Taskbasierte Umsetzung des parallelen hierarchischen Radiosity-Verfahrens in Nim mithilfe der Multithreading-Runtime Weave mit Untersuchung und Optimierung des Laufzeitverhaltens

Fabian Vießmann

Bayreuth Reports on Parallel and Distributed Systems

No. 15, März 2022

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Bachelorarbeit

Fabian Vießmann

10. Juni 2021

Universität Bayreuth

Fakultät Mathematik, Physik, Informatik
Institut für Informatik
Lehrstuhl Angewandte Informatik II - Parallele und verteilte Systeme

Bachelorarbeit

Taskbasierte Umsetzung des parallelen hierarchischen Radiosity-Verfahrens in Nim mithilfe der Multithreading-Runtime Weave mit Untersuchung und Optimierung des Laufzeitverhaltens

Task-based implementation of the parallel hierarchical radiosity method in Nim using the multithreading runtime Weave with analysis and optimization of the runtime behavior

Fabian Vießmann

1. Prüfer PD Dr. Matthias Korch
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

2. Prüfer Prof. Dr. Thomas Rauber
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

Betreuer PD Dr. Matthias Korch

Abgabedatum: 10. Juni 2021

Fabian Vießmann
Frankenwaldstraße 23
95369 Untersteinach
Matrikelnummer: 1565530

Bachelorarbeit
Taskbasierte Umsetzung des parallelen hierarchischen Radiosity-Verfahrens in Nim mithilfe der Multithreading-Runtime Weave mit Untersuchung und Optimierung des Laufzeitverhaltens, 10. Juni 2021
Prüfer: PD Dr. Matthias Korch und Prof. Dr. Thomas Rauber
Betreuer: PD Dr. Matthias Korch

Universität Bayreuth
Lehrstuhl Angewandte Informatik II - Parallele und verteilte Systeme
Institut für Informatik
Fakultät Mathematik, Physik, Informatik
Universitätsstrasse 30
95447 Bayreuth
Deutschland

Zusammenfassung

Das hierarchische Radiosity-Verfahren beschäftigt sich mit dem Problem der photorealistischen Darstellung von Bildern ausgehend von einem mathematischen Modell, welches eine Menge von Polygonen enthält. Die Grundlage hierzu bildet im Wesentlichen die Lösung der Rendering-Gleichung, welche jedoch sehr aufwendig zu lösen ist und es einer effizienteren Approximation bedarf, wie beispielsweise der in dieser Arbeit vorgestellten Radiosity-Gleichung. Um die volle Leistung der heutigen Rechner auszunutzen, bietet sich eine Parallelisierung des Programms an, was aufgrund der irregulären Struktur des Algorithmus jedoch nur auf taskbasierter Ebene erfolgen kann, sodass eine effiziente Lastbalancierung gewährleistet wird. Sowohl die Realisierung dieses Ansatzes und dessen theoretische Grundlagen als auch weitere Performanceanalysen sollen in der vorliegenden Arbeit präsentiert werden.

Abstract

The hierarchical radiosity method deals with the problem of the photorealistic representation of images based on a mathematical model that contains a number of polygons. The basis for this is essentially the solution of the rendering equation which is very difficult to solve and requires a more efficient approximation, such as the radiosity equation presented in this work. In order to utilize the full performance of today's computers, a parallelization of the program is recommended which can only be done on a task-based level due to the irregular structure of the algorithm, so that efficient load balancing can be guaranteed. Both the implementation of this approach and its theoretical basis as well as further performance analyzes are to be presented in this work.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Problembeschreibung 2

1.3 Verwandte Arbeiten 2

2 Theoretische Grundlagen **3**

2.1 Irreguläre Algorithmen und Parallelisierung durch Taskpools 3

2.2 Beleuchtungsmodelle und das Radiosity-Verfahren 5

2.2.1 Globale und lokale Beleuchtungsmodelle 5

2.2.2 Grundlagen des klassischen Radiosity-Verfahrens 6

2.2.2.1 Definition des Radiosity-Verfahrens 6

2.2.2.2 Die Radiosity-Gleichung 7

2.2.2.3 Berechnung der Formfaktoren 8

2.2.2.4 Grundstruktur des Algorithmus 9

2.2.3 Das hierarchische Radiosity-Verfahren 10

3 Beschreibung der Programmiersprache Nim und der Multithreading-Runtime Weave **13**

3.1 Überblick über Nim 13

3.2 Überblick über Weave 15

4 Vorstellung der Implementierung **17**

4.1 Architektur 17

4.1.1 Abhängigkeiten 17

4.1.2 Struktur 18

4.1.3 Nutzerschnittstelle 18

4.2 Beschreibung der konkreten Implementierung des hierarchischen Radiosity-Verfahrens 19

4.2.1 Verwendete Datenstrukturen 19

4.2.2 Phase 1: Einlesen der Eingabedaten und Einsortierung in den BSP-Baum 20

4.2.3 Phase 2: Berechnung des Strahlungsaustauschs und Verfeinerung der Elemente 24

4.2.4 Phase 3: Normalisierung 28

4.2.5 Parallelisierung mit Weave 28
4.2.6 Optimierungen 33
4.2.6.1 BSP-Baum 33
4.2.6.2 Einführung eines Testbereichs in der Sichtbarkeitsberechnung 33
4.2.6.3 Einführung eines Patch-Caches in der Sichtbarkeitsberechnung 34
4.2.6.4 Einführung von Objekt-Caches 34

5 Experimentelle Auswertung **35**
5.1 Testprobleme 35
5.2 Verwendetes System 36
5.3 Ergebnisse 37
5.3.1 AMD Epyc 7551 (x86-64) 37
5.3.1.1 Analyse der eigenen Implementierung 37
5.3.1.2 Vergleich mit der modifizierten SPLASH-2 Variante . 41
5.3.2 Thunder X2 (ARM) 45

6 Fazit und Ausblick **49**

Literaturverzeichnis **51**

1 Einleitung

1.1 Motivation

In den vergangenen Jahren hat das Gebiet der *Computergraphik* immer mehr an Bedeutung gewonnen. Dabei bezeichnet *Computergraphik* die Verwendung von Rechnern für das Erstellen und Manipulieren von Bildern, wobei vor allem drei Anwendungsgebiete von wesentlicher Bedeutung sind: Modellierung, Rendering und Animation (vgl. [SM09, S. 1]). Die Modellierung beschäftigt sich mit der Beschreibung von Objekten durch mathematische Primitive, beispielsweise durch 3D-Punkte oder 3D-Dreiecke, deren Gesamtheit dann ein *Modell* oder eine *Szene* bilden. Auf Grundlage dieses Modells können dreidimensionale, schattierte Bilder erstellt werden, was Rendering genannt wird. Verwendet man nun eine Abfolge von mehreren gerenderten Bildern, spricht man von Animation (vgl. [SM09, S. 1]).

Doch wie genau kann ein dreidimensionales Modell realistisch dargestellt werden? Im Wesentlichen beruht dies auf einer Projektion der Szene auf eine zweidimensionale Ebene, welche von Bildschirmen und somit von Pixeln wiedergegebenen werden kann (vgl. [RR00, S. 519]). Um dabei Farben darstellen zu können, wird jedem Eckpunkt eines mathematischen Primitivs, welches beispielsweise ein Dreieck ist, ein Farbwert zugewiesen, welcher dann zwischen allen Eckpunkten interpoliert wird, um die Gesamtfarbe zu erhalten (vgl. [SM09, S. 44]). Dies bildet die wichtige Grundlage zur Schattierung eines Bildes, welche der fundamentale Aspekt für eine realistische Darstellung ist. Dabei muss ein Farbwert so angepasst werden, damit eine Beleuchtung oder auch eine Reflexion abgebildet werden kann. Die Voraussetzung hierfür ist die Einbeziehung einer Lichtquelle und die Berechnung des Strahlungsaustauschs, um somit Licht und Schatten zu simulieren (vgl. [RR00, S. 519f]).

Für dieses Problem gibt es mittlerweile eine Vielzahl an Verfahren, die dieses lösen oder eher approximieren können, darunter auch der *Radiosity-Algorithmus*, welcher *betrachterunabhängig* in einer Szene von Objekten in einem geschlossenen Raum diffuses Licht simuliert (vgl. [RR00, S. 520]). Doch weshalb ist es nötig, derartige Algorithmen zu optimieren und gegebenenfalls auch parallel auszuführen?

1.2 Problembeschreibung

Um die Notwendigkeit der Optimierung und Parallelität dieser Verfahren zu verdeutlichen, wird nun die zugrundeliegende Problemstellung genauer beschrieben. Im Allgemeinen kann man den Strahlungsaustausch und somit auch die Schattierung durch die sogenannte *Rendering-Gleichung* berechnen, welche allerdings analytisch eher schwierig zu lösen ist. Daher gibt es verschiedene Verfahren, die diese Gleichung approximieren, darunter beispielsweise der Radiosity-Algorithmus (vgl. [Nü05, S. 5ff]). Korrespondierend zum Radiosity-Verfahren, gibt es den hierarchischen Radiosity Algorithmus, der die Problemstellung hierarchisch unterteilt und auf Grundlage dessen eine bessere Laufzeit vorweisen kann. Diese beiden Verfahren sind Vertreter der *irregulären Algorithmen*, bei denen das Berechnungsverhalten stark von der Eingabe abhängt und das Zugriffsverhalten auf Datenstrukturen a priori nicht vorhergesagt werden kann. Dies macht eine Parallelisierung mit manueller Lastbalancierung eher schwierig, weshalb in dieser Arbeit ein taskorientierter Ansatz vorgestellt wird, welcher die Last gleichmäßig verteilen soll (vgl. [RR00, S. 499ff]).

Im Rahmen dieser Bachelorabeit werden zunächst einige theoretische Grundlagen dargelegt. Dabei werden irreguläre Algorithmen und deren Parallelisierung durch Taskpools beschrieben. Da der Radiosity-Algorithmus ein Vertreter der irregulären Algorithmen ist, wird dieser nun genauer betrachtet. Darunter soll aufgezeigt werden, was globale und lokale Beleuchtungsmodelle sind und welches mathematische Modell zugrunde liegt. Anschließend werden noch einige Grundlagen zum klassischen als auch hierarchischen Radiosity-Verfahren dargelegt. Da das Programm in der modernen Programmiersprache *Nim* mithilfe der Multithreading-Runtime *Weave* geschrieben wurde, soll dies ebenfalls erläutert werden. Abschließend wird die Architektur des eigenen Programms beschrieben und Laufzeitmessungen betrachtet.

1.3 Verwandte Arbeiten

Die Thematik des hierarchischen Radiosity-Verfahrens ist schon in einigen Arbeiten bearbeitet worden. Als Hauptreferenz ist hier die SPLASH-2 Suite[1] zu nennen, in welcher mehrere Programme implementiert worden sind. Darunter zählt auch der hierarchische Radiosity-Algorithmus (vgl. [Woo+95]). An dieser Realisierung orientiert sich auch die Arbeitsweise und der Aufbau der eigenen Implementierung. Als weitere Referenz dient das Programm aus der Arbeit [Kor01], in der die Radiosity-Applikation der SPLASH-2 Suite modifiziert wurde.

[1] `https://github.com/staceyson/splash2`, abg. am 21.04.2021

2 Theoretische Grundlagen

2.1 Irreguläre Algorithmen und Parallelisierung durch Taskpools

Um ein erstes Fundament für folgende Kapitel zu legen, wird nun die Irregularität von Algorithmen und deren Parallelisierung durch Taskpools betrachtet. Bei einem *irregulären Algorithmus* hängt das Berechnungsverhalten stark von der Eingabe ab, wodurch sich zum einen die ausgeführten Berechnungen und zum anderen auch die Zugriffe auf die verschiedenen Datenstrukturen für unterschiedliche Eingaben stark unterscheiden können. Dies ist auch der Grund, weshalb bei diesen Algorithmen eine Ausnutzung der Speicherhierarchie schwierig ist, weil oft wenig zeitliche als auch räumliche Lokalität bei den Speicherzugriffen vorhanden ist (vgl. [RR00, S. 499] und [Kor01, S. 11]).

Durch dieses stark unterschiedliche Verhalten ist eine Parallelisierung dieser Algorithmen im gemeinsamen Adressraum durch manuelle Lastbalancierung eher schwierig, da man nicht vorhersagen kann, was die optimale Ausnutzung der Prozessorkerne ist. Eine mögliche Lösung wäre ein *taskorientierter Ansatz*, welcher auf einer Ausführung mehrerer *Tasks* beruht. Ein Task ist dabei eine Folge von Operationen, die nacheinander auf einem Prozessor ausgeführt werden. Um jedoch eine effiziente Parallelisierung zu ermöglichen, ist es nötig, ein geeignetes Verfahren zu finden, welches freie Tasks zu einem zur Verfügung stehendem Prozessor zuordnet. Dies sind beispielsweise *Taskpools*, welche mehrere *Tasks* enthalten, die dann von bestimmten Prozessoren entnommen werden können. Auf diese Art und Weise kann eine automatische Lastbalancierung erreicht werden, da jeder Prozessor einen Task entnimmt, dessen Aufwand der Prozessor gut bearbeiten kann. Zur Realisierung eines Taskpools kann man im Wesentlichen eine Warteschlange verwenden, welche entweder zentral oder dezentral implementiert werden kann: (vgl. [RR00, S. 499f] und [Kor01, S 11])

- zentrale Warteschlange: Alle Tasks werden von einer einzigen, zentralen Warteschlange verwaltet, auf die von allen Prozessoren aus zugegriffen werden kann. Um Konflikte zu vermeiden, müssen somit Lock-Mechanismen eingesetzt werden.

- dezentrale Warteschlange: Jeder Prozessor besitzt seine eigene Warteschlange, aus der er seine Tasks entnehmen bzw. einreihen kann. Bei diesem Verfahren müssen am Anfang alle zur Verfügung stehenden Tasks z.B. statisch auf die Prozessorwarteschlangen verteilt werden.

Zentrale Warteschlangen haben den Vorteil, dass sie im Vergleich zu einer dezentralen Variante einfacher zu implementieren sind und es nie Prozessoren geben wird, die zum aktuellen Zeitpunkt nichts ausführen, obwohl weiterhin Tasks vorhanden sind. Der große Nachteil ist allerdings die Verwendung von Lock-Mechanismen zur Vermeidung von Konflikten bei einer größeren Anzahl an Prozessoren. Dieser Nachteil ist bei der dezentralen Implementierung nicht vorhanden, jedoch existiert hier das Problem des Lastungleichgewichts, wenn in verschiedenen Warteschlangen Tasks von stark unterschiedlichem Aufwand vorhanden sind. So kann eine Situation auftreten, in der mehrere unterschiedlich schwere Aufgaben existieren, wodurch die Prozessoren verschieden schnell rechnen (vgl. [RR00, S. 500f]).

Um dies auszugleichen verwendet man eine Technik, die *Task Stealing* genannt wird. Hierbei ist es erlaubt, dass andere Prozessoren auf die eigene dezentrale Warteschlange zugreifen dürfen, um einen Task zu entnehmen. Diese Situation tritt genau dann auf, wenn eine Warteschlange eines Prozessors keinen Task mehr enthält und der Prozessor selbst nicht mehr ausgelastet ist. In diesem Fall kann der Prozessor dann einen Task aus einer anderen Warteschlange entnehmen. Idealerweise wählt man die Warteschlange mit den meisten Tasks. Der große Nachteil in der Verwendung dieser Technik ist die Notwendigkeit der Einführung von Lock-Mechanismen, um die Datenintegrität innerhalb der Warteschlangen zu wahren. Da der gleichzeitige Zugriff auf eine Warteschlange jedoch nur in der Situation vorkommt, wenn ein Prozessor eine leere Warteschlange besitzt, sind die Leistungseinbußen im Vergleich zu einer zentralen Implementierung eher gering (vgl. [RR00, S. 500f]).

Dieses Prinzip des dezentralen Taskpools wurde in der anzufertigenden Implementierung im Rahmen der Multithreading-Runtime *Weave* ebenfalls verwendet, um eine Lastbalancierung beim Radiosity-Verfahren zu erreichen, dessen Grundlagen im folgenden Kapitel nun näher erläutert werden sollen.

2.2 Beleuchtungsmodelle und das Radiosity-Verfahren

Nachdem das Prinzip des Taskpools und die Definition von irregulären Algorithmen dargelegt worden ist, gilt es nun in einem nächsten Kapitel das Verständnis für Beleuchtungsberechnungen in der Computergraphik zu schaffen.

2.2.1 Globale und lokale Beleuchtungsmodelle

Das Fundament hierzu bilden die sogenannten *globalen* und *lokalen Beleuchtungsmodelle*, welche einen wesentlichen Einfluss auf die Farbdarstellung eines Pixels und somit auch auf Glanz- und Spiegelungseffekte haben, die für eine photorealistische Darstellung notwendig sind (vgl. [BGZ02, S. 140]).

Bei den lokalen Beleuchtungsmodellen werden lediglich die Lichtverhältnisse in einem Punkt der Szene betrachtet. Dies bedeutet, dass weder Wechselwirkungen von Objektoberflächen noch Mehrfachreflexionen berücksichtigt werden. Das Licht in einem Punkt setzt sich dabei aus folgenden Teilen zusammen: *ambientes Licht (Hintergrundbeleuchtung)*, *diffuses reflektierendes Licht* und *spiegelndes Licht*. Diffus bedeutet dabei, dass alle einfallenden Lichtstrahlen gleichmäßig in alle Richtungen reflektiert werden. Da dies in der Realität jedoch nicht der Fall ist und es immer eine Richtung gibt, in die das Licht hauptsächlich reflektiert wird, kann dies beim spiegelnden Licht berücksichtigt werden (vgl. [BGZ02, S. 140ff]).

Um jedoch photorealistische Bilder darstellen zu können, ist ein globales Beleuchtungsmodell notwendig, bei denen jetzt alle Lichtwechselwirkungen zwischen allen Objekten Anwendung finden, wodurch nun auch Objekte, die selbst eigentlich keine Lichtquelle sind, nun auch Licht emittieren, welches vorher aufgenommen wurde. So werden ebenfalls Mehrfachreflexionen berücksichtigt. Für die Realisierung einer globalen Beleuchtung ist es nötig, den Strahlungsaustausch in einer Szene samt allen Wechselwirkungen mathematisch zu beschreiben. Dies findet sich in der sogenannten *Rendering-Gleichung* wieder, welche im Folgenden *grob* beschrieben wird (vgl. dazu [BGZ02, S. 166ff]):

$$L_0(x, \omega_0) = L_e(x, \omega_0) + L_r(x, \omega_0) \tag{2.1}$$

Dabei setzt sich das insgesamt von einer Fläche ausgesendete Licht $L_0(x, \omega_0)$ am Punkt x in Richtung ω_0 zusammen aus dem selbst von dieser Fläche emittiertem Licht $L_e(x, \omega_0)$ und dem an dieser Fläche reflektiertem Licht $L_r(x, \omega_0)$. Diese Gleichung scheint zunächst recht einfach zu lösen, da dies lediglich eine grobe Beschreibung darstellt. Das Problem an der Rendering-Gleichung ist, dass der Term $L_r(x, \omega_0)$ ein

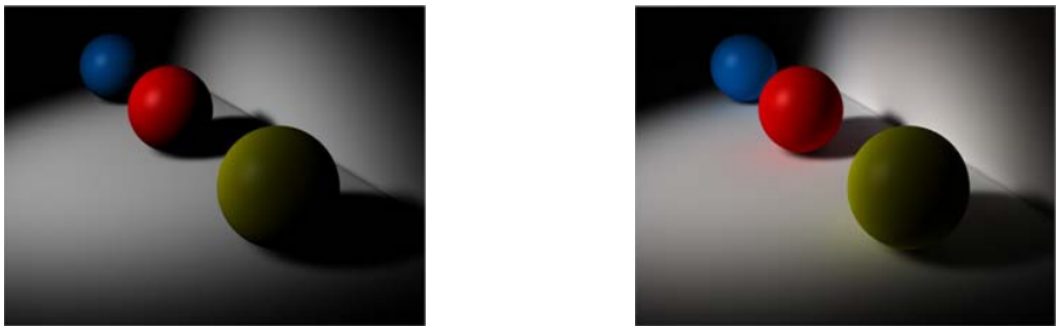
Integral beinhaltet, das eher schwierig zu lösen ist und die gesamte Berechnung daher approximiert werden muss (vgl. [BGZ02, S. 166ff]).

2.2.2 Grundlagen des klassischen Radiosity-Verfahrens

Nachdem nun die Rendering-Gleichung dargelegt worden ist, werden in einem nächsten Kapitel theoretische Grundlagen für den Hauptdarsteller dieser Arbeit vorgestellt, dem *Radiosity-Verfahren*.

2.2.2.1 Definition des Radiosity-Verfahrens

Dieser Algorithmus ist ein betrachterunabhängiges, globales Beleuchtungsverfahren, um diffuses Licht in einer dreidimensionalen Szene in geschlossenen Räumen zu simulieren. Betrachterunabhängig bedeutet dabei, dass die Szene jederzeit von einem neuen Blickpunkt dargestellt werden kann, da die Berechnung des Strahlungsaustauschs nicht auf Grundlage des Blickpunktes erfolgte. Als Eingabe wird die Geometrie der Szene erwartet, inklusive Farbe und Intensität von Lichtquellen. Auf dieser Grundlage berechnet der Algorithmus dann sogenannte *Radiosity-Werte*, welche die Lichtenergie pro Zeiteinheit und pro Flächeneinheit, auch Strahlungsenergie genannt, bezeichnen (vgl. [RR00, S. 520]). Abbildung 2.1 zeigt eine Beispielszene mit und ohne Einsatz des Radiosity-Verfahrens. Es wird deutlich, dass die Szene, welche mit Radiosity berechnet wurde, realistischer wirkt, da durch den Einsatz eines globalen Beleuchtungsmodells ebenfalls Mehrfachreflexionen berücksichtigt werden. Bei der Szene ohne Radiosity wird demnach ein lokales Modell verwendet, wodurch nur der direkte Lichtaustausch Anwendung findet (vgl. [Nü05, S. 3]). Um ein derartiges Resultat realisieren zu können, bedarf es eines mathematischen Modells, welches im Folgenden vorgestellt werden soll.



(a) ohne Radiosity **(b)** mit Radiosity

Abb. 2.1: Beispielszene mit und ohne Einsatz des Radiosity-Verfahrens (vgl. [Nü05, S. 3])

2.2.2.2 Die Radiosity-Gleichung

Um nun den Hauptdarsteller dieser Arbeit, das *Radiosity-Verfahren*, näher zu beschreiben, ist eine Erklärung des theoretischen Fundaments notwendig. Der Algorithmus basiert auf der *Radiosity-Gleichung*, welche die Rendering-Gleichung versucht zu approximieren. Die Herleitung der Radiosity-Gleichung erfolgt auf Grundlage der Rendering-Gleichung unter der Annahme der finiten Elemente. Dabei werden alle Objekte in weitere kleinere Teilobjekte (= *Elemente*) unterteilt, auf denen die Radiosity in jedem Punkt gleich ist. Dadurch ist es notwendig, diese Elemente sehr klein zu wählen, damit der korrespondierende Fehler ebenfalls gering ist und im Prinzip vernachlässigt werden kann (vgl. [Nü05, S. 12]). Die abgeleitete Radiosity-Gleichung wird durch folgende Formel beschrieben (vgl. [RR00, S. 521]):

$$B_i A_i = E_i A_i + \rho_i \sum_j F_{ji} B_j A_j \tag{2.2}$$

Dabei setzt sich die Strahlungsleistung $B_i A_i$ eines Flächenelementes i zusammen aus der selbst emittierten Strahlung des entsprechenden Flächeninhaltes $E_i A_i$ und der mit dem diffusen Reflexionskoeffizienten ρ_i gewichteten eintreffenden Strahlung $\sum_j F_{ji} B_j A_j$. Hierbei berücksichtigt die Summe die gesamte eintreffende Strahlung des Elements j, wobei die Strahlung eines Elements mit dem *Formfaktor* F_{ji} gewichtet wird, welcher den Anteil der Strahlung von Element j angibt, der auch i erreicht. Abbildung 2.2 zeigt eine Illustration dieser Formel. Aufgrund der Symmetriebedingung $A_j F_{ji} = A_i F_{ij}$ ergibt sich nun die finale Radiosity-Gleichung

$$B_i = E_i + \rho_i \sum_j F_{ij} B_j \tag{2.3}$$

und damit das lineare Gleichungssystem der Größe n, wobei n die Anzahl der Elemente darstellt (vgl. [RR00, S. 521]):

$$\begin{pmatrix} 1 & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\rho_{n-1} F_{n-1,n} \\ -\rho_n F_{n1} & \cdots & -\rho_n F_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} \tag{2.4}$$

Dieses Gleichungssystem kann nun effizient mit Hilfe von iterativen Algorithmen gelöst werden, beispielsweise mit dem Gauß-Seidel-Verfahren. Ein großes Problem stellt jedoch die Berechnung der Formfaktoren dar, welche in der obigen Matrix enthalten sind. Diese besitzt die Dimension $n \times n$ und beinhaltet somit n^2 Formfaktoren bei n Elementen. Da diese quadratische Laufzeit nicht wirklich zufriedenstellend ist, wird in einem späteren Kapitel das hierarchische Radiosity-Verfahren vorgestellt (vgl. [Kor01, S. 17]).

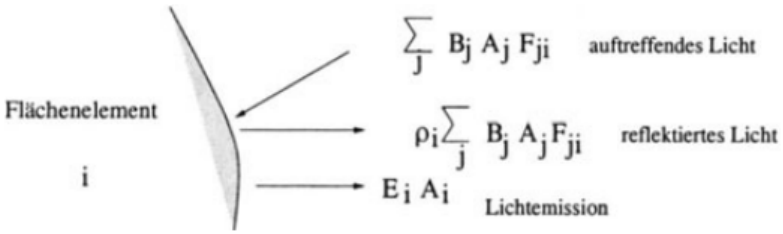


Abb. 2.2: Illustration der Radiosity-Gleichung (vgl. [RR00, S. 521])

2.2.2.3 Berechnung der Formfaktoren

Ein Formfaktor F_{ij} gibt den Anteil des Lichts an, der von i gesendet und auch von j empfangen wird. Doch wie genau wird dieser Faktor berechnet? Da er nur von der Geometrie der Szene und nicht etwa von der Wellenlänge des Lichts abhängt, kann man diesen anfangs einmalig berechnen und dann dauerhaft abspeichern. Man kann zeigen, dass für F_{ij} folgende Beziehung gilt (vgl. [BGZ02, S. 184]):

$$F_{ij} = \frac{1}{A_i} \cdot \frac{1}{\pi} \cdot \int_{f_i} \int_{f_j} G(x_i, x_j) \mathrm{d}A_j \mathrm{d}A_i \tag{2.5}$$

Dabei beschreibt A_i den Flächeninhalt des Elementes i und f_i bzw. f_j die Fläche i und j. $G(x, y)$ ist dabei folgendermaßen definiert (vgl. [BGZ02, S. 168ff]):

$$G(x_i, x_j) := V(x_i, x_j) \cdot \frac{\cos\theta_i(x_i, x_j) \cos\theta_j(x_i, x_j)}{|x - y|^2} \tag{2.6}$$

mit

$$V(x, y) := \begin{cases} 1 & \text{x sieht y,} \\ 0 & \text{sonst.} \end{cases} \tag{2.7}$$

Hierbei bezeichnen θ_i und θ_j die Winkel zwischen der Verbindungsstrecke von x_i nach x_j und der Flächennormale n_i der Fläche f_i bzw. der Flächennormale n_j der Fläche f_j. Dies ist in Abbildung 2.3 nochmals dargestellt. Da das Doppelintegral über alle Flächenelemente zu lösen ist, bedarf es hier einer Approximation, damit eine effiziente Berechnung möglich ist (vgl. [BGZ02, S. 184ff]). Dabei wird der Formfaktor F_{ij} von einer Fläche i zu einer Fläche j im Wesentlichen durch die Schätzung von mehreren Formfaktoren berechnet. Die Fläche i wird zunächst in vier disjunkte Teilflächen A_i' unterteilt, auf denen jeweils ein Punkt (beispielsweise der Mittelpunkt) ausgewählt wird. Als Nächstes werden vier Scheiben A_{D_l} mit $l = 1, ..., 4$ berechnet, die jeweils die gesamte Fläche A_j enthalten. Nun wird der Formfaktor

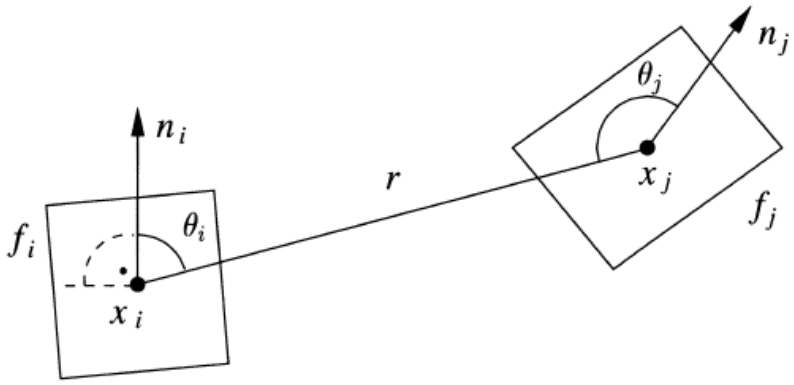


Abb. 2.3: Visualisierung der Winkel θ_i und θ_j (vgl. [BGZ02, S. 185])

zwischen den Flächen A_i und A_{D_l} durch folgende Formel berechnet (vgl. [PRR98, S. 8]):

$$F_{ij} = \frac{\cos\theta_i \cos\theta_j \pi R^2}{\pi r^2 + \pi R^2} \tag{2.8}$$

Dabei ist R der gesamte Flächeninhalt der Scheibe A_{D_l} und r der Abstand der beiden ausgewählten Punkte. Ausgehend von den vier Punkten auf den Teilflächen A'_i werden jeweils vier Formfaktoren $F_{A'_i A_{D_l}}$ berechnet, deren Summe dann die Approximation des eigentlichen Formfaktors $F_{A'_i A_j}$ ist. Da die Fläche A_i lediglich aus den Teilflächen A'_i besteht, wird dann der Durchschnitt aus den vier Formfaktoren $F_{A'_i A_j}$ gebildet. Dieser recht hohe Aufwand verdeutlicht nochmals die Notwendigkeit, die Anzahl der Formfaktoren zugunsten einer effizienten Laufzeit zu reduzieren (vgl. [PRR98, S. 8]).

2.2.2.4 Grundstruktur des Algorithmus

Die Grundstuktur des klassischen Radiosity-Verfahrens wird nun näher betrachtet und anhand der Abbildung 2.4 visualisiert. In einem ersten Schritt werden alle Objekte der Szene eingelesen und verarbeitet, darunter zählt zum Beispiel das Setzen der initialen Radiosity. Damit die Annahme, dass die Radiosity in jedem Punkt auf einer Objektfläche konstant ist, angewendet werden kann, müssen die Ausgangspolygone in kleinere Elemente zerteilt werden, wodurch ein genauerer Lichtaustausch berechnet werden kann. Als Nächstes müssen zwischen allen n Elementen, n^2 Formfaktoren berechnet werden, um dann im nächsten Schritt die Radiosity-Gleichung $M \cdot B = E$ aufstellen zu können. Dabei ist die Matrix M der Formfaktoren und der Vektor E bekannt, wodurch der Vektor B (Radiosity-Werte) berechnet werden muss. Dieses Gleichungssystem muss nun in einem nächsten Schritt gelöst werden, beispielsweise durch iterative Verfahren. Als Letztes müssen die berechneten Radiosity-Werte den

Polygonen und vor allem deren Eckpunkten zugewiesen werden, wodurch noch eine Durchschnittsberechnung und Normalisierung folgt (vgl. [Nü05, S. 15]).

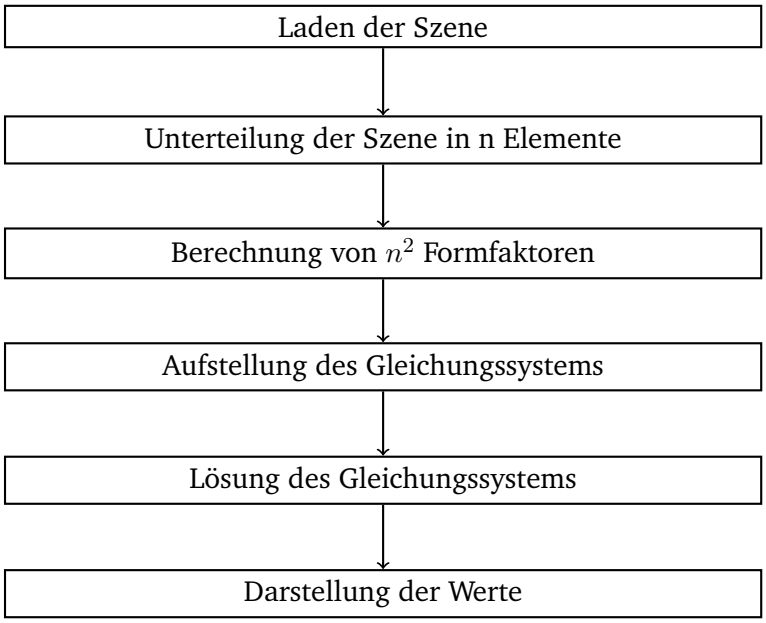


Abb. 2.4: Illustration des klassischen Radiosity-Algorithmus (vgl. [Nü05, S. 15])

2.2.3 Das hierarchische Radiosity-Verfahren

Das klassische Radiosity-Verfahren stößt schnell an seine Grenzen bezüglich Laufzeit und Speicher. Durch die konstante Unterteilung der Polygone in gleich große Elemente, die umgebungsunabhängig erfolgt, resultiert ein hoher Speicher- und Laufzeitbedarf bei der Formfaktorberechnung ($O(n^2)$). Hier werden nun zugunsten einer besseren Laufzeit und Speicherverwendung hierarchische Algorithmen, die adapativ arbeiten, eingesetzt. Hierarchische Algorithmen zerlegen dabei den Raum rekursiv in mehrere Teilgebiete abhängig von der Umgebung bis ein gewisser Fehler unterschritten ist (vgl. [RR00, S. 518]). Dies kann man beim Radiosity-Verfahren nun an der Stelle der Unterteilung in Patches einsetzen. Man startet mit wenigen Flächen und nur dort, wo sich die berechneten Radiosity-Werte auf benachbarten Flächenstücken stark unterscheiden, wird die korrespondierende Fläche unterteilt. Dies hat den Vorteil, dass auch nur dann eine erneute Formfaktorberechnung notwendig ist. Beispielsweise kann dies eingesetzt werden, wenn zwei Polygone weit entfernt sind. Hierbei kann weniger stark unterteilt werden, da der Strahlungsaustausch eher grob erfolgt. Falls die beiden Polygone jedoch nah beieinander liegen, erfolgt der Strahlungsaustausch in einer höheren Auflösung und es benötigt auch mehrere Unterteilungen, damit sich die Radiosity-Werte auf benachbarten Flächen nicht so stark unterscheiden (vgl. [BGZ02, S. 190]). Dieser Ansatz basiert auf dem *N-Körper-Problem*, dessen Anwendung sich eher in der Physik befindet. Dabei betrachtet man ein System mit n Partikeln, weshalb es insgesamt n^2 paarweise Interaktionen

gibt und eine quadratische Laufzeit resultiert. Das N-Körper-Problem besagt dabei, dass falls die Auswirkung einer Interaktion mit zunehmender Distanz abnimmt, eine Ansammlung von Partikeln als ein einziges Partikel modelliert werden kann. Falls man also mehrere Interaktionen zwischen zwei größeren Gruppen berechnen muss, die zueinander weit entfernt sind, können diese beiden Gruppen als jeweils ein Partikel modelliert werden. Dieser Ansatz spiegelt sich im hierarchischen Radiosity-Algorithmus wider, jedoch mit dem wesentlichen Unterschied, dass man nicht mit n Elementen startet, welche anschließend zu einem einzigen Element zusammengefasst werden. Vielmehr liegen beim Start einige wenige Elemente vor, die immer weiter schrittweise verfeinert werden (vgl. [Sin+95, S. 2ff]). Um die eigene Implementierung später zu verstehen, ist eine Erklärung der Terminologie des hierarchischen Radiosity-Verfahrens notwendig. Alle Eingabepolygone werden *Patches* genannt, die in einem BSP-Baum (= *Binary Space Partition*) eingefügt werden. Dies ermöglicht einen effizienten Sichtbarkeitstest, der in einem späteren Kapitel erklärt wird. Jeder Knoten im BSP-Baum beinhaltet nochmals einen *Quadtree*, der das Konzept der hierarchischen Algorithmen erfüllt. Jeder Patch kann nochmals in vier weitere Teilpolygone unterteilt werden, die als Kinder in den Quadtree eingefügt werden. Jeder Knoten im Quadtree wird *Element* genannt. Dieses Prinzip ist in Abbildung 2.5 illustriert (vgl. [Sin+95, S. 11]).

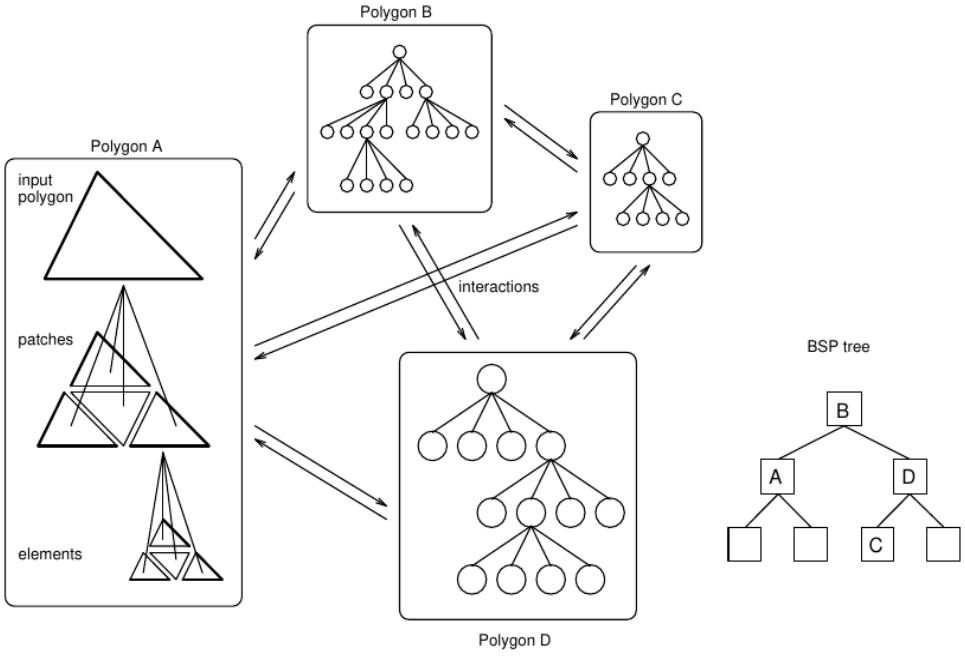


Abb. 2.5: Illustration des BSP-Baums und der Quadtrees (vgl. [Sin+95, S. 11])

3 Beschreibung der Programmiersprache Nim und der Multithreading-Runtime Weave

Nachdem einige theoretische Grundlagen zugrunde gelegt worden sind, folgt nun eine Beschreibung der Programmiersprache Nim, in welcher das zu implementierende hierarchische Radiosity-Verfahren angefertigt wurde. Dabei basiert Kapitel 3.1 auf [Pic17, S. 3ff] und Kapitel 3.2 auf dem korrespondierenden Repository[1].

3.1 Überblick über Nim

Nim ist eine moderne Programmiersprache, die im Vergleich zu herkömmlichen Programmiersprachen einige Besonderheiten aufweist. Für einen kurzen Einblick werden nun einige wesentliche Grundlagen dargelegt.

Nim verwendet eine Syntax, die ähnlich ist zu Python. So können beispielsweise Blöcke durch Einrückung abgegrenzt oder die logische Veroderung durch das Schlüsselwort „or“ ausgedrückt werden, wodurch die Syntax an Kompaktheit gewinnt.

Um Manipulationen vom bestehendem Code oder Generierung von neuem Code während der Kompilierung zu erlauben, existiert das Konzept der Metaprogrammierung, deren wesentlicher Bestandteil *Generizität*, *Templates* und *Makros* sind.

Eine weitere Besonderheit beinhaltet das Typsystem, welches generell entweder statisch oder dynamisch konzipiert werden kann. Nim verwendet ein stark statisches Typsystem, bei dem jede Variable bei der Deklaration einen festen Typ haben muss. Dies schränkt normalerweise die Programmiergeschwindigkeit ein, jedoch kann dies durch eine automatische Folgerung des Typs kompensiert werden. So kann der Nim Compiler den Typ aus dem Variablenwert schließen. Außerdem folgt durch das starke Typsystem ein robustes Programmiermodell.

Ein weiteres Merkmal ist die Speicherverwaltung. Hier ist es möglich, zwischen mehreren Speicherverwaltungen zu wählen, welche automatisch Objekte allokieren und freigeben können. Auch ist es möglich, diese komplett auszuschalten. Dies spielt eine große Rolle bei der Parallelisierung, da hier mehrere Threads das Programm abarbeiten. So gibt es einen Garbage-Collector *RefC*, der jedem Thread seinen eigenen Adressraum zuweist. Bei anderen Modellen, beispielsweise *Boehm*, ist aber auch ein

[1] `https://github.com/mratsim/weave`, abg. am 21.04.2021

gemeinsamer Adressraum für alle Threads vorhanden.

Ein weiteres Merkmal im Bereich der Speichersicherheit ist die Eigenschaft, dass ebenso systemnahe Programmierung möglich ist, obwohl der Speicher größtenteils vom Garbage-Collector verwaltet wird. So kann man Pointer definieren, die auf einen festgelegten Speicherbereich zeigen, wobei aber Pointerarithmetik nicht möglich ist.

Die größte Besonderheit stellt wohl der Kompilierungsprozess dar. Hierbei wird der Nim Code zunächst in eine andere Programmiersprache, darunter C, Objective C oder auch Javascript, transformiert, welcher dann als Eingabe für den jeweiligen Compiler dient. Dieser Prozess ist in Abbildung 3.1 für die Transformation in C-Code dargestellt. Durch diese Eigenschaft ist es ebenso möglich, C-Code in die eigene Nim Anwendung zu integrieren.

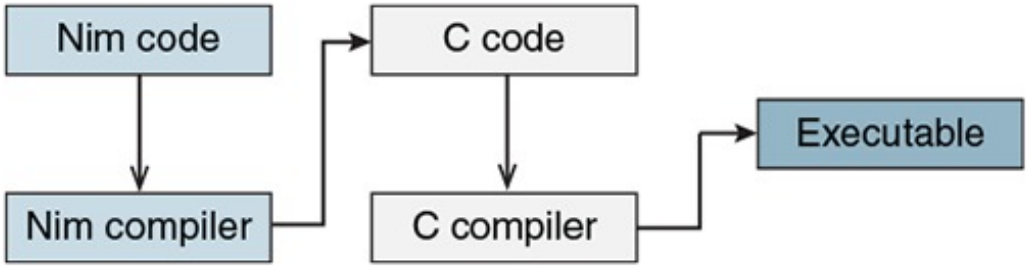


Abb. 3.1: Kompilizierungsprozess mit Transformation in C-Code (vgl. [Pic17, S. 11])

Zusammengefasst lässt sich sagen, dass Nim gegenüber anderen Programmiersprachen einige Vorteile bietet. So ist es durch die kompakte Syntax sehr einfach, in wenigen Zeilen kompliziertere Programme zu schreiben, die ebenso lesbar gestaltet sind. Durch das starke Typsystem entsteht eine sehr robuste Programmiersprache, da der Compiler schon im Vorhinein bestimmte Fehler ausschließen kann, die meistens durch ein dynamisches Typsystem bedingt sind. Der größte Vorteil bietet jedoch wohl der Kompilierprozess. So kann man je nach Anwendung, verschiedene Sprachen zur Transformation nutzen. Um eine effiziente Applikation zu gewinnen, kann man beispielsweise C wählen, da sich diese Sprache selbst durch Effizienz auszeichnet. Abschließend ist jedoch auch zu erwähnen, dass Nim durch die vielen Besonderheiten eine Sonderstellung einnimmt und es einer gewissen Gewöhnung und auch Einlesezeit bedarf, Programme in dieser Sprache zu erstellen.

3.2 Überblick über Weave

Im Rahmen dieser Arbeit wurde das hierarchische Radiosity-Verfahren parallel mithilfe der Multithreading-Runtime Weave erstellt, welche nun einen kurzen Überblick erhält.

Weave ist eine Laufzeitumgebung zur Erstellung und Verwaltung von Threads (= Multithreading-Runtime), welche für Linux, MacOS und Windows für die Architekturen x86, x86-64 und ARM64 getestet wurde, also eine hohe Portabilität aufweist. Dabei realisiert Weave im Wesentlichen das Konzept eines Schedulers mit Taskpool und Task-Stealing, welches in Kapitel 2.1 vorgestellt wurde. Als wesentliche Besonderheit verwendet Weave für jegliche Kommunikation *Message-Passing*, bei dem jeder Thread als Kommunikationsbasis die von Nim bereitgestellten Channels nutzt. Channels sind im Wesentlichen FIFO-Warteschlangen, welche als Kommunikationskanal zwischen bestimmten Threads fungieren (vgl. dazu [Pic17, S. 176]). So findet dies ebenso beim Konzept des Task-Stealings Anwendung. Anstatt einen Task aktiv von der Warteschlange eines anderen Threads zu stehlen, wird ein *Steal Request* gesendet, der zunächst von dem korrespondierenden Thread beantwortet werden muss. Durch die Verwendung von Message-Passing kann Weave auf jeder Architektur ausgeführt werden, die neben Atomics auch Warteschlangen, Channels oder Locks unterstützt. Dies impliziert vor allem auch die Möglichkeit der Ausführung auf Clustersystemen.

Doch inwiefern kann man ein gegebenenes Programm mit Weave parallelisieren? Dabei wurden drei Arten der Parallelisierung in der Laufzeitumgebung implementiert:

- Taskparallelität: Hiermit ist das explizite Erstellen von Tasks anhand eines Funktionsaufrufs *spawn(func)* möglich, bei dem ein mit der Funktion *func* assoziierter Task erstellt und in den Taskpool eingereiht wird.

- Datenparallelität: Hiermit ist die Parallelisierung auf Datensätzen möglich, die beispielsweise in einer Schleife abgearbeitet werden. Die Schleife wird dann automatisch von Weave anhand der aktuellen Last unterteilt und korrespondierende Tasks erstellt.

- Datenflussparallelität: Hiermit können feingranulare Datenabhängigkeiten zwischen Tasks definiert werden, wodurch ein Task nur ausgeführt wird, wenn alle Abhängigkeiten erfüllt worden sind.

In der anzufertigenden Implementierungen wurde dabei sowohl Taskparallelität als auch Datenparallelität verwendet.

4 Vorstellung der Implementierung

Nachdem nun alle theoretischen Grundlagen über das klassische und hierarchische Radiosity-Verfahren sowie über irreguläre Algorithmen zugrunde gelegt worden sind, gilt es in einem nächsten Kapitel, die Arbeitsweise und Verwendungsmöglichkeit der eigenen anzufertigenden Implementierung des taskbasierten parallelen hierarchischen Radiosity-Verfahrens zu beleuchten. Dabei stützt sich der Aufbau, die Arbeitsweise der Implementierung und dieses Kapitel auf der Variante des hierarchischen Radiosity-Algorithmus in der SPLASH-2 Suite[1] (vgl. [Woo+95]).

4.1 Architektur

Die Implementierung wurde in der Programmiersprache *Nim* angefertigt, deren Besonderheiten bereits in Kapitel 3.1 erläutert worden sind. Hierbei wurde Version 1.4.2 verwendet. Außerdem wird bei der Installation von Nim ebenfalls der Paketmanager *Nimble* mitgeliefert, welcher im Folgenden benötigt wird.

4.1.1 Abhängigkeiten

Um die Anwendung starten zu können, ist die Installation von zwei Abhängigkeiten notwendig. So wird für die taskbasierte Parallelisierung des Verfahrens die Multithreading-Runtime *Weave*[2] verwendet, welche in Kapitel 3.2 näher erklärt worden ist. Hierbei wird die Experimental Version benutzt, um auch den Zugriff auf neuere, experimentelle Features zu ermöglichen. Neben dieser Abhängigkeit wird ebenfalls das Paket *cligen*[3] eingebunden, um die Implementierung und Auswertung von Aufrufparametern einfach zu gestalten. Beide Pakete können über Nims Paketmanager *Nimble* installiert werden:

```
nimble install weave@#master und
nimble install cligen
```

Des Weiteren werden gewisse Module aus der Standardbibliothek importiert.

[1] `https://github.com/staceyson/splash2`, abg. am 21.04.2021
[2] `https://github.com/mratsim/weave`, abg. am 30.03.2021
[3] `https://github.com/c-blake/cligen`, abg. am 30.03.2021

4.1.2 Struktur

Im verwendeten Repository und auch auf der beiliegenden CD befinden sich mehrere Implementierungen des parallelen hierarchischen Radiosity-Verfahrens. Die erste Variante im Ordner *radiosity* realisiert das Verfahren mit dynamischer Allokation aller Objekte, wodurch das Programm auf allen Eingabedaten ausgeführt werden kann, ohne dass es manuelle Modifikationen benötigt. Dabei werden alle Daten auf dem *shared heap* allokiert. Die zweite Version befindet sich im Ordner *radiosity_static* und realisiert das Verfahren ohne dynamische Allokation aller Objekte, indem diese vorab in verschiedenen Caches vorallokiert und dann während der Laufzeit entnommen werden. Dadurch folgt eine bessere Grundperformance, da nicht während der eigentlichen Berechnungen Speicher reserviert werden muss. Die letzte Variante ist eine Implementierung aus der SPLASH-2 Bibliothek[4] (vgl. [Woo+95]), welche im Rahmen der Arbeit [Kor01] und innerhalb der eigenen Arbeit modifiziert worden ist. Diese Version diente vor allem als Grundlage für die eigene Implementierung, jedoch auch für Verifikationen und Laufzeitmessungen.

Weiterhin befindet sich in jedem Ordner eine Datei *bench.nim*, mit welcher man Laufzeitmessungen starten und korrespondierende Diagramme erstellen lassen kann.

4.1.3 Nutzerschnittstelle

Um die Implementierung zu kompilieren, stellt Nim selbst die Sprache NimScript zur Verfügung, die man als Build Tool verwenden kann. So kann man mit folgenden Befehlen relativ komfortabel, das Projekt bauen und korrespondierende Messungen starten, wodurch dann im Ordner *benchmark* entsprechende Grafiken erzeugt werden:

`nim build` und
`nim bench`

Die Implementierung selbst stellt eine Kommandozeilenschnittstelle zur Verfügung, um das Programm zu starten und weitere Argumente einzulesen. Alle implementierten Aufrufparameter sind in Tabelle 4.1 zu finden.

Standardmäßig wird das Programm mit den Testdaten bei eingeschaltetem Patch-Cache, überlappender erster und zweiter Phase, sequentiell mit einem BF-Epsilon von 0.015, einem Area-Epsilon von 2000 und einem Energy-Epsilon von 0.005 ohne Ausgabe von Statistiken gestartet.

[4] `https://github.com/staceyson/splash2`, abg. am 22.05.2021

Parameter	Beschreibung
-h / –help	Ausgabe eines Hilfstextes und Beendigung des Programms
-m / –model	Auswahl der Eingabedaten zwischen Testdaten, Roomdaten, Largeroomdaten und XRAF-Dateien
-f / –file	Liest eine XRAF-Datei ein
-v / –verbose	Auswahl des Debugmodus
-b / –bfe	Setzen des BF-Epsilons (für Unterteilung eines Elements)
-a / –ae	Setzen des Area-Epsilons (für Unterteilung eines Elements)
-e / -ee	Setzen des Energy-Epsilons (Konvergenzkriterium)
-t / –threads	Setzen der Anzahl der Threads
-p / –separatePhases	Phase 1 und 2 werden separat ausgeführt
-s / –statistics	Ausgabe der Statistik
-n / –noCache	Ausschalten des Patch-Caches im Sichtbarkeitstest

Tab. 4.1: Mögliche Aufrufparameter beim Start des Programms

4.2 Beschreibung der konkreten Implementierung des hierarchischen Radiosity-Verfahrens

Nach der Beschreibung und Verwendung der eigenen Implementierung, folgt nun eine genauere Erläuterung der Realisierung des parallelen hierarchischen Radiosity-Verfahrens inklusive der Datenstrukturen. Wie schon erwähnt, diente dabei die Variante in der SPLASH-2 Suite[5] (vgl. [Woo+95]) als Grundlage.

4.2.1 Verwendete Datenstrukturen

Um das Verfahren zu implementieren, bedarf es geeigneter Datenstrukturen zur Repräsentation von dreidimensionalen Objekten. Um ein Dreieck darstellen zu können, existieren die beiden Objekte *Vertex* (= Eckpunkt) und *Edge* (= Kante). Wie in Kapitel 2.2.3 schon beschrieben, arbeitet das Verfahren auf einem BSP-Baum, dessen Knoten wiederum Quadtrees enthalten, weshalb es in der Implementierung korrespondierende Datenstrukturen hierfür gibt. Für die inneren Knoten des BSP-Baums existieren *Patch*-Objekte, die beispielsweise den Radiosity-Wert, Kindknoten und andere Eigenschaften speichern können. Um den Quadtree zu repräsentieren, existiert weiterhin das Objekt *Element*, dessen Eckpunkte nicht nur durch einen einzelnen Vertex gespeichert werden, sondern vielmehr durch ein extra Objekt *ElemVertex*, welches geteilt für mehrere Dreiecke verwendet wird und einen Farbwert haben kann. Dies ist nötig, damit die Schattierung auf der Fläche des Dreiecks später zwischen allen Eckpunkten interpoliert werden kann, was jedoch in der eigenen Implementierung nicht umgesetzt wurde und existierenden Graphikbibliotheken überlassen wird.

[5] `https://github.com/staceyson/splash2`, abg. am 22.05.2021

Des Weiteren existiert ein Objekt für die Repräsentation einer Interaktion, welches ebenfalls den berechneten Formfaktor beinhaltet. Da jedes Element eine Liste von Interaktionen besitzt, musste hierfür ebenfalls eine geeignete Datenstruktur gefunden werden. Zwar könnte man hierfür Nims Datenstruktur *sequence* verwenden, welche eine Liste realisiert. Da das Programm jedoch mit jedem Garbage-Collector kompatibel sein sollte und der Standard Garbage-Collector *RefC* letztendlich einen verteilten Adressraum realisiert, konnte dies nicht verwendet werden, da es sein kann, dass ein Thread in der Sequence eines anderen Threads Interaktionen einfügen könnte. Dies würde innerhalb von Nim aufgrund des Aufbaus einer Sequence zu einem Laufzeitfehler führen, weshalb eine eigene Liste implementiert wurde, deren Knoten letztendlich Interaktionen sind und lediglich durch Pointer eingehängt werden. Dadurch ist beim Einfügen keine Speicherallokation eines Listenknotens mehr notwendig, weshalb auch kein Fehler ausgelöst wird.

4.2.2 Phase 1: Einlesen der Eingabedaten und Einsortierung in den BSP-Baum

Der Algorithmus startet mit dem Einlesen der Eingabedaten, wobei diese eine Szene von Rechtecken und Dreiecken beinhaltet. Da das Radiosity-Verfahren lediglich auf Dreiecken arbeitet, müssen Rechtecke zunächst in der Mitte geteilt werden. Dabei wird nun jedes Polygon mit verschiedenen Eigenschaften versehen, darunter die Ebenengleichung, welche durch eine Art *Hessesche Normalform* repräsentiert wird (vgl. [BHW12, S. 71]):

$$\langle \vec{n_0}, \vec{x} \rangle = c \text{ mit } c = \langle \vec{n_0}, \vec{p} \rangle \tag{4.1}$$

Dabei ist $\vec{x} \in \mathbb{R}^3$, $\vec{p}$ ein Punkt in der Ebene des Polygons, $c \geq 0$ und $\vec{n_0} \in \mathbb{R}^3$ ein Einheitsvektor, der senkrecht auf der Ebene steht. Bei der Implementierung wurde die dritte Bedingung $c \geq 0$ außer Acht gelassen, da dies letztendlich keine Auswirkung auf die Abstandsberechnung im Rahmen der eigenen Implementierung hätte und man sich somit weitere Berechnungen sparen kann. Nun erhält man durch die Auswertung dieser Gleichung mit einem beliebigen Punkt $\vec{x} \in \mathbb{R}^3$ den gerichteten Abstand dieses Punktes von der Ebene (vgl. [BHW12, S. 72]). So gilt nun folgender Zusammenhang, welcher unter anderem nochmals in Abbildung 4.1 illustriert ist (vgl. [Sch10, S. 57]):

1. Falls $d = 0$: Punkt $\vec{x} \in \mathbb{R}^3$ liegt in der Ebene, also innerhalb des Dreiecks

2. Falls $d > 0$: Punkt $\vec{x} \in \mathbb{R}^3$ liegt außerhalb der Ebene und auf der Seite, in die die Normale zeigt

3. Falls $d < 0$: Punkt $\vec{x} \in \mathbb{R}^3$ liegt außerhalb der Ebene und auf der entgegengesetzten Seite, in die die Normale zeigt

Jetzt ist es möglich, für einen beliebigen Punkt zu testen, ob dieser entweder auf oder außerhalb der Ebene liegt. Dies kann nun für die Einsortierung in den BSP-Baum verwendet werden, der letztendlich eine Raumaufteilung durchführt. Dafür wird jedes Dreieck nun in einen Patch umgewandelt und in den Baum eingefügt. Für einen gegebenen Patch P_1 im BSP-Baum können nun folgende zwei Fälle beim Einfügen von Patch P_2 eintreten (vgl. [Woo+95]):

1. P_1 und P_2 schneiden sich nicht

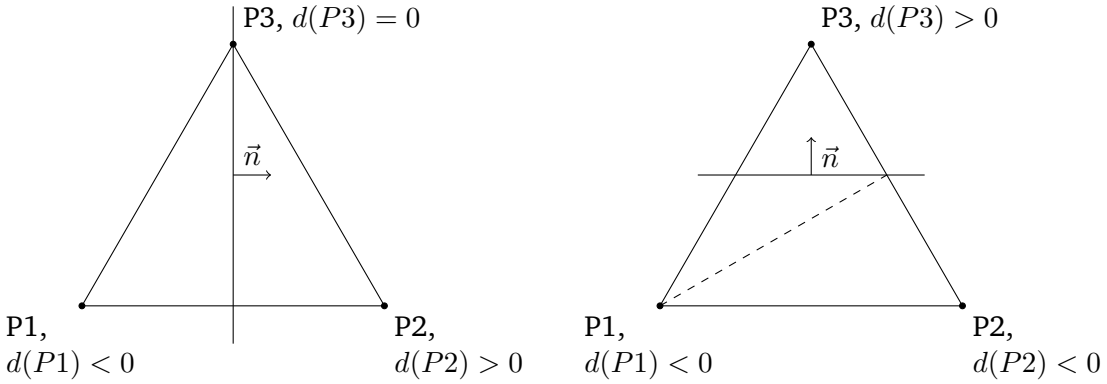
   a) Fall 1: P_2 liegt auf der Seite der Normalen von $P_1 \rightarrow$ Patch P_2 ist positiver Halbraum und wird damit positiver Kindknoten von P_1

   b) Fall 2: P_2 liegt auf der entgegengesetzten Seite der Normalen von $P_1 \rightarrow$ Patch P_2 ist negativer Halbraum und wird damit negativer Kindknoten von P_1

2. P_1 und P_2 schneiden sich: Je nach Schnittsituation muss P_2 in zwei oder drei Teildreiecke unterteilt und diese erneut in den BSP-Baum eingefügt werden

Doch wie genau kann man feststellen, dass sich P_1 und P_2 schneiden? Da sich P_1 schon im BSP-Baum befindet, wird die durch dieses Dreieck aufgespannte Ebene als Referenz betrachtet. Da P_2 aus drei Eckpunkten besteht, kann man von jedem Eckpunkt aus den gerichteten Abstand zur Ebene berechnen. Falls sich die Vorzeichen unterscheiden, liegt ein Schnitt vor. Da es jedoch auch wichtig ist, Kenntnis über die konkrete Situation zu erlangen, wären danach noch einige Vergleiche wichtig. Man müsste überprüfen, ob das Dreieck mittig durch P_1 geteilt wird. Dies stellt man fest, indem der Abstand eines Punktes gleich 0 ist und die der anderen beiden sich im Vorzeichen unterscheiden, also jeweils auf einer anderen Seite liegen. Die zweite Situation wäre ein ungerader Schnitt, bei dem P_2 in drei Teildreiecke unterteilt werden würde. Hierbei wäre kein Abstand gleich 0, vielmehr müssten unterschiedliche Vorzeichen vorliegen. Die beiden Schnittsituationen sind nochmals in Abbildung 4.1 illustriert. Um dies effizient zu berechnen und auf logische, unübersichtliche Vergleiche zu verzichten, wird jede Situation mit einem binären Code erkannt. Jedem Eckpunkt wird zunächst ein Code zugeordnet, welche folgendermaßen aussehen (vgl. [Woo+95]):

- 0 (binär: 00): Eckpunkt liegt auf der Schnittebene
- 1 (binär: 01): Eckpunkt liegt auf der Normalenseite der Schnittebene



(a) 2 Teildreiecke **(b)** 3 Teildreiecke

Abb. 4.1: Illustration der Schnittsituationen

- 2 (binär: 10): Eckpunkt liegt auf der entgegengesetzten Normalenseite der Schnittebene

Durch diese Zuordnung erhält jeder Eckpunkt seinen individuellen Code mit maximal zwei Ziffern. Um nun den Gesamtcode des Dreiecks zu ermitteln, wird jedem Eckpunkt eine Position zugewiesen, welche durch logisches Shifting erreicht wird. Eckpunkt 1 wird dabei nicht geshiftet, wodurch dieser Teilcode im Gesamtcode dann die ersten beiden Stellen einnimmt. Eckpunkt 2 wird um zwei nach links geshiftet, wodurch der ursprüngliche Teilcode dann an Stellen drei und vier zu finden ist. Eckpunkt 3 wird dann entsprechend um vier Stellen nach links geshiftet, weshalb dieser Teilcode dann an den Stellen 5 und 6 zu finden ist (vgl. [Woo+95]). Beispiel 4.2.1 verdeutlicht dies nochmal anhand der Abbildung 4.1a.

Beispiel 4.2.1. In der Schnittsituation in Abbildung 4.1a würden alle drei individuellen Codes zu den Eckpunkten folgendermaßen aussehen:

- Eckpunkt P1: 10
- Eckpunkt P2: 01
- Eckpunkt P3: 00

Durch die Anwendung der logischen Shifts resultiert folgender finaler Code: 000110.

Durch dieses Schema kann man nun schnell testen, ob ein Dreieck eine Schnittebene schneidet oder komplett auf einer der beiden Seiten liegt. Es gilt folgender Zusammenhang (vgl. [Woo+95]):

1. Code 010101: Dreieck liegt auf der Normalenseite der Schnittebene

2. Code 101010: Dreieck liegt auf der entgegengesetzten Normalenseite der Schnittebene

Nun wird folgendes Schema angewendet:

1. Um zu testen, ob das Dreieck komplett auf der Normalenseite liegt, wird eine Konjunktion zwischen dem Dreieckscode und dem zweiten obigen Code ausgeführt.

2. Um zu testen, ob das Dreieck komplett auf der entgegengesetzten Normalenseite liegt, wird eine Konjunktion zwischen dem Dreieckscode und dem ersten obigen Code ausgeführt.

3. Falls dann das Ergebnis 0 ist, war der Test erfolgreich.

Um dies zu verdeutlichen, wird Beispiel 4.2.2 angeführt, welches wieder 4.1a verwendet.

Beispiel 4.2.2. In der Schnittsituation in Abbildung 4.1a ist der Code 000110. Nun werden folgende beiden Berechnungen durchgeführt:

- Konjunktion zwischen 000110 und 010101 ergibt: 000100

- Konjunktion zwischen 000110 und 101010 ergibt: 000010

Da kein resultierender Code gleich 0 ist, liegt das Dreieck nicht komplett auf einer der beiden Seiten und muss daher für eine Einsortierung in den BSP-Baum unterteilt werden.

Durch dieses System kann die aktuelle Schnittsituation effizient und übersichtlich überprüft werden. Zum einen kann schnell festgestellt werden, ob das Dreieck komplett auf einer der beiden Seiten liegt, zum anderen ist bei einer nötigen Unterteilung ebenfalls die volle Information über alle Eckpunkte vorhanden.

Der letzte Teil der ersten Phase besteht in der Berechnung der Formfaktoren nach Kapitel 2.2.2.3. Hierbei wird für jedes Paar von Patches, welche einander zugewandt sind, eine Interaktion erstellt und diese Berechnung ausgeführt. Dabei sind zwei Patches einander zugewandt, wenn deren Normalen zueinander zeigen, also wechselseitig sichtbar sind. Nur so kann ein Strahlungsaustausch stattfinden.

4.2.3 Phase 2: Berechnung des Strahlungsaustauschs und Verfeinerung der Elemente

Nach der Fertigstellung des BSP-Baums wurde der Raum vollständig unterteilt und alle Interaktionen erstellt, weshalb nun die Berechnung des Strahlungsaustauschs erfolgen kann. Hierfür wird für alle Patches im BSP-Baum iterativ die Interaktionsliste betrachtet und abgearbeitet. Da im normalen hierarchischen Radiosity-Verfahren die Verfeinerung der Elemente nur auf Grundlage des Formfaktors, der lediglich die Geometrie der Szene berücksichtigt, ausgeführt wird, wird in dieser Arbeit die *strahlungsbasierte Zerlegung* betrachtet (vgl. [RR00, S. 529ff]). Hierbei wird neben dem Formfaktor auch der Sichtbarkeitsfaktor $V_{ij} \in [0, 1]$ berechnet, welcher den Anteil der Fläche von Element j angibt, die von Element i aus gesehen werden kann. Da in Phase 1 dieser Faktor noch nicht berechnet wurde, folgt dies nun in der zweiten Phase. Im Wesentlichen wird dies bei der Abarbeitung der Interaktionsliste berücksichtigt, wodurch hier zwei Fälle entstehen:

1. Sichtbarkeitsfaktor V_{ij} ist bereits bekannt: Falls dies der Fall ist, kann mit der endgültigen Berechnung des Radiosity-Wertes gestartet werden. In der allerersten Iteration sind alle Sichtbarkeitsfaktoren noch nicht berechnet, da in Phase 1 lediglich alle Interaktionen erstellt worden sind und die Sichtbarkeit mit -1 initialisiert wurde.

2. Sichtbarkeitsfaktor V_{ij} ist noch nicht bekannt: Hier muss dieser Sichtbarkeitsfaktor zunächst berechnet werden.

Doch wie kann der Sichtbarkeitsfaktor ermittelt werden? Um dies effizient zu berechnen, bedarf es einer Approximation, die nun im Folgenden vorgestellt werden soll (vgl. [Woo+95] und [Sin+95, S. 37ff]). Da der BSP-Baum bereits eine Raumaufteilung enthält, kann diese Information bei der Sichtbarkeitsberechnung nun berücksichtigt werden. In der zu implementierenden Anwendung werden hierzu die beiden beteiligten Elemente in 16 gleich große Teildreiecke unterteilt und deren Mittelpunkte berechnet. Zwischen diesen Mittelpunkten erfolgt dann ein Strahlungsaustausch nach Abbildung 4.2. Dieses Schema wird für jede Interaktion iterativ ausgeführt, wobei in jeder Iteration im Wesentlichen der BSP-Baum ausgehend vom Source-Element der Interaktion traversiert wird. Dies soll nun anhand der Abbildung 4.3 illustriert werden. Hier wird eine Interaktion zwischen Element B und Element I betrachtet, weshalb nun ein Lichtstrahl von B nach I ermittelt wird. Als nächstes wird der BSP-Baum ausgehend von B traversiert. Hierbei ist es wichtig zu erwähnen, dass in der Implementierung ein Testbereich eingeführt wurde, um nicht den kompletten Baum traversieren zu müssen (vgl. [Woo+95]). So wird ein Polygon nur auf einen Schnitt überprüft, wenn dieses zwischen dem Quell- und Zielelement liegt (also in

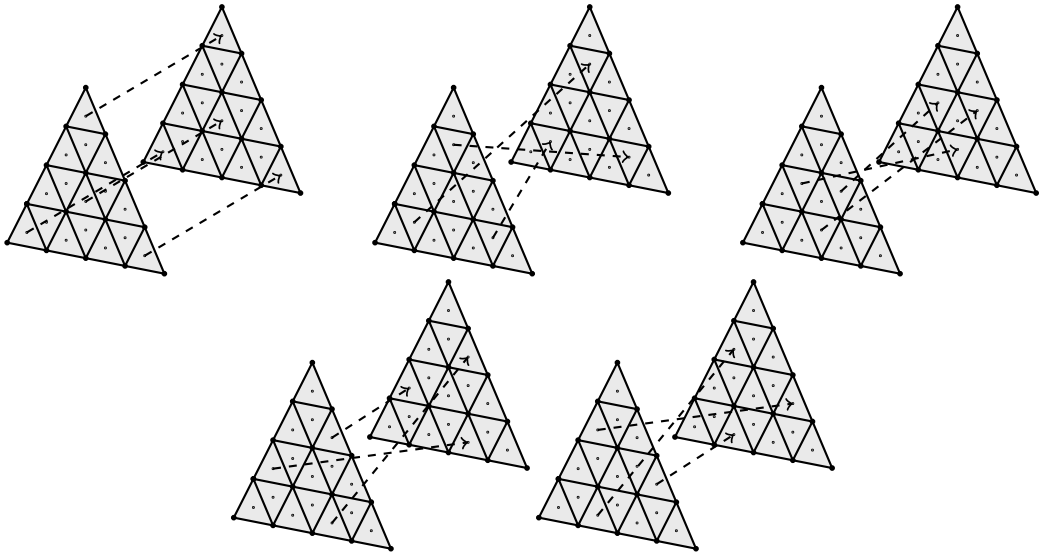


Abb. 4.2: Schema der Sichtbarkeitsberechnung

einem definierten Bereich). Falls das aktuelle Polygon außerhalb dieses Bereichs liegt, wird nur derjenige Halbraum untersucht, der den Schnittbereich enthält. Insgesamt existieren bei der Traversierung drei Schritte (vgl. [Woo+95] und [Sin+95, S. 37ff]):

1. Da Licht nur in den positiven Halbraum des Quellknotens gesendet wird, wird in einem ersten Schritt nur der positive Kindknoten überprüft. Auf diesen wird dann eine Traversierung nach Schritt zwei durchgeführt. Dies sieht man in Abbildung 4.3 bei der Abarbeitung von Knoten B, D und C.

2. Traversierung des Unterbaums: Der aktuelle Knoten wird nun darauf überprüft, ob dieser im aktuellen Testbereich liegt. Falls die Traversierung beendet ist, wird eine Bottom-Up-Traversierung ausgeführt.

   - Aktueller Knoten ist außerhalb: Führe Traversierung auf dem Kindknoten durch, welcher denjenigen Halbraum repräsentiert, der den Testbereich enthält.

   - Aktueller Knoten ist innerhalb: Führe Inorder-Traversierung inklusive Schnittest ausgehend vom aktuellen Knoten aus.

3. Bottom-Up-Traversierung: Da nun alle Teilbäume traversiert worden sind, muss vom aktuellen Knoten ausgehend eine Bottom-Up-Traversierung ausgeführt werden. Sobald ein neuer Knoten gefunden wird, der komplett im Testbereich liegt, wird dieser zunächst auf einen Schnitt mit dem Lichtstrahl überprüft. Danach wird jeweils der noch nicht traversierte Teilbaum überprüft, sofern

dieser den Testbereich enthält. In der Abbildung 4.3 sind beispielsweise die Knoten B, D und C schon überprüft worden, weshalb nun eine Bottom-Up-Traversierung folgt und Knoten E überprüft wird. Ausgehend von diesem Knoten wird dessen Teilbaum wieder getestet.

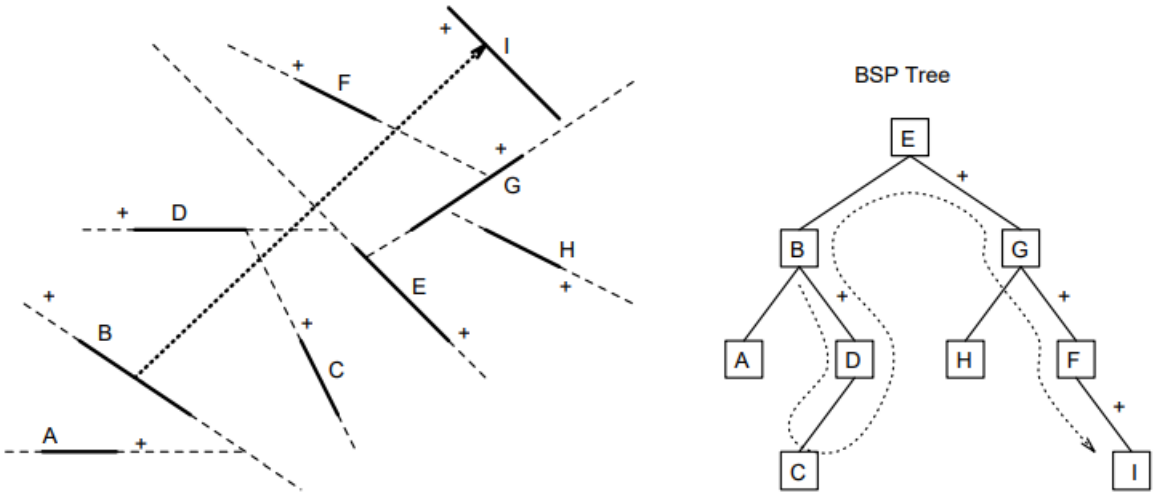


Abb. 4.3: Sichtbarkeitsberechnung anhand des BSP-Baums (vgl. [Sin+95, S. 37ff])

Dieser Algorithmus endet, sobald ein Polygon den ausgesendeten Lichtstrahl schneidet oder keine Polygone mehr verfügbar sind, wodurch nun der nächste Lichtstrahl getestet werden muss. Am Ende ist der Sichtbarkeitsfaktor der Anteil der ausgesandten Lichtstrahlen, die tatsächlich das Zielelement erreichen:

$$V_{ij} = \frac{\#\text{ Lichtstrahlen, die das Zielelement erreichen}}{\#\text{ Lichtstrahlen}} \tag{4.2}$$

Nachdem die Berechnung des Sichtbarkeitsfaktors dargelegt worden ist, kann nun der wesentliche Aspekt der Unterteilung erklärt werden. Um die Radiosity-Gleichung genau zu approximieren, dürfen sich die Radiosity-Werte an benachbarten Flächenstücken nicht großartig unterscheiden. Dies erfolgt auf Grundlage der schon erwähnten *strahlungsbasierten Zerlegung*. Man betrachte eine Interaktion zwischen Element i und j. Nun erfolgt eine Unterteilung, wenn folgende Bedingung erfüllt ist (vgl. [RR00, S. 530ff]):

$$V_{ij}F_{ij}B_j > BE_\epsilon \tag{4.3}$$

Dabei bezeichnet BE_ϵ eine definierte Fehlerschranke. Die Unterteilung ist jetzt abhängig von der Größe der betrachteten Elemente, wodurch zwei Fälle eintreten können (vgl. [RR00, S. 530ff]):

1. $A_i > A_j$: Hierbei muss Element i in vier gleich große Teilelemente unterteilt werden. Bei jedem Teilelement muss dann eine Interaktion zu Element j eingefügt werden.

2. $A_i < A_j$: Hierbei muss Element j in vier gleich große Teildreiecke unterteilt werden. Folglich müssen dann bei Element i vier Interaktionen zu jeweils einem Teilelement eingefügt werden.

Die strahlungsbasierte Zerlegung umfasst jedoch nicht nur die Unterteilung der Elemente, sondern vielmehr auch die Berechnung des Radiosity-Wertes, welche in derselben Iteration durchgeführt wird. Da alle Polygone nun in mehrere Flächenelemente zerlegt sind, die der Gesamtheit der Blätter aller Quadtrees entsprechen, bedarf es hierbei einer speziellen Berechnungsvorschrift, um alle Teildreiecke zu berücksichtigen. In einem ersten Schritt wird dabei zunächst der Radiosity-Wert der Blätter bestimmt, wobei dieser im Wesentlichen nach Gleichung 2.3 bestimmt wird. Um jedoch den Ansatz des N-Körper-Problems, der im hierarchischen Radiosity-Verfahren Anwendung findet, zu realisieren, wird nicht die Energiestrahlung jedes sichtbaren Elements berücksichtigt, sondern vielmehr nur die Energiestrahlung jedes Elements, das sich in der Interaktionsliste des jeweiligen Objektes befindet. Dies muss dadurch kein Blattknoten sein, sondern vielmehr auch eine gröbere Unterteilung. Der Radiosity-Wert wird dabei iterativ berechnet, wobei in jedem Schritt Radiosity-Werte für alle Knoten berechnet werden. Dabei müssen jedoch auch Anteile der Vorgängerknoten berücksichtigt werden, da diese Interaktionen mit größeren Flächenstücken beinhalten. Im Wesentlichen wird dadurch erst eine Top-Down-Traversierung eines Quadtrees ausgeführt, wodurch jeder Knoten einen neuen Radiosity-Wert erhält (vgl. [RR00, S. 527ff]):

$$B_i^{neu} = \begin{cases} \rho_i \sum_{j \in I(i)} F_{ij} B_j^{(l)} & \text{falls i Wurzelknoten} \\ \rho_i \sum_{j \in I(i)} F_{ij} B_j^{(l)} + B_{Elternknoten(i)}^{neu} & \text{falls i innerer Knoten} \\ \rho_i \sum_{j \in I(i)} F_{ij} B_j^{(l)} + B_{Elternknoten(i)}^{neu} + E_i & \text{falls i Blattknoten} \end{cases} \tag{4.4}$$

Dabei bezeichnet B_i den Radiosity-Wert des Elements i, ρ_i den Reflexionskoeffizient des Elements i, F_{ij} den Formfaktor und E_i die Eigenemission des Elements i. $B_j^{(l)}$ ist der Radiosity-Wert des Elements j in der letzten Iteration.
Beginnend bei den Blättern wird nun eine Bottom-Up-Traversierung ausgeführt, wodurch die Radiosity eines inneren Knotens durch Aufsummierung der mit der Fläche gewichteten Radiosity der Kinder berechnet wird. Da in der Implementierung jedes Dreieck in vier gleich große Unterteildreiecke zerlegt worden ist, genügt lediglich eine Durchschnittsbildung (vgl. [RR00, S. 527ff]):

$$B_i^{(l+1)} = \begin{cases} B_i^{neu} & \text{falls i Blattknoten} \\ \frac{1}{4} \sum_{j \in Kindknoten(i)} B_j^{(l+1)} & \text{falls i innerer Knoten} \end{cases} \tag{4.5}$$

Sobald diese Berechnung für jedes Element ausgeführt worden ist, gilt es in einer nächsten Iteration anhand weiterer Unterteilungen an Genauigkeit zu gewinnen.

Sollte jedoch Konvergenz eingetreten sein, wird die Abarbeitung dieser Phase beendet und die letzte Phase gestartet.

4.2.4 Phase 3: Normalisierung

Bisher sind lediglich Radiosity-Werte für Elemente, also Flächenstücke, berechnet worden. Damit diese jedoch in der Computergraphik eine Schattierung ergeben, müssen alle Eckpunkte ebenfalls mit Radiosity-Werten versehen werden. Dies dient dann als Basis für eine Interpolation der Schattierung auf einem Polygon, zum Beispiel mittels des *Gouraud-Shadings* (vgl. [BGZ02, S. 188f]).

Um alle Eckpunkte mit Radiosity-Werten zu versehen, ist eine Durchschnittsberechnung nötig, da jeder Eckpunkt in mehreren Polygonen enthalten sein kann. So ist es nicht möglich, jedem Eckpunkt die Radiosity des korrespondierenden Flächenstücks zuzuweisen. In der Implementierung wird zunächst für jeden Eckpunkt die Radiosity der angrenzenden Fläckenstücke gewichtet mit deren Flächeninhalt aufsummiert, wobei dann in einem nächsten Schritt durch die Summe der Gewichte geteilt wird. Dadurch wird eine Durchschnittsberechnung realisiert, die im Wesentlichen in zwei Phasen aufgeteilt ist: Aufsummierung und Normalisierung (vgl. [Woo+95]).

4.2.5 Parallelisierung mit Weave

Im Rahmen dieser Arbeit sollte das hierarchische Radiosity-Verfahren parallel implementiert werden. Wie in Kapitel 2.1 schon erwähnt, ist dieser Algorithmus irregulär. Dies sieht man vor allem an der unterschiedlichen Anzahl an Blättern in den Quadtrees der einzelnen Eingabepolygone, welche jeweils verschiedene Interaktionsmuster besitzen können. Zusätzlich können sich die Datenstrukturen in jeder Iteration nochmals ändern, sobald Polygone zerlegt werden müssen und dadurch neue Interaktionen erzeugt werden. Diese starke Ausprägung der Irregularität erfordert eine Parallelisierung durch einen taskorienterten Ansatz, wodurch es nötig ist, den Algorithmus in unabhängige Tasks zu unterteilen, die für sich parallel ausgeführt werden können (vgl. [RR00, S. 532ff] und [Kor01, S. 17ff]). Zu beachten ist, dass bei jedem Zugriff auf gemeinsame Datenstrukturen Locks verwendet werden müssen. Der Ansatz der Taskorientierung soll nun im Folgenden anhand jeder Phase erläutert werden.

In Phase 1 gilt es, die Eingabedaten einzulesen, in den BSP-Baum zu sortieren und korrespondierende Formfaktoren zu berechnen. Da die Struktur des BSP-Baums von der Reihenfolge der Einsortierung der Polygone abhängt und dieser Schritt nur einen sehr kleinen Teil der Gesamtlaufzeit benötigt, wird dieser Schritt sequentiell

ausgeführt, um einen eventuellen Overhead und weitere Irregularität durch unterschiedlich erzeugte BSP-Bäume zu vermeiden (vgl. [Kor01, S. 17]). Daher existiert in Phase 1 lediglich ein Tasktyp für die Berechnung der Formfaktoren, da dies voneinander unabhängig durchgeführt werden kann (vgl. [Woo+95] und [RR00, S. 533]). Dieser Task ist in Algorithmus 1 nochmals dargestellt.

Input: Elemente E_i und E_j
Output: Interaktionen zwischen E_i und E_j mit F_{ij} und F_{ji}
begin
  Berechne Interaktion I_{ij} zwischen E_i und E_j mit Formfaktor F_{ij} und Fehler;
  Berechne Interaktion I_{ji} zwischen E_j und E_i mit Formfaktor F_{ji} und Fehler;
  Füge I_{ij} in die Interaktionsliste von E_i ein;
  Füge I_{ji} in die Interaktionsliste von E_j ein;
end

Algorithmus 1: Formfaktor-Task für Elemente E_i und E_j

Da nun die Berechnung der Formfaktoren nebenläufig durchgeführt wird, besteht hier die Möglichkeit der Überlappung von Phase eins und zwei. Während Interaktionen noch berechnet werden, kann auf den bisher bestehenden Interaktionen und Formfaktoren bereits ein Strahlungsaustausch durchgeführt werden, um den Parallelitätsgrad zu erhöhen.

In Phase 2 gilt es nun den eigentlichen Strahlungsaustausch zu berechnen, gegebenenfalls abhängig vom Fehler weitere Unterteilungen der Dreiecke vorzunehmen und Sichtbarkeitsfaktoren zu ermitteln. Daraus resultieren folgende Tasks (vgl. [Woo+95] und [RR00, S. 533]):

- Ray-Task: Berechnet den Radiosity-Wert für ein gegebenes Element, indem die Interaktionsliste abgearbeitet wird. Falls der Fehler zu groß ist, wird eine Unterteilung des korrespondierenden Dreiecks vorgenommen.

- Visibility-Task: Berechnet den Sichtbarkeitsfaktor der korrespondierenden Interaktion

Algorithmus 2 zeigt die Realisierung des Ray-Tasks. Wichtig ist hierbei die Barrier, welche auf die Beendigung der Subtasks wartet, um korrekte Ergebnisse zu berechnen. Diese erzeugt jedoch kein klassisches Warten, sondern lässt den aufrufenden Thread dennoch weitere Tasks ausführen, wodurch die Effizienz gesteigert werden kann. Die beiden Variablen BF-Epsilon und Area-Epsilon repräsentieren dabei die beiden gleichnamigen Aufrufparameter, welche bereits in der Tabelle 4.1 erläutert worden sind.

```
Input: Element i
Output: Radiosity-Wert des Elements i, gegebenenfalls mit Unterteilung in
        Unterteildreiecke
begin
    forall Interaktionen I_ij do
        Berechne den gesamten Fehler err_total;
        if err_total > BF-Epsilon then
            if A_i > A_j then
                if A_i > Area-Epsilon then
                    Unterteile Element i;
                    Erzeuge ggf. Formfaktor-Tasks für neue Elemente;
                end
            else
                if A_j > Area-Epsilon then
                    Unterteile Element j;
                    Erzeuge ggf. Formfaktor-Tasks für neue Elemente;
                end
            end
        end
        if Sichtbarkeitsfaktor V_ij unbekannt then
            Erzeuge Visibility-Task;
        end
    end
    if alle Sichtbarkeitsfaktoren für Element i wurden berechnet then
        Sammle Strahlung auf;
        if Element i ist kein Blattknoten then
            Erzeuge Ray-Tasks für alle Kindknoten;
        end
        Warte auf Beendigung der Subtasks;
        Berechne die eigene Radiosity durch Durchschnittsbildung der
          Radiosity-Werte der Kindknoten;
    end
end
```

Algorithmus 2: Ray-Task für ein Element i

Algorithmus 3 veranschaulicht die Realisierung der Sichtbarkeitsberechnung, welche durch einen Ray-Task erzeugt werden.

Input: Element i mit Interaktion I_{ij}
Output: Radiosity-Wert des Elements i, gegebenenfalls mit Unterteilung in Unterteildreiecke
begin
  Unterteile i in 16 Unterteildreiecke und berechne alle Mittelpunkte;
  Berechne alle Lichtstrahlen nach Schema 4.2;
  reach $\leftarrow 0$;
  forall *Lichtstrahl* **do**
    traversiere BSP-Baum nach Schema 4.3;
    if *Lichtstrahl erreicht Ziel* **then**
      reach $\leftarrow$ reach $+ 1$;
    end
  end
  return $\frac{reach}{16}$
end

Algorithmus 3: Visibility-Task für eine Interaktion I_{ij}

In dieser Phase findet also Parallelität zum einen in der Berechnung des Strahlungsaustauschs, zum anderen aber auch bei der Berechnung der Sichtbarkeitsfaktoren statt.

In Phase 3 gilt es, die berechneten Radiosity-Werte der gesamten Flächenstücke auf die einzelnen Eckpunkte anhand einer Durchschnittsberechnung zu übertragen. Dies kann durch einen AVG-Task ausgedrückt werden, welche bei der Traversierung des Baumes erzeugt werden und bereits dann schon parallel abgearbeitet werden können (vgl. [Woo+95] und [RR00, S. 533]):

Input: Element i
Output: Radiosity-Werte der drei Eckpunkte
begin
  if i *ist Blattknoten* **then**
    Aufsummierung der mit dem Flächeninhalt der jeweiligen angrenzenden Dreiecke gewichteten Radiosity-Werte ODER Normalisierung;
  else
    Erzeuge AVG-Task für jeden Kindknoten;
  end
end

Algorithmus 4: AVG-Task für eine Interaktion I_{ij}

Nachdem nun die Funktionalität jedes Tasktyps vorgestellt worden ist, wird in einem nächsten Algorithmus 5 der Zusammenhang aller Phasen mit taskorientierter Parallelisierung verdeutlicht. Hierbei wird vor allem nochmal deutlich, dass sich die

Berechnung der Interaktionen in Phase eins mit dem Strahlungsaustausch in Phase zwei überlappen können, was jedoch durch die erste Barrier in der zweiten Phase ausgeglichen wird. Weiterhin sieht man vor allem die Zweiteilung der dritten Phase in Aufsummierung und Normalisierung, welche durch eine Barrier getrennt werden müssen.

```
Input: Menge von Polygonen und Lichtquellen
Output: Radiosity-Werte aller Polygone
begin
    /* Phase 1: Einlesen der Eingabedaten, Einsortierung in den
       BSP-Baum, Formfaktor-Berechnung                          */
    forall Eingabepolygone P do
        if P ist Rechteck then
            Unterteile P in zwei Dreiecke;
        end
        Sortiere P in den BSP-Baum ein;
        Erzeuge Formfaktor-Task(s) (Alg. 1) für P;
    end
    /* Phase 2: Berechnung der Radiosity-Werte, Verfeinerung
       der Dreiecke, Sichtbarkeitsberechnung                    */
    repeat
        forall Dreiecke P im BSP-Baum do
            Erzeuge Ray-Task (Alg. 2) für P;
        end
        Warte auf die Beendigung aller Ray-Tasks;
    until Konvergenz der Radiosity-Werte;
    /* Phase 3: Durchschnittsberechnung                         */
    forall Dreiecke im BSP-Baum do
        Erzeuge AVG-Task (Alg. 4) für die gewichtete Aufsummierung aller
         Radiosity-Werte;
    end
    Warte auf die Beendigung aller Tasks;
    forall Dreiecke im BSP-Baum do
        Erzeuge AVG-Task (Alg. 4) für die Normalisierung;
    end
    Warte auf die Beendigung aller Tasks;
end
```

Algorithmus 5: Paralleler hierarchischer Radiosity-Algorithmus

4.2.6 Optimierungen

Neben der eigentlichen Implementierung des parallelen hierarchischen Radiosity-Verfahrens, deren konkrete Realisierung im letzten Kapitel dargelegt worden ist, gilt es ebenfalls in dieser Arbeit mögliche Optimierungen zu untersuchen. Diese sollen nun im Folgenden vorgestellt werden, wobei der Vollständigkeit halber ebenfalls Aspekte aufgeführt sind, die von der Theorie her schon behandelt worden sind.

4.2.6.1 BSP-Baum

Wie schon in vorherigen Kapiteln erwähnt, wird in der Implementierung ein BSP-Baum verwendet, der die Eingabeszene räumlich aufteilt. Dies hat den Vorteil, dass alle Sichtbarkeitsfaktoren auf Grundlage dieses Baumes effizient berechnet werden können, da man hierbei die im Voraus berechnete Raumaufteilung nutzen kann (vgl. [Sin+95, S. 37ff]). Da die Generierung des BSP-Baumes einen sehr geringen Anteil der Gesamtlaufzeit darstellt, kann dieses Verfahren als effizient betrachtet werden (vgl. [Kor01, S. 17]).

4.2.6.2 Einführung eines Testbereichs in der Sichtbarkeitsberechnung

In Kapitel 4.2.3 ist schon erwähnt worden, dass für die Berechnung des Sichtbarkeitsfaktors bei der Traversierung des BSP-Baums ein Testbereich eingeführt wurde. Falls beispielsweise ein Lichtstrahl von Element i nach Element j ausgetauscht wird und ermittelt werden soll, ob dieser ein anderes Element schneidet, muss kein anderes Dreieck untersucht werden, das außerhalb des möglichen Schnittbereichs liegt. Sollte beispielsweise der positive Kindknoten von Element i außerhalb des Schnittbereichs liegen, muss zum einen kein Schnitt berechnet werden, zum anderen kann ein Teil des korrespondierenden Unterbaums abgeschnitten werden, was zu einer Zeitersparnis bei der Traversierung führt. Anstatt jedoch einen festen Bereich zu definieren, wird dieser in der Implementierung adaptiv an die aktuelle Situation angepasst (vgl. [Woo+95]).

4.2.6.3 Einführung eines Patch-Caches in der Sichtbarkeitsberechnung

Neben den bisherigen Optimierungen wurde noch ein Patch-Cache bei der Sichtbarkeitsberechnung eingeführt, um die Traversierung des BSP-Baumes weiter zu verkürzen. Da der Sichtbarkeitstest parallel ausgeführt wird, existiert dieser Cache lokal für jeden Thread einzeln. Ein Polygon wird in den Cache eingefügt, wenn dieser einen zuvor berechneten Lichtstrahl geschnitten hat. Falls nun ein neuer Lichtstrahl untersucht werden muss, wird zunächst der Cache nach einem schneidenden Polygon überprüft. Ist dies der Fall, muss keine Traversierung des BSP-Baums durchgeführt werden. Ansonsten wird der normale Sichtbarkeitstest berechnet. Falls dann ein potentiell schneidendes Polygon existiert, wird lediglich überprüft, ob dieses im Cache vorhanden ist. Falls ja, muss kein konkreter Schnitttest mehr durchgeführt werden, da dieses Polygon den Lichtstrahl nicht schneiden kann. Sonst wäre der Anfangstest durch die Überprüfung des Caches erfolgreich gewesen (vgl. [Woo+95]).

4.2.6.4 Einführung von Objekt-Caches

Wie schon in vorherigen Kapiteln erwähnt, ist ebenfalls eine zweite statische Version des hierarchischen Radiosity-Verfahrens implementiert worden. Hierbei wurde für alle Objekttypen ein Cache erstellt, der bei Initialisierung des Programms gefüllt wird. Falls nun ein Objekt benötigt wird, kann dieses aus dieser Liste entnommen werden, wodurch keine Allokation zur Laufzeit stattfinden muss. Hierdurch entsteht vor allem ein Performancevorteil, welcher auch im folgenden Kapitel 5 deutlich wird.

5 Experimentelle Auswertung

Nachdem sowohl alle theoretischen Grundlagen als auch die Realisierung der Implementierung zugrunde gelegt worden sind, wird nun das Laufzeitverhalten der Anwendung genauer analysiert.

5.1 Testprobleme

Um ein aussagekräftiges Ergebnis in der Laufzeitanalyse zu erzeugen, werden folgende Szenen betrachtet, die aufsteigend nach der Größe aufgelistet sind:

- „*largeroom*“ aus SPLASH-2[1] (vgl. [Woo+95])
  - Anzahl Eingabepolygone: 305
  - Anzahl Patches: 532
  - Anzahl Elemente: 20968
  - Anzahl Interaktionen: 239903
- „*Halle*“ aus der Arbeit [Kor01] (XRAF-Format)
  - Anzahl Eingabepolygone: 962
  - Anzahl Patches: 1157
  - Anzahl Elemente: 1157
  - Anzahl Interaktionen: 438890

[1] `https://github.com/staceyson/splash2`, abg. am 21.04.2021

- „*Raum11*“ aus der Arbeit [Kor01] (XRAF-Format)
  - Anzahl Eingabepolygone: 638
  - Anzahl Patches: 2957
  - Anzahl Elemente: 2957
  - Anzahl Interaktionen: 1701208

Hierbei fällt auf, dass sich bei *largeroom* die Anzahl der Elemente signifikant von der Anzahl der Patches unterscheidet, was auf eine weitere Unterteilung durch das hierarchische Verfahren schließen lässt. Bei *Halle* und *Raum11* existiert keine weitere Aufteilung, da die Patches bereits schon so klein gewählt worden sind, sodass der Fehler unterhalb der definierten Grenze liegt. Dies wird auch einen wesentlichen Einfluss auf die Laufzeit der dritten Phase nehmen, da diese lediglich auf den Eckpunkten der Elemente operiert.

5.2 Verwendetes System

Wie in Kapitel 3.2 schon erwähnt, ist Weave kompatibel zu den Architekturen x86-64 und ARM, weshalb diese in die Laufzeitmessung einbezogen werden sollen.

Für die Architektur x86-64 werden alle Messungen auf dem Rechner *clust03* des Lehrstuhls AI2 der Universität Bayreuth durchgeführt, welcher 2019 in Betrieb genommen worden ist. Auf diesem sind zwei Prozessoren des Typs AMD Epyc 7551 (x86-64) verbaut, die jeweils 32 Kerne aufweisen, wobei jeder Kern zwei Threads unterstützt. Insgesamt sind dem Rechner 512 Gigabyte Arbeitsspeicher zugeordnet.

Für die Architektur ARM werden alle Messungen auf dem Rechner *Thunder* des Lehrstuhls AI2 der Universität Bayreuth durchgeführt, welcher 2018 in Betrieb genommen worden ist. Auf diesem sind zwei Prozessoren des Typs Thunder X2 (ARM v8) verbaut, die jeweils 28 Kerne aufweisen, wobei jeder Kern vier Threads unterstützt. Insgesamt sind dem Rechner 128 Gigabyte Arbeitsspeicher zugeordnet.

5.3 Ergebnisse

Nachdem nun alle Testprobleme und die verwendeten Systeme vorgestellt worden sind, werden in einem nächsten Schritt Laufzeitanalysen präsentiert. Um diese Ergebnisse zu generieren, existiert für jedes Verfahren ein Skript, welches als Aufrufparameter die maximale Anzahl an Threads und Iterationen benötigt. Beim Ausführen wird dann das korrespondierende Verfahren iterativ gestartet, wobei in jedem Durchlauf die Threadanzahl um eins erhöht und die Laufzeit gemessen wird. Da die Ausführungsreihenfolge der Tasks bei dem parallelen Radiosity-Verfahren variieren kann, unterscheiden sich ebenfalls die Anzahl der Interaktionen der einzelnen Szenen. Um jedoch stabile Ergebnisse ermitteln zu können, wird jede Iteration mehrfach durchgeführt und dann der Durchschnitt der Laufzeiten gebildet. Aus diesen Daten kann nun der Speedup berechnet werden, worauf dann alle Ergebnisse in Diagrammen veranschaulicht werden können.

5.3.1 AMD Epyc 7551 (x86-64)

Für eine genaue Analyse wird zunächst die eigene Implementierung untersucht, worauf diese dann mit der modifizierten SPLASH-2 Version aus [Kor01] verglichen wird. Hierfür soll die Gesamtlaufzeit, der Speedup und die Laufzeiten der einzelnen Phasen herangezogen werden. Da die Vergleichsvariante auf der SPLASH-2 Suite basiert und diese in der Threadanzahl (max. 24 Threads) begrenzt ist, erfolgt die Analyse der eigenen Implementierung mit 128 Threads, der Vergleich jedoch mit 24 Threads. Jede Variante wird mit dem *gcc* Compiler übersetzt.

5.3.1.1 Analyse der eigenen Implementierung

Wie in Kapitel 4.2 schon beschrieben, sind zwei Versionen des hierarchischen Radiosity Verfahrens implementiert worden. Die dynamische Version allokiert alle benötigten Objekte zur Laufzeit dynamisch, wodurch man diesen Algorithmus auf beliebige Eingabegrößen ausführen kann, ohne den Quellcode ändern zu müssen. Die statische Variante hält alle Objekte vorallokiert in einer Liste, wodurch diese zur Laufzeit nur noch entnommen werden müssen. Da die modifizierte SPLASH-2 Variante nur separate Phasen unterstützt, also Phase eins und zwei getrennt behandelt werden, wird die eigene Implementierung ebenfalls auf diese Art analysiert, weshalb der Aufrufparameter *-p* verwendet werden muss.

Bei der Betrachtung der sequentiellen Gesamtlaufzeit beider Varianten in Abbildung 5.1 fällt zunächst auf, dass das Verfahren bei *Raum11* am längsten rechnet, gefolgt von *Halle* und *Largeroom*, was mit der Anzahl an Interaktionen korrespondiert.

Hierbei bemerkt man vor allem die Verwendung eines Patch-Caches im Sichtbarkeitstest, der die Gesamtlaufzeit signifikant verringern kann. Dieser Effekt nimmt jedoch mit einer höheren Anzahl an Threads ab, da dann Sichtbarkeitsberechnungen schneller abgeschlossen werden können und eine Verkürzung der Traversierung nicht mehr so stark in das Gewicht fällt. Weiterhin sieht man bei der statischen Variante eine kontinuierliche Verbesserung der Gesamtlaufzeit, was jedoch bei der dynamischen Variante nicht zu beobachten ist. Hier verringert sich zwar die Laufzeit bis ungefähr 30 Threads, steigt jedoch dann wieder an und bleibt dann auf einem konstanten Level. Dies kann man vor allem durch die dynamische Allokation während der Laufzeit erklären, wodurch der allokierende Thread warten muss. Bei der Verwendung von vielen Threads tritt diese Situation sehr häufig auf, weshalb die Skalierbarkeit und damit auch der Nutzen der Parallelität verloren geht. Auch schwankt die Kurve im Gegensatz zur statischen Variante äußerst stark, was ebenfalls an der Allokation während der Laufzeit und der damit einhergehenden unberechenbaren Wartezeit liegt. Insgesamt kann man beobachten, dass die dynamische Variante sowohl besser skaliert als auch eine geringere Laufzeit aufweist.

Zur Bewertung der Parallelität ist jedoch nicht nur die Betrachung der Laufzeit, sondern auch die des Speedups von Bedeutung. Abbildung 5.5 zeigt den Speedup in Abhängkeit der Variante und Threadanzahl. Bei der dynamischen Variante (Abbildung 5.5a) sieht man, dass der Speedup zunächst relativ gut ansteigt, jedoch dann ab ungefähr 15 bis 20 Threads wieder absinkt und dann konstant bleibt. Dies war auch schon in der vorhergegangenen Laufzeitanalyse der Fall und spiegelt sich nun auch im Speedup wider. Dies bekräftigt die Begründung, dass die Allokation während der Laufzeit den Parallelitätsgrad der Anwendung hemmt, da zu viele Threads auf die Speicherreservierung warten müssen. Bei der statischen Variante (Abbildung 5.5b) zeigt sich ein zum Laufzeitverhalten korrespondierendes Bild. Hier kann man sagen, dass bei den Szenen *Halle* und *Raum11* kontinuierlich ein starker Anstieg zu verzeichnen ist. Insgesamt ist bei der Szene *Raum11*, welche die meisten Interaktionen besitzt, der größte Speedup mit ungefähr 55 bei 128 Threads zu verzeichnen. Bei *largeroom* steigt die Kurve zunächst auch stark an, bleibt dann jedoch auf einem konstanten Level. Dies ist mit der eher geringen Anzahl an Interaktionen zu erklären, deren Berechnungen nicht mehr gut mit vielen Threads skalieren, da die vorhanden Lockmechanischem die Performance limitieren.

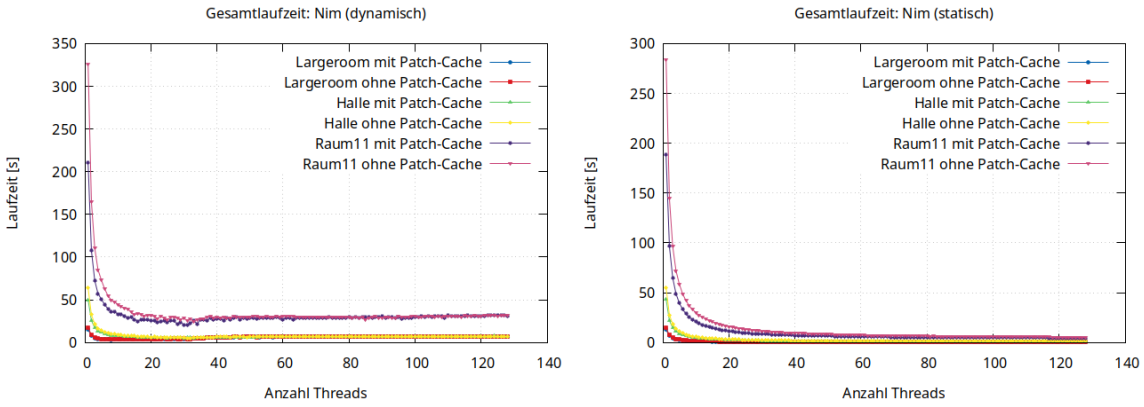
Um die Zusammensetzung der Gesamtlaufzeit genauer analysieren zu können, werden in den Abbildungen 5.2, 5.3 und 5.4 die Laufzeiten aller drei Phasen dargestellt.

Phase eins besteht aus dem Einlesen der Eingabedaten, dem Einsortieren in den BSP-Baum und der parallelen Berechnung der Formfaktoren. Bei der Betrachtung der Laufzeit bemerkt man in beiden Diagrammen 5.2a und 5.2b einen Anstieg der Laufzeit bei allen drei Szenen, wobei dieser bei der statischen Variante weniger ausgeprägt ist. Bei der dynamischen Variante steigt die Laufzeit mit mehreren Threads sogar über die Laufzeit der sequentiellen Ausführung und verbleibt dann auf einem

konstanten Level ab ungefähr 40 Threads. Dies kann man wieder auf die Allokation während der Laufzeit zurückführen, wodurch Wartezeiten resultieren. Bei der statischen Variante steigt die Laufzeit nicht über die der sequentiellen Berechnung, bleibt jedoch auch ab 40 Threads konstant. Insgesamt fällt diese jedoch wesentlich geringer aus.

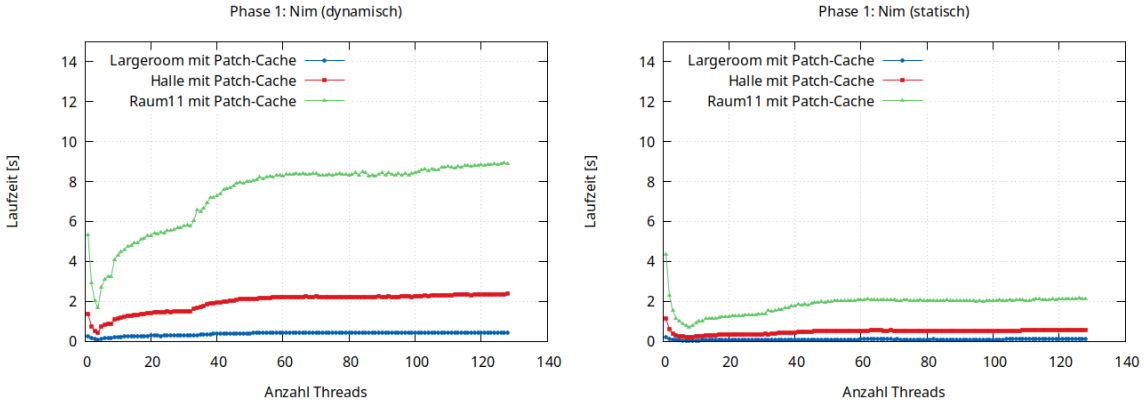
Der wesentliche Teil der Berechnungen des hierarchischen Radiosity-Verfahrens findet im Strahlungsaustausch, also in der zweiten Phase, statt. Dies zeigt sich auch anhand der Diagramme 5.3a und 5.3b, welche eine gute Skalierbarkeit aufzeigen, wobei insgesamt die statische Variante wieder besser abschneidet. Dies korrespondiert auch mit der Gesamtlaufzeit, welche vor allem von der zweiten Phase geprägt wird.

Die letzte Phase beinhaltet eine Normalisierung der einzelnen Eckpunkte. Insgesamt zeigen die beiden Diagramme 5.4a und 5.4b keine gute Skalierbarkeit, wobei dies bei der dynamischen Variante stärker ausgeprägt ist. In allen drei Szenen liegt hier die parallele Laufzeit ab einer gewissen Anzahl an Threads über der Laufzeit der sequentiellen Ausführung. Bei beiden Varianten ist dies bei der Szene *largeroom* am stärken ausgeprägt, da hier alle Eingabepolygone weiter unterteilt worden sind und demnach auch mehr Berechnungen auf insgesamt mehr Eckpunkten ausgeführt werden müssen.



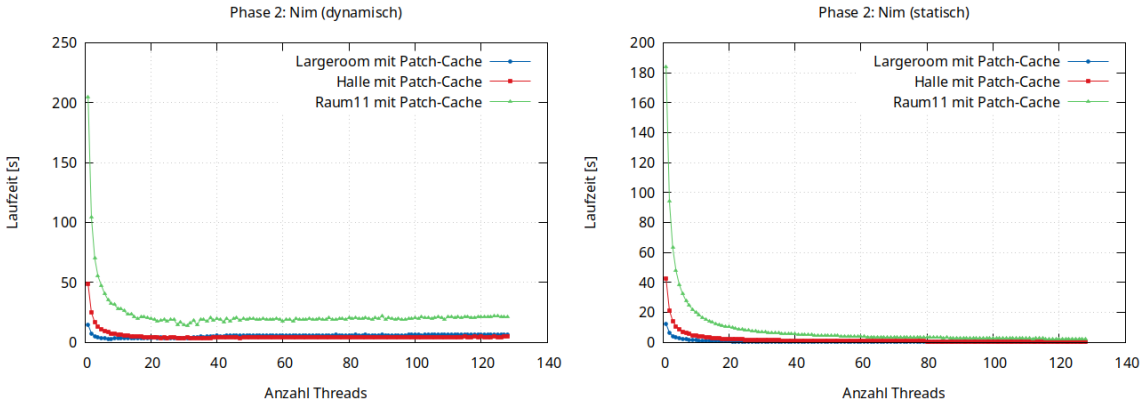
(a) dynamische Variante **(b)** statische Variante

Abb. 5.1: Gesamtlaufzeiten (Nim) auf x86-64



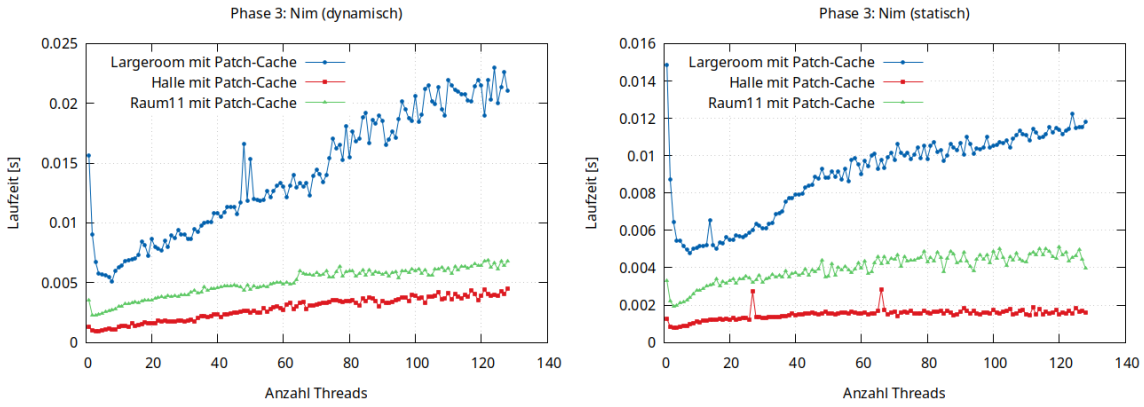
(a) dynamische Variante **(b)** statische Variante

Abb. 5.2: Laufzeiten Phase 1 (Nim) auf x86-64



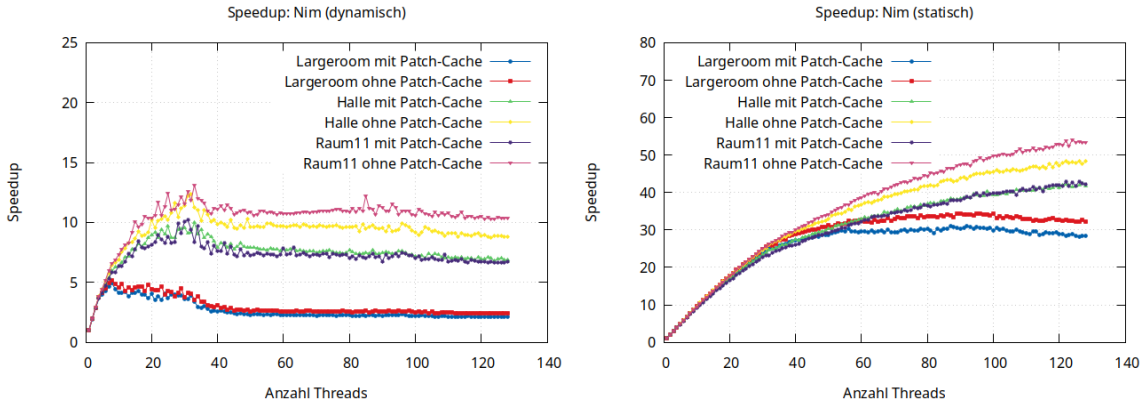
(a) dynamische Variante **(b)** statische Variante

Abb. 5.3: Laufzeiten Phase 2 (Nim) auf x86-64



(a) dynamische Variante (b) statische Variante

Abb. 5.4: Laufzeiten Phase 3 (Nim) auf x86-64



(a) dynamische Variante (b) statische Variante

Abb. 5.5: Speedups (Nim) auf x86-64

5.3.1.2 Vergleich mit der modifizierten SPLASH-2 Variante

Nachdem nun die beiden Versionen der eigenen Implementierung analysiert worden sind, gilt es nun in einem nächsten Schritt, einen Vergleich zwischen der modifizierten SPLASH-2 Variante aus [Kor01] und der eigenen Implementierung anzustellen. Da in SPLASH-2 lediglich 24 Threads möglich sind und die Berechnungen auf separaten Phasen mit Patch-Cache ausgeführt werden, werden die Diagramme bezüglich der eigenen Implementierung auf den Bereich bis 24 Threads mit Patch-Cache angepasst.

In Abbildung 5.6 ist die Gesamtlaufzeit der beiden eigenen Implementierungen und die der modifizierten SPLASH-2 Variante dargestellt. Hierbei fällt zunächst auf, dass die sequentielle Laufzeit der modifizierten SPLASH-2 Variante für alle drei betrachteten Szenen nur halb so hoch ausfällt, wodurch die eigene Implementierung

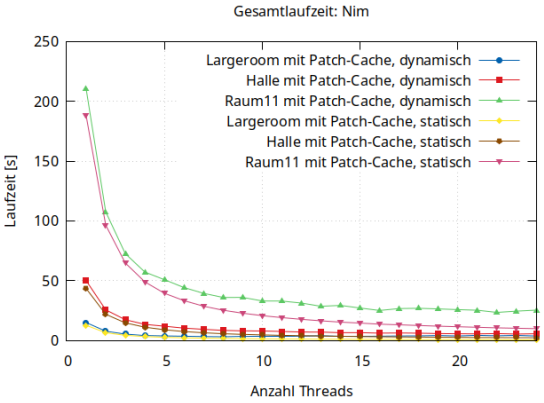
insgesamt sequentiell langsamer arbeitet. Bei der Erhöhung der Threadanzahl bemerkt man jedoch, dass beide Varianten skalieren und sich die Gesamtlaufzeiten bei hoher Anzahl an Threads sogar annähern. Beispielsweise erreicht man in SPLASH-2 eine Laufzeit von ca. 20 Sekunden nach ungefähr 8 Threads bei der Szene *Raum11*. Dies erreicht die eigene statische Implementierung ebenfalls, jedoch erst nach 15-20 Threads. Dies trifft ebenfalls auf die beiden anderen Szenen zu, wodurch man sagen kann, dass die eigene Implementierung insgesamt mehr Threads benötigt, um genauso effizient zu sein wie die SPLASH-2 Variante.

Diese Beobachtung korrespondiert mit dem Verlauf der Speedups, welche in Abbildung 5.10 dargestellt sind. Hierbei würde man zunächst vermuten, dass die eigene statische Implementierung ingesamt effizienter arbeitet, da hier ein kontinuierlicher Anstieg zu verzeichnen ist. Dieser Anstieg ist bei der SPLASH-2 Variante jedoch nur bis ungefähr 8 Threads zu beobachten, danach sinkt dieser sogar wieder ab und bleibt im Großen und Ganzen konstant. Da sich der Speedup jedoch auf die sequentielle Laufzeit bezieht und diese in der eigenen Implementierung von Grund auf schon langsamer ist, kann der Speedup insgesamt nur besser ausfallen. Obwohl also das Radiosity-Verfahren in der SPLASH-2 Suite einen wesentlich schlechteren Speedup aufweist, arbeitet die Variante dennoch aufgrund der besseren absoluten Laufzeit effizienter und benötigt zum Erreichen der Minimallaufzeit auch nicht so viele Threads wie die eigenen Implementierung.

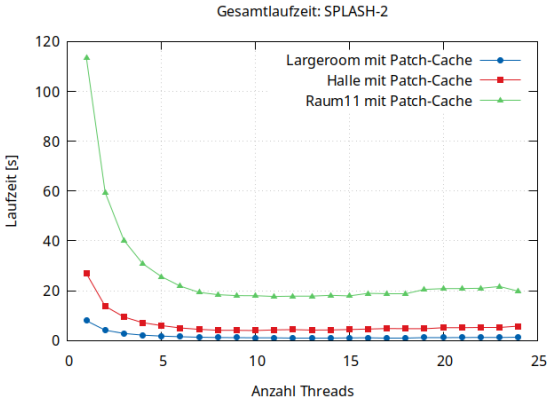
Abbildungen 5.7, 5.8 und 5.9 zeigen die Laufzeiten für alle drei Phasen des Radiosity-Verfahrens. In Phase eins kann man hierbei ein relativ ähnliches Verhalten beobachten. Wie schon im vorherigen Kapitel erwähnt, sinkt bei der eigenen Implementierung die Laufzeit zunächst ab, bevor sie dann wieder ansteigt. Dieser Anstieg ist bei der dynamischen Variante stärker als bei der statischen Variante. Dieser grobe Verlauf ist ebenfalls bei der SPLASH-2 Suite zu verzeichnen, so sinkt die Laufzeit zunächst ab, steigt jedoch dann sehr stark an. So ist die parallele Laufzeit der Phase eins bei allen drei Szenen über der Gesamtlaufzeit der sequentiellen Ausführung und liegt mit mehr als 15 Sekunden bei 20-24 Threads wesentlich darüber.

In der zweiten Phase zeigen sich exakt die Effekte, die bereits beim Vergleich der Gesamtlaufzeiten zwischen der eigenen Implementierung und der SPLASH-2 Variante vorgestellt worden sind. Dies ist hiermit zu erklären, dass die zweite Phase die eigentliche Berechnungsphase des hierarchischen Radiosity-Verfahrens ist.

In der letzten Phase sind die Resultate relativ ähnlich. Bei der Betrachtung der Szene *largeroom* fällt ein relativ ähnlicher Kurvenverlauf auf, wobei die Laufzeit bei der SPLASH-2 Variante wesentlich geringer ausfällt. Insgesamt sinkt die Laufzeit zunächst stark ab und bleibt dann auf einem konstanten Level. Man kann also sagen, dass die dritte Phase bei dieser Szene relativ gut skaliert. Im Gegensatz dazu stehen die beiden anderen Szenen *Halle* und *Raum11*, bei denen die Laufzeiten bei einer parallelen Ausführung stark ansteigen. Dies liegt jedoch, wie schon im vorherigen Kapitel erwähnt, an der Beschaffenheit der Eingabeszenen. Die Absolutlaufzeiten der beiden Szenen sind allerdings bei der eigenen Implementierung geringer.

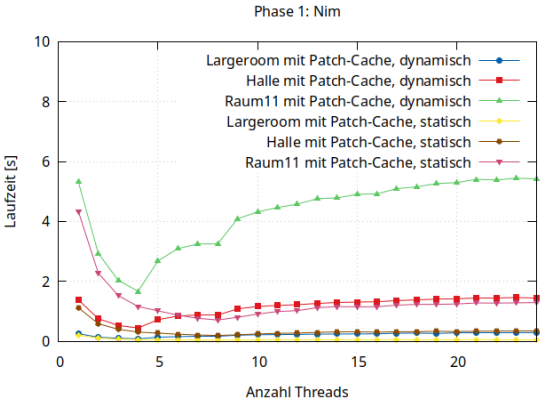


(a) Nim

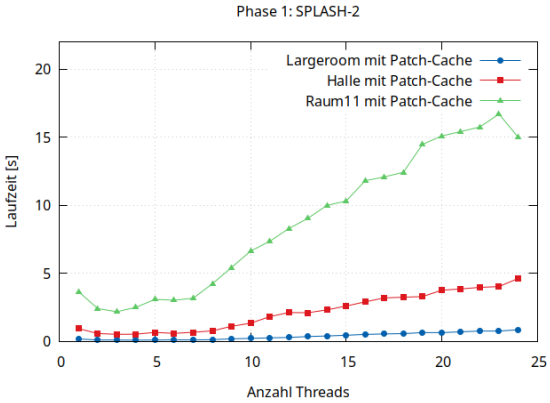


(b) Modifizierte SPLASH-2 Variante

Abb. 5.6: Gesamtlaufzeiten: Vergleich Nim und SPLASH-2 auf x86-64

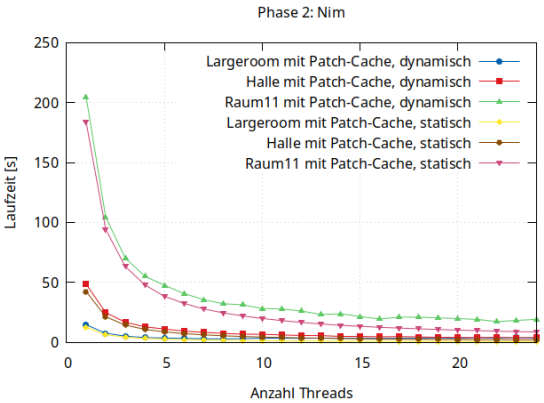


(a) Nim

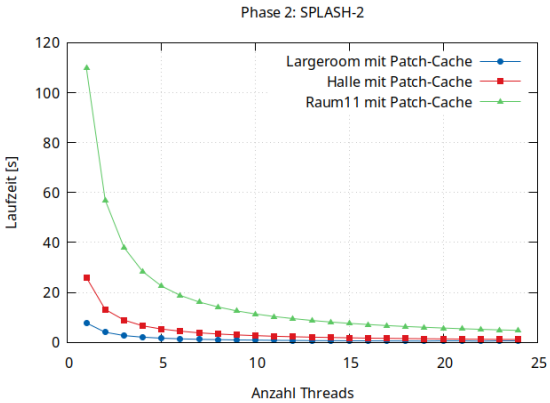


(b) Modifizierte SPLASH-2 Variante

Abb. 5.7: Laufzeiten Phase 1: Vergleich Nim und SPLASH-2 auf x86-64

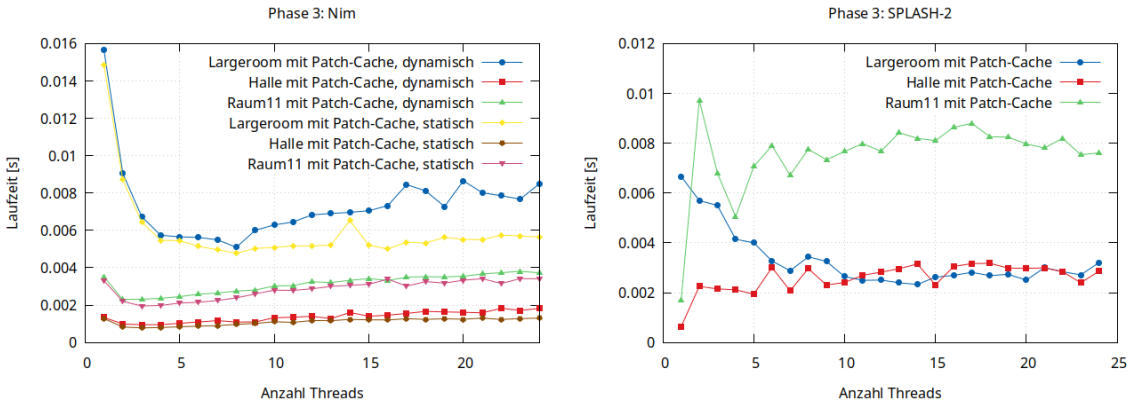


(a) Nim



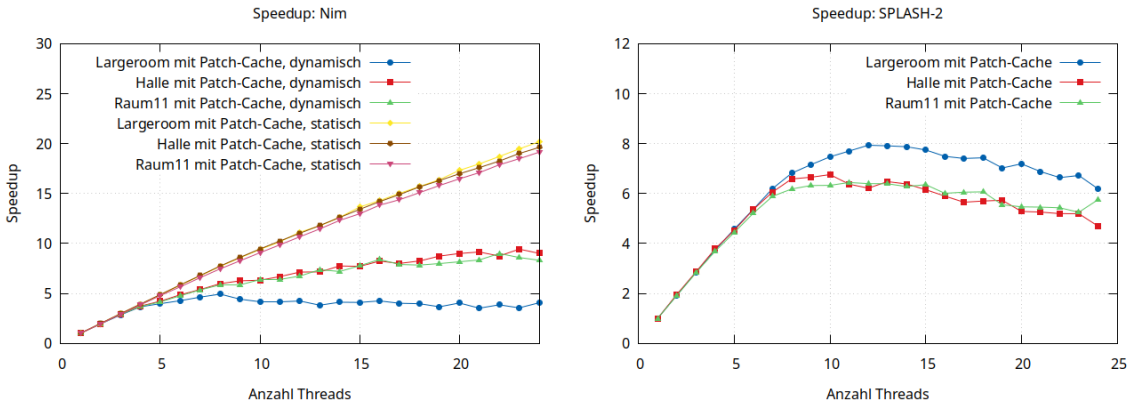
(b) Modifizierte SPLASH-2 Variante

Abb. 5.8: Laufzeiten Phase 2: Vergleich Nim und SPLASH-2 auf x86-64



(a) Nim **(b)** Modifizierte SPLASH-2 Variante

Abb. 5.9: Laufzeiten Phase 3: Vergleich Nim und SPLASH-2 auf x86-64



(a) Nim **(b)** Modifizierte SPLASH-2 Variante

Abb. 5.10: Speedup: Vergleich Nim und SPLASH-2 auf x86-64

5.3.2 Thunder X2 (ARM)

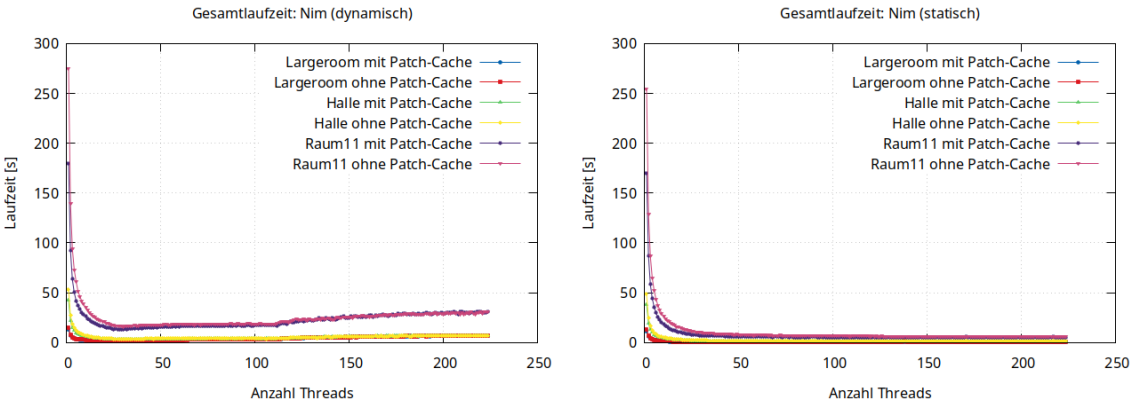
Nachdem nun verschiedene Laufzeitmessungen auf x86-64 durchgeführt worden sind, werden nun abschließend einige Analysen auf der ARM Architektur betrachtet. Hierzu werden die eigenen beiden Varianten bei maximal 224 Threads verglichen.

In Abbildung 5.11 sieht man die Gesamtlaufzeiten der dynamischen und statischen Variante in Abhängigkeit der Threadanzahl. Dabei fällt wieder auf, dass bei allen drei Szenen die Laufzeit der dynamischen Variante über der Laufzeit der statischen Variante liegt, was man durch notwendige Allokation während der Ausführung erklären kann. Weiterhin bemerkt man in beiden Varianten zunächst eine Leistungssteigerung durch Parallelität, deren Effekt jedoch bei der dynamischen Variante ab ungefähr 40 Threads rückläufig ist. Dies konnte man auch schon in Abbildung 5.1 auf der x86-64 Architektur sehen. Da jedoch nun 224 Threads möglich sind, sieht man den deutlichen Anstieg der Laufzeit ab 130 Threads. Daraus kann man folgern, dass der Flaschenhals durch die Allokation während der Laufzeit nun eine zentrale Rolle spielt und das Programm signifikant verlangsamt wird. Diesen Effekt kann man bei der statischen Variante nicht sehen, vielmehr verbessert sich im Großen und Ganzen die Leistung der Applikation kontinuierlich, wodurch auch die Laufzeit immer unter derjenigen der dynamischen Variante liegt.

Die Ergebnisse der Gesamtlaufzeit korrespondieren dabei mit den berechneten Speedups aus Abbildung 5.15. Bei der dynamischen Variante sieht man einen Anstieg des Speedups bis ungefähr 40 Threads, danach fällt dieser rapide ab. Bei der statischen Variante bemerkt man ebenfalls, dass der Speedup bis 130 Threads ansteigt und dabei wesentlich über dem Speedup der dynamischen Variante liegt. Bei allen Szenen außer *Raum11 ohne Patch-Cache* sinkt der Speedup ab 130 jedoch etwas ab, was vermutlich an der zu kleinen Szenengröße liegt. Da die Szene *Raum11 ohne Patch-Cache* mehr Eingabepolygone und durch den fehlenden Patch-Cache ebenfalls aufwendigere Sichtbarkeitstasks enthält, steigt hier der Speedup kontinuierlich an.

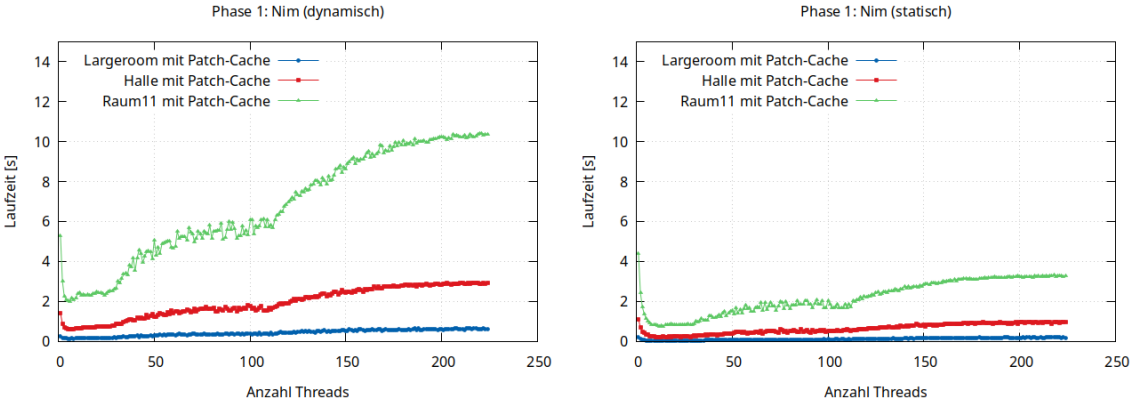
Bei der Betrachtung der Laufzeiten von Phase 1 in Abbildung 5.12 sieht man bei beiden Varianten im Vergleich zur sequentiellen Ausführung eine Verbesserung der Laufzeit, wobei diese jedoch bei Erhöherung der Threadanzahl wieder ansteigt. Bei der dynamischen Variante liegt diese sogar ab ungefähr 100 Threads über der sequentiellen Laufzeit, was bei der statischen Variante nie erreicht wird. Damit korrespondieren die Ergebnisse mit denjenigen aus den Analysen auf der x86-64 Architektur.

Ähnliche Ergebnisse liefern ebenfalls die Diagramme in Abbildung 5.14, wobei diese bei beiden Varianten stark schwanken. Dies liegt vermutlich daran, dass die Größe der Eingabeszene nicht mehr ausreicht, um stabile und gut skalierende Ergebnisse bei hohen Threadanzahlen zu liefern. Gleichzeitig ist der Aufwand der Tasks in Phase drei im Verhältnis zu den Threadanzahlen eher gering.



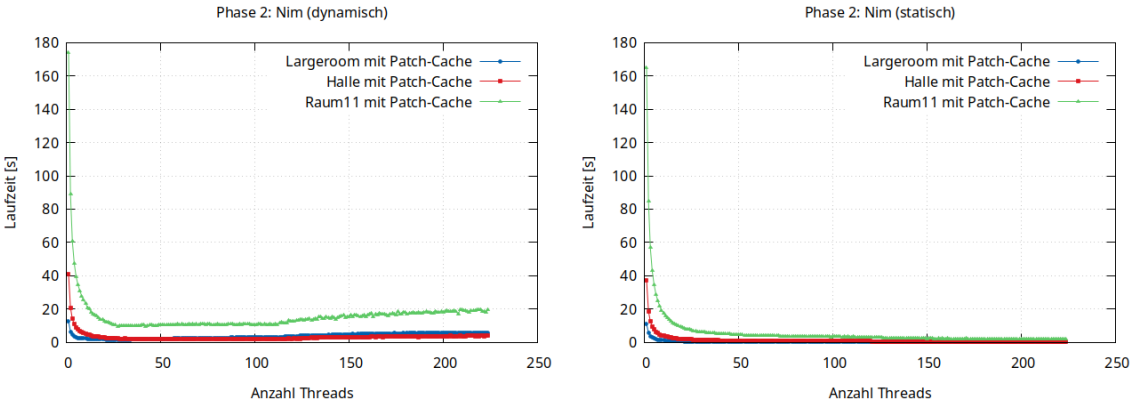
(a) dynamische Variante **(b)** statische Variante

Abb. 5.11: Gesamtlaufzeiten (Nim) auf ARM



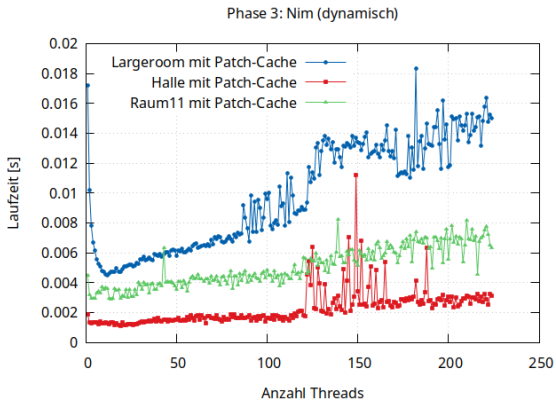
(a) dynamische Variante **(b)** statische Variante

Abb. 5.12: Laufzeiten Phase 1 (Nim) auf ARM

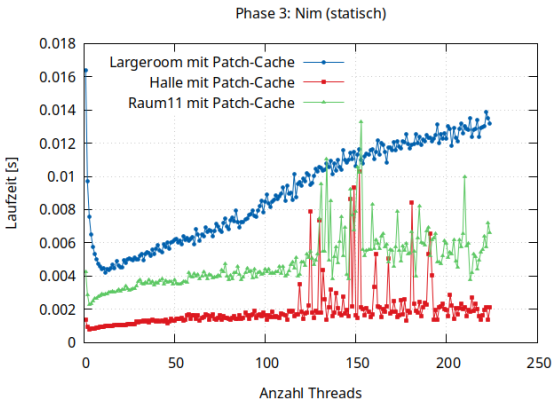


(a) dynamische Variante **(b)** statische Variante

Abb. 5.13: Laufzeiten Phase 2 (Nim) auf ARM

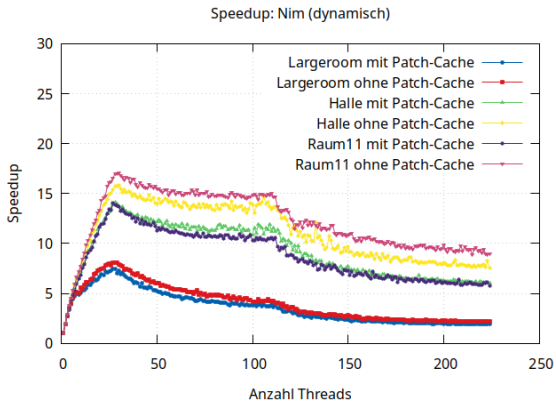


(a) dynamische Variante

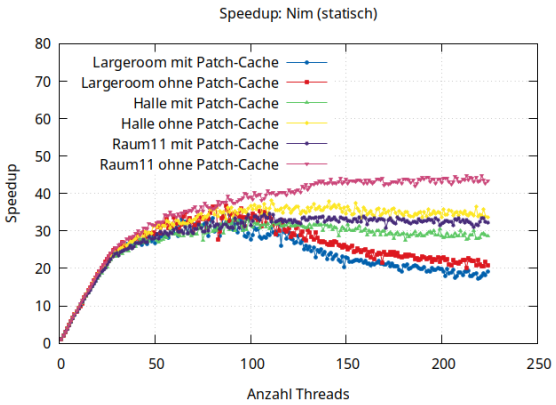


(b) statische Variante

Abb. 5.14: Laufzeiten Phase 3 (Nim) auf ARM



(a) dynamische Variante



(b) statische Variante

Abb. 5.15: Speedups (Nim) auf ARM

6 Fazit und Ausblick

In einem letzten Kapitel sollen nun alle bisherigen Ergebnisse gebündelt und darauf basierend sowohl ein Fazit als auch ein Ausblick gegeben werden.

In einem ersten Schritt sind in dieser Arbeit die theoretischen Grundlagen des klassischen und hierarchischen Radiosity-Verfahrens dargelegt worden. Dabei ist klar geworden, dass dieser Algorithmus die Lichtausbreitung in einer Szene inklusive Mehrfachreflexionen auf Grundlage der Radiosity-Gleichung berechnen kann und daher zu den globalen Beleuchtungsverfahren zählt. Aufgrund seiner irregulären Struktur, bedarf er jedoch einiger Optimierungen, die im hierarchischen Radiosity-Verfahren ihren Einzug gefunden haben. Hierbei wird die statische Struktur des klassischen Verfahrens, bei dem alle Eingabepolygone in eine feste Anzahl von Dreiecken unterteilt werden, aufgebrochen und ein Fehler berechnet, der nun als Grundlage für weitere Unterteilungen dienen soll.

In einem nächsten Kapitel sind zentrale Aspekte der Programmiersprache Nim und der Multithreading-Runtime Weave aufgezeigt worden, wobei danach auf die eigene Implementierung eingegangen worden ist. Anhand verschiedener Algorithmen ist dabei die Realisierung der Parallelität mit Weave vorgestellt worden.

In einem letzten Schritt wurde dann die statische und dynamische Variante der eigenen Implementierung miteinander verglichen, wobei dann in einem nächsten Kapitel die modizierte SPLASH-2 Variante als Referenz diente. Bei der Analyse der eigenen beiden Implementierungen ist aufgezeigt worden, dass die statische Variante in allen Aspekten der dynamischen Variante überlegen ist, da hier keine Allokation während der Laufzeit stattfindet, sondern vielmehr alle benötigen, vorallokierten Objekte aus einer Liste entnommen werden. Insbesondere fällt der Speedup der statischen Variante sehr gut aus, da hier ein kontinuierlicher Anstieg zu verzeichnen ist. Bei dem Vergleich zwischen der modifizierten SPLASH-2 Variante und der eigenen Implementierung ist dann vorgestellt worden, dass die sequentielle Laufzeit der eigenen Implementierung wesentlich schlechter ausfällt und daher auch der Speedup signifikant besser ist als jener der SPLASH-2 Variante. Im Großen und Ganzen ist jedoch festgestellt worden, dass die SPLASH-2 Variante insgesamt bessere Laufzeiten besitzt, die nur mit relativ vielen Threads durch die eigene Implementierung erreicht werden können. Dies lässt darauf schließen, dass die Nutzung von Locks in der eigenen Implementierung vermutlich relativ ineffizient erfolgt.

Als nächster Schritt wäre es denkbar, dass weitere Optimierungen an der statischen Variante vorgenommen werden, um noch kleinere Laufzeiten erzielen zu können. Mithilfe von ausführlichen Profiling-Analysen könnte dann der Flaschenhals in der eigenen Implementierung ausfindig gemacht und so die Laufzeiten verringert werden. Auch dies wurde in der vorliegenden Arbeit bereits angewandt, jedoch konnte die Gesamtlaufzeit nur gering verändert werden. Weiterhin wäre eine Implementierung von Texturen sinnvoll, um Objekten in den Szenen verschiedene Strukturen und unterschiedliches Aussehen zu verleihen. Dies ist in der eigenen Arbeit bisher übersprungen worden.

Digitale Abgabe

Auf der beiliegenden CD befindet sich der komplette Inhalt des verwendeten Repositories. Darunter zählt sowohl die eigene statische und dynamische Implementierung, als auch die modifizierte SPLASH-2 Variante aus [Kor01] zum Vergleich der Laufzeiten. Alle verwendeten Skripts und Diagramme sind ebenfalls vorhanden.

Literaturverzeichnis

[BGZ02] Hans-Joachim Bungartz, Michael Griebel und Christoph Zenger. *Einführung in die Computergraphik - Grundlagen, Geometrische Modellierung, Algorithmen; 2., überarbeitete und erweiterte Auflage*. Juni 2002 (zitiert auf den Seiten 5, 6, 8–10, 28).

[BHW12] K. Burg, H. Haf und F. Wille. *Burg, Höh.Mathe II*. Höhere Mathematik für Ingenieure. Vieweg+Teubner Verlag, 2012 (zitiert auf Seite 20).

[Kor01] Matthias Korch. „Einsatz von Taskpools in Pthreads und Java zur parallelen Implementierung irregulärer Algorithmen“. Diplomarbeit. Martin-Luther-Universität Halle-Wittenberg, 2001 (zitiert auf den Seiten 2, 3, 7, 18, 28, 29, 33, 35–37, 41, 50).

[Nü05] Frederik Nünning. „Radiosity auf der Grafik-Hardware“. Diplomarbeit. Universität Koblenz-Landau, 2005 (zitiert auf den Seiten 2, 6, 7, 10).

[Pic17] Dominik Picheta. *Nim in Action*. USA: Manning Publications Co., 2017 (zitiert auf den Seiten 13–15).

[PRR98] Axel Podehl, Thomas Rauber und Gudula Rünger. „A shared-memory implementation of the hierarchical radiosity method“. In: *Theoretical Computer Science* 196.1 (1998), S. 215–240 (zitiert auf Seite 9).

[RR00] T. Rauber und G. Rünger. *Parallele und verteilte Programmierung*. Springer, 2000 (zitiert auf den Seiten 1–4, 6–8, 10, 24, 26–29, 31).

[Sch10] R.H. Schulz. *Repetitorium Bachelor Mathematik: Zur Vorbereitung auf Modulprüfungen in der mathematischen Grundausbildung*. Vieweg Studium. Vieweg+Teubner Verlag, 2010 (zitiert auf Seite 20).

[Sin+95] J.P. Singh, C. Holt, T. Totsuka, A. Gupta und J. Hennessy. „Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity“. In: *Journal of Parallel and Distributed Computing* 27.2 (1995), S. 118–141 (zitiert auf den Seiten 11, 24–26, 33).

[SM09] Peter Shirley und Steve Marschner. *Fundamentals of Computer Graphics*. 3rd. USA: A. K. Peters, Ltd., 2009 (zitiert auf Seite 1).

[Woo+95] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh und A. Gupta. „The SPLASH-2 programs: characterization and methodological considerations“. In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. 1995, S. 24–36 (zitiert auf den Seiten 2, 17–19, 21, 22, 24, 25, 28, 29, 31, 33–35).

Abbildungsverzeichnis

2.1 Beispielszene mit und ohne Einsatz des Radiosity-Verfahrens (vgl. [Nü05, S. 3]) 6
2.2 Illustration der Radiosity-Gleichung (vgl. [RR00, S. 521]) 8
2.3 Visualisierung der Winkel θ_i und θ_j (vgl. [BGZ02, S. 185]) 9
2.4 Illustration des klassischen Radiosity-Algorithmus (vgl. [Nü05, S. 15]) 10
2.5 Illustration des BSP-Baums und der Quadtrees (vgl. [Sin+95, S. 11]) . 11

3.1 Kompilizierungsprozess mit Transformation in C-Code (vgl. [Pic17, S. 11]) 14

4.1 Illustration der Schnittsituationen 22
4.2 Schema der Sichtbarkeitsberechnung 25
4.3 Sichtbarkeitsberechnung anhand des BSP-Baums (vgl. [Sin+95, S. 37ff]) 26

5.1 Gesamtlaufzeiten (Nim) auf x86-64 40
5.2 Laufzeiten Phase 1 (Nim) auf x86-64 40
5.3 Laufzeiten Phase 2 (Nim) auf x86-64 40
5.4 Laufzeiten Phase 3 (Nim) auf x86-64 41
5.5 Speedups (Nim) auf x86-64 41
5.6 Gesamtlaufzeiten: Vergleich Nim und SPLASH-2 auf x86-64 43
5.7 Laufzeiten Phase 1: Vergleich Nim und SPLASH-2 auf x86-64 43
5.8 Laufzeiten Phase 2: Vergleich Nim und SPLASH-2 auf x86-64 43
5.9 Laufzeiten Phase 3: Vergleich Nim und SPLASH-2 auf x86-64 44
5.10 Speedup: Vergleich Nim und SPLASH-2 auf x86-64 44
5.11 Gesamtlaufzeiten (Nim) auf ARM 46
5.12 Laufzeiten Phase 1 (Nim) auf ARM 46
5.13 Laufzeiten Phase 2 (Nim) auf ARM 46
5.14 Laufzeiten Phase 3 (Nim) auf ARM 47
5.15 Speedups (Nim) auf ARM 47

Tabellenverzeichnis

4.1 Mögliche Aufrufparameter beim Start des Programms 19

Selbstständigkeitserklärung

Hiermit versichere ich, Fabian Vießmann, dass ich die vorliegende Arbeit und Implementierung selbstständig verfasst und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe. Des Weiteren versichere ich, dass diese Arbeit nicht bereits zur Erlangung des akademischen Grades eingereicht wurde.

Bayreuth, den 10. Juni 2021

Fabian Vießmann