Karlsruhe Institute of Technology

Adaptives Online-Tuning für kontinuierliche Zustandsräume

Masterarbeit
von

Timo Kopf

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	Dipl.-Inform. Philip Pfaffe
Zweiter betr. Mitarbeiter:	

Bearbeitungszeit: 01.06.2018 – 30.11.2018

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 28.11.2018

(**Timo Kopf**)

Publikationsgenehmigung

Melder der Publikation

Hildegard Sauer

Institut für Programmstrukturen und Datenorganisation (IPD)
Lehrstuhl für Programmiersysteme
Leiter Prof. Dr. Walter F. Tichy

+49 721 608-43934
hildegard.sauer@kit.edu

Erklärung des Verfassers

Ich räume dem Karlsruher Institut für Technologie (KIT) dauerhaft ein einfaches Nutzungsrecht für die Bereitstellung einer elektronischen Fassung meiner Publikation auf dem zentralen Dokumentenserver des KIT ein.

Ich bin Inhaber aller Rechte an dem Werk; Ansprüche Dritter sind davon nicht berührt.

Bei etwaigen Forderungen Dritter stelle ich das KIT frei.

Eventuelle Mitautoren sind mit diesen Regelungen einverstanden.

Der Betreuer der Arbeit ist mit der Veröffentlichung einverstanden.

Art der Abschlussarbeit:	Masterarbeit
Titel:	Adaptives Online-Tuning für kontinuierliche Zustandsräume
Datum:	28.11.2018
Name:	Timo Kopf

Karlsruhe, 28.11.2018

(**Timo Kopf**)

Kurzfassung

Raytracing ist ein rechenintensives Verfahren zur Erzeugung photorealistischer Bilder. Durch die automatische Optimierung von Parametern, die Einfluss auf die Rechenzeit haben, kann die Erzeugung von Bildern beschleunigt werden. Im Rahmen der vorliegenden Arbeit wurde der Auto-Tuner *libtuning* um ein generalisiertes Reinforcement Learning-Verfahren erweitert, das in der Lage ist, bestimmte Charakteristika der zu zeichnenden Frames bei der Auswahl geeigneter Parameterkonfigurationen zu berücksichtigen. Die hierfür eingesetzte Strategie ist eine ϵ-gierige Strategie, die für die Exploration das Nelder-Mead-Verfahren zur Funktionsminimierung aus *libtuning* verwendet. Es konnte gezeigt werden, dass ein Beschleunigung von bis zu 7,7% in Bezug auf die gesamte Rechenzeit eines Raytracing-Anwendungsszenarios dieser Implementierung gegenüber der Verwendung von *libtuning* erzielt werden konnte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In der vorliegenden Arbeit wird ein Hybridansatz aus klassischem Auto-Tuning und Reinforcement Learning vorgestellt, mit dem es möglich ist, eine automatische Parameteroptimierung zur Reduzierung der Laufzeit von Raytracing-Anwendungen durchzuführen.

1.1 Motivation

Computergenerierte Effekte sind aus Hollywood-Filmen und -Serien seit Jahren nicht mehr wegzudenken. Ganze Filmszenerien, Landschaften und Gebäude werden heutzutage immer öfter komplett digital am Computer erzeugt. Und auch komplett computeranimierte Filme sind seit dem ersten Toy Story-Film aus dem Jahr 1995 ein alltäglicher Gast in den Kinos. Doch auch Firmen wie zum Beispiel BMW nutzen am Computer generierte Bilder ihrer Fahrzeuge für den Druck von Prospekten anstelle von echten Fotografien. Der Grund hierfür ist der, dass in der heutigen Zeit Bilder am Computer photorealistisch erzeugt werden können, sodass diese auf einem Hochglanzdruck besser aussehen als Fotografien. Das Verfahren, was in diesen Anwendungsfällen der Computergrafik zum Einsatz kommt, heißt *Raytracing*. Dieses Verfahren erlaubt es, den Licht- und Schattenfall physikalisch korrekt zu simulieren und auf diese Weise Bilder zu erzeugen, die nur noch schwer mit dem bloßen Auge von einer Fotografie zu unterscheiden sind.

Das Grundprinzip von Raytracing besteht in dem Nachverfolgen von virtuellen Lichtstrahlen, die mit der Geometrie der virtuellen dreidimensionalen Szene, aus der ein Bildausschnitt gezeichnet werden soll, interagieren. Diese Lichtstrahlen treffen auf spiegelnde oder diffuse Oberflächen und werden von dort aus weiter reflektiert, bis sie irgendwann in die virtuelle Kameralinse fallen. Durch ein Zurückverfolgen dieser Lichtstrahlen, ausgehend von der virtuellen Kamera, über sämtliche Reflexionen an der Szenengeometrie hinweg, bis zum Erreichen einer virtuellen Lichtquelle wie einer Glühbirne oder einem virtuellen Himmel, werden die einzelnen Bildpunkte berechnet. Dies lässt erahnen, dass Raytracing einen entscheidenden Nachteil hat: Es ist ein sehr komplexes Verfahren, das sehr viel Rechenzeit in Anspruch nimmt. Aus diesem Grund wird Raytracing derzeit nur selten oder nur in beschränktem Umfang in Echtzeitanwendungen wie Videospielen eingesetzt. Doch Echtzeit-Raytracing wurde nie ganz aufgegeben und mit Computer-Hardware, die immer leistungsfähiger wird und einer fortschreitenden Forschung in diesem Themengebiet ist dieses Anwendungsgebiet von Raytracing nicht mehr in weiter Ferne. Um einen virtuellen Lichtstrahl zu verfolgen und zu berücksichtigen, wenn er auf eine Geometrieoberfläche trifft, muss für jeden Lichtstrahl überprüft werden, welche Geometrie dieser

Lichtstrahl schneidet. Aus der Menge der geschnittenen Geometrien muss dann das Objekt ausgewählt werden, das am nächsten zum Ursprung des Lichtstrahls ist, der gerade verfolgt wird. So ist ein beliebter Ansatz, die Rechenzeit von Raytracing zu reduzieren, der Einsatz einer räumlichen Datenstruktur, mit der die virtuelle Szene in einer baumartigen Struktur aufgeteilt wird. Durch den Einsatz einer solchen räumlichen Datenstruktur und geschickter Wahl der Trennebenen, mit der die Szene in ihre Teilbereiche unterteilt wird, kann die Anzahl dieser Schnitttests deutlich reduziert werden. Doch die Wahl für eine optimale Raumseparation ist nicht trivial und es existiert (derzeit) keine allgemeingültige Lösung für dieses Problem. Eine gute Wahl für die Parameter, die den Aufbau einer solchen räumlichen Datenstruktur beeinflussen, ist von vielen Faktoren abhängig, nicht zuletzt von der virtuellen Szene oder dem Computer, der das Raytracing durchführen soll. Für dieses Problem ist also eine Lösung notwendig, die automatisch diese Parameter so wählt, dass bei der Erstellung einer solchen räumlichen Datenstruktur für eine bestimmte virtuelle Szene eine Raumseparation dieser Szene entsteht, mit der die Anzahl der Schnitttests der Lichtstrahlen mit der Szenengeometrie möglichst gering sein wird und somit die Rechenzeit für das Raytracing möglichst klein wird. Der Ansatz, mit dem automatisch gute Werte für solche Parameter gefunden werden, heißt *Auto-Tuning*.

Doch konventionelles Online Auto-Tuning, also Auto-Tuning, das während des eigentlichen Produktiveinsatzes die Optimierung der Parameter vornimmt, hat speziell im Anwendungsgebiet von Raytracing den entscheidenden Nachteil, dass die Suche nach einer optimalen Parameterkonfiguration mit jedem Bild, das gezeichnet werden soll, neu gestartet werden muss, da die Wahl einer bestimmten Konfiguration von bestimmten Charakteristika des Bildes abhängt.

Im Rahmen dieser Arbeit wurde ein Verfahren entwickelt, das genau dieses Problem mithilfe von einem hybriden Ansatz aus Auto-Tuning und Maschinellem Lernen lösen soll. Neben einer ausführlichen Vorstellung dieses Verfahrens wird in dieser Arbeit auch untersucht, wie gut sich damit die Rechenzeit von Raytracing verkürzen lässt.

1.2 Ziel dieser Arbeit

Zum gegenwärtigen Zeitpunkt befindet sich der Auto-Tuner *libtuning* am Institut für Programmstrukturen und Datenorganisation (IPD) in Entwicklung. Dieser implementiert den Algorithmus von Nelder und Mead zur Parameteroptimierung. Mithilfe dieses Auto-Tuners hat Kevin Zerr in seiner Masterarbeit „Laufzeitoptimierung mittels Autotuning von Path-Tracing-Datenstrukturen“ bereits gezeigt, dass mithilfe von *libtuning* der Aufbau von räumlichen Datenstrukturen, sogenannten BVHs, automatisch zur Laufzeit derart optimiert werden kann, dass die Rechenzeit für das Raytracing signifikant reduziert wird [Zer17].

In einer anderern Masterarbeit wurde *libtuning* bereits um eine Variante erweitert, die *Reinforcement Learning*, ein Verfahren aus dem Bereich des Maschinellen Lernens, verwendet, um die automatische Laufzeitoptimierung in „Abhängigkeit des Anwendungs- also auch des Systemkontexts“ vorzunehmen [Wen16].

Das Ziel der vorliegenden Arbeit ist es, den Auto-Tuner *libtuning* um eine weitere Variante zu erweitern, die ebenfalls Reinforcement Learning einsetzt. Doch im Unterschied zu der Arbeit von Angré Wengert soll diese Implementierung von Reinforcement Learning in der Lage sein, auf kontinuierlichen Zustandsräumen zu operieren und somit für die Laufzeitoptimierung durch Parameteroptimierung des Aufbaus der räumlichen Datenstruktur verwendbar sein. Durch eine Unterstützung von kontinuierlichen Zustandsräumen soll der Auto-Tuner in die Lage versetzt werden, in Abhängigkeit von bestimmten Charakteristika der zu verarbeitenden Daten solche Parameterkonfigurationen auszuwählen, die für diese

Eingabedaten optimal in Bezug auf die Rechenzeit, die für die Verarbeitung des jeweiligen Eingabedatums benötigt wird, ist. Wie gut diese Implementierung im Vergleich zu *libtuning* in einem Raytracing-Anwendungskontext abschneidet, wird in der Evaluation dieser Arbeit gezeigt.

1.3 Struktur dieser Arbeit

In dem folgenden Kapitel werden zunächst die Grundlagen der Bildsynthese mittels Raytracing beschrieben. Hierbei wird auf die zentrale Gleichung in der Computergrafik, die Rendering-Gleichung, eingegangen und darauf, wie eine Lösung mithilfe der Monte-Carlo-Integration approximiert werden kann. Außerdem werden zwei beliebte räumliche Datenstrukturen vorgestellt, die zur Reduzierung der Anzahl Schnitttests der Lichtstrahlen mit der Geometrie eingesetzt werden. Des Weiteren werden sowohl die Grundlagen von Reinforcement Learning, mit Schwerpunkt Q-Learning, als auch die von Auto-Tuning beschrieben. Im letzten Abschnitt der Grundlagen wird auch auf die Auto-Tuning-Bibliothek *libtuning* eingegangen, die die Grundlage für die vorliegende Arbeit darstellt.

In Kapitel 3 werden verwandte Arbeiten aus den Bereichen Reinforcement Learning und Auto-Tuning, sowie Beschleunigung von Raytracing mittels Reinforcement Learning und Auto-Tuning vorgestellt.

In den Kapiteln 4 und 5 wird zunächst der Entwurf und anschließend die Implementierung dieser Arbeit vorgestellt. In diesen Kapiteln wird beschrieben, auf welche Weise die Auto-Tuning-Bibliothek *libtuning* um das Reinforcement Learning-Verfahren Q-Learning erweitert und wie die Unterstützung für kontinuierliche Zustandsräume umgesetzt wurde.

In Kapitel 6 werden die Experimente vorgestellt, die zur Evaluation des in dieser Arbeit entwickelten Auto-Tuners verwendet wurden und die Ergebnisse diskutiert.

Das letzte Kapitel gibt einen Rückblick auf diese Arbeit und fasst die aus der Evaluation gewonnenen Erkenntnisse noch einmal zusammen. Des Weiteren wird ein Ausblick auf mögliche weiterführende Fragestellungen, die diese Arbeit aufgeworfen hat, gegeben.

2. Grundlagen

Dieses Kapitel enthält die Grundlagen der Konzepte und Ideen, die in dieser Arbeit verwendet werden. Es wird auf Verfahren der photorealistischen Bildsynthese, Reinforcement Learning, ein Ansatz aus dem Bereich des Maschinellen Lernens, sowie auf die automatische Optimierung von Parametern eingegangen.

2.1 Raytracing

Raytracing ist ein Verfahren zum computergestützten Erzeugen von Bildern. Hierbei handelt es sich um einen Ansatz, der physikalisch korrekten Licht- und Schattenfall berücksichtigt und so in der Lage ist, photorealistische Bilder zu synthetisieren. Dies geschieht, wie der Name Raytracing schon andeutet, durch das Nachverfolgen von Strahlen, sogenannten Sichtstrahlen. Sie repräsentieren gedachte Photonenstrahlen, die durch die dreidimensionale Szene, die es zu zeichnen gilt, geschossen werden. Diese Strahlen werden durch den Raytracing-Algorithmus nachverfolgt und die Schnittpunkte mit der in der Szene enthaltenen Geometrie notiert. Diese Schnittpunkte werden zur Berechnung der Farbe jedes einzelnen Bildpunktes des daraus resultierenden Bildes herangezogen. Eine 3D-Szene besteht aus geometrischen Objekten, die oft aus Gründen der Einfachheit aus vielen Dreiecken zusammengesetzt sind, und Lichtquellen.

2.1.1 Grundlegendes Verfahren zur Bildsynthese mit Raytracing

In diesem Abschnitt wird das grundlegende Verfahren erklärt, das bei Raytracing zum Einsatz kommt, um ein Bild aus einer 3D-Szene zu erzeugen. Dieser Prozess wird *Rendering* genannt. Anders als in der echten Welt haben die Photonenstrahlen bei Raytracing ihren Ursprung nicht in den Lichtquellen, sondern im Mittelpunkt einer virtuellen Kamera. Das Koordinatensystem der Kamera lässt sich durch zwei dreidimensionale Vektoren p und $view$ beschreiben. p ist dabei der Stützvektor des orthonormalen Koordinatensystems, das durch die drei orthonormal zueinander stehenden Vektoren u, v und w aufgespannt wird. Dieses Koordinatensystem gibt die Orientierung der Kamera an. Abbildung 2.1 stellt das Koordinatensystem der Kamera grafisch dar. Aus dem $view$-Vektor der Kamera und dem up-Vektor, der in der zu rendernden Szene vertikal nach oben zeigt, lassen sich u, v und w mithilfe des Kreuzprodukts berechnen. u ist hierbei der Vektor, der aus Sicht der Kamera nach rechts zeigt. w ist das Inverse des $view$-Vektors. u ist also das Kreuzprodukt aus $up \times view$. Der Vektor v lässt sich nun aus dem Kreuzprodukt von u und $view$ berechnen [Shi09].

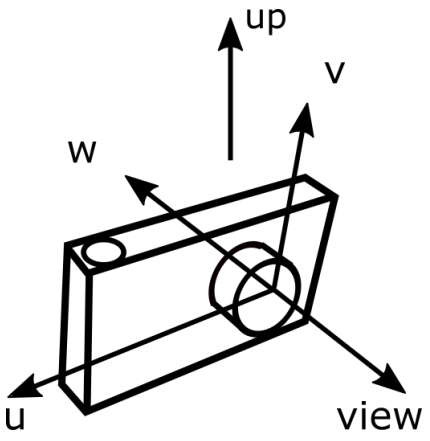

Abbildung 2.1: Grafische Darstellung des Kamerakoordinatensystems. Die Grafik ist adaptiert übernomen aus [Shi09]

Die Bildebene, auf die die 3D-Szene im Zuge des Raytracing-Vorgangs projiziert wird, schwebt zentral in einem Abstand d in Richtung $-w$ vor der Kamera. Ihre Ebene wird durch die Vektoren u und v aufgespannt. Die Bildebene hat eine Höhe und Breite, angegeben in Bildpunkten. Die Kamera schießt nun Sichtstrahlen durch jeden Bildpunkt der Bildebene in die Szene. Diese von der Kamera initial verschossenen Sichtstrahlen nennt man Primärstrahlen. Abbildung 2.2 illustriert den Vorgang des Verschießens der Primärstrahlen. Der Primärstrahl schneidet das blaue Dreieck. Am Schnittpunkt wird ein Sekundärstrahl erzeugt, der in Richtung der einzigen Lichtquelle, der Sonne, gerichtet ist. Um nun die Farbe für das Pixel, durch das der Primärstrahl geschossen wurde, zu berechnen, muss die Farbe der Oberfläche des geschnittenen Dreiecks ausgewertet werden. Dies geschieht in diesem Fall, indem der Sekundärstrahl in Richtung der Sonne weiterverfolgt wird. Da die Sonne eine Lichtquelle ist, endet hier die Weiterverfolgung. Es muss kein weiterer Sekundärstrahl mit Ursprung des Schnittpunkts des ersten Sekundärstrahls mit der Sonne verfolgt werden. Die Farbe der Oberfläche des Dreiecks am Schnittpunkt mit dem Primärstrahl ist also eine Kombination aus der Farbe und Helligkeit der Lichtquelle und der Farbe und Oberflächenbeschaffenheit des Dreiecks an diesem Punkt.

Beim Raytracing wird also jedes Pixel der Bildebene der Reihe nach abgetastet. Da zwischen den verschossenen Primärstrahlen keine Abhängigkeiten bestehen, lässt sich Raytracing auf naïve Weise zum Beispiel über die Zeilen oder Spalten des Bildes parallelisieren.

2.1.2 Die Rendering-Gleichung

In der Realität werden Oberflächen nicht nur direkt von Lichtquellen beleuchtet. Oft ist es sogar so, dass Oberflächen gar keine direkte Sicht auf eine Lichtquelle haben. Sie werden stattdessen indirekt von anderen Oberflächen, die oft selbst auch nur indirekt beleuchtet werden, angestrahlt. Das Licht einer Lichtquelle wird also von vielen Oberflächen reflektiert und zwischen ihnen hin und her geworfen, bevor es schlussendlich das Auge eines Betrachters beziehungsweise die Kameralinse erreicht. Oberflächen, die Licht reflektieren, beleuchten sich also auch gegenseitig. Dies wird das *global illumination problem* genannt [Shi09]. Eine formalisierte Problembeschreibung in Form einer Gleichung wurde zur selben Zeit von James T. Kajiya [Kaj86] und Immel et al. [ICG86] im Jahr 1986 beschrieben. Diese Gleichung ist bekannt als *Rendering-Gleichung*:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega^+} f_r(\omega_i, x, \omega_o) L(x, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i \tag{2.1}$$

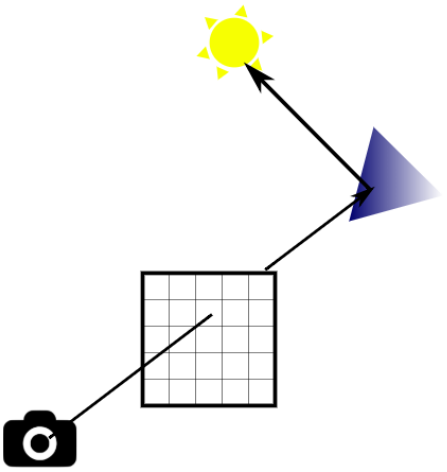

Abbildung 2.2: Die Kamera (unten links) verschießt einen Primärstrahl durch ein Pixel mit Koordinaten $(3, 2)$ der Bildebene, der auf ein Stück Geometrie trifft. Ein Sekundärstrahl entspringt dem Schnittpunkt in Richtung der einzigen Lichtquelle.

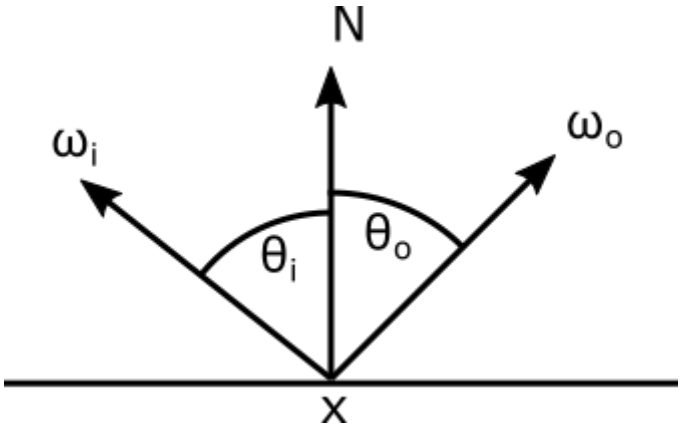

Abbildung 2.3: Lichtein- und Ausstrahlung auf einer Oberfläche.

Die Rendering-Gleichung beschreibt die Intensität der Lichtstrahlung von einem Oberflächenpunkt x in Richtung des Vektors ω. Die Definition der Rendering-Gleichung lässt sich aus der Definition der Bidirektionalen Reflexionsverteilungsfunktion, kurz BRDF (engl. bidirectional reflection distribution function), herleiten. Die BRDF ist eine Proportionalitätskonstante, die für eine bestimmte Oberfläche das Verhältnis von einfallendem zu ausfallendem Licht in Abhängigkeit von Einfalls- und Betrachtungswinkel angibt.

$$f_r(\omega_i, x, \omega_o) = \frac{\mathrm{d}L_r(x, \omega_o)}{\mathrm{d}E_i(x, \omega_i)} = \frac{\mathrm{d}L_r(x, \omega_o)}{L_i(x, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i}$$

$L_r(x, \omega_o)$ beschreibt die Strahldichte des in Richtung ω_o reflektierten Lichts. Die Einheit der Strahldichte ist Watt. $E_i(x, \omega_i)$ beschreibt die auf Punkt x aus Richtung ω_i einfallende Bestrahlungsstärke. Die Einheit der Bestrahlungsstärke ist $Watt \cdot m^{-2} \cdot \mathrm{sr}^{-1}$. Aus den Einheiten lässt sich der Zusammenhang der beiden Größen Strahldichte und Bestrahlungsstärke herleiten. Er ist:

$$L = \frac{\mathrm{d}E}{\mathrm{d}\omega}.$$

Da die Strahldichte die Strahlungsleistung pro Fläche mal Steradiant angibt, muss der

Term noch mit $\cos\theta_i$ multipliziert werden, um die Flächenreduzierung durch die Projektion der Fläche des einfallenden Lichts auf eine Oberfläche mit Winkel θ_i zwischen Einfallsrichtung und Oberflächennormale zu berücksichtigen. Für eine gegebene differentiale Einfallsrichtung $\mathrm{d}\omega_i$ lässt sich die von Oberflächenpunkt x aus in Richtung ω_o reflektierte differentiale Lichtstrahlung berechnen mit [Shi09]:

$$\mathrm{d}L_r(x,\omega_o) = f_r(\omega_i,x,\omega_o)L_i(x,\omega_i)\cos\theta_i\,\mathrm{d}\omega_i$$

Die gesamte Lichtstrahlung, die von Oberflächenpunkt x aus in Richtung ω_o reflektiert wird, ist demnach das Integral über die Oberfläche der positiven Hemisphäre Ω^+:

$$L_r(x,\omega_o) = \int_{\Omega^+} f_r(\omega_i,x,\omega_o)L_i(x,\omega_i)\cos\theta_i\,\mathrm{d}\omega_i$$

Die Gleichung 2.1 ergibt sich also aus der Summe der an dieser Stelle x in Richtung ω_o reflektierten Lichtstrahlung $L_r(x,\omega_o)$ und der Lichtstrahlung, die von der Oberfläche an Punkt x selbst in Richtung ω_o emittiert wird. Abbildung 2.3 zeigt, wie die einzelnen Vektoren und Winkel zueinander im Raum stehen. N ist der Normalenvektor der Oberfläche und entspringt dem Punkt x. ω_i beschreibt die inverse Einstrahlrichtung und ω_o analog dazu die Reflexionsrichtung. Die Winkel θ_i und θ_o sind die Winkel zwischen N und ω_i beziehungsweise ω_o.

2.1.3 Monte-Carlo-Integration

Die Monte-Carlo-Integration ist ein Verfahren zur Approximation von Integralen. Viele Verfahren aus dem Raytracing, die die Rendering-Gleichung lösen, basieren auf diesem Verfahren. Ein Integral I lässt sich wie folgt als Limes einer Summe schreiben:

$$I = \int_{x\in S} f(x)\,\mathrm{d}x = \lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \quad ,$$

wobei x_i eine mit Wahrscheinlichkeit $p(x_i)$ zufällig gezogene Stichprobe ist. Die Dichtefunktion p kann hierbei beliebig gewählt werden. Anzumerken ist, dass eine gute Wahl für p dafür sorgen kann, dass die Approximation des Integrals schneller zu dem Wert des Integrals konvergieren kann [Shi09].

Das Integral $L_r(x,\omega_o)$ ist demnach gleich dem Grenzwert der Summenfunktion multipliziert mit der Oberfläche der positiven Einheitshalbkugel Ω^+ geteilt durch N für $N\to\infty$:

$$L_r(x,\omega_o) = \lim_{N\to\infty}\frac{2\pi}{N}\cdot\sum_{i=1}^{N} f_r(\omega_i,x,\omega_o)\cdot L_i(x,\omega_i)\,\cos\theta_i \tag{2.2}$$

Eine Approximation von L_r kann numerisch durch die Berechnung der Summe mit einem festen $N<\infty$ erfolgen.

Dies wird in Abbildung 2.4 vereinfacht zweidimensional dargestellt. Zu sehen sind zwei Geometrieoberflächen in einer Szene, dargestellt durch die beiden Rechtecke. Gesucht ist $L_r(x,\omega_o)$, also all das Licht, das von Oberflächenpunkt x aus in Richtung ω_o reflektiert wird.. Um L_r zu approximieren, müssen die L_i aus den Richtungen ω_i mit der BRDF multipliziert aufsummiert werden. Die ω_i sind in der Abbildung diejenigen Pfeile, die von der rechten Oberfläche ausgehen. Einer dieser Pfeile trifft in seiner Verlängerung einen anderen Oberflächenpunkt in der Szene. Um für die Einfallsrichtung, deren Inverse die Richtung dieses Pfeils ist, L_i zu berechnen, muss die Lichtstrahlung an dem Oberflächenpunkt y ausgewertet werden. Die passiert analog zu der Auswertung von $L_r(x,\omega_o)$. Ist die Farbe an Oberflächenpunkt y ausgewertet worden, so kann auch der Strahl, der von x nach y zeigt, zurückverfolgt werden und wird Teil der Summe zur Berechnung der Farbe an Punkt x.

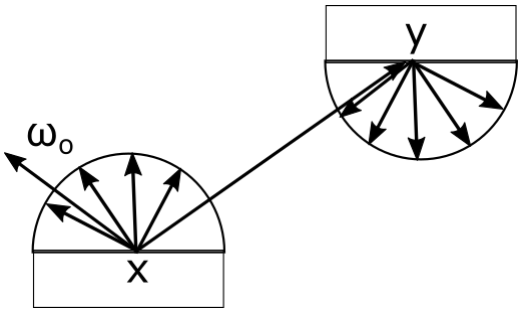

Abbildung 2.4: zweidimensionale Darstellung des Monte-Carlo-Verfahrens zur Berechnung des in Richtung ω_o reflektierten Lichts. Die Grafik ist adaptiert übernommen worden von [Zer17]

2.1.4 Pathtracing

Pathtracing ist ein Verfahren zur Lösung der Rendering-Gleichung. Es verwendet den im letzten Abschnitt angesprochenen Monte-Carlo-Ansatz. Es ist das von der Raytracing-Bibliothek Intel Embree eingesetzte Verfahren, die in dieser Arbeit verwendet wird. Bei Pathtracing werden ganze Pfade, deren Segmente Sichtstrahlen sind, verfolgt. Für jeden Primärstrahl, der ausgehend von der Kamera durch ein Pixel geschossen wird, und der eine Oberfläche an einem Punkt x schneidet, wird für diesen Punkt x zufällig ein Sekundärstrahl ausgewählt, der ausgehend von x weiter verfolgt wird und der dann unter Umständen eine weitere Oberfläche schneidet. Dies wird solange wiederholt, bis das Abbruchkriterium für dieses rekursive Pfadverfolgen erfüllt ist. Es findet mit dem rekursiven Aufstieg die Akkumulation der Farbwerte der einzelnen Schnittpunkte des Pfades statt. Diese Akkumulation der Farbwerte ergibt dann den n-ten Anteil der Farbe eines Samples eines Pixels des Ausgabebildes, das mit n Samples pro Pixel gerendert wird. Als Abbruchkriterium für den rekursiven Abstieg wird zum Beispiel *Russisch-Roulette* verwendet. Hierbei wird bei jeder Rekursion eine Zufallszahl gezogen. Ist diese Zufallszahl größer als ein vorher festgelegter Schwellwert, so wird die Rekursion beendet.
Die Primärstrahlen, die von der Kamera aus durch die Pixel geschossen werden, werden dabei nicht zwangsweise durch den Pixelmittelpunkt geschossen, sondern der Schnittpunkt des Primärstrahls mit der Bildebene wird pro Pixel zufällig in den Grenzen dieses Pixels gewählt.

2.1.5 Räumliche Datenstrukturen

Räumliche Datenstrukturen dienen im Raytracing der Beschleunigung des Rendering-Vorgangs. Für jeden Primär- und Sekundärstrahl, der verschossen wird, muss überprüft werden, welches Geometrieobjekt in der Szene von diesem Strahl geschnitten wird und die kürzeste Distanz zum Ursprung des Sichtstrahls hat. Der Einsatz räumlicher Datenstrukturen hat das Ziel, die Anzahl an Schnitttests mit der Geometrie der Szene zu reduzieren. In diesem Abschnitt werden zwei populäre räumliche Datenstrukturen vorgestellt.

2.1.5.1 kD-Bäume

Ein kD-Baum ist eine Verallgemeinerung eines 2D-Baums, der k-dimensionale Punkte in einer Baumstruktur speichert. Während ein 2D-Baum die Ebene durch das Platzieren von Geraden sukzessive unterteilt, unterteilt ein kD-Baum den k-dimensionalen Raum mit Hyperebenen. Der Raum besteht nach der Teilung durch die Hyperebene aus zwei Teilräumen. Die Wurzel der Baumstruktur speichert den gesamten Raum, die beiden Kindknoten jeweils einen der beiden durch die Hyperebene geteilten Teilräume. Rekursiv wird nun jeweils ein solcher Teilraum weiter durch eine neue Hyperebene separiert, die wieder zwei

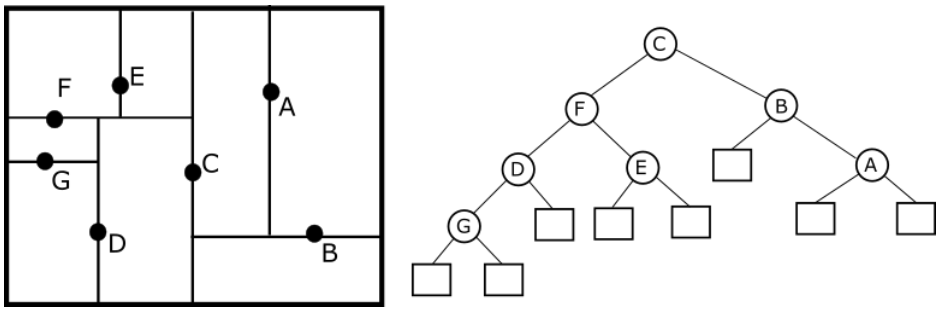

Abbildung 2.5: Unterteilung eine Fläche mithilfe eines 2D-Baums. Abbildung wurde adaptiert übernommen aus [Ben75]

Teilräume des ersten Teilraums entstehen lässt. Diese werden wiederum in Kindknoten gespeichert und die Unterteilung fährt weiter fort. Abbildung 2.5 zeigt dieses Verfahren beispielhaft an einem 2D-Baum. Der 2D-Baum aus der Abbildung setzt die Trennlinien auf die Mittelpunkte der Punkte. An welchen Punkten eine Trennlinie bzw. Trennhyperebene angesetzt wird, hat offensichtlich einen großen Einfluss darauf, wie schnell der Baum zu einem späteren Zeitpunkt traversiert werden kann.

Zur Bestimmung der Position einer Hyperebene gibt es mehrere Ansätze. Zum einen gibt es den Ansatz des räumlichen Mittels (engl. spatial median). Hierbei wird im *Round Robin*-Verfahren die Dimension ausgewählt, die durch die Hyperebene unterteilt werden soll. Die Hyperebene wird dann in die räumliche Mitte entlang dieser Dimension gesetzt. Die Unterteilung wird in der Regel solange fortgesetzt, bis eine Mindestanzahl an Objekten (im Fall von Abbildung 2.5 sind es Punkte, im Fall von Raytracing handelt es sich in der Regel um Dreiecke) pro Teilraum unterschritten wurde, oder bis der Baum eine Maximalhöhe erreicht hat [WH06].
Ein anderer, weitaus ausgeklügelterer Ansatz ist der der Surface Area Heuristic. Die SAH schätzt für eine mögliche Unterteilung eines Voxels V mit Ebene p wie hoch die erwarteten Kosten für die Traversierung ausfallen werden und schlägt so eine Unterteilung von V vor, für die die geschätzten erwarteten Kosten am geringsten sind. Hierzu zieht sie die resultierenden Kind-Voxel sowie die Anzahl der Primitive, die in diesen Kind-Voxeln enthalten sein werden, in Betracht [WH06].

2.1.5.2 Bounding Volume Hierarchies

Eine Bounding Volume Hierarchy ist eine räumliche Datenstruktur, die aus einer Hierarchie aus Bounding Volumes besteht. Diese Bounding Volumes werden ineinander verschachtelt und bilden so eine Baumstruktur, wobei die Wurzel des Baums das Bounding Volume ist, das sämtliche Objekte der Szene umfasst. Ein Bounding Volume im dreidimensionalen Raum ist ein geometrisches Primitiv wie eine Kugel oder ein Quader, das die sie einschließende Geometrie vollständig umfasst und dabei möglichst kompakt ist. Häufig werden Quader als Bounding Volumes eingesetzt. Man spricht hierbei von einer *Bounding Box*. Sind die Bounding Boxes an den Achsen des Szenenkoordinatensystems ausgerichtet, spricht man von *Axis Aligned Bounding Boxes* (kurz: AABBs).

Abbildung 2.6 zeigt beispielhaft eine Hierarchie von AABBs. Auf der linken Seite sieht man, wie die vier Dreiecke der zweidimensionalen Szene in mehrere ineinander geschachtelte AABBs aufgeteilt werden. Auf der rechten Seite sieht man die grafische Darstellung der Baumstruktur, die sich aus dieser Hierarchie der AABBs ergibt. AABBs werden gerne eingesetzt, da ihre Berechnung sehr einfach durchzuführen ist. In jeder Dimension muss für die AABB je ein Minimal- und ein Maximalwert berechnet werden. Dieser ergibt sich aus dem Minimum beziehungsweise Maximum in dieser Dimension der Koordinaten der

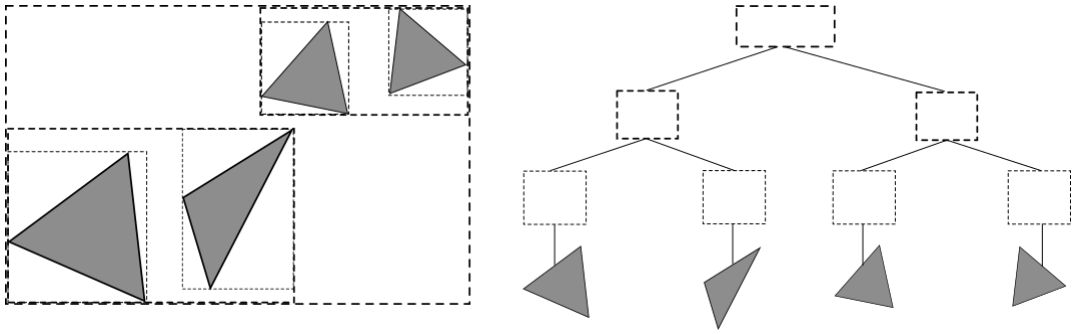

Abbildung 2.6: Darstellung einer zweidimensionalen Szene, die mit einer Hierarchie von AABBs unterteilt wurde (links), und der resultierenden Baumstruktur dieser Unterteilung (rechts)

in der AABB enthaltenen Objekte. Um auszuschließen, dass ein Sichtstrahl die Geoemtrie innerhalb einer Bounding Box schneidet, reicht es, einen Schnitttest mit der sie umschließenden Bounding Box durchzuführen. Schneidet der Strahl die Bounding Box nicht, so schneidet er auch keine Geometrie innerhalb dieser Bounding Box. Es ist allerdings nicht ausgeschlossen, dass sich AABBs, die sich auf der selben Ebene der Hierarchie befinden, überschneiden.

2.2 Reinforcement Learning

Reinforcement Learning, oder auf deutsch Bestärkendes Lernen, ist ein Teilgebiet des Maschinellen Lernens. Im Gegensatz zu Überwachtem Lernen (engl. supervised learning) benötigt Reinforcement Learning keine Information darüber, welche Entscheidung in einer bestimmten Situation die richtige ist. Stattdessen versucht der Reinforcement Learning-Agent durch Versuch und Irrtum herauszufinden, welche Aktion die größte Belohnung bringt. Hierzu interagiert der Agent mit seiner Umgebung, indem er bestimmte Aktionen auswählt und ausführt, die den Zustand seiner Umgebung beeinflussen. Er benutzt dann die Rückmeldung aus der Umgebung, um zu erfahren, wie gut oder schlecht eine gewählte Aktion gewesen ist. Das Ziel für den Reinforcement Learning-Agenten ist es, die gesamte Belohnung aus der Umgebung zu maximieren. Aktionen haben in der Regel nicht nur auf Einfluss auf die unmittelbare Belohnung, sondern auch auf alle zukünftigen Belohnungen. Als Reinforcement Learning-Agent wird die Entität im Kontext des Reinforcement Learnings bezeichnet, die die Entscheidungen darüber trifft, welche Aktion ausgeführt werden soll, und die auf Grundlage der Rückmeldung der Umgebung auf diese Aktion Erfahrungen sammelt und so lernt. Als Umgebung wird hierbei alles bezeichnet, was außerhalb des Agenten zu verorten ist.

Agent-Umgebung-Interaktion

Die Interaktion zwischen Agent und Umgebung wird in Abbildung 2.7 grafisch dargestellt. Der Agent und die Umgebung interagieren miteinander durch eine Folge diskreter Zeitschritte. In jedem Zeitschritt t wählt der Agent eine Aktion $a \in \mathcal{A}(s_t)$ aus. Diese Entscheidung trifft er auf Basis des Zustands $s_t \in \mathcal{S}$ der Umgebung zum Zeitpunkt t. Die Menge $\mathcal{A}(s_t)$ ist hierbei die Menge der Aktionen, die im Zustand s_t möglich sind. In Folge der Ausführung der Aktion a_t erfährt die Umgebung einen Zustandswechsel in den Folgezustand s_{t+1} im nächsten Zeitschritt $t + 1$. Der Folgezustand ist abhängig von der gewählten Aktion a_t. Als Folge der Ausführung der Aktion a_t gibt die Umgebung dem Agenten eine Belohnung r_{t+1}. In jedem Zeitschritt implementiert der Reinforcement Learning-Agent eine Zuordnung von Zuständen zu Wahrscheinlichkeiten für die Auswahl

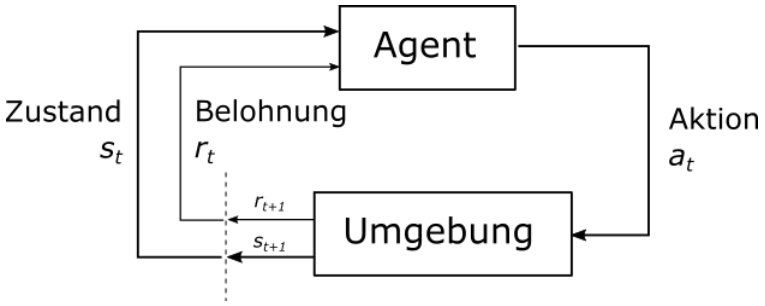

Abbildung 2.7: Interaktion des Reinforcement Learning-Agents mit der Umgebung. Grafik adaptiert übernommen aus [SB98]

jeder möglichen Aktion [SB98]. Diese Zuordnung ist die *Strategie* (engl. policy) des Agenten und wird mit π gekennzeichnet. Welche Metrik für den Belohnungswert, der dem Agenten mitgeteilt wird, zugrunde gelegt wird, entscheidet darüber, was der Agent versuchen soll, zu optimieren. Auf diesem Weg kann also dem Agent mitgeteilt werden, was seine Aufgabe ist. Ist die Metrik für den Belohnungswert in Bezug auf die zu lösende Aufgabe gut gewählt, so wird der Agent durch sein Bestreben, die Summe der Belohnungswerte, die er in jedem Zeitschritt erhält, zu maximieren, mit der Zeit eine Strategie entwickeln, mit der er die gestellte Aufgabe lösen kann. Der Agent wählt also in einem Zeitschritt t die Aktion $a_t \in \mathcal{A}(s_t)$ aus, die die höchste erwartete Gesamtbelohnung R_t nach sich zieht. Die erwartete Gesamtbelohnung kann nach Sutton und Barto wie folgt definiert werden [SB98]:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.3}$$

wobei der Parameter γ, mit $0 \leq \gamma \leq 1$, der Diskontierungssatz ist. Der Diskontierungssatz sorgt dafür, dass die Belohnungen, die ferner in der Zukunft liegen, einen geringeren Einfluss auf die Entscheidung des Agenten haben, als welche, die in der unmittelbaren Zukunft liegen.

Nutzenfunktion

Damit ein Reinforcement Learning-Agent seine Umgebung sinnvoll einschätzen kann, benötigt er eine Funktion, die dem Zustand s, in dem er sich gerade befindet, einen Wert zuweist, der angibt, wie wünschenswert es für den Agenten ist, sich in diesem zu befinden. Unter einer gegebenen Strategie π lässt sich diese Funktion beschreiben als der Erwartungswert zum Zeitpunkt t der gesamten Belohnung R_t.

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} \tag{2.4}$$

Diese Funktion wird die Zustandsnutzenfunktion (engl. state-value function) für die Strategie π genannt. Analog dazu lässt sich für eine Aktion a, die der Agent auswählt, während er sich in Zustand s befindet, eine Funktion definieren, die dieser Aktion ebenfalls einen Wert zuweist.

$$\mathcal{Q}^{\pi}(s, a) = E_{\pi}\{R_t \,|\, s_t = s, a_t = a\} \tag{2.5}$$

Dies ist die Aktionsnutzenfunktion (engl. action-value function). Sie gibt für eine Aktion a an, wie hoch die geschätzte erwartete Gesamtbelohnung sein wird, wenn der Agent diese Aktion ausgehend aus Zustand s auswählt [SB98].

Exploration und Exploitation

Der Reinforcement Learning-Agent befindet sich in jedem Zeitschritt in einem Dilemma: Der Agent hat das erklärte Ziel, die Gesamtbelohnung zu maximieren. Daher ergäbe es

für den Agenten Sinn, in jedem Zeitschritt die Aktion auszuwählen, die zum gegenwärtigen Zeitpunkt die höchste erwartete Gesamtbelohnung verspricht. Doch das würde heißen, dass er unter Umständen Aktionen, die er noch nie zuvor ausgewählt hat, bei der Entscheidung ignoriert. Dabei kann es sein, dass eine Aktion, zu der der Agent noch keine Erfahrung gesammelt hat, eine bessere erwartete Gesamtbelohnung nach sich zieht. Diese Aktion zu ignorieren würde bedeuten, dass er die ihm auferlegte Aufgabe, die maximale Gesamtbelohnung zu erhalten, nicht erfüllt. Dies ist das Dilemma von Exploration und Exploitation. Der Agent muss zwangsweise auf seinen Erfahrungsschatz zurückgreifen, um geeignete Aktionen auszuwählen, diesen also auszunutzen (engl. to exploit), andererseits muss er auch hin und wieder neue ihm unbekannte Aktionen auswählen, den Aktionsraum $\mathcal{A}(s_t)$ also explorieren. Eine beliebte Strategie, um diesem Dilemma gerecht zu werden, ist die ϵ-Greedy-Strategie. Hierbei agiert der Agent in jedem Zeitschritt mit der Wahrscheinlichkeit $(1-\epsilon)$ gierig, was bedeutet, dass er Exploitation betreibt und die Aktion auswählt, die zum gegenwärtigen Zeitpunkt in Bezug auf $Q^{\pi}(s,a)$ den höchsten Nutzen hat. Mit Wahrscheinlichkeit ϵ wird der Agent Exploration im Aktionsraum betreiben. Diese Exploration wird oft in Form einer zufälligen Wahl der Aktion implementiert.

2.2.1 Q-Learning

Bei Q-Learning handelt es sich um einen Temporal-Difference Learning-Ansatz. Eine Aktualisierung der Nutzenfunktion findet bei Temporal-Difference Learning schon im nächsten Zeitschritt statt. Die Zustandsnutzenfunktion zum Zeitpunkt t wird im Folgezeitschritt $t+1$ wie folgt aktualisiert:

$$V(s_t) \leftarrow V(s_t) + \alpha\Big[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\Big] \tag{2.6}$$

Nach Ausführen der letzten Aktion a_t ausgehend aus Zustand s_t befindet sich der Agent nun im Zustand s_{t+1}. Anhand seiner Zustandsnutzenfunktion kann er für beide Zustände den Nutzen abschätzen. Die Zustandsnutzenfunktion wird nun aktualisiert, indem auf den Wert des letzten Zustands s_t die mit α multiplizierte Summe aus der Belohnung r_{t+1} und der Differenz aus dem diskontierten Nutzen des aktuellen Zustands und des letzten Zustands addiert wird. Der Parameter α ist hier die Schrittweite und definiert, wie groß die schrittweise Aktualisierung sein soll.

Q-Learning ist eine Variante des Temporal-Difference Learnings, bei dem anstelle der Zustandsnutzenfunktion die Aktionsnutzenfunktion aktualisiert wird. Dies passiert im Gegensatz zu SARSA, einer On-Policy Temporal Difference Learning-Variante, Off-Policy. Das bedeutet, dass die Aktionsnutzenfunktion anhand einer anderen Strategie aktualisiert wird, als der, die verwendet wird, um die Entscheidung darüber zu fällen, welche Aktion im gegenwärtigen Zeitschritt ausgewählt werden soll. So benutzt Q-Learning als (diskontierten) Vergleichswert zum Nutzen $Q(s_t,a_t)$ des letzten Zeitschritts das Maximum der Aktionsnutzenfunktion unter allen $a \in \mathcal{A}(s_{t+1})$ ausgehend von dem aktuellen Zustand s_{t+1}. Im Gegensatz dazu nutzt die On-Policy-Variante SARSA den (diskontierten) Aktionsnutzenwert der aufgrund des aktuellen Zustands s_{t+1} gewählten Aktion a_{t+1} als Vergleichswert. Die Aktualisierungsfunktion von Q-Learning für die Aktionsnutzenfunktion $Q(s_t,a_t)$ ist wie folgt definiert [SB98]:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\Big[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\Big] \tag{2.7}$$

2.2.2 Eligibility Traces

Im klassischen Temporal Difference (TD) Learning und Q-Learning wird die Aktionsnutzenfunktion $Q(s_t,a_t)$ immer nur für die eine Aktion a_t aus dem vorherigen Zeitschritt t

aktualisiert. Eine Erweiterung dieses Konzepts ist TD(λ), das es unter Zuhilfenahme von *Eligibility Traces* erlaubt, die Aktionsnutzenfunktion für andere bereits in der Vergangenheit ausgewählte Aktionen zu aktualisieren. Analog zu Temporal Difference Learning gibt es diesen Ansatz auch als $Q(\lambda)$ und SARSA(λ). λ ist hierbei ein Parameter, mit $0 \leq \lambda \leq 1$, der die Zerfallsrate pro Zeitschritt der Traces angibt. Für die Umsetzung der Eligibility Traces gibt es zwei Möglichkeiten: Accumulating Traces und Replacing Traces. Bei Accumulating Traces in jedem Zeitschritt die Traces zuerst um λ diskontiert und dann wird der Eintrag für die aktuell gewählte Aktion a_t zum Zeitpunkt t um den Wert eins erhöht. Bei Replacing Traces werden in jedem Zeitschritt nach der Diskontierung die Traces-Werte für alle Aktionen, die nicht die aktuell gewählte Aktion a_t sind, auf den Wert null gesetzt und der Traces-Eintrag für die aktuelle Aktion auf eins.

2.2.3 Generalisierung auf kontinuierliche Zustandsräume

Bis zu diesem Punkt wurde in diesem Kapitel davon ausgegangen, dass ein Zustand s Element eines Zustandsraums ist, dessen Elemente unterteilbar sind, also diskrete Werte annehmen können. Doch dies ist nicht immer der Fall. Oft, weil die Anzahl diskreter Zustände je nach Aufgabenstellung und Umgebung sehr groß werden kann, vor allem, wenn der Zustand mehrdimensional ist. Aber vor allem dann, wenn er nicht nur aus Ganzzahlen besteht, sondern aus reellen Zahlen. In diesen Fällen kann man sich mit *Binning* behelfen, also dem manuellen Ziehen von Grenzen, die künstlich einen Zustand vom anderen separieren. Oder man approximiert den Zustandsraum über eine Abbildungsfunktion. Der aktuelle Zustand s_t ist nun nicht mehr als Eintrag einer Tabelle oder Index eines Arrays oder eines Vektors zu implementieren. An dessen Stelle tritt eine Approximation des Zustands s durch einen sogenannten *Feature*-Vektor ϕ_s. Dieser Vektor ist das Ergebnis einer Zustandsabbildungsfunktion. Eine solche Zustandsabbildungfunktion kann zum Beispiel ein Tile Coding sein, das den mehrdimensionalen Vektor, der den aktuellen Zustand vollständig beschreibt, in einen Vektor von Bits überführt. Eine andere Möglichkeit ist die Nutzung einer Radialen Basis-Funktion, deren Ergebnis ein Feature-Vektor reeller Zahlen ist [SB98].

Dies sind lineare Methoden zur Zustandsabbildung. Es ist allerdings auch möglich, neuronale Netze und andere nicht-lineare Funktionen zu verwenden. Dieses Thema geht allerdings über den Umfang dieser Arbeit hinaus und wird deshalb nicht weiter thematisiert.

2.3 Auto-Tuning

Die Encyclopedia of Parallel Computing definiert Auto-Tuning als einen durch Experimente geführten automatisierten Vorgang, der aus mehreren möglichen Programmimplementierungen die auswählt, mit der ein gewisses Leistungsziel erreicht werden kann [Pad11]. Im weiteren Sinne kann man unter einer Implementierung eines Programms eine bestimmte Konfiguration von Parametern verstehen, die Einfluss auf den Ablauf des Programms haben, den Algorithmus, den das Programm ausführt, also parametrisieren.
Ein klassisches Beispiel für einen solchen Parameter ist die Anzahl der Rechenfäden, die ein Algorithmus benutzen soll, um die Arbeitslast der Berechnung auf mehrere Rechenkerne zu verteilen. Unter einem Experiment versteht man in diesem Kontext die durch den Auto-Tuner hinsichtlich der Metrik des Leistungsziels überwachten Ausführung eines bestimmten Programm-Codes, der mit dem zu optimierenden Tuning-Parameter parametrisiert ist. Eine solche Metrik kann zum Beispiel die Ausführungszeit oder die Menge des für die Ausführung benötigten Arbeitsspeichers oder CPU-Last sein.

Online und Offline Auto-Tuning

Wird das Auto-Tuning zur Bestimmung guter Parameterwerte als separates Experiment vor der Ausführung des parametrisierten Programms durchgeführt, spricht man von Offline Auto-Tuning. Während des Produktivbetriebs werden die Werte der Parameter nicht mehr verändert. Dieser Vorgang des Offline Auto-Tunings kann zum Beispiel direkt im Anschluss an die Installation eines Programms auf dem Ausführungssystem oder beim ersten Start des Programms erfolgen. Bei allen folgenden Aufrufen des Programms sind dann die Parameter bestimmt und das Programm wird zu jedem weiteren Zeitpunkt mit der optimalen Parameterkonfiguration in Bezug auf das Optimierungsziel und in Abhängigkeit von der Beschaffenheit der Hardware-Konfiguration des Computers laufen. Der Nachteil dieser Methode ist, dass das Offline Auto-Tuning-Experiment unter Umständen viel Zeit in Anspruch nimmt. Hinzu kommt, dass durch die Tatsache, dass die Parameter nach Absolvierung des initialen Experiments festgelegt sind, der Tuner nicht mehr automatisch auf zum Beispiel eine sich geänderte Hardware-Konfiguration des Computers, auf dem das Programm ausgeführt wird, reagieren kann. Der Vorgang des Offline Auto-Tunings müsste in einem solchen Fall neu durchgeführt werden.

Im Gegensatz dazu vollzieht das Online Auto-Tuning die Parameteroptimierung während des Produktiveinsatzes. Hierbei wird nicht zwangsweise ein separater Programm-Code, der im Rahmen des Auto-Tuning-Experiments ausgeführt wird, ausgeführt, sondern die Messung der Metrik des Optimierungsziels wird bei Ausführung des Programms, dessen Parameter optimiert werden sollen, durchgeführt. Ein typischer Anwendungsfall ist hierbei ein Programm, das in einer Schleife eine Funktion wiederholt aufruft und das Ziel des Online Auto-Tuners soll es sein, die Laufzeit der Funktionsausführung zu minimieren. Hierzu wird die Funktion in jeder Schleifeniteration mit einer Parameterkonfiguration gestartet und ihre Laufzeit gemessen. Der Auto-Tuner exploriert im Anschluss an den Funktionsaufruf den Parameterraum und nähert sich mit der Zeit einer Parameterkonfiguration an, für die die Laufzeit der Funktion minimal wird. Der Nachteil dieses Ansatzes ist allerdings, dass die Laufzeit in den ersten Iterationen relativ hoch sein kann und erst mit den folgenden Schleifen- und somit auch Tuning-Iterationen zu ihrem Minimum konvergiert.

2.3.1 libtuning

Diese Arbeit basiert auf der Online Auto-Tuning-Bibltiothek *libtuning*, die am Institut für Programmstrukturen und Datenorganisation (kurz IPD) am Karlsruher Institut für Technologie entwickelt wird. Mit ihr kann automatisch und *online* die Laufzeit eines Programms durch Finden einer optimalen Parameterkonfiguration minimiert werden.

Benutzung

Ein Konsument dieser Bibliothek steuert den Tuner mithilfe weniger API-Aufrufe. Im ersten Schritt muss der Parametersuchraum definiert werden. Hierzu wird für jeden zu optimierenden Parameter der Suchrauminstanz der Klasse `SearchSpace` über die Methode `SearchSpace::addParameter(·)` dieser Parameter bekannt gemacht. Mit dieser Suchrauminstanz wird dann eine Instanz vom Typ `Tuner` initialisiert. Der zu optimierende Funktionsaufruf, der sich im Rumpf einer Schleife befindet, wird von den beiden Funktionsaufrufen `start()` und `stop()` der `Tuner`-Instanz eingeschlossen.

Der Tuning-Algorithmus

Die Bibliothek *libtuning* verwendet zur Exploration des Parameter(such)raums und Finden einer optimalen Parameterkonfiguration den Algorithmus von J. A. Nelder und R. Mead. Dieser Algorithmus ist ein Verfahren zur Minimierung einer Funktion von n Variablen.

Dieses Verfahren hängt vom Vergleich von Funktionswerten an den $(n + 1)$ Ecken eines allgemeinen Simplex ab [NM65]. Zu Beginn des Verfahrens wird ein zufälliger n-Simplex erzeugt, indem $(n + 1)$ zufällige Eckpunkte $P_0, \cdots, P_n$ im n-dimensionalen Raum erzeugt werden. Jeder dieser Eckpunkte P_i ist ein Vektor von Dimension n, wobei das j-te Element dieses Vektors dem Wert des j-ten Parameters entspricht. Die zu minimierende Funktion $f(x_1, \cdots, x_n)$ der n Variablen ist im Fall von *libtuning* die gemessene Laufzeit unter Einbindung der Parameterwerte x_i. Der Simplex wird in jeder Iteration der Tuning-Schleife je nach gemessener Laufzeit transformiert, bis schließlich das Konvergenzkriterium, der Standardfehler der Funktionswerte $y_i = f(P_i)$, unter einen vorher gesetzten Schwellwert gefallen ist.

3. Verwandte Arbeiten

Dieses Kapitel gibt eine Übersicht über Arbeiten und Forschungsprojekte, die sich ähnlichen Problemen gewidmet haben, wie das, das Thema der vorliegenden Arbeit ist. Es werden Arbeiten vorgestellt, die sich mit den Themen Reinforcement Learning in Kombination mit Raytracing, Auto-Tuning in Kombination mit Raytracing, oder Auto-Tuning generell befassen.

3.1 Learning Light Transport the Reinforced Way

In ihrer Arbeit „Learning Light Transport the Reinforced Way“ haben sich Ken Dahm und Alexander Keller die Frage gestellt, ob ein Path Tracer, der zur Approximation der *Importance* (Wichtigkeit) beim Importance Sampling einen Reinforcement Learning-Ansatz verwendet, bei gleicher Anazhl von Lichttransportpfaden einen Vorteil in der Bildqualität gegenüber einem klassischen Importance Sampling-Ansatz erreichen kann [DK17]. Unter Importance Sampling versteht man den Einsatz einer speziellen Wahrscheinlichkeitsdichtefunktion, die für Samples, die einen hohen Beitrag zur Approximation der abzutastenden Funktion, eine hohe Wahrscheinlichkeit vorhersagen. Die Motivation hinter der Arbeit ist, dass Licht in der echten Welt im Überfluss vorhanden ist und ein Computer unendlich viele Lichtstrahlen verfolgen müsste, um ein einhundertprozent akkurates Bild der Szene zu zeichnen. Da aber nur eine endlich große Anzahl an Strahlen verschossen werden können, ist es wichtig, bei diesen endlichen Samples so wenige wie möglich in Richtungen zu verschießen, in denen keine Lichtquelle ist. Im Kontext von Raytracing würde also eine optimale Wahrscheinlichkeitsdichtefunktion für eine Richtung, in die ein Sekundärstrahl verschossen werden soll, die einen großen Beitrag zur Farbe des Pixels, aus dem der entsprechende Primärstrahl kommt, eine hohe Wahrscheinlichkeit angeben. Der Ansatz von Dahm und Keller lernt mit der Hilfe von Q-Learning eine solche Dichtefunktion aus der gemachten Erfahrung, aus welcher Richtung bereits eine hohe Strahlungsdichte gekommen ist.

Dahm und Keller zeigen in ihrer Arbeit, dass die Rendering-Gleichung (siehe Kapitel 2.1.2) der Gleichung für die Aktualisierung der Gewichte $Q(s, a)$ der Reinforcement Learning-Methode Q-Learning (siehe Gleichung 2.7) stark ähnelt [DK17].

Würde man von einem kontinuierlichen Aktionsraum ausgehen, würde die modifizierte Rendering-Gleichung, die zur Aktualisierung von $Q(s, a)$ benutzt werden kann, wie folgt aussehen:

$$Q(x, \omega) = L_e(y, -\omega) + \int_{\Omega^+(y)} Q(y, \omega_i) f_s(\omega_i, y, -\omega) \cos\theta_i \, \mathrm{d}\omega_i \tag{3.1}$$

Analog zur Rendering-Gleichung ist der Term $L_e(y, -\omega)$ die Lichtstärke, die von der Oberfläche an Punkt y in Richtung $-\omega$ emittiert wird. Das Integral läuft über die positive Hemisphäre $\Omega^+(y)$ der Oberfläche, auf der der Punkt y liegt. f_s ist die BRDF und θ_i ist der Einstrahlwinkel der Richtung $\mathrm{d}\omega_i$. Bei dieser Gleichung wird für den Zustand des Reinforcement Learning-Agents der Punkt x genommen, dessen Farbe berechnet werden soll. Eine Aktion ist demnach eine Richtung ω, die die Richtung eines von dem Punkt x aus verschossenen Sekundärstrahls beschreibt. Man erkennt sehr schön die Eleganz dieses Lösungsansatzes: Die Güte Q für eine Aktion, ausgehend von einem bestimmten Zustand, das heißt, die Güte Q für die Wahl eines Sekundärstrahls ausgehend von einem Punkt der in der Szene entspricht der Strahlungsdichte, die aus dieser Richtung auf den Punkt einfällt. Dahm und Keller benutzen zur Diskretisierung des Zustandsraums ein dreidimensionales Gitter, das über die zu rendernde Szene gelegt wird und eine Einteilung der positiven Hemisphäre eines Oberflächenpunkts in differentielle Raumwinkel. Das Integral der Gleichung 3.1 wird in der Arbeit durch die folgende Summenformel approximiert [DK17]:

$$\int_{\mathcal{S}^+(y)} Q(y, \omega_i) f_s(\omega_i, y, -\omega) \cos\theta_i \, d\omega_i \approx \frac{2\pi}{n} \sum_{k=0}^{n-1} Q_k(y) f_s(\omega_k, y, -\omega) \cos\theta_k \,, \tag{3.2}$$

wobei n hier die Anzahl der differentiellen Raumwinkel ist.

Die Ergebnisse, die Dahm und Keller mit ihrer Arbeit erreichen konnten, fallen sehr positiv aus. Das Rauschen in den gezeichneten Bildern ist bei gleicher Anzahl an Lichttransportpfaden sichtbar geringer als bei klassischem Importance Sampling, bei einem zeitlichen Overhead von nur 20% [DK17].

Im Unterschied zu der vorliegenden Arbeit benutzen Dahm und Keller Reinforcement Learning, um in einem gegebenen Zustand eine optimale Richtung für das Verschießen des nächsten Sichtstrahls vorherzusagen, wohingegen die vorliegende Arbeit Reinforcement Learning einsetzt, um eine optimale Parameterkonfiguration für das Erstellen einer BVH vorherzusagen. Die beiden Arbeiten schließen sich gegenseitig nicht aus, sondern sie würden sich sogar sehr gut ergänzen.

3.2 Online-Autotuning of Parallel SAH kD-Trees

In ihrer Arbeit über „Online-Autotuning of Parallel SAH kD-Trees“ verwenden Tillmann et al. eine Online-Autotuning-Bibliothek namens *AtuneRT*, um den Aufbau eines kD-Trees mit Surface Area Heuristic (SAH) zu beschleunigen, indem sie die Bibliothek optimale Parameter für die SAH finden lassen [TPKT16].

Die Motivation dahinter ist die Tatsache, dass es keine global optimale Parameterkonfiguration für die SAH gibt. Stattdessen hängt die Wahl der optimalen Parameterkonfiguration von den Eingabedaten und der Hardware-Konfiguration des Computers ab, auf dem die Berechnungen, die den kD-Tree benötigen, ausgeführt werden sollen. Auch die in der Literatur vorgeschlagenen Werte für diese Parameter sind nicht immer optimal [TPKT16].

AtuneRT ist eine Online-Auto-Tuning-Bibliothek, die zur Funktionsminimierung den Algorithmus von Nelder und Mead verwendet. Sie zeichnet sich durch einfache Benutzung und eine kleine API aus.

Tillmann et al. definieren das Auto-Tuning-Problem, also das Problem, das mithilfe von *AtuneRT* gelöst werden soll, wie folgt: Ein gegebener Algorithmus a ist durch die Tuning-Parameter $\tau_{a,j}$ parametrisierbar. Der Suchraum T_a für *AtuneRT* lässt sich also wie folgt definieren:

$$T_a = \tau_{a,0} \times \cdots \tau_{a,J}$$

Eine Tuning-Parameterkonfiguration C_a ist ein Punkt in diesem Suchraum. Die zu minimierende Funktion $m_a(C_a, K)$ ist die Funktion der Laufzeit, die von der gewählten Parameterkonfiguration C_a und dem Messkontext K abhängt. Die gesuchte optimale Konfiguration ist also [TPKT16]:

$$C_{\text{opt},a} = \arg\min_{C_a} m_a(C_a, K)$$

SAH kD-Trees werden solange rekursiv unterteilt, bis eine Abbruchbedingung erfüllt ist. Diese Abbruchbedingung und die Entscheidung, wo die nächste Separierungsebene in der rekursiven Unterteilung gesetzt werden soll, sind nach Tillmann et al. die Freiheitsgrade, die durch die SAH formalisiert werden. Die für *AtuneRT* zu minimierende Kostenfunktion für Aufbau und Traversierung wird definiert als:

$$SAH(h, b) = C_T + p(l, b) \cdot N_l \cdot C_I + p(r, b) \cdot N_r \cdot C_I + (N_l + N_r - N_b) \cdot C_B\,, \tag{3.3}$$

wobei h die Separierungsebene ist, die die Bounding Box b in die Subquader l und r unterteilt. N_l und N_r geben die Anzahl der Primitive, die sich in dem Subquader l beziehungsweise r befinden, an. C_T gibt die konstanten Kosten für die Traveriserung eines Baumknotens an, C_I die Kosten für einen Schnitttest mit einem Primitiv und C_B die Kosten für das Duplizieren eines Primitivs und doppelte Einfügen in den Baum, sollte es durch die Separierungsebene geschnitten werden. Die Funktion

$$p(b_{\text{sub}}, b) = \frac{A(b_{\text{sub}})}{A(b)}$$

ist als Bruch der Flächen eines Unterquaders von b und der Fläche von b definiert und berechnet damit die Wahrscheinlichkeit für einen zufälligen Sichtstrahl, diesen Subquader b_{sub} zu schneiden, sollte er auch dessen Elternquader b geschnitten haben. Wird nun

$$\arg\min_{h} SAH(h, b)$$

gelöst, kann die optimale Separierungsebene gefunden werden [TPKT16].

Tillmann et al. implementieren vier Verfahren zur Konstruktion eines kD-Trees und testen ihren Ansatz, indem sie einen simplen Raytracing-Algorithmus 3 statische und 3 dynamische Szenen rendern lassen. Es zeigt sich, dass durch das Online Auto-Tuning Beschleunigungen um einen Faktor von bis zu 1,96 erreicht werden können. Desweiteren wird durch Vergleiche der gefundenen optimalen Parameterkonfigurationen auf einem anderen Computer mit anderen Szenen gezeigt, dass es keine global optimale Wahl für die SAH-Parameter gibt [TPKT16].

Die Arbeit von Tillmann et al. lässt sich als gedanklicher Vorgänger zu der vorliegenden Arbeit betrachten. Die Ziele beider Arbeiten sind sehr ähnlich: Sie haben beide das Ziel, die räumliche Datenstruktur beim Raytracing zu durch geschickte Parameterwahl online zu beschleunigen. Der Unterschied zu der vorliegenden Arbeit liegt daher neben der Wahl der räumlichen Datenstruktur in dem in der vorliegenden Arbeit implementierten Hybrid-Ansatz aus Nelder-Mead-Algorithmus und generalisiertem Reinforcement Learning.

3.3 Auto-Tuning Interactive Ray Tracing using an Analytical GPU Architecture Model

Da Raytracing ein sehr rechenaufwändiges Verfahren ist, wird es nur selten in der interaktiven (Echtzeit-) Bildsynthese eingesetzt. Ein Problem ist, dass die Render-Zeit eines Frames stark von der Komplexität der zu rendernden Szene abhängt und es so zu starken Schwankungen in der Bildwiederholrate kommt. Per Ganestam und Michael Doggett benutzen in

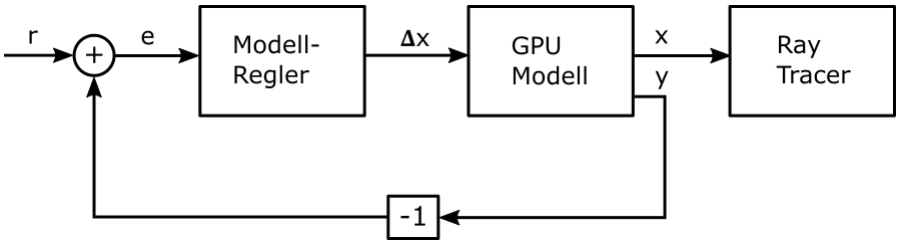

Abbildung 3.1: Raytracer-Autotuning mit GPU-Modell und Modellregler. Abbildung adaptiert übernommen aus [GD12]

ihrer Arbeit „Auto-Tuning Interactive Ray Tracing using an Analytical GPU Architecture Model“ einen Auto-Tuning-Ansatz, der ein „analytisches GPU-Leistungsmodell [verwendet], um die Render-Zeit des derzeitigen Frames vorherzusagen“ [GD12]. Das Ziel ist es, eine möglichst konstante Bildwiederholrate zu erreichen. Hierfür setzen sie eine Menge bestimmter Parameter ein, die die Bildwiederholrate beeinflussen. Um die Grafikkarten der beiden Hersteller Nvidia und AMD verwenden zu können, wird OpenCL als Programmierschnittstelle verwendet.

Ganestam und Doggett verwenden für ihre Arbeit das GPU-Modell von S. Hong und H. Kim [HK09]. Dieses Modell verwendet Eigenschaften des verwendeten Geräts, also in dieser Arbeit die der Grafikkarte, und Eigenschaften des auszuführenden Programms. Von diesem Modell berücksichtigte Eigenschaften des Programms sind zum Beispiel die Anzahl der *Compute*-Instruktionen oder die Anzahl der Speicherinstruktionen. Eigenschaften des verwendeten Geräts, die von diesem Modell berücksichtigt werden, sind unter anderem die gesamte Speicherbandbreite, die Anzahl Rechenkerne oder die Anzahl Zyklen, die für die Exekution einer Instruktion benötigt werden.

Um die Bildwiederholrate zu beeinflussen verwenden Ganestam und Doggett zwei Raytracing-Parameter: Die Benutzung oder Nichtbenutzung von Schattenstrahlen und die Anzahl der Strahlen zur Berechnung der Umgebungsverdeckung (engl. ambient occlusion).

Abbildung 3.1 „veranschaulicht, wie das GPU-Modell verwendet wird, um den Raytracer [...] mit den Raytracing-Parametern einzustellen“ [GD12]. x steht hier für die Raytracing-Parameterkonfiguration. Die Referenz r ist die Ziel-Frame-Render-Zeit, die über die Dauer des interaktiven Raytracings konstant gehalten werden soll, während y die von dem GPU-Modell geschätzte Frame-Render-Zeit ist. Der Fehler e ist die Differenz von y und r. „Diese Regelschleife wird mehrfach pro Frame iteriert und aktualisiert x solange, bis der Fehler e so klein wie möglich“ geworden ist [GD12].

„Die Vorsteuerung simuliert den Raytracer mit einem gewissen Fehler aufgrund unmodellierten Verhaltens“ [GD12]. Durch Hinzufügen einer Feedback-Schleife, die den Fehler zwischen vorhergesagter Frame-Render-Zeit y und der tatsächlich gemessenen Frame-Render-Zeit korrigiert konnte die Vorhersage der Frame-Render-Zeit deutlich verbessert werden. Ganestam und Doggett konnten auf diese Weise die Frame-Render-Zeit präzise vorhersagen. Der Vorhersagefehler liegt zwischen 0,2 und 1,7 % auf den drei getesteten Grafikkarten Nvidia Geforce 580, 8800 und AMD Radeon 5870.

Auch wenn die Arbeit von Ganestam und Doggett ähnlich wie die vorliegende Arbeit unter Einsatz von Online Auto-Tuning die Rendering-Laufzeit bei Raytracing positiv beeinflussen will, so liegt der große Unterschied zwischen beiden Arbeiten darin, dass Ganestam und Doggett das Ziel haben, bei *interaktivem* Raytracing eine vorher festgelegte konstante Frame-Render-Zeit zu halten, während die vorlegende Arbeit eine größtmögliche Minimierung derselben Zeit anstrebt. Dies hat zur Folge, dass Ganestam und Doggett es zulassen,

dass ihr Algorithmus Abstriche in der Bildqualität des Renderings zulässt, während die Bildqualität in der vorliegenden Arbeit unangetastet bleibt.

3.4 Workstation Capacity Tuning using Reinforcement Learning

Computer-Grids werden immer größer und komplexer. Daher wird es immer schwieriger, gute Parametereinstellungen für solche Grids per Hand einzustellen, sodass die eingesetzten Rechenkerne optimal ausgelastet werden. Bar-Hillel et al. versuchen in ihrer Arbeit „Workstation Capacity Tuning using Reinforcement Learning“ mithilfe von drei verschiedenen Reinforcement Learning-Methoden dieses Problem mittels Auto-Tuning zu lösen. Hierbei beschränken sie sich auf das Tuning eines einzigen Parameters, der zu Laufzeit des verwendeten Grids dynamisch an den gegenwärtigen Zustand des Grids angepasst wird. Bei diesem Parameter handelt es sich um die Anzahl nebenläufig ausgeführter Jobs einer einzelnen Grid Workstation. Die Strategie der drei Reinforcement Learning-Implementierungen wird gemessen an der Strategie, die normalerweise in dem verwendeten Grid eingesetzt wird: Die, dass jede Workstation einen Job pro Rechenkern ausführen soll [BHDNED+07].

Der Zustandsraum

Der Zustandsraum für die Reinforcement Learning Agents ist kontinuierlich und setzt sich aus den folgenden Indikatoren zusammen: Freier physikalischer Arbeitsspeicher, benutzter virtueller Arbeitsspeicher, CPU-Last, CPU-Idle-Zeit, CPU-Systemzeit (der Anzahl der Zeit, den die CPU System- und I/O-Prozeduren in den letzten 30 Sekunden ausgeführt hat), *Fit* des letzten Zeitstritts $t-1$ (siehe Gleichung 3.5) [BHDNED+07].

Der Aktionsraum

Der Aktionsraum der Reinforcement Learning Agents besteht aus drei Aktionen, die sich alle auf den zu optimierenden Parameter (Anzahl nebenläufiger Jobs) beziehen [BHDNED+07]:

- Anzahl um eins erhöhen
- Anzahl ums eins reduzieren
- Anzahl unverändert lassen

Die Belohnung

Der Belohnungswert r_t in Zeitschritt t wird mithilfe der beiden folgenden Terme definiert: Die *unmittelbare* Belohnung für einen Job j in Zeitschritt t:

$$r(j,t) = Wtime(j,t) \cdot \left(\frac{Utime(j,t)}{Wtime(j,t)} \right)^{\alpha}, \tag{3.4}$$

wobei $Wtime(j,t)$ der Anteil der Zeit ist, die Job j auf der jeweiligen Workstation während des Zeitschritts t alloziert war und $Utime(j,t)$ der Anteil der CPU-Ressourcen ist, die für Job j während des Zeitschritts t alloziert gewesen sind.

Der *Fit* in Zeitschritt t ist definiert als

$$f(t) = \sum_{j} r(j,t) \tag{3.5}$$

Die Belohnung r_t in Zeitschritt t wird von Bar-Hillel et al. definiert mit einem positiven Term und einem Strafterm: Der positive Term ist der *Fit*, der Strafterm „bestraft für

Jobs, die vorzeitig terminiert aufgrund einer Reduzierungsaktion“, die durch den RL-Agent ausgewählt worden ist. „In diesem Strafterm werden alle unmittelbaren Belohnungen, die in der Vergangenheit durch die relevanten Jobs erworben wurden, reduziert“ [BHDNED+07].

$$r(t) = f(t) - \sum_{j \in Term(t)} \sum_{t' \leq t} r(j, t') \,, \tag{3.6}$$

wobei $Term(t)$ die Menge aller Jobs ist, die vorzeitig in Zeitschritt t beendet wurden.

Reinforcement Learning-Implementierungen

Die drei in diesem Paper implementierten Reinforcement Learning-Algorithmen sind HMM-LP, TD(λ) und Fitted Q Iteration. HMM-LP (Hidden Markov Model-Linear Programming) zeichnet sich dadurch aus, dass hierbei davon ausgegangen wird, dass der Zustand nur mit einem hohen Signalrauschen wahrgenommen werden kann, weswegen das Lernen in zwei Phasen stattfindet: Einer Phase des Lernens des dem verrauschten Zustandssignal zugrundeliegenden tatsächlichen Zustands, gefolgt von einer zweiten Phase, in der die optimale Strategie für das in der ersten Phase gelernte Modell erstellt wird.

TD(λ) ist der einzige der drei Algorithmen, der *online* arbeitet. Für nähere Details siehe Kapitel 2.2.2.

Bei Fitted Q Iteration wird Reinforcement Learning auf „eine Reihe von überwachten Regressionsproblemen reduziert, wobei sukzessive die optimale Q-Funktion“ angenähert wird [BHDNED+07].

Ergebnis und Bewertung

Bar-Hillel et al. konnten mit ihrem Ansatz eine durchschnittliche Verbesserung des Durchsatzes von über 20% im Vergleich zur Standardstrategie erreichen. Auf Mehrkern-Workstations lag die durchschnittliche Durchsatzverbesserung bei ca. 40% [BHDNED+07].

Die Arbeit von Bar-Hillel et al. verwendet ähnlich wie die vorliegende Arbeit Reinforcement Learning auf kontinuierlichen Zustandsräumen für das Auto-Tuning eines stark parallelisierbaren Arbeitsauftrags. Im Unterschied zu der vorliegenden Arbeit ist *online* Auto-Tuning keine Notwendigkeit, sondern wird stattdessen nur in einer von drei Implementierungen eingesetzt. Außerdem ist der Aktionsraum verglichen mit dem aus dem Raytracing-Kontext sehr klein. Desweiteren unterscheiden sich beide Arbeiten in der Art, wie sie Exploration betreiben. Während Bar-Hillel et al. in ihrer einen zufallsbasierten Ansatz wählen, setzt die vorliegende Arbeit auf das in *libtuning* implementierte Verfahren zur Funktionsminimierung von J. A. Nelder und R. Mead [NM65].

3.5 Active Harmony: Towards Automated Performance Tuning

Active Harmony ist ein Online Auto-Tuner, mit dem es nicht nur möglich ist, ein Programm durch automatisches Finden einer optimalen Parameterkonfiguration zu optimieren. Es ist außerdem möglich, Active Harmony aus einer Menge von verschiedenen Implementierungen derselben Funktionalität die auswählen zu lassen, die in Bezug auf eine bestimmte Metrik optimal ist [ŢCH+02].

Die Benutzung der Bibliothek ist nach dem Client-Server-Modell aufgebaut. Der Client ist die zu optimierende Anwendung, der Server besteht hauptsächlich aus dem Active Harmony *Adaptation Controller*. Auf dem Client existiert eine Schicht, die dem Rest der Anwendung eine einheitliche Programmierschnittstelle anbietet, über die sie die Aufrufe

an die austauschbaren Implementierungen stellen kann. Diese Schicht, der *Library Specification Layer*, kommuniziert mit dem Adaption Controller im Server bei jeder Anfrage durch die Anwendung und liefert ihm gewisse Charakteristika dieser Anfrage. Aufgrund derer und der in der Vergangenheit ermittelten Performanz der zuvor ausgewählten Implementierungen wählt der Adaptation Controller eine Implementierung aus, an die dann durch den Library Specification Layer die Anfrage der Anwendung weitergeleitet wird.

Für die Parameteroptimierung nutzt Active Harmony den Algorithmus von J. A. Nelder und R. Mead [NM65] zur Funktionsminimierung.

Active Harmony und die vorliegende Arbeit basieren beide auf dem gleichen Algorithmus von J.A. Nelder und R. Mead für das Online Auto-Tuning. Ein Einsatz von Active Harmony für die Laufzeitoptimierung von Raytracing, wie sie in dieser Arbeit mithilfe des Hybridansatzes aus Nelder-Mead-Algorithmus und Reinforcement Learning vorgenommen wird, hätte den entscheidenden Nachteil, dass pro zu renderndem Sample eines Frames ein kompletter BVH-Neubau vollzogen werden müsste, da die Tuning-Schleife für den Nelder-Mead-Algorithmus zwangsweise mit der Sample-Rendering-Schleife zusammenfällt. Viele dieser BVH-Neubauten können mithilfe des in der vorliegenden Arbeit entwickelten Hybrid-Ansatzes eingespart werden.

4. Analyse und Entwurf

Wie in den vorherigen Kapiteln schon angerissen, soll das Problem der automatischen Laufzeitoptimierung einer rechenaufwändigen Arbeitslast mithilfe einer Erweiterung der Online Auto-Tuning-Bibliothek *libtuning* gelöst werden. Diese Erweiterung bedient sich einem Ansatz aus dem Bereich des Maschinellen Lernens, der sich Reinforcement Learning nennt. Auf diese Erweiterung, die den Kern dieser Arbeit darstellt, wird im Weiteren mit *librltuning* Bezug genommen. Konkret soll die automatische Laufzeitoptimierung im Kontext von *Raytracing* und dem Rendering einer Folge mehrerer Bilder behandelt werden.

4.1 Der Zustandsraum

Wie in Kapitel 2.2 erklärt, wählt der Reinforcement Learning-Agent in Abhängigkeit von dem Zustand, in dem er sich aktuell befindet, eine Aktion aus, von der er sich eine möglichst hohe Belohnung erwartet. Diese Belohnung ist davon abhängig, wie gut der Zustand ist, in den ihn die gewählte Aktion befördert. In *librltuning* wird allerdings davon ausgegangen, dass der Zustand von der Umgebung verändert wird und nicht durch Anwendung einer Aktion. Im Einsatzgebiet der Parameteroptimierung tritt ein Zustandswechsel immer dann ein, wenn ein neues Eingabedatum verarbeitet werden soll. Der Zustand ist in dem Fall von den Charakteristika des Eingabedatums abhängig.

Ein Zustand lässt sich durch einen Vektor

$$s \in \mathbb{R}^n$$

beschrieben. Dieser Vektor enthält n reelle Zahlen, die jeweils eine Eigenschaft dieses Zustands beschreiben. Wichtig ist hier, dass es sich um reelle Zahlen handelt und dass diese weder nach oben noch nach unten beschränkt sein müssen. Die Menge der möglichen Zustände ist also nicht abzählbar. Vielmehr ist der Zustandsraum einem Vektorraum reeller Zahlen mit fester Dimension n. Diese Erweiterung von einem diskreten Zustandsraum hin zu einem kontinuierlichen ist nötig für die Anwendung von Reinforcement Learning bei Problemstellungen, in denen die Eingabedaten nicht anhand bestimmter Merkmale kategorisiert werden können. Wenn zum Beispiel das Ziel eines humanoiden Roboters ist, an das Ende eines Raums zu gelangen und seine Eingabedaten die Koordinaten seiner gegenwärtigen Position im Raum sind, kann diese Position durch einen zweidimensionalen Vektor reeller Zahlen repräsentiert werden.

Im Kontext der Laufzeitoptimierung von Raytracing sind die Eingabedaten die Folge von Frames, die nacheinander gerendert werden sollen. Der Zustand leitet sich von gewissen

Charakteristika dieser Eingabedaten ab. Für diese Arbeit wurden zehn Kennzahlen eines Frames als Indikatoren für dessen Eigenschaften herangezogen. Das sind zum Einen vier Eigenschaften der zu rendernden Szene und zum anderen die Werte des sechsdimensionalen Vektors, der die Kameraposition und -orientierung beschreibt, aus deren Blickwinkel die Szene gerendert werden soll. Die in dieser Arbeit gewählten Szeneneigenschaften sind die folgenden:

- Anzahl der Vertexe
- Anzahl der Dreiecke
- *diffuse ratio* - Verhältnis des Flächeninhalts diffuser Oberflächen zum Flächeninhalt spekularer Oberflächen
- Summierte Fläche aller Oberflächen

Daraus ergibt sich ein Vektor

$$\mathcal{I} = \begin{pmatrix} \text{Anzahl Vertexe} \\ \text{Anzahl Dreiecke} \\ \text{diffuse ratio} \\ \text{Summierte Fläche} \\ \text{Kameraposition}.x \\ \text{Kameraposition}.y \\ \text{Kameraposition}.z \\ \text{Kameraorientierung}.x \\ \text{Kameraorientireung}.y \\ \text{Kameraorientierung}.z \end{pmatrix}$$

Einen solchen Vektor nennen wir Indikatorvektor.

Betrachtet man die Elemente dieses Indikatorvektors, sieht man schnell ein, dass die Wertebereiche einzelner Indikatoren sehr weit auseinander liegen. So liegt der Wertebereich der „diffuse ratio“ zwischen 0 und 1, der Wertebereich der summierten Fläche aller Oberflächen in der Szene zwischen 0 und unendlich. Da die einzelnen Indikatoren zur Berechnung einer optimalen Aktion herangezogen werden, würde dieser Umstand dafür sorgen, dass Indikatoren, die typischerweise große Werte annehmen, einen viel größeren Einfluss auf das Ergebnis dieser Berechnung haben als Indikatoren, die typischerweise eher kleinere Werte annehmen. Dabei ist es doch für die Aussagekraft eines bestimmten Indikatorwerts interessant, wie sehr sich sein Wert von dem Mittelpunkt unterscheidet. Aus diesem Grund wird eine Abbildungsfunktion ϕ eingesetzt, die die einzelnen Indikatoren anhand ihrer in der Praxis angenommenen Mittelpunkte zentriert, sodass einzelne Indikatoren keinen größeren Einfluss bei der Berechnung der erwarteten Qualität einer Aktion als andere haben.

In dieser Arbeit wurde auf eine Reihe von Radialen Basis-Funktionen ϕ_i zurückgegriffen, die einen Indikatorvektor s auf einen sogenannten Feature-Vektor ϕ abbilden. Eine solche Funktion ϕ_i ist wie folgt definiert:

$$\phi_i(s) := e^{-\frac{|s - c_i|^2}{2\sigma^2}} \tag{4.1}$$

Mit den Parametern s, c_i, σ in Kombination mit der frei wählbaren Dimension d des resultierenden Feature-Vektors ϕ ergeben sich für die Wahl einer konkreten Abbildungsfunktion von Indikatorraum auf Feature-Raum diverse Freiheitsgrade. ϕ_i ist also das i-te Feature, wobei $i \in \{1, \cdots, d\}$ gilt. c_i gibt den Mittelpunkt der Gaußglocke von ϕ_i an. σ^2 gibt die

Varianz und somit die Breite der Gaußglocke an. Der Wert eines Features berechnet sich aus dem Zustand s. Da wir für jeden Indikator einen Mittelwert berechnen, muss es genau so viele c_i geben, wie es Indikatoren gibt. Aus diesem Grund wird die Dimension d des Feature-Vektors gleich der Anzahl der Indikatoren $|\mathcal{I}|$ gesetzt. Jeder Indikator $\mathcal{I}_i$ wird demnach auf ein Feature ϕ_i abgebildet. Das s in dieser Formel ist demnach der i-te Indikator $\mathcal{I}_i$. Im Anwendungsfall Raytracing lässt sich dieser Mittelpunkt eines Indikators dadurch berechnen, dass für alle zu zeichnenden Szenen und Kamerapositionen und -orientierungen die Indikatorvektoren erzeugt werden und nun darüber ein Mittelpunktvektor berechnet wird, für jeden Indikator das Minimum sowie das Maximum bestimmt werden. Der jeweilige Mittelwert für einen Indikator ergibt sich nun aus dem Mittelwert dieser beiden Werte. Gegeben eine Liste von Indikatorvektoren I_j mit $j \in \{1, ..., l\}$, wobei l die Anzahl der zu zeichnenden Kombinationen aus 3D-Szene und Kameraposition- und orientierung und somit die Anzahl der Indikatorvektoren darstellt, lässt sich der Mittelpunktvektor $c \in \mathbb{R}^{|\mathcal{I}|}$ wie folgt berechnen:

$$c_i := min_j I_{j_i} + \frac{1}{2}(max_j I_{j_i} - min_j I_{j_i})$$

Der Wert für σ^2 ist frei wählbar und gibt an, wie breit die Gaußkurve sein soll. Als letzter Parameter der Radialen Basis-Funktion wurde σ^2 auf den Wert $0,5$ gesetzt. Der Grund hierfür ist, dass dadurch der Nenner gleich eins wird. Da die Wahl von σ^2 nur dafür sorgt, dass die Glockenkurve breiter oder schmaler wird und derselbe Wert für alle i gewählt wird, sollte ein bestimmter Wert für σ^2 im Vergleich zu einem anderen keinen Einfluss darauf haben, wie gleichmäßig die Indikatorwerte auf die Feature-Werte abgebildet werden. Außerdem wird mit der Wahl von $\sigma^2 = 0,5$ eine Divisionsoperation eingespart, da der Nenner gleich eins wird.

4.2 Aktionsraum und Suchraum

Für den Reinforcement Learning-Agenten ist der Aktionsraum eine endliche Liste von Aktionen, die ausgeführt werden können. In Abhängigkeit von dem Zustand, in dem sich der RL-Agent befindet, existiert eine Teilliste dieser Liste von Aktionen, die der RL-Agent ausführen kann. Dies ist in dieser Implementierung nicht der Fall. In jedem Zustand kann der Reinforcement Learning-Agent aus der gesamten Menge aller Aktionen auswählen.

Im klassischen Reinforcement Learning hat das Ausführen einer Aktion einen Effekt auf die Umgebung und überführt den aktuellen Zustand, in dem sich der RL-Agent zu Ausführungsbeginn befindet, in einen neuen Zustand. Da diese Reinforcement Learning-Implementierung davon ausgeht, dass der Zustand von außen verändert wird, zum Beispiel dadurch, dass der nächste zu zeichnende Frame ausgewählt wird, haben die Aktionen, die der Reinforcement Learning-Agent von *librltuning* auswählt, keinen Effekt auf den Zustand des nächsten Zeitschritts. Eine gewählte Aktion hat allerdings einen Effekt auf die Laufzeit der zu optimierenden Funktion, indem diese Aktion mit einer Konfiguration von Parametern korrespondiert, mit diese Funktion parametrisiert wird.

Der Suchraum hingegen ist wie in Kapitel 2.3 zu Auto-Tuning beschrieben der Raum aller möglichen Parameterkonfigurationen und der Raum innerhalb dessen der Nelder-Mead-Algorithmus operiert.

Der Suchraum von *libtuning* unterstützt unter anderem Intervallparameter. Hierbei handelt es sich um Parameter, die einen Wert in den für diesen Parameter angegebenen Grenzen annehmen können. Dies können reelle oder ganze Zahlen sein. Diese Arbeit ermöglicht zwar die Nutzung eines kontinuierlichen Zustandsraums, unterstützt allerdings nur diskrete Aktionsräume. Da die Reinforcement Learning-Implementierung von *librltuning* den

Nelder-Mead-Algorithmus aus *libtuning* zur Exploration verwendet, müssen Aktionsraum und Suchraum ineinander überführbar sein. Da also ausschließlich diskrete Aktionsräume unterstützt werden, werden auch nur solche Suchräume unterstützt, die ausschließlich aus ganzzahligen Intervallparametern bestehen.

Für den Algorithmus von Reinforcement Learning ist es notwendig, dass für alle Aktionen, die ausführbar sind, die erwartete Belohnung, die die Ausführung einer Aktion nach sich zieht, in Form eines Vektors $W \in \mathbb{R}^{|\mathcal{A}|}$ vorliegen muss. Die Anzahl $|\mathcal{A}|$ aller möglichen Aktionen lässt sich wie folgt berechnen:

$$|\mathcal{A}| := \prod_{i=1}^{N_P} |P_i|$$

wobei P_i der i-te der n Intervallparameter des Suchraums ist und $|P_i|$ die Anzahl der nominalen Werte des i-ten Nominalparemters beschreibt.

4.2.1 Wahl geeigneter Parameter

Das Ziel des Reinforcement Learning-Agenten ist im Grunde die Suche nach einer Abbildungsfunktion, die den Zustandsraum derart auf den Suchraum abbildet, sodass die Ausführungszeit des durch den abgebildeten Parametervektor parametrisierten Renderings eines Frames mit bestimmten Indikatoren minimal ist.

Die zu optimierenden Parameter sind Parameter für den Aufbau der räumlichen Datenstruktur BVH und stammen aus der BVH-Implementierung der Embree-Bibliothek. Bei diesen Parametern handelt es sich um acht Ganzzahlparameter und zwei reelle Parameter. Da sich diese Arbeit auf diskrete Aktionsräume beschränkt, wurden die beiden reellen Parameter ignoriert und bei ihrem Standardwert belassen. Die acht ganzzahligen Parameter für den Aufbau der BVH, die für die Laufzeitoptimierung verwendet werden, sind die folgenden:

- paralellMode
- PacketMode_Count
- ex_quality
- ex_maxBranchingFactor
- ex_maxDepth
- ex_sahBlockSize
- ex_minLeafSize
- ex_maxLeafSize

4.3 Umsetzung des Generalisierten Q-Learnings

Mit den Temporal Difference-Methoden SARSA und Q-Learning ist es möglich, einem Zustand-Aktions-Paar eine Güte zuzuweisen. Diese Güte ist eine Schätzung, wie hoch die Gesamtbelohnung nach dem letzten Zeitschritt sein wird. In jedem Zeitschritt wird diese Schätzung mithilfe der Rückmeldung aus der Umgebung für die Auswahl einer bestimmten Aktion verfeinert.

Die Aktualisierungsfunktion zur Verfeinerung der Güteschätzung $Q(s_t, a_t)$ mit SARSA ist wie folgt definiert [SB98]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big], \tag{4.2}$$

wobei s, a Zustand beziehungsweise Aktion des letzten Zeitschritts darstellen, s' der Zustand ist, der ausgehend von Zustand s erreicht worden ist, indem Aktion a durchgeführt worden ist, und a' die Aktion ist, die durch Anwendung der ausgewählten Strategie π als nächste anzuwendende Aktion ausgewählt worden ist.

Die Aktualisierungsfunktion für die Off-Policy Variante, Q-Learning, ist, wie in Kapitel 2.2.1 beschrieben, wie folgt definiert:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big] \tag{4.3}$$

Der praktische Unterschied zwischen beiden Lernfunktionen 4.2 und 4.3 liegt also an dem Wert, mit dem das aktuelle $Q(s_t, a_t)$ verglichen wird, um dessen Wert zu aktualisieren. Bei dem SARSA-Algorithmus wird also die Differenz zwischen der (um den Faktor γ diskontierten) Güte der aktuellen Aktion a_{t+1} ausgehend von dem aktuellen Zustand s_{t+1} und der Güte der zu aktualisierenden Kombination aus Zustand s_t und Aktion a_t aus dem letzten Zeitschritt gebildet wird, um die Güte von a_t und s_t zu berechnen. Bei Q-Learning hingegen wird die Differenz zwischen der Güte der Kombination von Zustand s_t und Aktion a_t zum Zeitpunkt t und dem Maximum aller Güte-Werte aller Aktionen a ausgehend von dem aktuellen Zustand s_{t+1} gebildet.

4.3.1 Zustandsraumgeneralisierung

Wie man an den Gleichungen 4.2 und 4.3 sehen kann, sind sowohl Zustand als auch Aktion diskret. Damit nun ein kontinuierlicher Zustandsraum, wie er in 4.1 beschrieben ist, eingesetzt werden kann, muss der Index s_t, der den diskreten Zustand zum Zeitschritt t identifiziert, ersetzt werden durch einen Zustands-Feature-Vektor ϕ, mit dessen Hilfe der Zustandsraum approximiert werden kann. Richard Sutton liefert eine mögliche Umsetzung einer Aktualisierungsfunktion für den in diesem Paper vorgestellten Generalisierten SARSA-Algorithmus [Sut96]:

$$w_b(f) := w_b(f) + \frac{a}{c}\Big[r + \sum_{f \in F'} w_{a'} - \sum_{f \in F} w_a\Big] \cdot e_b(f)\,, \forall b, \forall f. \tag{4.4}$$

$w_b(f)$ ist hierbei das Äquivalent zu $Q(s, a)$ aus Gleichung 4.2. Der Index $b \in \mathcal{A}$ identifziert eine diskrete Aktion während $f \in \{1, ..., |\phi|\}$ als der Index des f-ten Elements des Feature-Vektors ϕ zu verstehen ist. w lässt sich also als Matrix der Dimension $|\mathcal{A}| \times |\phi|$ reeller Zahlen interpretieren. α beschreibt weiterhin die Schrittweite, während c eine frei wählbare Konstante ist. Der Ausdruck $\sum_{f \in F} w_a$ berechnet für die zuletzt durchgeführte Aktion a ausgehend von dem Zustand, der durch den Feature-Vektor F repräsentiert wird, die Güte dieser Aktion. Die Feature-Vektoren F in diesem Paper sind binäre Features [Sut96]. Das bedeutet, dass die Summe über die $f \in F$ von w_a als die Summe der a-ten Zeile der Matrix w zu verstehen ist, wobei nur die Elemente dieser Zeile summiert werden, deren Index f Index eines Elements des Feature-Vektors F ist, das den Wert `wahr` inne hat.

Der Ausdruck $\sum_{f \in F'} w_{a'}$ berechnet demnach die Güte der als nächstes auszuführenden Aktion a' ausgehend von dem neuen Zustand, der durch den Feature-Vektor F' approximiert wird.

Die Ähnlichkeit der Gleichung 4.4 zur Gleichung 4.2 ist offensichtlich. Hieraus lässt sich die Aktualisierungsfunktion für ein Generalisiertes Q-Learning ableiten. Sie hat die folgende Form:

$$w_b(f) \leftarrow w_b(f) + \frac{\alpha}{c}\Big[r + \max_{a'} \sum_{finF'} w_{a'} - \sum_{f \in F} w_a\Big] \cdot e_b(f), \forall b, \forall f. \tag{4.5}$$

$max_{a'} \sum_{f in F'} w_{a'}$ ist demnach analog zu 4.3 der höchste Güte-Wert aller Aktionen a', der berechnet wird, indem pro Aktion a' all jene Elemente der a'-ten Zeile der Matrix w aufsummiert werden, deren Index f gleichzeitig der Index eines binären Features des Feature-Vektors F' ist, das den Wert `wahr` inne hat.

Doch die in dieser Arbeit verwendete Zustandsapproximationsfunktion, die Radiale-Basis-Funktion aus Gleichung 4.1, ist nicht-binär. Wie lässt sich dieser Umstand mit dem Ausdruck $\sum_{f \in F} w_a$ in Einklang bringen? Der binäre Feature-Vektor F aus dem Paper [Sut96] besteht offensichtlich aus Elementen, die die Werte $\{\texttt{wahr}, \texttt{falsch}\}$, oder auch $\{1, 0\}$ annehmen können. Die Summe aller Elemente der a-ten Zeile der Matrix w, deren Index f gleichzeitig Index eines binären Features ist, das den Wert `wahr` inne hat, ist im Grunde nichts anderes als die Summe der Elemente der a-ten Zeile der Matrix w, multipliziert mit F_f. Der Ausdruck kann also auch wie folgt geschrieben werden:

$$\sum_{f=1}^{|F|} w_a(f) \cdot F_f \tag{4.6}$$

Wird nun diese binäre Feature-Vektor F durch den in dieser Arbeit verwendeten nicht-binären Feature-Vektor ϕ ersetzt, ergibt sich der folgende Ausdruck:

$$\sum_{i=1}^{|\phi|} w_a(i) \cdot \phi_i = w_a \cdot \phi \tag{4.7}$$

4.3.2 Der Belohnungswert r

Der Belohnungswert r beschreibt die Rückmeldung der Umgebung an den Reinforcement Learning-Agent für das Ausführen einer bestimmten Aktion ausgehend aus einem bestimmten Zustand. Wie in Kapitel 2.2 beschrieben, hat der Reinforcement Learning-Agent das Ziel, die Summe der Belohnungswerte über die Zeit zu maximieren.

Intuitiv lässt sich nachvollziehen, dass große positive Werte für r dem Reinforcement Learning-Agent nahelegen, dass er eine gute Entscheidung getroffen hat, große negative Werte hingegen implizieren eine schlechte Wahl einer auszuführenden Aktion. Offensichtlich lässt sich das Verhalten des Reinforcement Learning Agents durch die Wahl der Belohnungsfunktion, also der Funktion, die jeder Aktion einen Belohnungswert zuorndet, beeinflussen. Eine gute Wahl für eine solche Funktion ist also von großer Bedeutung.

Da in dieser Arbeit das Problem der Laufzeitoptimierung gelöst werden soll, ist es naheliegend, r in Bezug zur Laufzeit des zu optimierenden Algorithmus zu setzen. Große r sollen andeuten, dass die Auswahl der letzten Aktion (Konfiguration) ausgehend von dem letzten Zustand (Indikatoren der Eingabedaten) eine kurze Laufzeit, kleine r hingegen sollen eine hohe Laufzeit andeuten, also dem Reinforcement Learning Agent mitteilen, dass die gewählte Konfiguration für die Indikatoren der Eingabedaten keine gute Wahl gewesen ist. Da die Laufzeitoptimierung *online*, also ohne Training oder Wissen im Vorfeld stattfinden soll, ist es nicht möglich, zu entscheiden, ob eine ermittelte Laufzeit nun gut oder schlecht zu bewerten ist. Wir können also nicht vorab bestimmen, ab welchem Schwellwert eine Laufzeit d (z.B. angegeben in Millisekunden) ein gutes oder ein schlechtes Ergebnis der gewählten Aktion ist. Daher wurde die Belohnungsfunktion, die eine gemessene Laufzeit in den Belohnungswert r übersetzt, wie folgt definiert:

$$r := \frac{1}{d} \tag{4.8}$$

Für hohe Laufzeiten d wird r klein, bleibt aber immer größer null. Für geringe Laufzeiten wird r groß. Je größer r, desto höher ist demnach die Güte der gewählten Aktion einzuschätzen. Je höher die Güte $Q(s_t, a_t)$ ausgehend von einem Zustand s_t ist, desto höher ist die Wahrscheinlichkeit, dass diese Aktion ausgewählt wird.

Sonderfall EmbreeBVH

Ein Sonderfall tritt allerdings bei der Anwendung im Raytracing-Kontext unter Zuhilfenahme der Embree-BVH-Bibliothek auf: Die beiden BVH-Parameter `minLeafSize` und `maxLeafSize` sind, wie die Namen schon vermuten lassen, von einander abhängig: So verweigert die Bibliothek den Bau der BVH, wenn nicht `minLeafSize` < `maxLeafSize` gilt. Dass *librltuning* eine solche Konfiguration, die diese Regel verletzt, a priori von der Auswahl ausschließt, kann derzeit nicht garantiert werden. Dieses Problem wurde dadurch umgangen, dass, sollte die Embree-BVH-Bibliothek eine Parameterkonfiguration ablehnen, dem Reinforcement Learning-Agent direkt ein Wert kleiner Null für r als Rückmeldung gegeben werden kann, sodass eine solche Konfiguration zumindest in Zukunft mit sehr hoher Wahrscheinlichkeit nicht erneut ausgewählt wird.

4.3.3 Die eingesetzte Strategie

Die Strategie (engl. "policy") beschreibt die Logik, mithilfe derer der Reinforcement Learning-Agent ausgehend von einem bestimmten Zustand die Aktion auswählt, die ihn in den nächsten Zustand überführen soll, mit dem Ziel, die Belohnung zu maximieren.

Die in dieser Arbeit verwendete Strategie ist eine ϵ-gierige Strategie (engl. „ϵ-greedy policy“). Eine solche Strategie entscheidet pro Zeitschritt, ob sie Exploitation oder Exploration betreibt, um die nächste Aktion auszuwählen. Das ϵ dient hierbei als die Wahrscheinlichkeit, dass in diesem Zeitschritt Exploration als Strategie verwendet werden soll. $1 - \epsilon$ ist demnach die Wahrscheinlichkeit, dass der Reinforcement Learning-Agent Exploitation einsetzt, also mithilfe einer gierigen Strategie (engl. "greedy") eine Aktion auswählt.

Eine gierige Strategie wählt zu jedem Zeitpunkt t die Aktion a_t ausgehend von einem Zustand s_t aus, die die unmittelbar höchste Belohnung verspricht. Die gierige Strategie wählt also eine solche Aktion a aus, für die gilt:

$$\arg\max_{a} (w_a \cdot \phi) \tag{4.9}$$

Wie in Kapitel 2.2.1 beschrieben, wählt eine typische ϵ-gierige Strategie im Explorationsfall per Zufall gleichverteilt eine Aktion aus dem Aktionsraum aus und führt diese aus. Doch in dieser Arbeit soll stattdessen der Nelder-Mead-Algorithmus der *libtuning*-Bibliothek gestartet werden, um eine geeignete Aktion auszuwählen.

4.4 Initialisierung

In diesem und dem folgenden Abschnitt wird erläutert, wie eine Erweiterung der Auto-Tuning-Bibliothek *libtuning* um generalisiertes Reinforcement Learning umzusetzen ist, um sie im Anschluss zur Laufzeitoptimierung einsetzen zu können. Das Ziel ist es, eine Schnittstelle für den Konsumenten dieser Bibliothek anzubieten, die der von *libtuning* ähnelt. In diesem Abschnitt wird näher auf die Anforderungen eingegangen, die eine Implementierung der Initialisierung erfüllen muss.

Auch hier soll es eine Tuner-Entität geben, die mit den benötigten Parametern initialisiert wird, die für die Laufzeitoptimierung verwendet wird. Im folgenden wird auf diese Entität mit `CATuner` Bezug genommen. Wie bei der Implementierung dieser Entität vorgegangen

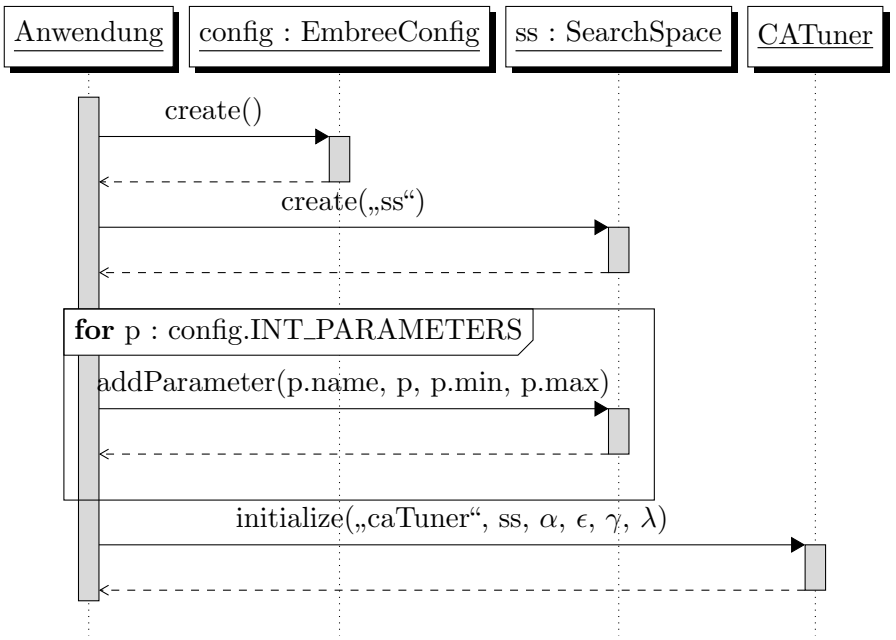

Abbildung 4.1: Sequenzdiagramm zur Darstellung der Initialisierung von CATuner

wurde, wird in Kapitel 5 näher erläutert. Ähnlich wie bei der Initialisierung von `Tuner` aus *libtuning* wird auch `CATuner` mit einem String als Name und der Referenz auf eine Instanz der Klasse `SearchSpace` aus *libtuning* initialisiert. Da `CATuner` im Gegensatz zu `Tuner` eine Implementierung von Reinforcement Learning beinhaltet, muss `CATuner` zusätzlich mit den vier Reinforcement Learning-Parametern α, ϵ, γ und λ parametrisiert werden. Eine typische Initialisierung im Kontext der Laufzeitoptimierung von Raytracing wird in Abbildung 4.1 illustriert. Um `CATuner` zu initialisieren, muss zuvor eine Instanz der Klasse `SearchSpace` aus dem Namespace `Tuning` erzeugt werden. Außerdem muss dieser Instanz die Liste der Parameter, die mithilfe von `CATuner` optimiert werden sollen, in diesen Suchraum aufgenommen werden. Abbildung 4.1 zeigt eine typische Initialisierung der benötigten Objekte für die anschließende Laufzeitoptimierung eines Raytracing-Auftrags. Die zu optimierenden Parameter entstammen einer Instanz der Klasse `EmbreeConfig`. Diese Instanz wird dazu verwendet, den Bau der räumlichen Datenstruktur (BVH) zu parametrisieren. Die `for`-Schleife in Abbildung 4.1 illustriert, wie die einzelnen Parameter der `EmbreeConfig`-Instanz `config` der Suchrauminstanz bekannt gemacht werden. Neben der Referenz auf den jeweiligen Parameter von `config` und dem Namen des entsprechenden Parameters müssen auch die Grenzen des Intervalls übergeben werden, innerhalb dessen sich der Wert des Parameters befinden muss.

Einen ähnlichen Ansatz verfolgte auch schon Andre Wengert in seiner Masterarbeit „Adaptives Auto-Tuning“ [Wen16], in der er ebenfalls einen Reinforcement Learning-unterstützten Auto-Tuner als Erweiterung von *libtuning* entwickelt hat. Der Unterschied zu der vorliegenden Arbeit ist der, dass Wengert Reinforcement Learning für diskrete Zustandsräume eingesetzt hat. Für die Initialisierung bedeutet das, dass `CATuner` kein Indikatorraum bekannt gemacht werden muss. Stattdessen muss für `CATuner` eine Funktion `setState` existieren, über die dem Reinforcement Learning-Agent, der in `CATuner` beheimatet ist, der gegenwärtige Zustand der Umgebung mitgeteilt werden kann. Im Anwendungskontext von Raytracing bedeutet das, dass dem Agenten für jeder neue Frame, der gerendert werden soll, ein Indikatorvektor $\mathcal{I}$ mithilfe dieser `setState`-Funktion mitgeteilt wird. Dieser Vektor muss für jedes Element mindestens drei Elemente besitzen: Einmal den Wert, den der entsprechende Indikator für diesen Frame innehat, zum anderen das jeweilige Minimum und Maximum, das der Wert des Indikators annehmen kann. Der letzte Aufruf, der

in Abbildung 4.1 dargestellt ist, zeigt schlussendlich die Initialisierung von `CATuner`. Der Aufruf beinhaltet den Namen des Tuners, die Referenz auf den Suchraum sowie die vier Reinforcement Learning-Parameter.

4.5 Ablauf der Optimierung

librltuning soll auf ähnliche Art und Weise wie *libtuning* zur Laufzeitoptimierung verwendet werden. In diesem Abschnitt wird der Vorgang genauer beleuchtet.

Wie in Kapitel 2.3 beschrieben operiert *libtuning* innerhalb einer Tuning-Schleife. Im Rumpf dieser Schleife wird von den `start()`- und `stop()`-Methodenaufrufen an die `Tuner`-Instanz umgeben die parametrisierte `Hotspot`-Funktion aufgerufen. Über die Schleifeniterationen und somit wiederholten Aufrufe an der `Hotspot`-Funktion hinweg soll so durch Beobachten der Laufzeit selbige durch geeignete Wahl der Parameter verkürzt werden. Diese Funktionen müssen also auch für `CATuner` existieren und eine vergleichbare Funktionalität bereitstellen. Da *librltuning* einen Reinforcement Learning-Ansatz mit einer ϵ-gierigen Strategie implementiert, und für die Exploration den in *libtuning* implementierten Nelder-Mead-Algorithmus verwendet, muss auch `CATuner` über die Aufrufe von `start` und `stop` im Falle der Exploration dieselbe Funktionalität wie die entsprechenden Aufrufe an `Tuner` bereitstellen.

Desweiteren muss es eine Funktion neben `setState` geben, die zu Beginn eines Zeitschritts t aufgerufen wird, und den gegenwärtigen Zustand der Umgebung dem Reinforcement Learning-Agenten mitteilt, die zum Ende des Zeitschritts t aufgerufen wird und die die Aktualisierung des Gewichtungsvektors $w_a(f)$ durchführt. Diese soll die Evaluationsfunktion `eval` sein.

Es ist leicht nachzuvollziehen, dass mit jedem Zeitschritt t, indem auch automatisch ein Zustandsübergang vollzogen wird, die Tuning-Schleife von vorne beginnen muss. Dies kann zum Beispiel durch die Verschachtelung zweier Schleifen umgesetzt werden. Im Anwendungsfall Raytracing bedeutet das, dass die äußere der beiden Schleifen über die Liste der zu zeichnenden Frames iteriert, während die innere Schleife über die Rendering Samples eines solchen Frames iteriert. Der Zustand des Reinforcement Learning-Agenten ändert sich mit jeder Iteration der äußeren Schleife. Dieses Anwendungsszenario wird in Abbildung 4.2 dargestellt. Sie zeigt die Aufrufe des Konsumenten der *librltuning*-Bibliothek, sowie dessen Verhalten im Explorations- bzw. Exploitation-Fall.

Zu Beginn eines jeden neuen Schleifendurchlaufs der äußeren Schleife wird `setState` aufgerufen und dem Reinforcement Learning-Agenten der gegenwärtige Zustand bekannt gemacht. $\mathcal{I}_s$ stellt in diesem Sequenzdiagramm den Indikatorvektor dar, der ausgehend von Zustand `s` erzeugt wurde. Außerdem muss hier festgelegt werden, ob die nächste Aktion durch Exploitation oder Exploration ausgewählt werden soll. Exploration wird zufällig mit der Wahrscheinlichkeit ϵ gewählt. Tritt der Fall Exploitation ein, so wird über die Funktion, die in Gleichung 4.9 beschrieben ist, die Aktion mit der unmittelbar höchsten zu erwartenden Belohnung ausgewählt. Tritt dagegen der Explorationsfall ein, so muss in den folgenden Aufrufen von `start` und `stop` die Exploration des Nelder-Mead-Algorithmus durch die entsprechenden Aufrufe an die `Tuner`-Instanz durchgeführt werden. Dies geschieht zu Beginn beziehungsweise am Ende jeder Tuning-Schleifeniteration. Befindet sich die `CATuner` hingegen im Exploitation-Modus, wird die gierig ausgewählte Aktion angewandt und die so ausgewählte Suchraumkonfiguration bleibt über alle Tuning-Schleifeniterationen konstant und ändert sich gegebenenfalls erst wieder beim nächsten Aufruf von `setState`.

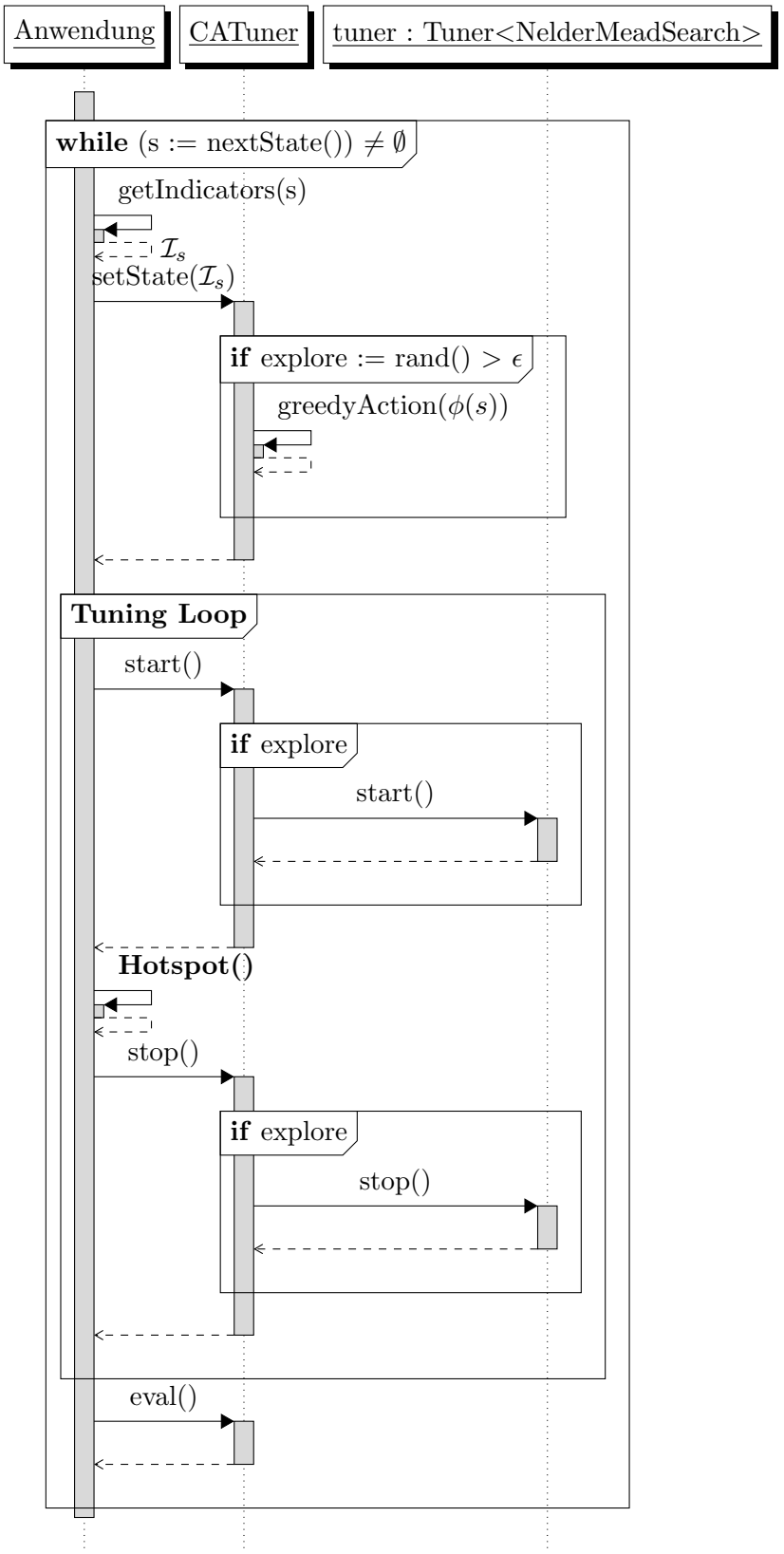

Abbildung 4.2: Sequenzdiagramm einer typischen Optimierungsschleife

5. Implementierung

In diesem Kapitel wird auf die Implementierungsdetails zu dem in Kapitel 4 vorgestellten Entwurf der Laufzeitoptimierungsbibliothek *librltuning* eingegangen. Das Kapitel Implementierung befasst sich damit, wie das Reinforcement Learning-Verfahren Q-Learning auf Generalisierten Zustandsräumen implementiert wurde.

5.1 Überblick

librltuning wurde als Erweiterung der Auto-Tuning-Bibliothek *libtuning* in der Programmiersprache C++14 umgesetzt. Diese Erweiterung befindet sich im Namensraum `Tuning::Adaptive::Continuous`. Der Kern dieser Arbeit ist die Umsetzung eines Generalisierten Q-Learnings auf kontinuierlichen Zustandsräumen und wurde in den beiden Klassen `CATunerQ` und `CATunerQ2` implementiert. Diese beiden Klassen dienen als Alternative zu der `Tuner`-Klasse aus dem Namensraum `Tuning` der Bibliothek *libtuning* und setzen die Anforderungen an eine Implementierung von `CATuner` aus den Kapiteln 4.4 und 4.5 um. Durch eine Benutzung einer der beiden Klassen `CATunerQ(2)` anstelle von `Tuner` kann somit die Reinforcement-Learning-Implementierung zur Laufzeitoptimierung verwendet werden.

Die beiden Klassen `CATunerQ` und `CATunerQ2` implementieren beide den selben in Kapitel 4 vorgestellten Algorithmus für ein generalisiertes Q-Learning, doch sie unterscheiden sich in der Art und Weise, wie der in Kapitel 4.3.1 vorgestellte Gewichtungsvektor $w_a(f)$ implementiert worden ist. Dies hat Auswirkungen darauf, wie die Abbildungsfunktion, die den Aktionsraum des Reinforcement Learning Agents auf den Suchraum der Klasse `Tuner` der `libtuning`-Bibliothek abbildet, sowie auf ihre Umkehrfunktion. Eine genauere Betrachtung der Unterschiede dieser beiden Klassen ist in Kapitel 5.3 zu finden.

Im Folgenden wird dann auf beide Klassen mit `CATunerQ(2)` Bezug genommen, wenn Sachverhalte beschrieben werden, die auf beide Klassen `CATunerQ` und `CATunerQ2` in gleicher Weise zutreffen.

Die Abbildung 5.1 zeigt in Form eines Klassendiagramms eine Übersicht über die verwendeten Klassen und wie sie untereinander in Bezug stehen. Da, wie in Kapitel 4 beschrieben, `CATunerQ(2)` im Explorationsfall der verwendeten ϵ-gierigen-Strategie den in *libtuning* implementierten Nelder-Mead-Algorithmus verwendet, um eine neue, aus Sicht des Reinforcement Learning Agents unbekannte Aktion auszuwählen, verwaltet `CATunerQ(2)` eine Instanz der Klasse `StopwatchTuner<NelderMeadSearch>` aus dem Namensraum `Tuning`.

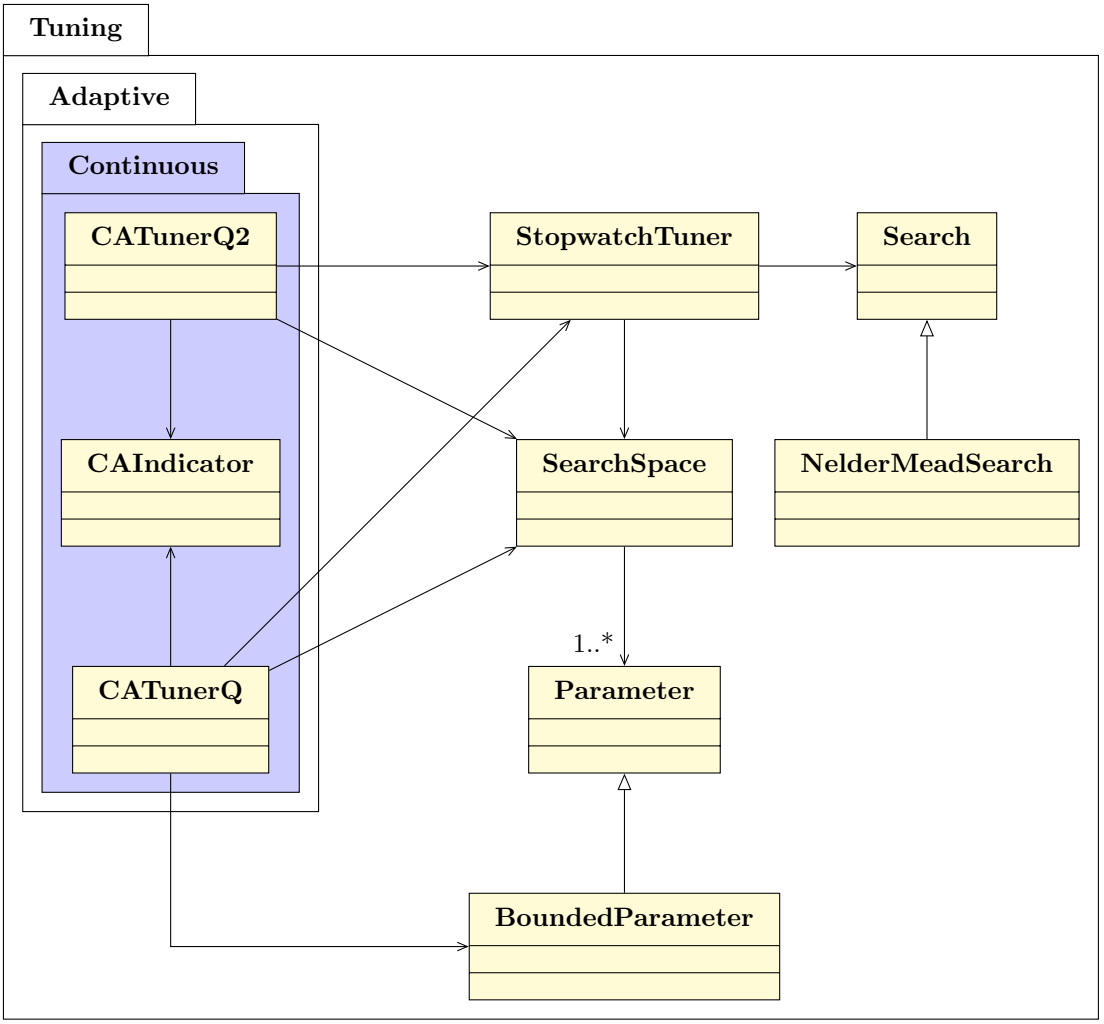

Abbildung 5.1: Übersicht Klassendiagramm

Beide Klassen verwalten eine Referenz auf die selbe Instanz einer Suchraumklasse `SearchSpace` aus dem Namensraum `Tuning`.

Die Klasse `CAIndicator` implementiert einen Indikator. Ein solcher Indikator ist ein Element des in Kapitel 4.1 beschriebenen Zustandsvektors des entsprechenden Zustandsraums. Da für die Berechnung des Feature-Vektors ϕ unter Anwendung einer Radialen Basis-Funktion (siehe Kapitel 4.1) zur Berechnung des Mittelpunktes c_i der Gaußglocke eines Features ϕ_i der Mittelwert des Wertebereichs eines jeden Indikators berechnet werden muss, besitzt die Klasse `CAIndicator` drei Eigenschaften: Den Wert dieses Elements, sowie dem bekannten Minimum und Maximum, den dieser Wert für diesen bestimmten Indikator annehmen kann. Diese Tatsache erfordert es, dass im Vorfeld des Produktiveinsatzes Wissen über die Eigenschaften sämtlicher zu verarbeitender Eingabedaten vorhanden sein muss. Die Implementierung von `CATunerQ` bedingt, dass für die Abbildungsfunktionen von Suchraum auf Aktionsraum und umgekehrt die Instanz von `CATunerQ` direkt auf die einzelnen Intervallparameterinstanzen zugreift, die Teil der `SearchSpace`-Instanz sind.

5.2 Initialisierung

Um mithilfe der *librltuning*-Bibliothek eine Laufzeitoptimierung einer parametrisierten `Hotspot`-Funktion auf einem Reihe ähnlicher Eingabedaten vorzunehmen, wird eine Instanz der Klasse `CATunerQ(2)` benötigt. In Abbdildung 5.2 sind die öffentlichen Methoden dieser Klasse dargestellt, die für den Einsatz zur Laufzeitoptimierung benötigt werden.

Konstruktion

Um eine Instanz der Klasse `CATunerQ(2)` zu erzeugen, benötigt es mindestens einen Tuner-Namen, eine Instanz eines Suchraums, in dem der Tuner gegeben eines Zustands eine optimale Konfiguration finden kann, sowie die Reinforcement Learning-Parameter α, ϵ, γ und λ. Die letzten beiden Parameter sind optional. Sie werden standardmäßig mit null initialisiert, da bei der Laufzeitoptimierung für die Auswahl einer Aktion in Abhängigkeit eines Zustands nur der Wert des gegenwärtigen Zustands von Interesse ist.

Ein weiterer Konstruktor dieser Klasse erlaubt eine spezifischere Anpassung des Verhaltens des Reinforcement Learning-Tuners. So kann statt eines konstanten Werts für ϵ ein initialer und ein minimaler Wert festgelegt werden, sowie eine Diskontierungsrate, mit der der ϵ-Wert bei jedem Aufruf der Methode `setState(·)` reduziert wird, bis er den minimalen ϵ-Wert `epsilonMin` angenommen hat.
Desweiteren kann über den Parameter `accuTraces` festgelegt werden, ob es sich bei den *Eligibility Traces* um *accumulating traces* oder *replacing traces* handeln soll, indem der Wert auf `true` beziehungsweise `false` gesetzt wird. Standardmäßig werden *accumulating traces* verwendet.
Wird für den Parameter `bootstrapRuns` ein Wert $b > 0$ gesetzt, so wird der `CATunerQ(2)` für die ersten b Aufrufe der Methode `setState(·)` den aktuellen Wert von ϵ ignorieren und stattdessen eine Aktion durch Exploration auswählen. Für ein $b = 0$ wird `CATunerQ2` dennoch in der ersten Iteration Exploration betreiben. Die Gründe hierfür sind in Kapitel 5.3 dargelegt.

setState(·)

Mit der Methode `setState(s)` kann dem Reinforcement Learning Agent der Klasse `CATunerQ(2)` der aktuelle Zustand mitgeteilt werden. Dies ist nötig, da für diese Implementierung des Reinforcement Learnings der Agent den aktuellen Zustand nicht selbst herausfinden kann. Auf Grundlage des aktuellen Zustands hat der Reinforcement Learning Agent nun die Aufgabe, entweder durch Exploration oder Exploitation eine Aktion auszuwählen. Eine Aktion aus Sicht des Reinforcement Learning Agents ist nichts anderes als ein Suchraumparameter mit bestimmten Werten. Diese Werte sind die Parameter, mit denen die `Hotspot`-Funktion parametrisiert wird, die die gegebenen Eingabedaten verarbeitet und deren Laufzeit durch geeignete Wahl solcher Parameter in Abhängigkeit des aktuellen Zustands optimiert werden soll.

Ein typischer Anwendungsfall für den Aufruf der Methode `setState` ist zu Beginn einer Schleife, die über eine Liste ähnlicher Eingabedaten iteriert, wobei der Parameter des Funktionsaufrufs der Indikatorvektor des Eingabedatums ist.

Die Methode hat eine Überladung, die einen weiteren Parameter vom `bool` annimmt, mit dem `CATunerQ(2)` mitgeteilt werden kann, dass er die zufallsbasierte Entscheidung darüber, ob er in diesem Aufruf Exploration oder Exploitation betreiben soll, ignorieren und stattdessen Exploitation einsetzen soll. Dies ist in Kombination mit dem anschließenden Nichtaufrufen der Methode `eval()` nützlich, wenn man den `CATunerQ(2)` in einen Modus versetzen möchte, in dem er nur auf Basis von Erfahrungen Aktionen auswählen soll, die er bis zu diesem Zeitpunkt gemacht hat und dabei seinen Erfahrungsschatz nicht erweitern soll.

eval()

Der Aufruf der Methode `eval` lässt `CATunerQ(2)` lernen. So werden die Eligibility Traces aktualisiert sowie beim nächsten Aufruf von `setState(·)` der Gewichtungsvektor $w_a(f)$, der den Erfahrungsschatz des Reinforcement Learning Agents darstellt, gemäß Gleichung

4.5 angepasst. Ein Nichtaufrufen nach Abschluss eines Zeitschritts t führt dazu, dass der Belohnungswert r_{t+1}, den die Umgebung dem Reinforcement Learning Agent nach Ausführen der Aktion a_t mitteilt ignoriert wird und die Aktualisierungsfunktion des Q-Learnings im Zeitschritt $t + 1$ nicht ausgeführt wird.

start()

Diese Methode startet die Zeitmessung. Sie wird innerhalb der Tuning-Schleife direkt vor Aufruf der `hotspot(·)`-Funktion aufgerufen. Wenn sich `CATunerQ(2)` im Explorationsmodus befindet, wird der Aufruf an die StopwatchTuner-Instanz, die von `CATunerQ(2)` verwaltet wird, weitergeleitet.

stop(·)

Direkt im Anschluss an den Aufruf der `hotspot`-Funktion wird `stop` aufgerufen. Dadurch wird die interne Zeitmessung von `CATunerQ(2)` gestoppt und der Belohnungswert r gemäß der Gleichung 4.8 berechnet. Außerdem wird der Aufruf im Explorationsfall die StopwatchTuner-Instanz, die von `CATunerQ(2)` verwaltet wird, weitergeleitet.

Wird der überladenen Methode eine Gleitkommazahl `time_ms`, die die Dauer einer Messung in Millisekunden darstellt, als Parameter übergeben werden, so dient dies zum Überschreiben der internen Laufzeitmessung zwischen dem Aufruf von `start` und `stop`. Dieser Wert wird im Explorationsfall auch der `StopwatchTuner`-Instanz zum Überschreiben dessen eigener internen Messung übergeben. Dies hat den Hintergrund, dass die Belohnungswertabbildungsfunktion aus Gleichung 4.8 für gemessene Laufzeiten, die von Natur aus nichtnegativ sind, keine negativen Werte für den internen Belohnungswert r der `CATunerQ2`-Instanz ermöglicht. Die Übergabe einer negativen Zahl für die überschriebene Laufzeit `time_ms` würde dazu führen, dass auch die von `CATunerQ2` verwaltete `StopwatchTuner`-Instanz diesen Wert als Rückmeldung erhält. Dies ist aber aus offensichtlichen Gründen nicht sinnvoll, da Laufzeiten per definitionem nichtnegativ sind. Ein Belohnungswert für einen Reinforcement Learning-Agent kann allerdings negativ sein. Um `CATunerQ2` also als Rückmeldung mitzuteilen, dass eine Aktion sehr schlecht ist, kann mit dem ersten Parameter eine extrem große Zahl und mit dem zweiten eine negative Zahl übergeben werden. Dieser Speziallfall kommt im Anwendungskontext des Raytracing zum Einsatz, wenn die BVH-Implementierung der Embree-Bibliothek verwendet wird. Diese Implementierung erlaubt nicht jede mögliche Parameterkonfiguration aus dem Suchraum. Sollte also durch die `CATunerQ2`-Instanz eine solche illegale Parameterkonfiguration vorgeschlagen werden, wird Embree den Bau der BVH verweigern. In diesem Fall muss dem `CATunerQ2` eine solche Kombination aus sehr großem Wert für den ersten Parameter und ein negativer Wert für den zweiten Parameter des Aufrufs der `stop`-Methode übergeben werden. Das Übergeben des zweiten Parameters setzt direkt den internen Wert für r und umgeht die Berechnung gemäß Gleichung 4.8.

setReward(·)

Dies ist eine Setter-Methode zum manuellen Setzen des aktuellen Belohnungswerts r.

5.2.1 Beispielhafte Verwendung

Quelltextausschnitt 5.1 zeigt einen typischen Anwendungsfall für einen Reinforcement Learning-unterstützten Auto-Tuner. Gegeben ist eine Liste von Eingabedaten des Types `InputData` und eine Funktion mit dem Namen `hotspot`. Dies ist die parametrisierte Funktion, die die Eingabedaten verarbeiten soll und deren Laufzeit aus einer Kombination aus

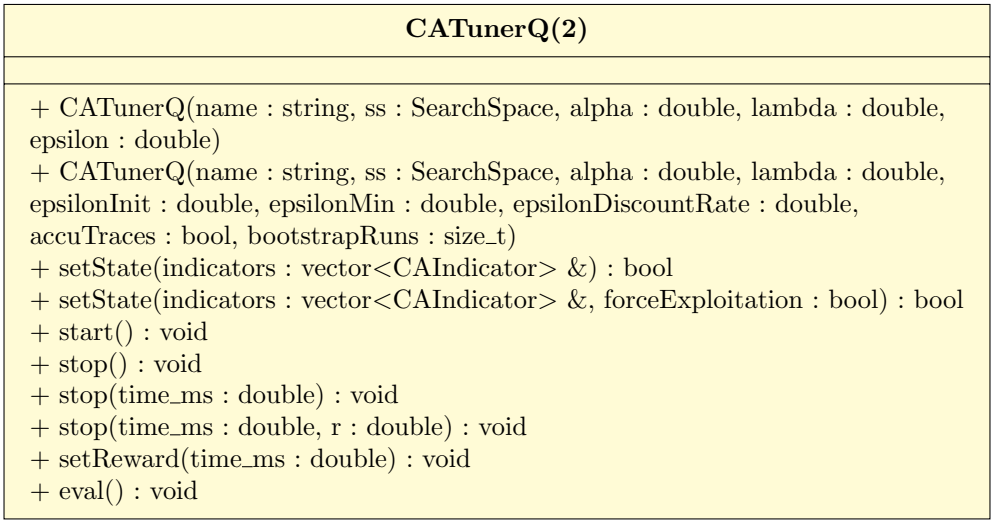

Abbildung 5.2: UML-Klassendiagramm von CATunerQ bzw. CATunerQ2

Eigenschaften des zu verarbeitenden Eingabedatums und den Werten der Parameter abhängt. Da die Eigenschaften des Eingabedatums unveränderlich sind, ist das Auto-Tuning-Problem, was in diesem Anwendungsfall gelöst werden soll, die Wahl solcher Werte für die Funktionsparameter, sodass die Laufzeit der `hotspot`-Funktion für dieses Eingabedatum minimal ist.

Um eine solche Laufzeitoptimierung durchzuführen, müssen zuerst die Parameter α, λ, γ und ϵ definiert werden (Zeile 3). Bei ihnen handelt es sich, wie in Kapitel 2.2 beschrieben, um die Parameter, die das Verhalten des Reinforcement Learning Agents steuern.

Als nächstes muss die Suchrauminstanz erzeugt werden. Aus diesem Suchraum wählt `CATunerQ(2)` die Parameterkonfigurationen aus, mit denen die `hotspot`-Funktion parametrisiert wird. Die Suchraumparameter werden in den Zeilen 6 und 7 definiert, die Suchrauminstanz in Zeile 10 erstellt und in den Zeilen 11 und 12 mit den Parametern bekannt gemacht. Da die in dieser Arbeit implementierte Version des Generalisierten Q-Learnings sich auf diskrete Aktionsräume beschränkt, sind im Suchraum nur ganzzahlige Intervallparameter erlaubt, weshalb jedem Aufruf von `addParameter` noch jeweils ein Wert für das Minimum und das Maximum übergeben wird, in deren Grenzen sich der Wert des Parameters befinden darf.

Die `CATunerQ(2)`-Instanz wird dann mit der Suchrauminstanz `ss`, sowie den Reinforcement Learning-Parametern in Zeile 15 erzeugt.

In einer Schleife (Zeile 18) werden nun die einzelnen Eingabedaten der Reihe nach verarbeitet. Da der Zustandsraum in diesem Fall identisch mit dem Indikatorraum der Eingabedaten ist, musste dieser im Vorfeld nicht definiert werden. Ein Indikator leitet sich von den Eigenschaften eines Eingabedatums ab. Der `CATunerQ(2)`-Instanz wird nun der aktuelle Zustand bekannt gemacht, indem die Methode `setState` mit dem Indikatorvektor des aktuell zu verarbeitenden Eingabedatums `d` aufgerufen wird. `caTuner` bestimmt nun entweder durch Exploration oder Exploitation eine Suchraumparameterkonfiguration, mit der die `hotspot`-Funktion für den nächsten Aufruf parametrisiert werden soll.

Die `hotspot`-Funktion wird nun in einer Schleife `TUNING_ITERATIONS`-mal mit demselben Eingabedatum, umgeben von den Aufrufen von `start()` und `stop()` aufgerufen. Befindet sich `caTuner` im Exploitation-Modus, so behalten die Suchraumparameter (hier `p_maxDepth` und `p_blockSize`) für die Dauer der Tuning-Schleife die Werte, die ihnen durch den Aufruf der `setState()`-Methode zugewiesen wurden. Befindet sich `caTuner`

```
1  void method(std::vector<InputData> data) {
2    // Reinforcement Learning-Parameter
3    double alpha = 0.1, lambda = 0.0, epsilon = 0.1;
4
5    // Suchraumparameter
6    int p_maxDepth = 3, p_maxDepth_min = 1, p_maxDepth_max = 7;
7    int p_blockSize = 42, p_blockSize_min = 24, p_blockSize_max = 64;
8
9    // Erzeuge den Suchraum
10   SearchSpace ss("");
11   ss.addParameter("p1", p_maxDepth, p_maxDepth_min, p_blockSize_max);
12   ss.addParameter("p2", p_blockSize, p_blockSize_min, p_blockSize_max);
13
14   // Erzeuge den Reinforcement Learning Tuner
15   CATunerQ caTuner("ca tuner q", ss, alpha, lambda, epsilon);
16
17   // Schleife über Eingabedatenliste
18   for (const InputData& d : data) {
19     std::vector<CAIndicator> indicators = d.getIndicators();
20
21     // Zustand des Tuners in Abhangigkeit des Eingabedatums d festlegen
22     caTuner.setState(indicators);
23
24     // Tuning-Schleife
25     for (size_t s = 0; s < TUNING_ITERATIONS; s++) {
26       // Tuning-Messung starten
27       caTuner.start();
28
29       hotspot(d, p_maxDepth, p_blockSize);
30
31       // Tuning-Messung beenden, Konfiguration und R anpassen
32       caTuner.stop();
33     }
34     // Aktualisieren des Gewichtungsvektors
35     caTuner.eval();
36   }
37 }
```

Quelltextausschnitt 5.1: Typische Verwendung der CATunerQ(2)-Schnittstelle zur Laufzeitoptimierung

im Explorationsmodus, so wird durch die Instanz von `StopwatchTuner`, die von `caTuner` verwaltet wird, eine Suchraumexploration gestartet und bei jedem Aufruf von `start()` eine neue Suchraumkonfiguration nach dem Nelder-Mead-Verfahren (siehe Kapitel 2.3.1) ausgewählt, bis dieses Verfahren konvergiert ist.

Da mit jedem Aufruf von `stop` auch der Belohnungswert r in `caTuner` gesetzt wird, gilt in beiden Fällen, dass nach Beendigung der Tuning-Schleife der Suchraum die zuletzt ausgewählte Suchraumparameterkonfiguration besitzt und der Belohnungswert, der für die Auswertung durch `caTuner` der entweder durch Exploration oder Exploitation ausgewählten Suchraumparameterkonfiguration und somit Anpassung des Gewichtungsvektors $w_a(f)$ notwendig ist, entsprechend der Laufzeit des letzten `hotspot`-Aufrufs gesetzt ist.

Durch den Aufruf der `eval`-Methode am Ende der Eingabedatenschleife in Zeile 35 wird `caTuner` mitgeteilt, den Gewichtungsvektor $w_a(f)$ gemäß der Gleichung 4.5 beim nächsten Aufruf von `setState` zu aktualisieren.

5.3 Suchraum und Aktionsraum

Während der Suchraum den Raum beschreibt, in dem der Nelder-Mead-Algorithmus nach einer optimalen Konfiguration zur Funktionsminimierung sucht, ist der Aktionsraum $\mathcal{A}(s_t)$ der Raum der Aktionen, die der Reinforcement Learning-Agent auswählen kann, wenn er sich zum Zeitpunkt t in Zustand s_t befindet. In unserem Fall der Reinforcement Learning-unterstützten Laufzeitoptimierung ist jede Aktion $a \in \mathcal{A}(s_t)$ äquivalent zu einer Parameterkonfiguration des Suchraums. Da bei der Laufzeitoptimierung jede Aktion ausgehend von jedem Zustand möglich ist, fällt die Abhängigkeit von s_t weg und der Aktionsraum lässt sich verkürzt mit $\mathcal{A}$ benennen. Der Aktionsraum $\mathcal{A}$ hat die Dimension 1. Man kann sich ihn als eine Liste von diskreten Aktionen vorstellen. Der Suchraum besteht in der Implementierung von *libtuning* aus einer Liste von Parameterobjekten. Da sich diese Arbeit auf die Implementierung von Reinforcement Learning auf disrketen Aktionsräumen beschränkt, sind für die Nutzung von *librltuning* nur solche Suchrauminstanzen erlaubt, die ausschließlich aus ganzzahligen Intervallparametern bestehen. Der Suchraum hat also die Dimension, die gleich der Anzahl der Parameter ist, die diesem Suchraum zugeordnet sind. Die Frage nach einer Abbildungsfunktion von Suchraum nach Aktionsraum und von Aktionsraum nach Suchraum hat die Entwicklung der beiden Reinforcement Learning-unterstützten Auto-Tuner-Implementierungen motiviert. So gibt es eine mehr oder weniger naive Implementierung, die dem PseudoCode aus der Arbeit von Sutton [Sut96] möglichst nahe kommt und eine, die dem Umstand geschuldet ist, dass in unserem Raytracing-Anwendungskontext mit dieser Implementierung zu viel Arbeitsspeicher für die Speicherung des Gewichtungsvektors $w_a(f)$ alloziert werden müsste (siehe Gleichung 5.1).

Im Folgenden wird auf die beiden Implementierungen der Darstellung des Suchraums für den Generalisierten Q-Learning-Algorithmus in den beiden Klassen `CATunerQ` und `CATunerQ2` im Detail eingegangen.

5.3.1 Suchraum in CATunerQ

Betrachtet man Gleichung 4.5 sowie die anderen Gleichungen 4.3 oder 4.2, so sieht man, dass die Aktion a immer als eine Art Index verwendet wird. Es existiert also eine diskrete Liste aller möglichen Aktionen, die über den Index a ausgewählt werden können. Gleichzeitig gibt dieser Index auch die Zeile des Gewichtungsvektors $w_a(f)$ an. Die a-te Zeile in w entspricht also den gelernten Gewichten für eine Aktion, die ebenfalls an a-ter Stelle in der Liste aller diskreten Aktionen steht. In `CATunerQ` existiert eine solche Liste aller diskreten Aktionen nur implizit. Und zwar in Form der Suchrauminstanz, auf der `CATunerQ` operiert. Es muss also eine Abbildungsfunktion geben, die, gegeben eine positive Ganzzahl

```
1 void setA(size_t a) {
2   for (size_t ip = 0; ip < ss.size(); ip++) {
3     auto *p = (BoundedParameter<int>*)ss[ip];
4     auto valCount = p->getMax() - p->getMin() + 1;
5     size_t iv = a % valCount;
6     p->setValue(iv);
7     a /= valCount;
8   }
9 }
```

Quelltextausschnitt 5.2: CATunerQ::setA(·)

a, die `SearchSpace`-Instanz in eine bestimmte Parameterkonfiguration versetzt. Dies ist das Auswählen der Aktion a. Das Anwenden der Aktion a ist also demnach das Ausführen der hotspot-Funktion, die mit den Parameterwerten parametrisiert ist, die dadurch gesetzt werden, dass der Suchraum in ebendiese Konfiguration versetzt wird.

setA(a)

Diese Abbildungsfunktion in `CATunerQ` heißt `setA(a)` und versetzt den Suchraum in die a-te Konfiguration. Das Verfahren sieht man in Quelltextausschnitt 5.2. Dies geschieht, indem jedem Intervallparameter des Suchraums ein bestimmter Wert zugewiesen wird. In einem festen Parameter a sind die Indizes der Werte der Parameter kodiert, die der entsprechende Parameter annehmen soll. Der Reihe nach wird für jeden Parameter p_i der Index des Wertes, den p_i annehmen soll, berechnet, indem der Rest der Division von a durch die Anzahl der Werte von p_i berechnet wird. Anschließend wird a durch die Anzahl der Werte geteilt. Der Rest der Division von a durch die Anzahl der Werte des nächsten Parameters p_{i+1} ist nun wieder der gesuchte Index des Wertes, den p_{i+1} annehmen soll.
Die Korrektheit dieses Verfahrens lässt sich intuitiv einsehen, wenn das folgende Beispiel herangezogen wird. Angenommen in dem Suchraum gebe es genau 3 ganzzahlige Intervallparameter, die alle exakt 10 Werte annehmen können. Offensichtlich ist die Anzahl der möglichen Aktionen $|\mathcal{A}| = 10^3$. Eine positive Ganzzahl a, die eine Suchraumkonfiguration kodiert, kodiert also für jeden der 3 Intervallparameter pro Dezimalstelle den Index des Wertes dieses Parameters. Sei also der Index des Wertes, den der i-te Parameter annehmen soll, I_i (für $i \in 0, 1, 2$). Dann wäre ein a, das eine solche Suchraumparameterkonfiguration kodiert, der Form

$$a = I_2 I_1 I_0,$$

wobei jedes I_i die i-te Ziffer der dreistelligen Zahl a ist. a lässt sich also durch die Summe

$$\sum_{i=0}^{2} I_i \cdot 10^i$$

berechnen. Um nun umgekehrt die i-te Ziffer der Zahl a zu erhalten, die den Index des Wertes des i-ten Parameters darstellt, muss a einfach i-mal durch 10 geteilt werden und aus dem Ergebnis der Modul der Division durch 10 berechnet werden. Analoges passiert in der Methode `setA(a)`, nur dass dort die Anzahl der Werte der Parameter nicht gleich 10 sind, sondern den Wert `valCount` des i-ten Parameters haben.

getA()

`getA()` ist die inverse Funktion zu `setA(a)`. Sie bildet eine Suchraumparameterkonfiguration auf eine positive Ganzzahl a ab. Das Verfahren hierzu ist in Quelltextausschnitt 5.3 zu sehen. Es verfährt nach der gleichen Idee, wie `setA(a)`. Für jeden ganzzahligen

```
1  size_t getA() {
2    size_t a = 0;
3    size_t n = ss.size();
4    for (size_t ip = 0; ip < n; ip++) {
5      size_t i = n - 1 - ip;
6      auto *p = (BoundedParameter<int>*)ss[i];
7      auto valCount = p->getMax() - p->getMin() + 1;
8      auto idx = p->get() - p->getMin();
9      a *= valCount;
10     a += idx;
11   }
12   return a;
13 }
```

Quelltextausschnitt 5.3: CATunerQ::getA()

Intervallparameter p_i des Suchraums wird a mit der Anzahl der Werte von p_i multipliziert (Zeile 8) und dann um den Index des aktuellen Werts in der Liste der Werte erhöht (Zeile 9). Zu beachten ist, dass die Schleife in dieser Methode rückwärts zählt.
Dies ergibt intuitiv Sinn, wenn man sich wieder das Beispiel des vorherigen Abschnitts vor Augen führt. Sei a also wieder die Kodierung der Konfiguration von 3 Parametern mit je 10 möglichen Werten. Dann entspricht jede der drei Dezimalstellen von a einem Parameterwertindex. Ziel ist also, dass I_2 an erster Stelle, I_1 an zweiter und I_0 an dritter Dezimalstelle von a steht. a kann also wie folgt berechnet werden:

$$a = ((I_2 \cdot 10) + I_1) \cdot 10 + I_0.$$

Dies entspricht exakt dem Verfahren, das in `getA()` angewandt wurde, mit dem einzigen Unterschied, dass die Parameter nicht zwangsläufig exakt 10 mögliche Werte annehmen können, sondern jeweils `valCount` viele.

5.3.2 Suchraum in CATunerQ2

Die naïve Implementierung des Aktionsraums in `CATunerQ` hat allerdings eine Limitierung. Und das ist die Größe des Arbeitsspeichers des ausführenden Computers. Diese Limitierung tritt speziell im Anwendungsgebiet Raytracing auf. Wie ja schon in Kapitel 4 angesprochen, wird die Reinforcement Learning-Unterstützte Auto-Tuning-Bibliothek mit dem Anwedungsgebiet der Laufzeitoptimierung eines typischen Raytracing-Anwendungsfalls, wie er zum Beipsiel in der Filmindustrie vorkommt, entwickelt. Das bedeutet, dass der Arbeitsauftrag für das Raytracing eine Folge mehrere Frames ist, die es zu zeichnen gilt. Diesem Anwendungsfall entsprang auch die Idee für die Evaluation dieser Reinforcement Learning-Implementierung. Als Raytracing-Bibliothek wird Intel Embree verwendet, einschließlich der BVH-Implementierung (ab hier „EmbreeBVH“ genannt) dieser Bibliothek. Die EmbreeBVH-Konstruktionsparameter werden also als die Parameter herangezogen, die das Rendering eines Frames parametrisieren sollen (Der Rendering-Aufruf eines Samples eines Frames entspricht der `hotspot`-Funktion). In Kapitel 4.2 wurden schon die einzelnen Parameter vorgestellt. Sie bilden den Suchraum. Das praktische Problem der Implementierung von `CATunerQ` ist nun, dass für jede mögliche Aktion aus dem Aktionsraum zwei Gleitkommazahlvektoren, deren Länge der Anzahl der Features ϕ_i entspricht, existieren müssen. Diese beiden Vektoren von Vektoren sind der Gewichtungsvektor $w_a(f)$ sowie die Eligibility Traces $e_a(f)$ (siehe Kapitel 4.3.1). Analog zu dem in Abschnitt 5.3.1 beschriebenen Beispiel berechnet sich die Anzahl aller möglichen Aktionen $|A|$ wie folgt:

$$|\mathcal{A}| = \prod_{i=1}^{n} |P_i|\,,$$

```
1 void CATunerQ2::setA(size_t a) {
2   Configuration C = this->knownConfigs[a];
3   this->ss.applyConfiguration(C);
4 }
```

Quelltextausschnitt 5.4: CATunerQ2::setA(·)

wobei $|P_i|$ die Anzahl der möglichen Werte, die der Parameter P_i annehmen kann, beschreibt. Für die Ganzzahlkonstruktionsparameter der EmbreeBVH „parallelMode“, „PacketMode_Count“, „ex_quality“, „ex_maxBranchingFactor“, „ex_maxDepth“, „ex_sahBlockSize“, „ex_minLeafSize“ und „ex_maxLeafSize“ sind jeweils die Anzahl der möglichen Werte in der Datei `SceneConstructionConfig.h` des *embree-autotune*-Projekts gegeben mit 4, 36865, 3, 64, 64, 33, 29 und 29. Das *embree-autotune*-Projekt wurde von Kevin Zerr im Rahmen seiner Masterarbeit „Laufzeitoptimierung mittels Autotuning von Path-Tracing-Datenstrukturen“ erstellt [Zer17]. Für die Evaluation der vorliegenden Arbeit wurde dieses Projekt leicht angepasst verwendet. Das Produkt dieser Zahlen ist $|\mathcal{A}| = 46879765954560$. Der Feature-Vektor ϕ hat nach Kapitel 4.1 eine Länge von 10. Da für jedes ϕ_i eine Gleitkommazahl mit doppelter Genauigkeit gespeichert wird, ergibt sich, dass eine Zeile in $w_a(f)$ sowie in $e_a(f)$ genau $10 \cdot 8Bytes = 80Bytes$ an Speicherplatz benötigt. Daraus ergibt sich eine benötigte Speicherkapazität für $w_a(f)$ und $e_a(f)$ von zusammen

$$2 \cdot |\mathcal{A}| \cdot 80 \text{ Bytes} = 7500762552729600 \text{ Bytes} \approx 6821,9 \text{ TiBytes} \tag{5.1}$$

Offensichtlich ist es also nicht praktikabel, für große $|\mathcal{A}|$ alle möglichen Aktionen zu speichern oder gar im Arbeitsspeicher zu halten.

Die Umgehungslösung für dieses Problem wurde in `CATunerQ2` implementiert. Hier werden nicht alle möglichen Aktionen gespeichert. Stattdessen werden nur die Aktionen gespeichert, deren Suchraumparameterkonfigurationsäquivalent mindestens einmal am Ende einer Exploration mithilfe der `StopwatchTuner`-Instanz, die von `CATunerQ2` verwaltet wird, ausgewählt worden ist. Dies wird umgesetzt, indem immer bei Aufruf der `eval`-Methode von `CATunerQ2` die aktuelle Suchraumkonfiguration in eine interne Liste von Konfigurationen kopiert wird, sollte eine identische Konfiguration in dieser Liste noch nicht existieren. Im gleichen Zug werden $w_a(f)$ und $e_a(f)$ um eine neue Zeile erweitert.

Die Methoden `setA(a)` und `getA()` gestalten sich entsprechend einfach: a ist nun nicht mehr der Index der Aktion innerhalb der Liste aller möglichen Aktionen A, sondern der Index der Suchraumkonfiguration innerhalb der Liste der dem `CATunerQ2` bekannten Suchraumkonfigurationen. Um eine Aktion a auszuwählen, muss entsprechend die a-te Konfiguration aus der Liste der bekannten Suchraumkonfigurationen auf den Suchraum angewandt werden. Dies wird in Quelltextausschnitt 5.4 dargestellt.

Der Index a der aktuellen Suchraumkonfiguration ist demnach der Listenplatz innerhalb der Liste der bekannten Suchraumkonfigurationen. Quelltextausschnitt 5.5 zeigt das Verfahren, um a zu berechnen.

5.4 Umsetzung des generalisierten Q-Learnings

Der Kern der Implementierung des generalisierten Q-Learnings besteht in der Aktualisierungsfunktion für $w_a(f)$ (siehe Kapitel 4.3.1). Bei jedem Aufruf (den ersten Aufruf ausgenommen) der Methode `setState` wird der Gewichtungsvektor $w_a(f)$ aktualisiert, wenn am Ende des letzten Zeitschritts die Methode `eval` aufgerufen wurde. Die für die Aktualisierung wichtigen Teile von `setState` sind in Quelltextausschnitt 5.6 zu sehen. Da

```
1  size_t CATunerQ2::getA() {
2    Configuration C = this->ss.getConfiguration();
3    auto it = std::find(this->knownConfigs.begin(), this->knownConfigs.
       end(), C);
4    size_t a = std::distance(this->knownConfigs.begin(), it);
5    return a;
6  }
```

Quelltextausschnitt 5.5: CATunerQ2::getA()

```
1  bool CATunerQ2::setState(std::vector<CAIndicator> &inds, ...) {
2    auto oldF = F;          // Feature-Vektor des Zeitschritts
       zwischenspeichern
3    F = features(inds);     // Indikatorvektor auf Feature-Vektor abbilden
4    if (firstRun) firstRun = false;
5    else {  // W aktualisieren
6      if (evalCalled) {
7        double D = R + gamma * maxW(this->F) - lineSumW(getA(), oldF);
8        for (size_t b = 0; b < W.size(); b++) {
9          for (size_t f = 0; f < W[b].size(); f++) {
10           W[b][f] += alpha * D * E[b][f];
11         }
12       }
13       evalCalled = false;
14     }
15   }
16   ...
17   isExploring = ...;
18   if (isExploring) {
19     tuner = ...; // StopwatchTuner neu setzen
20   } else {
21     // Aktion a mit gieriger Strategie auswählen
22     setA(greedyAction(F));
23   }
24   ...
25   return isExploring;
26 }
```

Quelltextausschnitt 5.6: Ausschnitt aus CATunerQ2::setState()

ein neuer Zeitschritt t mit dem Aufruf von `setState` eingeleitet wird und so der neue Zustand in Form des Indikatorvektors `inds` dem Reinforcement Learning-Agenten bekannt gemacht wird, muss zuvor der alte Zustand, der in der Instanzvariable F gespeichert ist, gesichert werden, bevor er in Zeile 3 aktualisiert wird. Dieser Feature-Vektor des letzten Zeitschritts wird für die Aktualisierung von $w_a(f)$ benötigt. Die Aktualisierung wird nur dann im Zeitschritt t durchgeführt, wenn am Ende des Zeitschritts $t-1$ die Methode `eval()` aufgerufen wurde. Denn dieser Aufruf setzt unter anderem die Instanzvariable `evalCalled`, die in Zeile 6 überprüft wird, auf `true`.

$w_a(f)$ und $e_a(f)$ werden in der `CATunerQ`-Implementierung als Instanz einer für diese Arbeit geschriebenen `Matrix`-Klasse, die auf einem `std::vector<double>` fester Länge operiert, umgesetzt. In der `CATunerQ2`-Implementierung wurde ein `std::vector<std::vector<double>>` verwendet, da eine dynamische Erweiterung der Zeilenanzahl notwendig ist, wie aus Kapitel 5.3.2 hervorgeht. Im Quelltextausschnitt werden $w_a(f)$ und $e_a(f)$ durch `W` beziehungsweise `E` repräsentiert. $w_a(f)$ wird in Zeile 11 gemäß Gleichung 4.5 aktualisiert. Der erste Index (a) ist die Zeile des Gewichtungsvektors und gibt für die `a`-te Aktion den Ge-

```
1 double CATunerQ2::maxW(std::vector<double> &v) {
2   double maxSum = 0.0;
3   for (auto &w_a : W) {
4     double s = std::inner_product(w_a.begin(), w_a.end(), v.begin(),
          0.0);
5     if (s > maxSum) maxSum = s;
6   }
7   return maxSum;
8 }
```

Quelltextausschnitt 5.7: Ausschnitt aus CATunerQ2::maxW()

```
1 double CATunerQ2::lineSumW(size_t a, std::vector<double>& v) {
2   double sum = std::inner_product(W[a].begin(), W[a].end(), v.begin(),
        0.0);
3   return sum;
4 }
```

Quelltextausschnitt 5.8: Ausschnitt aus CATunerQ2::lineSumW()

wichtevektor an. Wird das Skalarprodukt zwischen einem Gewichtevektor `W[a]` und einem Feature-Vektor `F`, der einen Zustand s_t in Zeitschritt t repräsentiert, erhält man für die Aktion `a` einen Wert, der die approximierte Güte dieser Aktion ausgehend aus dem Zustand s_t angibt.

`maxW` ist in Quelltextausschnitt 5.7 dargestellt. Diese Methode berechnet für alle `a` den höchsten Wert aller Resultate der Skalarprodukte aus `W[a]` und dem Feature-Vektor und gibt diesen zurück. `lineSumW` berechnet für die `a`-te Zeile das Skalarprodukt von `W[a]` und dem Feature-Vektor und gibt diesen Wert zurück. Der Eligibility Traces-Vektor $e_a(f)$ wird durch Aufruf der `eval()`-Methode in Abhängigkeit des zuletzt gesetzten Feature-Vektors F und in Abhängigkeit davon, ob es sich um *accumulating traces* oder *replacing traces* handelt, entsprechend aktualisiert.

5.4.1 Umsetzung der gierigen Strategie

librltuning verwendet eine ϵ-gierige Strategie. Das bedeutet, dass in jedem Zeitschritt t mit einer Wahrscheinlichkeit von $1 - \epsilon$ durch den Reinforcement Learning-Agenten Exploitation unter Einsatz einer gierigen Strategie eingesetzt wird. Diese „Entscheidung“ wird in Zeile 17 des Quelltextausschnitts 5.6 getroffen, in der die Instanzvariable `isExploring` unter Zuhilfenahme eines Pseudozufallsgenerators entweder auf `true` oder `false` gesetzt wird. Fällt der Wert auf `false`, wird Exploitation betrieben und die Methode `greedyAction` aufgerufen, die die gierige Strategie implementiert. Ihre Implementierung ist in Quelltextausschnitt 5.9 zu sehen. Es wird hierbei die Aktion ausgewählt, die zum gegenwärtigen Zeitpunkt den höchsten approximierten Nutzenwert in Abhängigkeit des Feature-Vektors des gegenwärtigen Zustands hat.

```
1  size_t CATunerQ2::greedyAction(std::vector<double>& F) {
2    size_t bestA = 0;
3    double bestSum = 0.0;
4    for (size_t a = 0; a < W.size(); a++) {
5      double sumA = std::inner_product(W[a].begin(), W[a].end(), F.begin
           (), 0.0);
6      if (sumA > bestSum) {
7        bestSum = sumA;
8        bestA = a;
9      }
10   }
11   return bestA;
12 }
```

Quelltextausschnitt 5.9: Ausschnitt aus CATunerQ2::greedyAction()

6. Evaluation

In diesem Kapitel wird der implementierte Reinforcement Learning Auto-Tuner *librltuning* auf seine Performanz hin getestet. Als Referenz dient die Auto-Tuning-Bibliothek *libtuning*. Als Vergleichsmetrik wird die Laufzeit eines im folgenden definierten Raytracing-Arbeitsauftrags herangezogen. Im Rahmen der Beantwortung der drei Forschungsfragen, die in Kapitel 6.2 vorgestellt werden, wird der Laufzeitvergleich in verschiedenen Szenarien durchgeführt, um so zu beurteilen, wie gut die Laufzeitoptimierung von *librltuning* im Vergleich zu *libtuning* ist.

Raytracing-Arbeitsauftrag

Der Raytracing-Arbeitsauftrag besteht aus einer Folge von neun 3D-Szenen, die jeweils aus zehn Kameraperspektiven gezeichnet werden. Diese neun 3D-Szenen werden in Kapitel 6.1 genauer betrachtet. Eine Kameraperspektive ist beschrieben durch einen sechsdimensionalen Vektor, der die Position und Orientierung einer virtuellen Kamera im dreidimensionalen Raum definiert. Diese Kameraperspektive definiert also den Bildausschnitt der 3D-Szene, der gezeichnet werden soll. Ein solches Bild wird auch *Frame* genannt. Zwei Frames der selben 3D-Szene, die aber aus verschiedenen Kameraperspektiven gezeichnet werden, können demnach verschiedene Laufzeiten für das Zeichnen haben.

Verwendete Arbeiten

Für das Rendering der Frames kommt eine Gabelung (engl. Fork) der Raytracing-Bibliothek „Intel Embree“[1] zum Einsatz. Dieser Fork mit dem Namen „embree-autotune“ wurde im Rahmen der Masterarbeit von Kevin Zerr am Karlsruher Institut für Technologie an der Fakultät für Informatik am Institut für Visualisierung und Datenanalyse entwickelt [Zer17]. Die in dieser Arbeit verwendete BVH-Implementierung „Intel EmbreeBVH“ ist Teil von Intel Embree und somit auch Teil der Arbeit von Zerr. Diese Version des Embree-Projekts erlaubt es dem Konsumenten der Raytracing-Bibliothek, Einfluss auf die BVH-Bauparameter zu nehmen, weswegen diese Version in der vorliegenden Arbeit zum Einsatz kommt. Es handelt sich hierbei um die Embree Version 2.17.0.

Die Zeitmessung

Ein Frame wird gezeichnet, indem zuerst für diesen Frame die BVH gebaut und anschließend eine bestimmte Menge an Samples dieses Frames per Raytracing erzeugt werden.

[1] Intel Embree: https://embree.github.io/

Testsystem	
Gastbetriebssystem	Ubuntu 18.04.1 64-Bit
Host-Betriebssystem	Windows 10 Education 64-Bit
Virtualisierungs-Software	VMWare Workstation 14 Player
Prozessor	Intel Core i5 4570
Gast: Kerne (Threads)	4 (4)
Gast-RAM	6 GB

Tabelle 6.1: Evaluations-Testsystem

Je mehr Samples für einen einzelnen Frame erzeugt werden, desto höher ist in der Regel die Bildqualität des resultierenden Bildes. Sämtliche Messungen zur Rendering-Laufzeit setzen sich aus der Zeit für den Aufbau der BVH und der Zeit für das eigentliche Zeichnen eines Samples zusammen. Um das Rauschen in den Messergebnissen beim Zeichnen der Samples zu reduzieren, wird für jede Laufzeitmessung eines einzigen Samples das Raytracing dieses Sample sechs mal durchgeführt. Von diesen sechs Zeitmessungen werden die ersten drei verworfen und über die letzten drei Zeitmessungen der Durchschnitt gebildet. Dieser Durchschnitt ergibt dann die Sample-Render-Zeit für dieses Sample. Das Verwerfen der ersten drei Messungen je Sample hat den Sinn, eventuell auftretende Cache-Effekte aus der Laufzeitmessung auszuschließen. Für das Rendering wurde eine Bildauflösung von 800 * 450 Pixel gewählt.

Testsystem

Für die Durchführung der Evaluation wurde das Testsystem aus Tabelle 6 verwendet.

6.1 Getestete Szenen

In diesem Abschnitt werden kurz die Szenen vorgestellt, die zur Evaluation von *librltuning* verwendet werden. Die Komplexität der Szenen reicht von 36 Dreiecken in der Szene Cornell Box bis hin zu 1,6 Millionen Dreiecken in der Szene Audi R8.

6.1.1 Audi R8

Abbildung 6.1: Render-Bild der Szene Audi R8

Die erste Szene ist eine Szene, die frei verfügbar auf der Plattform Free3D.com ist. Sie wurde von dem Benutzer „ahmetsalih“ am 29. Juli 2015 eingereicht [ahm15]. Sie zählt zu

den komplexen Szenen in dieser Auswahl mit 1,62 Millionen Dreiecken. Abbildung 6.1 zeigt ein mit Blender gezeichnetes Bild dieser Szene.

6.1.2 Bugatti Chiron 2017 Sports Car

Abbildung 6.2: Render-Bild der Szene Bugatti Chiron 2017 Sports Car

Die zweite Szene ist die Szene Bugatti Chiron 2017 Sports Car, kurz Bugatti. Sie ist ebenfalls auf der Plattform Free3D.com frei verfügbar. Sie wurde vom Benutzer „kimzauto“ am 27. August 2017 eingereicht [kim17]. Sie ist mit 1,45 Millionen Dreiecken ähnlich komplex wie die erste Szene. Abbildung 6.2 zeigt ein mit Blender gezeichnetes Bild dieser Szene.

6.1.3 Cornell Box

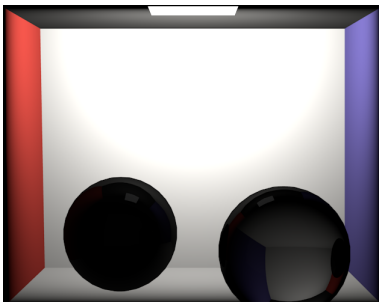

Abbildung 6.3: Render-Bild der Szene Cornell Box

Die Szene Cornell Box ist ein Klassiker unter den 3D-Szenen im Bereich der Computergrafik. Mit nur 36 Dreiecken ist die Szene sehr primitiv und stellt keine großen Anforderungen an ein Raytracing-System. Die Szene ist über die Website von Morgan McGuire frei zugänglich [McG17]. Abbildung 6.3 zeigt ein mit Blender erstelltes Bild dieser Szene.

6.1.4 Dabrovic Sponza

Abbildung 6.4: Render-Bild der Szene Dabrovic Sponza

Auch die Sponza-Szene wurde von der Website von McGuire bezogen [McG17]. Sie besteht aus 66 Tausend Dreiecken und siedelt sich so im unteren Viertel der Szenen bezogen auf die Komplexität an. Abbdildung 6.4 zeigt ein mit Blender gezeichnetes Bild dieser Szene.

6.1.5 House

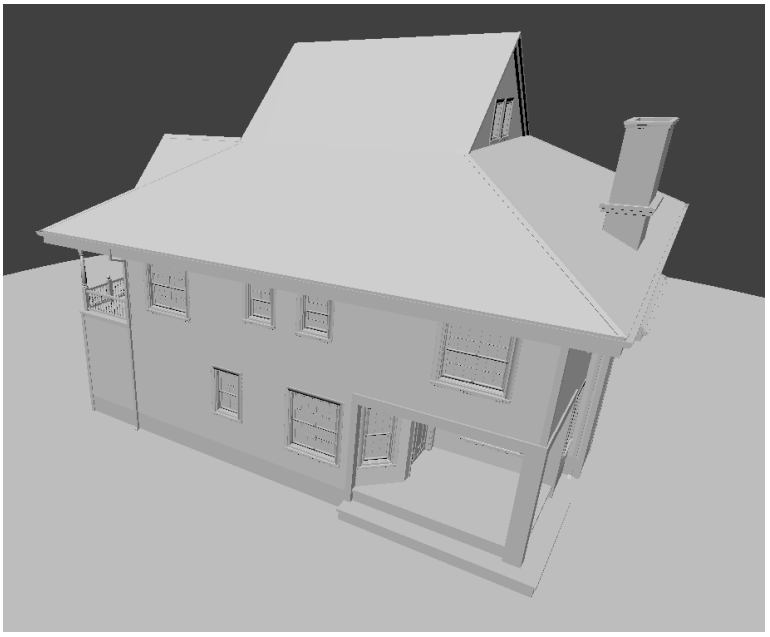

Abbildung 6.5: Render-Bild der Szene House

Die Szene House wurde schon in der Masterarbeit von Kevin Zerr [Zer17] verwendet und ist über die Seite von Benedikt Bitterli bezogen worden [hou]. Die Szene besteht aus 207 Tausend Dreiecken. Abbildung 6.5 zeigt ein mit Blender gezeichnetes Bild dieser Szene.

6.1.6 Living Room

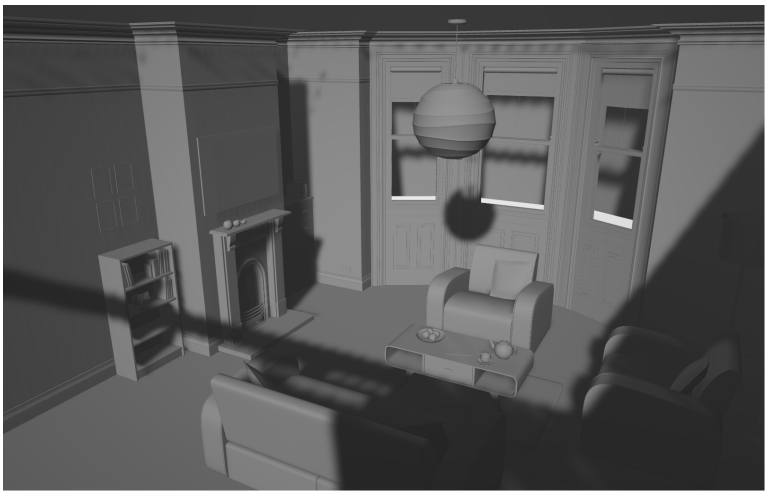

Abbildung 6.6: Render-Bild der Szene Living Room

Die Szene Living Room wurde ebenfalls von McGuires Website bezogen [McG17]. Sie besteht aus 580 Tausend Dreiecken. Abbildung 6.6 zeigt ein mit Microsoft 3D-Viewer gezeichnetes Render-Bild dieser Szene.

6.1.7 Lost Empire

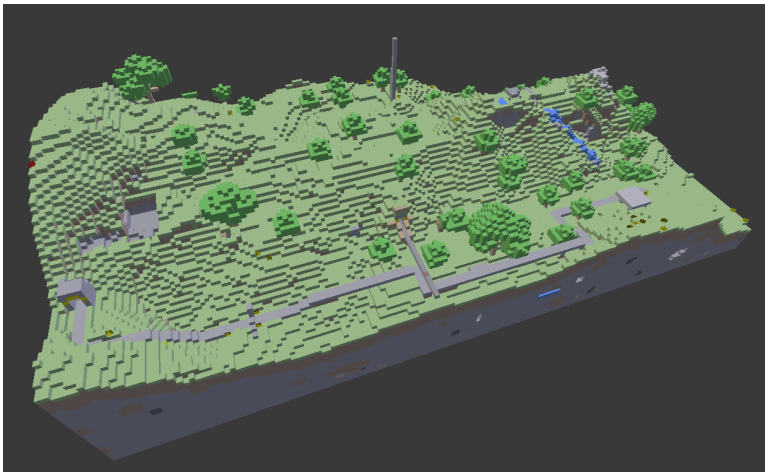

Abbildung 6.7: Render-Bild der Szene Lost Empire

Auch die Szene Lost Empire stammt aus dem Archiv von Morgan McGuire und setzt sich aus 225 Tausend Dreiecken zusammen [McG17]. Abbildung 6.7 zeigt ein mit Blender erstelltes Bild dieser Szene.

6.1.8 Sibenik Cathedral

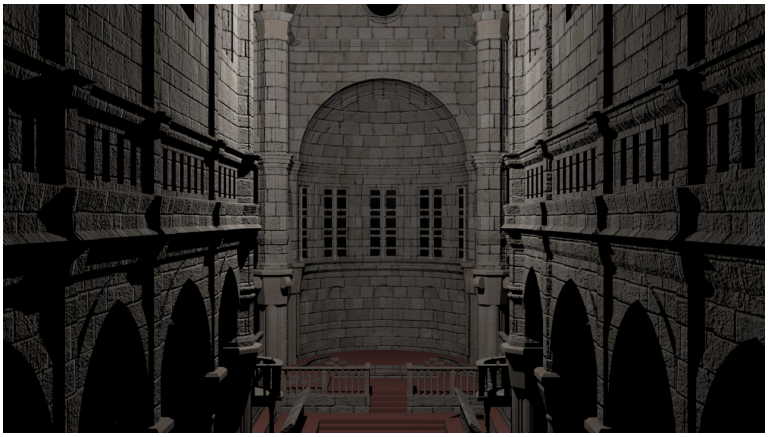

Abbildung 6.8: Render-Bild der Szene Sibenik Cathedral

Die Szene Sibenik Cathedral entstammt ebenfalls dem Archiv von McGuire [McG17] und besteht aus 75 Tausend Dreiecken. Abbildung 6.8 zeigt ein mit Blender erstelltes Bild dieser Szene.

6.1.9 Vokselia Spawn

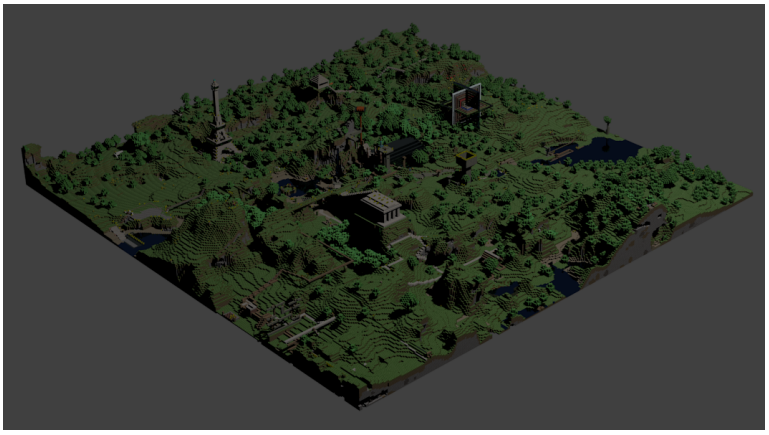

Abbildung 6.9: Render-Bild der Szene Vokselia Spawn

Vokeslia Spawn ist die letzte der neun verwendeten Szenen. Auch sie wurde der Website von Morgan McGuire entnommen [McG17]. Sie besteht aus 1,6 Millionen Dreiecken. Abbildung 6.9 zeigt ein mit Blender gezeichnetes Bild dieser Szene.

6.2 Forschungsfragen

Zur Beantwortung der Frage, wie gut Performanz von *librltuning* im Vergleich zu *libtuning* ist, wurden mehrere Experimente durchgeführt. In drei Forschungsfragen sollen mithilfe dieser Experimente verschiedene Aspekte der Performanz beider Auto-Tuner vergleichen werden. In Forschungsfrage 1 wird evaluiert, wie viel Luft nach oben *libtuning librltuning* lässt, also wie hoch eine realistische Laufzeitoptimierung im Vergleich zu *lituning* überhaupt ausfallen kann. Forschungsfrage 2 stellt die Frage danach, wie gut *librltuning* sich

in seinen vorgeschlagenen Parameterkonfigurationen über die Zeit des Produktiveinsatzes steigert und wie nah diese Werte an das in Forschungsfrage 1 ermittelte theoretische Optimum kommen. Im Rahmen von Forschungsfrage 3 werden neben dem Einfluss verschiedener RL-Parameterkonfigurationen auf die Laufzeiten mit *librltuning* vor allem die Gesamtlaufzeiten, inklusive dem Overhead durch die Auto-Tuning-Berechnungen, untersucht.

6.2.1 Forschungsfrage 1

Die erste Forschungsfrage ist die Frage nach dem Optimierungspotential in Bezug auf die Gesamt-Render-Laufzeit unter Einstaz von *libtuning* zur Laufzeitoptimierung. Im Raytracing wird die Farbe jedes Bildpunkts mehrfach berechnet, indem die Primärstrahlen, die durch einen Bildpunkt gehen, nicht durch den Mittelpunkt des Bildpunkts, sondern zufällig verteilt zwischen den Eckpunkten des Bildpunkts durch den Bildpunkt hindurch verschossen werden. Die Farbe eines Bildpunkts ergibt sich aus dem gemittelten Farbwert aller zufällig durch diesen geschossenen Primärstrahlen. Dieses Verfahren kann man so umsetzen, dass ein einzelner Frame mehrfach gezeichnet wird und bei jedem Zeichenvorgang jeder Primärstrahl zufällig verteilt zwischen den Eckpunkten seines jeweiligen Bildpunktes geschossen wird. Nach diesem Vorgang existieren nun N sogenannter *Samples* dieses Bildes. Das endgültige Bild wird nun dadurch erzeugt, dass die N Samples übereinander gelegt werden. Jeder Bildpunkt des endgültigen Bildes ergibt sich also aus dem durchschnittlichen Farbwert der Bildpunkte aller N Samples, die die selben (u, v)-Koordinaten haben.

Wird nun ein solcher Frame unter Zuhilfenahme von *libtuning* gezeichnet, startet *libtuning* seinen Explorationsprozess zu Beginn des Zeichnens des ersten Frame Samples. Die Tuning-Schleife ist die Schleife über alle Samples, die für einen Frame gezeichnet werden. Mit jeder Tuning-Schleifeniteration wird der Nelder-Mead-Algorithmus solange eine neue Parameterkonfiguration für den Bau der BVH vorschlagen, bis er zu einer bestimmten Konfiguration konvergiert ist. Diese Konfiguration ist mit hoher Wahrscheinlichkeit das globale Minimum der zu minimierenden Funktion der Sample-Render-Dauer in Abhängigkeit von der gewählten Parameterkonfiguration für diesen einen Frame. Es ist allerdings nicht auszuschließen, dass der Algorithmus zu einem lokalen statt zu dem globalen Minimum konvergiert und so unter Umständen bei Konvergenz eine Parameterkonfiguration vorschlägt, die nicht die beste unter allen möglichen für diesen Frame ist.

Aber angenommen, für jeden Frame f existiert eine solche optimale Parameterkonfiguration. Würde der Bau der BVH für f mit dieser Konfiguration parametrisiert werden, hätte dies zu Folge, dass es keine andere Parameterkonfiguration ist, die eine geringere Dauer für das Rendering eines Frame Samples zur Folge hat. Allerdings kennen wir eine solche optimale Konfiguration nicht. Aus diesem Grund wird für die Berechnung des Optimierungspotentials für einen Frame die geringste Sample-Render-Dauer herangezogen, wie durch wiederholtes Rendering mit Laufzeitoptimierung durch *libtuning* des Frames gemessen werden konnte. Die Parameterkonfiguration, mit der die geringste Sample-Render-Dauer für einen Frame gemessen wurde, wird im Folgenden ersatzweise als die optimale Konfiguration verwendet.

Nehmen wir weiter an, es existiert eine ideale Reinforcement Learning Auto-Tuner-Instanz, die für jeden Frame f genau eine solche optimale Parameterkonfiguration vorschlägt. Die daraus resultierende Dauer für das Rendering eines einzelnen Samples dieses Frames f ist die optimale Sample-Render-Dauer

$$d_f^{\text{opt}}.$$

Wird ein Frame f unter Einsatz von *libtuning* gerendert, so lässt sich die Dauer für das

Rendering des s-ten Samples ($s \in 1, \cdots, N$) eines Frames f schreiben als:

$$d_f^s \,.$$

Mithilfe dieser beiden Zahlen lässt sich nun das eingangs angesprochene Optimierungspotential P_f für einen Frame f definieren als:

$$P_f := \frac{\sum_{s=1}^{N} d_f^s}{N \cdot d_f^{\text{opt}}} \,. \tag{6.1}$$

Dieses Verhältnis P_f gibt für einen Frame f die Beschleunigung der Render-Dauer für f an, der unter Zuhilfenahme von *librltuning* maximal möglich ist. Das bedeutet, dass wenn zum Beispiel für einen Frame ein Optimierungspotential von 1,23 bekannt ist und die Laufzeit des Renderings dieses Frames mit *libtuning* 42 Sekunden gedauert hat, derselbe Vorgang nur 42 Sekunden : $1,23 = 34,15$ Sekunden dauern würde, wenn für alle Samples des Frames die bekannte optimale Parameterkonfiguration verwendet worden wäre.

6.2.1.1 Das Experiment

Die Daten zur Bestimmung des Optimierungspotentials für einen Reinforcement-Learning-unterstützten Auto-Tuner im Vergleich zu *libtuning* wurden wie folgt erhoben. Der Raytracing-Arbeitsauftrag, der die in Kapitel 6.1 beschriebenen neun 3D-Szenen, sowie jeweils zehn Kameraperspektiven pro Szene enthält, wurde zehn mal durchgeführt. Jeder dieser 90 Frames wurde also zehn mal gerendert, während dabei *libtuning* die Parameteroptimierung durchgeführt hat. Es wurden also insgesamt 900 Frame-Rendering-Durchläufe mithilfe von *libtuning* optimiert und die Laufzeit jedes einzelnen Sample-Renderings aufgezeichnet. Jeder Frame wurde mit 100 Samples gezeichnet.

Um einen idealisierten Reinforcement Learning Auto-Tuner zu simulieren, der für jeden Frame und dessen Indikatoren (siehe Kapitel 4.1) die perfekte Parameterkonfiguration in Bezug auf die Dauer des Frame-Renderings auswählt, wurde aus den erhobenen Rohdaten pro Frame unter allen zehn Frame-Render-Durchläufen und 100 Sample-Render-Durchläufen pro Frame derjenige Frame-Sample-Render-Durchlauf ausgewählt, dessen Frame-Sample-Render-Dauer die geringste ist. Dies ist die optimale Sample-Render-Dauer d_f^{opt} für diesen Frame f.

Um das Optimierungspotential P_f gemäß Gleichung 6.1 zu berechnen, muss zuerst ein „sinnvoller“ Wert für N berechnet werden. N beschreibt die Anzahl der ersten N Samples, die für die Berechnung des Optimierungspotentials herangezogen werden. Wenn wir davon ausgehen, dass *libtuning* mit ausreichend vielen Iterationen der Tuning-Schleife (also mit dem Rendern ausreichend vieler Samples eines Frames) zu einer „guten“ Parameterkonfiguration in Bezug auf die Sample-Render-Laufzeit konvergiert, bedeutet das, dass das Optimierungspotential für $N \longrightarrow \infty$ gegen 0 strebt. „Interessant“ ist also der Bereich der ersten N Samples, bei denen *libtuning* noch nicht konvergiert ist. Dies macht die Definition eines Konvergenzkriteriums erforderlich.

Sei $d_{f_r}^s$ die Dauer für das Rendering des s-ten Samples des r-ten wiederholten Renderings des Frames f und sei die Standardabweichung einer Menge von Datenpunkten durch die Funktion $\sigma\{\cdot\}$, sowie das arithmetische Mittel einer Menge von Datenpunkten durch die Funktion $\overline{x}\{\cdot\}$ beschrieben. Dann ist das Konvergenzkriterium definiert als

$$\frac{\sigma\{d_{f_r}^{s=N}, ..., d_{f_r}^{s=100}\}}{\overline{x}\{d_{f_r}^{s=N}, ..., d_{f_r}^{s=100}\}} > 0,015 \tag{6.2}$$

wobei N hier die gesuchte Anzahl der ersten N interessanten Samples ist. N wird durch einen iterativen Prozess ermittelt, der bei dem Intervall $[98, 100]$ der Sample-Indizes anfängt und in jeder Iteration das Intervall nach links um eins vergrößert. In der i-ten Iteration wird das Intervall $[100-2-i], 100]$ betrachtet. In jeder Iteration wird für das jeweilige Intervall die Standardabweichung und das arithmetische Mittel der Samples gebildet, deren Index in diesem Intervall liegen. Wird in Iteration $\hat{i}$ der Schwellwert von 0,015 zum ersten Mal überschritten, wird die Iteration abgebrochen und N wird der Wert $\hat{i}$ zugewiesen. Wird dieser Schwellwert selbst bei $[1, 100]$ nicht überschritten, so wird N auf 1 gesetzt.

6.2.1.2 Die Ergebnisse

Mit einem N_{f_r} für den r-ten Durchgang jedes Frames f und dessen Sample-Render-Referenzzeit d_f^{opt} können nun die Optimierungspotentiale berechnet werden. Jeder Frame wurde zehn mal gerendert. Die Optimierungspotentiale für einen Frame f wurden nun gemäß Gleichung 6.1 berechnet. Die Ergebnisse wurden in Form von Boxplots in den Abbildungen 6.10 und 6.11 grafisch dargestellt. Jedes Diagramm zeigt die die Optimierungspotentiale einer 3D-Szene. Die x-Achse gibt den Index der Kameraperspektive an, aus der diese 3D-Szene gezeichnet wurde. Jeder Boxplot stellt die Verteilung der zehn errechneten Optimierungspotentiale für einen Frame dar. Der Median kann als mittleres Optimierungspotential eines Frames betrachtet werden. Die schwarzen Punkte außerhalb der Antennen der Boxplots markieren die Ausreißer.

Betrachtet man alle neun Boxplot-Diagramme, so sieht man, dass das Optimierungspotential aller Szenen bis auf zwei Ausnahmen (Cornell Box und Dabrovic Sponza) innerhalb der selben Größenordnung liegt. Auch fällt auf, dass die Optimierungspotentiale der Frames derselben Szene ebenfalls sehr ähnlich sind. Interessant sind auch in dem Zusammenhang die Anzahl der Ausreißer, die vor allem bei der Cornell Box-Szene relativ groß ist, während sie gleichzeitig die geringsten Optimierungspotentiale besitzt. Möglicherweise hängt es damit zusammen, dass die Cornell Box-Szene die mit Abstand kleinste und am wenigsten komplexe Szene ist. Da der Median der Optimierungspotentiale fast aller zehn Cornell Box-Frames unter dem Faktor 1,05 liegt, liegt die Vermutung nahe, dass der Einfluss verschiedener Parameterkonfigurationen auf die Frame-Render-Dauer eher gering ist. Im Gegensatz dazu zeigen die Szenen „Audi R8“ und „Bugatti“ mit die höchsten Werte was das Optimierungspotential anbelangt. Beide Szenen gehören zum oberen Drittel aller Szenen in Bezug auf die Anzahl der Dreiecke. Tabelle 6.2 gibt eine Übersicht über die mittleren Optimierungspotentiale der einzelnen 3D-Szenen (Zeilen) und Kameraperspektiven (Spalten).

Neben den Mittelwerten sind auch die Extremstellen interessant. Zu wissen, wie viel Optimierungspotential maximal, wie viel minimal gemessen wurde, macht es einfacher, einen einzelnen gemessenen Wert für ein Optimierungspotential in das richtige Verhältnis zu setzen und einzuschätzen, ob ein bestimmter Wert für ein Optimierungspotential als „gut“ oder „schlecht“ einzuordnen ist. Der Rendering-Durchgang, der das höchste Optimierungspotential unter allen 900 Durchgängen hat, wurde auf der Szene „Audi R8“ aus Kameraperspektive 8 ausgeführt. Es handelt sich hierbei um Durchlauf 5. In Abbildung 6.12 sieht man den Verlauf der Sample-Render-Zeiten über den nacheinander gezeichneten 100 Frame-Samples. Es zeigt, wie die Laufzeit zu Beginn des Frame-Render-Durchgangs sehr hoch ist, aber relativ schnell in Richtung der roten Linie, die die optimale Sample-Render-Dauer d_f^{opt} dieses Frames f darstellt, abfällt. Abbildung 6.13 zeigt den selben Durchgang aber mit akkumulierten Laufzeiten. Jeder Punkt $(\hat{x}, y_{\hat{x}})$ auf dem Diagramm ist die Summe der ersten $\hat{x}$ Sample-Render-Zeiten. Man erkennt gut den Laufzeitunterschied, der am Ende des Renderings dieses Frames zum Vorschein tritt, wenn man anstelle von *libtuning* einen idealisierten Reinforcement Learning-unterstützten Auto-Tuner einsetzen würde. Dieser

Szene / Kamera	0	1	2	3	4	5	6	7	8	9
Audi R8	1,10	1,12	1,11	1,11	1,08	1,10	1,08	1,09	1,10	1,11
Bugatti	1,10	1,12	1,09	1,09	1,08	1,07	1,09	1,08	1,07	1,07
Cornell Box	1,04	1,02	1,03	1,03	1,04	1,04	1,03	1,04	1,03	1,05
Dabrovic Sponza	1,04	1,04	1,04	1,04	1,04	1,05	1,05	1,05	1,04	1,04
House	1,06	1,07	1,06	1,05	1,06	1,06	1,05	1,06	1,06	1,06
Living Room	1,04	1,05	1,05	1,04	1,04	1,05	1,04	1,04	1,04	1,04
Lost Empire	1,04	1,06	1,04	1,03	1,05	1,05	1,04	1,05	1,04	1,04
Sibenik Cathedral	1,05	1,06	1,05	1,04	1,05	1,06	1,04	1,04	1,04	1,04
Vokselia Spawn	1,14	1,12	1,09	1,09	1,06	1,10	1,06	1,09	1,08	1,06

Tabelle 6.2: Mittleres Optimierungspotential pro Frame

Durchlauf hat ein Optimierungspotential von 1,494. Für diesen Durchlauf wurde mithilfe des Konvergenzkriteriums ein $N = 3$ errechnet.

Im starken Kontrast dazu zeigt der Durchlauf mit dem geringsten Optimierungspotential ein anderes Bild: Wie in Abbildung 6.14 sehr schön zu sehen ist, verändern sich die Laufzeiten über die Zeit hinweg so gut wie nicht. Der Unterschied zur optimalen Sample-Render-Dauer, die auch hier wieder in rot eingezeichnet ist, ist minimal. Ein ähnliches Bild zeigt Abbildung 6.15, die die akkumulierte Sample-Render-Dauer über den gezeichneten Frame-Samples darstellt. Die beiden Linien, die der akkumulierten Laufzeit des gemessenen Durchlaufs und die der akkumulierten Sample-Render-Referenzzeit, unterscheiden sich nur marginal. Entsprechend gering fällt das Optimierungspotential mit 1,012 aus. Es wurden die ersten $N = 99$ Samples zur Berechnung dieses Optimierungspotentials herangezogen.

Das mittlere Optimierungspotential für den gesamten Durchlauf beträgt nach Berechnung mittels des Geometrischen Mittels 1,061 und nach Berechnung mittels des Medians 1,052.

6.2.1.3 Schlussfolgerung

Das Ergebnis dieses Experiments kann als Richtwert dafür betrachtet werden, wie hoch die Beschleunigung des Raytracings durch den in der vorliegenden Arbeit entwickelten Reinforcement Learning-Tuner im besten Fall ausfallen kann. Je nach Berechnung des Mittelwerts liegt dieses Potential bei 5 beziehungsweise 6% Verbesserung der Laufzeit.

6.2.2 Forschungsfrage 2

Bei der zweiten Forschungsfrage handelt es sich um die Frage, wie gut die im Rahmen dieser Arbeit entwickelte Reinforcement Learning Auto Tuner aus vergangenen Erfahrungen lernt. Genauer gefragt: Kann *librltuning* für einen identischen Frame f, der zu einem späteren Zeitpunkt ein weiteres Mal gerendert wird, eine bessere Parameterkonfiguration für das Rendering vorschlagen, sodass die Frame-Render-Dauer geringer ist, als beim Rendering zuvor und wie nah kommt die gemessene Dauer an die optimale Frame-Render-Dauer aus Forschungsfrage 1 heran? Ein späterer Zeitpunkt bedeutet in diesem Kontext, dass zwischen dem ersten und zweiten Rendering dieses Frames weitere Frames gerendert werden und *librltuning* deren Rendering-Vorgang optimiert. Die Frage hierbei ist nun, ob *librltuning* die aus der Laufzeitoptimierung dieser anderen Frames gewonnenen Erfahrungen nutzen kann, um beim zweiten Rendering von f durch geschicktere Wahl eine Parameterkonfiguration die Laufzeit im Vergleich zum ersten Rendering zu senken.

6.2.2.1 Das Experiment

Um diese Frage zu beantworten wurde das folgende Experiment durchgeführt. Eine Instanz der Klasse `CATunerQ2` aus *librltuning* wird erzeugt. Diese Instanz beherbergt die

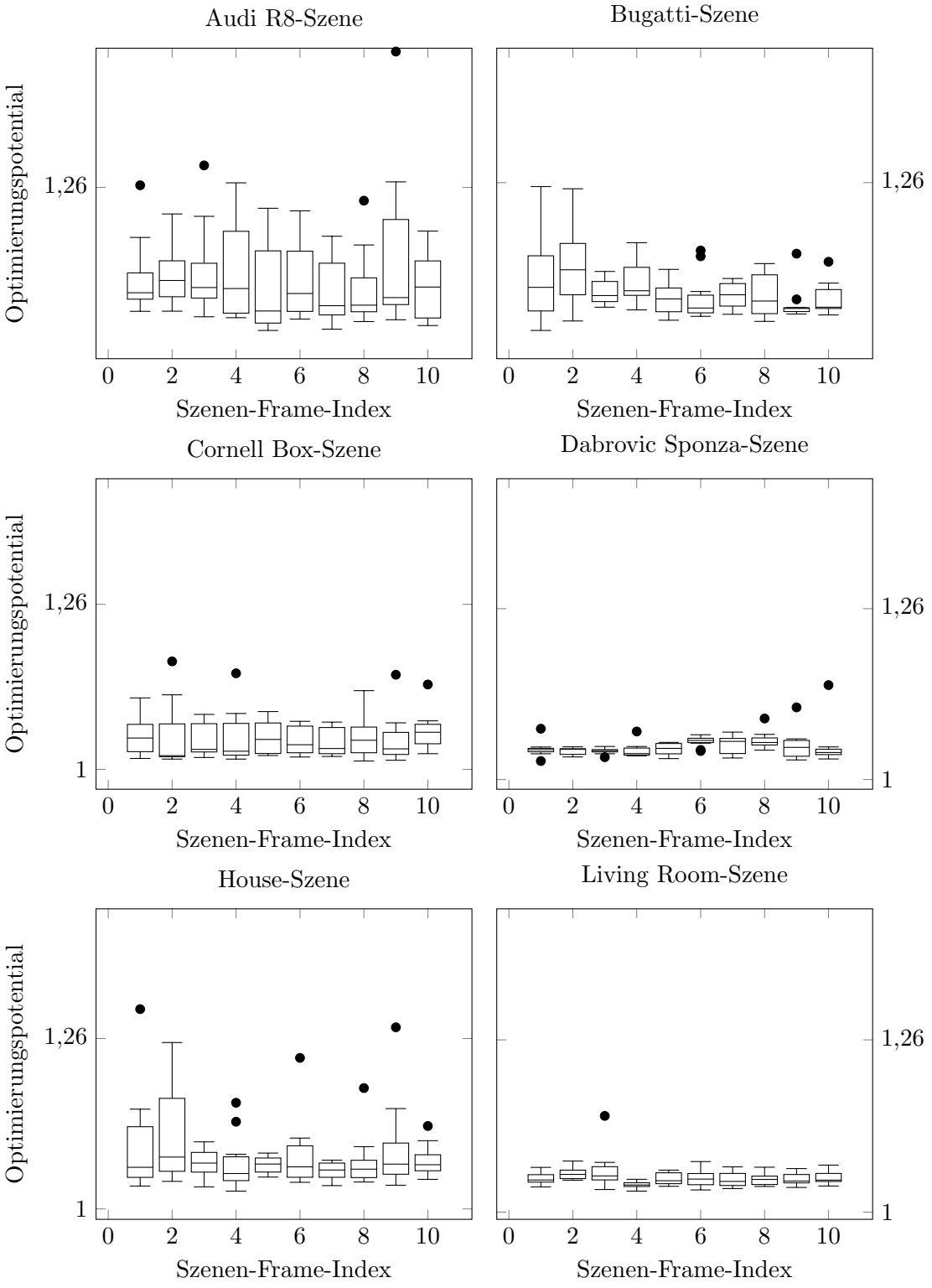

Abbildung 6.10: Darstellung der Optimierungspotentiale pro Szene mit Boxplots. Die x-Achse gibt den Index der jeweiligen Kameraperspektive an, die zusammen mit der Szene den jeweiligen Frame definiert.

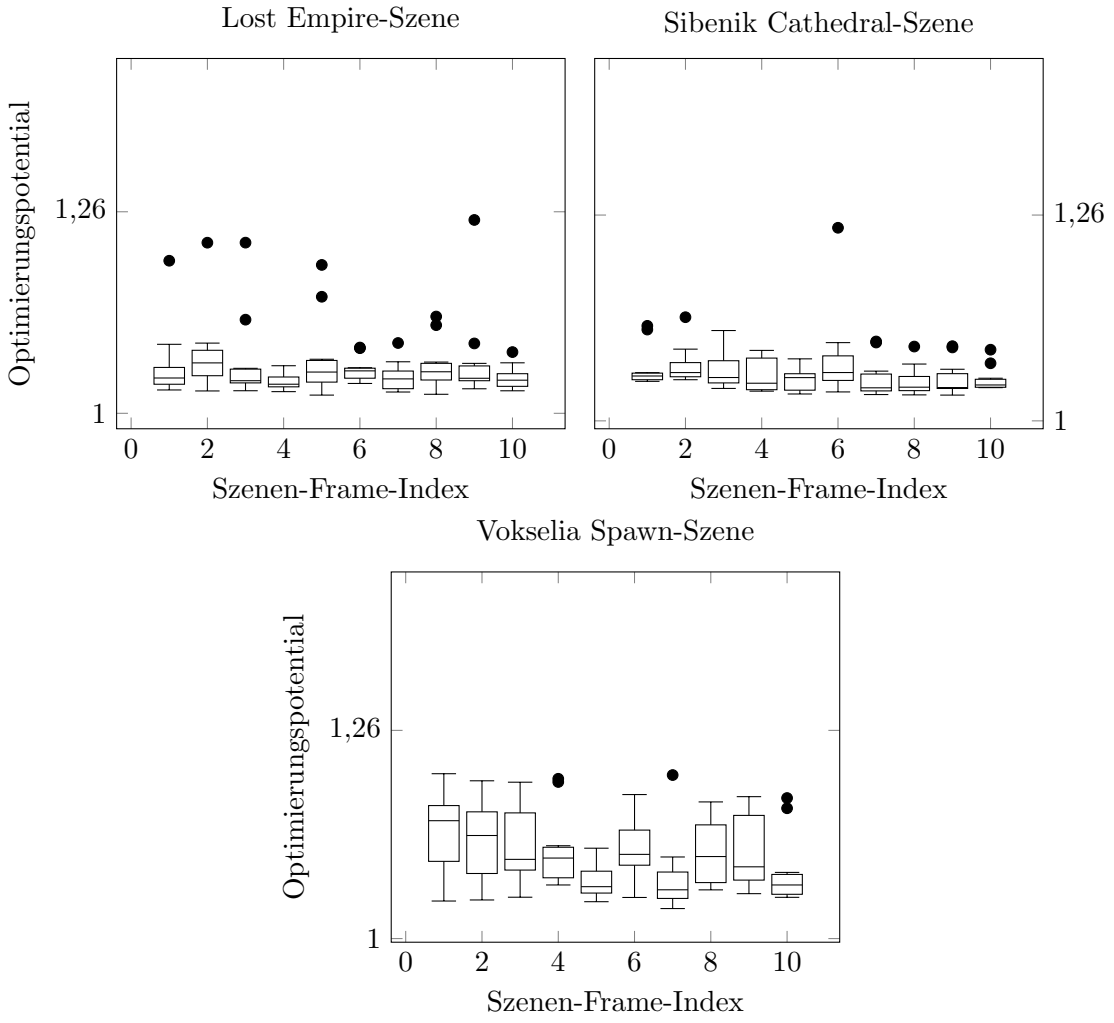

Abbildung 6.11: Fortsetzung: Darstellung der Optimierungspotentiale pro Szene mit Boxplots. Die x-Achse gibt den Index der jeweiligen Kameraperspektive an, die zusammen mit der Szene den jeweiligen Frame definiert.

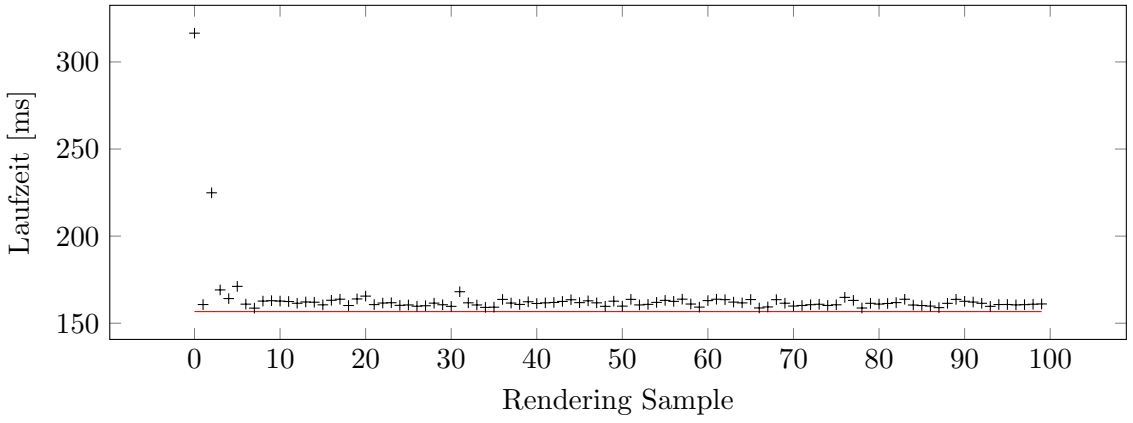

Abbildung 6.12: Laufzeit pro Sample beim Durchgang mit dem höchsten Optimierungspotential. Es handelt sich hierbei um Durchlauf Nummer 5 der Szene „Audi R8“ aus Kameraperspektive 8.

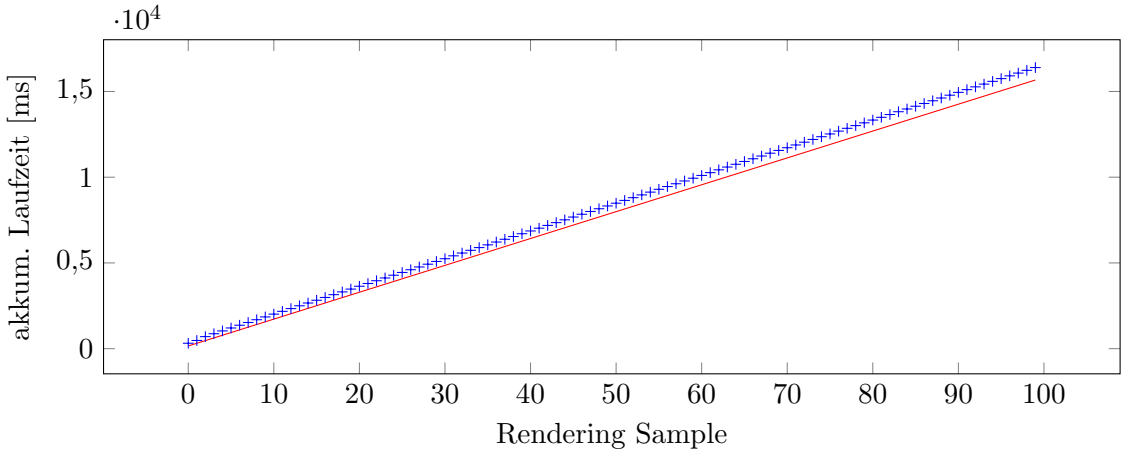

Abbildung 6.13: Akkumulierte Laufzeit pro Sample beim Durchgang mit dem höchsten Optimierungspotential. Es handelt sich hierbei um Durchlauf Nummer 5 der Szene „Audi R8“ aus Kameraperspektive 8.

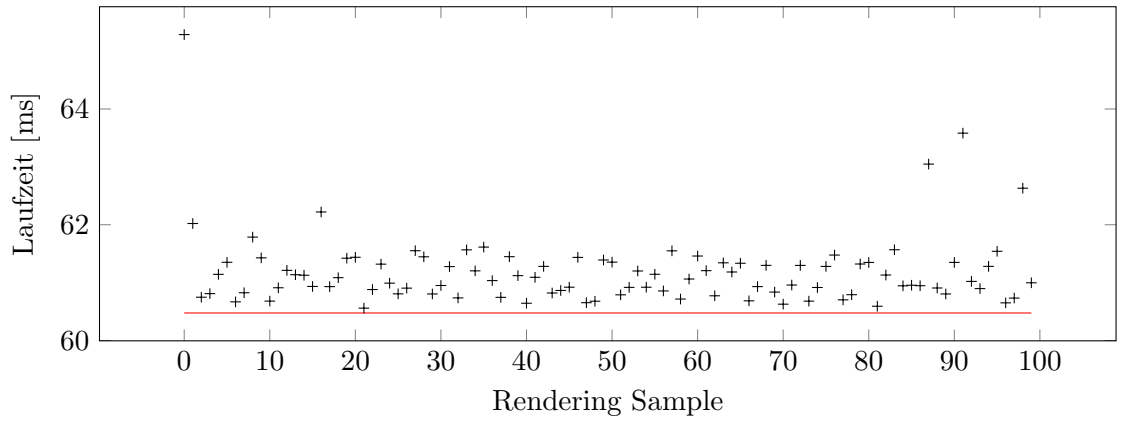

Abbildung 6.14: Laufzeit pro Sample beim Durchgang mit dem geringsten Optimierungspotential. Es handelt sich hierbei um Durchlauf Nummer 7 der Szene „Cornell Box“ aus Kameraperspektive 7.

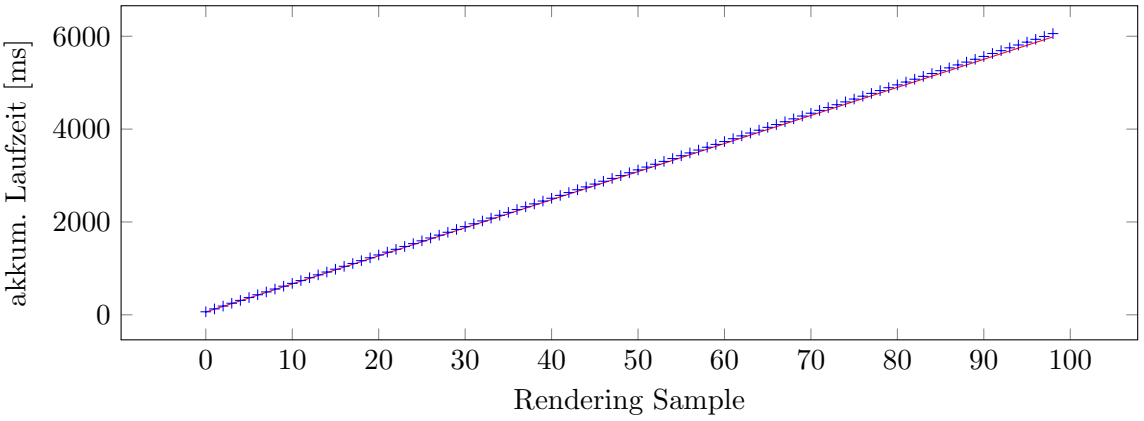

Abbildung 6.15: Akkumulierte Laufzeit pro Sample beim Durchgang mit dem höchsten Optimierungspotential. Es handelt sich hierbei um Durchlauf Nummer 7 der Szene „Cornell Box“ aus Kameraperspektive 7.

Reinforcement Learning-Implementierung, die in den Kapiteln 4 und 5 vorgestellt worden ist.

Für dieses Experiment werden zwei Raytracing-Arbeitsaufträge definiert: Ein Trainings-Arbeitsauftrag und ein Evaluations-Arbeitsauftrag. Der Trainings-Arbeitsauftrag besteht aus den 90 Frames, die auch in Forschungsfrage 1 verwendet wurden, allerdings in einer zufällig permutierten Reihenfolge. Der Evaluations-Arbeitsauftrag besteht aus einer 30 Frames großen Teilmenge besagter 90 Frames. Bei diesen 30 Frames handelt es sich um die 30 Frames, die das höchste mittlere Optimierungspotential aller 90 Frames haben. Das mittlere Optimierungspotential ist für jeden Frame in Tabelle 6.2 aufgeführt. Abbildung 6.16 gibt einen Überblick über den Ablauf des Experiments. In Blöcken von jeweils 10 Frames wird der Trainings-Arbeitsauftrag ausgeführt. `CATunerQ2` führt dabei die Laufzeitoptimierung durch. Nach Beendigung eines Trainings-Blocks wird `CATunerQ2` in einen reinen Exploitation-Modus geschaltet. In diesem Modus wird eine nächste Aktion ausschließlich mit der gierigen Strategie ausgewählt. Außerdem wird das Lernen aus der gemachten Erfahrung unterbunden. Der Gewichtungsvektor bleibt in diesem Modus unverändert. Während sich `CATunerQ2` nun in diesem Modus befindet, wird der Evaluations-Arbeitsauftrag ausgeführt und die Frame-Render-Zeiten gemessen. Nach Beendigung des Evaluations-Arbeitsauftrags wird `CATunerQ2` wieder in seinen normalen Modus versetzt und ein weiterer Block von 10 Frames des Trainings-Arbeitsauftrags wird gezeichnet. Dieser Vorgang wiederholt sich, bis der Trainings-Arbeitsauftrag vollständig ausgeführt worden ist. Nach Beendigung dieses Experiments wurde der Evaluations-Arbeitsauftrag insgesamt neun mal ausgeführt und die jeweiligen Frame-Render-Zeiten gemessen.

Parameterwahl für den RL-Tuner

Reinforcement Learning wird durch vier Parameter parametrisiert. Dies sind α, λ, ϵ und γ. Da für den Einsatzzweck zur Laufzeitoptimierung vergangene Erfahrungen keine Rolle für die Entscheidung der Auswahl einer geeigneten Parameterkonfiguration aus Basis der Frame-Indikatoren spielen, werden λ und γ auf 0 gesetzt. α wurde auf 0,1 und ϵ ebenfalls auf 0,1 gesetzt. Mit dem Setzten von λ und γ auf 0 wird für die Aktualisierung des Gewichtungsvektors $w_a(f)$ ausschließlich der vorherige Zeitschritt in Betracht gezogen. Dies ist sinnvoll, da bei der Auswahl einer Aktion für die Laufzeitoptimierung nur der gegenwärtige Zustand von Interesse ist und und der Zustand der vorherigen Zeitschritte keine Bedeutung für den gegenwärtigen Zustand und die Wahl der entsprechenden Aktion hat. α und ϵ wurden in Anlehnung an [SB98] auf jeweils 0,1 gesetzt, da für eine bestimmte Wahl für diese Parameter keine rationale Grundlage besteht und Sutton und Barto in ihrem Buch ebenfalls oft diesen Wert in ihren Beispielen verwenden. Die Wahl, ob *Accumulating (Eligibility) Traces* oder *Replacing (Eligibility) Traces* verwendet werden sollen, erübrigt sich durch die Wahl von $\lambda = 0$, da sich hierdurch das Verhalten beider Varianten nicht unterscheidet. `CATunerQ2` operiert so, dass, beim ersten Aufruf von `setState` immer exploriert wird. Dies ist der Implementierung von `CATunerQ2` geschuldet, die, wie in Kapitel 5.3.2 beschrieben, nur aus den Aktionen unter Anwendung der gierigen Strategie eine auswählen kann, die zuvor durch Exploration dem RL-Tuner bekannt gemacht worden sind.

6.2.2.2 Die Ergebnisse

Nach Beendigung des Experiments liegen nun für jeden Frame neun Messungen der Laufzeiten für das Rendering dieses Frames vor. Jeweils eine für die Durchführung des Evaluationsarbeitsauftrags der 30 Evaluations-Frames im Anschluss an einen Trainingsblock von zehn zufällig gewählten Frames. Jeder Frame wurde dabei mit 100 Samples gerendert. Die Laufzeiten für das Rendering der Evaluations-Frames sind in den Abbildungen 6.17, 6.18 und 6.19 grafisch dargestellt. Die Überschrift jedes Diagramms ist der Name der Szene und in Klammern der Index der Kameraperspektive, aus der diese Szene gezeichnet worden ist.

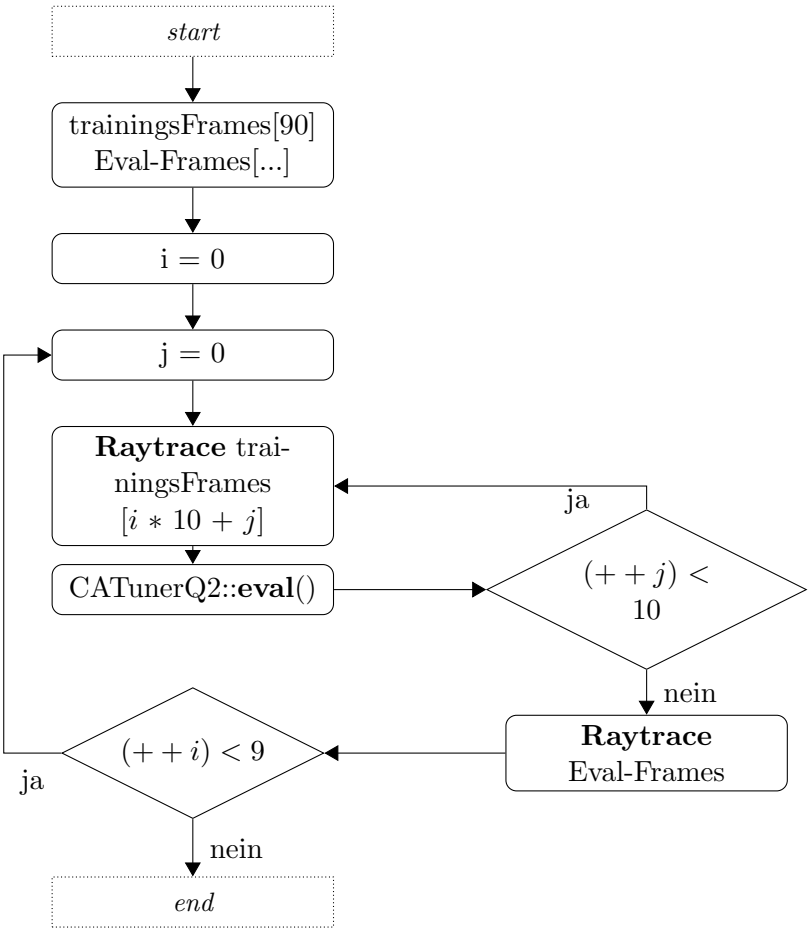

Abbildung 6.16: Programmablaufplan des Experiments

Die schwarze Linie zeigt die Frame-Render-Dauer über der jeweiligen Lerniteration an. Die rote Linie stellt die optimale Frame-Render-Dauer dar. Da jeder Frame mit 100 Samples gerendert wurde, berechnet sie sich durch:

$$D_f^{\text{opt}} := d_f^{\text{opt}} \cdot 100\,,$$

wobei d_f^{opt} die optimale Sample-Render-Dauer für einen Frame f ist, die im Rahmen von Forschungsfrage 1 errechnet wurde.

Die Ergebnisse fallen auffallend heterogen aus. Betrachtet man die Frame-Render-Laufzeiten nach der ersten und der letzten Lerniteration, fällt auf, dass nur bei wenigen Frames letztere geringer ausfällt als erstere. Nämlich Vokeslia Spawn (0), Audi R8 (1), Audi R8 (9), Vokselia Spawn (2), Vokselia Spawn (8), House (1) sowie House (4). Die Beschleunigung der Frame-Render-Zeit bezogen auf den Unterschied zwischen den Laufzeiten nach der ersten und der letzten Lerniteration liegen zwischen 1,345 bei Frame Vokselia Spawn (0) und 1,06 bei Frame Vokeslia Spawn (2). Die hier angegebenen Verhältnisse für einen Frame f sind das Ergebnis der Division

$$\frac{D_f^{l=1}}{D_f^{l=9}}$$

der Frame-Render-Dauer nach der ersten und der nach der letzten Lerniteration. Bei den anderen 23 Frames ist die Frame-Render-Dauer nach der letzten Lerniteration höher als nach der ersten. Der größte Anstieg in der Frame-Render-Dauer wurde bei Frame Audi R8 (6) gemessen. Die Beschleunigung von der ersten auf die letzte Lerniteration liegt bei 0,886.

Viel interessanter ist allerdings die Tatsache, dass die Frame-Render-Dauer im Vergleich zur Messung nach der ersten Lerniteration für fast alle Frames bis zur vierten Lerniteration teilweise deutlich sinken. Dies ist der Fall bei 24 Frames. Der Frame, bei dem die größte Frame-Render-Dauer-Beschleunigung

$$\frac{D_f^{l=1}}{D_f^{l=4}}$$

gemessen wurde, ist Vokselia Spawn (0). Die Beschleunigung beträgt hier 1,44. Die geringste positive Beschleunigung, also eine Beschleunigung, die größer als eins ist, wurde bei sieben Frames gemessen. Sie liegt bei allen sieben mit eine Beschleunigung von 1,00025 (bei House (2)) bis 1,009 (bei Vokselia Spawn (7)) nur unwesentlich über dem Faktor eins. Die insgesamt geringste Beschleunigung von Lerniteration eins auf Lerniteration vier wurde für Frame Audi R8 (7) gemessen. Bei diesem Frame hat sich die Frame-Render-Dauer um den Faktor 0,94 verbessert, also effektiv verschlechtert.

Interessant ist, dass das Muster, dass die Frame-Render-Laufzeiten ab Lerniteration 6 wieder ansteigen, bei so gut wie allen Evaluations-Frames auftritt. Einzig zwei der drei House-Frames, nämlich House (1) und House (4), sind davon nicht betroffen. Die Vermutung liegt nahe, dass in der sechsten Lerniteration mindestens ein Trainings-Frame einen wie auch immer gearteten negativen Einfluss auf die Vorhersagefähigkeit des Reinforcement Learning Agenten hat. Da diese Lerniteration keinen negativen Effekt auf die Frame-Render-Dauer von zwei der drei House-Frames hatte, könnte es etwas mit diesen Szenen zu tun haben. Um diesem Muster auf den Grund zu gehen, wurde dieser Test wiederholt. Dabei wurde die selben Folge von Evaluations-Frames genommen, aber eine neue zufällige Permutation der Reihenfolge der Trainings-Frames verwendet. Dort zeichnet sich ein anderes Bild: Für viele Evaluations-Frames sinkt die Frame-Render-Dauer bis hin zu einschließlich Lerniteration 7. Im Anschluss an die achte Lerniteration steigt bei vielen Evaluations-Frames die Frame-Render-Dauer stark an. Wenn die Theorie stimmt, dass mindestens ein Trainings-Frame ursächlich für den Anstieg der Frame-Render-Zeiten

der Evaluations-Frames ist, müsste es eine Überschneidung der Trainings-Frames aus der Lerniteration 6 des ursprünglichen Durchlaufs und denen aus Lerniteration 8 des neuen Durchlaufs geben.

Und diese Schnittmenge ist nicht leer. Ignoriert man die Kameraperspektive, aus der die Frames gerendert werden, und betrachtet nur die Szenen, so ist die Schnittmenge die Menge der folgenden Szenen: Lost Empire, Dabrovic Sponza und Sibenik Cathedral. Betrachtet man die Frames einschließlich der Kameraperspektive, so hat die Schnittmenge die Größe 1. Der Frame Dabrovic Sponza (3) ist in beiden Lerniterationen enthalten.

Um nun zu evaluieren, wie gut die erreichten Evaluations-Frame-Render-Dauern im besten Fall im Vergleich zu den gemessenen Frame-Render-Dauern mit *libtuning* aus Forschungsfrage 1 sind, bilden wir für jeden Evaluations-Frame das Verhältnis aus optimaler Frame-Render-Dauer ${D_f}^{\text{opt}}$ und der kürzesten gemessenen Evaluations-Frame-Render-Dauer

$$m_f := \min_l D_f^l$$

der 9 Lerniterationen. ${D_f}^{\text{opt}}$ ist die rote Linie in den Diagrammen 6.17, 6.18 und 6.19, D_f^l entspricht dem Verlauf der schwarzen Linie je Frame f. Der Median und das geoemtrische Mittel der Verhältnisse

$$\frac{m_f}{{D_f}^{\text{opt}}}$$

über alle Evaluations-Frames f ist der Wert, der mit dem mittleren Optimierungspotential aus Forschungsfrage 1 verglichen wird, um zu klären, wie gut die Ergebnisse mit *librltuning* im besten Fall einzuordnen sind. Der Median dieser Verhältnisse ist 1,062 und das geometrische Mittel beträgt 1,066. Der Median der Optimierungspotentiale aus Forschungsfrage 1 beträgt 1,052 und das geometrische Mittel 1,06.

6.2.2.3 Schlussfolgerung

Die Ergebnisse dieser Auswertung sind als eher positiv zu bezeichnen. Man sieht anhand der Diagramme, dass der in dieser Arbeit entwickelte Reinforcement-Learning Auto-Tuner definitiv online aus den gemachten Erfahrungen lernt, was dazu führt, dass die Frame-Render-Laufzeiten bei 80% (24 von 30) der Evaluations-Frames maximal vier Lerniterationen lang geringer werden. In vielen Fällen tritt eine deutlich positive Entwicklung der Frame-Render-Zeit schon nach der zweiten Lerniteration ein. Man kann also feststellen, dass *librltuning* schon nach 20 gerenderten Frames einen positiven Lerneffekt zeigt. Diese Frame-Render-Zeiten werden auch über weitere 20 Trainings-Frames weitesgehend gehalten. Ob der Anstieg der Frame-Render-Zeiten ab der sechsten Lerniteration nun ursächlich mit dem Frame Dabrovic Sponza (3) zu tun hatte oder mit den Szenen aus der Schnittmenge, lässt sich aus diesem Experiment nicht mit Sicherheit sagen. Die Diagramme zeigen allerdings auch, dass die rote Linie, also die optimale Frame-Render-Dauer, zu keinem Zeitpunkt berührt wird. Stattdessen sind die Abstände der Frame-Render-Zeiten immer noch relativ hoch. Betrachtet man die im letzten Abschnitt berechneten Vergleichswerte im Vergleich zu dem Median beziehungsweise dem geometrischen Mittel des Optimierungspotentials aus Forschungsfrage 1, sieht man, dass selbst die besten Ergebnisse mit *librltuning* aus diesem Experiment minimal schlechter sind als die gemittelten Ergebnisse aus Forschungsfrage 1. Dies lässt den Schluss zu, dass *librltuning* tendenziell höhere Frame-Render-Laufzeiten produziert als *libtuning*.

6.2.3 Forschungsfrage 3

Bei der dritten Forschungsfrage geht es um die Frage, wie die Gesamt-Rander-Dauer und auch die Gesamtdauer eines Raytracing-Arbeitsauftrags, wie er in den Forschungsfragen

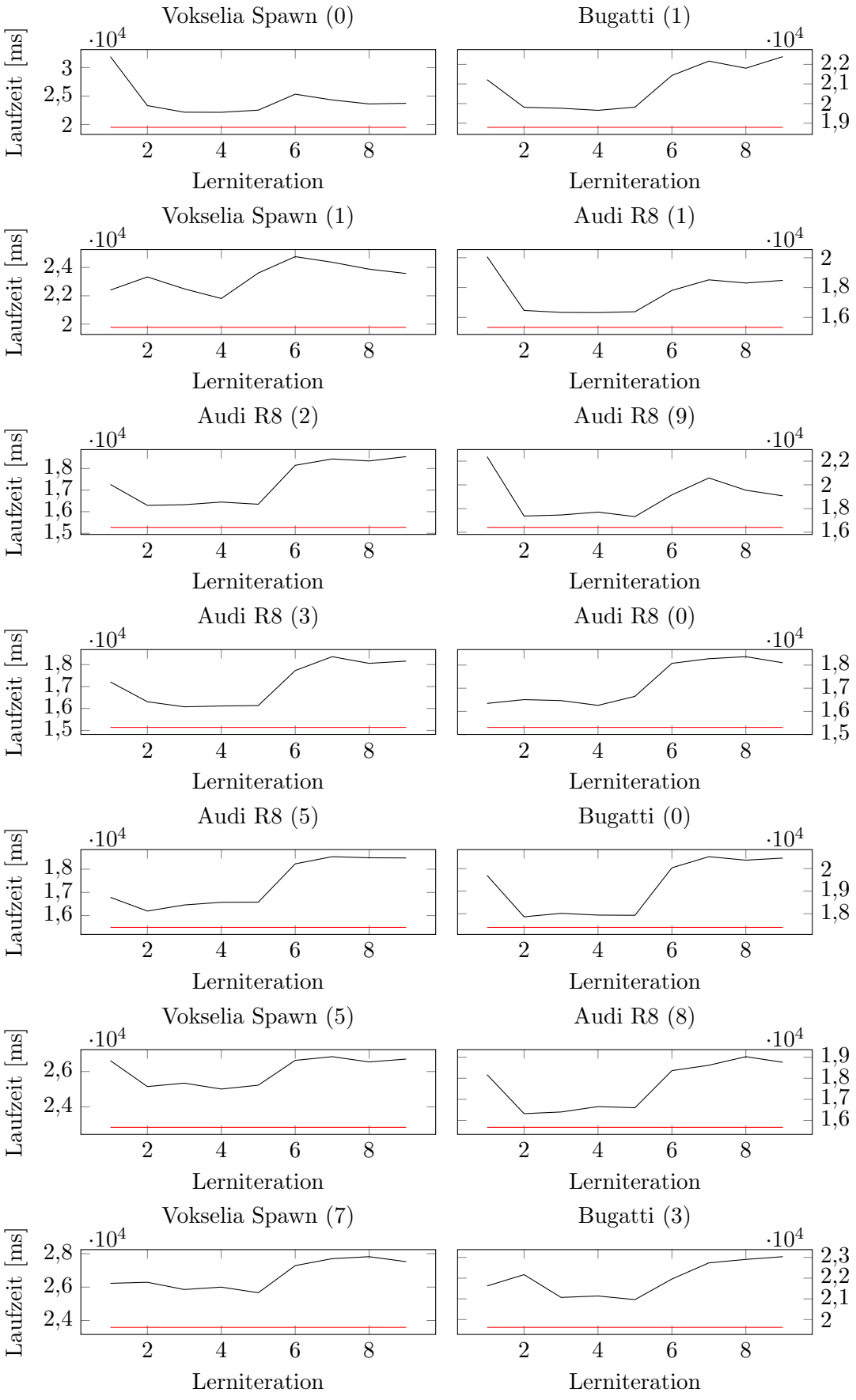

Abbildung 6.17: Render-Dauer des jeweiligen Evaluations-Frames im Anschluss an die x-te Lerniteration.

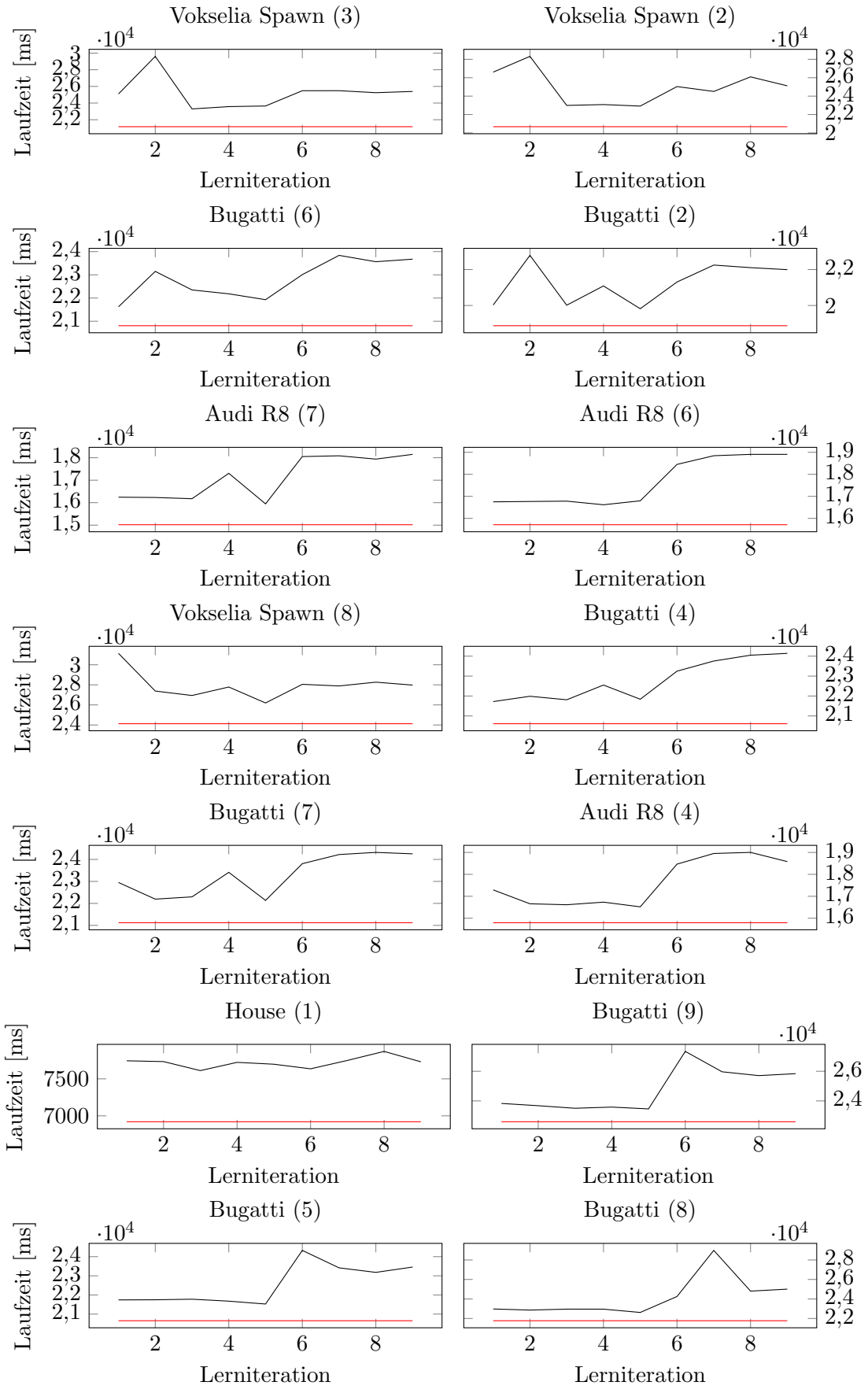

Abbildung 6.18: Fortsetzung: Render-Dauer des jeweiligen Evaluations-Frames im Anschluss an die x-te Lerniteration.

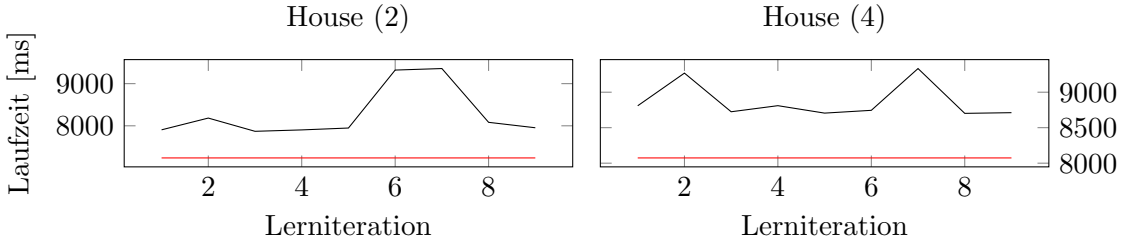

Abbildung 6.19: Fortsetzung: Render-Dauer des jeweiligen Evaluations-Frames im Anschluss an die x-te Lerniteration.

1 und 2 verwendet wurde, von verschiedenen Faktoren abhängen. Diese Faktoren umfassen verschiedene RL-Parameterkombinationen für *librltuning*, verschiedene Permutationen der Reihenfolge der Frames des Raytracing-Arbeitsauftrags sowie die verwendeten Auto-Tuner. Mit Gesamtdauer ist die Dauer der gesamten Abarbeitung eines Raytracing-Arbeitsauftrags gemeint, die nicht nur die Render-Dauer, sondern auch zum Beispiel die Zeit für die Auto-Tuning-Berechnungen mit einschließt.

6.2.3.1 Das Experiment

In diesem Abschnitt werden die Teilexperimente, aus denen sich das Experiment zur Beantwortung dieser Forschungsfrage zusammensetzt, besprochen. Dieses Experiment besteht aus mehreren Durchläufen des Raytracing-Arbeitsauftrags zum Rendern der 90 Frames. Zur Bestimmung der Laufzeiten für *librltuning* werden zwei Permutationen der Reihenfolge der 90 zu zeichnenden Frames unter Zuhilfenahme der Klasse `CATunerQ2` acht mal abgearbeitet. Bei jedem dieser acht Durchläufe wird die eingesetzte Instanz `CATunerQ2` mit einer anderen Kombination von Parametern instanziiert.
Die erste Permutation $\mathcal{P}_L$, die hier zum Einsatz kommt, ist die Permutation, die durch eine lexikalische Sortierung der Frame-Bezeichnungen entsteht. Die Frame-Bezeichnung setzt sich aus dem Namen der Szene und in Klammern dem Index der Kameraperspektive zusammen. Wie im letzten Abschnitt ist die Bezeichnung für zum Beispiel den Frame, der die Szene „Audi R8“ aus Kameraperspektive 3 zeigt, „Audi R8 (3)“. Das Interessante an dieser Permutation ist, dass immer alle zehn Frames einer 3D-Szene direkt nacheinander gerendert werden, was bedeutet, dass immer für zehn aufeinanderfolgende Frames die Szeneneigenschaften, die einen Teil der Indikatoren für den Reinforcement Learning Agent ausmachen, gleich bleiben.

Die zweite Permutation $\mathcal{P}_R$ ist die zufällig permutierte Reihenfolge der Liste der „Trainings-Frames“ aus Kapitel 6.2.2.1.

Einfluss der Parameter für `CATunerQ2` auf die Render-Zeit

Jede Permutation wird unter Zuhilfenahme einer Instanz von `CATunerQ2` acht mal gezeichnet, die die Laufzeitoptimierung des Render-Vorgangs vornimmt. Bei jedem dieser acht Durchläufe einer Permutation wird eine neue Instanz von `CATunerQ2` verwendet, die mit anderen Konfigurationsparametern instanziiert wird. Die Parameter, die für dieses Experiment manipuliert werden, sind die folgenden: $\alpha \in \{0,1; 0,2\}$, $\epsilon \in \{0,1; 0,2\}$ und $b \in \{5; 1\}$. α und ϵ sind zwei der vier Reinforcement Learning-Parameter. b gibt die Anzahl der *Bootstrap*-Durchläufe an. Die Reinforcement Learning-Parameter λ und γ werden wie im letzten Abschnitt auf 0 gesetzt.

Vergleich der Render-Zeiten der beiden Tuner

Für den Vergleich der Render-Zeiten des eingangs angesprochenen Raytracing-Arbeitsauftrags unter Zuhilfenahme von *libtuning* und *librltuning*, wurden die folgenden Daten miteinan-

der vergleichen: Um einen durchschnittlichen Durchlauf des Raytracing-Arbeitsauftrags für *libtuning* zu erhalten, wurden die Messwerte aus Forschungsfrage 1 herangezogen. Dort wurde derselbe Arbeitsauftrag der 90 Frames unter Zuhilfenahme von *libtuning* zehn mal abgearbeitet und die Render-Laufzeiten gemessen. Für jeden einzelnen Frame f wurden also zehn Frame-Render-Dauern $D_{f,r}^{\mathrm{AT}}$ erfasst. Einer für jeden Durchlauf $r \in \{1, \cdots, 10\}$. Für jeden Frame f wird nun der jeweilige Durchschnitt der zehn Durchläufe gebildet. Die durchschnittliche Frame-Render-Dauer für Frame f unter Zuhilfenahme des Auto-Tuners *libtuning* ist demnach gegeben durch:

$$\overline{D}_f^{\,\mathrm{AT}} = \frac{1}{10} \cdot \sum_{r=1}^{10} D_{f,r}^{\mathrm{AT}}$$

Der hochgestellte Index „AT“ weist darauf hin, dass die Laufzeiten unter Zuhilfenahme von *libtuning* erhoben wurden. Die Frame-Render-Dauer $D_{f,r}^{\mathrm{AT}}$ ergibt sich durch die Summe der 100 Sample-Render-Dauern dieses Frames f und Durchlaufs r.

Um einen Vergleich mit dem durchschnittlichen Durchlauf des Raytracing-Arbeitsauftrags mit *libtuning* zu erhalten, wurde auf analoge Weise ein durchschnittlicher Durchlauf für *librltuning* gebildet: In dem vorherigen Experiment wurden jeweils acht Durchläufe mit *librltuning* mit lexikalischer und zufälliger Permutation des Raytracing-Arbeitsauftrags durchgeführt. Da die Messwerte, die für die Berechnung von $\overline{D}_f^{\,\mathrm{AT}}$ herangezogen wurden, bei lexikalischer Permutation erhoben wurden, werden auch hier für die Ermittlung des durchschnittlichen Arbeitsauftrags für *librltuning* die acht Durchläufe mit lexikalischer Permutation verwendet. Wir erhalten also pro Frame f eine gemittelte Frame-Render-Dauer

$$\overline{D}_f^{\,\mathrm{RL}} = \frac{1}{|\mathcal{C}|} \cdot \sum_{c \in \mathcal{C}} D_{f,c}^{\mathrm{RL}} \quad \forall f \in \mathcal{P}_L \,.$$

$\mathcal{C}$ ist hierbei die verwendete Reinforcement Learning-Parameterkonfiguration mit

$$\mathcal{C} := \underbrace{\{0,1;0,2\}}_{\alpha} \times \underbrace{\{0,1;0,2\}}_{\epsilon} \times \underbrace{\{5;1\}}_{b} \,.$$

$\mathcal{P}_L$ gibt an, dass es sich hierbei um die Frames in lexikalischer Permutation des Raytracing-Arbeitsauftrags handelt. Der hochgestellte Index „RL“ weist darauf hin, dass die Laufzeiten unter Zuhilfenahme von *librltuning* erhoben wurden. Die Frame-Render-Dauer $D_{f,c}^{\mathrm{RL}}$ ergibt sich durch die Summe der 100 Sample-Render-Dauern $d_{f,c}^{s,\,\mathrm{RL}}$ der 100 Rendering Samples, mit denen jeder Frame gezeichnet wurde.

Vergleich der gesamten Ausführungszeiten

Die Gesamtdauer ist die Dauer, die ein kompletter Durchlauf der Schleife, die über alle zu rendernden Frames iteriert, dauert, deren Schleifenrumpf sowohl das Bauen der BVH und das eigentliche Raytracing sowie auch die Aufrufe an die jeweiligen Tuner-Instanzen beinhaltet.

Die Gesamtdauer für *libtuning* wurde ermittelt, indem die gemessene Zeit, die die zehn Durchläufe des kompletten Raytracing-Arbeitsauftrags auf Forschungsfrage, durch zehn geteilt wurde. Dies ist die durchschnittliche Gesamtdauer

$$\overline{G}^{\,\mathrm{AT}} = \frac{1}{10} \cdot \sum_{r=1}^{10} G_r^{\,\mathrm{AT}}$$

für die Abarbeitung eines Raytracing-Arbeitsauftrags unter Zuhilfenahme von *libtuning*.

Die gemessene Gesamtdauer für die Durchführung eines Raytracing-Arbeitsauftrags mit *librltuning* mit Parameterkonfiguration $\mathcal{C}$ und Permutation $\mathcal{P}$ ist

$$G_{\mathcal{C},\mathcal{P}}^{\mathrm{RL}} \,.$$

Permutation			Frame								
α	ϵ	b	31	32	33	34	35	36	37	38	Summe
0,1	0,1	1	19,1	18,2	19,5	19,7	19,8	20,0	20,3	19,9	156,5
0,1	0,1	5	19,4	18,9	20,1	20,2	18,2	20,3	20,4	20,2	157,7
0,1	0,2	1	19,1	16,9	19,4	19,4	19,6	19,7	17,4	19,2	150,7
0,1	0,2	5	17,7	18,7	20,2	18,2	20,2	20,1	17,5	20,0	152,5
0,2	0,1	1	18,9	18,2	19,1	19,3	19,5	19,7	21,4	19,4	155,6
0,2	0,1	5	18,1	17,2	18,5	18,5	18,4	18,2	18,6	18,2	145,6
0,2	0,2	1	17,9	17,3	18,3	18,3	18,7	18,6	17,8	17,8	144,7
0,2	0,2	5	18,2	17,4	18,4	18,2	18,2	18,8	17,9	17,6	144,6

Tabelle 6.3: Übersicht der Frames im Bereich 31 bis 38 des lexikalisch permutierten Raytracing-Arbeitsauftrag mit *librltuning*. Die verwendete RL-Parameterkonfiguration steht in der linken Spalte. Die rechte Spalte ist die jeweilige Zeilensumme der mittleren Spalten.

6.2.3.2 Die Ergebnisse

In diesem Abschnitt werden nun jeweils die Ergebnisse der einzelnen Teilexperimente besprochen.

Einfluss der Parameter für `CATunerQ2` auf die Render-Zeit

Der Raytracing-Arbeitsauftrag wurde für jede Reinforcement Learning-Parameterkombination $\mathcal{C}$ und Permutation $\mathcal{P}$ einmal durchlaufen, wobei `CATunerQ2` aus *librltuning* die Laufzeitoptimierung durchgeführt hat. Die Abbildung 6.20 zeigt den Laufzeitverlauf der lexikalischen Permutation $\mathcal{P}_L$. Dieses Diagramm beherbergt acht Kurven. Eine für jede Parameterkombination $\mathcal{C}$ für `CATunerQ2`. Die x-Achse des Diagramms zeigt den Index der Frames in der Reihenfolge der lexikalischen Permutation. Die y-Achse gibt die jeweilige Frame-Render-Dauer an. Jeder Frame wurde mit 100 Samples gerendert.

Wie man sofort erkennen kann, sind liegen die Kurven sehr nah beieinander. Größere Schwankungen sieht man vor allem in den Frames 31 bis 38 und 81 bis 90. Bei den Frames 1 bis 19 sind die Schwankungen insgesamt nicht so hoch, allerdings gibt es dort mehrere große Ausreißer nach oben.

In Tabelle 6.3 sind die Frame-Render-Zeiten für die Frames 31 bis 38 ausführlich dargestellt. Die Durchläufe dieser Frames mit den Konfigurationen $C_6 = \{0,2;0,1;5\}$, $C_7 = \{0,2;0,2;1\}$ und $C_8 = \{0,2;0,2;5\}$ liefern die insgesamt geringsten Frame-Render-Laufzeiten. Die höchste summierte Frame-Render-Dauer wurde mit Konfiguration $C_2 = \{0,1;0,1;5\}$ mit 157,7 Sekunden gemessen. Für Konfiguration $C_5 = \{0,2;0,1;1\}$ wurde hingegen die höchste Frame-Render-Dauer von 21,4 Sekunden in Frame 37 gemessen.

In Tabelle 6.4 sind die Frame-Render-Zeiten für die Frames 81 bis 90 ausführlich dargestellt. Hier zeigt sich ein leicht anderes Bild. Die beiden Konfigurationen $C_3 = \{0,1;0,2;1\}$ und $C_4 = \{0,1;0,2;5\}$ liefern hier die geringsten summierten Frame-Render-Laufzeiten. Die höchste summierte Laufzeit wurde mit den Konfigurationen C_6 und C_7 gemessen. Sie liegt bei 251,4 beziehungsweise 251,8 Sekunden. Die beiden größten Ausreißer nach oben pro Frame wurden mit den Konfigurationen C_7 und C_8 gemessen. Außerdem gibt es einen sehr großen Ausreißer nach oben bei Frame 81 mit Konfiguration C_1.

Betrachten wir nun die großen Ausreißer nach oben bei den Frames 1 bis 19. In diesem Frame-Bereich sind die Frame-Render-Laufzeiten sehr homogen mit kaum Varianz zwischen den einzelnen Konfigurationen, bis auf die folgenden fünf großen Ausreißer nach oben bei Frame 3, 6, 10, 13 und 19.

Permutation			Frame										
α	ϵ	b	81	82	83	84	85	86	87	88	89	90	Summe
0,1	0,1	1	28,6	20,8	22,1	21,9	25,7	24,2	25,4	26,2	26,6	26,8	248,3
0,1	0,1	5	23,8	20,4	21,3	21,7	25,5	25,2	25,5	26,4	26,3	26,6	242,6
0,1	0,2	1	21,6	20,2	21,9	21,6	25,3	23,5	25,3	24,1	25,5	26,5	235,6
0,1	0,2	5	20,5	20,2	20,8	21,6	25,1	24,4	25,2	25,0	26,6	27,3	236,8
0,2	0,1	1	21,8	20,9	23,9	22,1	27,0	23,6	27,3	26,2	24,7	28,9	246,4
0,2	0,1	5	22,1	23,7	22,5	24,5	26,3	25,7	26,4	26,0	26,0	28,3	251,4
0,2	0,2	1	20,7	21,4	21,7	22,7	28,4	24,6	26,8	32,1	25,9	27,4	251,8
0,2	0,2	5	21,2	21,7	23,7	23,7	27,0	23,7	26,4	24,9	27,0	28,4	247,8

Tabelle 6.4: Übersicht der Frames im Bereich 81 bis 90 des lexikalisch permutierten Raytracing-Arbeitsauftrag mit *librltuning*. Die verwendete RL-Parameterkonfiguration steht in der linken Spalte. Die rechte Spalte ist die jeweilige Zeilensumme der mittleren Spalten.

Permutation			Frame					
α	ϵ	b	3	6	10	13	19	Summe
0,1	0,1	1	16,2	16,2	17,5	20,2	22,9	93,0
0,1	0,1	5	16,7	16,2	17,5	19,8	22,9	93,1
0,1	0,2	1	20,7	16,6	16,9	20,1	23,1	97,4
0,1	0,2	5	16,4	16,1	16,9	20,2	23,0	92,6
0,2	0,1	1	16,0	16,4	17,3	20,1	25,5	95,3
0,2	0,1	5	16,5	19,2	21,9	20,4	23,5	101,5
0,2	0,2	1	16,2	16,2	17,1	24,4	23,9	97,8
0,2	0,2	5	16,5	16,0	17,0	19,7	22,8	91,9

Tabelle 6.5: Übersicht der Frames im Bereich 1 bis 19, bei denen es im lexikalisch permutierten Raytracing-Arbeitsauftrag mit *librltuning* größere Ausreißer gegeben hat. Die verwendete RL-Parameterkonfiguration steht in der linken Spalte. Die rechte Spalte ist die jeweilige Zeilensumme der mittleren Spalten.

Tabelle 6.5 zeigt die Ausreißer im Frame-Bereich 1 bis 19. Der Ausreißer bei Frame 3 wurde mit C_3, die bei Frame 6 und 10 mit C_6, der bei Frame 13 mit C_7 und der bei Frame 19 mit Konfiguration C_5 gemessen.

Die Lehre, die sich daraus ziehen lässt, dass bis auf eine einzige Konfiguration alle Konfigurationen entweder für sehr hohe summierte Frame-Render-Laufzeiten in den jeweiligen Frame-Bereichen sorgen oder häufig bei Ausreißern nach oben vertreten sind. Die einzige Konfiguration, die bei dieser Betrachtung *nicht* negativ aufgefallen ist, ist $C_4 = \{0,1;0,2;5\}$.

Betrachtet man die fünfte Spalte der Tablle 6.6, so decken sich die Ergebnisse der vorherigen Analyse der Ausreißer fast mit der gesamten Render-Laufzeit eines Durchlaufs. Mit Konfiguration $C_4 = \{0,1;0,2;5\}$ wurde bei den Durchläufen mit lexikalischer Sortierung die zweitgeringste gesamte Render-Laufzeit gemessen. Die geringste gesamte Render-Laufzeit wurde mit Konfuration $C_8 = \{0,2;0,2;5\}$ gemessen.

In Abbildung 6.21 sind die Frame-Render-Zeiten bei zufälliger Permutation aufgezeichnet. Im Gegensatz dazu zeigt der Verlauf der Frame-Render-Zeiten bei zufälliger Permutation des Raytracing-Arbeitsauftrags nur zwei nennenswerte Ausreißer nach oben. Nämlich einmal bei Frame 63 mit Konfiguration C_8 und einmal bei Frame 64 mit Konfiguration C_2. Weitere Schlüsse lässt dieser Graph leider nicht zu. Betrachtet man die gesamte Render-Laufzeit für einen Durchlauf in der fünften Spalte der Tabelle 6.6, sieht man, dass bei mit diesen beiden Konfigurationen die Werte im unteren Mittelfeld angesiedelt sind. Die beste gesamte Render-Laufzeit wird bei zufälliger Permutation mit Konfiguration C_1 erreicht. Generell liegen die Unterschiede in den Gesamt-Render-Laufzeiten allerdings im unteren einstelligen Prozentbereich. Eine abschließende Empfehlung und Nichtempfehlung einer oder mehrere Konfigurationen kann hier leider nicht ausgesprochen werden.

Und auch die Permutation hat nur einen geringen Einfluss auf die Gesamt-Render-Laufzeit. Summiert man die Gesamt-Render-Laufzeiten der jeweils acht Durchläufe von Permutation $\mathcal{P}_L$ und $\mathcal{P}_R$ auf, so erhält man $\overline{D}_{\mathcal{P}_L}^{\mathrm{RL}} = 9798211$ Millisekunden und $\overline{D}_{\mathcal{P}_R}^{\mathrm{RL}} = 9919784$ Millisekunden. Der Unterschied fällt mit

$$1 - \frac{\overline{D}_{\mathcal{P}_L}^{\mathrm{RL}}}{\overline{D}_{\mathcal{P}_R}^{\mathrm{RL}}} = 1 - \frac{9798211}{9919784} \approx 0,0123 = 1,23\%$$

entsprechend gering aus.

Es lässt sich also festhalten, dass weder die hier getesteten Parameterkombinationen noch die Reihenfolge der zu rendernden Frames einen nennenswerten Einfluss auf die Render-Zeit des gesamten Raytracing-Arbeitsauftrags hat, wenn sie mit Hilfe von `CATunerQ2` optimiert wird.

Vergleich der Gesamt-Render-Zeiten der beiden Tuner

In diesem Abschnitt werden die gemittelte Gesamt-Render-Zeit, die für die Abarbeitung des Raytracing-Arbeitsauftrags mit lexikalischer Permutation unter Zuhilfenahme von *libtuning* benötigt wird, mit der gemittelten Gesamt-Render-Zeit für denselben Arbeitsauftrag verglichen, wobei hier *librltuning* die Laufzeitoptimierung vornimmt.

Abbildung 6.22 zeigt die Frame-Render-Zeiten über den Frame-Indizes für beide Tuner. Auffällig ist, dass auch hier die Unterschiede zwischen den Kurven sehr gering ausfallen. Die einzigen sichtbaren Unterschiede wurden zwischen Frame 31 und 38 gemessen. In diesem Bereich ist die Frame-Render-Zeit mit *librltuning* ca. 6,88% höher.

Summiert man die Frame-Render-Laufzeiten für jeden Tuner auf, so erhält man $\overline{\mathcal{D}}^{\mathrm{RL}} = 1224776$ Millisekunden für die Gesamt-Render-Dauer der gemittelten Frame-Render-Dauern

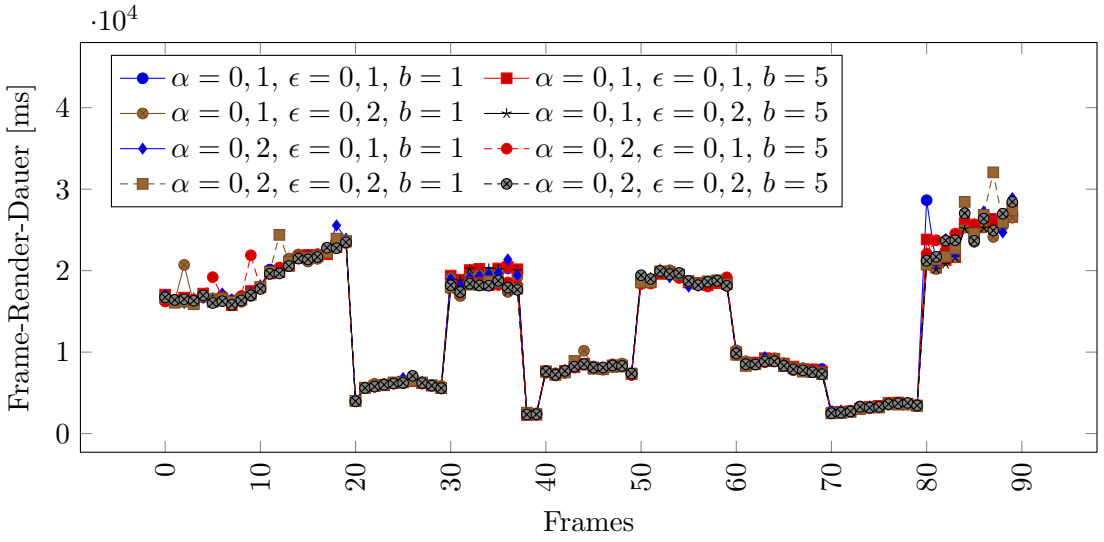

Abbildung 6.20: Jede Kurve zeigt den Verlauf der Frame-Render-Dauer für einen Frame aus dem lexikalisch permutierten Raytracing-Arbeitsauftrag mit *librltuning* für eine RL-Parameterkonfiguration

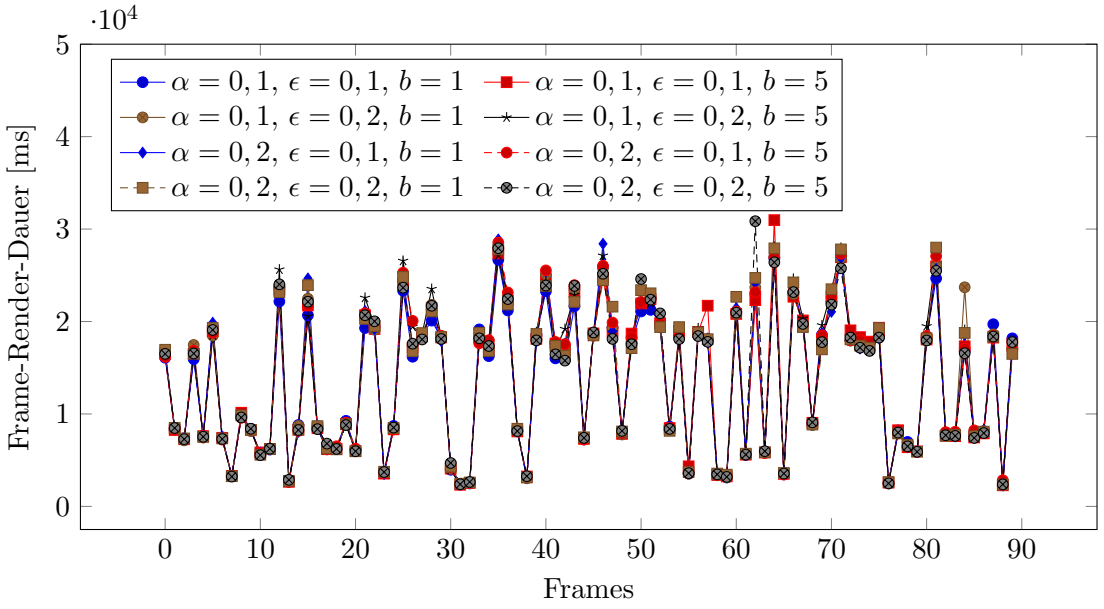

Abbildung 6.21: Jede Kurve zeigt den Verlauf der Frame-Render-Dauer für einen Frame aus dem zufällig permutierten Raytracing-Arbeitsauftrag mit *librltuning* für eine RL-Parameterkonfiguration

α	ϵ	b	Permutation	Gesamt-Render-Dauer [ms]	Gesamtdauer [ms]
0,1	0,1	1	Lexikalisch	1232128	4445000
0,1	0,1	5	Lexikalisch	1226702	4500750
0,1	0,2	1	Lexikalisch	1221044	4565770
0,1	0,2	5	Lexikalisch	1218069	4521590
0,2	0,1	1	Lexikalisch	1231556	4437400
0,2	0,1	5	Lexikalisch	1227991	4331050
0,2	0,2	1	Lexikalisch	1224839	4356070
0,2	0,2	5	Lexikalisch	1215883	4358500
0,1	0,1	1	Zufällig	1218168	4341490
0,1	0,1	5	Zufällig	1240644	4325290
0,1	0,2	1	Zufällig	1239268	4231290
0,1	0,2	5	Zufällig	1256214	4341670
0,2	0,1	1	Zufällig	1241492	4300090
0,2	0,1	5	Zufällig	1243833	4294800
0,2	0,2	1	Zufällig	1244751	4330650
0,2	0,2	5	Zufällig	1235416	4415810

Tabelle 6.6: Übersicht über die Gesamt-Render-Laufzeit und Gesamtlaufzeit eines Durchlaufs des Raytracing-Arbeitsauftrags mit *librltuning* unter Angabe der RL-Parameter und der Permutation des Arbeitsauftrags

bei Einsatz von *librltuning*, sowie $\mathcal{D}_V^{\mathrm{AT}} = 1205956$ Millisekunden, den Vergleichswert für *libtuning*. Das Rendering des gesamten Raytracing-Arbeitsauftrags unter Zuhilfenahme von *librltuning* ist somit um

$$1 - \frac{\mathcal{D}_V^{\mathrm{AT}}}{\overline{\mathcal{D}}^{\mathrm{RL}}} = 1 - \frac{1205956}{1224776} \approx 0,015366 = 1,54\%$$

langsamer. Dies ist natürlich nur ein rein theoretischer Vorsprung. Der Unterschied ist derart gering, dass er als Teil des Rauschens angesehen werden kann. In Bezug auf die gesamte Render-Zeit sind beide Tuner also fast gleich performant.

Vergleich der gesamten Ausführungszeiten

Die gesamte Ausführungszeit ist die Dauer für einen kompletten Durchlauf eines Raytracing-Arbeitsauftrags. Dies schließt neben der Zeit, die für das reine Raytracing oder den Bau der BVH benötigt wird, auch die Zeit für die Berechnungen des eingesetzten Auto-Tuners, mit ein. In diesem letzten Teilexperiment wird untersucht, mit welchem der beiden Auto-Tuner, *libtuning* und *librltuning*, eine geringere Gesamtausführungszeit erreicht wird.

Die durchschnittliche Gesamtdauer für einen kompletten Durchlauf mit *libtuning* beträgt:

$$\overline{G}^{\mathrm{AT}} = 4718580\,\text{Millsekunden} \approx 1,3\,\text{Stunden}.$$

Als Vergleichswert für die durchschnittliche Gesamtlaufzeit für einen kompletten Durchlauf mit *librltuning* wurden die gemessenen Gesamtlaufzeiten aller 16 Durchläufe mit *librltuning* aus Forschungsfrage 3 herangezogen. Die durchschnittliche Gesamtlaufzeit für *librltuning* ergibt sich aus dem arithmetischen Mittel der rechten Spalte der Tabelle 6.6:

$$\overline{G}^{\mathrm{RL}} = 4381076\,\text{Millisekunden} \approx 1,2\,\text{Stunden}.$$

Das Verhältnis der beiden Zahlen ist:

$$\frac{\overline{G}^{\mathrm{AT}}}{\overline{G}^{\mathrm{RL}}} = \frac{4718580}{4381076} \approx 1,077\,.$$

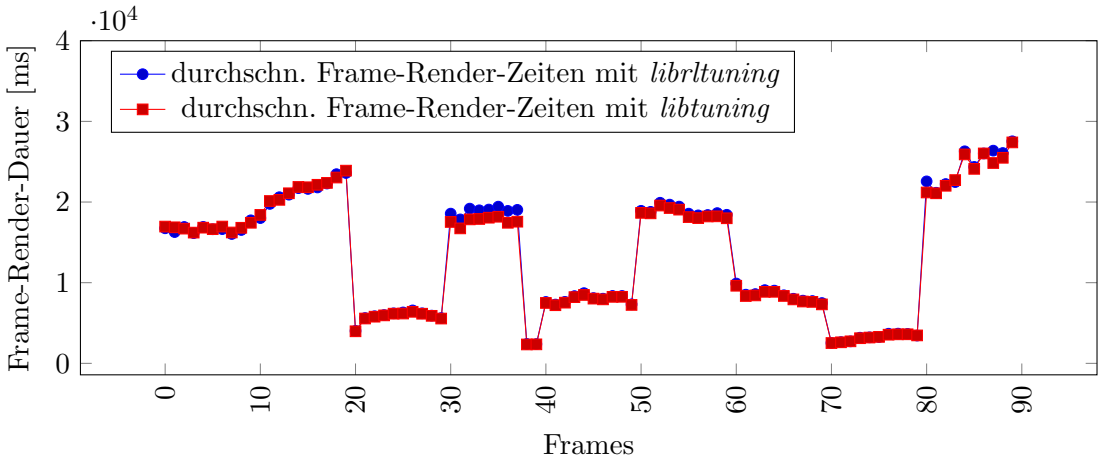

Abbildung 6.22: Vergleich der Frame-Render-Laufzeiten unter Verwendung von *libtuning* und *librltuning*. Die x-Achse gibt den Index des Frames im Raytracing-Arbeitsauftags an, dessen durchschnittliche Render-Laufzeit in der y-Achse notiert ist.

Die Gesamtlaufzeit mit *librltuning* ist also um 7,7% geringer als mit *libtuning*.

6.2.3.3 Schlussfolgerung

Das erste was beim Betrachten dieser Zahlen auffällt, ist, dass sie ungefähr um den Faktor 3,5 größer sind als die Gesamt-Render-Zeiten. Der Grund hierfür liegt nicht etwa in dem übermäßigen Mehraufwand, den beide Auto-Tuner verursachen, sondern in der Tatsache, dass für die Laufzeitmessung des Raytracings eines Frame Samples das Raytracing insgesamt sechs mal durchgeführt wird, um Rauschen in den Messdaten zu reduzieren, wie zu Beginn dieses Kapitels beschrieben.

Zwei Faktoren bedingen die Tatsache, dass die Gesamtlaufzeit mit *librltuning* geringer ist als mit *libtuning*, obwohl doch bei den Gesamt-Render-Laufzeiten ein gar für *librltuning* minimal negatives Ergebnis erzielt worden ist. Ein Faktor kann sein, dass möglicherweise *librltuning* tatsächlich einen geringeren Mehraufwand als *libtuning* verursacht. Der andere Faktor, der höchstwahrscheinlich einen deutlich größeren Einfluss auf die Gesamtlaufzeitdifferenz haben wird, ist der, dass der Einsatz des Nelder-Mead-Algorithmus es notwendig macht, dass in jeder Sample-Rendering-Iteration die BVH erneut gebaut werden muss, solange der Algorithmus nicht konvergiert ist. Bei der Benutzung von Reinforcement Learning wird bei Exploitation für einen Frame die BVH nur einmal gebaut und kann für alle Iterationen der Sample-Rendering-Schleife unverändert gelassen werden. Dieser Tatsache trägt eine Optimierung in dem Programm, das für die Durchführung der Evaluation dieser Arbeit entwickelt wurde, Rechnung, indem sie nur dann einen BVH-Neubau vollzieht, wenn entweder ein neuer Frame gerendert wird, oder sich die BVH-Bauparameterkonfiguration seit der letzten Sample-Rendering-Schleifeniteration nicht geändert hat.

Zusammenfassend lässt sich also festhalten, dass durch einen Austausch von *libtuning* durch *librltuning* in einem Raytracing-Kontext zwar für die reine Render-Zeit kein Vorteil entsteht, tendenziell sogar ein minimaler Nachteil. Allerdings kann durch den Einsatz von Reinforcement Learning in Kombination mit dem Nelder-Mead-Algorithmus dessen Limitierung im Einsatz bei Raytracing, die einen BVH-Neubau für jedes Sample benötigt, aufgehoben werden.

6.3 Schlussfolgerung der Evaluation

Die Ergebnisse der Evaluation sind schwer zu deuten. Auf der einen Seite hat das Experiment aus Forschungsfrage 1 ergeben, dass *libtuning* schon nach relativ wenigen Frame-Samples zu einer sehr guten Parameterkonfiguration konvergiert. Außerdem hat sich bei Auswertung der Render-Laufzeiten ersten Samples, bei denen das Konvergenzkriterium noch nicht erreicht worden ist, schon ein sehr geringes Optimierungspotential ergeben. mit 5 bis 6% durchschnittlicher möglicher Verbesserung bleibt nicht viel Luft für *librltuning*. Dies hat sich auch in Forschungsfrage 2 bestätigt. Das mittlere Optimierungspotential wurde selbst im Mittel der Evaluations-Frame-Render-Dauern der pro Frame besten Lerniteration nicht erreicht. Die eigentliche Frage dieses Experiments aus Forschungsfrage 2, konnte hingegen positiv beantwortet werden: Die Reinforcement Learning-Implementierung lernt aus der Erfahrung. Die Laufzeiten der Evaluations-Frame-Renderings werden mit der Zeit besser. Zumindest in den ersten vier Lerniterationen. Was nun schlussendlich für einen Anstieg der Evaluations-Frame-Render-Zeiten ab Iteration sechs gesorgt hat, ist noch nicht klar, aber der positive Aspekt der Resultate lässt sich festhalten mit der Erkenntnis, dass *librltuning* nur ungefähr 40 Zeitschritte benötigt, bis die bestmöglichen Laufzeiten mit dieser Implementierung erreicht werden. Die verschiedenen Experimente aus Forschungsfrage 3 haben das Verhalten von *librltuning* genauer unter die Lupe genommen. Sie haben gezeigt, dass weder die getesteten RL-Parameterkombinationen noch die Permutation der Frame-Reihenfolge einen nennenswerten Einfluss auf die gesamte Render-Dauer haben. Und auch der Vergleich der Frame-Render-Dauer mit den Messwerten aus Forschungsfrage 1 bestätigen den Vergleich aus Forschungsfrage 2. Die Render-Laufzeiten mit *librltuning* sind minimal schlechter als mit *libtuning*. Doch überflüssig ist *librltuning* deswegen nicht. Das zeigt das letzte Experiment aus Forschungsfrage 3. So ist es mit *librltuning* möglich, eine Schwachstelle in dem Ansatz, die Laufzeitoptimierung durch die BVH-Parameteroptimierung durchzuführen, zu aufzuheben. Mit *libtuning* ist es nämlich unumgänglich, den BVH-Bau in der Sample-Rendering-Schleife in jeder Iteration erneut durchzuführen, was unter normalen Umständen nur für jeden Frame einmal passieren muss. Hier kann unter Einsatz von *librltuning* auf im Schnitt 90% der unnötigen BVH-Neubauten verzichtet werden (für $\epsilon = 0,1$). Diese Tatsache hat zur Folge, dass die gesamte Dauer für einen Raytracing-Arbeitsauftrag von 90 Frames mit *librltuning* ungefähr 7,7% schneller vonstatten geht als mit *libtuning*, obwohl die gesamte Render-Dauer mit *librltuning* minimal schlechter ist.

7. Zusammenfassung und Ausblick

Die im Rahmen dieser Arbeit entwickelte Erweiterung von *libtuning* mit Reinforcement Learning auf kontinuierlichen Zustandsräumen ist die konzeptionelle Weiterentwicklung der Arbeit von André Wengert [Wen16]. Durch die Möglichkeit, kontinuierliche Zustandsräume zu akzeptieren, ist es nun möglich, Eigenschaften oder Charakteristika der Eingabedaten beim Auto-Tuning-Prozess miteinzubeziehen und hierbei nicht darauf angewiesen zu sein, eine Unterteilung einzelner Zustandsraumdimensionen zur Diskretisierung vorzunehmen. Die Unterstützung eines kontinuierlichen Zustandsraums wurde über eine Approximationsfunktion umgesetzt, die den Vektor von Indikatoren, der die spezifische Ausprägung der Charakteristika eines Eingabedatums sowie den gegenwärtigen Zustand repräsentiert, in einen Feature-Vektor übersetzt. Durch iteratives Ausführen einer Reihe von ähnlichen Eingabedaten und der Parameteroptimierung durch den Auto-Tuner lernt dieser durch Exploration und Exloitation mit der Zeit, den Zustandsraum zu generalisieren und für neue unbekannte Zustände geeignete Parameterkonfigurationen vorzuschlagen, die einem Optimum in Bezug auf die Rechenzeit nahe kommen. Hierfür wird dem Auto-Tuner in jeder Iteration der gegenwärtige Zustand, also der Indikatorvektor des zu verarbeitenden Eingabedatums, mitgeteilt. Der Auto-Tuner schlägt daraufhin eine Parameterkonfiguration vor und überwacht die Laufzeit der Verarbeitung dieses Datums und zieht daraus Rückschlüsse darüber, wie gut die gewählte Parameterkonfiguration hinsichtlich der Laufzeit gewesen ist. Als ein Hybride, der für die Exploration den aus *libtuning* bekannten Nelder-Mead-Algorithmus zur Funktionsminimierung und zur Exploitation eine geriege Strategie einsetzt, vereint er die Vorteile beider Welten aus konventioneller Optimierung und Maschinellem Lernen, das aus dem Ausführen der konventionellen Optimierung Schlüsse zieht und lernt zu generalisieren. Damit wird der Auto-Tuner in die Lage versetzt, für zukünftige Eingabedaten einer Folge „gute“ Vorschläge für Parameterkonfigurationen zu machen ohne eine vollständige Exploration des Suchraums durchführen zu müssen.

Bei der Durchführung der Evaluation hat sich schnell herausgestellt, dass die ursprüngliche Implementierung des Auto-Tuners `CATunerQ` nicht in der Lage ist, mit derart großen Suchräumen und damit verbunden großen Anzahl diskreter Aktionen umzugehen, wie es für die Evaluation unter Verwendung der Intel Embree BVH-Implementierung notwendig gewesen wäre. Aus diesem Grund wurde die weitere Implementierung des Reinforcement Learning-Verfahrens Q-Learning implementiert, das nicht für jede mögliche Aktion ein Platzhalter für deren approximierte Güte bereithält, sondern nur die approximierte Güte von solchen Aktionen speichert, die mindestens einmal als Ergebnis der Suchraumexploration einen Wert für die approximierte Güte zugewiesen bekommen haben. Inwiefern die

ursprüngliche Implementierung bei dieser Evaluation abgeschnitten hätte, konnte somit nicht beurteilt werden. Die Ergebnisse der Evaluation beziehen sich damit nur auf die zweite Implementierung `CATunerQ2`. Nach Durchführung des Experiments der ersten Forschungsfrage hat sich gezeigt, dass der Spielraum für eine weitere Optimierung der Laufzeit im Vergleich zu einer Optimierung mit *libtuning* mit ca. 5 bis 6 % Optimierungspotential sehr gering ist. Das Experiment der zweiten Forschungsfrage hat ergeben, dass `CATunerQ2` dieses Optimierungspotential aber nicht einmal erreicht, geschweige denn übertrifft. Die Hoffnung war, dass `CATunerQ2` die Lücke schließen und ein geringeres Optimierungspotential produzieren könnte, sodass die durch `CATunerQ2` (durch Parameteroptimierung) erzeugten Laufzeiten näher an einem theoretischen Optimum liegen als mit *libtuning*, was nicht der Fall ist. Allerdings konnte in Forschungsfrage 2 gezeigt werden, dass `CATunerQ2` aus den gemachten Erfahrungen begrenzt lernen kann. Eine Laufzeitreduzierung über mehrere Lerniterationen hinweg konnte festgestellt werden. Doch scheint die Implementierung empfindlich gegenüber schlechten Erfahrungen zu sein. So hat es nach Analyse der Laufzeiten je Lerniterationen den Anschein, dass bestimmte Eingabedaten dazu führen, dass die Vorhersage in darauffolgenden Iterationen wieder abnimmt und somit die Laufzeit wieder ansteigt. Doch es kann dennoch ein positives Fazit gezogen werden. Denn das letzte Experiment der dritten Forschungsfrage zeigt, dass die gesamte Laufzeit unter Einsatz von `CATunerQ2` dadurch gegenüber `libtuning` reduziert werden kann; dass in dem speziellen Anwendungsfall Raytracing Rechenoperationen, die für das Neuerstellen der BVH für jedes Frame Sample benötigt werden, eingespart werden können, wenn der Reinforcement Learning-Agent des Auto-Tuners `CATunerQ2` Exploitation betreibt, um eine Parameterkonfiguration vorzuschlagen. Diese Tatsache sorgt dafür, dass die gesamte gemessene Laufzeit mit `CATunerQ2` ungefähr 7,7% geringer ausfällt als mit *libtuning*. Dieses Aussparen der BVH-Neuerzeugungen ist indes bei typischem Raytracing nicht als Kniff oder Optimierung zu verstehen. Wird Raytracing auf konventionelle Art durchgeführt ohne den Einsatz des Nelder-Mead-Algorithmus zum Auto-Tuning, ist das Neubauen der BVH für jedes Frame Sample nicht nötig. Die BVH muss normalerweise nur einmal für jeden Frame erstellt werden und bleibt für aller Frame Samples hinweg unangetastet.

Aus den Entwurfsentscheidungen und der Tatsache, dass `CATunerQ` in dieser Arbeit nicht evaluiert worden ist, ergeben sich neue Themen für mögliche weiterführende Arbeiten.

7.1 Ausblick

Unterstützung kontinuierlicher Aktionsräume

Die wohl größte Limitierung dieser Reinforcement Learning-Implementierung ist die Tatsache, dass sie auf diskrete Aktionsräume beschränkt ist. Eine weitere Generalisierung auf kontinuierliche Aktionsräume würde erstens die Notwendigkeit einer Umgehungslösung für das Problem einer zu großen Menge an möglichen Aktionen aufheben und zweitens würde der Auto-Tuner damit in die Lage versetzt werden können, mit solchen Suchraumparametertypen umzugehen, die Gleitkommazahlen als Werte erlauben.

Verwendung einer anderen linearen Zustandsapproximationsfunktion

In der vorliegenden Arbeit wurde eine Radiale Basis-Funktion für die Approximation des Zustands verwendet. Hierbei handelt es sich um eine von vielen linearen Methoden zur Zustandsapproximation. Tile Coding oder Kanerva Coding sind weitere typische Vertreter der linearen Methoden. Die Wahl eines anderen Zustandsapproximators könnte Auswirkungen auf die Vorhersagefähigkeiten des RL-Agenten haben.

Verwendung nichtlinearer Zustandsapproximationsfunktionen

Mithilfe sogenannter Neuronaler Netze lässt sich ein nichtlinearer Zusammenhang zwischen gegenwärtigem Zustand und der für diesen Zustand optimalen Aktion herstellen, indem nicht etwa eine Liste von Werten für die Güte jeder Aktion approximiert wird, sondern die Aktualisierungsfunktion die Gewichte an den Kanten des Neuronalen Netzes entsprechend der Belohnung der Umgebung anpasst.

Weitere Möglichkeiten der Evaluation

In dieser Arbeit wurde sich bewusst auf die BVH-Implementierung der Intel Embree-Bibliothek beschränkt. Doch es existieren noch weitere Implementierungen für BVHs sowie andere räumliche Datenstrukturen, wie zum Beispiel kD-Bäume, die mit jeweils einem anderen Satz an Parametern, die optimiert werden können, ausgestattet sind.

Anwendung auf einer Grafikkarte

Sowohl das Erstellen der BVH als auch das eigentliche Raytracing werden mit der Intel Embree-Bibliothek auf der CPU ausgeführt. Eine Umsetzung von Raytracing, die sowohl auf der Grafikkarte als auch auf der CPU ausgeführt werden kann, würde neue Möglichkeiten für die Anwendung von *librltuning* schaffen. So könnte neben den Parametern für den Aufbau der räumlichen Datenstruktur auch das Verhältnis der Verteilung der Rechenlast zwischen CPU und Grafikkarte durch einen Auto-Tuner wie den in dieser Arbeit entwickelten in Bezug auf Charakteristika der zu zeichnenden Szenen optimiert werden.

Literaturverzeichnis

[ahm15] AHMETSALIH: *Audi R8*. https://free3d.com/3d-model/audi-r8-14024.html. Version: 2015. – Abgerufen 27.11.2018 (zitiert auf Seite 50).

[Ben75] BENTLEY, Jon L.: Multidimensional Binary Search Trees Used for Associative Searching. In: *Commun. ACM* 18 (1975), September, Nr. 9, S. 509–517 (zitiert auf den Seiten xi und 10).

[BHDNED+07] BAR-HILLEL, Aharon ; DI-NUR, Amir ; EIN-DOR, Liat ; GILAD-BACHRACH, Ran ; ITTACH, Yossi: Workstation Capacity Tuning Using Reinforcement Learning. In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. New York, NY, USA : ACM, 2007 (SC '07), S. 32:1–32:11 (zitiert auf den Seiten 21 und 22).

[DK17] DAHM, Ken ; KELLER, Alexander: Learning Light Transport the Reinforced Way. In: *CoRR* abs/1701.07403 (2017). http://arxiv.org/abs/1701.07403 (zitiert auf den Seiten 17 und 18).

[GD12] GANESTAM, Per ; DOGGETT, Michael: Auto-tuning interactive ray tracing using an analytical GPU architecture model. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* ACM, 2012, S. 94–100 (zitiert auf den Seiten xi und 20).

[HK09] HONG, Sunpyo ; KIM, Hyesoon: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: *ACM SIGARCH Computer Architecture News* Bd. 37 ACM, 2009, S. 152–163 (zitiert auf Seite 20).

[hou] *House*. http://benedikt-bitterli.me/resources/mitsuba/house.zip. – Abgerufen 27.11.2018 (zitiert auf Seite 52).

[ICG86] IMMEL, David S. ; COHEN, Michael F. ; GREENBERG, Donald P.: A radiosity method for non-diffuse environments. In: *Acm Siggraph Computer Graphics* Bd. 20 ACM, 1986, S. 133–142 (zitiert auf Seite 6).

[Kaj86] KAJIYA, James T.: The Rendering Equation. In: *SIGGRAPH Comput. Graph.* 20 (1986), August, Nr. 4, S. 143–150 (zitiert auf Seite 6).

[kim17] KIMZAUTO: *Bugatti Chiron 2017 Sports Car*. https://free3d.com/3d-model/bugatti-chiron-2017-model-31847.html. Version: 2017. – Abgerufen 27.11.2018 (zitiert auf Seite 51).

[McG17] MCGUIRE, Morgan: *McGuire Computer Graphics Archive*. http://casual-effects.com/data/index.html. Version: 2017. – Abgerufen 27.11.2018 (zitiert auf den Seiten 51, 52, 53 und 54).

[NM65] Nelder, J. A. ; Mead, R.: A Simplex Method for Function Minimization. In: *The Computer Journal* 7 (1965), Nr. 4, S. 308–313 (zitiert auf den Seiten 16, 22 und 23).

[Pad11] Padua, David (Hrsg.): *Encyclopedia of Parallel Computing*. Springer US, 2011. http://dx.doi.org/10.1007/978-0-387-09766-4. http://dx.doi.org/10.1007/978-0-387-09766-4 (zitiert auf Seite 14).

[SB98] Sutton, Richard ; Barto, Andrew: *Reinforcement Learning*. The MIT Press, 1998 (zitiert auf den Seiten xi, 12, 13, 14, 28 und 62).

[Shi09] Shirley, Peter ans Steve M.: *Fundamentals of Computer Graphics*. Third Edition. CRC Press, 2009 (zitiert auf den Seiten xi, 5, 6 und 8).

[Sut96] Sutton, Richard: Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In: *Advances in Neural Information Processing Systems* (1996) (zitiert auf den Seiten 29, 30 und 41).

[ŢCH+02] Ţăpuş, Cristian ; Chung, I-Hsin ; Hollingsworth, Jeffrey K. u. a.: Active harmony: Towards automated performance tuning. In: *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* IEEE Computer Society Press, 2002, S. 1–11 (zitiert auf Seite 22).

[TPKT16] Tillmann, Martin ; Pfaffe, Philip ; Kaag, Christopher ; Tichy, Walter F.: Online-Autotuning of Parallel SAH kD-Trees. In: *Parallel and Distributed Processing Symposium, 2016 IEEE International* IEEE, 2016, S. 628–637 (zitiert auf den Seiten 18 und 19).

[Wen16] Wengert, André: *Adaptives Auto-Tuning*, Karlsruher Institut für Technologie, Diplomarbeit, 2016 (zitiert auf den Seiten 2, 32 und 77).

[WH06] Wald, Ingo ; Havran, Vlastimil: On building fast kd-trees for ray tracing, and on doing that in O (N log N). In: *2006 IEEE Symposium on Interactive Ray Tracing* IEEE, 2006, S. 61–69 (zitiert auf Seite 10).

[Zer17] Zerr, Kevin: *Laufzeitoptimierung mittels Autotuning von Path-Tracing-Datenstrukturen*, Karlsruher Institut für Technologie, Diplomarbeit, 2017 (zitiert auf den Seiten xi, 2, 9, 44, 49 und 52).