder zerstört wird. Stattdessen entsteht nur jenes Objekt, das tatsächlich eingefügt wird[138, § Summary of Motivation]. Hiervon verspricht man sich Code, welcher gleich performant bis performanter ist als die Alternative mit `insert` bzw. `push_back` und einem temporären Objekt[99, S. 295]. Mit Clang-Tidy kann dabei

```
container.push_back(Element(1, "abc"));
```

in

```
container.emplace_back(1, "abc");
```

umgewandelt werden, `insert` zu `emplace` erfolgt analog[74, B. emplace].

8.4 = default und = delete

Es gibt Fälle, in denen es unerwünscht ist, dass ein Objekt in einer bestimmten Operation verwendet wird, beispielsweise Kopieren. Um dies zu verhindern wurden vor C++11 private *copy-constructors* und *copy-assignment-operators* angelegt. Dadurch wurde jeder Kopier-Versuch zu einem Fehler zur Kompilierzeit. Typen verlieren dabei jedoch ihre trivialen Eigenschaften[66, S. 39f]. Gleiches gilt bei der Deklaration eines Konstruktors: Der sonst implizit angelegte triviale Standard-Konstruktor steht danach nicht mehr zur Verfügung, auch wenn dieser gegebenenfalls effizienter wäre[16]. Seit C++11 kann mit `= default` eine sonst implizit angelegte Funktion explizit angefordert werden, mit `= delete` kann die Verwendung einer unerwünschten Funktion als Fehler markiert werden. Dadurch verliert ein Typ keine trivialen Eigenschaften unnötigerweise.

Clang-Tidy kann dabei leere Konstruktoren und Destruktoren mit `= default` versehen, sowie nicht verwendete private Funktionen erkennen und mit `= delete` markieren[74, B. equals-default, B. equals-delete]. Die danach nicht mehr notwendigen `private` Zugriffs-Spezifikatoren müssen allerdings von Hand entfernt werden. Diese stehen nämlich der Fehler-Diagnostik des Compilers im Weg[99, S. 75f]. Auch können ansonsten leere Definitionen, die jedoch einen Kommentar enthalten, nicht automatisch umgewandelt werden. Dadurch, dass nun gewisse Konstruktoren trivial sind und somit garantiert keinerlei Nebeneffekte haben, erkennt der Compiler mehrere Fälle von unbenutzten Variablen, die daraufhin entfernt werden können.

8.5 Move Semantics

Move semantics ergänzen seit C++11 die schon seit Beginn der Sprache existierenden *copy semantics*. Eine *move*-Operation oder auch Verschiebung ist dazu gedacht, dass der Inhalt eines Objektes, welches nicht mehr benötigt wird, an ein anderes übergeben wird. Dies ist oft performanter als eine

Kopie anzufertigen und anschließend das Original zu zerstören. *Move semantics* existierten als Idee bereits vor C++11 u.a. in `auto_ptr`. Dies war jedoch nicht ausreichend und wies einige Probleme auf, wie zum Beispiel, dass es inkompatibel mit temporären Werten war[49].

Die Wiederverwertung des Inhalts eines nicht mehr benötigten Objektes kommt besonders Typen wie `string` und `vector` zugute, welche dynamischen Speicher verwalten. Es gibt aber auch Typen, die gar nicht erst kopiert werden können wie `unique_ptr`. Es muss bei einem *move* kein neuer Speicher für eine Kopie reserviert werden, sondern der Besitzer des Speichers wechselt vom Original zum Ziel der Verschiebung (vgl. Abbildung 8.1). Insbesondere bei einem `vector` von `string`-Objekten kann dabei viel eingespart werden. Die wohl größten unmittelbaren Vorteile von *move semantics* kommen durch die Änderungen in der Standard-Bibliothek. Da OGRE jedoch bereits C++11 verwendete und somit auf eine Standard-Bibliothek zurückgriff, welche sich *move semantics* bediente, ist dieser Unterschied nicht mehr messbar. Entsprechende Messungen sind aber an anderen Beispielen durchgeführt worden, mit klar positivem Ergebnis für *move semantics*[66, S. 823].

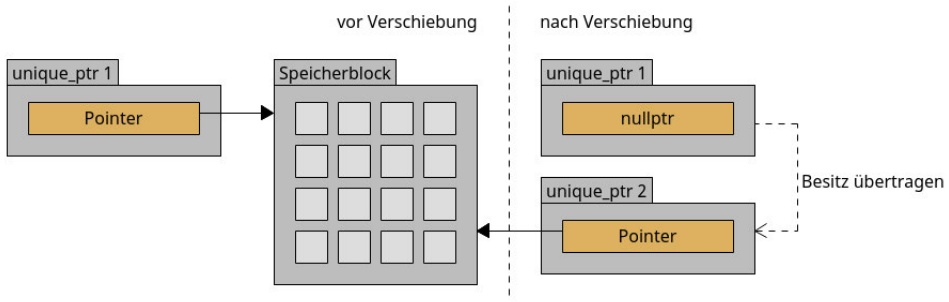

Abbildung 8.1: Verschiebung am Beispiel eines unique_ptr

Auch gestaltet es sich als schwierig, im gesamten Quellcode nach Objekten zu suchen, bei denen ihre letzte Verwendung die Anfertigung einer Kopie ist. Dennoch können an einigen Stellen die Vorteile von *move semantics* genutzt werden: Clang-Tidy kann in Konstruktoren mit Parametern der Form `T const&`, bei denen das jeweilige Argument schließlich kopiert wird, eine Verschiebung generieren. Der Typ des Parameters wird dabei auf `T` reduziert und das Argument wird mit `move` weitergereicht. Jeder Aufruf des Konstruktors führt somit zu einem temporären Objekt des Typs `T`. Wird der Konstruktor mit einem bereits existierenden Objekt aufgerufen, wird nach wie vor eine Kopie angefertigt, welche anschließend mit `move` weitergereicht wird. Ist das Argument jedoch ein temporäres Objekt, so werden stattdessen zwei Verschiebungen angewandt, die im Idealfall günstiger sind als eine Kopie. Dies kommt dadurch zu Stande, dass sowohl *copy constructor* als auch *move constructor* von `T` implizit sind und der Compiler somit den bestmöglichen auswählt. Letztlich wäre diese Umwandlung auch bei jeder anderen Funktion denkbar. Clang-Tidy unterstützt dies jedoch nur für Konstruktoren[74, B. pass-by-value].

Eine automatisierte Umwandlung ist allerdings kritisch abzuwägen, da es nun statt zu einer Kopie zu je einer Kopie und einer Verschiebung oder zu je zwei Verschiebungen kommt, was nicht immer performanter ist. Auch kann es bei einem Parameter-Typen `T`, welcher als Basis-Klasse dient, vorkommen, dass ein Objekt einer erbenden Klasse das Argument ist. Dabei wird lediglich ein neues Objekt der Basis-Klasse erstellt. Jegliche Informationen aus der erbenden Klasse gehen beim sog. *slicing* verloren. Es gilt im Einzelfall zu entscheiden und zu messen, ob sich die Umstellung lohnt[99, S. 291].

8.6 Any

Die Klasse `any` dient dazu, einzelne Objekte beliebiger Typen zu speichern, ohne dass diese in irgendeiner Weise zusammenhängen müssen. So können sowohl `string` als auch `int` ein und derselben Variablen zugewiesen werden. `any` gibt es schon seit längerer Zeit in der Drittanbieter-Bibliothek *Boost*[20]. Mit C++17 ist sie nun ein Teil der Standard-Bibliothek geworden[21]. Auch OGRE hat seine eigene Implementierung in Form von `Any`. Darin wird ein Pointer zu einer Schnittstelle gehalten. Diese wird von einem Template implementiert, das schließlich den konkreten Typen abspeichert. Es unterscheidet sich von `any` darin, dass immer dynamisch Speicher reserviert wird, auch bei kleineren Objekten. `any` hingegen speichert kleinere Objekte lokal[63, § 22.7.4.1]. Dieser Sachverhalt ist in Abbildung 8.2 dargestellt.

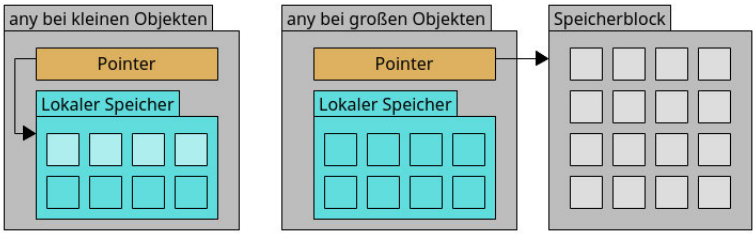

Abbildung 8.2: Speichverwaltung in any bei kleinen und großen Objekten

OGRE setzt zudem auf die Klasse `AnyNumeric`, welche arithmetische Operatoren definiert, sodass diese für numerische Typen durchgeführt werden können, ohne dass der konkrete Typ bekannt sein muss. Diese Klasse erbt jedoch von `Any` und erweitert darin geschachtelte Schnittstellen. Mit dem Ersetzen von `Any` durch `any` ist dies nicht mehr möglich. Stattdessen wird nun ein `any` Objekt als Daten-Member gehalten und dazu eine künstliche Schnittstellen-Tabelle für den gespeicherten Typ angefertigt. Dies ist ein Zeiger auf eine Template-Variable, was seit C++14 möglich ist[33]. Sie wird mit verschiedenen Lambdas initialisiert, die auf das jeweilige Template-Argument zurückgreifen. Zudem wurden designierte Initialisierer aus C++20 angewandt, um die Leserlichkeit zu verbessern[121]. Im Folgenden ist `get` eine Funktion, welche mittels `any_cast` das enthaltene Objekt aus `AnyNumeric` extrahiert.

```
template<typename ValueType>
static VTable const constexpr VTableFor
{ .additionAssignment =
    +[](AnyNumeric& lhs, AnyNumeric const& rhs) noexcept
    {
      get<ValueType>(lhs) += get<ValueType>(rhs);
    },
  // weitere Lambdas...
};
```

8.7 String View

Soll mit einem unveränderlichen String in einer Funktion gearbeitet werden, so hat diese meist einen Parameter des Typs `string const&` oder `char const*`. Wegen der bereits definierten Hilfsfunktionen in der Klasse `string` und der gespeicherten Länge ist eher erstere Möglichkeit vorzuziehen.

Diese hat jedoch den entscheidenden Nachteil, dass wenn der Argument-String von einem anderen Typen besessen wird (z.B. ein `char`-Array), zunächst ein temporäres `string` Objekt erzeugt werden muss. Dieses reserviert gegebenenfalls Speicher nur um ihn am Ende der Operation wieder freizugeben.

Um solche unnötigen Kopien zu vermeiden wurde `string_view` in der einen oder anderen Form schon länger als alternativer Parameter eingesetzt[150, § Overview] und fand mit C++17 auch offiziell in der Standard-Bibliothek Einzug. Dieser Typ referenziert lediglich einen String und besitzt ihn nicht. Dieser String kann dabei sowohl von `string`, einem Array oder einem ganz anderen Typen besessen werden. Dadurch kann es jedoch vorkommen, dass der Besitzer des Strings diesen zerstört noch während er vom `string_view` referenziert wird. Ein entscheidender Vorteil ist, dass das Unterteilen des Strings wesentlich schneller von statten geht: Es müssen lediglich Anfang und Ende des `string_view` angepasst werden, wobei diese in denselben String verweisen. Dadurch können weitere Kopien gespart werden[41, S. 246f].

Allerdings hat dies auch eine große Tücke: Dadurch, dass kein neuer String erzeugt wird, ist das Ende des Strings nicht durch eine binäre Null markiert. Übergibt man dann beispielsweise einen Pointer zum String an eine C Funktion wie `strtoul`, so wird diese nur das ursprüngliche Ende des Strings sehen und nicht das angepasste, da sie weiterhin nach der binären Null sucht[88, § 7.23.1.7]. Diese Problematik ist mit großer Vorsicht zu behandeln. Idealerweise werden solche Fälle von Tests abgefangen oder führen direkt zu gravierenden Fehlern im Programm, welche erkannt und beseitigt werden können.

Eine Modernisierung zu `string_view` umfasst vor Allem Parameter der Typen `string const&`. Die durch die Konvertierung in Abschnitt 8.5 entstandenen Parameter des Typs `string` mit anschließender `move` Anweisung sind ebenfalls Kandidaten, jedoch ohne `move`. Es muss nun nämlich nicht mehr zu temporären `string` Objekten kommen. Alle Vorkommen von `string` durch `string_view` mit einem Schlag zu ersetzen ist nicht zu empfehlen. Die Schnittstelle von `string_view` ist zwar größtenteils an `string` angepasst, aber eben nicht vollständig. So ist beispielsweise die Funktion `c_str` nicht vorhanden. Mit Absicht, denn genau an diesen Stellen kann es zu eben genannten Problemen mit binären Nullen kommen. Ist die abschließende binäre Null jedoch garantiert vorhanden, ist die `data` Funktion ein adäquater Ersatz.

Wurde wie in Abschnitt 9.8 beschrieben schon jeder Aufruf an `printf` durch `format` ersetzt, ist an diesen Stellen die Funktion `c_str` nicht mehr notwendig. An anderen Stellen ist eine Konvertierung zum Typen `path` empfehlenswert, was in Abschnitt 7.8 genauer beschrieben ist. Auch die Funktion `clear` existiert nicht, kann aber durch eine Zuweisung eines leeren Strings ersetzt werden. Weiterhin ist für `string_view` keine Verkettung mit dem + Operator vorgesehen. Verkettungen sind jedoch ohnehin besser durch die flexiblere `format` Funktion zu ersetzen.

Für eine schrittweise Migration eignet sich folgende Strategie: Es wird ein benutzerdefinierter Typ angelegt, welcher von `string_view` erbt. Dieser erweitert die Basis-Klasse um die fehlenden Funktionen, welche die jeweilige Alternative implementieren. Zudem ist dieser Typ implizit von und zu `string` konvertierbar. Hat man im Quellcode dann `string const&` durch diesen Typen ersetzt, kann man schrittweise die Funktionen entfernen und durch die Alternative ersetzen, bis der Typ gänzlich durch `string_view` ersetzt werden kann. Problematisch sind hierbei vor Allem Variablen, deren Typ explizit als `string` angegeben ist. Werden diese beispielsweise mit dem Rückgabewert einer Funkti-

on initialisiert, welche nun einen `string_view` zurück gibt, kommt es zu einem Kompilations-Fehler. Dies ist jedoch nützlich, da sonst stillschweigend mehrere Kopien entstehen würden. In manchen Fällen ist eine Kopie jedoch wünschenswert, was nun eine explizite Konvertierung erfordert, beispielsweise mit geschweiften Klammern statt mit =. In den meisten Fällen ist es jedoch am besten, die Variable ist mit `auto` deklariert – ein Feature, welchem sich Abschnitt 6.4 widmet. Andernfalls kann es nämlich zu einem temporären String kommen, welcher dann später mit einem `string_view` referenziert wird, selbst nachdem es ersteren nicht mehr gibt. Ohne diese Kopie referenziert der `string_view` das ursprüngliche Objekt, was eher noch existiert als eine temporäre Kopie in einer Funktion. Da `string_view` nicht implizit in `string` konvertierbar ist, können durch die Umstellung zudem Funktionen lokalisiert werden, deren Parameter fälschlicherweise `string` oder `string&` ist, wo sich ebenfalls `string_view` anbietet.

In vielen Containern, welche `string` Objekte speichern, kann ein `string_view` mit `emplace` statt `insert` eingefügt werden, womit sich Abschnitt 8.3 befasst. Diente der ursprüngliche `string` jedoch als Schlüssel für einen assoziativen Container, ist gegebenenfalls eine explizite Konvertierung gefragt. Diese kann bei bestimmten Funktionen noch vermieden werden, wenn der Container transparente Vergleicher verwendet, wie Abschnitt 9.3 beschreibt. Hierbei ist für geordnete Container `less<>` und für ungeordnete `equal_to<>` in Kombination mit einer benutzerdefinierten und transparenten Hash-Funktion zu verwenden[63, § 24.2.7.1, § 24.2.8.1]. Für den `[]` Operator muss jedoch immer explizit ein `string` erzeugt werden, denn gegebenenfalls wird dieser direkt im Container gespeichert. Die Funktion `erase` ist seit C++23 ebenfalls transparent[8], dies ist in der verwendeten Version von Clang jedoch noch nicht gegeben, weswegen eine Kombination aus `erase` und der bereits transparenten Funktion `find` Anwendung fand[71]. Zudem ist keine Konvertierung notwendig, wenn man gleich den Schlüssel-Typen in `string_view` umwandelt. Auch hier ist wieder Vorsicht geboten, da der Schlüssel so lediglich referenziert wird und noch während der Lebenszeit des Containers zerstört werden kann. Mit dem AddressSanitizer von Clang können diese Fälle jedoch ausfindig gemacht werden.

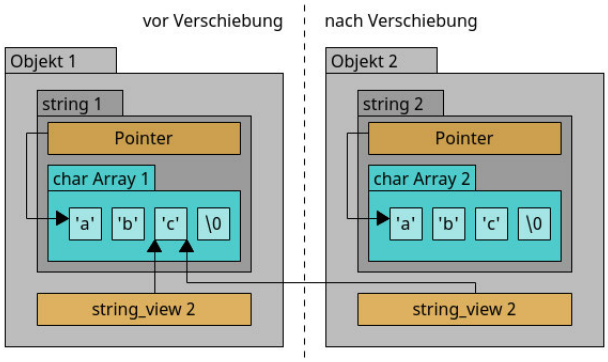

Abbildung 8.3: Beispiel eines ungültigen string_view nach einem Verschieben des lokalen Speichers

Ein weiterer interessanter Fall sind Objekte, in denen sowohl ein `string` als auch Unterabschnitte dessen gespeichert werden. Auf den ersten Blick erscheint die Optimierung verlockend, die Unterabschnitte durch Objekte des Typs `string_view` zu ersetzen und sich somit Kopien zu sparen. Dies ist jedoch unter bestimmten Voraussetzungen inkorrekt. Werden viele dieser Objekte in einem Container wie `vector` gespeichert, so kann es passieren, dass wenn der Container wächst auch seine Elemente verschoben werden. Nun ist es Implementierungen der Standard-Bibliothek erlaubt, kurze Strings innerhalb von `string` zu speichern anstelle innerhalb eines reservierten Speicherblocks. Dies führt

dazu, dass Objekte des Typs `string_view`, welche auf eben diesen lokalen Speicher verweisen, invalidiert werden, sobald der Container wächst. Insofern ist es besser, die Kopien beizubehalten. Diese Problematik ist ist Abbildung 8.3 illustriert.

8.8 Constinit

Freistehende Variablen sowie statische Daten-Member werden zu Programmstart initialisiert. Dies hat nicht nur Konsequenzen auf die Dauer des Programmstarts, sondern kann auch zu nur schwer ersichtlichen Fehlern führen, da die Reihenfolge der Initialisierung unbestimmt ist. Ein Teil dieser Variablen kann mit `constexpr` deklariert werden, womit sie bereits zur Kompilierzeit initialisiert werden. Abschnitt 9.2 befasst sich damit genauer. Dies ist jedoch nicht für alle Variablen möglich. Soll eine Variable zur Laufzeit verändert werden oder ist der Initialisierer getrennt von der Deklaration, entfällt `constexpr` als Möglichkeit. Genau für diesen Zweck wurde das Schlüsselwort `constinit` in C++20 eingeführt. Es erlaubt eine Variable zur Kompilierzeit zu initialisieren, womit eben genannte Probleme gar nicht erst auftreten können. Die Initialisierung ist zu Programmstart bereits abgeschlossen[36].

Kandidaten für `constinit` sind freistehende Variablen und statische Daten-Member, deren Initialisierung in einer anderen Datei stattfindet. Auch statische Variablen innerhalb von Funktionen können davon profitieren. Dafür muss der jeweilige Typ der Variablen jedoch zur Kompilierzeit initialisierbar sein. Für `string` ist dies zum Beispiel nicht der Fall, jedoch kann – sofern der `string` sich nicht später ändern soll – stattdessen `string_view` verwendet werden. Wie in Abschnitt 8.7 beschrieben hat dieser Typ eine hinreichend ähnliche Schnittstelle, was eine Substitution in den meisten Fällen ermöglicht.

9 Fehler-Sicherheit

9.1 Allgemein

Programmierer entdecken immer wieder Fehler gleicher Art. Damit sie zukünftig seltener auftreten, gibt es verschiedene Modernisierungen, welche weniger fehleranfällig sind als ihre älteren Alternativen. Sei es mit Warnungen, Fehlermeldungen oder automatischen Anpassungen: Die in den folgenden Abschnitten beschriebenen Modernisierungen erreichen, dass OGRE robuster gegenüber Fehlern wird, die durch Änderungen am Quellcode verursacht werden können. Nicht zuletzt ist dies auch bei der Fehler-Vermeidung während dem Durchführen der Modernisierungen hilfreich.

9.2 Ersatz für Macros

Macros erlauben es, Vorkommen eines Identifikators durch benutzerdefinierten Text zu ersetzen. Mit Macro-Funktionen können zudem Argumente mitverarbeitet werden. Die Verwendung von Macros wird jedoch als gefährlich eingestuft. Fehlende Klammern, ungewollte mehrfache Ausführung von Ausdrücken sowie fehlende Typ-Sicherheit sind dabei nur die häufigsten von vielen möglichen Fehlerquellen[118, S.128ff][149, S. 546ff]. Es wird in den meisten Situationen davon abgeraten Macros zu verwenden, nicht zuletzt auch, weil sich Analyse-Tools schwer tun, Macros richtig zu behandeln[130, § ES.30, § ES.31]. Sie können durch die bloße Reihenfolge ihrer Definition das Programm ungewollt erheblich verändern.

Für die spätere Migration zu Modulen in Kapitel 10 haben sie weiterhin den Nachteil, dass sie nicht transitiv exportiert werden können. Seit C++11 gibt es immer mehr Alternativen für Macros welche diesen vorzuziehen sind.

> *„[Sie sollten] stattdessen (Inline- oder Template-)Funktionen, Konstanten mit* `const`*,* `constexpr` *und* `enum class` *sowie eigene Typen oder* `using`*/*`typedef` *verwenden. Die Typsicherheit ist größer, die Fehlerdiagnose einfacher und der Code im Normalfall besser verständlich.“*[149, S. 542]

Mit dem Schlüsselwort `constexpr` ist es möglich, Funktionen und Konstanten zu definieren, welche zur Kompilierzeit auswertbar sind. Davor musste zwischen der Sicherheit von Funktionen und der Effizienz von Macros abgewogen werden[32, § 3.2]. Zunächst waren die Beschränkungen auf `constexpr` Funktionen noch hoch. C++14 lockert diese und erlaubt Verzweigungen, Schleifen und Veränderung lokaler Variablen[122]. C++17 führt `if constexpr` ein[86], womit es nun auf Funktionsebene einen adäquaten Ersatz zur Präprozessor-Direktive `#if` gibt. Außerdem können `constexpr` Variablen auf `namespace` Ebene mit `inline` markiert werden, womit es niemals zu mehreren Instanzen ein und derselben Variablen kommen kann[35]. C++20 schließlich lockert die Beschränkungen noch weiter: So können nun virtuelle Funktionen[25], `typeid` und `dynamic_cast`[26], wechselnde `union` Daten-Member[28], das Schreiben in nicht initialisierte Objekte[58] und Container [24] in `constexpr` Funktionen verwendet werden. Darüber hinaus ist es erlaubt, dass nicht evaluierte Assembly-[60] und `catch`-Blöcke[27] vorkommen.

Mit der Klasse `source_location` aus dem gleichnamigen C++20 Header können nun außerdem die Macros `__FILE__`, `__LINE__` und `__func__` vermieden werden. Letztere sind besonders für Fehlermeldungen hilfreich, um den Ursprung des Fehlers im Quellcode schneller ausfindig machen zu können. Für die vereinfachte Anwendung dieser Macros sind dabei in vielen Projekten – auch in OGRE – weitere Macros entstanden, die nun stattdessen eine standardisierte Handhabung erlauben[30]. Diese Macros kamen beispielsweise auch bei der Generierung der Fehlermeldungen aus Abschnitten 2.3 und 2.4 zum Einsatz.

Das Ersetzen dieser Macros durch Funktionen mit `source_location::current()` als Standard-Argument ist recht simpel. Gegebenenfalls müssen die Argumente von einer Funktion zur nächsten überreicht werden, damit die ursprüngliche `source_location` erhalten bleibt. Jedoch ist es nicht mehr möglich, die Text-Repräsentation von Variablen und Ausdrücken in die Fehlermeldungen zu integrieren, was allerdings wegen der genauen Angabe der Fehlerquelle kaum eine Rolle spielen sollte.

Auch die Konvertierung weiterer Macro-Funktionen zu `constexpr`-Funktionen gestaltet sich als relativ einfach. Im Zweifel sind die Parameter der Funktion mit `auto` zu deklarieren, damit es zu keinen unerwünschten impliziten Konvertierungen kommt oder falls mehrere unterschiedliche Typen abgedeckt werden müssen[149, S. 549f].

Macro-Konstanten lassen sich mit Clang-Tidy weitestgehend automatisiert in anonyme Enumerationen umwandeln[74, B. macro-to-enum]. Die wenigen danach übrig gebliebenen Macros müssen schließlich von Hand konvertiert werden. Dies verbessert die Wartbarkeit der Konstanten[130, § Enum.1]. Die anonymen Enumerationen sind dabei implizit zu `int` konvertierbar. In den meisten Fällen reicht dies aus, dass das Programm weiterhin kompiliert.

In wenigen Fällen, welche auf Templates zurückgreifen, kann es jedoch zu Problemen kommen. Ein solcher Fall ist beispielsweise die `min` Funktion aus der Standard-Bibliothek. Dadurch, dass der exakte Typ der Konstanten nun eine anonyme Enumeration anstelle eines simplen `int` ist, ist nicht mehr eindeutig, mit welchem Typen `min` instanziiert werden soll. Hier hilft entweder eine explizite Konvertierung mit `static_cast` oder eine Umwandlung der Enumeration in eine `constexpr inline` Variable des Typs `int`.

Letztlich verbleiben Macros, welche Plattform-spezifische Attribute definieren. Die Abschnitte 7.3, 7.4, 7.5 und 7.6 beschreiben Attribute, welche standardisiert sind und nun keine Macros für Plattform-Unabhängigkeit mehr benötigen. Sind die Attribute nicht standardisiert, gibt es mehrere Optionen: Man behält die Macros bei oder man entfernt sie komplett. In diesem Fall wurde die letztere Option gewählt, wobei keine bedeutende Veränderung des Laufzeitverhaltens festgestellt werden konnte.

Eine dritte Option ist das Verwenden der standardisierten Attribut-Syntax für Compiler-spezifische Attribute. So kann das Clang spezifische Attribut der Form `__attribute__((no_sanitize))` in die standardisierte Form `[[clang::no_sanitize]]` gebracht werden. Mit Compilern, die dieses Attribut nicht unterstützen, soll es ignoriert werden[66, S. 13f]. Insofern könnten alle Attribute ohne Macros verwendet werden, ohne dass es mit unterschiedlichen Compilern zu Problemen käme. Da sich diese Arbeit jedoch explizit auf Clang beschränkt, wurde davon kein Gebrauch gemacht.

9.3 Transparente Funktoren

Transparente Funktoren wurden wegen ihrer Funktion des heterogenen Vergleichs bereits in Abschnitt 5.3 erwähnt. Sie können verhindern, dass ein temporäres Objekt nur wegen einem Vergleich konstruiert wird, was unter Umständen nicht trivial ist. Aber auch homogene Vergleiche können davon profitieren. Ist beispielsweise der Vergleicher-Typ in `map` als

```
std::map<short, Value*, std::less<short>>
```

explizit angegeben, muss der Typ `short` immer zwei Mal abgeändert werden. Sollte man von `short` auf einen größeren Typen wechseln wollen und ändert dabei nur die erste Erwähnung von `short` um, so wird jeder Vergleich mit Argumenten des größeren Typs zuerst auf die Größe von `short` zugeschnitten. Dadurch kann die Sortierung falsch vollzogen werden und bei einem assoziativen Container wie `map` gegebenenfalls sogar Elemente fehlen, eben weil zwei eigentlich unterschiedliche Schlüssel auf `short` verkürzt im Vergleich als derselbe Schlüssel evaluiert werden. Mit

```
std::map<short, Value, std::less<>>
```

bestünde dieses Problem nicht. Lediglich an einer Stelle müsste `short` abgeändert werden[69]. OGRE verwendet an manchen Stellen einen expliziten Vergleicher-Typen. Diese können mit Clang-Tidy automatisch in ihre transparente Form umgewandelt werden[74, B. transparent-functors]. Bis auf die Symbol-Namen bleibt das Assembly dabei gleich.

9.4 Literal-Konstanten

Die Schlüsselwörter `true` und `false` existierten schon in C++98[62, § 2.13.5]. Sie sind Konstanten des Typs `bool`. Nichtsdestotrotz werden auch in OGRE immer noch die Konstanten des Typs `int` `0` und `1` an Stellen verwendet, wo der Typ `bool` gefragt ist. Clang-Tidy kann diese Vorkommen finden und modernisieren[74, B. bool-literals].

Ein ähnlicher Fall ist die Verwendung von `0` als Null-Pointer-Konstante. Es herrscht eine Zweideutigkeit zwischen `0` als Konstante des Typs `int` und als Konstante jedes Pointer-Typen. Problematisch wird dies, wenn es mehrere Funktionen mit unterschiedlichen Parameter-Typen gibt:

```
void Function(int);     // #1
void Function(char*);   // #2
```

Der Aufruf an `Function` mit Argument `0` löst zu #1 auf. Um #2 aufzurufen ist der Typ explizit anzugeben: `(char*)0`. Auch das Macro `NULL` hilft hier nicht weiter[133]. Ein weiterer Problemfall ist die Änderung des Typs einer Variablen von `int*` zu `int&`. Eine Zuweisung mit `0` ist für beide Typen gültig, hat jedoch vollständig unterschiedliche Effekte[S. 100f][66].

Stattdessen kann seit C++11 `nullptr` verwendet werden. Diese Literal-Konstante hat einen einzigartigen Typen der implizit in jeden anderen Pointer konvertiert werden kann. Anders als `0` oder `NULL` ist eine Konvertierung in arithmetische Typen nicht möglich. Auch ist der hergeleitete Typ von `0` und `NULL` in einem Template immer `int`. Dadurch funktioniert die implizite Konvertierung zu einem Pointer nicht, wenn der Typ erst hergeleitet werden muss[111][130, § ES.47]. Durch den einzigartigen

Typen von `nullptr` bleibt diese Eigenschaft jedoch immer erhalten. Auch für diese Umwandlung bietet Clang-Tidy eine automatische Modernisierung an[74, B. nullptr], welche das Assembly nicht beeinflusst.

9.5 Override

Objekt-Orientiertes Programmieren ist in C++ oft mit virtuellen Funktionen verbunden. Dabei deklariert eine Basis-Klasse eine Funktion mit dem Schlüsselwort `virtual`. Von ihr erbende Klassen können dann diese Funktion überschreiben, sofern die Deklaration die gleiche ist. Hier öffnet sich schon die Tür für Fehler: Weicht die Deklaration unabsichtlich ab, wird die Funktion nicht wie gewollt überschrieben. Das Programm kompiliert allerdings fehlerlos. Der Compiler sieht eine zweite Funktion in der erbenden Klasse, unabhängig von der virtuellen Funktion in der Basis-Klasse[99, S. 81f].

```
class Base
{
  virtual void Function1(std::uint16_t);
  virtual void Function2(int);
};

class Derived1 : Base
{
  virtual void Function1(std::int16_t); // Anderer Parameter
  virtual void Funciton2(int);          // Tippfehler
};
```

Damit diese Fehler schon zur Kompilierzeit entdeckt werden können, wurden mit C++11 die Kontext-gebundenen Schlüsselwörter `override` und `final` eingeführt[84][144]. Diese erlauben es dem Programmierer dem Compiler explizit den Wunsch auszudrücken, dass eine Funktion in der erbenden Klasse eine Funktion in der Basis-Klasse überschreiben soll.

```
class Derived1 : Base
{
  void Function1(std::uint16_t) override;
  void Function2(int) final;
};
```

Wird keine passende Deklaration in der Basis-Klasse gefunden oder ist diese Deklaration bereits mit `final` markiert, ist das Programm ungültig.

Obwohl sich OGRE vieler virtueller Funktionen bedient, findet `override` nicht überall Anwendung. Clang-Tidy kann dies zwar nachträglich hinzufügen, dies verhindert aber lediglich Fehler in der Zukunft. Auf diesem Weg können keine Funktionen gefunden werden, die eigentlich eine andere Funktion überschreiben sollen, es aber nicht tun. In diesen Fällen kommt es nicht einmal zu einer Warnmeldung[66, S. 106]. `override` hat dabei keine Auswirkungen auf das Assembly.

9.6 Smart Pointer

Smart Pointer sind Typen, die die Lebensdauer eines anderen Objektes verwalten, auf welches sie einen Pointer haben. Es gab `auto_ptr` schon vor C++11, dieses Template war allerdings problematisch und wurde in C++11 von `unique_ptr` abgelöst. Ausschlaggebend waren dabei die neu hinzugekommenen *move semantics*[50, § 20.4.5]. Ein `unique_ptr` *besitzt* ein Objekt im dynamischen Speicher und zerstört dieses, sobald es selbst zerstört wird. Es kann nicht kopiert werden, aber mit *move semantics* ist es möglich, den Besitz weiterzugeben. Für Anwendungsfälle, in denen der Besitz eines Objektes geteilt werden muss, wurden zudem `shared_ptr` und `weak_ptr` eingeführt. Ein `weak_ptr` erlaubt keinen direkten Zugriff auf das verwaltete Objekt, kann aber einen `shared_ptr` kreieren. Erst nachdem alle aktiven `shared_ptr` zerstört wurden, die sich dasselbe Objekt teilen, wird auch dieses zerstört[22].

Ein entscheidender Vorteil von *Smart Pointern* ist, dass keine expliziten Aufrufe an `delete` notwendig sind. Wie in Abschnitt 2.6 bereits aufgezeigt wurde, vermeidet dies Memory-Leaks. Es ist sicherzustellen, dass es zu jedem `new` auch genau ein `delete` gibt[130, § R.11]. Dies wird erst durch die Funktionen `make_unique` (seit C++14) und `make_shared` möglich[23, § IV][70]. Unter Angabe des gewünschten Objekt-Typen und der Konstruktor-Argumente geben sie einen entsprechenden *Smart Pointer* zurück. Clang-Tidy kann Aufrufe an `new` durch einen Aufruf an die entsprechende Funktion ersetzen[74, B. make-unique, B. make-shared]. Diese Umwandlung machte in OGRE lediglich einen `?:` Ausdruck ungültig, da sich nun die Typen der letzten beiden Unterausdrücke unterschieden. Eine Umstellung nach `if` und `else` reicht aber, um den Code wieder gültig zu machen.

OGRE verwendete bereits an einigen Stellen *Smart Pointer*. Nichtsdestotrotz gab es noch mehrere hunderte Aufrufe jeweils an `new` und `delete`. Sie alle in *Smart Pointer* beziehungsweise eine `make`-Funktion umzuwandeln übersteigt den Rahmen dieser Arbeit. Für Demonstrations-Zwecke wurden jedoch all diejenigen Aufrufe an `delete` umgewandelt, welche unmittelbar in einem Destruktor auftraten. Hierdurch ist der Besitz eines Objektes sofort klar und muss nicht erst durch aufwendiges Studieren des Quellcodes ermittelt werden.

Zu aller erst muss dabei festgehalten werden, dass die Verwendung von `default_delete`, dem Standard-Argument von `unique_ptr`, mit welchem das besessene Objekt zerstört wird, vollständige Typen voraussetzt[63, § 20.3.1.3.3]. Das heißt, dass neben dem Header `<memory>` auch der jeweilige Header des dynamischen Objektes eingeschlossen werden muss. Hierdurch kam es in einem Fall zu einem zyklischen Einschluss zweier Header. Da der eine jedoch nicht auf die Definition des anderen angewiesen ist, konnte diese leicht behoben werden.

Alternativ ist es möglich, die speziellen Member-Funktionen erst zu deklarieren und schließlich dort zu definieren, wo der Typ vollständig ist. Dies ist ratsam, um die Kompilationsdauer zu reduzieren[99, S. 150ff]. Um die spätere Umsetzung der Module in Kapitel 10 durch möglichst wenige zyklische Abhängigkeiten zu vereinfachen, wurde sich für die erste Alternative entschieden.

In einem anderen Fall war der Destruktor des dynamischen Objektes privat und wurde vorher über eine `friend` Deklaration zugänglich. Hier muss nun `default_delete` zum `friend` deklariert werden. Um nicht jeder Instanz von `default_delete` Zugriff zu gestatten, kann stattdessen ein benutzerdefinierter Typ explizit als zweites Template-Argument in `unique_ptr` angegeben werden. Dieser ist dann statt `default_delete` zum `friend` deklariert.

Durch das Wegfallen des expliziten Aufrufs an `delete` im Destruktor können letztere in einzelnen Fällen weggelassen oder mit `default` definiert werden. Dabei gilt es jedoch darauf zu achten, dass die Reihenfolge der `delete` Aufrufe gleich bleibt. Verlässt man sich nämlich auf die implizite Reihenfolge der `unique_ptr` Daten-Member (die umgekehrte Initialisierungs-Reihenfolge[63, § 11.4.7]), so kann es zu Speicherzugriffsfehlern kommen, weicht diese von der ursprünglich expliziten Reihenfolge im Destruktor ab. Dies muss schließlich dadurch behoben werden, dass die Reihenfolge der Daten-Member an diese ursprüngliche Reihenfolge angepasst wird, sodass die `delete` Aufrufe ordnungsgemäß bleiben.

Ein `unique_ptr` ist nicht implizit in einen entsprechenden Pointer konvertierbar. An Stellen, wo ein Pointer erwartet wird, ist es daher notwendig, die Funktion `get` über das `unique_ptr` Objekt aufzurufen, um den Pointer zu erhalten. Dies trat vor Allem bei Aufrufen an benutzerdefinierte Vergleicher-Funktionen auf, welche je zwei Pointer erwarten. Die Funktion `find` ist dabei ein besonderer Fall. In einem Container von `unique_ptr` kann damit nämlich nicht mehr ein bestimmter Pointer gefunden werden. Stattdessen ist es notwendig `find` mit `find_if` und einem benutzerdefinierten Prädikat zu ersetzen. Genau hierfür sind die ebenfalls mit C++11 eingeführten Lambda-Ausdrücke ideal[57, § 2]:

```
std::find(owner.begin(), owner.end(), pointer);
// wird zu
std::find_if
( owner.begin(), owner.end(),
  [pointer](auto const& unique) -> bool
  {
    return pointer == unique.get();
  }
);
```

Gelegentlich ist statt `.get()` aber auch `.release()` erforderlich. Dies ist dann der Fall, wenn im Anschluss der `unique_ptr` zerstört wird. Mit `.get()` würde der resultierende Pointer danach auf gelöschten Speicher verweisen. Nach `.release()` aber besitzt der `unique_ptr` das Objekt nicht mehr und der Pointer bleibt valide[63, § 20.3.1.3.6]. Zuweisungen an den `unique_ptr` mit = sind entweder mit `reset` oder `make_unique` zu ersetzen: Je nachdem, ob der jeweilige Pointer bereits existiert oder erst mit `new` erzeugt wird.

Eine weitere Inkompatibilität kommt bei der Verwendung von `insert` Funktionen zustande, wird diese auf einen Container angewandt, der seine Elemente nicht besitzt. So kompiliert

```
borrowContainer.insert
( borrowContainer.end(),
  owner.begin(), owner.end()
);
```

nicht mehr, nachdem die Elemente des Besitzers in `unique_ptr` abgeändert wurden. Der Algorithmus versucht `unique_ptr` in den Leih-Container aus einfachen Pointern einzufügen. Mangels impliziter Konvertierung von `unique_ptr` zum verwalteten Pointer schlägt dies fehl. Stattdessen muss ein anderer Algorithmus zum Einsatz kommen um die Konvertierung zu ermöglichen[63, § 27.7.4]:

```
std::transform
( owner.begin(), owner.end(),
  std::inserter(borrowContainer, borrowContainer.end()),
  std::mem_fn(&std::unique_ptr<Element>::get)
);
```

Jedes Element des Besitzers wird mit der `get` Funktion durch `mem_fn` umgewandelt, wodurch der Pointer erhalten wird[63, § 22.10.16]. Danach wird dieser mittels der Funktion `insert` in den Leih-Container eingefügt[63, § 25.5.2].

In einem in OGRE definierten Sortier-Algorithmus kommt es zum Kopieren der einzelnen Elemente. `unique_ptr` ist allerdings nicht kopierbar. Um dies zu Umgehen muss der Algorithmus mit `move` (für einzelne Objekte) sowie `move_iterator` (für Iteratoren) versehen werden. Auch in der Standard-Bibliothek mussten Sortier-Algorithmen nach C++11 an Typen angepasst werden, welche nicht kopierbar sind[51, § 25.3]. Die Vorteile der damit verbundenen *move semantics* sind in Abschnitt 8.5 geschildert.

Die Umstellung von einfachen Pointern zu `unique_ptr` bringt zudem einige riskante Code-Abschnitte zum Vorschein, welche diesem Muster folgen:

```
auto* p = new Element;
owner.push_back(p);
```

Es kann vorkommen, dass `push_back` mehr Speicher reservieren muss, bevor das Element eingefügt werden kann. Dabei ist es möglich, dass eine Exception geworfen wird. Der Container, welcher eigentlich das Element besitzen soll, hat darauf an dieser Stelle jedoch noch keine Referenz und kann den Pointer nicht ordnungsgemäß entsorgen. Es entsteht ein Memory-Leak. Mit `unique_ptr` müsste an dieser Stelle zwar `emplace_back` verwendet werden, das Problem bleibt aber das Gleiche[99, S. 297f]. Stattdessen ist es notwendig, den Pointer unmittelbar einem Besitzer zuzuweisen[130, § R.12]. Es bietet sich also `make_unique` mit folgender Umstellung an:

```
owner.push_back(std::make_unique<Element>());
auto* p = owner.back().get();
```

9.7 Ersatz für Unions

Um am selben Speicherort Objekte unterschiedlicher Typen zu speichern, sieht C++ `union` Typen vor. Dabei kann jedoch immer nur ein Objekt den Speicher belegen. Bis auf gewisse Ausnahmen, wenn die Typen Standard-Layout aufweisen, stellt der Zugriff auf ein inaktives Objekt UB dar[63, § 11.5.1]. Dennoch nutzt OGRE die Eigenschaft, dass unterschiedliche Typen über dieselbe Adresse zugreifbar sind, um die Bits eines Typen als einen anderen zu interpretieren. Vor C++20 wäre die beste Alternative eine Kombination aus `aligned_storage` und `memcpy` gewesen. Das ist jedoch fehleranfällig und kann nicht in konstanten Ausdrücken verwendet werden. Dem schafft `bit_cast` Abhilfe. Diese spezielle Funktion konvertiert ein Objekt eines Typen in ein Objekt eines gleich großen anderen Typen, solange beide Typen trivial kopierbar sind[7]. Somit können bestimmte `union` Typen weggelassen und durch `bit_cast` ersetzt werden. Dadurch wird nun UB zu wohl definiertem Verhalten.

Ein anderer Anwendungsfall einer `union` beinhaltet das Notieren des aktiven Typen in einer zusätzlichen Variablen, beispielsweise einer Enumeration oder einem Index. Auch diese manuelle Buchhaltung ist anfällig für Fehler und erfordert oft langatmige `switch` Blöcke um von der Enumeration zurück auf den Typen zu kommen. Schlägt die Buchhaltung dabei an einer Stelle fehl, kommt es zu unsicheren Speicherzugriffen.

C++17 führt mit `variant` eine sichere Alternative zur `union` ein. Solche Alternativen gab es allerdings auch schon vorher in Drittanbieter-Bibliotheken. Bei einer `variant` werden alle möglichen Typen als Template-Argumente explizit angegeben. Intern hält ein Index den jeweils aktiven Typen fest[104]. Anstelle eines `switch` Blocks kann die Funktion `visit` auf das `variant` Objekt angewandt werden. Das jeweils erste Argument dabei ist in der Regel ein Objekt mit je einem `operator()` für jeden möglichen Typen. Durch `visit` wird die zum aktiven Objekt zugehörige Funktion aufgerufen, was einen `switch` obsolet macht. In der Tat kann sich der notwendige Code erheblich reduzieren, wenn der Code der einzelnen `switch`-Fälle hinreichend ähnlich ist:

```
switch(type)
{
  case Int:
    Function(union_value.int_value);
    break;
  case Float:
    Function(union_value.float_value);
    break;
  // ... mehr nach dem gleichen Muster
}
```

Dies kann mit einem einzigen Lambda-Ausdruck ersetzt werden:

```
std::visit
( [](auto const& value){ Function(value); },
  variant_value
);
```

Ein Hindernis bei der Umstellung auf `variant` sind Fälle, in denen mehrere Werte der Enumeration ein und demselben Objekt zugeordnet sind. Eine Übertragung eins zu eins nach `variant` ist so nicht möglich. Stattdessen kann der ursprüngliche Typ, welchem mehrere Enumerations-Werte zugewiesen sind, als Basis-Klasse für neue Typen dienen. Diese definieren dabei keine neuen Member. Dadurch wird es möglich, in `variant` die einzelnen Fälle anhand des Typs zu unterscheiden. Die Basis-Klasse kann dabei weiterhin benutzt werden, um Code-Wiederholungen zu vermeiden. Vor dem Einfügen in die `variant` kann dann schließlich `bit_cast` diese in die jeweilige neu definierte Klasse umwandeln. Hierdurch entfallen dann die vorher notwendigen Enumerationen. Im Einzelfall kann schließlich noch die `index` Funktion von `variant` helfen, den aktuellen Typ auf eine Zahl abzubilden.

Es konnte beobachtet werden, dass `variant` gegebenenfalls mehr Speicher einnimmt, da bei einer von Hand optimierten `union` Padding besser vermieden werden kann als bei der generischen `variant`. Abbildung 9.1 zeigt das Speicher-Layout einer `union`, die entweder ein Objekt mit drei

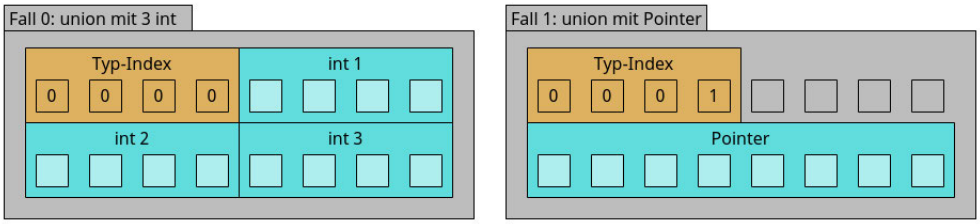

Abbildung 9.1: Speicher-Layout einer union mit zwei möglichen Typen

`int` oder ein Objekt mit einem Pointer enthält. Dabei wird in `Typ-Index` festgehalten, um welchen der beiden Typen es sich handelt. Der `Typ-Index` wurde in beiden Typen an erster Stelle in ihr jeweiliges Layout eingefügt. Standard-Layout erlaubt so immer das Lesen dieses Wertes[63, § 11.4.1]. Abbildung 9.2 zeigt hingegen, wie sich das Speicher-Layout mit `variant` verhält. Dadurch, dass ein möglicher Typ ein Pointer ist, wird die Ausrichtung der geschachtelten `union` auf 8 Bytes festgelegt. Die Bytes zwischen `Typ-Index` und der `union` sowie zwischen der `union` und dem Ende des `variant` Objektes werden dann mit Padding aufgefüllt, um die Ausrichtung zu gewährleisten. Insgesamt wächst diese `variant` also um 8 Bytes im Vergleich zur `union` davor.

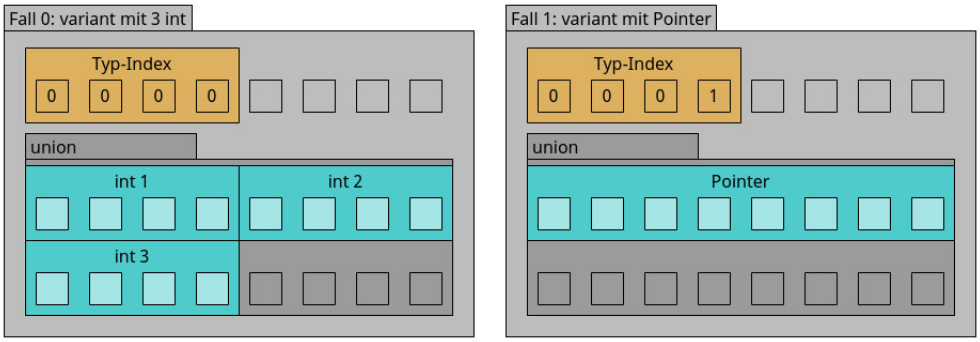

Abbildung 9.2: Speicher-Layout einer variant mit zwei möglichen Typen

9.8 Format

Oft kann es notwendig sein, die Werte von Variablen in Text zu integrieren, sei es für die Benutzeroberfläche oder für Fehlerausgaben. Vor C++20 gab es dafür zwei Methoden im Standard: `printf` und I/O Streams. OGRE verwendete eine an `printf` aufbauende Funktion `StringUtil::format`. Ein großer Kritikpunkt an ersterer ist die fehlende Typ-Sicherheit der zu formatierenden Argumente[118, S. 118]. Diese werden nämlich an einen Ellipsen-Parameter gebunden, wobei jegliche dem Compiler bekannte Typ-Informationen verloren gehen. In der Zeichenkette, welche das Format angibt, werden dann schließlich bestimmte Format-Spezifikatoren angegeben, die den Typen repräsentieren. Weiterhin gestaltet sich die Formatierung benutzerdefinierter Typen sowie die Verwaltung des notwendigen Speichers als schwierig, sodass es oft zu Fehlern kommt. Dies ist mit I/O Streams zwar bereits besser, damit lässt sich die dargestellte Reihenfolge der zu formatierenden Argumente jedoch nicht abändern, was bei einer Übersetzung in eine andere Sprache durchaus erwünscht ist.

Diese Probleme soll der Header <format> mit C++20 lösen. Die Format-Syntax ist dabei ähnlich zu printf aufgebaut, geht dabei jedoch sicher mit Typen um und verzichtet auf deswegen obsolete Spezifikatoren, was die Format-Zeichenkette vereinfacht. Die Platzhalter sind dabei für jeden Typen {}, mit optionalen Spezifikatoren innerhalb der geschweiften Klammern. So steht eine Zahl für den Index des übergebenen Arguments[152].

Zusätzliche Typ-Sicherheit wird in C++23 dadurch erreicht, dass – falls möglich – die Format-Zeichenkette bereits zur Kompilierzeit mit den Typen der Argumente abgeglichen wird. Hierfür ist ein besonderes Template vorgesehen, welches nicht außerhalb der Standard-Bibliothek zur Verfügung steht[153, § 5]. Dies hat jedoch zur Folge, dass die Funktion StringUtil::format nicht einfach an format aus der Standard-Bibliothek weiterleiten kann. Letztere muss direkt im Code verwendet werden. Mit einer einfachen Ersetzung im Quellcode von OGRE ist es jedoch noch nicht getan, denn die Format-Spezifikatoren müssen, falls vorhanden, angepasst werden. In vielen Fällen ist es jedoch ausreichend, lediglich {} zu verwenden. Es kommt allerdings auch vor, dass zusätzliche Informationen wie darzustellende Länge im Format enthalten sind. So sind die beiden Ausdrücke

```
StringUtil::format
( "line %d: %s: '%.*s'\n",
  line, error, length, token
);
std::format
( "line {0}: {1}: '{3:.{2}}'\n",
  line, error, length, token
);
```

gleichbedeutend, nur dass letztere Variante eine bessere Typ-Sicherheit hat. Die Umwandlung erfordert dabei Kenntnisse der Format-Spezifikatoren von printf und format. Zudem müssen die Zeichen { und } jeweils zu {{ und }} erweitert werden, damit diese weiterhin in der formatierten Zeichenkette enthalten sind und nicht als Format-Spezifikatoren interpretiert werden.[8]

Zusätzlich zu dem Ersetzen printf-ähnlicher Funktionen kann format als Ersatz zu Verknüpfungen von Zeichenketten mit + dienen. Dabei wird nämlich im schlimmsten Fall pro + neuer Speicher für die resultierende Zeichenkette reserviert. Auch müssen Zahlen vor dem Aufruf an den + Operator in Zeichenketten umgewandelt werden, was ebenfalls eine Speicherreservierung benötigt. Hierfür dienten in OGRE überladene Hilfsfunktionen mit dem Namen StringConverter::toString.

Mit format können die Argumente mit einem Funktionsaufruf verknüpft und somit im besten Fall einige Speicherreservierungen eingespart werden. Für die Umwandlungen können reguläre Ausdrücke dienen, wobei nach Mustern wie + „ oder + StringConverter::toString gesucht wird. Das Weglassen der Funktion StringConverter::toString führt in einigen Fällen dazu, dass die Kompilation fehlschlägt. Grund hierfür ist, dass nun für einige benutzerdefinierte Typen unbekannt ist, wie diese in eine Zeichenkette umzuwandeln sind. Insbesondere für Enumerationen, welche vorher implizit in integrale Typen konvertiert wurden, es aufgrund der Typ-Sicherheit jedoch nicht mehr werden, ist eine Definition gefragt.

[8] Ein Ausdruck der Form std::format(„<{}>", arg) kompiliert mit der verwendeten Version von Clang nicht. Unter MSVC v19.32 produziert dies jedoch keinen Fehler. GCC unterstützt <format> noch nicht. Es wird davon ausgegangen, dass es sich hier um einen Fehler in Clang handelt. Da hier die spitzen Klammern lediglich der Hervorhebung des Arguments dienten, wurden diese mit einfachen Anführungszeichen ersetzt.

Für diesen Zweck gibt es das Template `formatter`. Es kann für benutzerdefinierte Typen spezialisiert werden, wobei die Funktionen `format` und `parse` existieren müssen[63, § 22.14.5]. Um für die vielen verschiedenen Enumerations-Typen in OGRE das gleiche Verhalten wie vorher zu gewährleisten empfiehlt sich folgende Spezialisierung:

```
template<typename T, typename CharT>
requires
  std::is_enum_v<T>
struct std::formatter<T, CharT>
{
  std::formatter<std::underlying_type_t<T>, CharT> underlying;

  auto constexpr parse(auto& pc)
  {
    return underlying.parse(pc);
  }

  auto constexpr format(T t, auto& fc)
  {
    return underlying.format(std::to_underlying(t), fc);
  }
};
```

Hier kommen gleich mehrere Features aus neueren C++ Standards zum Einsatz. Dank mit `requires` beschriebenem Constraint (C++20) greift diese Spezialisierung nur für Enumerationen, da nur für diese `is_enum_v` (C++17) wahr ist[81, § TT.2.3][92][137, § 17.5.4, § 17.10.2]. `underlying_type_t` (C++14) extrahiert den der Enumeration zugrunde liegenden integralen Typen[10][19]. Damit kann schließlich dessen `formatter` Spezialisierung verwendet werden. Mit `to_underlying` (C++23) wird der übergebene Enumerations-Wert schließlich in den zugrundeliegenden Typen konvertiert[90, § 3]. Da die Funktionen mit `constexpr` markiert sind (was für diese Funktionen erst seit C++14 gelten kann[122, § Clause 7]), ist es zudem möglich, die Verifikation des Formats noch während der Kompilierzeit durchzuführen.

9.9 Typ-sichere Enumerationen

Enumerationen sind nützlich, um zusammengehörige Konstanten zu definieren. Zudem wird mit einer neuen Enumeration auch ein neuer Typ definiert. Jedoch konvertiert dieser implizit zu einem integralen Typen, was die Typ-Sicherheit schmälert und auf unterschiedlichen Plattformen zu unterschiedlicher Größe von Variablen der Enumeration führen kann. Die Namen der Konstanten sind darüber hinaus ohne explizite Angabe der Enumeration sichtbar, was entweder zu Namenskollisionen, langen Präfixen oder mehrdeutigen Abkürzungen dieser Präfixe führt. Einige dieser Nachteile können zwar vermieden werden, allerdings nur, wenn man stellenweise auf die Verwendung des Enumerations-Typen verzichtet und auf explizite integrale Typen zurückgreift oder neue Typen definiert, die die Enumeration umschließen.

Mit C++11 kann für alle Enumerationen ein expliziter integraler Typ angegeben werden, welcher dessen Größe fest definiert. Darüber hinaus gibt es nun eine neue Kategorie von Enumerationen. Statt Schlüsselwort `enum` definiert man diese mit `enum class` oder `enum struct`. Eine solche neue Enumeration konvertiert niemals implizit zu einem integralen Typen. Die Konstanten sind außerdem nur über die explizite Angabe des Enumerations-Typen sichtbar[100].

```
enum Old                        // unbestimmter integraler Typ
{ Old_Enum1, Old_Enum2          // lange Namen
};
int oldValue = Old_Enum1;       // implizite Konvertierung

enum class New : char           // expliziter integraler Typ
{ Enum1, Enum2                  // kurze Namen
};
New newValue = New::Enum1;      // keine Konvertierung
```

Eine Umwandlung des Schlüsselwortes `enum` in `enum class` ist schnell auf ein gesamtes Projekt angewandt. Damit ist es jedoch noch nicht getan, wenn auf der impliziten Konvertierung aufgebaut oder sogar stattdessen gleich ein Parameter integralen Typs genommen wurde. Zunächst sind die Vorkommen integraler Typen als Funktions-Parameter oder -Rückgabe-Typ sowie Daten-Member ausfindig zu machen, welche stellvertretend für eine bestimmte Enumeration stehen. Sie sind nun durch die eigentliche Enumeration zu ersetzen. Hierbei wird es gegebenenfalls nötig, neue Header-Dateien einzuschließen. Dadurch, dass die Namen der Enumerations-Konstanten nicht mehr von außen aus sichtbar sind, ist es nicht mehr notwendig, dass eine Enumeration innerhalb einer Klasse definiert wird. Somit können etwaige zyklische Abhängigkeiten durch geschickte Umplatzierung der jeweiligen Enumeration vermieden werden. Im Regelfall haben diese nämlich keine Abhängigkeiten, bis auf den zugrundeliegenden integralen Typen. Dieser kann nun explizit als derjenige Typ angegeben werden, der bereits als Platzhalter in Parametern und Daten-Membern diente. Wo diese vorher mit integralen Konstanten wie `0` oder `0xFF` initialisiert wurden, schlägt dies nun gegebenenfalls fehl. Es sind explizite Konvertierungen mit `static_cast` gefragt. Jedoch ist es seit C++17 auch möglich, eine solche Initialisierung mit geschweiften Klammern zu vollziehen[34, § 1].

Bei dieser Umwandlung gibt es zwei große Ausnahmen. So kann ein integraler Typ nicht nur als Platzhalter für eine Enumeration dienen, sondern auch für mehrere. Dabei kann sowohl der alte Platzhalter beibehalten werden, ein neuer expliziter Platzhalter-Typ definiert werden oder aber man kann `variant` verwenden, wie Abschnitt 9.7 beschreibt. Der nächste Ausnahmefall tritt auf, wenn eine Enumeration nur dazu dient, die Konstanten einer zweiten zu ergänzen oder diese unter anderen Namen verfügbar zu machen. Wo vorher beide Enumerationen implizit in denselben integralen Typen konvertiert wurden, gäbe es nun zwei unterschiedliche Typen. Stattdessen können die neuen Konstanten als statische Daten-Member innerhalb einer leeren Klasse mit dem Typ der ursprünglichen Enumeration definiert werden. Dies gewährleistet Typ-Sicherheit und Interoperabilität der unterschiedlichen Konstanten.

Dadurch, dass die Konstanten einer Enumeration nur noch unter expliziter Angabe ihres Typs sichtbar sind, können sie nach der Umstellung nicht mehr gefunden werden. Ein erster Ansatz, dies zu beheben, ist es, das jeweilige Namens-Präfix durch den Namen der Enumeration gefolgt von `::` zu ersetzen. Bei Funktionen, welche Enumerationen in Strings umwandeln, ist Vorsicht walten zu

lassen: So gibt es in OGRE nämlich Dateien, welche zur Laufzeit geladen werden. In ihnen kann eine Enumerations-Konstante mit ihrem Namen referenziert werden. Die Umbenennung des Namens kann somit zu Fehlschlägen beim Programmstart führen. Namensänderungen müssen also auch mit Dateien synchronisiert werden, welche keinen C++ Code enthalten.

Zu Fehlern kommt es außerdem, wenn mehrere Enumerationen dasselbe Präfix haben. Jedoch muss der Rest des Namens zwangsläufig einzigartig sein, da es bei gleichem Präfix zu keinen Namenskollisionen gekommen war. Die neue Typ-Sicherheit verhindert außerdem die versehentliche Verwechslung zweier gleichnamiger Konstanten aus unterschiedlichen Enumerationen. Insofern ist es nicht möglich, das Programm zu kompilieren ohne diese Fehler vorher behoben zu haben.

Ein weiteres Problem sind Namenskollisionen der nun gekürzten Konstanten-Namen mit Macros. So kollidierte bei der Umstellung in OGRE das Macro `NULL` mit einem so gekürzten Konstanten-Namen. Es ist generell vorzuziehen, dass nur Macros in Großbuchstaben geschrieben werden[130, § Enum.5, § ES.32]. Von daher wurde diese Konstante in `Null` umbenannt.

Nun gibt es allerdings Fälle, in denen der vollständige Name der Enumerations-Konstante sehr lang ist, beispielsweise wenn diese innerhalb einer Klasse definiert ist und sich diese Klasse in einem anderen `namespace` befindet. Insbesondere bei einem `switch` über alle Werte einer Enumeration wird dies schnell unübersichtlich. Seit C++20 können jedoch einzelne oder gleich alle Namen aus einer Enumeration auf einmal sichtbar gemacht werden, wodurch die Notwendigkeit den vollständigen Namen anzugeben entfällt. Hierfür dient `using` für einzelne Namen und `using enum` für alle Namen in einer bestimmten Enumeration[3].

So ist es theoretisch möglich, nach jeder `enum class` einmal `using enum` zu nutzen und wie gewohnt auf die Namen der Enumeration zuzugreifen ohne diese explizit anzugeben. Dadurch geht jedoch gleich wieder einer der Vorteile von Typ-sicheren Enumerationen verloren, weswegen `using enum` in OGRE hauptsächlich in Zusammenhang mit `switch` Anweisungen umgesetzt wurde. Bei diesen konnte nach der Umstellung zudem eine neue Warnmeldung beobachtet werden. Sie weist darauf hin, dass nicht für alle Konstanten der Enumeration ein `case` existiert. In allen Fällen wurde dies jedoch als beabsichtigt eingestuft. Die Warnung ist dabei mit folgendem Einschub in den `switch` Block zu unterdrücken:

```
default:
  break;
```

Die Nutzung der impliziten Konvertierung hat einige weitere Konsequenzen. So werden in OGRE an vielen Stellen Enumerations-Konstanten als Indices für Arrays verwendet. Wo dies mit `enum` implizit möglich war, erfordert `enum class` eine explizite Konvertierung. Dies kann ein `static_cast` sein oder seit C++23 die Funktion `to_underlying`. Letztere ist dabei nicht anfällig für Fehler, wenn der zugrundeliegende Typ der Enumeration abgeändert wird[90].

`to_underlying` ist zudem nützlich für die Definition neuer Funktionen auf Enumerationen. Ist deren Verwendungszweck nämlich eine Bitmask, sind die Operatoren `bitand` (`&`), `bitor` (`|`) und `compl` (`~`) erforderlich, deren Definition wie folgt aussehen kann:

```
auto operator bitor(Enum e1, Enum e2) -> Enum
{ return static_cast<Enum>
  ( std::to_underlying(e1)
  bitor
    std::to_underlying(e2)
  );
}
```

Vorher waren diese nicht erforderlich, da stattdessen die eingebauten Operatoren der integralen Typen genutzt wurden. Die neuen Funktionen stellen jedoch eine Fehlerquelle dar und erhöhen den Wartungsaufwand[66, S. 347]. Zudem wird mit `bitand` oft geprüft, ob ein bestimmtes Bit in der Bitmask gesetzt ist. Mit `enum` konnte dabei das Ergebnis in Abhängigkeiten des Kontexts (z.B. eine `if` Anweisung) in `bool` konvertiert werden, was mit `enum class` nicht mehr möglich ist. Eine Möglichkeit wäre eine explizite Konvertierung zu `bool` oder eine designierte Funktion, welche diese Aufgabe übernimmt. Da jedoch auch der Operator `not` (`!`) an solchen Stellen vorkommt, wurde stattdessen dieser definiert und gegebenenfalls zweimal angewendet. Der erste Aufruf reduziert die Enumeration auf einen Wert des Typs `bool` und negiert diesen. Der zweite Aufruf geht an den eingebauten `not` Operator von `bool` und hebt die Negierung wieder auf. Somit ist `!!(...)` in gewisser Weise die Kurzform von `static_cast<bool>(...)`. Auch ein expliziter Vergleich mit `0` ist nicht mehr direkt möglich. Alternativen sind die Definition eines entsprechenden Vergleichs-Operators oder die Verwendung des Ausdrucks `Enum{}`, dessen Wert ebenfalls `0` entspricht.

10 Module

10.1 Allgemein

Module werden als eine der bedeutendsten Erweiterungen in C++ seit der Standardisierung angesehen. Nicht nur versprechen sie die Kompilationsdauer eines C++ Projektes zu reduzieren, sie eröffnen neue Möglichkeiten zur Abkapselung einzelner Bestandteile der Software und reduzieren den Bedarf des Präprozessors erheblich[118, S. 290f]. Module erreichen dies, indem sie eine Alternative zum herkömmlichen Einschließen von Header-Dateien anbieten: Das Importieren und Exportieren von Modul-Einheiten. Dies hat drei entscheidende Vorteile gegenüber Header-Dateien:

1. Es kann nun explizit ausgewählt werden, welche Deklarationen außerhalb des Moduls und welche ausschließlich innerhalb dessen sichtbar sind. In Headern muss oft mit umständlichen Namen umgangen werden, dass ein Nutzer des Headers auf Details der Implementierung zugreift.
2. Module sind *hermetisch*. Anders als Header, bei denen Deklarationen und Macros außerhalb des Headers seinen Inhalt beeinflussen können, bleibt der Inhalt eines Moduls unabhängig vom Kontext seines Imports.
3. In großen Projekten wurde festgestellt, dass die Kompilationsdauer näherungsweise quadratisch mit dessen Größe anwächst. Jeder Header wird nämlich pro Einschluss in eine Translations-Einheit einmal kompiliert. In Kombination mit dem nicht zu unterdrückenden transitiven Einschluss von Headern resultiert dies schnell in mehrfache Kompilation desselben Headers. Dem sollen Module entgegenwirken indem jede Datei, welche zu einem Modul gehört, selbst eine Translations-Einheit ist, die nur einmal kompiliert werden muss[124, § 2].

Es gibt bis zu vier verschiedene Kategorien von Translations-Einheiten in einem Modul. Notwendig ist in jedem Fall die primäre Modul-Schnittstelle, die nur einmal vorkommen darf. Sie legt fest, was außerhalb des jeweiligen Moduls sichtbar ist. Modul-Implementations-Einheiten dienen der Definition von Deklarationen in der primären Modul-Schnittstelle. Das Verhältnis ist somit ähnlich zu dem von Headern und ihren Implementierungs-Dateien. Jedoch ist es erlaubt, das Modul gänzlich in der primären Modul-Schnittstelle zu definieren. Hierfür dient das sogenannte *private Modul-Fragment*, welches mit der Zeile

```
module :private;
```

eingeleitet wird. Änderungen in dem Bereich darunter beeinflussen Importeure des Moduls nicht[63, § 10.5].

Idealerweise könnte jeder Header in eine primäre Modul-Schnittstelle und dessen Implementierungs-Datei in eine Implementations-Einheit des Moduls umgewandelt werden. Es gilt jedoch zu beachten, dass – bis auf wenige Ausnahmen – jede Deklaration an das Modul gebunden wird, in welchem sie vorkommt. Das heißt, kommen zwei an sich identische Deklarationen in verschiedenen Modulen vor, werden sie auch als zwei verschiedene Deklarationen behandelt. Dies tritt vor allem dann auf, wenn mit Hilfe von Forwärts-Deklarationen zyklische Abhängigkeiten aufgelöst wurden. Die Konsequenz ist, dass alle diese Deklarationen im selben Modul sein müssen wie die letztliche Definition. Die beste Option ist dabei die Verwendung von Modul-Partitionen. Sie ermöglichen die Unterteilung des

Moduls in mehrere Dateien und sind nur innerhalb des Moduls sichtbar. Dabei gibt es die Modul-Schnittstellen-Partitionen, welche von der primären Modul-Schnittstelle zu exportieren sind, und die Modul-Implementations-Partitionen, deren Deklarationen nur innerhalb des Moduls sichtbar sind[125, § 2.1, § 2.2].

Um die Kompatibilität zu Quellcode zu bewahren, welcher nicht auf Module zurückgreift, gibt es das sogenannte *globale Modul-Fragment*. Darin können wie bisher Header eingeschlossen werden, welche dann jedoch nur innerhalb der jeweiligen Translations-Einheit sichtbar sind und nicht wie bisher transitiv eingeschlossen werden. Insbesondere können Macros so keine Importeure des Moduls beeinflussen. Dies gilt auch für die sogenannten *Header-Units*. Dabei kann ein Header dem Compiler so gekennzeichnet werden, dass er ihn kompiliert als wäre er eine primäre Modul-Schnittstelle. Diese exportiert den gesamten Inhalt des Headers als Teil des globalen Modul-Fragments. Macros werden jedoch niemals transitiv exportiert. Module können auch von herkömmlichen Dateien importiert werden[125, § 2.3]. Relevant ist dies für eine schrittweise Migration sowie die notwendige Definition der `main`-Funktion, welche nicht an ein Modul gebunden sein kann, aber durchaus auf importierte Deklarationen zugreifen soll.

10.2 Kompilation mit CMake

Zum Zeitpunkt des Schreibens ist die Kompilation eines größeren Projektes mit Modulen nicht trivial. Nicht nur ist die Umsetzung im verwendeten Compiler nicht vollständig[75], auch CMake, mit welchem OGRE erstellt wird, hat noch keine offizielle Unterstützung für C++ Projekte mit Modulen[4, S. 644].[9] Zum Zwecke dieser Arbeit wurde mit benutzerdefinierten Funktionen in CMake jedoch eine zeitweilige Lösung umgesetzt, welche diesem Anwendungs-Muster folgt:

```
add_module(
  # Datei mit Primaerer Schnittstelle
PARTITION
  # Dateien mit Partitions-Einheiten...
IMPLEMENTATION
  # Dateien mit Implementations-Einheiten...
INCLUDES
 # Pfade zu eingeschlossenen Dateien...
)
```

Unter Angabe aller Dateien, welche zum jeweiligen Modul gehören, wird eine statische Bibliothek generiert. Dabei werden die Dateien in vereinfachter Form analysiert und Informationen wie Modul-Name und Modul-Importe extrahiert. Möglich wird dies dadurch, dass bereits zum Zwecke einer schnellen Analyse der Modul-Abhängigkeiten gewisse Restriktionen auf die Deklaration von Modulen und deren Importe festgelegt wurden[127, § 3.1].

Sowohl der Name des Moduls (eingeleitet mit `module`) als auch seine Importe (eingeleitet mit `import`) müssen in jeweils einer mit Semikolon endenden Zeile vorkommen, welche lediglich mit Leerzeichen oder `export` beginnen darf. Diese Deklarationen dürfen zudem nicht von Macros

[9] Der aktuelle Stand zur Unterstützung von C++20 Modulen in CMake kann unter https://gitlab.kitware.com/cmake/cmake/-/issues/18355 (25.07.2022) verfolgt werden.

abhängig sein und eine `module` Deklaration darf auch nicht aus einer eingeschlossenen Datei stammen[127, § 2]. Somit reicht im Regelfall[10] bereits eine zeilenweise Suche mit einem regulären Ausdruck, um alle relevanten Informationen aus der Datei zu extrahieren.

Für jede Partition und die primäre Modul-Schnittstelle wird mit CMake ein benutzerdefinierter Befehl erstellt[14, B. add_custom_command], der mit Compiler-spezifischen Optionen daraus die Modul-Schnittstelle als Binär-Datei generiert. Im Falle von Clang ist dies eine `.pcm` Datei[78]. Die aus den Dateien gelesenen Importe werden dabei verarbeitet als Abhängigkeiten in CMake angegeben, womit die richtige Reihenfolge beim Erstellen der Module gewährleistet wird. Aus den Implementations-Einheiten wird schließlich die statische Bibliothek zusammengesetzt, wobei jeweils auf notwendige Binär-Dateien der importierten Module verwiesen wird. Diese benutzerdefinierten CMake-Dateien liegen dieser Arbeit bei. Es sei hierbei explizit darauf hingewiesen, dass es sich um keine vollwertige Modul Umsetzung handelt und lediglich zur Überbrückung bis zu einer offiziellen Lösung in CMake dient.

10.3 Einteilung der Dateien in Module

Wie bereits in Abschnitt 10.1 erwähnt, haben Forwärts-Deklarationen zur Folge, dass keine direkte Übersetzung von Headern zu primären Modul-Schnittstellen möglich ist. Auch gibt es Fälle, in denen die Implementierungs-Datei eines Headers einen zweiten Header einschließt, dessen Implementierungs-Datei wiederum den ersten Header einschließt. Solche zyklischen Abhängigkeiten verkomplizieren die Umwandlung zu Modulen und sind im Idealfall zu vermeiden[130, § SF.9]. Die Lösung hierfür sind Modul-Partitionen. Befinden sich die jeweiligen Dateien in einem einzigen Modul, kommt es zu keinen zyklischen Abhängigkeiten zwischen Modulen. Zu klären ist jedoch, wie effizient festgestellt werden kann, welche Dateien in einem Modul zusammengefasst werden sollen.

Ein erster Anhaltspunkt ist die Dokumentation von OGRE. Diese gibt bereits eine Einteilung in „Modules“ vor[116, B. Modules]. Das „Modul“ *Core* besteht dabei beispielsweise aus Untermodulen *Animation*, *General* und anderen. Untermodule sind zwar nicht direkt in der Sprache vorgesehen, da Modul-Namen jedoch neben Buchstaben und Unterstrichen auch Punkte zulassen, bieten sich letztere für die Trennung von Modul-Namen und Untermodul-Namen an.

Nun gibt es in dem Untermodul *General* jedoch eine zentrale Klasse `Root`. Diese hängt von vielen Komponenten aus anderen Untermodulen ab und viele Komponenten aus anderen Untermodulen hängen von `Root` ab. Es handelt sich um einen Fall des *God Class Anti-Patterns*, welches in einem Software-Projekt eher zu vermeiden ist[118, S. 231]. Die Konsequenz ist, dass es keine triviale Aufgabe mehr ist zu ermitteln, welche Dateien in ein Modul gehören.

Für diesen Zweck entstand ein Analyse-Programm, welches dieser Arbeit beiliegt. Es analysiert vom Compiler hinterlassene Dateien, in denen alle Abhängigkeiten einer Translations-Einheit festgehalten werden. Diese Dateien können in den meisten Fällen einem Header und einer Implementierungs-Datei anhand ihres Namens zugeordnet werden. Letztlich werden damit zyklische Abhängigkeiten

[10] Ein Ausnahmefall sind Kommentare, welche zum jetzigen Stand nicht ignoriert werden. Es konnten in OGRE jedoch auch keine Kommentare gefunden werden, die fälschlicherweise als ein Modul-Name oder -Import interpretiert wurden.

identifiziert. Unter Angabe einer Datei (z.B. `OgreRoot.hpp`) werden dann alle Dateien in einem bestimmten Ordner darauf geprüft, ob sie in dasselbe Modul gehören wie diese Datei. Dabei werden Dateien in folgende Kategorien unterteilt:

- **Erforderliche Dateien:** In diesen wurden zyklische Abhängigkeiten ermittelt. Insofern gehören sie in dasselbe Modul.
- **Reine Abhängigkeiten:** Dem Modul zugeordnete Dateien hängen von diesen ab, jedoch nicht umgekehrt. Sie können wahrscheinlich zu einem anderen Modul gehören.
- **Umgekehrte Abhängigkeiten:** Dateien, welche von mindestens einer Datei abhängig sind, die dem Modul zugeordnet wurde. Das Modul scheint jedoch nicht von diesen abhängig zu sein und sie können wahrscheinlich zu einem anderen Modul gehören. Es gibt zudem Implementierungs-Dateien, zu denen kein Header gefunden wurde. Gibt es dennoch einen zugehörigen Header und ist dieser dem Modul zugeordnet, so gehören diese Dateien ebenfalls zum Modul.
- **Header Abhängigkeiten:** Header-Dateien, zu denen keine Implementierungs-Datei gefunden wurde. Entsprechend gibt es auch keine Translations-Einheit und keine Datei, welche die Abhängigkeiten festhält. Insofern ist es schwer einzuschätzen, ob diese Header zum Modul gehören oder nicht. Exklusive Header Abhängigkeiten kommen als Abhängigkeit jedoch nur innerhalb des Moduls vor und können von daher mit hoher Wahrscheinlichkeit dem Modul zugeordnet werden.
- **Unabhängige Dateien:** Weder konnte eine Abhängigkeit von diesen Dateien zum Modul noch vom Modul zu diesen Dateien festgestellt werden. Wegen unklaren Abhängigkeiten bestimmter Header-Dateien kann jedoch trotzdem nicht ausgeschlossen werden, dass diese zum Modul gehören.

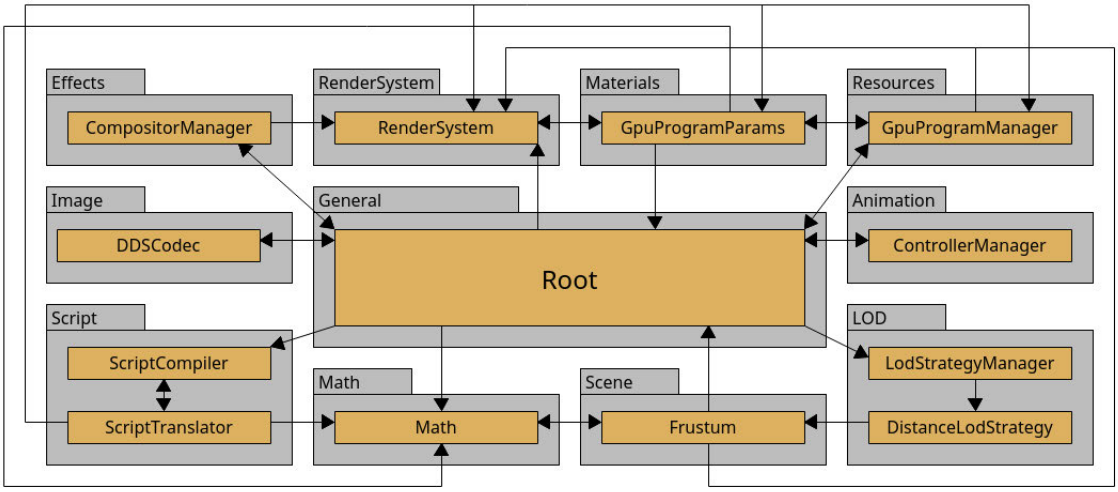

Abbildung 10.1: Auswahl zyklischer Abhängigkeiten der Core „Module“ in Ogre

Die Analyse brachte bei OGRE zu Tage, dass die meisten Dateien des „Moduls“ *Core* in der Dokumentation in dasselbe C++ Modul gehören. Insbesondere gehören Dateien aus allen Untermodulen der Dokumentation in dieses C++ Modul. Abbildung 10.1 illustriert die zyklischen Abhängigkeiten in den Untermodulen von *Core* anhand einer minimalen Auswahl von Komponenten. Es existieren weitere zyklische Abhängigkeiten, die hier aber nicht dargestellt sind. Somit ist auch eine Verschiebung weniger Dateien in andere „Module“ nicht ausreichend. Da es die nicht vorhandenen Abhängigkeiten bestimmter Header-Dateien zudem schwer machen, eine exakte Abgrenzung zu bestimmen, wurde die Entscheidung getroffen alle Untermodule in ein Modul `Ogre.Core` zusammenzufassen. Im Projekt entsprach dies dem Inhalt des Ordners *OgreMain*, welcher schließlich in *Core* umbenannt wurde. Weitere Module, jeweils mit Ogre als Präfix, umfassen:

1. **STBICodec:** Ein PlugIn zur Anbindung von der Image-Codecs von stb.
2. **GLSupport:** Hilfsbibliothek für die Anbindung von *OpenGL*.
3. **GL:** Umsetzung des Render-Systems mit *OpenGL*.
4. **RTShaderSystem:** Ermöglicht OGRE die Generierung von GPU Programmen (Shadern) in Echtzeit (Real Time).
5. **Overlay:** Zuständig für die Anzeige der Benutzeroberfläche.
6. **Bites:** Verarbeitet die Eingaben des Nutzers mit Hilfe von SDL2.
7. **Samples:** Diente ursprünglich der Anzeige mehrerer verschiedener Scenes, welche die Funktionalität von OGRE demonstrierten. In der minimierten Version kann lediglich *New Instancing* angezeigt werden, alle anderen Samples wurden entfernt.

10.4 Umwandlung

Eine vollständige Umwandlung von Headern zu Modulen umfasst logischerweise alle C++ Dateien des Projektes. Eine manuelle Umwandlung ist aufgrund des Arbeitsaufwandes nicht zu vertreten. Für diesen Zweck wurde stattdessen ein Programm geschrieben, welches die meiste Arbeit der Umwandlung automatisiert. Das Programm liegt dieser Arbeit bei.

Das Grundprinzip besteht darin, pro Ordner mit Quellcode-Dateien ein Modul zu erstellen. Es werden bestimmte Konventionen in den Dateien vorausgesetzt. So ist die Datei-Endung von Header-Dateien mit `.hpp` anzugeben, die von Implementierungs-Dateien mit `.cpp`. Header-Dateien, welche sich in einem `include` Ordner befinden, werden zu Modul-Schnittstellen-Partitionen umgewandelt. Das heißt ihr Inhalt steht außerhalb des Moduls zur Verfügung. Die zugrundeliegende Konvention hat zur Folge, dass unter Angabe des Pfads zum `include` Ordner nur auf diejenigen Header zugegriffen werden kann, bei denen dies beabsichtigt ist. Header-Dateien in einem `src` Ordner sind jedoch nach außen hin nicht sichtbar. Diese werden in Modul-Implementations-Partitionen umgewandelt. Für Dateien mit Endung `.cpp` hingegen erfolgt eine Umwandlung in eine Modul-Implementations-Einheit.

Zunächst werden für Header-Dateien die sogenannten *Include Guards* entfernt. Diese umschließen in der Regel alle Deklarationen und Einschlüsse anderer Header. Mit ihnen wird verhindert, dass ein Header mehr als einmal in dieselbe Translations-Einheit eingeschlossen wird. Auch die Anweisung `#pragma once` ist in OGRE anzutreffen. Dabei handelt es sich um eine nicht standardisierte Erweiterung, welche den gleichen Zweck erfüllt[130, § SF.8]. In Modulen sind beide Techniken nicht mehr notwendig.

In jeder Modul-Einheit wird eine Modul-Deklaration erwartet. An dieser Deklaration wird entschieden, um was für eine Einheit es sich handelt. Die Modul-Deklarationen setzen sich analog zu folgenden Beispielen zusammen:

- **Primäre Schnittstelle:** `export module Ogre.Core;`
- **Implementations-Einheit:** `module Ogre.Core;`
- **Schnittstellen-Partition:** `export module Ogre.Core:Root;`
- **Implementations-Partition:** `module Ogre.Core:DDSCodec;`

Wo vorher `#include` Anweisungen vorkamen, muss entschieden werden, ob diese in einen `import` umgewandelt werden oder ob sie in das globale Modul-Fragment verschoben werden. Letzteres wird noch vor der Modul-Deklaration mit der Zeile

```
module;
```

eingeleitet. Im Umwandlungs-Programm wird jedem Header ein (für die primäre Schnittstelle leerer) Partitions-Name zugewiesen. Er wird aus dem Datei-Pfad abgeleitet. Dadurch wird für Header gewährleistet, dass keine Partition in demselben Modul denselben Namen hat, was nicht erlaubt wäre[52, S. 379f].

Wird zu einer `#include` Anweisung eine Partition gefunden, wird sie durch einen Import dieser Partition ersetzt. Für Partitionen innerhalb desselben Moduls erfolgt dies mit

```
import :Partition;
```

in einer Zeile. Für Partitionen aus einem anderen Modul wird stattdessen mit

```
import Modul;
```

das gesamte andere Modul importiert. Zwar muss dieses nur einmal kompiliert werden, jedoch muss immer auf die vollständige Kompilation gewartet werden. Idealerweise sind Module also so klein wie möglich, um Wartezeiten während der Kompilation zu verkürzen. In Schnittstellen-Einheiten wird allen Importen zudem `export` vorangestellt, damit die vorherige Transitivität von Headern noch erhalten bleibt.

Bestimmte Header der Standard-Bibliothek sind zudem als *Header Units* importierbar[63, § 16.4.2.3]:

```
import <vector>;
```

Alle anderen `#include` Anweisungen werden in das globale Modul-Fragment verschoben. Hinzu kommen unter Einhaltung der ursprünglichen Reihenfolge vorausgehende Macro-Definitionen und andere Präprozessor-Anweisungen, da diese den Inhalt der Header beeinflussen können. Damit die Bedeutung der Importe gleichbedeutend zum vorherigen Stand bleibt, ist es zu empfehlen, jegliche Abhängigkeiten zu Macros vor der Umwandlung aufzulösen und gegebenenfalls nachträglich wieder einzufügen. Bestenfalls bemängelt der Compiler fehlende Macro-Definitionen, im schlechtesten Fall bleibt dies gänzlich unbemerkt mit ungewollten Auswirkungen auf das Programm. Die sicherste und nachhaltigste Herangehensweise ist, diese Macros vor der Migration aufzulösen oder durch exportierbare Entitäten zu ersetzen. Ist dies nicht möglich, so müssen sie in *Header-Units* verlagert werden.

In Schnittstellen-Einheiten ist es darüber hinaus erforderlich, dass die Deklarationen explizit exportiert werden. Für Deklarationen innerhalb eines `namespace` Blocks reicht ein `export` vor diesem bereits aus, um dessen Inhalt zu exportieren. Für Deklarationen außerhalb gestaltet es sich schwieriger automatisiert auszumachen, wo ein `export` hinzugefügt werden muss. Das Umwandlungs-Programm orientiert sich dabei an bestimmten Schlüsselwörtern wie `auto`, `class` oder `template` um möglichst viele Fälle abzudecken.

Für die primäre Schnittstelle eines jeden Moduls wird eine Datei mit passendem Namen erstellt, welche lediglich alle Schnittstellen-Partitionen des Moduls exportiert. Existiert bereits eine Datei mit diesem Namen wird diese jedoch nicht automatisch angepasst. Stattdessen gibt das Programm die erwarteten Exporte aus, auf dass sie der Anwender manuell einfügt.

Mit der automatisierten Umwandlung ist es jedoch noch nicht getan. Zunächst müssen auch die CMake-Dateien angepasst werden, wobei bis zur offiziellen Unterstützung auf die in Abschnitt 10.2 erwähnten Dateien zurückgegriffen werden kann. In CMake war es zudem erforderlich, Optionen für den Compiler-spezifischen Umgang mit der Sichtbarkeit von Deklarationen zu deaktivieren. Dies ist mit Modulen auch nicht mehr notwendig. An manchen Stellen fehlten in den umgewandelten Dateien Importe oder Header-Einschlüsse. Dies ist auf die nun fehlende Transitivität des globalen Modul-Fragments zurückzuführen. Mit Hilfe einer vorherigen Anwendung von IWYU kann dies zwar minimiert, jedoch nicht gänzlich eliminiert werden.

Importe eines Moduls in sich selbst kamen fälschlicherweise vor, sind aber leicht zu entfernen. Auch sind Forwärts-Deklarationen zu Typen, deren Definition sich im globalen Modul-Fragment befindet, so zu verschieben, dass sie sich ebenfalls im globalen Modul-Fragment befinden. Andernfalls sieht der Compiler hier zwei unterschiedliche Typen. Die Orientierung an bestimmten Schlüsselwörtern zum Einfügen von `export` führt in wenigen Fällen zu zu vielen Vorkommen von `export` – insbesondere bei `using namespace` ist kein `export` erlaubt – was jedoch leicht zu beheben ist. Es gab weiterhin einen Header, welcher hauptsächlich Macros definiert. Als Modul ist dieser ungeeignet. Von daher wurden die automatisierte Umwandlung rückgängig gemacht und er wurde stattdessen als *Header Unit* markiert.

Auf `namespace` Ebene führt die Verwendung des Schlüsselwortes `static` dazu, dass Funktionen nicht außerhalb einer Translations-Einheit sichtbar sind. Für eine solche Deklaration in einem Header heißt das nach der Umwandlung, dass sie nur innerhalb der jeweiligen Modul-Einheit sichtbar ist. Soll sie in einer anderen Translations-Einheit sichtbar sein, muss `static` entfernt werden. Ein Export ist jedoch nicht notwendig, da innerhalb eines Moduls alle Deklarationen desselben sichtbar sind. `main` Funktionen im Programm und in den zugehörigen Tests wurden zudem aus dem jeweiligen Modul entfernt und gegebenenfalls in eine separate Datei ausgelagert.

Module sollen zwar unterbinden, dass Macros und Deklarationen den Inhalt eines Headers verändern, bei der Verwendung von *stb* im neuen Modul `STBICodec` kam es jedoch vor, dass dies explizit erwünscht war und eine Funktion über das Macro `STBIW_ZLIB_COMPRESS` in einen Header injiziert wurde[5, D. stb_image_write.h]. Die Deklaration im globalen Modul-Fragment ist notwendig, die Definition erfordert jedoch die Importe, welche erst danach kommen. Zu lösen ist dieses Dilemma durch die explizite Angabe von `extern „C"`. Dies ist nämlich einer der Ausnahmefälle, in denen eine Deklaration nicht an das Modul gebunden wird, indem sie vorkommt[63, § 10.1]. Sollte *stb* jemals auf Module umgestellt werden, könnte diese Injektion beispielsweise so gelöst werden, dass dort ein Modul über einen festen Namen importiert wird. Der Anwender schreibt dann dieses Modul nach seinen Wünschen und exportiert es unter dem festgelegten Namen.

Mit dem Beheben der Kompilations-Fehler ist es ebenfalls noch nicht getan. Denn im Link-Prozess, in welchem die statischen Bibliotheken und kompilierten Objekt-Dateien zu einer ausführbaren Anwendung zusammengeführt werden, kommt es zunächst zu weiteren Fehlern. In Headern definierte Variablen wurden vorher in einzelnen Translations-Einheiten definiert. Da nun die jeweilige Datei eine

eigene Translations-Einheit ist, ist für das gleiche Verhalten wie vorher das Schlüsselwort `inline` in der Deklaration der Variablen notwendig. Ein weiterer Fehler zeigt sich mit Meldungen folgender Form:

```
error: undefined symbol:
  operator new@Ogre.Components.Bites(unsigned long)
```

Der einzige Hinweis, den diese Meldung liefert, ergibt sich aus dem Modul-Namen nach dem `@`. Die eigentliche Ursache besteht darin, dass es in diesem Modul Forwärts-Deklarationen zu Typen gibt, welche in einem anderen Modul definiert sind. Ohne die Definition kann auch `operator new` nicht gefunden werden. Hier sind es nicht etwa zyklische Abhängigkeiten, die die Forwärts-Deklarationen begründen, sondern es wurde hier lediglich der Einschluss eines Headers vermieden um die Kompilationsdauer zu verkürzen. Mit Modulen ist dies nicht mehr erforderlich und Forwärts-Deklarationen sollten nach Möglichkeit durch einen Import des entsprechenden Moduls ersetzt werden.

10.5 Auswirkungen

Neben der verbesserten Kompilationsdauer ist nach einer Umstellung in Module auch zu erwarten, dass sich in manchen Fällen die Ausführungsgeschwindigkeit reduziert. Das liegt nicht direkt an den Modulen, sondern an folgender Änderung: Funktionen, deren Definition sich innerhalb einer Klasse befindet, sind innerhalb eines Moduls nicht mehr implizit mit `inline` versehen. Als Konsequenz werden diese Funktionen nun aufgerufen, statt dass ihr Code in die aufrufende Funktion kopiert wird. Dies kann zu einer verschlechterten Ausführungsgeschwindigkeit führen.

Grund für diese Änderung ist allerdings, dass nun eine Deklaration nicht mehr von ihrer Definition getrennt werden muss. Das Modul exportiert nämlich nur die Deklaration. Als Translations-Einheit kann es zudem die Definition an sich binden, sodass sie letztlich nur einmal im gesamten Programm vorkommt. Mit Headern ist dies nicht möglich, da mit ihnen jede Translations-Einheit ihre eigene Definition hätte. Der Vorteil, den man dadurch erreicht, ist, dass man die Definition nicht mehr unabhängig von ihrer zugehörigen Deklaration – oder umgekehrt – abändern kann. Somit können bestimmte Fehler vermieden werden und der Wartungsaufwand wird reduziert. Auch können nun injizierte `friend` Funktionen innerhalb der Klasse definiert werden, ohne dass sie zwangsweise `inline` sind[47].

Um jedoch das gleiche Verhalten wie vorher zu erzielen, wäre im Zweifel wieder ein automatisiertes Umwandlungs-Programm gefragt. Dieses fügt dann zu jeder Funktion, die innerhalb einer Klasse definiert ist, das Schlüsselwort `inline` hinzu. Clang hatte in der verwendeten Version diese Änderung jedoch noch nicht umgesetzt[75][11]. Aus diesem Grund konnten die Auswirkungen nicht gemessen werden und es entstand kein Programm zum automatisierten Hinzufügen von `inline`.

[11] Dies gilt für die verwendete Vorabversion von Clang 15 zum Stand vom 07. Juli 2022. Seit 26. Juli 2022 wird dies jedoch offiziell unterstützt. Die Änderung kam allerdings zu spät, als dass sie in dieser Arbeit noch hätte berücksichtigt werden können.

11 Auswertung

11.1 Kompilationsdauer

Viele der durchgeführten Modernisierungen hatten kaum einen messbaren Einfluss auf die Kompilationsdauer, weder in der Debug noch in der Release Konfiguration. Die Abweichungen betragen oft weniger als eine Sekunde, bei einer gesamten Kompilationsdauer zwischen ein und zwei Minuten. Die Abbildungen 11.1 und 11.2 stellen die relevantesten Änderungen dar. In den Anhängen F und H finden sich die vollständigen Datensätze.

An einigen wenigen Stellen können jedoch wesentliche Verlangsamungen beobachtet werden. Den größten Ausschlag macht die Nutzung des Headers `<format>`, welcher in Abschnitt 9.8 beschrieben wurde. `format` arbeitet nämlich mit vielen Templates. Diese werden im Bedarfsfall für jede Translations-Einheit instanziiert, was sich in großen Blöcken neu hinzugekommener Funktionen in den Assemblies widerspiegelt. Auch das Deaktivieren der nicht standardisierten präkompilierten Header beeinträchtigt die Kompilationsdauer erheblich. In Anbetracht dessen, dass diese Technik genau dafür entwickelt wurde, die Kompilationsdauer zu senken, ist dies selbsterklärend.

Die Umstellungen auf C++20 beziehungsweise Clang 15 sind ebenfalls für eine nicht unerhebliche Verlangsamung verantwortlich. In beiden Fällen ist dies besonders auf Änderungen in der Standard-Bibliothek zurückzuführen, welche die bereits in das Projekt eingeschlossenen Header um neue Funktionalitäten erweitern. Auch ohne die Nutzung der Erweiterungen müssen sie darauf länger kompiliert werden.

Ein interessanter Fall ist `constinit` aus Abschnitt 8.8. Hier steigt die Kompilationsdauer lediglich für die Debug Konfiguration an, jedoch nicht für die Release Konfiguration. Es ist davon auszugehen, dass die nun explizit angeforderte Initialisierung während des Kompilierens bereits in der Release Konfiguration stattgefunden hat und nun auch in der Debug Konfiguration Zeit beansprucht.

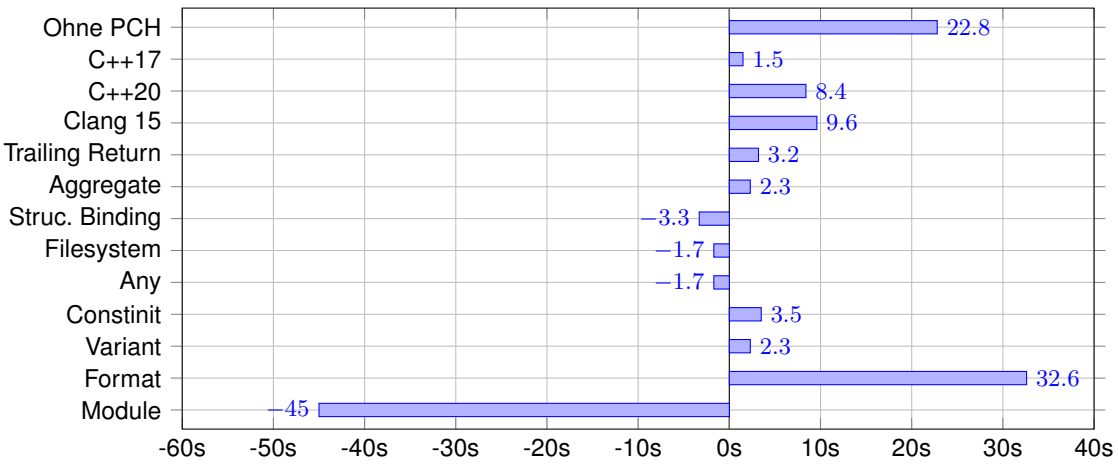

Abbildung 11.1: Relevante Differenzen Kompilationsdauer Debug

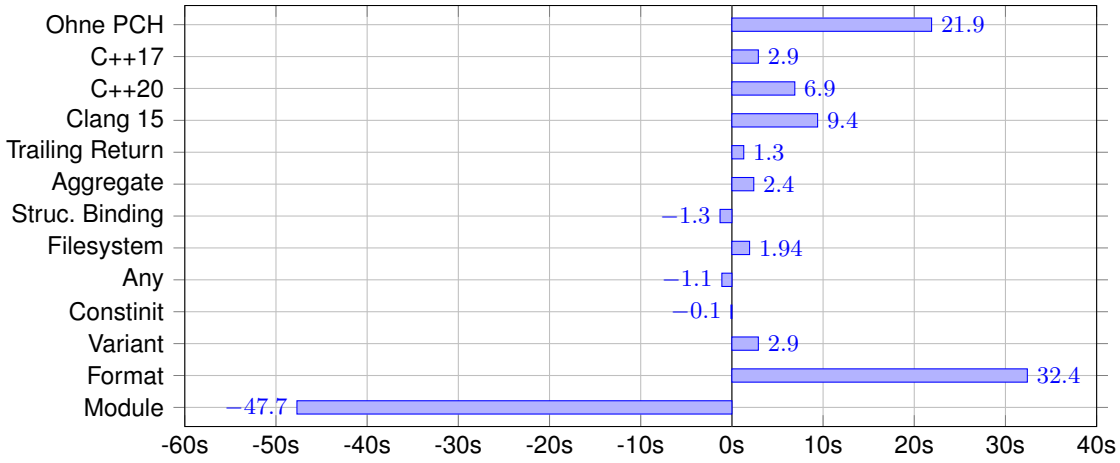

Abbildung 11.2: Relevante Differenzen Kompilationsdauer Release

Beschleunigt wird die Kompilation allen voran durch Module. Dabei fällt die Differenz sogar größer aus als die Verlangsamung durch das Deaktivieren präkompilierter Header, bleibt jedoch unter der kumulierten Verlangsamung aller Modernisierungen. Insgesamt dauert die Kompilation eines vollständig modernisierten Programms also etwas länger. Nun ist es jedoch der Fall, dass für das Erstellen mit Modulen keine offizielle Unterstützung von CMake gegeben ist. Es ist nicht auszuschließen, dass eine solche bessere Ergebnisse erzielen kann. Auch erschweren die zyklischen Abhängigkeiten in OGRE, kleinere Module zu bilden, welche gegebenenfalls parallel kompiliert werden könnten. Insofern liefern Module zwar die angepriesene Verbesserung in der Kompilationsdauer, ihr Potenzial kann aber noch nicht vollständig ausgeschöpft werden.

11.2 Startdauer

Im Großen und Ganzen haben alle Modernisierungen nur geringfügige Auswirkung auf die Startdauer des Programms. Abbildung 11.3 stellt die größten Abweichungen dar. Der vollständige Datensatz ist in Anhang J dargestellt. Die größte Differenz konnte durch die Modernisierung um Schleifen und Ranges aus Abschnitt 6.12 beobachtet werden. Mit gerade mal 25 Millisekunden bei einer gesamten Startdauer von etwas über einer Sekunde ist der Unterschied nicht wirklich bemerkbar, zumal der Start immer nur einmal durchgeführt wird. Außerdem konnten in einzelnen Messungen desselben Modernisierungs-Schrittes Abweichungen von bis zu 85 Millisekunden gemessen werden (vgl. Anhang I).

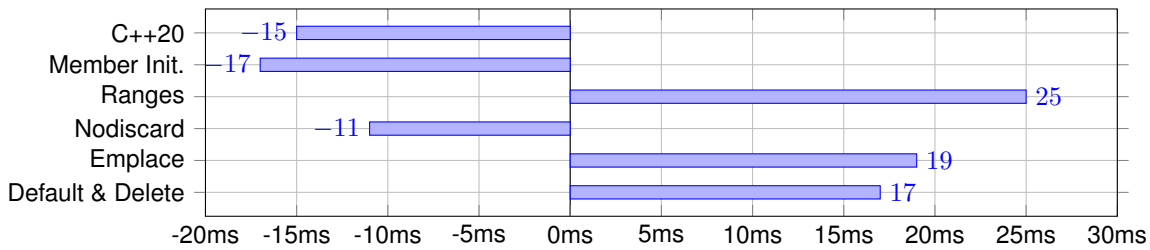

Abbildung 11.3: Relevante Differenzen Startdauer

Insbesondere `constinit` aus Abschnitt 8.8 hatte eigentlich erwarten lassen, dass Berechnungen vom Programmstart zur Kompilierzeit ausgelagert werden. Davon ist jedoch keine besondere Auswirkung zu verzeichnen. Wie Abschnitt 11.1 nahelegt, wurde dies in der Release Konfiguration bereits

an den Stellen durchgeführt, wo es möglich war. Es ist jedoch denkbar, dass umfangreichere Modernisierungen, welche möglichst viele Funktionen mit `constexpr` versehen, mehr Berechnungen vom Programmstart in die Kompilierzeit verlagern können. Der damit verbundene Arbeitsaufwand erscheint jedoch den zu erwartenden Gewinn zu übersteigen und empfiehlt sich von daher lediglich für Erweiterungen.

11.3 Speicherverbrauch

Die überwiegende Mehrheit der Modernisierungen hat keinen oder nur einen minimalen Einfluss auf den Speicherverbrauch des Programms. Die Abbildungen 11.4 und 11.5 beschränken sich von daher nur auf relevante Messungen. In den Anhängen L und M befinden sich die ungekürzten Versionen.

Abbildung 11.4: Relevante Differenzen Speicherverbrauch zu Programmstart

Am bedeutendsten ist hierbei nicht etwa eine Änderung im Quellcode von OGRE, sondern der Wechsel von Clang 14 auf Clang 15. Beim Start des Programms werden über 24 Megabytes eingespart, wobei sich der Gesamtwert über alle Messungen hinweg zwischen 140 und 172 Megabytes bewegt. Wie Abschnitt 5.7 bereits aufführt, wird seit Version 15 `allocate_at_least` in libc++ verwendet. Dies hat zur Folge, dass Container wie zum Beispiel `vector` beim Reservieren von Speicherblöcken immer dessen gesamte Größe erhalten. Vorher wurde nur der bis dahin notwendige Teil des Speicherblocks verwendet und ein neuer Speicherblock reserviert, sobald mehr Speicher notwendig war. Durch `allocate_at_least` kann die neue Reservierung hinausgezögert oder sogar gänzlich vermieden werden.

Große Ersparnisse werden zudem mit `string_view` erzielt. Das Vermeiden von Kopien eines `string`, wie in Abschnitt 8.7 beschrieben, resultiert in fast sieben eingesparten Megabytes beim Programmstart und durchschnittlich 64 Bytes pro berechnetem Frame. Im Vergleich zu den etwa 470 Kilobytes pro Frame insgesamt über alle Messungen hinweg ist dies jedoch eher zu vernachlässigen. Auch mit `format` können wie in Abschnitt 9.8 erwartet unnötige Kopien eingespart werden. Dies ist auf zusammengeführte Verkettungen zurückzuführen und beläuft sich auf etwa 720 Kilobytes Differenz zu Programmstart.

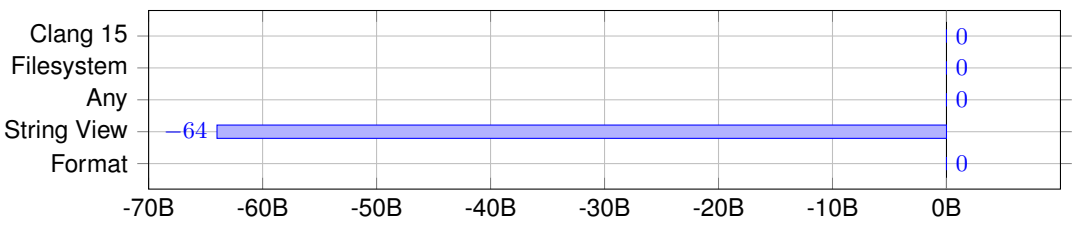

Abbildung 11.5: Relevante Differenzen Speicherverbrauch pro Frame

Mehr Speicher hingegen verbrauchen `<filesystem>` aus Abschnitt 7.8 und `any` aus Abschnitt 8.6. Die etwa 250 Kilobytes von `any` lassen sich dadurch erklären, dass `any` – anders als `Any` vorher – für kleinere Objekte einen lokalen Speicher reserviert. Sind jedoch viele Objekte zu groß für diesen Speicher, ist dieser ohne Nutzen zusätzlich reserviert worden. Oder aber der kleine Speicher ist immer noch größer als viele der darin gespeicherten Objekte. Eine benutzerdefinierte Anpassung der Größe dieses Speichers ist jedoch nicht möglich. Im Falle von `<filesystem>` sind die 2,17 zusätzlichen Megabytes dadurch zu erklären, dass diese Modernisierung zeitlich nach der `string_view` Modernisierung stattfand. Letztere konnte als Ersatz zu `string` Kopien einsparen. Mit der Klasse `path` allerdings kommt es wieder zu solchen Kopien, ohne eine alternative Klasse analog zu `string_view`.

11.4 Berechnungsdauer pro Frame

In der ursprünglichen Version von OGRE vor jeglichen Modernisierungen betrug die durchschnittliche Zeit zum Berechnen eines Frames 31,3 Millisekunden. In der letzten Messung betrug sie 31,1 Millisekunden. Kumuliert ergaben sich also keine Verluste, jedoch auch keine bedeutenden Gewinne. Nur wenige Modernisierungen hatten einen größeren Einfluss auf diese Zeit. Sie sind in Abbildung 11.6 dargestellt. Die restlichen Daten finden sich in Anhang O.

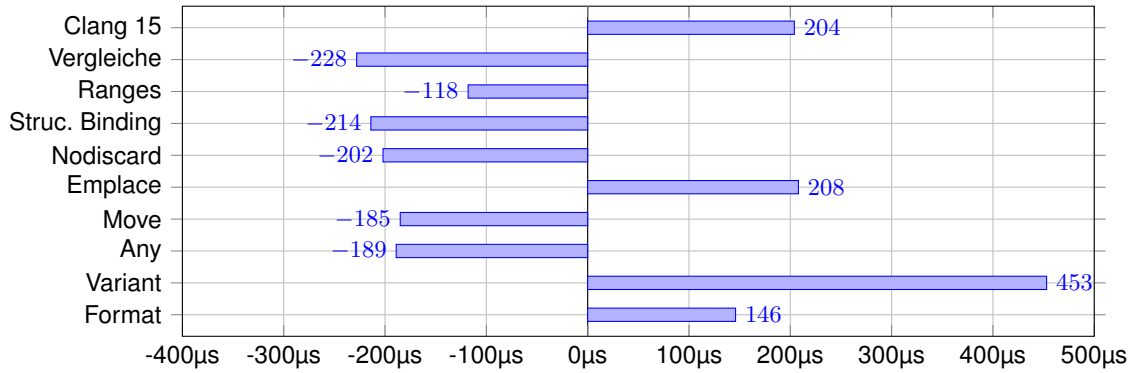

Abbildung 11.6: Relevante Differenzen Berechnungsdauer pro Frame

Auch wenn in Abschnitt 11.3 aufgezeigt wurde, dass ein Versionsupdate des Compilers (Abschnitt 5.7) zu deutlichen Einsparungen im Speicherverbrauch führen kann, gilt dies in diesem Fall nicht auch für die Berechnungsdauer eines Frames. So konnte hier eine Verschlechterung gemessen werden. Mit lediglich vier identischen Assembly-Dateien im Vergleich zum Stand davor ist es jedoch schwer auszumachen, woran die Verlangsamung genau liegt. Auch die einzelnen Assembly-Dateien unterscheiden sich erheblich. Es ist am wahrscheinlichsten, dass dies auf Änderungen in der Optimierung des Programms zurückzuführen ist. Es ist zudem möglich, dass die Verlangsamung zum Teil durch `allocate_at_least` verursacht wird. Wo vorher lediglich der Pointer zu einem neu reservierten Speicherblock verarbeitet wurde, wird nun zusätzlich dessen eigentliche Größe ausgewertet. Das Einsparen von unnötigen Reservierungen kommt also mit einem Preis.

Weitere Verlangsamungen werden durch `emplace`, `variant` und `format` aus den Abschnitten 8.3, 9.7 und 9.8 verursacht. Insbesondere für `emplace` widerspricht dies den Erwartungen. Alle drei haben gemeinsam, dass sie auf Templates zurückgreifen. Dies hat zur Folge, dass verschiedene Instanzen derselben Templates mehrere Male zum Assembly hinzugefügt werden, inklusive neue

Aufrufe an sie. Durch die Verwendung zusätzlicher Funktionen müssen diese ebenfalls in den Speicher des Prozessors geladen werden, bevor sie ausgeführt werden können. In der Tat ließ sich bei allen dreien mit *Cachegrind* feststellen, dass es auf der obersten Cache-Ebene für Instruktionen zu einer höheren Rate an Fehlzugriffen kommt. Im Vergleich zum Stand davor kommen bei insgesamt knapp 700 Millionen Fehlzugriffen für `emplace` und `variant` etwa 18 Millionen, für `format` etwa 13 Millionen hinzu. Im Falle von `insert` oder `printf`, welche jeweils nur einmal existieren müssen, sind die Instruktionen im besten Fall bereits wegen eines vorherigen Aufrufs geladen, wegen der unterschiedlichen Instanzen von `emplace` oder `format` ist dies weniger wahrscheinlich. `variant` verteilt mit dem Aufruf an `visit` die Funktionalität eines vorherigen `switch` auf mehrere Funktionen. Dadurch liegen sie im Speicher weiter auseinander und es wird wahrscheinlicher, dass sie separat geladen werden statt zusammen.

Umgekehrt verhält es sich mit der Einführung synthetisierter Vergleichs-Operatoren aus Abschnitt 6.11. Dadurch, dass mehrere Funktionen aus einer synthetisiert werden, muss jeweils nur eine Funktion im Assembly existieren und für die Ausführung geladen werden. Auch die Modernisierungen rund um Schleifen (Abschnitt 6.12) verzeichnen einen Gewinn in der Berechnungsdauer. Dies ist dadurch zu erklären, dass in den *range based for* Schleifen `end` immer nur einmal aufgerufen wird, während dies vorher gegebenenfalls einmal pro Schleifendurchlauf stattfand.

Da *structured binding* in Abschnitt 6.13 in Schleifen Anwendung fand, wirkt sich das Einsparen von Kopien pro Schleifendurchlauf besonders stark aus. Explizite Kopien des Schlüssels oder Werts eines Elements eines assoziativen Containers entfallen durch Referenzieren dieser Objekte. In Schleifen heißt dies logischerweise das Wegfallen besonders vieler Kopien mit entsprechenden Auswirkungen auf die gesamte Berechnungsdauer. Weitere Kopien entfallen durch die Anwendung von *move semantics*. Obwohl in Abschnitt 8.5 davon noch kaum Nutzen gemacht wurde, steigern die wenigen Stellen, an denen nun eine Kopie eingespart wird, die Ausführungsgeschwindigkeit messbar. Es ist davon auszugehen, dass eine umfassendere Nutzung von *move semantics* größere Unterschiede erzielen könnte. Das hätte jedoch den Umfang dieser Arbeit überstiegen.

Der Ersatz von `Any` durch `any` in Abschnitt 8.6 führte, trotz erhöhtem Speicherverbrauch, zu einer Verbesserung der Ausführungsgeschwindigkeit. Es ist davon auszugehen, dass hier besonders der lokale Speicher für kleine Objekte genutzt wurde. Zugriff auf diesen benötigt keine Indirektion auf externen Speicher und ist somit performanter. Überraschend ist auf den ersten Blick die hohe Auswirkung des Attributs `[[nodiscard]]`, immerhin sollte diese Modernisierung lediglich Warnmeldungen beeinflussen. Wie in Abschnitt 7.4 erwähnt, konnten jedoch mit Hilfe dieser Warnmeldung nicht notwendige Funktionsaufrufe entfernt werden. Die so eingesparte Arbeit reflektiert sich in den gemessenen Werten.

Insgesamt ist festzustellen, dass nicht nur diejenigen Modernisierungen die Berechnungsdauer beeinflussen, von denen dies bereits erwartet wurde. Auch solche, die eher auf die Verbesserung der Syntax und generelle Sicherheit abzielen, konnten größere Differenzen erreichen. Es ist jedoch auch festzuhalten, dass bei manchen Modernisierungen, die zwar eine Beschleunigung in Aussicht stellten, diese so gering ausfiel, dass sie hier nicht aufgeführt ist. Darüber hinaus hatte die Verwendung von `emplace`, welche eigentlich das Programm beschleunigen sollte, eine Verlangsamung zur Folge. Die Auswirkungen zusätzlicher Funktionen im Assembly fallen somit größer aus als erwartet.

12 Schlussfolgerungen

12.1 Hindernisse

Bevor irgendeine Modernisierung durchgeführt werden kann, ist zuallererst zu klären, ob der verwendete Compiler diese Modernisierung bereits unterstützt. Wie im Verlauf dieser Arbeit mehrmals festgestellt wurde, war dies nicht immer der Fall, obwohl eine Vorabversion zum Einsatz kam. So ist die Verwendung von Aggregaten beispielsweise dadurch eingeschränkt, dass ihre Initialisierung mit runden Klammern noch nicht möglich ist, was ebenfalls Auswirkungen auf die Verwendung von `emplace` und `make_unique` hat. Dabei sind Aggregate nicht die einzige Modernisierung, die wichtige Ergänzungen erst in späteren Standardisierungen erhielt. Insofern sollte vor einer Modernisierung sichergestellt werden, dass alle relevanten Features auch tatsächlich vom Compiler unterstützt werden.

Die Erweiterungen in der Standard-Bibliothek hatten teilweise zur Folge, dass existierender Code nicht mehr korrekt war. Sei es, dass veraltete Komponenten entfernt wurden oder wie in Abschnitt 5.7 beschrieben, eine benutzerdefinierte Funktion nicht mehr zum Einsatz kommt. Auch steigt die Kompilationsdauer mit jeder neuen Version tendenziell an. Im Großen und Ganzen war das Umschalten auf den neuesten Standard jedoch ohne größere Probleme möglich. Mit ihm kann auf sinnvolle Ergänzungen direkt zugegriffen werden und Modernisierungen müssen nicht für jede Standard-Version einzeln durchgeführt werden, wenn man gleich die neueste Version nimmt.

In Abschnitt 2.2 wurde aufgezeigt, dass es zu Inkompatibilitäten kommen kann, sollte nicht der gesamte Quellcode mit derselben Version des Compilers und der Standard-Bibliothek kompiliert sein. Möglich war die Neukompilation hier dadurch, dass es sich um ein Open Source Projekt handelte. Sind in einem Projekt jedoch C++ Komponenten von Drittanbietern vertreten, für die keine erneute Kompilation möglich ist, kann auch die Modernisierung des eigenen Projektes scheitern. Dieses Problem ist jedoch nicht unlösbar: Die Inkompatibilitäten kommen durch Änderungen der ABI in C++ zustande. Nun kann aber eine C++ Bibliothek mit einer reinen C Schnittstelle versehen werden. Sie hätte dann eine stabile ABI. Dadurch wird es ihren Nutzern möglich, ihr eigenes Projekt unabhängig von der Bibliothek zu modernisieren.

Bei verschiedenen Modernisierungen, welche mittels regulärer Ausdrücke über das gesamte Projekt angewandt wurden, kam es zunächst zu einer großen Zahl an Kompilations-Fehlern, wie zum Beispiel bei der Umstellung auf Typ-sichere Enumerationen in Abschnitt 9.9. Das Problem ist dabei, dass Tests-Programme und Tools wie AddressSanitizer bei der Fehlererkennung zwar äußerst nützlich, ohne vorherige Neukompilation jedoch nicht ausführbar sind. Etwaige Logik-Fehler, welche beim Beheben der Kompilations-Fehler entstehen können, sind somit erst nach dem Beheben aller Fehler zu entdecken. Dadurch ist die Anzahl der möglichen Fehlerquellen hoch und schwer zu überschauen.

Die vorherige Reduktion auf eine minimale Untermenge hat hierbei sicher geholfen diese Anzahl zu reduzieren, dennoch ist eine Modernisierung in mehreren Unterschritten für die Vermeidung von Logik-Fehlern empfehlenswert, auch wenn diese Schritte dabei sehr ähnlich ablaufen. Doch gerade wenn diese sich nur leicht unterscheiden, sind die Tore für Logik-Fehler geöffnet. In Abschnitt 6.8

war dies der Fall, wo sich die Reihenfolge der Konstruktor-Parameter von der Reihenfolge der Daten-Member unterschied. Auch in Abschnitt 9.6 kam es vor, dass die Reihenfolge der Daten-Member und die damit verbundene Aufruf-Reihenfolge ihrer Destruktoren inkonsistent zur Reihenfolge der `delete` Aufrufe war, was Fehler nach sich zog. Quellcode, welcher gewissen Mustern und Konventionen folgt, ist somit einfacher zu modernisieren.

Automatisierte Modernisierung hat sich dabei als wesentlich stabiler erwiesen. Nur in wenigen Fällen musste nach einem Durchlauf von Clang-Tidy nachgebessert werden. Die Weiterentwicklung und Erweiterung solcher Tools ist dabei vielversprechend. Ähnlich verhielt es sich mit dem entstandenen Umwandlungs-Programm für Module aus Abschnitt 10.4. Auch wenn dieses gerade zum ersten Mal zum Einsatz kam, so war die tatsächliche Fehlerrate vergleichsweise gering und könnte in einer Weiterentwicklung weiter reduziert werden.

Das größte Problem bei der Umstellung auf Module ist nicht etwa, dass dies jede einzelne Datei betrifft. Zum einen sind es Macros, welche jedoch selbst in den meisten Fällen modernisierbar sind. Zum anderen sind es die Architektur und Design-Entscheidungen der jeweiligen Software. Große Cluster zyklischer Abhängigkeiten erschweren die Umstellung, wenn sie durch Forwärts-Deklarationen verschleiert werden. Module scheinen jedoch lediglich ein Licht auf diese Schwachstellen. Sie werden auch abseits von Modulen als problematisch angesehen.

Fehler- und Warnmeldungen des Compilers, Test-Programme sowie Analyse-Tools haben sich bei der Modernisierung als unabdingbar erweisen. Oft geben sie Hilfestellung bei der Suche nach einem neu eingeführten Logik-Fehler. Mit dem entsprechenden Wissen über die Eigenheiten der Modernisierungen sind diese auch schnell zu beheben, vorausgesetzt die Fehlerquelle kann eingeschränkt werden. Automatisierte Ersetzungen mittels regulärer Ausdrücke sowie Modernisierungs-Tools reduzieren den Arbeitsaufwand zusätzlich. Jedoch gibt es hier Einschränkungen und eine allumfassende Modernisierung ist nicht zu vertreten.

Interessanterweise ist das größte Hindernis für eine Modernisierung ein bestehender Mangel an Modernisierung. Seien es die Verwendung veralteter Macros, die Nutzung veralteter Drittanbieter-Bibliotheken oder aber Compiler, die die gewünschten Features noch nicht umgesetzt haben. Problematische Architektur, Design-Entscheidungen und Code-Muster stehen der Modernisierung zwar im Weg, sie täten dies jedoch auch bei jeder Wartung. Eine Modernisierung ist somit eher eine Möglichkeit, diese Probleme zu erkennen und zu beseitigen.

12.2 Vorteile

Allein schon das Verwenden einer aktuelleren Compiler-Version kann Vorteile bringen, wie Abschnitt 11.3 zeigt. Auch besteht in neueren Versionen immer die Möglichkeit, dass vorher bestehende Fehler ausgebessert wurden und neue Warnmeldungen problematischen Quellcode hervorheben können. So konnte auch die Umstellung auf C++20 in Abschnitt 5.5 eine solche Stelle ausfindig machen. Darüber hinaus ermöglichten die Warnmeldungen, welche durch `[[nodiscard]]` in Abschnitt 7.4 und die Markierung bestimmter Funktionen mit = `default` in Abschnitt 8.4 hinzukamen, unbenutzten Code zu eliminieren. Das hat positiven Auswirkungen auf das Laufzeitverhalten. Andere

Modernisierungen erzeugen statt Warnungen Fehlermeldungen, die nicht einmal ignoriert werden können. So bringt die Verwendung von geschweiften Klammern verlustbehaftete Konvertierungen zum Vorschein.

Neue Features erlauben es, bestimmte Sachverhalte vereinfacht oder überhaupt erst auszudrücken wie Typ-Herleitung in den Abschnitten 6.4 und 6.5. Die Vereinfachung hat dabei nicht nur für Wartungsarbeiten positive Konsequenzen. So konnte in Abschnitt 11.4 gezeigt werden, dass zusammengeführte Vergleichs-Operatoren, vereinfachte Schleifen sowie *structured binding* die Ausführungsgeschwindigkeit erhöhen.

Gewisse Plattform-spezifische Wartungsarbeiten können zudem entfallen, wenn eine standardisierte Modernisierung Anwendung findet. Die Wartung verlagert sich damit nämlich in den Compiler beziehungsweise in die Standard-Bibliothek. Mit Nutzern aus vielen verschiedenen Projekten sind deren Umsetzungen mit hoher Wahrscheinlichkeit umfangreicher getestet, als es eine eigene Umsetzung je sein könnte. Zudem bestehen kaum messbare Unterschiede zwischen den standardisierten und den nicht-standardisierten Features.

Verbesserungen in Bezug auf Ausführungsgeschwindigkeit, Speicherverbrauch und Kompilationsdauer waren in manchen Fällen deutlich messbar. Dabei handelte es sich nicht um vollständige Umsetzungen der jeweiligen Modernisierung. *Move semantics* aus Abschnitt 8.5 und `string_view` aus Abschnitt 8.7 konnten aufgrund des hohen Arbeitsaufwandes nur auf einen beschränkten Teil des Quellcodes angewandt werden. Es ist zu erwarten, dass sich die dadurch gemessenen Vorteile durch umfangreichere Modernisierungen nur vergrößern. Auch Module könnten noch größere Einsparungen bei der Kompilationsdauer erreichen, wenn durch die Auflösung zyklischer Abhängigkeiten kleinere Module möglich werden.

Schließlich erlauben es neue Features auch, den Quellcode sicherer und robuster gegenüber Fehlern zu machen. Vor Allem Memory-Leaks konnten mit *Smart Pointern* vermieden werden. Insofern behob die Modernisierung existierende Fehler statt neue hinzuzufügen. Auch sorgte Typ-Sicherheit in Abschnitt 9.9 dafür, dass Fehler behafteter Quellcode nach einer Modernisierung gar nicht erst kompilierte. Manche Modernisierungen hatten dabei jedoch auch einen Preis in Bezug auf Kompilationsdauer oder Ausführungsgeschwindigkeit. Insbesondere `format` aus Abschnitt 9.8 sticht hier hervor. Zwar kann damit Speicher eingespart werden, die Kompilationsdauer erhöht sich jedoch bedeutend und auch das Programm insgesamt verlangsamt sich.

Selbstverständlich weist jede einzelne Modernisierung ihre eigenen spezifischen Vorteile auf. Insgesamt ist es eher die Ausnahme, dass diese nennenswerte Verschlechterungen an anderer Stelle mit sich bringen. So blieben in den meisten Fällen die gemessenen Werte in Kompilationsdauer, Speicherverbrauch, Startdauer und Ausführungsgeschwindigkeit gleich oder schwankten nur leicht. In den verbleibenden Einzelfällen sind die jeweiligen Vorteile jedoch immer mit der zu erwartenden Verschlechterung abzuwägen.

12.3 Mögliche Verbesserungen am Standard

Im Verlauf der während dieser Arbeit durchgeführten Modernisierungen konnte an einigen Stellen festgestellt werden, dass bestimmte Ergänzungen zum aktuellen C++ Standard nützlich gewesen wären. Im Folgenden werden diese Stellen festgehalten und kurz beschrieben, wie eine mögliche Neuerung im C++ Standard aussehen könnte. Das Ziel ist es, zukünftige Modernisierungen zu vereinfachen.

Im Bezug auf Ranges in Abschnitt 6.12 kam es zur mehrfachen Verwendung einer benutzerdefinierten Klasse `LocalSpan`. Diese führt zwei Iteratoren zusammen um eine Range zu bilden. Obwohl `span` bereits existiert, ist diese Klasse ungeeignet für Iteratoren von Containern wie einer `map`. Eine Standardisierung einer generischen Lösung für diese Container würde die Nutzung von Ranges mit Iterator-Paaren solcher Container vereinfachen.

Im ursprünglichen Vorschlag zu *structured binding* wurden bereits mögliche Kritikpunkte aufgelistet. Diese haben sich in Abschnitt 6.13 bewahrheitet. Es müssen Variablen benannt werden, die gar nicht verwendet werden und Variablen, die konstant bleiben sollen, können nicht mit `const` deklariert werden. Es ist zu hoffen, dass sich dies in einem künftigen Standard verbessert.

`string_view` aus Abschnitt 8.7 hat sich als eine überaus sinnvolle und lohnenswerte Ergänzung zu `string` erwiesen. Dabei gab es jedoch drei Probleme. Das erste tritt bei der Kompatibilität mit C Funktionen durch das Fehlen der binären Null auf. Diese kann aber auch nicht mitten in einen konstanten String eingefügt werden. Idealerweise gäbe es immer eine alternative C Funktion, die keinen mit binärer Null endenden String, sondern einen String und seine Länge oder einen String und sein Ende als Argumente erhält. Somit wäre `string_view` immer mit C Funktionen kompatibel. Auch könnten in C Strings leichter unterteilt werden, ohne jedes mal einen neuen String erzeugen zu müssen.

Das zweite Problem bezieht sich auf den Einsatz von `path` in Abschnitt 7.8. Diese Klasse ermöglicht zwar den einfachen Zugriff auf Operationen des Datei-Systems, sie erfordert jedoch immer wieder Kopien. `string_view` könnte Kopien einsparen, kann aber nicht für Zugriffe auf das Datei-System verwendet werden. Eine sinnvolle Ergänzung wäre hier eine analoge Klasse `path_view`, welche keine Kopien erfordert, jedoch weiterhin Zugriff auf das Datei-System gewährleistet.

Das dritte Problem tritt außerdem bei `any` aus Abschnitt 8.6 auf. Es handelt sich um den lokalen Speicher für kleinere Strings bzw. Objekte. Dieser ist gegebenenfalls zu groß für viele kleine Objekte oder zu klein für viele große Objekte. Ist dies schon vor der Laufzeit des Programms bekannt, besteht kein triftiger Grund, diesen ungenutzten Speicher zu reservieren. Idealerweise wäre die Größe des Speichers benutzerdefiniert. So kann man einen kleinen Speicher wählen, wenn bekannt ist, dass alle Objekte klein sind oder einen etwas größeren Speicher für etwas größere Objekte. Für besonders große Objekte könnte der Speicher ganz entfallen, da man ihn nie verwenden würde, sondern immer externen Speicher reserviert. Letzterer Fall wäre auch von Vorteil, wenn ein `string_view` auf diesen verweist. Selbst wenn der zugehörige `string` im Speicher verschoben werden würde, so wäre der `string_view` immer noch gültig. `string_view` hat bereits die Eigenschaft, dass es egal ist, welchem Typ der eigentliche String gehört. Von daher könnte sich dieser problemlos in der Größe des lokalen Speichers ändern. Entsprechend würde `any` eine Klasse `any_view` benötigen, um benutzerdefinierte Typen kompatibel zu halten.

Auch bei `variant` konnte ein Problem festgestellt werden. Anders als bei einer `union` mit manueller Buchhaltung ist eine optimierte Neuanordnung der Objekte für die Minimierung des Padding nicht möglich. Die messbaren Auswirkungen dessen waren in OGRE noch zu vernachlässigen. Für andere Projekte könnte dies jedoch durchaus bedeutend mehr Speicher verbrauchen. Die Optimierung erfordert jedoch, dass das Layout der möglichen Typen der `variant` beliebig verändert werden kann. Möglich wäre dies beispielsweise mit ATR.

In Abschnitt 9.9 wurde in zwei Fällen die Sinnhaftigkeit von Typ-sichereren Enumerationen in Frage gestellt. Im ersten Fall geht es um die Verwendung einer Enumeration als Index eines Arrays. Es kann durchaus Sinn ergeben, jedem möglichen Index einen eigenen Namen zu geben. Auch kann es von Vorteil sein, mit Typ-Sicherheit zu erreichen, dass Indices unterschiedlicher Arrays niemals verwechselt werden können. Die Notwendigkeit einer expliziten Konvertierung für jeden Zugriff auf ein Element des Arrays macht dies jedoch weniger attraktiv. Eine mögliche Lösung wäre ein spezialisiertes Template analog zu `array` aus der Standard-Bibliothek. Dieses würde ausschließlich Indices einer bestimmten Enumeration akzeptieren und diese intern explizit konvertieren. Somit könnten die Vorteile der Typ-Sicherheit ohne ihre Nachteile genutzt werden.

Der zweite Fall ist die Nutzung einer Enumeration als Bitmask. Die zu definierenden Operatoren wiederholen sich für jede einzelne Enumeration. Dies ist sowohl zusätzlicher Arbeitsaufwand als auch eine potentielle Quelle für Fehler. Ein Template für alle Enumerationen zu definieren ist jedoch nicht erwünscht. Immerhin sollen nur manche Enumerationen als Bitmask verwendet werden. Eine mögliche Lösung, welche jedoch Änderungen an der Sprache erfordern würde, wäre Definitionen folgender Form gültig zu machen:

```
auto operator bitor(Enum, Enum) -> Enum = default;
auto operator bitand(Enum, Enum) -> Enum = default;
```

Dies würde ähnlich wie in Abschnitt 6.11 den Compiler die Definition auf Basis des zugrundeliegenden integralen Typen erzeugen lassen. Die Gefahr für Fehler wäre damit deutlich reduziert und auch der Schreib- und Wartungsaufwand würde sinken. Auch wäre es nützlich, Konvertierungs-Operatoren für Enumerationen definieren zu können. So müsste nicht auf Tricks wie die doppelte Negation zurückgegriffen werden.

Auch wenn es einige Neuerungen in aktuelleren Standards gibt, welche Neuerungen aus früheren Standards ergänzen, gibt es immer noch Stellen, an denen Nachbesserungsbedarf besteht. Gegebenenfalls können aus dieser Arbeit Vorschläge hervorgehen, welche, sofern sie in den C++ Standard aufgenommen werden würden, zukünftige Modernisierungen vereinfachen werden.

12.4 Fazit und Ausblick

Anders als noch zu Beginn dieser Arbeit erwartet, kann nicht pauschal gesagt werden, dass die Vorteile aller Modernisierungen den für sie notwendigen Arbeitsaufwand rechtfertigen. So wirkte sich die Modernisierung um `emplace` negativ aus, wo mit einer positiven Wirkung gerechnet wurde. Auch `noexcept` widersprach der Annahme, einen positiven Einfluss auf die Ausführungsgeschwindigkeit zu haben. Zwar wurde auch keine nennenswerte Verlangsamung gemessen, der für die Modernisierung notwendige Arbeitsaufwand ist aber nicht unwesentlich. Verbunden mit dem Risiko, dass

das Programm unerwartet abbricht, wenn `noexcept` falsch gesetzt wurde, ist von dieser Modernisierung eher abzuraten. Ebenfalls konnte sich keine Rechtfertigung für `constinit` finden, da die gewünschten Effekte bereits durch Optimierung des Compilers statt zu finden scheinen.

Ein gemischtes Bild zeigen die Modernisierungen rund um `variant` und `format`. Beide Fälle erhöhen zwar die Typ-Sicherheit, es kommt jedoch zu messbaren Verschlechterungen in Kompilationsdauer und Ausführungsgeschwindigkeit im Vergleich zum Stand davor. Immerhin kann `format` noch Speicher einsparen. Für beide ist der damit verbundene Arbeitsaufwand moderat und es ist im Einzelfall abzuwägen, ob dieser sich lohnt.

Andere Modernisierungen sind zwar vielversprechend, scheinen aber noch unvollständig. So sind für `<filesystem>` Ergänzungen notwendig um unnötige Kopien zu vermeiden. Typ-sichere Enumerationen umzusetzen kann zum Teil mit sehr hohem Arbeitsaufwand verbunden sein, der durch Erweiterungen der Sprache und Standard-Bibliothek erheblich gesenkt werden könnte.

Nichtsdestotrotz rechtfertigen die meisten Modernisierungen ihren Aufwand. So kann einerseits der Aufwand gering sein, wie beim Auflösen bestimmter Macros durch ein entsprechendes Attribut oder gar das automatisierte Hinzuzufügen von `[[nodiscard]]`. Auch das bloße Verwenden eines aktuelleren Compilers kann schon Vorteile mit sich bringen. Oder aber die Gewinne sind so bedeutend, dass sie größeren Arbeitsaufwand rechtfertigen. Dies war bei *move semantics*, `string_view` und Modulen der Fall. Verbesserte Modernisierungs-Tools könnten in Zukunft den Arbeitsaufwand weiter reduzieren und gegebenenfalls mehr leisten, als es reguläre Ausdrücke und Handarbeit zur Zeit können.

Zudem gibt es Modernisierungen, welche Fehler im Programm ausfindig machen können. Nicht zuletzt auch dadurch, dass ohnehin vor und nach den Modernisierungen mit Tests und Tools wie AddressSanitizer sichergestellt werden muss, dass keine neuen Fehler entstehen, verbessert sich die Korrektheit des Programms. Darüber hinaus legen Modernisierungen wie Module ungünstige Design-Entscheidungen offen, welche danach angegangen werden können. Somit wird die Qualität des Quellcodes nachhaltig verbessert.

Insgesamt hat diese Arbeit das Ziel erreicht, eine 3D-Engine zu modernisieren. Sie kann dazu noch bei der Abwägung helfen, welche Modernisierungen in einem anderen Software-Projekt lohnenswert sind. Es steht nun in weiterer Arbeit die Tür offen für eine Integration von ATR. Auch stehen Vergleichswerte für die Messungen bereit. Die Ursachen für einige der gemessenen Verschlechterungen – viele Template-Instanzen und unveränderliches Daten-Layout – bekräftigen darüber hinaus die Motivation hinter ATR. Mit dieser Technik können Funktionen mit identischem Code zusammengeführt und Daten-Layouts beliebig angepasst werden[102, S. 14ff]. Sollten dann die Messungen deutliche Verbesserungen verzeichnen ist womöglich auch ATR in Zukunft Teil einer Modernisierung.

Anhang A: Abkürzungsverzeichnis

ABI	Application Binary Interface	Schnittstelle zwischen Programm-Teilen, die in kompilierter Form vorliegen. Abhängig von der Plattform und Version des Compilers.
ATR	Archetypische Rekomposition	Programmiertechnik die in C++ alternativ zu Vererbung mit virtuellen Funktionen verwendet werden kann. Erfordert moderne C++20 Features. Erfunden im Rahmen des Forschungsmoduls des Autors an der Hochschule Mittweida.
B.	Begriff	Verwendet in Quellenangaben. Dient zur Suche eines bestimmten Stichwortes auf Internet-Seiten. Schließt gegebenenfalls Unterseiten mit ein.
D.	Datei	Verwendet in Quellenangaben. Spezifiziert eine bestimmte Datei in einem Repository, beispielsweise Dokumentation. Die Datei befindet sich gegebenenfalls in Unterordnern.
GCC	GNU Compiler Collection	Software-Paket, welches unter anderem den C++ Compiler g++ beinhaltet. Es ist gebräuchlich GCC zu verwenden, wenn eigentlich g++ gemeint ist.
I.	Issue	Verwendet in Quellenangaben. Spezifiziert einen bestimmten Issue in einem Repository.
IWYU	Include What You Use	Programm, welches Quellcode so modifizieren kann, dass nur und ausschließlich verwendete Header eingeschlossen werden.
M.	Minute	Verwendet in Quellenangaben. Spezifiziert eine genauere Zeitangabe für eine Video-Quelle.
OGRE	Object-Oriented Graphics Rendering Engine	Eine hauptsächlich in C++ geschriebene Engine, welche Zugriffe auf 3D Hardware abstrahiert, sodass diese für jede Plattform gleich verwendet werden kann.
PCH	Precompiled Header	Technik, bei der Header vor allen anderen Dateien kompiliert werden. Ihr Inhalt wird dann bereitgestellt für die Kompilation anderer Dateien, sodass Header nicht mehrmals zu kompilieren sind.
S.	Seite	Verwendet in Quellenangaben. Spezifiziert eine bestimmte Seite in einem Buch oder Dokument.
SDL	Simple DirectMedia Layer	Bibliothek, die unter anderem Zugriffe auf Hardware wie GPU, Maus und Controller abstrahiert.
UB	Undefined Behaviour	Hat ein Programm UB, so verfallen Garantien jeglicher Art. Der Compiler nimmt an, dass UB niemals auftreten kann. Dies ist nützlich für Optimierungen, kann jedoch auch zu Fehlern führen, sollte UB jemals tatsächlich auftreten.
§	—	Verwendet in Quellenangaben. Spezifiziert ein bestimmtes Kapitel oder einen Abschnitt mit einem Stichwort aus dem Titel oder der zugehörigen Nummer.

Anhang B: Verwendetes System für Messungen

Betriebssystem	GNU/Linux. 5.15.04-43-generic (64 bit). Kubuntu 22.04.
Prozessor	AMD Ryzen 7 4800H @ 16 x 4200 MHz. L1-Cache: 512 KiB. L2-Cache: 4 MiB. L3-Cache: 8 MiB.
Arbeitsspeicher	16 GiB SODIMM DDR4 Synchronous Unbuffered @ 3200 MHz.
Grafikprozessor	NVIDIA GeForce RTX 2060 Mobile @ 1560 MHZ. 6 GiB GDDR6.
Datenspeicher	Samsung SSD 980 PRO 250GB @ PCI-e 4.0.

Anhang C: Verwendete Hilfsmittel

Automatisierte Modernisierung	Clang-Tidy. Versionen 14.0.0 und 15 (C. 9a0471). Online: clang.llvm.org/extra/clang-tidy (20.06.2022).
Build System	Ninja. Version 1.10.1. Online: ninja-build.org (13.08.2022)
Build Script Generator	CMake. Version 3.22.1. Online: cmake.org (13.08.2022).
Compiler	Clang. Versionen 14.0.0 und 15 (C. 9a0471). Online: clang.llvm.org (13.08.2022). GCC. Version 11.0.0. Online: gcc.gnu.org (13.08.2022).
Compiler Vergleich	Compiler Explorer. Online: godbolt.org (13.08.2022).
Datei Vergleich	KDiff3. Version 1.9.5. Online: apps.kde.org/kdiff3 (13.08.2022).
Diagramme	UMLetino - Free Online UML Tool for Fast UML Diagrams. Online: umletino.com (13.08.2022).
Entwicklungsumgebung	KDevelop. Version 22.04.2. Online: kdevelop.org (13.08.2022).
LaTeX Editor	Kile. Version 2.9.93. Online: apps.kde.org/kile (13.08.2022).
Programm Laufzeit Analyse	Valgrind. Version 3.18.1. Online: valgrind.org (13.08.2022).
Versionsverwaltung	Git. Version 2.31.1. Online: git-scm.com (13.08.2022).

Anhang D: Liste beigefügter Dateien

Assembly.tar.gz Dieses Archiv beinhaltet je einen Ordner pro Modernisierungs-Schritt. Darin sind alle Assembly-Dateien enthalten, welche im Verlauf dieser Arbeit miteinander verglichen und ausgewertet wurden.
MD5 Prüfsumme: `b949a6699b253b358acdc4584e902b0e`.

ATR-Verbessert.tar.gz Dieses Archiv beinhaltet die verbesserten Versionen der im Forschungsmodul entstandenen Beispielprogramme. Hervorzuheben ist dabei besonders die im Vergleich kürzere Kompilationsdauer von ATR sowie die Nutzung von Modulen.
MD5 Prüfsumme: `0b8fc84bfe3bf3e614fb3104d348a374`.

CMake.tar.gz Dieses Archiv beinhaltet diejenigen Dateien, welche für die Nutzung von C++20 Modulen mit CMake erforderlich waren. Sie dienen zur Überbrückung bis CMake Module offiziell unterstützt.
MD5 Prüfsumme: `dfaf0a334af57d7448e2500247585ad5`.

Forschungsbericht.tar.gz Dieses Archiv beinhaltet die Abgabe zum Forschungsmodul an der Hochschule Mittweida. Dies umfasst den eigentlichen Bericht, welcher ATR sowie dessen Motivation und Messungen beschreibt, die gemessenen Beispielprogramme und deren Messdaten.
MD5 Prüfsumme: `d2f4dc07f9928d70105c380e2421e3ba`.

Modularize.tar.gz Dieses Archiv beinhaltet das während dieser Arbeit entstandene Modernisierungs-Programm, welches für Module und das Finden ungenutzter Dateien zum Einsatz kam. Anweisungen befinden sich in der Datei README.md.
MD5 Prüfsumme: `f9265fbb62f70f084e1132b3460df99d`.

Clang15.tar.gz Dieses Archiv beinhaltet die Installations-Pakete der verwendeten Version von LLVM 15 vom 07. Juli 2022 für Debian basierte Systeme. Dies umfasst auch Clang 15 und libc++ 15. Spätere Versionen wurden nicht mehr getestet. Zudem liegen die notwendigen Dateien für die Verwendung des Headers `<source_location>` mit Installationsanweisungen bei.
MD5 Prüfsumme: `f61df55cf4a8c4f72ee03ee63ce45880`.

Ogre.tar.gz Dieses Archiv beinhaltet eine Momentaufnahme von OGRE nach jeder durchgeführten Modernisierung. Es liegen Anweisungen für die Durchführung der Messungen bei.
MD5 Prüfsumme: `bbbdbda1dafc36e7bb579bcf60a01661`.

Anhang E: Unverarbeitete Messdaten Kompilationsdauer Debug

Modernisierung	Messung #1	Messung #2	Messung #3	Messung #4	Messung #5
—	39,2 s	39,4 s	39,5 s	39,6 s	39,5 s
No PCH	62,7 s	62,8 s	62,7 s	62,2 s	60,8 s
C++14	62,3 s	62,1 s	62,2 s	62,1 s	62,2 s
C++17	63,6 s	63,8 s	63,8 s	63,7 s	63,8 s
C++20	72,2 s	72,1 s	72,0 s	72,0 s	72,2 s
C++23	72,2 s	72,8 s	72,7 s	72,8 s	72,6 s
Void Param	71,5 s	72,0 s	72,0 s	71,9 s	72,1 s
Return Init.	71,8 s	72,0 s	71,8 s	72,0 s	71,9 s
Inline	72,0 s	72,2 s	72,8 s	72,7 s	72,7 s
Nodiscard	72,1 s	72,5 s	75,1 s	73,2 s	72,7 s
Noreturn	72,0 s	72,3 s	72,6 s	72,7 s	72,8 s
Byteswap	72,3 s	72,9 s	73,0 s	72,5 s	72,9 s
Fallthrough	72,5 s	72,9 s	72,9 s	72,9 s	73,0 s
Funktoren	72,2 s	72,4 s	72,6 s	72,4 s	72,5 s
Lit. Konstanten	72,5 s	72,8 s	72,9 s	73,0 s	73,0 s
Override	71,5 s	72,2 s	72,2 s	72,5 s	72,2 s
Noexcept	71,9 s	72,2 s	72,1 s	72,1 s	72,5 s
Using Alias	72,1 s	72,5 s	72,4 s	72,6 s	72,5 s
Emplace	71,7 s	72,1 s	72,1 s	72,3 s	72,4 s
Member Init.	72,2 s	72,5 s	72,8 s	72,8 s	72,8 s
Default&Delete	73,1 s	73,5 s	73,7 s	73,3 s	73,8 s
Auto	71,9 s	72,4 s	72,5 s	72,5 s	72,6 s
Smart Pointer	72,7 s	73,1 s	73,1 s	73,2 s	73,5 s
Nested Names.	73,1 s	73,5 s	73,5 s	73,6 s	73,6 s
String-Literale	73,6 s	73,9 s	73,9 s	73,8 s	73,8 s
Move	73,5 s	74,3 s	76,1 s	77,4 s	74,3 s
Variant	77,3 s	77,2 s	77,5 s	77,3 s	77,5 s
Any	75,1 s	75,6 s	75,6 s	75,8 s	76,0 s
Vergleiche	75,5 s	76,1 s	76,1 s	76,5 s	76,5 s
Clang 15	85,6 s	85,8 s	85,9 s	85,5 s	86,1 s
Ranges	85,5 s	85,6 s	85,8 s	85,8 s	85,9 s
Format	118,5 s	118,3 s	118,3 s	118,7 s	118,1 s
Trailing Return	118,1 s	123,0 s	122,1 s	122,6 s	122,2 s
Struc. Binding	118,3 s	118,5 s	118,2 s	118,4 s	118,1 s
String View	117,2 s	117,6 s	117,1 s	117,3 s	117,2 s
Macros	117,5 s	117,6 s	117,7 s	117,0 s	117,7 s
Enumerationen	118,1 s	117,7 s	117,9 s	117,8 s	117,8 s
Aggregate	120,3 s	119,7 s	120,4 s	120,2 s	120,2 s
Constinit	123,6 s	123,9 s	124,2 s	123,9 s	122,7 s
Filesystem	121,6 s	121,9 s	122,4 s	122,2 s	121,8 s
Module	81,6 s	76,0 s	76,0 s	76,0 s	75,3 s

Tabelle E.1: Unverarbeitete Messdaten Kompilationsdauer Debug

Anhang F: Alle Differenzen Kompilationsdauer Debug

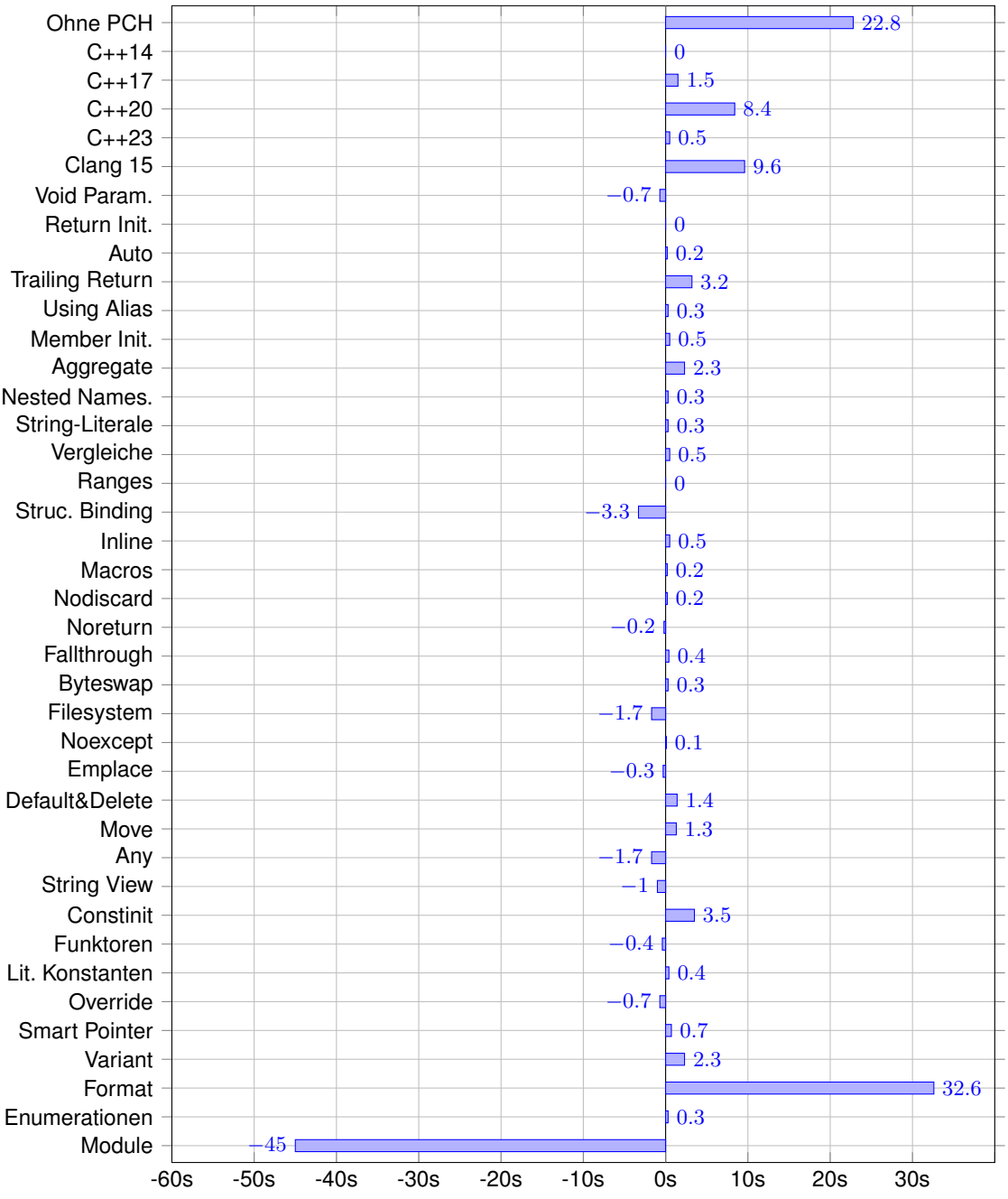

Abbildung F.1: Alle Differenzen Kompilationsdauer Debug

Anhang G: Unverarbeitete Messdaten Kompilationsdauer Release

Modernisierung	Messung #1	Messung #2	Messung #3	Messung #4	Messung #5
—	35,9 s	35,0 s	35,1 s	35,1 s	35,1 s
No PCH	56,7 s	57,0 s	57,0 s	57,1 s	56,8 s
C++14	58,4 s	58,2 s	58,4 s	58,5 s	58,3 s
C++17	59,6 s	59,7 s	59,5 s	59,9 s	67,4 s
C++20	68,0 s	68,1 s	68,0 s	68,1 s	68,2 s
C++23	68,6 s	68,5 s	68,3 s	67,6 s	66,8 s
Void Param	67,8 s	68,0 s	67,9 s	67,2 s	66,2 s
Return Init.	67,9 s	67,8 s	67,6 s	67,0 s	66,2 s
Inline	68,6 s	68,5 s	68,5 s	68,4 s	68,7 s
Nodiscard	69,9 s	70,0 s	69,9 s	68,1 s	67,8 s
Noreturn	68,6 s	68,8 s	68,5 s	67,5 s	68,2 s
Byteswap	68,5 s	68,7 s	68,8 s	67,5 s	66,9 s
Fallthrough	68,7 s	68,5 s	68,7 s	67,7 s	66,9 s
Funktoren	68,6 s	68,4 s	68,4 s	68,3 s	68,4 s
Lit. Konstanten	68,7 s	68,8 s	69,0 s	69,1 s	69,6 s
Override	68,2 s	68,2 s	68,2 s	67,2 s	66,5 s
Noexcept	67,9 s	67,8 s	68,2 s	67,2 s	66,4 s
Using Alias	68,5 s	68,6 s	68,2 s	67,4 s	66,6 s
Emplace	68,4 s	68,0 s	68,3 s	67,3 s	66,4 s
Member Init.	68,6 s	68,7 s	68,5 s	67,7 s	66,7 s
Default&Delete	69,0 s	69,4 s	69,4 s	67,7 s	67,3 s
Auto	68,1 s	68,5 s	68,2 s	68,4 s	68,5 s
Smart Pointer	68,7 s	69,1 s	69,0 s	68,2 s	67,0 s
Nested Names.	69,4 s	69,1 s	69,3 s	68,1 s	67,3 s
String-Literale	69,6 s	69,1 s	68,4 s	69,6 s	69,2 s
Move	69,7 s	69,8 s	69,9 s	69,7 s	70,2 s
Variant	72,7 s	72,9 s	72,8 s	72,6 s	72,9 s
Any	71,4 s	71,3 s	71,8 s	71,4 s	72,2 s
Vergleiche	71,9 s	71,8 s	72,2 s	70,4 s	69,9 s
Clang 15	80,6 s	80,6 s	80,6 s	80,6 s	80,7 s
Ranges	80,5 s	78,6 s	78,4 s	78,4 s	80,8 s
Format	111,8 s	111,5 s	111,8 s	111,7 s	111,7 s
Trailing Return	115,8 s	113,3 s	111,7 s	112,1 s	112,2 s
Struc. Binding	111,7 s	112,2 s	111,2 s	112,2 s	111,5 s
String View	112,3 s	112,5 s	112,6 s	112,3 s	112,8 s
Macros	112,5 s	112,6 s	112,5 s	112,6 s	112,5 s
Enumerationen	112,7 s	113,2 s	113,0 s	113,2 s	113,1 s
Aggregate	115,5 s	115,2 s	115,6 s	115,4 s	115,3 s
Constinit	115,0 s	115,4 s	115,5 s	115,5 s	115,3 s
Filesystem	117,3 s	117,3 s	117,4 s	117,2 s	117,3 s
Module	71,3 s	68,8 s	69,3 s	69,3 s	69,2 s

Tabelle G.1: Unverarbeitete Messdaten Kompilationsdauer Release

Anhang H: Alle Differenzen Kompilationsdauer Release

Abbildung H.1: Alle Differenzen Kompilationsdauer Release

Anhang I: Unverarbeitete Messdaten Startdauer

Modernisierung	Messung #1	Messung #2	Messung #3	Messung #4	Messung #5
—	1,091 s	1,042 s	1,053 s	1,075 s	1,037 s
No PCH	1,073 s	1,088 s	1,022 s	1,090 s	1,062 s
C++14	1,087 s	1,058 s	1,025 s	1,040 s	1,097 s
C++17	—	—	—	—	—
C++20	1,060 s	1,012 s	1,069 s	1,037 s	1,052 s
C++23	—	—	—	—	—
Void Param	—	—	—	—	—
Return Init.	—	—	—	—	—
Inline	—	—	—	—	—
Nodiscard	1,068 s	1,035 s	1,062 s	1,028 s	1,046 s
Noreturn	—	—	—	—	—
Byteswap	—	—	—	—	—
Fallthrough	—	—	—	—	—
Funktoren	—	—	—	—	—
Lit. Konstanten	—	—	—	—	—
Override	—	—	—	—	—
Noexcept	1,046 s	1,049 s	1,045 s	1,048 s	1,052 s
Using Alias	—	—	—	—	—
Emplace	1,070 s	1,065 s	1,047 s	1,094 s	1,060 s
Member Init.	1,036 s	1,085 s	1,042 s	1,041 s	1,047 s
Default&Delete	1,082 s	1,036 s	1,068 s	1,071 s	1,078 s
Auto	—	—	—	—	—
Smart Pointer	1,106 s	1,068 s	1,070 s	1,060 s	1,052 s
Nested Names.	—	—	—	—	—
String-Literale	—	—	—	—	—
Move	1,088 s	1,103 s	1,046 s	1,025 s	1,045 s
Variant	1,046 s	1,056 s	1,055 s	1,079 s	1,079 s
Any	1,054 s	1,057 s	1,080 s	1,063 s	1,066 s
Vergleiche	1,058 s	1,054 s	1,077 s	1,055 s	1,058 s
Clang 15	1,063 s	1,047 s	1,054 s	1,050 s	1,053 s
Ranges	1,081 s	1,082 s	1,072 s	1,079 s	1,078 s
Format	1,086 s	1,079 s	1,069 s	1,079 s	1,073 s
Trailing Return	—	—	—	—	—
Struc. Binding	1,090 s	1,085 s	1,082 s	1,079 s	1,079 s
String View	1,087 s	1,071 s	1,075 s	1,082 s	1,067 s
Macros	1,085 s	1,061 s	1,070 s	1,065 s	1,076 s
Enumerationen	1,069 s	1,065 s	1,066 s	1,064 s	1,077 s
Aggregate	1,074 s	1,071 s	1,055 s	1,063 s	1,079 s
Constinit	1,082 s	1,067 s	1,091 s	1,060 s	1,079 s
Filesystem	1,064 s	1,091 s	1,070 s	1,070 s	1,085 s
Module	1,084 s	1,075 s	1,082 s	1,082 s	1,089 s

Tabelle I.1: Unverarbeitete Messdaten Startdauer

Anhang J: Alle Differenzen Startdauer

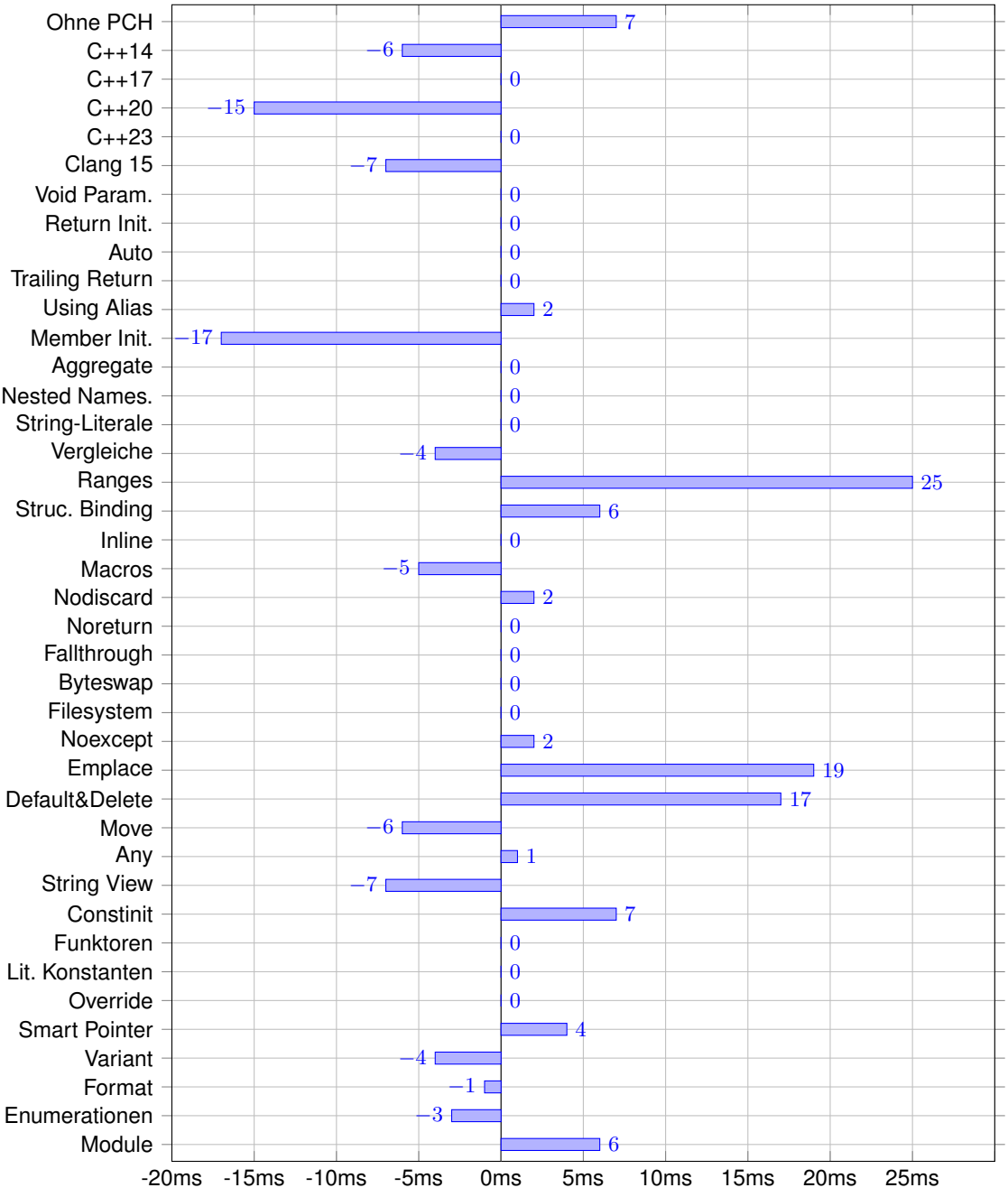

Abbildung J.1: Alle Differenzen Startdauer

Anhang K: Unverarbeitete Messdaten Speicherverbrauch

Modernisierung	Messung Frame 0	Messung Frame 20	Messung Frame 666
—	172,09 MB	203,19 MB	507,15 MB
No PCH	172,09 MB	203,19 MB	507,15 MB
C++14	172,09 MB	203,19 MB	507,15 MB
C++17	—	—	—
C++20	172,09 MB	203,19 MB	507,15 MB
C++23	—	—	—
Void Param	—	—	—
Return Init.	—	—	—
Inline	—	—	—
Nodiscard	172,09 MB	203,19 MB	507,15 MB
Noreturn	—	—	—
Byteswap	—	—	—
Fallthrough	—	—	—
Funktoren	—	—	—
Lit. Konstanten	—	—	—
Override	—	—	—
Noexcept	172,09 MB	203,19 MB	507,15 MB
Using Alias	—	—	—
Emplace	172,09 MB	203,19 MB	507,15 MB
Member Init.	172,09 MB	203,19 MB	507,15 MB
Default&Delete	172,09 MB	203,19 MB	507,15 MB
Auto	—	—	—
Smart Pointer	172,09 MB	203,19 MB	507,15 MB
Nested Names.	—	—	—
String-Literale	—	—	—
Move	172,09 MB	203,19 MB	507,15 MB
Variant	172,09 MB	203,19 MB	507,15 MB
Any	172,34 MB	203,44 MB	507,40 MB
Vergleiche	172,34 MB	203,44 MB	507,40 MB
Clang 15	147,92 MB	179,03 MB	482,99 MB
Ranges	147,92 MB	179,03 MB	482,99 MB
Format	147,20 MB	178,31 MB	482,26 MB
Trailing Return	—	—	—
Struc. Binding	147,20 MB	178,31 MB	482,27 MB
String View	140,52 MB	169,52 MB	473,43 MB
Macros	140,52 MB	169,52 MB	473,43 MB
Enumerationen	140,52 MB	169,52 MB	473,43 MB
Aggregate	140,52 MB	169,52 MB	473,43 MB
Constinit	140,52 MB	169,52 MB	473,43 MB
Filesystem	142,69 MB	171,74 MB	475,65 MB
Module	142,69 MB	171,74 MB	475,65 MB

Tabelle K.1: Unverarbeitete Messdaten Speicherverbrauch

Anhang L: Alle Differenzen Speicherverbrauch zu Programmstart

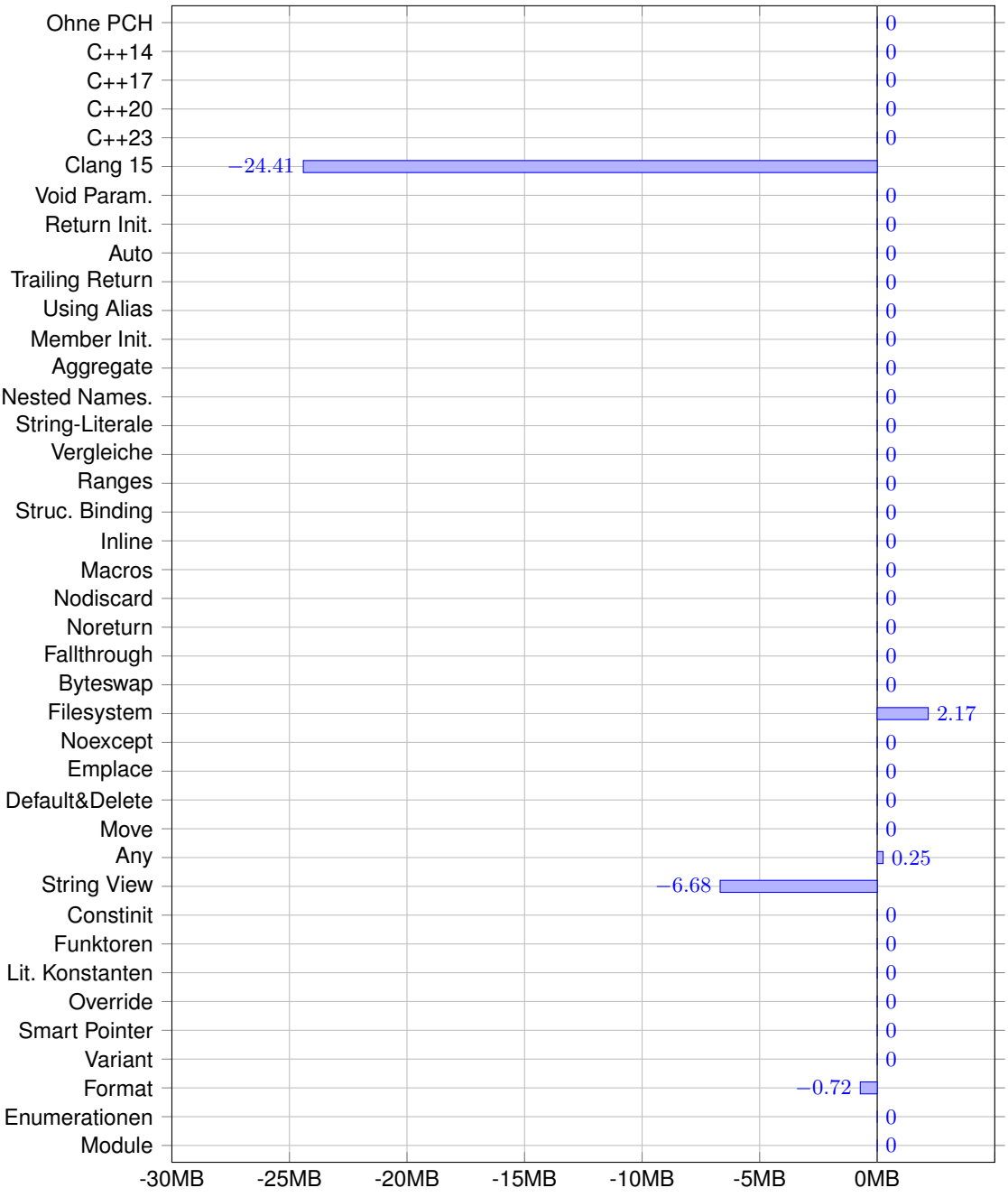

Abbildung L.1: Alle Differenzen Speicherverbrauch zu Programmstart

Anhang M: Alle Differenzen Speicherverbrauch pro Frame

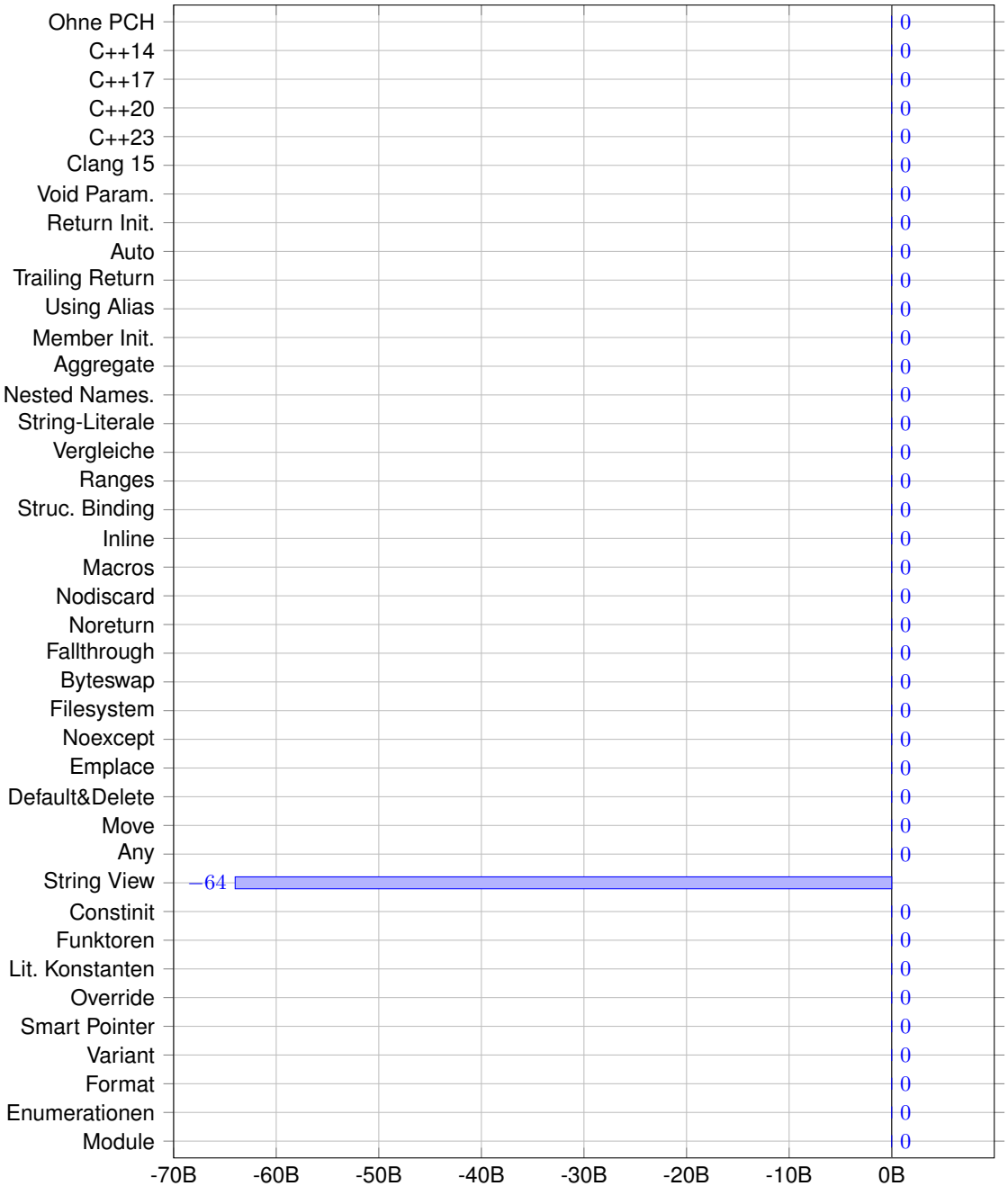

Abbildung M.1: Alle Differenzen Speicherverbrauch pro Frame

Anhang N: Unverarbeitete Messdaten Berechnungsdauer pro Frame

Modernisierung	Messung #1	Messung #2	Messung #3	Messung #4	Messung #5
—	31,381 ms	31,316 ms	31,351 ms	31,345 ms	31,371 ms
No PCH	31,373 ms	31,282 ms	31,340 ms	31,419 ms	31,368 ms
C++14	31,340 ms	31,416 ms	31,150 ms	31,381 ms	31,193 ms
C++17	—	—	—	—	—
C++20	31,408 ms	31,294 ms	31,237 ms	31,287 ms	31,155 ms
C++23	—	—	—	—	—
Void Param	—	—	—	—	—
Return Init.	—	—	—	—	—
Inline	—	—	—	—	—
Nodiscard	31,066 ms	31,122 ms	31,104 ms	31,065 ms	31,014 ms
Noreturn	—	—	—	—	—
Byteswap	—	—	—	—	—
Fallthrough	—	—	—	—	—
Funktoren	—	—	—	—	—
Lit. Konstanten	—	—	—	—	—
Override	—	—	—	—	—
Noexcept	31,175 ms	31,096 ms	30,972 ms	31,086 ms	31,116 ms
Using Alias	—	—	—	—	—
Emplace	31,236 ms	31,322 ms	31,292 ms	31,286 ms	31,347 ms
Member Init.	31,269 ms	31,343 ms	31,292 ms	31,231 ms	31,408 ms
Default&Delete	31,374 ms	31,314 ms	31,365 ms	31,227 ms	31,451 ms
Auto	—	—	—	—	—
Smart Pointer	31,265 ms	31,367 ms	31,306 ms	31,284 ms	31,365 ms
Nested Names.	—	—	—	—	—
String-Literale	—	—	—	—	—
Move	31,103 ms	31,145 ms	31,119 ms	31,169 ms	31,129 ms
Variant	31,575 ms	31,611 ms	31,565 ms	31,545 ms	31,632 ms
Any	31,450 ms	31,318 ms	31,378 ms	31,424 ms	31,411 ms
Vergleiche	31,203 ms	31,198 ms	31,200 ms	31,114 ms	31,124 ms
Clang 15	31,364 ms	31,280 ms	31,410 ms	31,459 ms	31,348 ms
Ranges	31,229 ms	31,184 ms	31,276 ms	31,310 ms	31,272 ms
Format	31,388 ms	31,455 ms	31,399 ms	31,427 ms	31,334 ms
Trailing Return	—	—	—	—	—
Struc. Binding	31,105 ms	31,253 ms	31,126 ms	31,278 ms	31,172 ms
String View	31,227 ms	31,130 ms	31,035 ms	31,156 ms	31,083 ms
Macros	31,095 ms	31,015 ms	31,140 ms	31,125 ms	31,165 ms
Enumerationen	31,040 ms	30,977 ms	31,198 ms	31,000 ms	31,102 ms
Aggregate	31,102 ms	31,184 ms	31,185 ms	31,200 ms	31,008 ms
Constinit	31,211 ms	31,203 ms	31,286 ms	31,249 ms	31,091 ms
Filesystem	31,215 ms	31,202 ms	31,176 ms	31,134 ms	31,079 ms
Module	31,050 ms	31,168 ms	31,134 ms	31,104 ms	31,085 ms

Tabelle N.1: Unverarbeitete Messdaten Berechnungsdauer pro Frame

Anhang O: Alle Differenzen Berechnungsdauer pro Frame

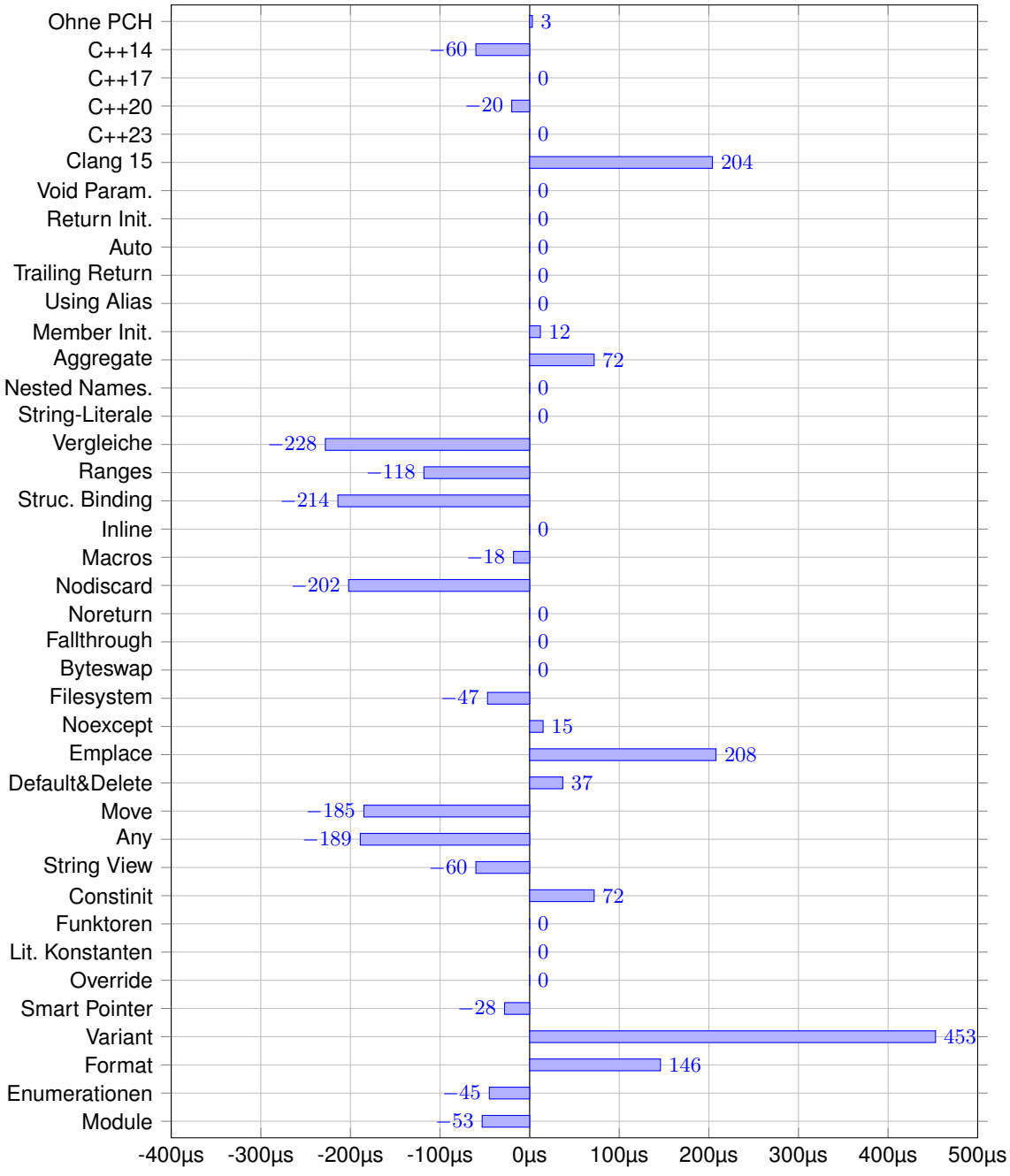

Abbildung O.1: Alle Differenzen Berechnungsdauer pro Frame

Anhang P: Kommentar zum aktuellen Stand von Archetypischer Rekomposition

Obwohl ATR die zugrundeliegende Motivation hinter dieser Arbeit war, kam es letztlich nicht zum Einsatz dieser Technik. Dennoch wurde die zugehörige Implementierung seit dem Forschungsmodul erweitert und verbessert. Die verbesserte Version liegt dieser Arbeit bei. Der wohl größte Kritikpunkt an ATR war die Kompilationsdauer, welche bei ATR zehnmal so lange dauerte wie beim Vergleichs-Programm mit virtuellen Funktionen. Dies konnte reduziert werden, sodass die Kompilation nun nur noch etwa 75% länger dauert. Die zwei Hauptgründe für diese Verbesserung sind folgende:

1. **Aggregate:** In ATR wird viel mit Template-Klassen gearbeitet. Sie hatten dabei in der ursprünglichen Version Konstruktoren, welche ebenfalls Templates waren. Die verbesserte Version vereinfacht diese Klassen durch das Weglassen dieser Konstruktoren und initialisiert sie als Aggregate. Dadurch kommt es weder zu Instanziierungen der Konstruktor-Templates noch zu einem Auswahlverfahren um zu bestimmen, welcher überladene Konstruktor am besten geeignet ist.
2. **Type-Erasure:** Eine entscheidende Technik, die es ATR erlaubt Funktionen gleichen Codes zusammenzuführen, ist *Type-Erasure*. Dabei werden Typ-Informationen aus den Parametern einer Funktion gelöscht. Stattdessen sind nur noch die relevanten Abhängigkeiten in einem Template-Argument abgespeichert. Eine solche Abhängigkeit wäre beispielsweise ein Objekt vom Typ `int`, welches sich an der vierten Byte-Position im Argument befindet. Stimmen die Abhängigkeiten zweier Instanziierungen überein resultiert dies in dieselbe Funktion. Eine ähnliche Technik kann auch schon zur Kompilierzeit eingesetzt werden. Ein Template erzeugt für jeden Typen als Argument eine neue Instanziierung. Meist sind die relevanten Informationen jedoch lediglich, dass sich zwei Typen unterscheiden. Typen können stattdessen auf eindeutige Konstanten bijektiv abgebildet werden. Diese Konstanten selbst sind dabei jedoch alle vom selben Typen. Die Eigenschaft, dass sie sich unterscheiden, bleibt erhalten. Fügt man der Konstanten noch weitere Werte wie die Größe oder Ausrichtung des Typs hinzu, so kann aus einer Liste von Typen ein Array von Typ-Informationen generiert werden. Dieses ist mit herkömmlichen Algorithmen schnell sortierbar. Dadurch, dass die Neuanordnung des Layouts ein essenzieller Bestandteil von ATR ist, trägt auch die Sortierung ein großes Gewicht. Kann beim Sortieren zur Kompilierzeit also auf effiziente Algorithmen zurückgegriffen werden, welche nur eine geringe Anzahl an Template-Instanziierungen erfordern, sinkt die Kompilationsdauer erheblich.

Ein weiterer Kritikpunkt bestand in den schwer leserlichen Debug-Symbolen. So resultierte ein Identifikator der Form „`ABC"_ID` in ein Symbol der Form `ID<(char)65, (char)66, (char)67>`. Die Template-Parameter von `ID` hatten nämlich den Typen `char`. In der Verbesserung ist das resultierende Symbol `ID<A, B, C>`. Zwar stören immer noch Kommas den Lesefluss, die Bedeutung des Symbols ist aber intuitiv verständlich. Erreicht wird dies durch Template-Parameter des Typs `char const&`. Die Referenz wird mit dem Namen der Variablen repräsentiert, an welche sie gebunden ist. Ist eine solche Variable also `A` benannt und befindet sie sich in keinem `namespace`, so erreicht man den oben genannte Symbol-Namen. Nun sind aber Variablen-Namen von einem Buchstaben außerhalb jedes `namespace` problematisch. Das Risiko für Namenskollisionen ist sehr hoch. Sind die Variablen jedoch innerhalb eines Moduls definiert und werden sie nicht exportiert, so besteht dieses Risiko nicht mehr.

Anstelle von Schnittstellen verwendet ATR Concepts. Mit ihnen kann entschieden werden, ob eine Funktion für einen konkreten Typen existiert oder nicht. Ihre Angabe war in der ursprünglichen Version jedoch sehr langatmig. Jede Funktions- und jede Daten-Abhängigkeit, welche Teil der „Schnittstelle“ sind, musste separat angegeben werden. Dies sorgte für viel Schreibaufwand und fehleranfällige Wiederholungen. In der verbesserten Version erfolgt die Angabe der „Schnittstelle“ beispielsweise so: `ProtoMemberInterface<„Height", „Width">`. Dieses Concept ist genau dann erfüllt, wenn das Argument die Daten-Member `Height` und `Width` besitzt. Entscheidend ist dabei der `[]` Operator mit Parameter des oben genannten Templates `ID`. Möglich wird dies durch das Generieren eines logischen Terms zur Kompilierzeit. So übersetzt sich die „Schnittstelle“ in das Prädikat *„hat Daten-Member Height UND hat Daten-Member Width“*. Dieser Term wird optimiert und dann in ein Concept umgewandelt. Die Größe des Terms ist dabei beschränkt.

In der ursprünglichen Version von ATR gab es keine Unterstützung für Bit-Feld Daten-Member. Auch konnte der Speicher nicht für mehrere `bool` Daten-Member optimiert werden. Jedes `bool` Objekt benötigt lediglich 1 Bit, belegt aber in der Regel ein ganzes Byte. In der verbesserten Version werden alle Bit-Felder und alle `bool` Daten-Member gemeinsam in einem Byte-Array abgelegt. Schreib- und Lesezugriffe erfolgen nach dem gleichen Prinzip wie `bitset` aus der Standard-Bibliothek. Der entscheidende Vorteil gegenüber Bit-Feldern ist, dass immer nur die minimal notwendige Anzahl an Bytes reserviert wird. Ein Bit-Feld der Länge 33 nimmt in C++ 8 Bytes ein, obwohl nur 5 notwendig wären. Bit-Felder können in ATR zudem besser komprimiert werden. So nehmen 8 Bit-Felder der Länge 7 ganze 8 Bytes ein, sofern der zugrundeliegende Typ 8 Bits groß ist. In ATR werden die Bit-Felder komprimiert und nehmen lediglich 7 Bytes insgesamt in Anspruch.

Der Integration von ATR in OGRE stehen einige entscheidende Hindernisse im Weg: Zunächst erfordert die Umwandlung einer Klasse in einen Archetypen, dass alle Daten-Member sowohl trivial sind als auch Standard-Layout haben. So müssten für eine Abänderung der Klasse `Node` in OGRE auch die Klassen `vector`, `set`, `string`, `Quaternion`, `Vector3` und `Affine3` angepasst werden. Für Komponenten von OGRE heißt diese Anpassung wiederum, dass an vielen anderen Stellen in der Engine, die auf diese Typen zurückgreifen, zusätzliche Anpassungen notwendig werden. Für eine Änderung von `string` hilft die Modernisierung nach `string_view` bereits, da diese Anpassungen dadurch größtenteils entfallen können. `vector` und `set` aus der Standard-Bibliothek jedoch benötigen eine Container-Alternative in ATR, damit sie sowohl trivial als sein als auch Standard-Layout haben können. Nun ist für Container jedoch essenziell, dass der reservierte Speicher im Destruktor wieder freigegeben wird. Dies ist keine triviale Operation. Für ATR wäre es von daher notwendig, das Layout eines Containers von seinem Destruktor zu trennen. Für mehr als einen Container als Daten-Member heißt das jedoch, dass die Destruktoren nach der Neuanordung des Layouts ebenfalls neu zusammengesetzt werden müssten. Diese Funktionalität ist in ATR noch nicht vorhanden.

Darüber hinaus sollte ein neuer Container die Stärken von ATR ausnutzen. Dies wären bezogen auf Container die Sortierung des Layouts sowie die Lokalität des Speichers. Ersteres ist dadurch zu nutzen, dass anstelle eines dynamischen Arrays einzelner Objekte nun mehrere dynamische Arrays der einzelnen Daten-Member abgespeichert werden könnten. Dadurch lägen alle Daten eines Daten-Members dichter zusammen. Von Vorteil ist dies besonders dann, wenn es sich um einen Daten-Member des Typs `bool` handelt. Gegebenenfalls könnten 64 `bool` Daten-Member nebeneinander liegen. Um dann zu bestimmen, ob alle `false` sind, müsste der Speicher lediglich als eine 64 Bit große Zahl interpretiert und mit `0` verglichen werden. Die Notwendigkeit einer Iteration über alle einzelnen Elemente entfiele.

In dem Beispielprogramm für virtuelle Funktionen wurde ein `vector` von `unique_ptr` verwendet, um Objekte unterschiedlicher Größe mit derselben Schnittstelle zu speichern. Hierdurch kommt es zu zwei Indirektionen bis der Speicher eines Elements erreicht wird. Die Elemente schließlich können weit auseinander im Speicher liegen. ATR konnte eine Indirektion einsparen und die Elemente nah hintereinander im Speicher ablegen. Dies erhöhte somit die Lokalität des Speichers. OGRE verwendet in `Node` zwar gleichermaßen einen `vector` von Pointern, dabei ist jedoch eine weitere Eigenschaft wichtig: Wächst der `vector`, so bleiben Pointer auf die einzelnen Elemente weiterhin gültig. In ATR würden die Elemente verschoben und es käme zu ungültigen Pointern. Von daher ist eine andere Strategie gefragt. Bereits einfügte Elemente dürfen also nicht mehr verschoben werden und trotzdem sollen die einzelnen Elemente möglichst nah beieinander im Speicher liegen. Ein möglicher Ansatz wäre, ebenfalls einen `vector` mit `unique_ptr` umzusetzen, jedoch statt einem Element immer die oben genannten Arrays aus Daten-Membern abzulegen. Somit würden diese nicht nachträglich verschoben und wären trotzdem nah beieinander.

Die Hindernisse lassen sich also wie folgt zusammenfassen: Die Umstellung einer Klasse benötigt zunächst eine Umstellung aller Daten-Member. Die Umstellung der Daten-Member wiederum benötigt eine Umstellung aller Komponenten, die von ihnen abhängen. Für Daten-Member aus der Standard-Bibliothek sind nun benutzerdefinierte Typen gefragt. Eine Alternative zu `vector` ist aber nicht vorhanden. Die Umsetzung benötigt vor Allem die Möglichkeit, Destruktoren neu zusammensetzen zu können. Diese Funktionalität ist ebenfalls noch nicht vorhanden. Das Ablegen der Daten-Member in Arrays kann zwar durch die Neuanordung des Layouts erreicht werden, ob dies jedoch alle notwendigen Anwendungsfälle abdecken würde, benötigt eine intensivere Studie des Quellcodes von OGRE. Alles in allem wurde der Arbeitsaufwand als zu groß eingeschätzt, als dass eine Integration von ATR in OGRE neben der notwendigen Modernisierung noch Teil dieser Arbeit hätte sein können, auch wenn die möglichen Gewinne in Ausführungsgeschwindigkeit und Speicherverbrauch vielversprechend sind. Die Aufgabe der Integration verbleibt somit für eine zukünftige Arbeit.

Literaturverzeichnis

[1] ABRAHAMS D. ET AL.: N3050: Allowing Move Constructors to Throw (Rev. 1). ISO/IEC C++ Standards Committee (März 2010). Online: wg21.link/n3050 (13.08.2022).

[2] ANDRIST B. & SEHR V.: C++ High Performance. Second Edition. Master the art of optimizing the functioning of your C++ code. Packt Publishing Ltd., Birmingham (Dezember 2020). ISBN: 978-1-83921-654-1.

[3] AŽMAN G. & MÜLLER J.: P1099R5: Using Enum. ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p1099r5 (13.08.2022).

[4] BANCILA M.: Modern C++ Programming Cookbook. Second Edition. Master C++ core language and standard library features, with over 100 recipes, updated to C++20. Packt Publishing Ltd., Birmingham (September 2020). ISBN: 978-1-80020-898-8.

[5] BARRETT S. ET AL: stb. single-file public domain (or MIT licensed) libraries for C/C++. (September 2021). Online: github.com/nothings/stb/ (13.08.2022).

[6] BARBATI A.: N3760: [[deprecated]] attribute. ISO/IEC C++ Standards Committee (September 2013). Online: wg21.link/n3760 (13.08.2022).

[7] BASTIEN JF: P0476R2: Bit-casting object representations. ISO/IEC C++ Standards Committee (November 2017). Online: wg21.link/p0476r2 (13.08.2022).

[8] BYARINOV K. ET AL.: P2077R3: Heterogeneous erasure overloads for associative containers. ISO/IEC C++ Standards Committee (September 2021). Online: wg21.link/p2077r3 (13.08.2022).

[9] BROWN W.: N1737: A Proposal to Restore Multi-declarator auto Declarations. ISO/IEC C++ Standards Committee (November 2004). Online: wg21.link/n1737 (13.08.2022).

[10] BROWN W.: N3546: TransformationTraits Redux, v2. ISO/IEC C++ Standards Committee (April 2013). Online: wg21.link/n3655 (13.08.2022).

[11] BROWN W.: P0768R1: Library Support for the Spaceship (Comparison) Operator. ISO/IEC C++ Standards Committee (November 2017). Online: wg21.link/p0768r1 (13.08.2022).

[12] BROWNING J. & SUTHERLAND B.: C++20 Recipes. A Problem-Solution Approach. Second Edition. Apress Media LLC, New York (2020). ISBN: 978-1-4842-5713-5.

[13] CHAN T. ET AL.: Algorithms for computing the sample variance: analysis and recommendations. Technical Report #222. Yale University (Mai 1982). Online: cpsc.yale.edu/sites/default/files/files/tr222.pdf (13.08.2022).

[14] CMAKE.ORG: CMake Reference Documentation. Kitware, Inc. and Contributors (2022). Online: cmake.org/cmake/help/v3.23/ (13.08.2022).

[15] CIVIL G. ET AL.: GoogleTest. Google Inc. (2022). Online: github.com/google/googletest (13.08.2022).

[16] CROWL L: N2326: Defaulted and Deleted Functions. ISO/IEC C++ Standards Committee (Juli 2007). Online: wg21.link/n2346 (13.08.2022).

[17] DAWES B.: N1975: Filesystem Library Proposal for TR2 (Revision 3). ISO/IEC C++ Standards Committee (April 2006). Online: wg21.link/n1975 (13.08.2022).

[18] DAWES B.: N2146: Raw String Literals (Revision 1). ISO/IEC C++ Standards Committee (Januar 2007). Online: wg21.link/n2146 (13.08.2022).

[19] DAWES B.: N2984: Additional Type Traits for C++0x (Revision 1). ISO/IEC C++ Standards Committee (Oktober 2009). Online: wg21.link/n2984 (13.08.2022).

[20] DAWES B. ET AL.: N3804: *Any* Library Proposal (Revision 3). ISO/IEC C++ Standards Committee (Oktober 2013). Online: wg21.link/n3804 (13.08.2022).

[21] DAWES B. & MEREDITH A.: P0220R1: Adopt Library Fundamentals V1 TS Components for C++17 (R1). ISO/IEC C++ Standards Committee (März 2016). Online: wg21.link/p0220r1 (13.08.2022).

[22] DIMOV P. ET AL.: N1450: A Proposal to Add General Purpose Smart Pointers to the Library Technical Report. Revision 1. ISO/IEC C++ Standards Committee (März 2003). Online: wg21.link/n1450 (13.08.2022).

[23] DIMOV P. & DAWES B.: N2232: Improving shared_ptr for C++0x. ISO/IEC C++ Standards Committee (April 2007). Online: wg21.link/n2232 (13.08.2022).

[24] DIMOV P. ET AL.: P0784R7: More constexpr containers. ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p0784r7 (13.08.2022).

[25] DIMOV P. ET AL.: P1064R0: Allowing Virtual Function Calls in Constant Expressions. ISO/IEC C++ Standards Committee (Mai 2018). Online: wg21.link/p1064r0 (13.08.2022).

[26] DIMOV P. ET AL.: P1327R1: Allowing `dynamic_cast`, polymorphic `typeid` in Constant Expressions. ISO/IEC C++ Standards Committee (November 2018). Online: wg21.link/p1327r1 (13.08.2022).

[27] DIONNE L.: P1002R1: Try-catch blocks in constexpr functions. ISO/IEC C++ Standards Committee (November 2018). Online: wg21.link/p1002r1 (13.08.2022).

[28] DIONNE L. & VANDERVOORDE D. P1330R0: Changing the active member of a union inside constexpr. ISO/IEC C++ Standards Committee (November 2018). Online: wg21.link/p1330r0 (13.08.2022).

[29] DMITROVIĆ S.: Modern C++ for Absolute Beginners. A Friendly Introduction to C++ Programming Language and C++11 to C++20 Standards. Apress Media LLC, New York (2020). ISBN: 978-1-4842-6047-0.

[30] DOUGLAS R.: N4417: Source-Code Information Capture. ISO/IEC C++ Standards Committee (April 2015). Online: wg21.link/n4417 (13.08.2022).

[31] DOS REIS G.: N1449: Proposal to add template aliases to C++. ISO/IEC C++ Standards Committee (April 2003). Online: wg21.link/n1449 (13.08.2022).

[32] DOS REIS G. ET AL.: N2235: Generalized Constant Expressions – Revision 5. ISO/IEC C++ Standards Committee (April 2007). Online: wg21.link/n2235 (13.08.2022).

[33] DOS REIS G.: N3651: Variable Templates (Revision 1). ISO/IEC C++ Standards Committee (April 2013). Online: wg21.link/n3651 (13.08.2022).

[34] DOS REIS.G: P0138R2: Construction Rules for enum class Values. ISO/IEC C++ Standards Committee (März 2016). Online: wg21.link/p0138r2 (13.08.2022).

[35] FINKEL H. & SMITH R.: P0386R2: Inline Variables. ISO/IEC C++ Standards Committee (Juni 2016). Online: wg21.link/p0386r2 (13.08.2022).

[36] FISELIER E.: P1143R2: Adding the `constinit` keyword. ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p1143r2 (13.08.2022).

[37] FRIEDMAN B.: P0040R3: Extending memory management tools. ISO/IEC C++ Standards Committee (Juni 2016). Online: wg21.link/p0040r3 (13.08.2022).

[38] GARCIA J.: N3114: `throw()` becomes `noexcept`. ISO/IEC C++ Standards Committee (August 2010). Online: wg21.link/n3114 (13.08.2022).

[39] GIT-SCM.COM: Reference. Software Freedom Conservancy (2022). Online: git-scm.com/docs (13.08.2022).

[40] GONZALVE S.: Why you should move your legacy code to smart pointers. CPPP Conference, Paris (Dezember 2021). Online: youtube.com/watch?v=OanwYoajPQw (13.08.2022).

[41] GOTTSCHLING P.: FORSCHUNG mit modernem C++. C++17-INTENSIVKURS für Wissenschaftler, Ingenieure und Programmierer. Carl Hanser Verlag, München (2019). ISBN: 978-3-446-45981-6.

[42] GRÄSMAN K. ET AL.: Include What You Use. University of Illinois at Urbana-Champaign (2022). Online: github.com/include-what-you-use/include-what-you-use (13.08.2022).

[43] GREGOR D. ET AL.: N2080: Variadic Templates (Revision 3). ISO/IEC C++ Standards Committee (September 2006). Online: wg21.link/n2080 (13.08.2022).

[44] GREGOR D.: N3051: Deprecating Exception Specifications. ISO/IEC C++ Standards Committee (März 2010). Online: wg21.link/n3051 (13.08.2022).

[45] GRIGORIEV V.: LWG issue 2268: Setting a default argument in the declaration of a member function `assign` of `std::basic_string`. ISO/IEC C++ Standards Committee (November 2016). Online: wg21.link/lwg2268 (13.08.2022).

[46] GRIGORYAN V. & WU S.: Expert C++. Become a proficient programmer by learning coding best practices with C++17 and C++20's latest features. Packt Publishing Ltd., Birmingham (April 2020). ISBN: 978-1-83855-265-7.

[47] HERRING S.: P1779R3: ABI isolation for member functions. ISO/IEC C++ Standards Committee (Januar 2020). Online: wg21.link/p1779r3 (13.08.2022).

[48] HINNANT H.: LWG issue 2064: More `noexcept` issues in `basic_string`. ISO/IEC C++ Standards Committee (November 2016). Online: wg21.link/lwg2064 (13.08.2022).

[49] HINNANT H. ET AL.: N1377: A Proposal to Add Move Semantics Support to the C++ Language. ISO/IEC C++ Standards Committee (September 2002). Online: wg21.link/n1377 (13.08.2022).

[50] HINNANT H.: N1856: Rvalue Reference Recommendations for Chapter 20. ISO/IEC C++ Standards Committee (August 2005). Online: wg21.link/n1856 (13.08.2022).

[51] HINNANT H.: N1860: Rvalue Reference Recommendations for Chapter 25. ISO/IEC C++ Standards Committee (August 2005). Online: wg21.link/n1860 (13.08.2022).

[52] HORTON I. & VAN WEERT P.: Beginning C++20. From Novice to Professional. Sixth Edition. Apress Media LLC, New York (2020). ISBN: 978-1-4842-5884-2.

[53] ISO.ORG: ISO/IEC 14882:2020. Programming languages – C++. International Organization for Standardization, Genf (Dezember 2022). Online: iso.org/standard/79358.html (13.08.2022).

[54] ISOCPP.ORG: Results summary: 2022 Annual C++ Developer Survey „Lite“. Standard C++ Foundation (Juni 2022). Online: isocpp.org/blog/2022/06/results-summary-2022-annual-cpp-developer-survey-lite (13.08.2022).

[55] JÄRVI J. ET AL.: N1478: Decltype and auto. ISO/IEC C++ Standards Committee (April 2003). Online: wg21.link/n1478 (13.08.2022).

[56] JÄRVI J. ET AL.: N1978: Decltype (revision 5). ISO/IEC C++ Standards Committee (April 2006). Online: wg21.link/n1978 (13.08.2022).

[57] JÄRVI J. ET AL.: N2529: Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 3). ISO/IEC C++ Standards Committee (Februar 2008). Online: wg21.link/n2529 (13.08.2022).

[58] JOHNSON CJ: P1331R2: Permitting trivial default initialization in constexpr contexts. ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p1331r2 (13.08.2022).

[59] KAWULAK R.: N4230: Nested namespace definition (revision 2). ISO/IEC C++ Standards Committee (Oktober 2014). Online: wg21.link/n4230 (13.08.2022).

[60] KEANE E.: P1668R1: Enabling `constexpr` Intrinsics By Permitting Unevaluated inline-assembly in `constexpr` Functions. ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p1668r1 (13.08.2022).

[61] KNIGHT J.: [Libcxx] Add <source_location> header. The LLVM Compiler Infrastructure (2022). Online: reviews.llvm.org/D120634 (13.08.2022).

[62] KOENIG A.: N1146: Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++. ISO/IEC C++ Standards Committee (November 1997). Online: open-std.org/jtc1/sc22/wg21/docs/wp/pdf/nov97-2/ (13.08.2022).

[63] KÖPPE, T.: N4910: Working Draft, Standard for Programming Language C++. ISO/IEC C++ Standards Committee (März 2022). Online: wg21.link/n4910 (13.08.2022).

[64] KÖPPE, T.: P0305R1: Selection statements with initializer. ISO/IEC C++ Standards Committee (Juni 2016). Online: wg21.link/p0305r1 (13.08.2022).

[65] KÖPPE, T.: P0614R1: Range-based `for` statements with initializer. ISO/IEC C++ Standards Committee (November 2017) Online: wg21.link/p0614r1 (13.08.2022).

[66] LAKOS J. ET AL.: Embracing Modern C++ *Safely*. Pearson Education, Inc., Boston (2022). ISBN: 978-0-13-738035-0.

[67] LANTINGA S. ET AL.: Simple DirectMedia Layer (SDL) Version 2.0. (2022). Online: github.com/libsdl-org/SDL (13.08.2022).

[68] LATTNER C. ET AL.: The LLVM Compiler Infrastructure. (2022). Online: github.com/llvm/llvm-project/ (13.08.2022).

[69] LAVAVEJ S.: N3421: Making Operator Functors greater<>. ISO/IEC C++ Standards Committee (September 2012). Online: wg21.link/n3421 (13.08.2022).

[70] LAVAVEJ S.: N3588: make_unique (Revision 1). ISO/IEC C++ Standards Committee (März 2013). Online: wg21.link/n3588 (13.08.2022).

[71] LAVAVEJ S.: N3657: Adding heterogeneous comparison lookup to associative containers (rev 4). ISO/IEC C++ Standards Committee (März 2013). Online: wg21.link/n3657 (13.08.2022).

[72] LAVAVEJ S.: N4190: Removing auto_ptr, random_shuffle(), And Old <functional> Stuff. ISO/IEC C++ Standards Committee (Oktober 2014). Online: wg21.link/n4190 (13.08.2022).

[73] LLVM.ORG: AddressSanitizer. The LLVM Compiler Infrastructure (2022). Online: clang.llvm.org/docs/AddressSanitizer.html (28.06.2022).

[74] LLVM.ORG: Clang-Tidy Checks. The LLVM Compiler Infrastructure (2022). Online: clang.llvm.org/extra/clang-tidy/checks/list.html (20.06.2022).

[75] LLVM.ORG: C++ Support in Clang. The LLVM Compiler Infrastructure (2022). Online: clang.llvm.org/cxx_status.html (07.07.2022).

[76] LLVM.ORG: Documentation. The LLVM Compiler Infrastructure (2022). Online: llvm.org/docs/ (20.06.2022).

[77] LLVM.ORG: LLVM 2.8 Release Notes. The LLVM Compiler Infrastructure (Oktober 2010). Online: releases.llvm.org/2.8/docs/ReleaseNotes.html (13.08.2022).

[78] LLVM.ORG: Modules. The LLVM Compiler Infrastructure (2022). Online: clang.llvm.org/docs/Modules.html (25.07.2022).

[79] LLVM.ORG: Precompiled Header and Modules Internals. The LLVM Compiler Infrastructure (2022). Online: clang.llvm.org/docs/PCHInternals.html (20.06.2022).

[80] MACINTOSH N. & LAVAVEJ S.: P0122R7: span: bounds-safe views for sequences of objects. ISO/IEC C++ Standards Committee (März 2018). Online: wg21.link/p0122r7 (13.08.2022).

[81] MADDOCK J.: N1424: A Proposal to add Type Traits to the Standard Library. ISO/IEC C++ Standards Committee (März 2003). Online: wg21.link/n1424 (13.08.2022).

[82] MALTSEV M.: P0457R2: String Prefix and Suffix Checking. ISO/IEC C++ Standards Committee (November 2017). Online: wg21.link/p0457r2 (13.08.2022).

[83] MAURER J. & WONG M.: N2761: Towards support for attributes in C++ (Revision 6). ISO/IEC C++ Standards Committee (September 2008). Online: wg21.link/n2761 (13.08.2022).

[84] MAURER J. ET AL.: N3206: Override control: Eliminating Attributes. ISO/IEC C++ Standards Committee (November 2010). Online: wg21.link/n3206 (13.08.2022).

[85] MAURER J.: P0012R1: Make exception specifications be part of the type system, version 5. ISO/IEC C++ Standards Committee (Oktober 2015). Online: wg21.link/p0012r1 (13.08.2022).

[86] MAURER J.: P0292R2: constexpr if: A slightly different syntax. ISO/IEC C++ Standards Committee (Juni 2016). Online: wg21.link/p0292r2 (13.08.2022).

[87] MAURER J.: P2360R0: Extend *init-statement* to allow *alias-declaration*. ISO/IEC C++ Standards Committee (April 2021). Online: wg21.link/p2360R0 (13.08.2022).

[88] MENEIDE J. & WIEDIJK F.: N2912: Programming languages – C. ISO/IEC C (Jun 2022). Online: open-std.org/jtc1/sc22/wg14/www/docs/n2912.pdf (13.08.2022).

[89] MENEIDE J. & MUERTE I.: P1301R4: [[nodiscard(„should have a reason“)]]. ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p1301r4 (13.08.2022).

[90] MENEIDE J.: P1682R3: std::to_underlying for enumarations. ISO/IEC C++ Standards Committee (Januar 2021). Online: wg21.link/p1682r3 (13.08.2022).

[91] MEREDITH A. ET AL.: N2668: Concurrency Modifications to Basic String. ISO/IEC C++ Standards Committee (Juni 2008). Online: wg21.link/n2668 (13.08.2022).

[92] MEREDITH A.: P0006R0: Adopt Type Traits Variable Templates from Library Fundamentals TS for C++17. ISO/IEC C++ Standards Committee (September 2015). Online: wg21.link/p0006r0 (13.08.2022).

[93] MEREDITH A.: P0174R2: Deprecating Vestigial Library Parts in C++17. ISO/IEC C++ Standards Committee (Juni 2016). Online: wg21.link/p0174r2 (13.08.2022).

[94] MEREDITH A. ET AL.: P0619R4: Reviewing Deprecated Facilities of C++17 for C++20. ISO/IEC C++ Standards Committee (Juni 2018). Online: wg21.link/p0619r4 (13.08.2022).

[95] MEREDITH A.: P1094R2: Nested Inline Namespaces. ISO/IEC C++ Standards Committee (November 2018). Online: wg21.link/p1094r2 (13.08.2022).

[96] MERRILL J.: N2535: Namespace Association („inline namespace“). ISO/IEC C++ Standards Committee (Februar 2008). Online: wg21.link/n2535 (13.08.2022).

[97] MERRILL J. & VANDEVOORDE D.: N2640: Initializer Lists — Alternative Mechanism and Rationale (v. 2). ISO/IEC C++ Standards Committee (Mai 2008). Online: wg21.link/n2640 (13.08.2022).

[98] MERRIL J.: N3638: Return type deduction for normal functions. ISO/IEC C++ Standards Committee (April 2013). Online: wg21.link/n3638 (13.08.2022).

[99] MEYERS S: Effective Modern C++. 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media, Inc., Sebastopol (Mai 2018). ISBN: 978-1-491-90399-5.

[100] MILLER D. ET AL.: N2347: Strongly Typed Enums (revision 3). ISO/IEC C++ Standards Committee (Juli 2007). Online: wg21.link/n2347 (13.08.2022).

[101] MUERTE I. & JABOT C.: P1272R4: Byteswapping for fun&&nuf. ISO/IEC C++ Standards Committee (September 2021). Online: wg21.link/p1272r4 (13.08.2022).

[102] MÜLLER K.: Archetypische Rekomposition. Kann Vererbung in C++ durch eine performantere Alternative abgelöst werden? Forschungsbericht Hochschule Mittweida, Fakultät Angewandte Computer- und Biowissenschaften (März 2021).

[103] MUTZ M.: P0646R1: Improving the Return Value of Erase-Like Algorithms I: `list`/`forward_list`. ISO/IEC C++ Standards Committee (Juni 2018). Online: wg21.link/p0646r1 (13.08.2022).

[104] NAUMANN A.: P0086R0: Variant design review. ISO/IEC C++ Standards Committee (September 2015). Online: wg21.link/p0086r0 (13.08.2022).

[105] NELSON C.: P0035R4: Dynamic memory allocation for over-aligned data. ISO/IEC C++ Standards Committee (Juni 2016). Online: wg21.link/p0035r4 (13.08.2022).

[106] NIEBLER E.: P0184R0: Generalizing the Range-Based For Loop. ISO/IEC C++ Standards Committee (Februar 2011). Online: wg21.link/p0184r0 (13.08.2022).

[107] NIEBLER E. ET AL.: P0896R4: The One Ranges Proposal. ISO/IEC C++ Standards Committee (November 2018). Online: wg21.link/p0896r4 (13.08.2022).

[108] NINJA-BUILD.ORG: The Ninja build system. v1.11.0. ninja-build.org (2022). Online: ninja-build.org/manual.html (13.08.2022).

[109] OTTOSEN T.: N1868: Proposal for new for-loop (revision 1). ISO/IEC C++ Standards Committee (August 2005). Online: wg21.link/n1868 (13.08.2022).

[110] PERSSON B.: LWG issue 2069: Inconsistent exception spec for `basic_string` move constructor. ISO/IEC C++ Standards Committee (Februar 2016). Online: wg21.link/lwg2069 (13.08.2022).

[111] PION S.: LWG issue 767: Forwarding and backward compatibility. ISO/IEC C++ Standards Committee (Februar 2016). Online: wg21.link/lwg767 (13.08.2022).

[112] REVZIN B.: P1185R2: <=> != ==. ISO/IEC C++ Standards Committee (Februar 2019). Online: wg21.link/p1185r2 (13.08.2022).

[113] REVZIN B.: P1186R3: When do you actually use <=>? ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p1186r3 (13.08.2022).

[114] REVZIN B.: P1614R2: The Mothership has Landed. Adding <=> to the Library. ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p1614r2 (13.08.2022).

[115] REVZIN B.: P1959R0: Remove `std::weak_equality` and `std::strong_equality`. ISO/IEC C++ Standards Committee (November 2019). Online: wg21.link/p1959r0 (13.08.2022).

[116] ROJTBERG P. ET AL.: API Reference. Torus Knot Software Ltd. (2022). Online: ogrecave.github.io/ogre/api/13/ (13.08.2022).

[117] ROJTBERG P. ET AL.: Object-Oriented Graphics Rendering Engine. Torus Knot Software Ltd. (2022). Online: github.com/ogrecave/ogre (13.08.2022).

[118] ROTH S.: Clean C++20. Sustainable Software Development Patterns and Best Practices. Second Edition. Apress Media LLC, New York (2021). ISBN: 978-1-4842-5949-8.

[119] SÁNCHEZ J.: LWG issue 2003: String exception inconsistency in erase. ISO/IEC C++ Standards Committee (November 2016). Online: wg21.link/lwg2003 (13.08.2022).

[120] SCHUTS M. ET AL.: Large-scale semi-automated migration of legacy C/C++ test code. Software: Practice and Experience 2022; 52(7): 1543-1580. John Wiley & Sons, Inc. (März 2022). doi: 10.1002/spe.3082 (13.08.2022).

[121] SHEN T.: P0329R0: Designated Initialization. ISO/IEC C++ Standards Committee (Mai 2016). Online: wg21.link/p0329r0 (13.08.2022).

[122] SMITH R.: N3652: Relaxing constraints on `constexpr` functions. `constexpr` member functions and implicit `const`. ISO/IEC C++ Standards Committee (April 2013). Online: wg21.link/n3652 (13.08.2022).

[123] SMITH R.: P0135R0: Guaranteed copy elision through simplified value categories. ISO/IEC C++ Standards Committee (September 2015). Online: wg21.link/p0135r0 (13.08.2022).

[124] SMITH R.: P0947R1: Another take on Modules. ISO/IEC C++ Standards Committee (März 2018). Online: wg21.link/p0947r1 (13.08.2022).

[125] SMITH R.: P1103R3: Merging Modules. ISO/IEC C++ Standards Committee (Februar 2019). Online: wg21.link/p1103r3 (13.08.2022).

[126] SNYDER J. & DIONNE L.: P0732R2: Class Types in Non-Type Template Parameters. ISO/IEC C++ Standards Committee (Juni 2018). Online: wg21.link/p0732r2 (13.08.2022).

[127] SPENCER M.: P1857R3: Modules Dependency Discovery. ISO/IEC C++ Standards Committee (February 2020). Online: wg21.link/p1857r3 (13.08.2022).

[128] SPERTUS M. & SEYMOUR B.: N2756: Non-static data member initializers. ISO/IEC C++ Standards Committee (September 2008). Online: wg21.link/n2756 (13.08.2022).

[129] STATES U.: LWG issue 3330: Include `<compare>` from most library headers. ISO/IEC C++ Standards Committee (Februar 2021). Online: wg21.link/lwg3330 (13.08.2022).

[130] STROUSTRUP B. & SUTTER H.: C++ Core Guidelines. Standard C++ Foundation and its contributors (April 2022). Online: isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines (13.08.2022).

[131] STROUSTRUP B.: C++11 - the new ISO C++ standard. stroustrup.com (August 2016). Online: stroustrup.com/C++11FAQ.html (13.08.2022).

[132] SUTTER H. & GLASSBOROW F.: N1986: Delegating Constructors (revision 3). ISO/IEC C++ Standards Committee (April 2006). Online: wg21.link/n1986 (13.08.2022).

[133] SUTTER H. & STROUSTRUP B.: N2431: A name for the null pointer: nullptr (revision 4). ISO/IEC C++ Standards Committee (Oktober 2007). Online: wg21.link/n2431 (13.08.2022).

[134] SUTTER H. ET AL.: P0144R2: Structured bindings. ISO/IEC C++ Standards Committee (März 2016). Online: wg21.link/p0144r2 (13.08.2022).

[135] SUTTER H. ET AL.: P0515R3: Consistent comparison. ISO/IEC C++ Standards Committee (November 2017). Online: wg21.link/p0515r3 (13.08.2022).

[136] SUTTON A. & SMITH R.: N4191: Folding expressions. ISO/IEC C++ Standards Committee (Oktober 2014). Online: wg21.link/n4191 (13.08.2022).

[137] SUTTON A.: P0734R0: Wording Paper, C++ extensions for Concepts. ISO/IEC C++ Standards Committee (Juli 2017). Online: wg21.link/p0734r0 (13.08.2022).

[138] TALBOT A.: N2680: Proposed Wording for Placement Insert (Revision 1). ISO/IEC C++ Standards Committee (Juni 2008). Online: wg21.link/n2680 (13.08.2022).

[139] TOMAZOS A.: P0068R0: Proposal of [[unused]], [[nodiscard]] and [[fallthrough]] attributes. ISO/IEC C++ Standards Committee (September 2015). Online: wg21.link/p0068r0 (13.08.2022).

[140] UBUNTU.COM: Jammy Jellyfish Release Notes. Canonical Ltd. (2022). Online: discourse.ubuntu.com/t/jammy-jellyfish-release-notes/24668 (13.08.2022).

[141] VALGRIND.ORG: Valgrind Documentation. Valgrind Developers (2022). Online: valgrind.org/docs/manual (13.08.2022).

[142] VALI F. ET AL.: N3559: Proposal for Generic (Polymorphic) Lambda Expressions (Revision 2). ISO/IEC C++ Standards Committee (März 2013). Online: wg21.link/n3559 (13.08.2022).

[143] VANDEVOORDE D.: N2927: New wording for C++0x Lambdas (rev. 2). ISO/IEC C++ Standards Committee (Juli 2009). Online: wg21.link/n2927 (13.08.2022).

[144] VOUTILAINEN V. ET AL.: N2928: Explicit Virtual Overrides. ISO/IEC C++ Standards Committee (Juli 2009). Online: wg21.link/n2928 (13.08.2022).

[145] VOUTILAINEN V.: P0960R0: Allow initializing aggregates from parenthesized list of values. ISO/IEC C++ Standards Committee (Februar 2018). Online: wg21.link/p0960r0 (13.08.2022).

[146] VOUTILAINEN V. ET AL.: P1141R2: Yet another approach for constrained declarations. ISO/IEC C++ Standards Committee (November 2018). Online: wg21.link/p1141r2 (13.08.2022).

[147] WAKELY J.: LWG issue 2210: Missing allocator-extended constructor for allocator-aware containers. ISO/IEC C++ Standards Committee (Februar 2016). Online: wg21.link/lwg2210 (13.08.2022).

[148] WAKELY J. & KENNELLY C.: P0401R6: Providing size feedback in the Allocator interface. ISO/IEC C++ Standards Committee (Januar 2021). Online: wg21.link/p0401r6 (13.08.2022).

[149] WILL T.: C++: Das umfassende Handbuch. Rheinwerk Verlag GmbH, Bonn (Juli 2020). ISBN: 978-3-8362-7595-8.

[150] YASSKIN J: N3921: `string_view`: a non-owning reference to a string, revision 7. ISO/IEC C++ Standards Committee (Februar 2014). Online: wg21.link/n3921 (13.08.2022).

[151] YUAN Z.: P0849R8: auto(x): decay-copy in the language. ISO/IEC C++ Standards Committee (Juli 2021). Online: wg21.link/p0849r8 (13.08.2022).

[152] ZVEROVICH V.: P0645R10: Text Formatting. ISO/IEC C++ Standards Committee (Juli 2019). Online: wg21.link/p0645r10 (13.08.2022).

[153] ZVEROVICH V.: P2216R3: std::format improvements. ISO/IEC C++ Standards Committee (Februar 2021). Online: wg21.link/p2216r3 (13.08.2022).

Glossar

Argument Ein Argument ist eine Entität, welche an den Parameter einer (Macro)-Funktion oder eines Templates gebunden wird. Argumente werden durch Kommas getrennt, bei binären Operatoren durch das Symbol des jeweiligen Operators. Für Funktionen erfolgt mit Argumenten ein Aufruf, für Templates erfolgt eine Instanziierung.

Attribut Mit Attributen können Entitäten im Quellcode mit zusätzlichen Eigenschaften notiert werden. Es gibt in C++ sowohl standardisierte Attribute als auch Compiler-spezifische.

Ausrichtung Die Ausrichtung eines Typs bestimmt, an welchen Speicheradressen sich Objekte befinden dürfen, die entweder diesem Typen angehören oder ein Unterobjekt dieses Typs haben. Beispielsweise hat ein Pointer auf einer 64 Bit Architektur eine Ausrichtung von 8 Bytes. Das heißt, dass die letzten drei Bytes der Speicheradresse eines Objektes, welches einen Pointer hält, immer 0 sind. Damit die Ausrichtung der Unterobjekte innerhalb eines Objektes immer gewährleistet wird, wird gegebenenfalls Padding verwendet um die Speicheradresse aufzurunden.

Container Sammelbegriff für alle Objekte, in denen eine variable Zahl anderer Objekte enthalten sein kann. In der Regel können Container dynamisch wachsen. Es gibt aber auch Container mit statischer Größe.

Definition Eine Definition weißt einer Deklaration Bedeutung zu. Beispiele für Definitionen sind der Körper einer Funktion oder die Member-Deklarationen in einer Klasse. In vielen Fällen finden Deklaration und Definition gleichzeitig statt. Zu jeder Deklaration darf es nur eine Definition geben. Gegebenenfalls sind mehrere erlaubt, sofern diese identisch sind.

Deklaration Mit einer Deklaration kann ein Name an eine Entität gebunden werden. Über ihren Namen ist sie dann im folgenden Kontext verwendbar. Bei einer Forwärts-Deklaration handelt es sich um Deklarationen benutzerdefinierter Typen, die ihrer Definition vorausgehen. Dadurch kann der Name der Entität bereits verwendet werden, ohne dass eine direkte Abhängigkeit zur Definition entsteht.

Einschluss Das Einschließen einer Datei in einer andere erfolgt mit der `#include` Anweisung. Dabei wird der Text der eingeschlossenen Datei in die einschließende Datei kopiert bevor letztere weiterverarbeitet wird. Meistens werden Header-Dateien eingeschlossen.

Entität Ein Oberbegriff für alle Werte, Objekte, Referenzen, *structured bindings*, Funktionen, Enumeratoren, Typen, Member einer Klasse, Bit-Felder, Templates, Template-Spezialisierungen, Namespaces und Packs[§ 6.1][63].

Fork Eine Abzweigung eines bestehenden git Repositories zu einem bestimmen Zeitpunkt. Sie kann unabhängig davon weitergeführt werden oder nach Wunsch Updates aus dem Original integrieren.

Header Ein Header ist eine etablierte Datei-Konvention, die C++ von C übernommen hat. Innerhalb eines Headers befinden sich oft Deklarationen. Dadurch kann auf diese unabhängig von ihrer jeweiligen Definition zugegriffen werden. In bestimmten Fällen sind auch Definitionen vertreten. Das

ist gegebenenfalls erlaubt, allerdings nur dann, wenn alle Definitionen derselben Deklaration in allen Translations-Einheiten identisch sind. Seit C++20 ist es möglich, einen Header als Header-Unit zu kompilieren. Dabei wird dessen Inhalt so interpretiert, als wäre er ein Modul, welches seinen gesamten Inhalt exportiert. Anders als bei Modulen schließt dies jedoch auch Macros mit ein.

Integral Oberbegriff für alle Daten-Typen in C++, die ganze Zahlen repräsentieren. Beispiele hierfür sind `char` und `int`.

Objekt Objekte entstehen zur Laufzeit eines Programms. Ein Objekt hat einen Typen, der genauere Eigenschaften bestimmt, und belegt bis zu seiner Zerstörung Speicher. In dem Speicher eines Objektes können sich weitere Objekte – die Unterobjekte – befinden.

Padding Die Größe eines Objektes ist durch ein natürliches Vielfaches seiner Ausrichtung vorgegeben. Alle Bits, welche nicht zum Wert des Objektes beitragen, sich aber in dessen Speicher befinden, werden als Padding bezeichnet.

Parameter Ein Parameter kann den Objekt-Typen vorgeben, in welchen Argumente einer Funktion beim Aufrufen konvertiert werden. In Templates sind neben Objekten zudem Typen und Klassen-Templates als Parameter erlaubt. In Macros wird jede Art von Text akzeptiert. Ein Parameter kann mit einem Namen versehen werden, über den im folgenden Kontext auf den Wert übergebener Argumente zugegriffen werden kann.

Pointer Ein Pointer verweist auf eine bestimmte Stelle im Speicher. Über einen Pointer kann auf Objekte, welche in diesem Speicher liegen, zugegriffen werden. Verweist der Pointer auf eine Funktion, so kann diese über den Pointer aufgerufen werden.

Standard-Layout Standard-Layout ist eine Eigenschaft bestimmter Typen, die gewisse Optimierungen erlaubt. Für skalare Typen ist sie immer erfüllt. Für Klassen tritt sie dann auf, wenn sich die Deklaration aller nicht-statischen Daten-Member innerhalb einer Klasse befinden, darin dieselbe Sichtbarkeit haben und ebenfalls Standard-Layout aufweisen. Zudem darf die Klasse keine virtuellen Funktionen oder virtuellen Basis-Klassen haben. Arrays erfüllen die Eigenschaft, wenn alle ihre Elemente sie erfüllen.

Template Für bestimmte Entitäten kann ein Template angelegt werden. Es akzeptiert eine vorgegebene Anzahl an Argumenten und injiziert diese in seine Definition. Daraus ensteht dann eine neue Entität, eine Instanz des Templates. Dies kann beispielsweise eine Klasse oder Funktion sein. Das Template wird dabei *instanziiert*.

Translations-Einheit Der Quellcode, der entsteht, nachdem alle in eine Datei eingeschlossenen Header aufgelöst wurden. Translations-Einheiten werden unabhängig voneinander kompiliert. Dabei entsteht die sog. Objekt-Datei. Im Link-Prozess werden sie schließlich zusammengeführt. Dabei darf für dieselbe Deklaration insgesamt nur eine Definition vorliegen. Es gelten jedoch Ausnahmen, in denen mehrere Definitionen existieren dürfen, solange sie identisch sind.

Trivial Bestimme Operationen wie das Erstellen, Zerstören, Kopieren oder Verschieben eines Objektes können trivial sein. Das heißt, sie erfolgen ohne Nebeneffekte. Das Erstellen oder Zerstören eines trivialen Objektes führt keine Schreibvorgänge auf dessen Speicher aus. Das Kopieren oder Verschieben eines trivialen Objektes ist äquivalent zum Kopieren der einzelnen Bytes des Objektes.

Eidesstattliche Erklärung

Hiermit versichere ich – Kai Philipp Müller – an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Mittweida, 15. August 2022 — Kai Philipp Müller, B.Sc.

Ort, Datum