UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

GPU basiertes Ray Tracing mit Bounding Volume Hierarchie

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von

Robin Schrage

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
Institut für Computervisualistik, AG Computergraphik

Zweitgutachter: Gerrit Lochmann
Institut für Computervisualistik, AG Computergraphik

Koblenz, im September 2016

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Ray Tracing als Bildsyntheseverfahren ist relevant für viele Anwendungsbereiche, da es Aspekte des Lichttransports physikalisch korrekt simulieren kann. Aufgrund des hohen Berechnungsaufwands sind der Einsatz von Datenstrukturen zur Beschleunigung und die parallele Verarbeitung notwendig. GPUs sind inzwischen hoch parallele, programmierbare Prozessoren mit zahlreichen Kernen und eignen sich aufgrund ihrer hohen Leistungsfähigkeit dazu, aufwändige, parallelisierbare Probleme zu lösen. In dieser Arbeit geht es um GPU Ray Tracing, beschleunigt durch Bounding Volume Hierarchien (BVH). Auf Basis relevanter Veröffentlichungen zu Aufbau und Traversierung von BVHs und der Abbildung des Ray Tracing Prozesses auf die GPU Architektur wird ein GPU Ray Tracer konzeptioniert und entwickelt. Während der BVH Aufbau vorab auf dem Host stattfindet, wird der gesamte Ray Tracing Prozess durch mehrere Kernel komplett auf der GPU ausgeführt. Die Implementierung der Kernel erfolgt in Form von OpenGL Compute Shader Programmen, und die Aufteilung des Ray Tracers auf mehrere Kernel ist durch die GPU Achitektur und das SIMT Ausführungsmodell motiviert. Für die Speicherorganisation der binären BVHs werden zwei Varianten betrachtet, klassisch und als MBVH, wobei sich die MBVH Organisation als effizienter erweist. Zudem werden verschiedene Varianten für die Traversierung ohne Stack und für die Stack-basierte Traversierung umgesetzt und bewertet. Der in mehrere Kernel strukturierte GPU Ray Tracer wird zudem mit einer Einzelkernel Version verglichen. Die besten Ergebnisse erreicht die Traversierung ohne Stack mit einem *while-while* Ablauf und MBVH im Rahmen des aufgeteilten GPU Ray Tracers.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Ziel der Arbeit . . . 3
- 1.2 Aufbau der Arbeit . . . 4

2 Grundlagen **4**
- 2.1 Die Rendering Equation und Beleuchtungsmodelle . . . 4
- 2.2 Ray Tracing . . . 6
  - 2.2.1 Antialiasing . . . 8
  - 2.2.2 Stochastisches Ray Tracing . . . 9
  - 2.2.3 Schnittpunktberechnung mit einem Strahl . . . 10
- 2.3 Hierarchische Beschleunigungsdatenstrukturen . . . 12
  - 2.3.1 Bounding Volume Hierarchie . . . 14
- 2.4 GPU Architektur und parallele Programmierung . . . 19
- 2.5 OpenGL Compute Shader . . . 22

3 Verwandte Arbeiten **31**
- 3.1 GPU Ray Tracer Struktur . . . 31
- 3.2 Bounding Volume Hierarchie Konstruktion . . . 36
- 3.3 GPU Ray Tracing und BVH Traversierung . . . 47
- 3.4 Zum Vergleich von BVHs und kd-Trees . . . 61

4 Konzept und Implementierung eines GPU Ray Tracers mit BVH **63**
- 4.1 Strukturierung des GPU Ray Tracers . . . 64
- 4.2 Repräsentation der Daten . . . 66
- 4.3 Host Programm . . . 67
- 4.4 BVH Aufbau . . . 68
- 4.5 BVH Traversierung . . . 69
- 4.6 Beleuchtung . . . 71
- 4.7 Ablauf der Kernel Ausführung . . . 72

5 Ergebnisse und Diskussion **74**
- 5.1 Testsystem . . . 74
- 5.2 Testszenen und Ergebnisse . . . 75
- 5.3 Diskussion . . . 83

6 Fazit & Ausblick **86**

Abkürzungsverzeichnis **91**

Literaturverzeichniss **93**

Abbildungsverzeichnis

1 Hilfskonstruktion für das perspektivische Zeichnen: Ein Ring an der Wand (Kamera/Betrachter), ein Faden mit Gewicht vom Ring ausgehend (Sehstrahl) durch einen Rahmen, in dem mit zwei weiteren Fäden die Position des Fadens fixiert wird (Bildebene) bis zu einem Punkt auf dem Objekt. Abbildung aus (Dürer 1525). 1
2 Backward Ray Tracing mit Sicht- und Schattenstrahlen. . . . 7
3 Whitted Ray Tracing: An jedem Schnittpunkt eines Strahls wird ein Schattenstrahl zur Lichtquelle verfolgt, je nach Material der getroffenen Oberfläche wird zusätzlich ein Strahl in Reflexionsrichtung (weiße Pfeilspitzen) und ggf. in Transmissionsrichtung erzeugt. Zur Verdeutlichung sind hier Schatten und Reflexion direkt in der Szene skizziert. 8
4 Distributed Ray Tracing: Mehrere Strahlen tasten ein Pixel ab (hier zur einfacheren Darstellung nur drei Strahlen), Verteilung der Schatten- und reflektierten Strahlen. Tiefenunschärfe und weiche Schatten sind zur Veranschaulichung in der Szene skizziert. 10
5 Bounding Volume Hierarchie in 2D und depth-first Traversierung (Pfeile) des entsprechenden Baums: der Strahl (Pfeil links) trifft die Wurzel und beide Kindknoten, der linke wird als erstes Traversiert (1), auf der nächsten Ebene werden wieder beide Kindknoten getroffen, wobei erneut der linke näher ist und zuerst besucht wird (2). Der Strahl trifft nun das rechte Blatt, aber nicht das enthaltene Dreieck. Der zuvor getroffene, noch nicht besuchte Knoten wird nun aufgesucht (4) und der Strahl trifft das enthaltene Dreieck. Schritt (5) wird nicht mehr durchgeführt, da die Distanz zum Schnittpunkt mit dem Dreieck kürzer ist als das zu prüfende BV. 13
6 Der Indexraum umfasst *num_goups_x* $\times$ *num_groups_y* $\times$ *num_groups_z* Work Groups der Größe *local_size_x* $\times$ *local_size_y* $\times$ *local_size_z* bestehend aus Work Items. 24
7 Aufteilung und Datenfluss des Streaming Ray Tracers nach Purcell et al. (Purcell et al. 2002). 32
8 Aufteilung der Ray Tracing Pipeline nach Garanzha et al. (Garanzha und Loop 2010): Strahlen-Erstellung, Strahlen-Sortierung, Frustum-Erstellung, Traversierung der BVH, Primitiv-Schnitttests, Beleuchtung akkumulieren 34

9 Unterteilung des GPU Ray Tracers auf vier Kernel: *Ray Generation* für das Erstellen und Speichern der Primärstrahlen, *Traversal & Intersection* für die Traversierung der BVH und Schnitttest mit Dreiecken, *Shadow* für Traversierung und Schnitttest mit Schattenstrahlen, *Shading & Secondar Rays* für Beleuchtung und Erstellen (und Verarbeiten) von Sekundärstrahlen für Reflexion und ggf. Refraktion. 65
10 Vereinfachte Darstellung der Klassen für den BVH Aufbau. . 68
11 Gerenderte Testszenen: *Crytec Sponza*, *Mad Science*, *Stanford Dragon*, *Conference Room*. 76
12 Darstellung der BVH für *Crytec Sponza* und *Conference Room*. 76

Listings

1 Compute Shader Ausführung 23
2 Erstellen des Shader Objekts, Laden des Shader Codes, Kompilieren, Binden an Programm Objekt und Verknüpfen des Programm Objekts in OpenGL. 25
3 Compute Shader in GLSL, mit Work Group Größe 32 x 32 . . 26
4 Einfaches Beispiel für die Verwendung von 2D Bild Objekten mit 32Bit Floating Point RGBA Werten. 27
5 Beispiel für die Definition eines OpenGL Shader Storage Buffer Objekts für die Nutzung in einem Compute Shader. . . . 28
6 Beispiel für die Definition eines Shader Storage Blocks mit einem Array of Structures 29
7 Beispiel Atomic Counter Buffer 30
8 Beispiel Atomic Counter im Compute Shader 31
9 Pseudocode *while-while* Traversierungs-Strategie („while-while trace()“ nach Aila und Laine 2009) 49
10 Pseudocode *if-if* Traversierungs-Strategie („if-if trace()“ nach Aila und Laine 2009) 49
11 Initialisierung und Vorverarbeitungsschritte für den GPU Ray Tracer 67
12 Dispatch des ersten Kernels, analog für weitere Kernel . . . 72
13 Bildpunkt Position und Strahl-Index 73
14 Speichern eines 1 Bit Shadowflag für eine Lichtquelle 73

Tabellenverzeichnis

1 Äquivalenz der Terme 24
2 Konfiguration der Testszenen 77
3 Strahlen pro Frame, ermittelt für Primär- Schatten- und Sekundärstrahlen und deren Summe (Rays/F) 77
4 Ergebnisse für den in vier Kernel strukturierten Ray Tracer. Zu den getesteten Varianten werden die erreichte Framerate (FPS), die Zeit pro Frame in Millisekunden (ms/F), die Anzahl der Strahlen pro Sekunde in Millionen (MRays/s) und die Leistung relativ zum Referenzwert von *MBVH while-while* (Ref.) gezeigt. 79
5 Ergebnisse für die Stack-basierte Traversierung mit einer Stackgröße von 64 für *while-while* mit BVH und MBVH. Die Spalte „Cor." gibt die Leistung relativ zur korrespondierenden Methode in Tabelle 4 an, Spalte „Ref." gibt die Leistung relativ zur MBVH while-while Referenz in Tabelle 4 an. . . . 80
6 Ray Tracing (4 Kernel) Ergebnisse mit BVH aus dem Aufbauverfahren mit Unterteilungssuche auf der Achse mit der größten Ausdehnung der Dreiecks-Schwerpunkte (Centroid Bounds). Die Spalte „Cor." gibt die Leistung relativ zur korrespondierenden Methode in Tabelle 4 an. Spalte „Ref." gibt die Leistung relativ zur MBVH while-while Referenz in Tabelle 4 an. 81
7 SAH Kosten und Anzahl der Knoten zu den BVHs auf Basis der Unterteilungssuche auf allen drei Achsen (BVH SAH 3 Axes) und der Unterteilungssuche entlang der Achse mit der größten Ausdehnung der Primitiv-Schwerpunkte (BVH SAH Centroid Extend). 82
8 Ergebnisse für die Einzelkernel Version des Ray Tracers. Die Leistung wird relativ zur korrespondierenden Methode aus Tabelle 4 gezeigt (Cor.) und zudem relativ zur Referenz aus Tabelle 4 (Ref.) 82
9 Aufgeteilter Ray Tracer mit *MBVH while-while* in 1920×1080. 83
10 Angenäherte EPO-Werte auf Basis der Dreiecks-BVs für BVHs aus beiden Varianten des Aufbauverfahrens: „3 Axes" und „CB" (Centroid Bounds). Spalte „Dif." gibt die relative Differenz der angenäherten EPO-Werte der BVHs beider Varianten an. Bei der *Dragon* Szene wurde die Bodenfläche entfernt. . . 86

1 Einleitung

Eines der Ziele in der Computergraphik ist es, aus einer dreidimensionalen Szenenbeschreibung photorealistische Bilder zu berechnen und darzustellen. Auch in der Kunstgeschichte findet man das Ziel wieder, möglichst realitätsgetreue Abbildungen zu erreichen. Bereits in der Rennaissance entwickelte sich das Bestreben, einen Ausschnitt der Realität möglichst originalgetreu abzubilden. Neben der zeichnerischen Umsetzung von Perspektive (Scriba et al. 2015) befasste man sich auch mit dem präzisen Abbilden von Beleuchtungssituationen mit Reflexion und Brechung des Lichts (Dutré et al. 2006). Es wurden Hilfmittel für die geometrische Konstruktion von Bildern entwickelt, wie z.B. ein Raster, durch das der Künstler die Szene betrachtete und das auf seinem Zeichenpapier repliziert war, Koordinaten für Bildpunkte und das materialisierte Äquivalent zu Sehstrahlen in Form von Schnüren, die von einem Bildpunkt zu seiner Entsprechung in der Szene gespannt wurden (Dürer 1525).

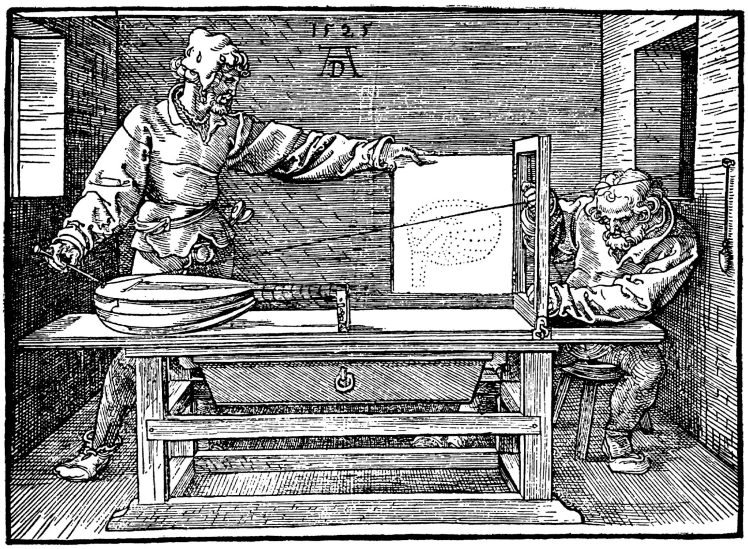

Abbildung 1: Hilfskonstruktion für das perspektivische Zeichnen: Ein Ring an der Wand (Kamera/Betrachter), ein Faden mit Gewicht vom Ring ausgehend (Sehstrahl) durch einen Rahmen, in dem mit zwei weiteren Fäden die Position des Fadens fixiert wird (Bildebene) bis zu einem Punkt auf dem Objekt. Abbildung aus (Dürer 1525).

Jahrhunderte später, in der Computergraphik, werden verschiedene Verfahren der Bildsynthese entwickelt, die auf Basis einer mathematischen, geometrischen und physikalischen Beschreibung einer Szene Bilder erzeugen. Durch die Anwendungsgebiete in verschiedenen Industriezweigen, wie z.B. der Computerspiel- und der Film-Industrie, wird insbesondere die Hardwareentwicklung angetrieben (Purcell et al. 2002, Dutré et al. 2006,

Friedrich et al. 2006, Kirk und Hwu 2010). Die Verbesserung und Neuentwicklung von Verfahren zusammen mit der Weiter- und Neuentwicklung von Hardware führen kontinuierlich zu höherer Darstellungsqualität, sowohl im Echtzeit- als auch im Offline Rendering. Wenn photorealistische Ergebnisse erreicht werden sollen, müssen die physikalischen Gegebenheiten des Lichttransports simuliert werden. Je genauer dies geschieht, desto besser ist das Ergebnis, gleichzeitig steigt jedoch der Berechnungsaufwand.

Ein grundlegendes Bildsyntheseverfahren, das darauf ausgelegt ist Aspekte des Lichttransports physikalisch korrekt zu simulieren, um realistische Bilder erzeugen zu können, ist das Ray Tracing. Insbesondere eignet sich Ray Tracing für die Darstellung von präzisen Schatten, Reflexionen und Lichtbrechung (Refraktion), also einer Teilmenge der globalen Beleuchtungsphänomene. Anwendung finden Ray Tracing Verfahren unter anderem für Animationsfilme, um entsprechende Beleuchtungseffekte realistisch darstellen zu können. So zum Beispiel in „Shrek 2“ für Reflexionen, Refraktion und in Kombination mit Irradiance Caching für das Final Gathering (Tabellion und Lamorlette 2004), in „Cars“ für Schatten, Reflexionen und Ambient Occlusion (Christensen, Fong et al. 2006), in „Monster Universität“ und weiteren, die im Rahmen verschiedener Verfahren für die Beleuchtungssimulation auch Ray Tracing einsetzten (Christensen, Harker et al. 2012).

Die Kombination aus realen Schauplätzen und Schauspielern mit virtuellen, gerenderten Schauplätzen, Objekten, Effekten und Charakteren erfordert ebenfalls eine entsprechende Beleuchtungssimulation. Einige Beispiele sind „Alice im Wunderland“ (2010), „Captain America“ (2011), oder „Pacific Rim“ (2013), welche „Arnold“ einsetzten, ein Renderer, der auf Monte Carlo Ray Tracing (Path Tracing) basiert (Seymour 2012, Solid Angle S.L. 2016). Path Tracing geht über klassisches Ray Tracing hinaus und erlaubt weitere Aspekte des Lichttransports physikalisch korrekt abzubilden, wie Kaustiken und diffuse Interreflexion. Inzwischen hat sich Path Tracing als Standardverfahren im Bereich Visual Effects (VFX) etabliert und wird von großen Renderingsystemen der Branche eingesetzt (Keller et al. 2015).

Computerspiele müssen in der Regel dynamische Umgebungen in Echtzeit darstellen können und streben dabei häufig eine möglichst hohe visuelle Qualität an. Das macht sie zu einem offensichtlichen, aber auch besonders herausfordernden Anwendungsgebiet für Ray Tracing (Schmittler et al. 2005, Friedrich et al. 2006, Bikker 2007, Pohl 2009). Häufig werden hybride Verfahren (Varianten des „Screen Space Ray Tracing“) angewendet, um die Geschwindigkeit der Rasterisierung auszunutzen und durch Kombination mit dem Prinzip des Ray Tracing (häufig in 2D bis 2,5D) die Darstellung durch angenäherte sekundäre Beleuchtungseffekte zu ergänzen (Bürger et al. 2007, McGuire und Mara 2014, Voica 2014). Für klassisches Ray Tracing in diesem Anwendungsbereich ist die Beschleunigung durch geeignete Daten-

strukturen für dynamische Szenen eines der zentralen Forschungsgebiete (Wald, Mark et al. 2007).

Damit der rechenintensive Ray Tracing Prozess beschleunigt werden kann, werden Datenstrukturen wie z.B. Grids, kd-Bäume, Octrees oder Bounding Volume Hierarchien verwendet. Diese Beschleunigungsdatenstrukturen und deren Traversierung erlauben es, die Anzahl von Schnittpunkttests (Strahl mit geometrischem Primitiv), welche sonst den größten Berechnungsaufwand beim Ray Tracing ausmachen (und je nach Auflösung und Szene für Milliarden von Strahlen berechnet werden müssten), auf das Notwendige zu reduzieren.

Der Gewinn an Performanz, sowohl durch leistungsfähigere Hardware als auch durch den Einsatz von Beschleunigungsdatenstrukturen, führte dazu, dass in der Forschung vermehrt auf interaktives und Echtzeit-Ray Tracing abgezielt wurde. Zudem eröffnete die GPU Entwicklung hin zu programmierbaren Shader-Einheiten um das Jahr 2001 (Purcell et al. 2002) die Möglichkeit Verfahren wie das Ray Tracing, das über viele Jahre ausschließlich auf CPUs lief, auch für die Ausführung auf GPUs umzusetzen. Da die Shader Programmierbarkeit sukzessive erweitert wurde und die Leistungsfähigkeit von GPUs bezüglich der Floating-Point Berechnungen pro Sekunde im Jahr 2009 bereits ca. 10 Mal so hoch ist, wie bei Mehrkern CPUs (Kirk und Hwu 2010), können gut parallelisierbare Probleme von einer Ausführung auf GPUs profitieren.

Mit der Einführung von Compute Shader Programmen und Shader Storage Buffer Objects (SSBO) in der OpenGL 4.3 Kernspezifikation (Segal und Akeley 2013) eröffnen sich erweiterte Möglichkeiten für die Implementierung von Verfahren oder Teilen einer Anwendung, die zwar im Rahmen einer Graphikanwendung eingesetzt werden, jedoch nicht direkt auf die Graphik Pipeline passen (wie z.B. Ray Tracing) im OpenGL Kontex. Für Anwendungen, die auf OpenGL basieren, können Compute Shader eine Alternative zu etablierten, dedizierten GPGPU APIs wie CUDA und OpenCL darstellen, da keine zusätzliche API notwendig ist und kein Kontextwechsel stattfinden muss.

1.1 Ziel der Arbeit

Im Rahmen dieser Arbeit wird ein GPU Ray Tracer entwickelt, inklusive dem Aufbau und Einsatz einer Bounding Volume Hierarchie (BVH) zur Beschleunigung. Der Aufbau der BVH findet auf der CPU statt, da die Konstruktion für statische Szenen als Vorverarbeitungsschritt ausgeführt wird und nicht für jeden Frame wiederholt werden muss. Es werden jedoch auch Ansätze für die Konstruktion auf der GPU diskutiert. Die Traversierung erfolgt auf der GPU. Der Ray Tracer soll statische 3D-Szenen aus Dreiecken, beleuchtet durch (maximal 32) Punktlichtquellen, rendern und

im Sinne eines klassischen, auch genannt „Whitted-“ Ray Tracers sollen Schatten, Reflexion und Refraktion dargestellt werden. implementiert wird der Ray Tracer im OpenGL Kontext per OpenGL Compute Shader. Den Rahmen für den zu entwickelnden Ray Tracer bildet das CVK-Framework der Arbeitsgruppe Computergraphik (Universität Koblenz Landau). Für die Umsetzung erfolgt die Einarbeitung in die Themengebiete Ray Tracing, Beschleunigungsdatenstrukturen (insbesondere BVH), GPU Architektur und parallele Programmierung, OpenGL 4.3 / 4.5 sowie die Recherche und Analyse bestehender Verfahren für GPU Ray Tracing und Bounding Volume Hierarchien.

1.2 Aufbau der Arbeit

Im Weiteren ist diese Arbeit wie folgt aufgebaut: In Abschnitt 2 werden die relevanten Grundlagen vorgestellt, zunächst die Rendering Equation als eine mathematische Beschreibung des Lichttransports, gefolgt von Einführungen in die Themengebiete Ray Tracing und Bounding Volume Hierarchien anhand der frühen, grundlegenden Veröffentlichungen auf dem jeweiligen Gebiet. Zwei weitere Abschnitte im Bereich der Grundlagen behandeln die Architektur moderner GPUs und darauf basierend Möglichkeiten, Einschränkungen und Hinweise für die (GP)GPU Programmierung (2.4), sowie die Grundlagen zum OpenGL Compute Shader (2.5). In Abschnitt 3 werden die Themen anhand von ausgewählten Veröffentlichungen vertieft, zunächst bezüglich der Struktur verschiedener GPU Ray Tracer (3.1), gefolgt von der Beschreibung verschiedener Verfahren für den Aufbau von BVHs (3.2) und schließlich werden BVH Traversierungsverfahren für (GPU) Ray Tracing vorgestellt (3.3). Darauf folgt in Abschnitt 4 die Beschreibung der Konzeption und Implementierung des im Rahmen dieser Arbeit entwickelten GPU Ray Tracers mit BVH. Abschnitt 5 präsentiert die Ergebnisse des entwickelten BVH Ray Tracers, Abschnitt 5.3 diskutiert die Ergebnisse und Abschnitt 6 beschließt diese Arbeit mit einem Fazit und dem Ausblick auf die weitere Entwicklung.

2 Grundlagen

2.1 Die Rendering Equation und Beleuchtungsmodelle

Die *Rendering Equation* stellt eine mathematische Formulierung der Beleuchtungsphänomene dar, die durch zahlreiche Bildsyntheseverfahren teilweise oder komplett simuliert bzw. angenähert werden sollen. Sie wurde erstmals von Kayija formuliert (Kajiya 1986) und ist seither in verschiedenen Ausführungen in Verwendung, davon ist eine (aus Müller 2008) in Gleichung 1 vorgestellt.

$$L_o(dA_e, d\vec{\boldsymbol{\omega}}_o) = L_e(dA_e, d\vec{\boldsymbol{\omega}}_e) + \int_i^{2\pi} fr(dA_e, d\vec{\boldsymbol{\omega}}_i, d\vec{\boldsymbol{\omega}}_o) \cdot L_i(dA_e, d\vec{\boldsymbol{\omega}}_i) \cdot \cos\theta_i d\vec{\boldsymbol{\omega}}_i \tag{1}$$

Beschrieben wird das Licht (genauer die Leuchtdichte) L_o, das von einem Oberflächenelement dA_e in eine beliebige Richtung $\vec{\boldsymbol{\omega}}_o$ (Raumwinkel) abgestrahlt wird, als die Summe des emittierten Lichts L_e vom Oberflächenelement dA_e in Richtung $\vec{\boldsymbol{\omega}}_e$ und des gesamten Lichts L_i, das aus allen Einfallsrichtungen $\vec{\boldsymbol{\omega}}_i \in 2\pi$ der Hemisphere eingeht und in Ausfallsrichtung $\vec{\boldsymbol{\omega}}_o$ reflektiert wird. Dabei ist fr die sechs-dimensionale (da dA_e einbezogen wird) Bidirektionale Reflexions-Verteilungsfunktion (Bidirectional Reflectance Distribution Function, BRDF), welche die Materialeigenschaften von dA_e repräsentiert und für jeden Einfallswinkel bestimmt, wie viel Licht (ausgehende Leuchtdichte zur eingestrahlten Beleuchtungsstärke) von dA_e in Richtung $d\vec{\omega}_o$ reflektiert wird.

Für die praktische Anwendung werden diskretisierte und oft weiter vereinfachte Formen der Rendering Equation verwendet, um sie annäherungsweise lösen zu können. Vereinfachte Beleuchtungsmodelle waren bereits vor Veröffentlichung der Rendering Equation im Einsatz, lassen sich aber auch darauf abbilden bzw. davon ableiten. So z.B. das lokale Beleuchtungsmodell, welches die Basis beim Scanline Rendering darstellt (Gleichung 2 nach Müller 2008), oder ein Modell für klassisches Ray Tracing (Gleichung 3 nach Müller 2008), das zusätzlich Beiträge mit einbezieht, die aus der Reflexions- und Transmissionsrichtung einfallen.

$$I = kd \cdot I_a + \sum_{i=1}^{q} (kd \cdot \cos\varphi_i + ks \cdot \cos^n \psi_i) \cdot I_{L_i} \tag{2}$$

$$I = kd \cdot I_a + \sum_{i=1}^{q} (kd \cdot \cos\varphi_i + ks \cdot \cos^n \psi_i) \cdot I_{L_i} \cdot \delta_i + \rho \cdot I_R + \tau \cdot I_T \tag{3}$$

Bei den hier gezeigten vereinfachten, diskreten Modellen wird nicht jede mögliche Einfallsrichtung berücksichtigt, sondern nur über die Anzahl an Lichtquellen q summiert. Dabei werden die Materialeigenschaften dargestellt durch kd als Koeffizient für diffuse Reflexion (Albedo) sowie ks als Koeffizient für glänzende Reflexion von der Lichtquelle (*specular*). Der Term φ_i ist der Winkel zwischen Normale $\vec{\boldsymbol{n}}$ des Oberflächenelements und Richtung zur Lichtquelle $\vec{\boldsymbol{l}}$ und $\cos\varphi_i = \vec{\boldsymbol{n}} \circ \vec{\boldsymbol{l}}_i$, gibt somit abhängig davon, wie Lichtquelle und Oberfläche zueinander ausgerichtet sind an, welcher Anteil des Lichts das Oberflächenelement trifft (Lamberts Kosinusgesetz). Durch $\cos^n \psi_i$ werden Glanzlichter oder Glanzpunkte, also die Reflexion einer

Lichtquelle auf der beleuchteten Oberfläche dargestellt, sofern diese einen glänzenden Anteil (ks) hat. Nach dem Phong Modell ist ψ_i der Winkel zwischen der Richtung zum Betrachter $\vec{v}$ und der Reflexionsrichtung (Richtung idealer Spiegelung) $\vec{r}_i = 2(\vec{n} \circ \vec{l}_i) \cdot \vec{n} - \vec{l}$ für das einfallende Licht in Bezug auf die Oberflächennormale. Der Exponent n ist der „Glanzexponent" und gibt grob einen Anhaltspunkt für die Oberflächenrauheit, indem er die Größe und Schärfe des Glanzpunktes bestimmt. Beim klassischen Phong Modell haben ks und n allerdings keine physikalische Bedeutung, es handelt sich um ein rein empirisches Modell. Der Term I_a steht für das Umgebungslicht (ambient light), eine Grundbeleuchtung, die ebenfalls eine Vereinfachung darstellt. Der Ambiente Term kann auch betrachtet werden als ein konstanter Stellvertreter der indirekten Beleuchtungsbeiträge aus der Umgebung, welche durch die anderen Terme nicht abgedeckt werden.

Gleichung 3 beachtet zusätzlich die rekursiven Beiträge aus spiegelnder Reflexion I_R und Transmission I_T sowie Schatten. Für Schatten wird ein Faktor (Schattenfühler) δ_i verwendet, für den Gleichung 4 gilt.

$$\delta_i = \begin{cases} 1 & \text{wenn Lichtquelle } i \text{ sichtbar} \\ 0 & \text{wenn Lichtquelle } i \text{ verdeckt} \end{cases} \tag{4}$$

Eine vollständige Simulation der globalen Beleuchtung, wie sie die Rendering Equation beschreibt, ist durch Path Tracing zu erreichen, indem die Lösung der Integralgleichung per Monte Carlo Integration angenähert wird (Kajiya 1986).

2.2 Ray Tracing

Die Idee für Ray Tracing im Kontext der Bildsynthese basiert auf gleichnamigen Verfahren aus der Physik (Optik). Beim Design und der Beurteilung von Linsen wurden Lichtstrahlen von einer Lichtquelle aus durch die Linse geschickt, um deren Pfad zu verfolgen und auf Papier abzubilden (Glassner 1989a). In den sechziger Jahren entstehen erste Verfahren in der Computergraphik, welche das Prinzip des Ray Tracing nutzen. Da jedoch viele der Lichtstrahlen einer Lichtquelle nie beim Auge des Betrachters ankommen, stellte man fest, dass es günstiger ist, den Prozess bei der Bildsynthese rückwärts durchzuführen (Backward Ray Tracing), der Strahl wird also vom Betrachter aus durch die Bildebene in die Szene bis zum vordersten Schnittpunkt mit einer Oberfläche verfolgt (Glassner 1989c). Das erste Backward Ray Tracing Verfahren stammt von Arthur Appel (Appel 1968), dem es darauf ankam, die Darstellung von Tiefe und räumlicher Beziehungen von Objekten (repräsentiert durch Linienzeichnungen flacher Oberflächen), durch Beleuchtung von beliebig positionierten Lichtquellen und entsprechenden Schatten zu verbessern. Appels Verfahren sendet dafür Strahlen vom Betrachter durch die Bildfläche in die Szene und bestimmt für jeden

Strahl den vordersten Schnittpunkt (und somit den sichtbaren Punkt). Vom Schnittpunkt aus wird ein Strahl zur Lichtquelle geschickt und festgestellt, ob der Punkt im Schatten liegt, also eine andere Fläche vor der Lichtquelle getroffen wird (Abb. 2). Bis dahin beschränkte sich Ray Tracing also auf die Bestimmung der Sichtbarkeit eines Oberflächenpunktes und der Schattierung. Das Verfahren ist, auch wegen der damals verfügbaren Hardware, noch weit entfernt von der Erzeugung realistischer Bilder.

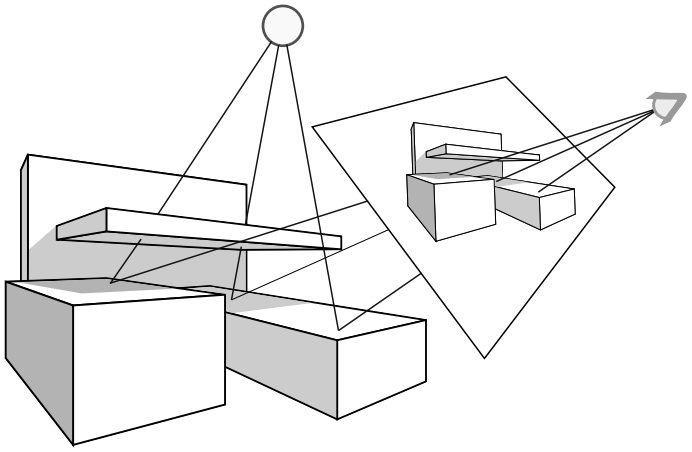

Abbildung 2: Backward Ray Tracing mit Sicht- und Schattenstrahlen.

Deutlich später, mit der Entwicklung leistungsfähigerer Hardware, wurde das Ray Tracing Verfahren konsequent weiterentwickelt. Da Bilder auf der Netzthaut oder einem Kamerasensor durch Licht entstehen, das auf Oberflächen der Umgebung trifft, je nach Materialeigenschaften der Oberflächen ggf. absorbiert, gebrochen oder wieder reflektiert wird und letztendlich ein bestimmter Anteil den Sensor erreicht, geht die Problemstellung über das Finden von Schatten hinaus. Gesucht ist die Beschreibung vom gesamten Licht, das von einem Oberflächenpunkt zum Auge transportiert wird (für jeden sichtbaren Oberflächenpunkt) (Glassner 1989c). Das erste Verfahren, welches diese Aspekte einbezieht, wird auch als „klassisches Ray Tracing“ bezeichnet und wurde von Turner Whitted vorgestellt (Whitted 1980). Es führt die Sichtbarkeitsberechnung nach dem Auffinden des nächsten Schnittpunktes fort und umfasst somit, neben den direkten Strahlen vom Auge, Schattenstrahlen vom Schnittpunkt zur Lichtquelle sowie das rekursive Weiterverfolgen von reflektierten und transmittierten Strahlen (Abb. 3). Damit sind, neben scharfen Schatten, einige globale Beleuchtungsinformationen wie (mehrfache) Reflexionen und Lichtbrechung darstellbar.

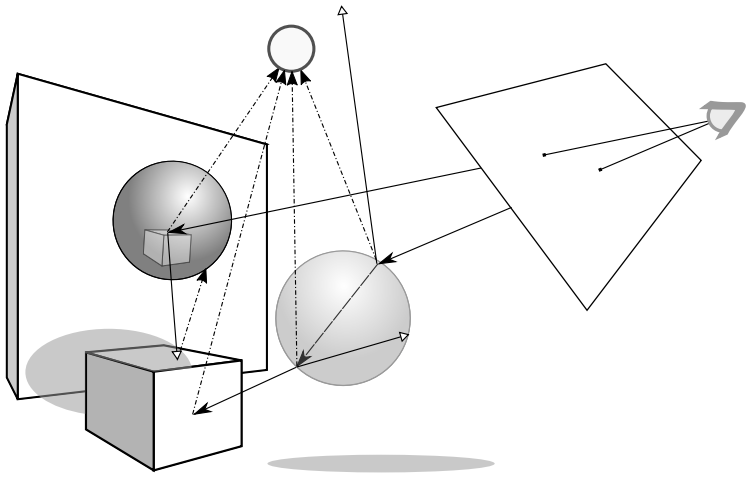

Abbildung 3: Whitted Ray Tracing: An jedem Schnittpunkt eines Strahls wird ein Schattenstrahl zur Lichtquelle verfolgt, je nach Material der getroffenen Oberfläche wird zusätzlich ein Strahl in Reflexionsrichtung (weiße Pfeilspitzen) und ggf. in Transmissionsrichtung erzeugt. Zur Verdeutlichung sind hier Schatten und Reflexion direkt in der Szene skizziert.

2.2.1 Antialiasing

Ein Problem der Bildsynthese und so auch des Ray Tracing (als eine Form von Point Sampling), ist räumliches Aliasing aufgrund der Diskretisierung durch das Pixelgrid. Wenn nur ein Strahl, also ein Sample pro Pixel, verfolgt wird, entstehen vor allem in Regionen starker Intensitätsveränderungen (z.B. an Objektkanten und Silhouetten) treppenartige Artefakte im Bild. Für eine präzisere und glattere Darstellung sind mehr Samples notwendig, die Szene muss also genauer abgetastet werden. Mehr Strahlen bedeuten jedoch auch mehr Berechnungsaufwand. Anti-Aliasing Verfahren versuchen daher, den Einsatz zusätzlicher Samples so gering wie möglich zu halten und dabei das sichtbare Aliasing weitestgehend zu reduzieren. Im Folgenden werden einige Samplingstrategien für Anti-Aliasing kurz vorgestellt. Ein einfaches und naives Verfahren ist das Supersampling. Dabei wird ein Pixel von einer festen Anzahl an Strahlen abgetastet und die Pixelfarbe wird aus den einzelnen Werten gemittelt. Das entspricht einer Unterteilung des Pixels in ein weiteres gleichmäßiges Raster. Je mehr Strahlen eingesetzt werden, desto weniger Aliasing tritt auf, aber gleichzeitig erhöht sich der Aufwand linear (Glassner 1989c). Besser ist das adaptive Supersampling, das zunächst eine feste Anzahl an Strahlen pro Pixel einsetzt, z.B. vier an den Ecken und einen in der Mitte, jedoch weiter unterteilt, wenn die Farben bei den einzelnen Strahlen sich ausreichend voneinander unterscheiden. In Regionen mit höheren Frequenzen wird somit feiner abgetastet. Problematisch ist,

dass ein Pixel weiterhin nur durch ein reguläres Grid unterteilt wird. Zudem kann die feste Anzahl an Samples zu Beginn dazu führen, dass initial bereits sehr kleine Objekte verpasst werden und somit weiterhin Aliasing entsteht (Glassner 1989c). Whitted hat adaptives Supersampling eingesetzt, allerdings lediglich mit vier Samples an den Pixel-Ecken (Whitted 1980).

Die Aliasing Probleme durch das reguläre Grid lassen sich durch stochastisches Sampling lösen. Die Samples werden nicht mehr gleichmäßig sondern „zufällig“ verteilt. Eine Möglichkeit stochastisches Sampling umzusetzen ist, die Samples zuerst in einem regulären Grid über ein Pixel zu verteilen und dann die Sample Positionen durch „jittering“ bzw. „Stratified Sampling“ (Addieren von Rauschen) zu verändern. Die resultierenden Abtastwerte können wiederum gemittelt werden, um die Pixelfarbe zu erhalten. Anstatt Aliasing kann Rauschen auftreten, was für das menschliche visuelle System jedoch deutlich weniger störend erscheint (Glassner 1989c) (Cook 1989). Eine Einführung in die Sampling- und Filter Theorie sowie in weitere Antialiasing Verfahren wie Multisampling (MSAA) lassen sich z.B. bei (Möller, Haines et al. 2008, Kapitel 5.6) finden.

2.2.2 Stochastisches Ray Tracing

Die zusätzlichen Strahlen, die für Antialiasing verwendet werden, eröffnen die Möglichkeit, weitere realistische Effekte per Ray Tracing darzustellen. In (Cook et al. 1984, Cook 1989) zeigen die Autoren, dass eine Verteilung der Strahlen entsprechend der analytischen Funktion, die sie abtasten, die Darstellung von Bewegungsunschärfe (Abtasten in Zeit), Tiefenunschärfe (Abtasten der Fläche der Kameralinse), weiche Schatten (Abtasten des Raumwinkels der Lichtquelle), weiche Reflexionen (Verteilung um die Reflexionsrichtung) und Transluzenz (Verteilung um die Transmissionsrichtung) erlaubt. Diese Erweiterung vermeidet einige der vereinfachenden Annahmen über den Lichttransport und wird verteiltes oder stochastisches Ray Tracing (Distributed Ray Tracing) genannt (Abb. 4). Damit sind nicht mehr nur Punktlichtquellen, sondern auch flächige Lichtquellen sowie weitere Materialeigenschaften und Kameraparameter zu simulieren.

Viele Aspekte des Lichttransports bzw. der globalen Beleuchtung lassen sich also durch Ray Tracing simulieren. Zwei auffällige Effekte fehlen jedoch: Diffuse Interreflexion (sichtbar in Form von „Color Bleeding“) und Kaustiken (Glassner 1989b). Diffuse Interreflexion lässt sich durch Monte-Carlo Ray Tracing (Path Tracing) erreichen, einer Erweiterung des stochastischen Ray Tracing, die von James T. Kajiya vorgestellt wurde (Kajiya 1986). Kaustiken werden durch eine Erweiterung um Photon Mapping (Jensen und Christensen 1995) darstellbar, wie z.B. auch in (Jensen und Christensen 2007) beschrieben, oder durch bidirektionales Path Tracing (Lafortune und Willems 1993).

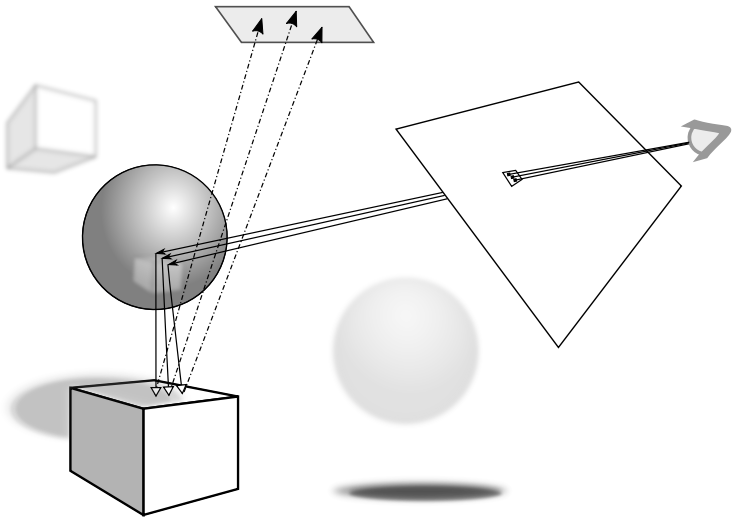

Abbildung 4: Distributed Ray Tracing: Mehrere Strahlen tasten ein Pixel ab (hier zur einfacheren Darstellung nur drei Strahlen), Verteilung der Schatten- und reflektierten Strahlen. Tiefenunschärfe und weiche Schatten sind zur Veranschaulichung in der Szene skizziert.

2.2.3 Schnittpunktberechnung mit einem Strahl

Mathematisch lässt sich ein Strahl wie in Gleichung 5 als Funktion über einem Parameter $t \in \mathbb{R}$ beschreiben, welche die Menge aller Punkte auf einer Linie erzeugt (parametrische Form).

$$\mathbf{r}(t) = \vec{\boldsymbol{o}} + t \cdot \vec{\boldsymbol{d}} \tag{5}$$

Dabei ist $\vec{\boldsymbol{o}}$ der Ursprung des Strahls und $t > 0$ eine Variable, die für alle Punkte auf dem Strahl den Abstand vom Ursprung in die Strahlrichtung $\vec{\boldsymbol{d}}$ angibt. Für die Schnittpunktberechnung sollte die Richtung des Strahls normalisiert werden, damit t die Distanz in Weltkoordinaten angibt und nicht durch die Länge des Richtungsvektors beeinflusst wird. Das grundlegende Prinzip bei der Berechnung des Schnittpunktes zwischen einem Strahl und einer Oberfläche ist das Einsetzen der Strahlgleichung in die entsprechende Oberflächengleichung. Oberflächen können mathematisch z.B. in impliziter ($f(x, y, z) = 0$), expliziter ($z = f(x, y)$) oder parametrischer ($x = f_x(u, v), y = f_y(u, v), z = f_z(u, v)$) Form beschrieben werden. Für die Schnittpunktberechnung zwischen einem Strahl und einer Kugel, beschrieben durch ihre implizite Gleichung aus Mittelpunk $\vec{\boldsymbol{c}}$ und Radius r (Gleichung 6), kann die Strahlgleichung (Gleichung 5) für einen Punkt $\vec{\boldsymbol{p}}$ auf der Kugeloberfläche eingesetzt werden (Gleichung 7), um zu prüfen ob ein Punkt auf dem Strahl auch auf der Kugeloberfläche liegt, also die Gleichung erfüllt. Dafür wird nach der Unbekannten t aufgelöst und man erhält ggf. den Schnittpunkt durch Einsetzen des Wertes für t in Gleichung 5.

$$f(\vec{\boldsymbol{p}}) = (\vec{\boldsymbol{p}} - \vec{\boldsymbol{c}})^2 - r^2 = 0 \tag{6}$$

$$f(\mathbf{r}(t)) = (\vec{\boldsymbol{o}} + t \cdot \vec{\boldsymbol{d}} - \vec{\boldsymbol{c}})^2 - r^2 = 0 \tag{7}$$

Ob ein Strahl ein Dreieck trifft kann zum Beispiel festgestellt werden, indem der Strahl zunächst auf einen Schnittpunkt mit der Ebene getestet wird, in der sich das Dreieck befindet, um dann ggf. zu prüfen ob der Schnittpunkt mit der Ebene innerhalb des Dreiecks liegt (Haines 1989). Dafür müssen die Ebenengleichungen entweder gespeichert, oder während der Schnittpunktsuche berechnet werden. Ein schneller Algorithmus, der ohne die Ebenengleichungen auskommt, wird in (Möller und Trumbore 1997) vorgestellt und benutzt zur Parametrisierung der Punkte innerhalb eines Dreiecks baryzentrische Koordinaten. Ein Dreieck sei gegeben durch seine Eckpunkte $\vec{\boldsymbol{v_1}}, \vec{\boldsymbol{v_2}}$ und $\vec{\boldsymbol{v_3}}$. Ein Punkt $\vec{\boldsymbol{p}}$ auf einer Dreiecksfläche kann dann durch seine baryzentrischen Koordinaten (u, v) angegeben werden als

$$\vec{\boldsymbol{p}} = (1 - u - v) \cdot \vec{\boldsymbol{v_1}} + u \cdot \vec{\boldsymbol{v_2}} + v \cdot \vec{\boldsymbol{v_3}}, \tag{8}$$

mit $u \geq 0$ und $v \geq 0$ sowie $u + v \leq 1$. Setzt man nun die Strahlgleichung 5 für den Punkt $\vec{\boldsymbol{p}}$ in Gleichung 8 ein und formt die resultierende Gleichung um (Gleichung 9), erhält man ein lineares Gleichungssystem, durch dessen Lösung die drei Unbekannten (t, u, v) zu finden sind (Gleichung 10).

$$\begin{aligned} \vec{\boldsymbol{o}} + t \cdot \vec{\boldsymbol{d}} &= (1 - u - v) \cdot \vec{\boldsymbol{v_1}} + u \cdot \vec{\boldsymbol{v_2}} + v \cdot \vec{\boldsymbol{v_3}} \\ \Rightarrow \vec{\boldsymbol{o}} - \vec{\boldsymbol{v_1}} &= t \cdot \vec{\boldsymbol{d}} + u \cdot (\vec{\boldsymbol{v_2}} - \vec{\boldsymbol{v_1}}) + v \cdot (\vec{\boldsymbol{v_3}} - \vec{\boldsymbol{v_1}}) \end{aligned} \tag{9}$$

$$[\vec{\boldsymbol{d}}, \vec{\boldsymbol{v_2}} - \vec{\boldsymbol{v_1}}, \vec{\boldsymbol{v_3}} - \vec{\boldsymbol{v_1}}] \cdot \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \vec{\boldsymbol{o}} - \vec{\boldsymbol{v_1}} \tag{10}$$

Geometrisch kann man sich den Vorgang vorstellen als eine Transformation des Dreiecks und des Strahls in ein Koordinatensystem, das durch u, v und t aufgespannt wird, so dass der Strahl entlang der t Achse verläuft und das Dreieck ein Einheitsdreieck in der Ebene von u und v ist (Möller und Trumbore 1997). Das lineare Gleichungssystem kann durch Anwendung der Cramerschen Regel gelöst werden.

Eine schöne Übersicht der Schnittpunktberechnung für Strahlen mit Oberflächen bieten (Haines 1989) und Kapitel 16 in (Möller, Haines et al. 2008). In (Hanrahan 1989) wird detailliert auf die Schnittpunktberechnung für verschiedene Oberflächenmodelle (implizite, explizite, parametrische Oberflächen) eingegangen.

2.3 Hierarchische Beschleunigungsdatenstrukturen

Alle Ray Tracing Beschleunigungsverfahren durch hierarchische räumliche Datenstrukturen basieren auf dem Ziel, den Berechnungsaufwand der Schnittpunkttests zu minimieren, indem zum einen die Anzahl auszuführender Tests reduziert wird und zum anderen komplexere Schnittpunktroutinen (für Primitive wie Dreiecke) weitestgehend durch einfachere ersetzt werden (für Bounding Volumes, Ebenen oder Voxel). Dafür konstruiert man eine hierarchische Datenstruktur bzw. Baumstruktur als Repräsentation der Szene und traversiert diese für die Schnittpunktsuche. Anstatt jeden Strahl mit jedem Primitiv der Szene zu testen wird geprüft, ob der Strahl, angefangen bei der Wurzel, einen Knoten des Baumes trifft und ggf. mit dem entsprechenden Kindknoten fortgefahren. Der Test auf einen Schnittpunkt zwischen Strahl und Primitiv(en) muss nur dann ausgeführt werden, wenn die Traversierung bis zu einem Blattknoten gelangt (also alle inneren Knoten auf dem jeweiligen Pfad vom Strahl getroffen wurden). Andernfalls kann abgebrochen werden, sobald kein Knoten mehr getroffen wurde. Der Aufwand für die Schnittpunktsuche lässt sich dadurch theoretisch von O(n) (linear mit Anzahl der Primitive) auf $O(log\, n)$ verringern.

Verfahren für den Aufbau solcher Beschleunigungsdatenstrukturen lassen sich in zwei grundlegende Strategien einordnen: die Raumunterteilung und die Objektunterteilung. Bei den Raumunterteilungsverfahren wird das gesamte Volumen, das die Szene umschließt, sukzessive durch Ebenen aufgeteilt. Die entstehenden Bereiche sind (in der Regel) räumlich disjunkt, Primitive können Bereiche überlappen und die Summe der Volumen aller Blattknoten ergibt das Ausgangsvolumen. Gängige Datenstrukturen dieser Art sind z.B. der Octree und Varianten des BSP-Tree (Binary Space Partitioning); im Ray Tracing Kontext kommt häufig der kd-Tree zum Einsatz (k-dimensionaler Baum), welcher ein Spezialfall des BSP-Baumes ist.

Bei den Objektunterteilungsverfahren hingegen, geht man nicht vom Raum sondern von den Objekten bzw. Primitiven aus, die durch einfachere Hüllkörper (Bounding Volume, BV) so platzsparend wie möglich komplett umschlossen werden. Das impliziert natürlich auch eine räumliche Aufteilung durch die Ausdehnung der Primitive und damit der begrenzenden Volumen. Im Gegensatz zur Raumunterteilung wird jedes Primitiv genau einem Volumen zugeteilt (die Menge von Primitiven verschiedener Volumen sind disjunkt), die Volumen können sich überschneiden und die Summe der Volumen aller Blattknoten kann kleiner sein als das Ausgangsvolumen. Diese Datenstruktur wird Bounding Volume Hierarchie (BVH) genannt (Abb. 5). Beide Strategien haben ihre Vor- und Nachteile, die sich je nach dem konkreten Verfahren mehr oder weniger stark äußern können. Havran zeigt, dass es möglich ist kd-Bäume durch BVHs und umgekehrt BVHs durch kd-Bäume mit sechs Dimensionen zu emulieren (Havran 2007), um darzulegen, dass Raum- und Objektunterteilungsverfahren in ihrer Ausdrucksfähigkeit

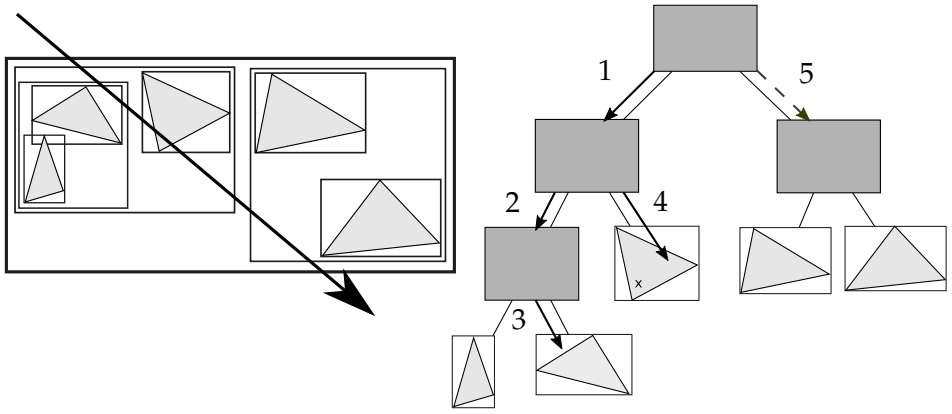

Abbildung 5: Bounding Volume Hierarchie in 2D und depth-first Traversierung (Pfeile) des entsprechenden Baums: der Strahl (Pfeil links) trifft die Wurzel und beide Kindknoten, der linke wird als erstes Traversiert (1), auf der nächsten Ebene werden wieder beide Kindknoten getroffen, wobei erneut der linke näher ist und zuerst besucht wird (2). Der Strahl trifft nun das rechte Blatt, aber nicht das enthaltene Dreieck. Der zuvor getroffene, noch nicht besuchte Knoten wird nun aufgesucht (4) und der Strahl trifft das enthaltene Dreieck. Schritt (5) wird nicht mehr durchgeführt, da die Distanz zum Schnittpunkt mit dem Dreieck kürzer ist als das zu prüfende BV.

bezüglich räumlicher Sortierung für Ray Tracing equivalent sind (nicht aber algorithmisch equivalent). Welche Strategie die Bessere bzw. algorithmisch Effizientere ist, kann nicht generell beantwortet werden, sondern hängt jeweils von der konkreten Implementierung auf einer bestimmten Hardware ab. BVHs bieten den Vorteil, dass sie neben der einfachen Konstruktion und geringem Speicherbedarf einfache Anpassungen bzw. Aktualisierungen für dynamische Szenen erlauben (Wald, Boulos et al. 2007). Untersuchungen darüber, welche Beschleunigungsstruktur auf GPUs hinsichtlich der Ray Tracing Performanz besser ist, variieren in den Ergebnissen je nach untersuchten Aufbau- und Traversierungsverfahren, dargestellten Szenen und verwendeter GPU Architektur. In Abschnitt 3.4 werden zwei aktuellere vergleichende Arbeiten besprochen.

In dieser Arbeit stehen Bounding Volume Hierarchien im Fokus und werden im nächsten Abschnitt einführend behandelt. Eine Einführung zu räumlichen Datenstrukturen inklusive Raumunterteilungsverfahren ist z.B. in (Arvo und Kirk 1989) und in Kapitel 14 von (Möller, Haines et al. 2008) zu finden, eine Übersicht zum Einsatz von Beschleunigungsdatenstrukturen für animierte Szenen lässt sich z.B. in (Wald, Mark et al. 2007) finden.

2.3.1 Bounding Volume Hierarchie

Im Folgenden werden Bounding Volume Hierarchien anhand grundlegender Arbeiten eingeführt. Eine formale Definition für eine BVH sei gegeben durch (nach Yoon und Manocha 2006 in angepasster Form):

Definition 1 *Eine Bounding Volume Hierarchie (BVH) ist ein gerichteter azyklischer Graph $G(\mathbf{N}, \mathbf{E})$, wobei $\mathbf{N}$ eine Menge von Bounding Volume Knoten BV_i^k (der i-te BV Knoten auf der k-ten Ebene der Hierarchie mit $n = |\mathbf{N}|$ Knoten und d Ebenen, mit $i, k, n, d \in \mathbb{N}$ wobei $0 < i \leq n$ und $0 < k \leq d$) ist und $\mathbf{E}$ eine Menge gerichteter Kanten von einem Knoten BV_i^k mit $k < d$, zu jedem Kindknoten $Left(BV_i^k)$ und $Right(BV_i^k)$ in der BVH. Ein Blattknoten BV_i^d hat keine ausgehenden Kanten. Eine BV-Anordnung einer BVH $G(\mathbf{N}, \mathbf{E})$ ist eine eins zu eins Abbildung, $\varphi\colon \mathbf{N} \to \{1, \ldots, n\}$, von BV Knoten auf Positionen in der BVH.*

Für die Wahl der Bounding Volumes, die eingesetzt werden um die Berechnungskosten für Schnittpunkttests zu verringern, gibt es verschiedene Möglichkeiten, die sich anhand folgender Kriterien qualifizieren: Schnell zu berechnende Schnitttests, möglichst wenig leerer Raum bei der Umhüllung der zugrunde liegenden Geometrie, schnelle Berechnung und wenig Speicheraufwand. Gängige Hüllkörper sind Kugeln (Bounding Spheres), oder konvexe Polyeder wie beliebig orientierte Quader (Oriented Bounding Box, OBB) oder nach den Hauptachsen ausgerichtete Quader (Axis Aligned Bounding Box, AABB) sowie diskret orientierte k-Polytope (Discrete Oriented Polytopes, k-DOPs). Im Rahmen dieser Arbeit werden als Bounding Volumes AABBs eingesetzt, die vor allem durch ihre Einfachheit bzgl. Schnitttests, Berechnung, Speicherbedarf und Unterteilung überzeugen (Wald, Mark et al. 2007). Eine AABB sei definiert als (nach Möller, Haines et al. 2008, Kapitel 16.2):

Definition 2 *Eine Axis Aligned bounding Box (AABB) ist ein Quader, bei dem die Normalen der sechs Flächen so ausgerichtet sind, dass sie mit den drei Standard-Hauptachsen des kartesischen Koordinatensystems übereinstimmen. Eine AABB kann beschrieben werden durch zwei seiner diagonal zueinander liegenden Eckpunkte $\vec{\boldsymbol{a}}_{min}$ und $\vec{\boldsymbol{a}}_{max}$, wobei gilt $\vec{\boldsymbol{a}}_{min}.i \leq \vec{\boldsymbol{a}}_{max}.i, \forall i \in \{x, y, z\}$*

Detaillierte Informationen und mathematische Definitionen zu den anderen Hüllkörpern können z.B. in Kapitel 16 von (Möller, Haines et al. 2008) gefunden werden.

Der erste Einsatz einer Bounding Volume Hierarchie im Ray Tracing wurde 1980 durch Rubin und Whitted demonstriert. Durch die hierarchische Unterteilung des Objektraumes konnten die Autoren eine deutliche Beschleunigung erreichen (Rubin und Whitted 1980). Als Bounding Volumes wurden Quader eingesetzt und so ausgerichtet, dass sie die repräsentierte

Geometrie möglichst eng umschließen können. Ausgehend von dem Volumen, das die gesamte Szene begrenzt und der Beschreibung der Geometrie, wurde der Objektraum in einem semi-automatischen Verfahren von einem Benutzer in weiteren Volumen gruppiert. Dabei musste der Benutzer beim Aufbau der Hierarchie Kriterien wie Kohärenz der Objekte und Passgenauigkeit der Volumen beachten. Dass dieses teilweise manuelle bottom-up Vorgehen sehr aufwendig war, sieht man daran, dass der Hierarchieaufbau mit (nur) hunderten von Blattknoten, laut den Autoren, ein oder zwei Tage benötigte.

Weghorst et al. argumentierten, dass es für den Aufbau der Hierarchie in der Regel ausreichend ist, nach der Strukturierung vorzugehen, die vom Benutzer bei der Modellierung der Ausgangsszene festgelegt wurde und so der Gruppierungsprozess von Elementen zu Bounding Volumes zu vereinfachen sei (Weghorst et al. 1984).

Kay und Kajiya präsentierten 1986 ein Verfahren für den automatischen Aufbau einer BVH zusammen mit Kriterien für eine gute Hierarchie und einem Traversierungsverfahren für Ray Tracing. Als BVs führten sie k-DOPs ein, welche die zugrunde liegenden Objekte durch eine gerade Anzahl an paarweise parallelen Ebenen umschließen. Der Raum zwischen jeweils zwei parallelen Ebenen wird als „Slab“ bezeichnet und die Ausrichtung durch „plane-set normals“ definiert (Kay und Kajiya 1986). Für den Schnitttest eines Strahls mit dem Volumen werden zunächst die Schnittpunkte für jedes Slab berechnet. Bei einem Treffer ergeben sich pro Slab zwei Distanzwerte, t_{min} und t_{max}. Dann wird der Schnitt zwischen diesen Intervallen aus dem Maximum der t_{min} Werte und dem Minimum der t_{max} Werte gebildet. Gilt nun für das verbleibende Intervall $t_{min} \leq t_{max}$, dann gibt es einen Schnittpunkt zwischen Strahl und BV. Für den Aufbau guter Hierarchien legen die Autoren folgende Kriterien fest:

- Räumliche Nähe der Objekte berücksichtigen: Jeder Teilbaum soll diejenigen Objekte beinhalten, die nah beieinander liegen.
- Das Volumen jedes Knotens (jedes BVs) soll minimal sein.
- Die Summe der Volumen aller BVs soll minimal sein.
- Die Konstruktion des Baumes sollte sich auf Knoten nahe an der Wurzel konzentrieren, da es größere Ersparnisse bringt, wenn ein Teilbaum nahe der Wurzel von weiteren Berechnungen ausgeschlossen werden kann.
- Die Zeit, die für den Aufbau der Hierarchie benötigt wird, soll sich in Bezug auf die Zeiteinsparung beim Rendering auszahlen.

Der Aufbau der Hierarchie erfolgt bei Kay und Kajiya von der Wurzel ausgehend, top-down, und basierend auf dem Median der Objekte („median-cut

scheme“). Auf jeder Ebene werden zunächst alle Objekte im Volumen entlang einer Achse des Koordinatensystems (alternierend, beginnend mit der x-Achse) sortiert. Die Aufteilung in zwei Subvolumen erfolgt dann anhand des Medians der Objekte entlang der Achse. Dieser Prozess wird rekursiv fortgeführt. Durch die Verwendung des Median wird zwar die Nähe der Objekte zum Teil berücksichtigt, allerdings nur entsprechend der jeweiligen Achse, die verwendet wurde. Das kann zu schlechten Unterteilungen führen, bei denen das Kriterium der räumlichen Nähe verletzt wird.

Die Traversierung der Hierarchie erfolgt in einer Reihenfolge, die während dem Traversierungsprozess durch eine Priority Queue (implementiert als Heap) bestimmt wird. Für den vordersten Knoten in der Queue (inital die Wurzel), werden seine Kindknoten auf Schnitt mit dem Strahl getestet. Getroffene Knoten werden als Kandidatenknoten anhand der Distanz zum Schnittpunkt (t_{min}) in die Queue eingeordnet. So wird als nächstes stets derjenige Knoten verarbeitet, der die geringste Distanz entlang des Strahls aufweist. Im Falle eines Blattknotens entspricht dies der geschätzten Distanz zum gekapselten Objekt, so dass in der Queue eine totale Ordnung für die Abarbeitung von Strahl-Objekt Schnitttests entsteht. Wird ein Objekt getroffen und die tatsächliche Schnittpunkt Distanz ist kleiner als zu einem zuvor getroffenen Objekt, handelt es sich um einen Kandidaten für den vordersten Schnittpunkt. Die Traversierung terminiert, wenn die Queue leer ist, oder wenn der vorderste Kandidatenknoten in der Queue (und damit alle verbleibenden) eine größere Distanz aufweist, als ein zuvor getroffenes Objekt. Andernfalls muss weiter getestet werden, da der erste Treffer mit einem Objekt nicht unbedingt zum tatsächlich sichtbaren Objekt gehört (BVs können sich überlappen). Durch die Ordnung für die Strahl-Objekt Schnittpunktsuche erfolgt die Verarbeitung der gegebenen Objekte unabhängig von der BVH in der gleichen Reihenfolge, so dass der Unterschied zwischen einer gut oder schlecht konstruierten Hierarchie, laut den Autoren, nur geringfügig ausfällt. Zwar werden durch eine schlechtere Hierarchie dadurch nicht mehr Strahl-Objekt Schnittpunkttests berechnet, allerdings mehr Strahl-BV Tests, was durchaus mehr Berechnungsaufwand bedeutet.

Kay und Kajiya haben zwar einige nachvollziehbare Kriterien für den Aufbau guter Hierarchien angesprochen, jedoch gleichzeitig die Bedeutung der Qualität einer Bounding Volume Hierarchie in Bezug auf das vorgestellte Traversierungsverfahren relativiert. Zudem wurden noch keine geeigneten Maßnahmen zur Sicherung und Messung der Qualität diskutiert. Goldsmith und Salmon adressieren 1987 das Problem der Qualität von BVHs und argumentieren, dass sich die Berechnungszeit des Ray Tracing Prozesses bei verschiedenen Bäumen leicht um das Fünfzigfache unterscheiden könne (Goldsmith und Salmon 1987). Als Metrik für die Qualität eines Baumes wird hier die Anzahl der Bounding Volume Schnitttests verwendet, die beim

Ray Tracing mit dem Baum stattfinden. Somit besteht direkter Bezug zur Ray Tracing Performanz bei Verwendung des Baumes. Die Autoren stellen fest, dass manuell erstellte Bäume in der Regel nicht gut geeignet sind, um Ray Tracing zu beschleunigen, und auch die Hierarchie, die beim Modellaufbau entsteht, aufgrund divergierender Ziele keine ausreichende Qualität liefert. Ansätze für den automatischen Aufbau, wie z.B. von Kay und Kajiya, seien besser, jedoch weiterhin nicht ausreichend auf das Ziel ausgerichtet, den Rendering Prozess (in dem Fall Ray Tracing) zu beschleunigen, da viel Potenzial verloren gehe. Goldsmith und Salmon präsentieren eine Lösung in Form einer Kostenfunktion und einer darauf basierenden Heuristik, die eingesetzt werden kann, um den automatischen Aufbau von Bounding Volume Hierarchien zu lenken, so dass Bäume von hoher Qualität erzeugt werden.

Als Basis für die Kostenfunktion dient die bedingte Wahrscheinlichkeit für das Treffen eines BV durch einen Strahl, wenn er bereits das BV der Wurzel getroffen hat. Für Strahlen mit Ursprung in einer festen Distanz zu einem BV ist die Wahrscheinlichkeit, dass ein beliebiger solcher Strahl das BV trifft, proportional zum Raumwinkel, der durch die Oberfläche des BV aufgespannt wird. Für konvexe Objekte ist dies auf großer Distanz annähernd proportional zur Größe der Oberfläche, welche sich für Quader der Maße $a \times b \times c$ nach Gleichung 11 berechnen lässt.

$$S(BV) = 2ab + 2ac + 2bc \tag{11}$$

Die Beziehung zwischen der Oberfläche und der Wahrscheinlichkeit, dass ein beliebiger Strahl aus großer Distanz ein BV trifft, ist annähernd linear, was intuitiv erscheint. Basierend auf „Random Search“ Verfahren (Stone 1976, Kapitel 1) werden für die Heuristik folgende vereinfachende Annahmen vorausgesetzt: Strahlen starten außerhalb der Wurzel in ausreichend großer Distanz, und Strahlen sind (im Suchraum bzw. hier Objektraum) zufällig und gleichverteilt (Goldsmith und Salmon 1987, MacDonald und Booth 1990). Da sich alle BVs im Wurzel BV befinden, und das Treffen der Wurzel vorausgesetzt wird, kann die bedingte Wahrscheinlichkeit für das Treffen eines Bounding Volumes unter obigen Annahmen, mit der Oberfläche des entsprechenden BV $S(BV_i)$ und des Wurzel-BV $S(BV_{root})$, durch Gleichung 12 angenähert werden.

$$P(hitBV_i|hitBV_{root}) = S(BV_i)/S(BV_{root}) \tag{12}$$

Die durchschnittlichen Gesamtkosten (bzgl. Anzahl Schnitttests) für einen Knoten BV_i entsprechen dem Produkt aus der Anzahl seiner Kindknoten $N_c(BV_i)$ und der bedingten Wahrscheinlichkeit $P(hitBV_i|hitBV_{root})$ (Gleichung 13).

$$C(BV) = N_c(BV) \cdot S(BV)/S(BV_{root}) \tag{13}$$

Die Kosten für einen Baum mit n Knoten lassen sich dementsprechend aus der Summe der Kosten seiner Knoten abschätzen, der Aufwand dafür beträgt $O(n)$. Während dem Aufbau eines Baumes werden lediglich Vergleiche zwischen den Wahrscheinlichkeiten (in Bezug auf die Oberflächen) benötigt, daher kann die Division entfallen. Für die Bewertung eines gesamten Baumes hingegen ist eine entsprechende Skalierung der Wahrscheinlichkeiten notwendig, wobei die Division herausfaktorisiert werden kann. Auch die Oberflächenberechnung ist zu vereinfachen, da nur das Verhältnis zwischen Oberflächen benötigt wird. Somit ergibt sich für Quader, wie AABBs, die vereinfachte Oberflächenberechnung nach Gleichung 14.

$$S_{approx}(BV) = (a + b)c + ab \tag{14}$$

Der Hierarchieaufbau beginnt bei Goldsmith und Salmon an der Wurzel. Objekte werden nach und nach in den entstehenden Baum eingefügt, indem für jedes Objekt der Baum nach einem geeigneten Platz durchsucht wird. Diese Baumsuche hat eine Zeitkomplexität von $O(log\, n)$. Von einem gegebenen Knoten aus betrachtet man dafür die Zunahme der Volumenoberfläche $S(BV)$, welche das Einfügen des neuen Objekts als Kindknoten mit sich bringen würde. Die Einfügung bzw. die weitere Suche findet in dem Teilbaum statt, der die geringsten Kosten (repräsentiert durch die Zunahme der Oberflächengröße bei Einfügung) für den betrachteten Knoten aufweist. Bei gleichen Kosten bzgl. mehrerer Knoten können entweder alle entsprechenden Teilbäume durchsucht werden, oder man kann nach der Nähe des Objekts zum Zentrum des Bounding Volume entscheiden. Alternativ kann man das Objekt zufällig zuordnen. Insgesamt ergibt sich eine Zeitkomplexität von $O(n\, log\, n)$ für den Aufbau. Während des Aufbaus kann die Kostenabschätzung für den gesamten Baum inkrementell mitberechnet werden. Folgende Fälle können auftreten:

- Ein neuer Knoten BV_c wird einem Knoten BV als direkter Kindknoten hinzugefügt, so dass BV zu BV_e erweitert wird. Die inkrementellen Kosten für BV_e sind: $(S(BV_e) - S(BV) \cdot N_c(BV_e) + S(BV_e)$, wobei $N_c(BV_e)$ die Anzahl an Kindknoten des erweiterten Knotens BV_e ist.
- Ein neuer Knoten BV_c wird mit einem Blattknoten BV_l vereinigt, ein neuer Elternknoten BV_e entsteht und hat die inkrementellen Kosten: $2 \cdot S(BV_e)$.
- Keiner der beiden Fälle tritt ein, sondern es wird auf der nächsten Ebene weiter gesucht, also bei den Kindknoten des zuvor betrachteten Knotens. Inkrementelle Kosten für diesen Vorfahren BV, dessen Oberfläche dadurch vergrößert wird ($S(BV_e)$), müssen beachtet werden: $(S(BV_e) - S(BV)) \cdot No_c(BV)$.

Die Reihenfolge, in der die Objekte in den Baum eingefügt werden, beeinflusst die Qualität der Hierarchie. Die Autoren stellen fest, dass die

Ordnung aus dem Modellierungsprozess eine gute Reihenfolge darstellen kann, jedoch für den Fall einer schlechten vordefinierten Ordnung bessere Ergebnisse durch Durchmischen erreicht werden. Am schlechtesten fielen die Tests mit Sortierung der Objekte aus (nähere Angaben zu Sortierverfahren und Kriterien wurden nicht angegeben).

Die Auswertung der Autoren zeigt, dass die erwartete Anzahl an Schnitttests pro Strahl für primäre Strahlen nah an der tatsächlichen Anzahl liegt, während die Anzahl für sekundäre Strahlen in der Regel etwas höher ausfällt als der erwartete Wert.

Die Heuristik, basierend auf den Oberflächen der Knoten, erhielt später den Namen „Surface Area Heuristic“ (SAH) und wurde zur bekanntesten Methode für die Maximierung der Traversierungs-Effizienz beim Aufbau hierarchischer Datenstrukturen (Wald 2007). Ihre Namensgeber, MacDonald und Booth, haben die Kostenfunktion weiter formalisiert, um konstante Kosten-Faktoren ergänzt und ein Konstruktionsverfahren für kd-Bäume entwickelt, das die Gesamtkosten für einen Baum minimieren soll, indem die zu unterteilenden Knoten gemäß der SAH gewählt werden (MacDonald und Booth 1990). Die neuen Faktoren sind im Einzelnen die Kosten für die Traversierung eines inneren Knotens C_i, eines Blattknotens C_l und für den Schnittest mit einem Objekt C_o. Die Kostenabschätzung für einen gesamten Baum ist in Gleichung 15 gezeigt.

$$C_{tree} = \frac{C_i \cdot \sum_{i=1}^{n_i} S(i) + C_l \cdot \sum_{l=1}^{n_l} S(l) + C_o \cdot \sum_{l=1}^{n_l} S(l) \cdot N(l)}{S(BV_{root})} \tag{15}$$

Dabei bezeichnet n_i die Anzahl innerer Knoten, n_l die Anzahl der Blattknoten und $N(l)$ die Anzahl von Objekten im Blattknoten l. Darüber hinaus leiten die Autoren eine Funktion ab, die ausgehend von einem Parameter für die Positionierung der Unterteilungsebene in einem Knoten des kd-Trees die Kosten für die entsprechende Unterteilung abschätzt. Dies kann wiederum für die Anwendung bei der Unterteilung von Bounding Volumes angepasst werden, wie z.B. in (Wald, Boulos et al. 2007, Wald 2007). Diese und weitere BVH Konstruktionsverfahren sowie (GPU) Traversierungsverfahren werden in Abschnitt 3.2 behandelt.

2.4 GPU Architektur und parallele Programmierung

Die Entwicklung von der festen Graphikpipeline hin zur Integration programmierbarer Shader begann 2001 (Nvidia GeForce 3) mit dem Vertex Shader, gefolgt vom Fragment Shader, deren Fähigkeiten, Flexibilität und Ressourcen in den folgenden Jahren stetig erweitert wurden (Möller, Haines et al. 2008, Kirk und Hwu 2010). Purcell et al. beschreiben 2002 auf Basis der ihnen zur Verfügung stehenden Informationen, ihre Einschätzung der zukünftigen Entwicklung von GPUs zu parallelen (general purpose) Streaming

Prozessoren (Purcell et al. 2002). Durch eine entsprechende Weiterentwicklung der Hardwareplattform und der Programmierbarkeit wurde die GPU aufgrund ihrer hohen Leistungsfähigkeit bei parallelen Berechnungen für Anwendungen außerhalb des Graphik Kontextes interessant, z.B. allgemein für rechenintensive wissenschaftliche Probleme. Daraus entstand das „General Purpose Programming“ auf GPUs (GPGPU). Zu Beginn mussten die jeweiligen Probleme auf die verfügbaren Graphik APIs abgebildet werden, um die GPU für die Berechnungen ausnutzen zu können. Das änderte sich 2007 mit Nvidias „Compute Unified Device Architecture“ (CUDA), welche sowohl als grundlegende Hardwareplattform auf Nvidia GPUs, als auch als API für den GPGPU Bereich eingeführt wurde. GPUs (von Nvidia) wurden dadurch als hochparallele streaming Prozessoren unabhängig von Graphik APIs nutzbar. Ungefähr zwei Jahre später folgte mit OpenCL auch ein offenes Programmiermodell, das treiberseitig auf GPUs verschiedener Hersteller unterstützt wurde. Beide APIs unterscheiden zwischen Host (z.B. CPU) und Device (z.B. GPU) und vom Host werden die Kernel (Funktionen bzw. Programme) aufgerufen, die auf dem Device ausgeführt werden. Aber auch im Kontext von Graphikanwendungen, die auf Graphik APIs wie OpenGL basieren, gibt es Probleme, die sich nicht so gut auf die Graphik Pipeline abbilden lassen. Die Ergänzung durch weitere APIs wie CUDA oder OpenCL ist möglich, erfordert aber den Kontextwechsel. Seit 2013 gibt es eine mögliche Alternative direkt in OpenGL integriert, die Compute Shader (siehe Abschnitt 2.5).

Moderne GPUs sind organisiert in Streaming Multiprozessoren (SM), bei AMD „Compute Unit“ genannt, welche wiederum eine Anzahl an Streaming Prozessoren (SP), bei Nvidia als „Cuda Cores“ bezeichnet, besitzen. Streaming Prozessoren verarbeiten Instruktionen für mehrere Threads gleichzeitig und teilen Kontroll-Logik und bestimmte Ressourcen eines SM wie L1 Cache, Shared Memory und Befehlsspeicher (Instruction Cache). Der SM hält zudem eine Registerdatei, dessen Plätze an die Threads verteilt werden. Neben den sehr begrenzten, aber schnellen Chip-lokalen Speicher-Ressourcen, steht der GPU der globale Speicher, GDDR DRAM („graphics double data rate dynamic random access memory“), zur Verfügung, welcher für aktuelle Desktop GPUs zwischen 4 und 12 GB beträgt. Eine Menge von auszuführenden Threads (Thread Block in CUDA, Work Group in OpenGL Compute Shader, siehe Abschnitt 2.5) wird vom SM in Gruppen, genannt Warps, eingeteilt und eine gemeinsame Anweisung wird für alle Threads eines Warps parallel ausgeführt. Ein SM kann mehrere Warps (z.B. bis zu 64 auf Nvidias Maxwell Architektur, Nvidia 2015b) unterbringen. In der Regel umfasst ein Warp auf Nvidia Hardware 32 Threads. Auf AMD Hardware werden Gruppen von 64 Threads gebildet und als Wavefront bezeichnet.

Das Ausführungsmodell auf Basis dieser hochparallelen Architektur wird SIMT (single instruction multiple threads) genannt (Kirk und Hwu

2010 Kapitel 6, Laine et al. 2013, Nvidia 2015a). In SIMT verarbeiten alle Threads die Daten in eigenen Registern. Die Programmierung für SIMT auf GPUs ist verglichen mit dem SIMD (single instruction multiple data) Modell auf CPUs einfacher, da die einzelnen Datenelemente, die durch eine Instruktion verarbeitet werden sollen, nicht explizit aufbereitet und in ein gemeinsames Register gepackt werden müssen (Kirk und Hwu 2010, Kapitel 6). Zudem wird SIMT Thread Divergenz durch die Hardware behandelt. Die Vektor Organisation, die SIMD erfordert und die manuelle Behandlung von Divergenz ist für SIMT somit nicht notwendig. Thread Divergenz durch datenabhängige Verzweigungsbedingungen innerhalb eines Warps führt jedoch zu Einbußen in der Performanz, da beide Pfade ausgeführt werden müssen, wobei Threads, die nicht zum aktuellen Pfad gehören, temporär deaktiviert werden (Günther et al. 2007, Aila und Laine 2009, Garanzha und Loop 2010, Kirk und Hwu 2010, Nvidia 2015a). Divergenz durch Kontrollflussanweisungen innerhalb von Warps sollte daher, soweit möglich, vermieden werden. Häufig wird auch im Kontext des SIMT Modells und der zugrunde liegenden Architektur von einer Ausführung nach „SIMD Art und Weise“ und von „SIMD Effizienz“ gesprochen. Das bezieht sich auf die Gemeinsamkeiten der Ausführungsweise (eine Instruktion für zahlreiche Elemente gleichzeitig).

Warps, die bereit sind ihre nächste Anweisung auszuführen, werden durch Warp Scheduler ausgewählt und die SPs führen die Anweisung für die Threads aus. Die Zeit in Clock Cycles, welche ein Warp benötigt um für die Ausführung der nächsten Anweisung bereit zu sein, wird als Latenz bezeichnet. Die Fähigkeit der Warp Scheduler Latenz zu verstecken hängt davon ab, ob genügend Warps auf dem SM verfügbar sind, so dass im optimalen Fall jeder Scheduler zu jedem Clock Cycle eine Anweisung an die aktiven Threads geben kann, so dass die Hardwareleistung voll genutzt wird (Hardware Multithreading) (Nvidia 2015a). Die Anzahl an Work Groups und Warps, die zusammen auf einem SM verarbeitet werden können, hängt stark von der Nutzung der lokalen Speicherressourcen ab. Besonders die Anzahl der Register und die Menge an geteiltem Speicher (Shared Memory) sind in dieser Hinsicht limitierende Faktoren. Einem SM steht eine gewisse Menge an Registern (und Shared Memory) zur Verfügung, die er auf die zu verarbeitenden Work Groups bzw. Threads aufteilen muss. Je mehr Register und geteilte Variablen ein Kernel (hier Compute Shader Programm) verwendet, desto niedriger wird die Anzahl gleichzeitig verfügbarer Warps und damit auch die Fähigkeit der Hardware die Latenzen (z.B. bei Zugriff auf den DRAM) zu verstecken (Günther et al. 2007, Kirk und Hwu 2010, Kapitel 6, Wald 2011, Laine et al. 2013, Nvidia 2015a). Dementsprechend wird die Leistung der Hardware nicht mehr voll genutzt, wenn ein Kernel eine große Menge der verfügbaren Register belegt und gleichzeitig Latenzen entstehen. Als Beispiel sei Nvidias Maxwell Architektur (zweite Generation) betrachtet. Der GM204 Chip hat je nach Ausführung 16 (GeForce GTX 980)

oder 13 (GTX 970) SMs, die jeweils 128 SPs (CUDA Cores) enthalten (Bell 2014). Ein SM kann maximal 32 Work Groups (Thread Blocks) bzw. 64 Warps gleichzeitig unterbringen. Jedem SM steht eine Register-Datei mit 64.000 Registern zu jeweils 32 Bit zur Verfügung, von denen maximal 255 Register pro Thread genutzt werden können. Die Kapazität des Shared Memory beträgt 96KB, wobei pro Thread Block 48KB nutzbar sind, empfohlen wird maximal 32KB pro Thread Block zu nutzen (Nvidia 2015b). Anhand der Daten lässt sich berechnen, dass pro SM maximal $64 \cdot 32 = 2048$ Threads gleichzeitig verarbeitet werden können. Bei 16 SMs können insgesamt 32.768 Threads auf dem GM204 untergebracht werden, z.B. in 512 Work Groups zu jeweils 64 Threads (2 Warps), wobei jede Work Group 2000 Registerplätze nutzen könnte (rund 31 pro Thread). Angenommen ein Kernel verwendet 62 Registerplätze und die Größe der Work Groups sei definiert als 16×16 (256 Threads), dann werden pro Work Group 15.872 Register benötigt. Somit können 4 Work Groups pro SM untergebracht werden und von den verfügbaren 2048 Thread Slots werden 1024 genutzt. Wird die Registernutzung des Kernels nur um 1 Register erhöht, finden 3 Work Groups auf einem SM Platz, so dass nur noch 768 Thread Slots pro SM in Benutzung sind. Nur ein Register mehr führt in diesem (vereinfachten) Beispiel also zu einer Verringerung der parallel verfügbaren Elemente pro SM um 25%. Neben dem Bestreben möglichst wenige Register zu nutzen, ist jedoch auch abzuwägen, ob die Nutzung zusätzlicher Register (oder auch geteilter Variablen) Vorteile bringen kann, welche die geringere Parallelität aufwiegen, z.B. wenn Lesezugriffe auf den globalen Speicher reduziert werden können.

Zusammenfassend lassen sich folgende grundlegende Anhaltspunkte für die GPU Programmierung festhalten:

- Zerlegen der Arbeit, so dass die parallele Architektur ausgenutzt wird,
- Anzahl verwendeter Register minimieren,
- Menge von verwendetem Shared Memory minimieren,
- aber Register oder Shared Memory gewinnbringend einsetzen, um Zugriffe auf den globalen Speicher zu reduzieren,
- Vermeiden von divergentem Kontrollfluss innerhalb eines Warps.

2.5 OpenGL Compute Shader

Compute Shader haben keinen festen Platz in der Graphik Pipeline und keine vordefinierte Verbindung zu den Graphik Shader Stufen. Somit können sie an beliebiger Stelle ausgeführt werden. Man kann sich Compute Shader als eine Pipeline mit nur einer Stufe vorstellen, oder nach der OpenGL 4.3 Kernspezifikation als „*single stage machine that runs generic shaders*“ (Segal und Akeley 2013). Abgesehen von einigen internen Variablen, die Daten

über den Ausführungszustand festhalten, haben Compute Shader keine vordefinierten Ein- und Ausgaben. Sämtliche Eingabedaten und Ergebnisse einer Ausführung müssen explizit aus dem Speicher gelesen bzw. in den Speicher geschrieben werden, z.B. in Form von Uniform Buffer Objects (UBO, nur Lesezugriff) Bild Objekten, Texturen, oder Shader Storage Buffer Objekten (SSBO). Compute Shader werden für räumlich definierte Arbeitsgruppen ausgeführt, im Folgenden Work Groups genannt. Beim Aufruf (Dispatch) eines Compute Shaders wird dem Programm die Anzahl der Work Groups, mit denen der Compute Shader auszuführen ist, in ein- bis drei Dimensionen übergeben (Listing 1). In dem so festgelegten „Indexraum" sind die Work Groups angeordnet und lassen sich bei der Ausführung identifizieren. Die Verarbeitungs-Reihenfolge für die Work Groups aus einem Dispatch wird vom System bestimmt und ist nicht notwendigerweise nach den Indizes geordnet.

Listing 1: Compute Shader Ausführung

```
void glDispatchCompute(GLuint num_groups_x,
                       GLuint num_groups_y,
                       GLuint num_groups_z);

//eine Alternative benötigt als Parameter ein Offset in ein
//Buffer Objekt, das die Parameter (wie oben) enthält:
void glDispatchComputeIndirect(GLintptr indirect);
```

Manchmal wird in dem Zusammenhang von einer „global Work Group" gesprochen, welche für die Ausführung im Shader wiederum in die entsprechende Anzahl von „local Work Groups" eingeteilt wird (Wright et al. 2016). Der Begriff Work Group ohne weitere Eingrenzung bezieht sich dann in der Regel auf die lokalen Arbeitsgruppen. Im Folgenden soll der Work Group Term gemäß der Spezifikation (Segal und Akeley 2016) verwendet werden, d.h. der Begriff der „globalen Work Group" entfällt und es gilt:

- Beim Dispatch wird die **Anzahl der auszuführenden Work Groups** in ein bis drei Dimensionen angegeben: (*num_groups_x*, *num_groups_y*, *num_groups_z*),
- im Compute Shader wird die **lokale Größe** der **Work Groups** in ein bis drei Dimensionen definiert: (*local_size_x*, *local_size_y*, *local_size_z*),
- eine Work Group besteht aus n **Work Items**, wobei $n = local\_size\_x \cdot local\_size\_y \cdot local\_size\_z$ der lokalen Größe der Work Group entspricht (definiert durch *layout qualifier* im Shader).
- Ein **Work Item** wird durch einen Shader Aufruf verarbeitet. Eine Work Group kann also auch verstanden werden, als eine Menge von Shader Aufrufen, die den gleichen Code, wenn möglich parallel, ausführen.

- Ein Dispatch führt ($num\_groups\_x \cdot num\_groups\_y \cdot num\_groups\_z$) · ($local\_size\_x \cdot local\_size\_y \cdot local\_size\_z$) Aufrufe des Compute Shaders aus.

Darüber hinaus soll der Bereich, der die Anzahl der auszuführenden Work Groups definiert, hier als Indexraum bezeichnet werden. In Abbildung 6 ist die hierarchische Aufteilung des Indexraums dargestellt. Tabelle 1 zeigt die Entsprechungen der oben aufgeführten Terminologie zwischen OpenGL Compute Shader, CUDA und OpenCL (Segal und Akeley 2016, Nvidia 2015a, Howes 2015).

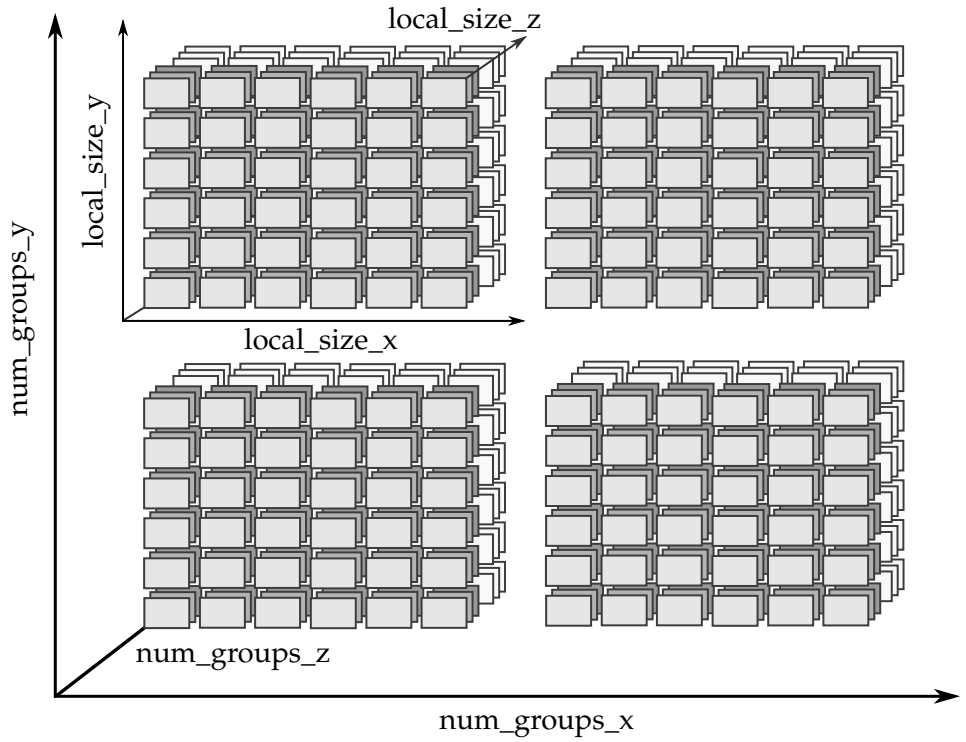

Abbildung 6: Der Indexraum umfasst $num\_goups\_x \times num\_groups\_y \times num\_groups\_z$ Work Groups der Größe $local\_size\_x \times local\_size\_y \times local\_size\_z$ bestehend aus Work Items.

Tabelle 1: Äquivalenz der Terme

OpenGL Compute Shader	CUDA	OpenCL
(Indexraum / global Work Group)	Grid	NDRange
Work Group	Thread Block	Work Group
Work Item	Thread	Work Item

Die Semantik des Indexraums und der Work Groups wird erst durch die Anwendung bestimmt. So könnte z.B. die Verarbeitung von Bilddaten ein

zweidimensionales Problem darstellen. Entsprechend werden die x und y Dimensionen der Work Groups gesetzt und repräsentieren Bildbereiche; die Work Items arbeiten dann auf Bildpunkten.

Die erwähnten internen Variablen des Compute Shaders geben Auskunft über die Konfiguration für die Ausführung und erlauben es, sowohl Work Groups als auch einzelne Work Items (Aufrufe) zu identifizieren:

- `in uvec3 gl_NumWorkGroups` Anzahl Work Groups in 3 Dimensionen (wie in `glDispatchCompute`).
- `const uvec3 gl_WorkGroupSize` Größe der Work Groups in 3 Dimensionen (wie im `layout` Qualifizierer).
- `in uvec3 gl_WorkGroupID` ID der Work Group des aktuellen Shader Aufrufs in $[0,$ `gl_NumWorkGroups`$]$).
- `in uvec3 gl_LocalInvocationID` lokale ID des Shader Aufrufs innerhalb der Work Group in $[0,$ `gl_WorkGroupSize`$]$).
- `in uvec3 gl_GlobalInvocationID` eindeutige ID des Shader Aufrufes in der Gesamtheit aller Aufrufe durch den Compute Dispatch.
- `in uint gl_LocalInvocationIndex` lokale Aufruf ID (innerhalb einer Work Group in 1D, auf `gl_LocalInvocationID` und `gl_WorkGroupSize` basierend).

Wie andere OpenGL Shader, werden auch Compute Shader in GLSL geschrieben. Auch die Erstellung läuft analog zu anderen Shader Programmen, mit dem Unterschied, dass als Shader Typ `GL_COMPUTE_SHADER` übergeben wird (Listing 2). Das Programm Objekt wird durch `glUseProgram()` für den aktuellen Rendering Zustand aktiviert und per Dispatch Befehl (siehe Listing 1) ausgeführt. Im Shader Code selbst wird die lokale Größe der Work Groups per Layout Qualifier definiert (Listing 3).

Listing 2: Erstellen des Shader Objekts, Laden des Shader Codes, Kompilieren, Binden an Programm Objekt und Verknüpfen des Programm Objekts in OpenGL.

```
GLuint computeShader = glCreateShader(GL_COMPUTE_SHADER);
glShaderSource(computeShader, 1, shaderSource, NULL);
glCompileShader(computeShader);
GLuint computeProgram = glCreateProgram();
glAttachShader(computeProgram, computeShader);
glLinkProgram(computeProgram);
```

Die Wahl einer geeigneten lokalen Größe hängt von dem jeweiligen Compute Shader Programm und den verwendeten Ressourcen (vor allem Register und Shared Memory) sowie von der zugrunde liegenden Hardware

und den Treibern ab. Es wird daher empfohlen, eine geeignete Konfiguration experimentell zu ermitteln, wobei die Anzahl an Work Items pro Work Group so gewählt werden sollte, dass sie einem Vielfachen der Warp Größe entspricht (Bailey 2013, Nvidia 2015a, Kapitel 5), denn die Threads der Work Groups werden vom Multiprozessor der GPU in Warps von 32 Threads (für Nvidia GPUs) eingeteilt (bei AMD GPUs „Wavefronts“ von 64 Threads).

Listing 3: Compute Shader in GLSL, mit Work Group Größe 32 x 32

```
#version 450 core
layout (local_size_x = 32, local_size_y = 32) in;
void main(void)
{
// Hier wird gearbeitet...
}
```

Die Threads eines Warps laufen parallel in lock-step eine Instruktion nach der anderen ab, während darüber hinaus alle Aufrufe der selben Work Group wenn möglich zwar parallel, aber nicht in lock-step ausgeführt werden. Allerdings können alle Threads bzw. Aufrufe der selben Work Group über geteilte Variablen (Shared Variables) und Atomic Operations kommunizieren. Für Threads aus verschiedenen Work Groups besteht diese Möglichkeit nicht. Die geteilten Variablen werden durch den `shared` Qualifizierer definiert und in einem dafür vorgesehenen Speicherbereich gehalten, so dass der Zugriff deutlich schneller erfolgt als Speicherzugriffe auf den globalen Speicher. Für den geteilten Speicher steht jedoch deutlich weniger Platz zur Verfügung, und dieser ist zwischen allen gleichzeitig aktiven Work Groups aufgeteilt. Laut OpenGL Spezifikation werden minimal 32KB vorausgesetzt, die "MaxwellArchitektur von Nvidia stellt 96KB zur Vefügung, begrenzt auf Maximal 48KB pro Work Group bzw. Thread Block (Nvidia 2015b). Der Wert kann abgefragt werden durch:

```
void glGetIntegerv(GLenum pname, GLint* params);
//Beispiel zur Abfrage der Größe des Shared Memory:
GLint size;
glGetIntegerv(GL_MAX_COMPUTE_SHARED_MEMORY_SIZE, &size);
```

Der Zugriff auf geteilte Variablen erfolgt inkohärent, es gibt also keine Garantie, dass Schreib- oder Lesezugriffe eines Aufrufs in der angegebenen Reihenfolge ausgeführt wurden und für andere Aufrufe sichtbar sind, wenn ein nachfolgender Zugriff auf dieselbe Variable stattfindet. Daher müssen die Speicherzugriffe explizit geordnet werden. Dies wird durch Speicherbarrieren (Memory Barriers) erreicht, und GLSL bietet verschiedene Möglichkeiten wie z.B. `memoryBarrierShared()` speziell für geteilte Variablen, `memoryBarrier()` für alle Speicher-Typen, `groupMemoryBarrier()` nur für alle Aufrufe einer Work Group. Speicherzugriffe, die vor einer solchen Barriere vorgenommen wurden, sind für alle Shader Aufrufe (oder

diejenigen in derselben Gruppe im Fall von `groupMemoryBarrier()`) nach der Barriere garantiert sichtbar. Darüber hinaus muss auch die Ausführungsreihenfolge aller Shader Aufrufe innerhalb einer Work Group synchronisiert werden, um sicherzugehen, dass z.B. ein bestimmter Aufruf den Wert einer geteilten Variablen erst liest, nachdem ein anderer Aufruf innerhalb der Work Group diesen geschrieben hat. Dafür kommt `barrier()` zum Einsatz. Die Ausführung innerhalb einer Work Group wird erst fortgesetzt, wenn alle Aufrufe aus der Work Group die Barriere erreicht haben. Eine Synchronisierung kann dazu führen, dass Warps auf andere Warps der gleichen Work Group warten müssen. Das kann die Ausführung ausbremsen, vor allem dann, wenn dem Multiprozessor nicht genügend andere Work Groups zur Verfügung stehen, welche er in der Zeit weiterverarbeiten könnte. Die Ordnung von Speicherzugriffen und die Synchronisierung der Aufrufe ist auch einzusetzen, wenn andere Speicher-Typen verwendet werden, um Daten aus einem Aufruf für einen anderen verfügbar zu machen. Auch wenn Ergebnisse eines Compute Shaders im OpenGL Host Programm weiterverwendet werden sollen, ist in der Regel eine Synchronisierung notwendig, um sicher zu gehen, dass die Ergebnisse komplett geschrieben wurden. Dazu wird `void glMemoryBarrier(GLbitfield barriers)` verwendet, z.B. für `GL_SHADER_STORAGE_BARRIER_BIT`, so dass alle Schreibzugriffe auf Shader Storage Buffer Objekte abgeschlossen sind, bevor mit den Daten weitergearbeitet wird.

Die Compute Shader Ein- und Ausgaben müssen, wie erwähnt, explizit aus einem Speicher gelesen bzw. in einen Speicher geschrieben werden. Dafür stehen mit Shader Image load/store (seit OpenGL 4.2) und Shader Storage Buffer Objekten flexible Möglichkeiten zur Verfügung, um Lese- und Schreibzugriffe an beliebiger Stelle im Bild bzw. Speicherobjekt vornehmen zu können (random-access reads/writes). Bilder (einzelne Schichten aus Textur Objekten) werden per `glBindImageTexture()` an Image Binding Points gebunden. Im Shader können auf dem Bild, anders als auf kompletten Texturen, an beliebiger Stelle Lese- und Schreibzugriffe vorgenommen werden. Wie bei allen Speicherzugriffen ist ggf. auch hier explizit für Synchronisation zu sorgen, sobald mehrere Aufrufe auf dieselbe Position bzw. denselben Platz im Speicher schreibend und lesend zugreifen. Ein Beispiel für die Verwendung eines 2D Bild Objektes im Compute Shader zeigt Listing 4. Weitere Details sind z.B. in (Segal und Akeley 2016) und (Wright et al. 2016) zu finden.

Listing 4: Einfaches Beispiel für die Verwendung von 2D Bild Objekten mit 32Bit Floating Point RGBA Werten.

```
//Das Bild wurde im Host Programm bereits aus einem Textur
//Objekt gebunden. Der Typ der Bild Variablen muss dem des
//zugehörigen Textur Objekts entsprechen,z.B. GL_TEXTURE_2D.
//Ohne einschränkende Qualifizierer wie writeonly sind sowohl
//Lese- als auch Schreibzugriffe auf derselben Bild
```

```
//Variablen möglich.
layout(binding = 0, rgba32f) uniform readonly image2D imgIn;
layout(binding = 1, rgba32f) uniform writeonly image2D imgOut;

layout (local_size_x = 16, local_size_y = 16) in;
void main(void)
{
 ivec2 pixel = ivec2(gl_GlobalInvocationID.xy);
 ivec2 size = imageSize(imgIn);
 //imgOut hat die gleiche Größe
 vec4 color = imageLoad(imgIn, pixel);
 color += //Verarbeitung des Farbwertes
 imageStore(imgOut, pixel, color);
}
```

Noch mehr Flexibilität bringen Shader Storage Buffer Objekte (SSBO), welche beliebige Daten speichern können, auch Strukturen und Arrays von Strukturen (Array of Structures, AOS), und deren Speichergröße in der Regel nur durch den verfügbaren Graphikspeicher beschränkt ist. Ein Buffer Objekt, das im OpenGL Host Programm definiert wurde, kann im Shader durch Shader Storage Blocks verwendet werden.

In Listing 5 wird ein Beispiel der Erstellung eines SSBO für die Nutzung im Shader gezeigt. Als Datenpointer wird in diesem Beispiel `NULL` übergeben, da der Shader die Daten selbst schreiben wird. Wenn Daten an den Shader übergeben werden sollen, kann an der Stelle entsprechend ein Pointer angegeben werden. In `glBufferData()` wird als erwartetes Nutzungsmuster `GL_STATIC_DRAW` eingesetzt, da die Daten in diesem Beispielszenario nicht von der CPU im Rahmen des Host Programms geändert werden (`GL_DYNAMIC_DRAW`), sondern ausschließlich von der GPU. So wird dafür gesorgt, dass die Daten im GPU Speicher gehalten werden (Bailey 2013).

Listing 5: Beispiel für die Definition eines OpenGL Shader Storage Buffer Objekts für die Nutzung in einem Compute Shader.

```
//Struktur für die Daten, die vom Shader in den Speicher
//geschrieben werden sollen
Ray{
 vec4 origin;
 vec4 direction;
};

//Erstellen des Buffer Objekts und Allokation des Speichers.
GLuint ssbo; //Name des Buffers
GLuint bufferSize = sizeof(Ray) * workSizeX * workSizeY;
glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, bufferSize, NULL,
             GL_STATIC_DRAW);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

```
//...
//Vor Ausführung des Shaders wird der Buffer an einen Index
//gebunden.
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, ssbo);
```

Im Compute Shader wird der Buffer als Menge von Buffer Variablen, gruppiert in einem Shader Storage Block, nutzbar gemacht. Dabei kann als Speicher-Layout (Storage Layout Qualifier) neben *std140*, anders als bei Uniform Blocks, auch *std430* zum Einsatz kommen. Letzterer hat den Vorteil, dass Arrays von skalaren Typen und Vektoren (mit 2 oder 4 Elementen) sowie entsprechende Strukturen im Speicher dicht gepackt werden können und erlaubt so eine effizientere Speicherausnutzung (Segal und Akeley 2016, Wright et al. 2016). Uniform Blocks sind zudem deutlich eingeschränkter in Bezug auf die Speichergröße und können nur für Eingabedaten (z.B. Uniform Variablen gruppiert in Uniform Buffer Objects) verwendet werden. Für kleinere Eingabedatenmengen, welche über die Shaderausführung konstant bleiben, eignen sich UBOs und Uniform Blocks, da durch die Einschränkungen potenziell Optimierungen seitens Hardware und Treibern möglich sind, die bei SSBOs und Shader Storage Blocks nicht stattfinden können (Wright et al. 2016 Kapitel 5). Nvidia GPUs verfügen z.B. über 64KB „Constant Memory“, auf dem Lesezugriffe durch Verwendung eines „read-only constant cache“ beschleunigt werden (Nvidia 2015a). Listing 6 zeigt ein einfaches Beispiel für die Nutzung des in Listing 5 erstellten SSBOs. Im Storage Block wird der gleiche Binding Point angegeben wie beim Binden des entsprechenden Buffer Objekts. Als letztes Element kann der Shader Storage Block ein Array mit dynamischer Größe enthalten.

Listing 6: Beispiel für die Definition eines Shader Storage Blocks mit einem Array of Structures

```
#version 450 core
struct Ray{
 vec4 origin;
 vec4 direction;
};

//Definition des Shader Storage Blocks
layout (binding = 1, std430) buffer rayBuffer
{
 Ray rays[];
};
```

In obigem Beispiel wird als Storage Layout *std430* eingesetzt, für Strukturen wird die Ausrichtung (Alignment) im Speicher somit anhand des größten Elements der Struktur festgelegt. Hier wären das $4 \cdot 4 = 16$ Byte für `vec4`. Wäre zusätzlich eine Floating Point Variable in der Struktur, würde auch diese an 16 Byte anstatt 4 Byte ausgerichtet werden, so dass eine andere

Definition der Struktur (z.B. indem auch die Vektoren als einzelne Floating Point Variablen definiert werden) ggf. günstiger ist. Ein manuelles Auffüllen (Padding) mit zusätzlichen Variablen verhindert Probleme beim Speicherzugriff.

Shader Storage Blöcke, Bild Objekte und geteilte Variablen erlauben atomare Operationen (Atomic Operations), ununterbrochene „lese-modifiziere-schreibe“ Sequenzen auf einem Speicherplatz durch einen Aufruf. Versuchen mehrere Aufrufe gleichzeitig eine atomare Operation auf demselben Speicherplatz auszuführen, werden sie serialisiert. Die atomaren Funktionen sind für Integer und vorzeichenlose Integer definiert. Die Funktionen erhalten zwei Parameter `inout int mem` und `int data` (beide entweder vom Typ `int` oder `uint`) und haben den entsprechenden Rückgabetyp. Zum Beispiel liest `atomicAdd()` das Datum an Speicherposition `mem`, addiert es zu `data`, schreibt das Ergebnis zurück in den Speicher und gibt den ursprünglichen Wert zurück. Die nachfolgenden Aufrufe können das entsprechende Datum erst aus dem Speicher lesen, wenn die Operation abgeschlossen ist. Nach dem gleichen Prinzip funktionieren auch die anderen verfügbaren atomaren Operationen, einige Beispiele sind `atomicAnd()`, um die bitweise Verundung vom Wert bei `mem` und `data` in den Speicher zu schreiben, `atomicMin()`, um das Minimum beider Werte in den Speicher zurückzuschreiben, `atomicExchange()`, um die Werte auszutauschen.

Eine spezielle GLSL Variable, Atomic Counter, deren Wert vom Typ `uint` ist, erlaubt nur drei atomare Operationen: Lesen, Inkrementieren und Dekrementieren. Atomic Counter basieren auf einem Buffer Objekt, das in OpenGL definiert und an das `GL_ATOMIC_COUNTER_BUFFER` Ziel gebunden werden muss, damit eine Verwendung im Shader möglich ist. Der Speicherbereich wird zwischen Aufrufen geteilt, so dass sich Atomic Counter auch als Hilfsmittel zur Kommunikation zwischen Aufrufen eignen sowie zur Auswertung von Algorithmen, zum Debugging und für andere Aufgaben, die durch ein Aufruf-übergreifendes Zählen erleichtert werden. Listing 7 zeigt die Erstellung des Buffer Objekts in OpenGL, Listing 8 die Deklaration und Verwendung im Shader.

Listing 7: Beispiel Atomic Counter Buffer

```
GLuint atomicBuffer;
glGenBuffers(1, &atomicBuffer);
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, atomicBuffer);
//durch 2 * sizeof(GLuint) ist im Buffer Platz für 2 Counter
glBufferData(GL_ATOMIC_COUNTER_BUFFER, 2 * sizeof(GLuint),
             NULL, GL_DYNAMIC_DRAW);
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, 0);
// ...
//Binden and bestimmten Index
glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, atomicBuffer);
```

```
// Verwendung im Shader ...

//Zurücksetzen der Counter
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, atomicBuffer);
GLuint resetData[2] = {0, 0};
glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 0 ,
                2 * sizeof(GLuint), resetData);
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, 0);

//...
//Auslesen der Counter in OpenGL
GLuint aCounters[2];
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, atomicBuffer);
glGetBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 0,
                   2 * sizeof(GLuint), aCounters);
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, 0);
countedValueOne = aCounters[0];
countedValueTwo = aCounters[1];
```

Listing 8: Beispiel Atomic Counter im Compute Shader

```
//... Deklaration im Shader
layout (binding = 0, offset=0) uniform atomic_uint a_ctr_one;
layout (binding = 0, offset=4) uniform atomic_uint a_ctr_two;

// ... und Verwendung; Rückgabe ist uint
atomicCounterIncrement(a_ctr_one);
//atomicCounterDecrement(a_ctr_one);
//atomicCounter(a_ctr_one);
```

Als Quellen für weitere Informationen zu Compute Shadern und verwandte Bereiche sei (Wright et al. 2016) genannt sowie die OpenGL Spezifikation (Segal und Akeley 2016).

3 Verwandte Arbeiten

3.1 GPU Ray Tracer Struktur

Der erste vollständig auf der GPU implementierte Ray Tracer wurde von Purcell et al. vorgestellt (Purcell et al. 2002) und unter Verwendung der zu diesem Zeitpunkt noch sehr eingeschränkten Möglichkeiten programmierbarer Fragment Shader realisiert. Die Autoren schlagen bereits 2002 vor, die GPU als Streaming Prozessor zu verstehen, da das Streaming-Modell die Ausnutzung der Parallelität, effizienten Einsatz der Bandbreite und Verstecken von Speicherlatenz erlaubt. Ihren Ray Tracer strukturieren sie für die Abbildung auf das Streaming Modell in mehrere (Fragment) Shader Programme bzw. Kernel (Abb. 7). Die Verbindung zwischen den Programmen ist der „Stream“ aus Ein- und Ausgabedaten.

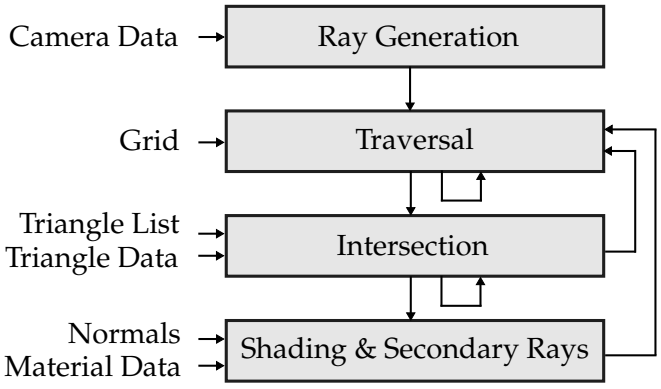

Abbildung 7: Aufteilung und Datenfluss des Streaming Ray Tracers nach Purcell et al. (Purcell et al. 2002).

Die Eingabedaten werden bei Purcell et al. in Form von Texturen übergeben, die an der Bildgröße ausgerichtet sind und Ausgaben werden zurück in Texturen geschrieben. Ausführung der Fragment Programme wird durch Rendern eines Rechtecks in Bildgröße erreicht. Damit die jeweiligen Kernel nur für aktive Strahlen bezogen auf einen Status (*traversing*, *intersecting*, *shading*, *done*) ausgeführt werden, kommt der Stencil Buffer zum Einsatz. Als Beschleunigungsstruktur verwenden Purcell et al. ein Uniform Grid, da dies besonders einfach auf der GPU zu implementieren ist. Das Grid wird als 3D Texture Map repräsentiert. Die sekundären Strahlen werden vom Shading Kernel (Beleuchtung) mit ihrer jeweiligen Gewichtung in Texturen geschrieben, um wieder an den Traversierungs-Kernel weitergereicht zu werden. Da noch kein Branching auf der GPU möglich war, wird der Prozess in zahlreichen Durchläufen (Passes) als Multipass Verfahren ausgeführt. Die Autoren entwickeln zum Vergleich einen Simulator für die Multipass-Architektur und eine Branching-Architektur, welche alle Kernel in einem Durchlauf ausführen kann. Die Auswertung für die Multipass- und die Branching-Architektur im Simulator zeigt, dass die Branching-Architektur weniger Instruktionen und deutlich weniger Bandbreite benötigt. Während die Multipass-Architektur die GPU Leistung (zu dem Zeitpunkt Nvidia GeForce3) bezüglich Bandbreite und Rechenleistung ausnutzt, nutzt die Branching-Architektur kaum die Bandbreite und ist somit durch die Rechenleistung limitiert. Die Leistung der frühen GPU Ray Tracer wird im Jahr 2003 als vergleichbar mit den zu dem Zeitpunkt schnellsten Software Ray Tracing Systemen bezeichnet und erreicht die Leistung von Einzel-CPU Ray Tracing Systemen der Zeit (Wald, Purcell et al. 2003).

Neben dem Ziel, den Ray Tracer durch Strukturierung auf das Streaming-Modell abzubilden, war eine Unterteilung des Ray Tracers auf mehrere Kernel bei Purcell et al. letztendlich auch durch die hardwareseitigen Einschrän-

kungen notwendig (maximale Anzahl an Instruktionen, Registerplätzen und an Multiple Render Targets).

Eine konzeptionell ähnliche Aufteilung hinsichtlich der Datenparallelität wird deutlich später von Garanzha et al. vorgenommen, um einen GPU Ray Tracer (implementiert in CUDA) auf die parallele GPU Architektur abzubilden (Garanzha und Loop 2010). Die resultierende Ray Tracing Pipeline besteht aus sechs Stufen, die in jeweils einem Kernel implementiert sind. Der Trace-Vorgang nimmt davon vier Stufen (und damit auch vier Kernel bei der Implementierung) ein, davon sind zwei spezifisch für die vorgestellte Traversierungsmethode von Strahlenpaketen (Weiteres dazu in Abschnitt 3.3). Die gewählte Aufteilung soll die Ausnutzung der Datenparallelität ermöglichen und Warp internes divergentes (mehrfach-) Branching minimieren. Diese vier Stufen umfassen die Sortierung der Strahlen in kohärente Pakete, die Erstellung von Frusta für die Pakete, die breadth-first Frustum Traversierung einer BVH und die Strahl-Primitiv-Schnitttests. Die gesamte Ray Tracing Pipeline nach Garanzha et al. ist in Abbildung 8 dargestellt. Wenn man die beiden spezifischen Stufen für die Strahlenpakete (in Abb. 8 dunkler dargestellt) weglässt, ergibt sich ein sehr ähnliches Konzept wie bei der Struktur von (Purcell et al. 2002), mit dem Unterschied, dass ein Kernel für die Generierung primärer und sekundärer Strahlen zuständig ist.
Parker et al. stellen 2010 Nvidia OptiX, eine Ray Tracing Engine bzw. Ray Tracing API für GPUs und andere parallele Architekturen vor, welche mit einem domänenspezifischen just-in-time Compiler aus benutzerdefinierten Programmen und einem internen OptiX Kernel einen Ray Tracing Kernel erstellt (Parker et al. 2010). Die benutzerdefinierten Programme basieren auf sieben vorgegebenen „Programmtypen“, deren exakte Funktion je nach Ray Tracing basierter Anwendung durch den Benutzer zu implementieren sind und zusammen mit dem OptiX Kernel eine programmierbare Ray Tracing Pipeline ergeben. Diese sieben Programmtypen umfassen die Strahlen-Generierung, Schnitttest Programme, Closest Hit Programme z.B. für Shading Berechnungen, nachdem der nächste Schnittpunkt gefunden wurde und ggf. für das Versenden weiterer Strahlen, Any Hit Programme für die Behandlung von Strahlen bei Schnittpunkten mit einem Objekt, z.B. Terminieren von Schattenstrahlen, Programme für die Behandlung von Strahlen, die keine Geometrie treffen („Miss“), Exception Handling und „Selector Visit“ Programme, die z.B. für die Änderung des Detailgrades von Bereichen der Szene je nach Distanz für einzelne Strahlen eingesetzt werden können. Zusätzlich kommt ein Programm für die Berechnung von Bounding Boxes zum Einsatz, um Beschleunigungsstrukturen zu ermöglichen. Die Flexibilität von OptiX ergibt sich aus der Programmierbarkeit dieser Programme, ähnlich den programmierbaren Shader Programmen der Graphik Pipeline. Während das OptiX Programmiermodell für den Benutzer obige Aufteilung präsentiert, verbindet der just-in-time Compiler letztendlich alle

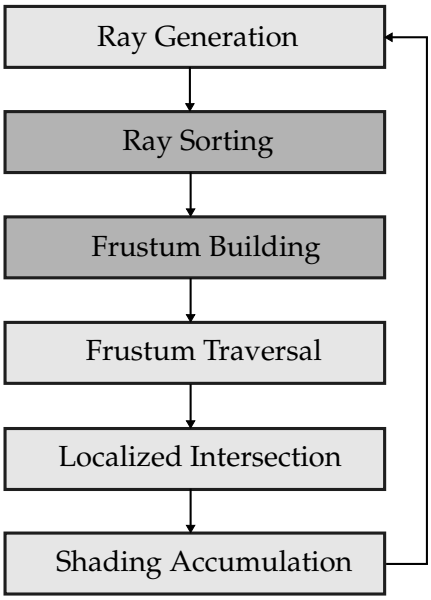

Abbildung 8: Aufteilung der Ray Tracing Pipeline nach Garanzha et al. (Garanzha und Loop 2010): Strahlen-Erstellung, Strahlen-Sortierung, Frustum-Erstellung, Traversierung der BVH, Primitiv-Schnitttests, Beleuchtung akkumulieren

Einzelteile mit dem OptiX Code zu einem „Megakernel“. Die Aufteilung wird für die Ausführung also nicht beibehalten. Die Autoren begründen dieses Ausführungsmodell mit einem geringen Overhead bezüglich der Kernel Ausführungen, erwähnen jedoch auch die ggf. geringere Prozessorauslastung aufgrund hoher Registernutzung. Dies kann die Hardware Multithreading Kapazität für das Verstecken von Latenzen einschränken. Durch die verfolgte Flexibilität und die Abstraktionsschichten der OptiX API ist ein Ray Tracer auf der Basis von OptiX im Vergleich zu einem spezifischen und entsprechend von Hand optimierten Ray Tracer wie von (Aila und Laine 2009) deutlich langsamer (Parker et al. 2010).

Die Nachteile eines einzelnen Megakernels gegenüber einer geeigneten Strukturierung in mehrere Kernel adressieren Laine et al. im Rahmen ihres „Wavefront Path Tracers“ (Laine et al. 2013). Sie unterteilen den Path Tracer in mehrere spezialisierte Kernel, um die Kontrollfluss-Divergenz zu reduzieren, so die Anzahl der tatsächlich aktiven SIMT Threads zu erhöhen und zudem die Verwendung von Ressourcen gering zu halten, damit genügend Threads für das Verstecken von Latenzen auf der GPU verfügbar sind. Die größten Probleme bei einem Megakernel sind:

- Mehr Kontrollfluss-Divergenz innerhalb eines Warps: Inaktive Threads,

aufgrund der Terminierung von Strahlen bzw. Pfaden zu verschiedenen Zeitpunkten oder durch divergierende Ausführungszweige, die seriell durchlaufen werden müssen.

- Die hohe Registernutzung einzelner Bereiche beeinträchtigt in einem Megakernel für das gesamte Programm die Fähigkeit der Hardware, Latenzen von Speicherzugriffen zu verstecken, da Hotspots in der Ressourcennutzung dazu führen, dass insgesamt weniger Threads auf den Multiprozessoren der GPU gehalten werden können.

- Für sehr große Kernel mit hoher Divergenz zwischen Warps kann der Instruction Cache überlaufen.

Ein Nachteil bei einer Aufteilung auf mehrere Kernel ist die Notwendigkeit, den Status der Pfade im globalen Speicher abzulegen anstatt den Status über die gesamte Ausführung hinweg in Registern zu halten. Dies sei jedoch, laut Laine et al., durch ein geeignetes Layout im Speicher zu kompensieren. Dafür werden die Daten in einer Struktur von Arrays (Structure Of Arrays, SOA) organisiert, da der Logik Kernel (siehe unten) den Status für alle Pfade der Reihe nach ausliest (die anderen Kernel nicht), finden so pro Warp 32 Lese- bzw. Schreibzugriffe auf 32 Bit Status Variablen an aufeinanderfolgenden Speicherpositionen statt. Gegenüber Arrays aus Strukturen geben die Autoren eine große Beschleunigung durch das SOA Speicherlayout an.

Die Aufteilung des Path Tracers geht von der Verwendung komplexer, mehrschichtiger Materialien aus, deren Auswertung einen hohen Anteil der Berechungskosten ausmacht. Die einzelnen Kernel sind in drei Stufen eingeordnet: Die „*logic stage*“ besteht aus einem Kernel für die gemeinsame Logik, welche für alle Pfade ausgeführt wird, die „*material stage*“ umfasst n Kernel für die Auswertung von (komplexen) Materialien und einen Kernel für die Generierung neuer Pfade und die „*ray cast stage*“ mit jeweils einem Kernel für Ray Casting und für Ray Casting von Schattenstrahlen. Jeder Kernel (abgesehen vom Logik Kernel) hat eine eigene Queue mit fester Größe, in der die vorherige Stufe die auszuführende Arbeit für den entsprechenden Kernel der nächsten Stufe ablegt. Die Kommunikation zwischen den Stufen geschieht über den Status der Pfade und die Queues, beides wird im globalen Speicher gehalten. Beispielsweise gibt der Logik Kernel für terminierte Pfade die Aufgabe zur Erstellung eines neuen Pfades an die Queue für die Pfadgenerierung, dessen Kernel wiederum eine Aufgabe zur Strahlenverfolgung an den Ray Casting Kernel gibt. Die Ergebnisse der Ray Tracing Kernel werden durch entsprechende Indizes in Ergebnis-Buffer geschrieben und die Indizes werden zusätzlich beim Status des Pfades gespeichert.

Die Arbeitsverteilung basiert auf einem Pool von einer Million aktiven Pfaden. Ein terminierter Pfad wird innerhalb derselben Iteration durch einen neuen Pfad ersetzt. In jeder Iteration werden alle Pfade um ein Segment fortgesetzt. Der Status eines Pfades wird in 212 Bytes im DRAM gespeichert

und umfasst auch die Strahlen und Ergebnisse der Strahlenverfolgung zu dem Pfad. Für den Ray Tracing Prozess werden die Kernel von (Aila und Laine 2009, Aila, Laine und Karras 2012, siehe Abschnitt 3.3) verwendet und für den BVH-Aufbau das SBVH (siehe Abschnitt 3.2) Verfahren aus (Stich et al. 2009).

Jedes komplexe Material wird in einem eigenen Kernel ausgewertet, und es findet eine Verdichtung der auszuführenden Arbeit bezogen auf den Kernel statt. Bei (Wald 2011) (beschrieben in Abschnitt 3.3) wird eine Verdichtung von mehreren, nicht voll aktiven Warps zu weniger, voll besetzten Warps mit zunehmend weniger kohärenten Strahlen durchgeführt, bei Laine et al. hingegen geht es um eine Verdichtung der Arbeit, geordnet nach der jeweiligen Aufgabe, insbesondere der Materialauswertung eines bestimmten komplexen Materials. Dazu werden die Queues eingesetzt, in denen die „Anfragen" für die auszuführende Arbeit in Form von Indizes der entsprechenden Pfade aufgenommen werden, um sie in kohärenter Ordnung zu verarbeiten. Für die Zählung der Elemente werden Atomic Counter verwendet. Damit die kohärente Ordnung der zu verarbeitenden Elemente sichergestellt und die Inkrementierung effizienter durchgeführt werden kann, wird nicht sofort in die Queues geschrieben, sondern die atomaren Operationen werden zunächst Warp intern aggregiert.

Die Ergebnisse zeigen anhand eines Vergleichs zwischen der Leistung des „Wavefront" Path Tracers und einem auf dem gleichen Code basierenden Megakernels, dass eine Unterteilung für die Ausführung auf GPUs einen großen Leistungsschub bringen kann. Der Path Tracer, der in mehrere spezialisierte Kernel unterteilt ist, läuft je nach Szene zwischen 36% und 221% schneller als die Megakernel Variante. Somit wird der zusätzliche Aufwand für die unterteilte Variante (Speichern und Auslesen der Statusdaten von Pfaden, Ausführen der Kernel und Verwaltung der Queues) amortisiert. Hier zeigt sich also deutlich ein positiver Effekt durch die Aufteilung auf mehrere Kernel.

3.2 Bounding Volume Hierarchie Konstruktion

Wald et al. wenden die Surface Area Heuristik, welche im Kontext von Bounding Volume Hierarchien entwickelt (siehe Abschnitt 2.3.1), dann auch für Konstruktionsverfahren von kd-Trees adaptiert und verwendet wurde, wiederum für den Aufbau von BVHs an. Dabei gehen sie auf ähnliche Weise vor wie beim SAH-basierten Aufbau von kd-Trees. Sie wählen aus einer großen Kandidatenmenge von Unterteilungsebenen (basierend auf den Schwerpunkten der Dreiecke), per SAH basierter Kostenfunktion das lokale Optimum (Wald, Boulos et al. 2007). Zur Bestimmung des lokal optimalen Kandidaten aus dieser Menge werden zwei Durchläufe („Sweeps") durch die Menge der Dreiecke ausgeführt und die Kostenfunktion für jeden Kandidaten ausgewertet. Im Folgenden wird ein darauf basierendes

und daher ähnliches, aber beschleunigtes Verfahren von Wald detailliert beschrieben. Dieses Verfahren wurde aufgrund der hohen Qualität bzgl. SAH bei zugleich schnellem Aufbau auch im Rahmen der vorliegenden Arbeit implementiert.

Wald verwendet für den schnelleren Aufbau SAH-basierter BVHs (auf der CPU) eine Binning Technik für die Auswahl der Unterteilungsebene, die sich zu der Zeit (2007) für kd-Trees bereits bewährt hat. Er zeigt, dass diese Technik für BVHs sogar besser geeignet und deutlich schneller ist als für kd-Bäume (Wald 2007). Dies führt er vor allem darauf zurück, dass BVHs weniger Knoten haben als vergleichbare kd-Trees, Primitive, die eine Unterteilungsebene (split plane) überschneiden nicht gesondert behandelt werden müssen, da sich ein Primitiv (bzgl. seines Mittelpunkts) im BVH auf genau einer Seite befindet und damit einhergehend die Anzahl an Knoten nach oben begrenzt ist, durch maximal $2N - 1$ mit N Primitiven. Somit ist keine Verwaltung zusätzlicher Speicherressourcen für Knoten notwendig. Darüber hinaus hat die Annäherung des lokalen Optimums durch Binning für BVHs weniger Auswirkungen auf die Qualität der entstehenden Hierarchie als für kd-Trees. Die Ausbreitung der entsprechenden BVs wird in allen drei Dimensionen exakt anhand der BVs der Primitive (iterativ) berechnet und nicht anhand des eindimensionalen Bereichs eines Bins bestimmt, und die kleinste mögliche Unterteilung wird durch die BVs der Primitive begrenzt (bei kd-Trees werden Primitive ggf. an den Begrenzungen eines Knotens weiter unterteilt, die Wahl unter den Kandidaten für eine solche Unterteilung, „perfect splits“, hat Auswirkungen auf die Qualität).

Basierend auf der SAH können die Traversierungs-Kosten für einen Teilbaum BV (bzw. ein Volumen, das einen Teilbaum repräsentiert) mit N Dreiecken, bezogen auf eine bestimmte Unterteilung in zwei Kindknoten BV_L mit N_L Dreiecken und BV_R mit N_R Dreiecken, abgeschätzt werden als:

$$Cost_{part}(BV \rightarrow BV_L, BV_R) = C_T + C_I \cdot (\frac{S(BV_L)}{S(BV)} \cdot N_L + \frac{S(BV_R)}{S(BV)} \cdot N_R) \quad (16)$$

Dabei sind C_T und C_I Konstanten, für die abgeschätzten Kosten der Traversierung eines Knoten und für einen Schnitttest mit einem Primitiv. Die Unterteilung, welche BV_L und BV_R ergibt, so dass die Kostenfunktion minimal ist, wird gesucht. Für BV gäbe es $2^N - 2$ Möglichkeiten, eine Unterteilung vorzunehmen. Alle auszuwerten wäre zu aufwendig, daher wird, wie bei kd-Trees üblich, eine Menge von Unterteilungsebenen betrachtet. Festzustellen auf welcher Seite einer Unterteilungsebene ein Primitiv liegt, ist für BVHs einfacher, da Primitive nicht aufgeteilt werden und es somit ausreicht, anhand des Schwerpunktes des Primitiv-BVs zu entscheiden. Die Unterteilungsebene hat bei BVHs aber keinen direkten Bezug zur Ausdehnung der entstehenden Volumen, welche die Ebene je nach der tatsächlichen Ausdehnung der enthaltenen Primitive überschneiden oder nicht berühren. Anders als zuvor in (Wald, Boulos et al. 2007) wird eine deutlich

kleinere Kandidatenmenge von Unterteilungsebenen, hier in Form von Bins, verwendet. Wald beschreibt, dass bereits eine Anzahl von 16 Bins (15 Unterteilungsebenen) ausreicht, um nahezu optimale Ergebnisse zu erreichen. Eine höhere Anzahl habe die Qualität kaum oder nicht erhöht. Gleiches gilt für die Anpassung der Anzahl je nach Menge der Dreiecke. Die Genauigkeit der Terme für die Kostenfunktion hängt nicht von der Anzahl der Bins ab.

Als Bounding Volumes verwendet Wald AABBs, und die BVH hat ein binäres Verzweigungsverhältnis. Die Eingabedaten für den „Binned BVH" Aufbau umfassen das BV der gesamten Szene bv_{scene}, die BVs der Dreiecke tbv_i und deren Schwerpunkte c_i und zusätzlich ein BV für die Ausdehnung aller Schwerpunkte innerhalb eines BV cb („Centroid Bounds"). Die Wahl der Unterteilungsachse k („split axis") wird entsprechend der Achse gewählt, auf welcher cb die größte Ausdehnung hat. Alternativ können alle drei Achsen betrachtet werden. Das Volumen der Schwerpunkte cb wird entlang der Unterteilungsachse in K Bins, $B_1, ..., B_K$ von gleicher Breite gleichmäßig unterteilt. Die Bins repräsentieren $K-1$ Unterteilungsebenen für mögliche Unterteilungen der Menge der Dreiecke T in zwei Teilmengen $TLj = B_1, ..., B_j$ und $T_{Rj} = B_{j+1}, ..., B_K$ für $j \in k, ..., K-1$. Für jedes Bin B_i wird beim Binning-Prozess die Anzahl an Dreiecken n_i gezählt, so dass für jede mögliche Unterteilung die Anzahl von Dreiecken auf der linken Seite $N_{L,j}$ und auf der rechten Seite $N_{R,j}$ ermittelt werden kann durch

$$N_{L,j} = \sum_{i=1}^{j} n_i \text{ und } N_{R,j} = \sum_{i=j+1}^{K} n_i. \tag{17}$$

Die Kostenfunktion in Gleichung 16 kann vereinfacht werden, da die Kosten verschiedener Unterteilungen ausschließlich relativ zueinander zu betrachten sind. Durch Entfernen von Konstanten und gemeinsamen Termen ergibt sich Gleichung 18 für die Kosten einer Unterteilung j, wobei $A_{L,j}$ und $A_{R,j}$ jeweils die Größe der Oberfläche von $bv_{L,j}$ bzw. $bv_{R,j}$ der entsprechenden Seite der Unterteilung ist.

$$Cost_j = A_{L,j} \cdot N_{L,j} + A_{R,j} \cdot N_{R,j} \tag{18}$$

Damit die Oberflächengrößen für beide Seiten berechnet werden können, benötigt man die tatsächliche Ausdehnung aller Primitive, die einem Bin zugeordnet werden. Während dem Binning-Prozess wird für jedes Bin daher iterativ das BV bb_i (Bin Bounds) mitberechnet. Während Bins als eine eindimensionale Projektion auf die Unterteilungsachse nur eine Annäherung darstellen, erhält man so letztendlich dennoch die exakten dreidimensionalen BVs für die Unterteilung und dessen Oberflächengröße für die Kostenfunktion. Die Größen der Oberflächen sind dann zu berechnen durch

$$A_{L,j} = S(\bigcup_{i=1}^{j} bb_i) \text{ und } A_{R,j} = S(\bigcup_{i=j+1}^{K} bb_i). \tag{19}$$

Der Binning-Prozess entspricht der Bestimmung einer Bin ID für jedes Primitiv, ggf. gefolgt von der Aktualisierung von n_i und bb_i. Die Bin ID für einen gegebenen Primitiv-Schwerpunkt c_i wird berechnet durch

$$binID_i = \frac{K \cdot (1-\epsilon) \cdot (c_{i,k} - cb_{min,k})}{cb_{max,k} - cb_{min,k}}, \tag{20}$$

wobei k die Unterteilungsachse ist und der Faktor $(1-\epsilon)$ dazu dient, dass ein Primitiv auch dann in ein Bin eingeordnet wird, wenn sein Schwerpunkt genau auf die Grenze von cb fällt. Alle Elemente außer c_i sind konstant und können somit vorberechnet werden. So ergibt sich für die Berechnung von $binID_i$ Gleichung 21, gefolgt von der Rundung auf einen Integer Wert.

$$binID_i = a \cdot (c_{i,k} - cb_{min,k}) \text{ mit } a = \frac{K(1-\epsilon)}{cb_{max,k} - cb_{min,k}}. \tag{21}$$

Die Auswahl der (lokal) besten Unterteilung geschieht in zwei linearen Durchläufen der Bins. Von links, um $N_{L,i} = N_{L,i-1} + n_i$ sowie $A_{L,i}$ zu bestimmen und von rechts, um entsprechend $N_{R,i}$ und $A_{R,i}$ zu bestimmen und die Kostenfunktion für jede Unterteilungsebene auszuwerten. Die Durchführung der Unterteilung verläuft auf einem Array mit den IDs der Dreiecke. Die Sequenz des Arrays, die zum gerade unterteilen Knoten gehört, wird entsprechend der gewählten Unterteilung per Quicksort in-place sortiert. Dabei wird die ID der Unterteilung als Pivot eingesetzt. Ein Sweep von links über das Intervall der Dreiecksliste bestimmt für jedes Dreieck die Bin ID bis ein Dreieck erreicht wird, das zur rechten Seite gehört. Analog wird ein Sweep von rechts ausgeführt, bis der Iterator eine ID findet, die zur linken Seite gehört. Die IDs werden ausgetauscht, und der Prozess wird wiederholt, bis sich beide Iteratoren treffen. Währenddessen werden die BVs für beide Seiten und für die Schwerpunkte akkumuliert, und die neuen BVH Knoten werden in einem Knoten-Array gespeichert. Der Aufbau-Prozess wird für die neuen Knoten fortgeführt, bis am Ende alle Verweise auf Dreiecke in einem Blattknoten (bzw. einem Teilbaum) in entsprechender Ordnung kontinuierlich im Speicher liegen.

Das Verfahren terminiert die Unterteilung und erstellt einen Blattknoten, wenn eine definierte minimale Anzahl an Dreiecken (hier 2 oder 4) erreicht wird, oder die Ausdehnung der Schwerpunkte (Centroid Bounds) für den Knoten zu klein wird, oder wenn die geschätzten Kosten für eine Unterteilung höher sind als für einen Blattknoten.

Wald beschreibt zudem verschiedene Möglichkeiten bezüglich der Parallelisierung des Verfahrens für Mehrkern CPUs. Die Ergebnisse im Vergleich zum vollständigen SAH Sweep in (Wald, Boulos et al. 2007) zeigen bereits für die serielle Version einen deutlich schnelleren Aufbau durch das Binning Verfahren bei hoher Qualität von 91% bis 100% relativ zum SAH Sweep Verfahren.

Parallel zu (Wald 2007) wurde ein sehr ähnliches Verfahren in (Günther et al. 2007) vorgestellt. Günther et al. verwenden, anders als Wald, eine adaptive Anzahl $k = n/r$ an Bins pro Dimension, abhängig von der Anzahl an Primitiven n und Bin-Verhältnis r, und begrenzt durch das Intervall $[k_{min}, k_{max}]$. Als Standard werden angegeben: $k_{max} = 128, k_{min} = 8, r = 6$; und für schnellen Aufbau: $k_{max} = 32, k_{min} = 4, r = 16$. Die erreichte Qualität der BVHs von Günther et al. wird, relativ zur vollen SAH Auswertung, je nach Szene mit über 100% angegeben. Die Autoren sehen dadurch bestätigt, dass es sich bei der verwendeten SAH Funktion um eine lokale, Greedy-Optimierung handelt, welche nicht das globale Optimum liefern kann. Die Ray Tracing Performanz auf der GPU ist laut den Autoren vergleichbar bis besser als der zu dem Zeitpunkt schnellste kd-Tree (mit „Ropes“ und Traversierung ohne Stack) GPU Ray Tracer. Begründet wird dies durch die höhere GPU Ausnutzung aufgrund niedrigerer Registernutzung und einfacherem Traversierungs-Algorithmus.

Stich et al. wenden für den BVH Aufbau ein weiteres Prinzip an, dass von kd-Bäumen bekannt ist: Die räumliche Aufteilung von Primitiven („spatial splitting“) (Stich et al. 2009). Dadurch wird die Einschränkung, dass ein Primitiv genau einem BV zugeordnet ist, aufgehoben und die Überschneidung von BVs stark reduziert. Dies führt zu höherer Qualität der Hierarchie, gemessen an der Ray Tracing Performanz, insbesondere da diejenigen Fälle reduziert werden, in denen ein Strahl beide BVs trifft und somit beide Teilbäume traversieren muss. Gleichzeitig ist der Aufbauprozess durch die erweiterten Unterteilungsmöglichkeiten aufwändiger. Der Fokus von Stich et al. liegt auf möglichst hoher Qualität der resultierenden Hierarchie und nicht auf der Effizienz des Aufbaus. Die Auswahl der Unterteilung wird per SAH gesteuert: Ein Knoten wird entweder anhand der Objekte oder räumlich unterteilt, je nachdem, welche Unterteilung die geringeren SAH Kosten aufweist. Für die Beschreibung des Verfahrens, genannt SBVH („Split bounding Volume Hierarchy“), werden AABBs verwendet und binäre Bäume aufgebaut. Anders als in vorhergehenden Verfahren, welche die Aufteilung von Primitiven für BVHs angewendet haben (siehe Stich et al. 2009), wird das „Splitting“ der Primitive während des Aufbauprozesses durchgeführt und beachtet die gesamte Menge der Primitive eines Knotens.

Der Aufbauprozess arbeitet mit Referenzen zu Dreiecken (bzw. Primitiven), z.B. in Form von Indizes zu den Dreiecken und deren BVs. Eine Unterteilung eines Dreiecks führt zur Duplizierung der entsprechenden Referenz sowie der Aktualisierung der beiden BVs. Eine SBVH wird basierend auf diesen Referenzen gebildet. Die Flexibilität, die dieses Vorgehen für die Wahl der Unterteilung eines Knotens mit sich bringt, führt gleichzeitig dazu, dass der Prozess für das Finden einer guten Unterteilung komplexer wird. Dieser Prozess ist aufgeteilt in das Finden eines Objekt-Unterteilungskandidaten und das Finden eines räumlichen Unterteilungskandidaten, gefolgt von der

Wahl des Kandidaten mit den geringsten SAH Kosten. Für die Suche nach einem Objekt-Unterteilungskandidaten geben Stich et al. an, dass sie eine vollständige SAH Suche anwenden, was einer SAH Auswertung aller Möglichkeiten für eine Objektunterteilung auf allen drei Achsen bezüglich der Objekt-Schwerpunkte entsprechen dürfte. Die Suche nach einem räumlichen Unterteilungskandidaten ähnelt dem Unterteilungsprozess bei kd-Bäumen und wird als „Chopped Binning" bezeichnet.

Der Binning-Prozess zur Kandidatensuche für BVHs verläuft ähnlich wie oben im Rahmen der Beschreibung zu (Wald 2007) ausgeführt, mit dem Unterschied, dass die Primitiv-Referenzen nicht anhand ihres Schwerpunktes nur einem einzigen Bin zugeordnet werden. Stattdessen wird geprüft, welche Bins vom BV des Primitivs geschnitten werden und das Primitiv wird ggf. an den Bin-Grenzen zerteilt, so dass sich BVs für die Teile des Primitivs ergeben. Die BVs der Teile werden den entsprechenden Bins zugeordnet und die BVs der Bins angepasst. Zusätzlich werden pro Bin zwei Zähler für die Anzahl der beginnenden und der endenden Primitiv-Referenzen in einem Bin verwendet. Stich et al. bezeichnen dies als Anzahl der „*entries*" und „*exits*" von Referenzen, was etwas missverständlich ist und dazu verleiten könnte, für jedes Bin zu zählen ob die linke und rechte Bin-Grenze von der Referenz überschritten wird, also das Primitiv das Bin betritt oder/und verlässt. Gezählt wird aber nur für die Bins, welche den Anfang und das Ende der Referenz enthalten, (durch einen Anfangs- und einen End-Zähler), nach folgendem Beispiel: Eine Referenz, die komplett innerhalb eines Bins liegt, erhöht beide Zähler des Bins, während eine zerschnittene Referenz nur den entsprechenden Zähler der beiden Bins erhöht, in denen das Anfangsteil bzw. das Endteil der Referenz liegt. Für Bins, welche ein inneres Teil einer Referenz enthalten, wird keiner der beiden Zähler inkrementiert. Nach dem Binning erfolgt die Auswertung der SAH Kosten für alle Unterteilungsebenen an den Bin-Grenzen. Dafür werden die Kind-BVs durch die Vereinigung der BVs aller Bins links von der Unterteilungsebene bzw. rechts davon ermittelt. Für die Anzahl der Referenzen im linken und rechten Knoten wird die Summe der Anfangs-Zähler links der Unterteilung und die Summe der End-Zähler rechts von der Unterteilung berechnet. Die Unterteilungsebene mit den geringsten SAH Kosten wird mit der besten gefundenen Objektunterteilung verglichen. Stich et al. verwenden 256 Bins für alle Hierarchie Ebenen (zum Vergleich: Wald verwendet für den Binned BVH Aufbau 16 Bins).

Damit die SAH Kosten der Hierarchie weiter verbessert werden, lassen die Autoren kleine BV Überschneidungen auch bei räumlichen Aufteilung zu, so das eine hybride Form der Unterteilung entsteht. Dafür wird die räumliche Aufteilung zunächst wie oben beschrieben berechnet, dann wird für jede zerteilte Referenz geprüft, ob die SAH Kosten sinken, wenn die Referenz wieder zusammengesetzt und nur in einer der Kind-BVs gespeichert wird. Das „Unsplitting" führt laut den Autoren in fast allen Fällen zu einer

leichten Verbesserung der gesamten SAH Kosten der Hierarchie.

Die duplizierten Primitiv-Referenzen führen zu einem höheren Speicherbedarf im Vergleich zu regulären BVHs und schmälert somit den Vorteil bezüglich flacherer Hierarchien und geringerem Speicherbedarf gegenüber kd-Trees. Stich et al. begegnen diesem Problem, indem sie die räumlichen Unterteilungen auf diejenigen Fälle einschränken, in denen sonst große Überschneidungen der BVs entstehen würden. Als Kriterium wird die Oberflächengröße des Schnitt-Volumens der Kindknoten aus der Objektunterteilung relativ zur Oberflächengröße des Wurzelknotens mit einem benutzerdefinierten Schwellenwert ($\alpha \in [0, 1]$) verglichen. Eine Berechnung des räumlichen Unterteilungskandidaten wird nur bei Überschreiten des Schwellenwerts durchgeführt und gegen den Kandidaten der Objektunterteilung ausgewertet, sonst erfolgt direkt eine Objektunterteilung. Durch den Bezug zur Oberfläche des Wurzelknotens finden die räumlichen Unterteilungen und ggf. Primitiv-Unterteilungen vor allem in den oberen Ebenen des Baumes statt, wo sie den größten Effekt haben. Die Autoren geben als guten Schwellenwert $\alpha = 10^{-5}$ an, da zum einen die möglichen Verbesserungen der SAH Kosten erreicht und zum anderen deutlich weniger Primitiv-Duplikate erzeugt werden, als bei der vollen SBVH Variante (mit $\alpha = 0$). Die Ergebnisse zeigen, dass SBVH beim Ray Tracing mit primären und Ambient Occlusion Strahlen eine höhere Beschleunigung erreicht, als reguläre BVHs und vorhergehende Ansätze mit Primitiv-Unterteilung.

Ein anderes Ziel verfolgen Lauterbach et al., die sich auf den schnellen Aufbau von Bounding Volume Hierarchien konzentrieren, und erste parallele Verfahren für die BVH Konstruktion auf GPUs präsentieren (Lauterbach et al. 2009). Für die Implementierung verwenden die Autoren CUDA. Das erste beschriebene Verfahren ist darauf ausgelegt die Kosten des Aufbauprozesses zu minimieren und wird „Linear Bounding Volume Hierarchy“ (LBVH) genannt. Es reduziert den Aufbau-Prozess auf ein Sortierproblem, indem die Primitive anhand räumlicher Morton Codes entlang einer raumfüllenden Morton Kurve angeordnet werden. Die resultierende Sequenz wird dann rekursiv in Intervalle aufgeteilt, welche den Knoten der Hierarchie entsprechen. Der LBVH Aufbau ist um ein Vielfaches schneller als SAH basierte Verfahren, jedoch ist die Qualität in Bezug auf die SAH deutlich schlechter, so dass die Ray Tracing Performanz um bis zu 85% niedriger ausfällt.

Als zweites Verfahren stellen die Autoren einen SAH basierten Aufbau vor, der breadth-first vorgeht und Queues für Ein- und Ausgaben verwendet, um den Aufbau zu parallelisieren. Lauterbach et al. führen die Knoten-Unterteilungen einer Ebene parallel aus, indem sie iterativ die ausstehenden Unterteilungen aus einer Eingabe-Queue verarbeiten und die folgenden Unterteilungen für die neuen Knoten in eine Ausgabe-Queue schreiben. Vor der nächsten Iteration wird ein Verwaltungsschritt durchgeführt, der die

Ausgabe-Queue bereinigt, bevor sie in kompakter Form als Eingabe verwendet wird. Ähnlich wie in (Wald 2007) wird eine Menge von gleichverteilten Unterteilungskandidaten (hier allerdings für alle drei Achsen) verwendet, welche parallel getestet werden. Für die oberen Ebenen eines top-down Aufbaus ist die Parallelisierung schwierig, da noch nicht genügend Arbeit in Form von durchzuführenden Unterteilungen vorhanden ist, um die Kapazitäten der GPU auszunutzen. Auf den untersten Ebenen entsteht ein Flaschenhals, da nur noch kleine Unterteilungen mit wenigen Primitiven durchzuführen sind. Letzteres wird durch einen weiteren Kernel speziell für die kleinen Unterteilungen gelöst, die anhand eines Schwellenwerts in eine eigene Queue abgelegt werden. Der Kernel verwendet lokalen Speicher für die Queue und die entsprechenden Primitive des Teilbaums und wird zum Schluss für alle verbleibenden kleinen Unterteilungen ausgeführt. Die Beschleunigung durch die Sonderbehandlung kleiner Unterteilungen wird mit 15% bis 20% angegeben.

Der Flaschenhals bei der Parallelisierung für die obersten Baumebenen wird im dritten Verfahren adressiert, indem die zuvor genannten Verfahren kombiniert werden, so dass die obersten Ebenen per LBVH und die übrigen Ebenen mit der SAH basierten Methode aufgebaut werden. Das hybride Verfahren erreicht eine deutliche Beschleunigung des Aufbaus gegenüber der SAH basierten Methode bei ungefähr gleich bleibender SAH Qualität.

Karras und Aila stellen ein GPU Verfahren für die schnelle parallele Verbesserung der Qualität einer bestehenden BVH vor (Karras und Aila 2013). Das Verfahren zielt auf den Einsatz in interaktiven Anwendungen ab und wird hier stellvertretend für Methoden, die auf einer Verbesserung einer Hierarchie anstatt dem kompletten Neuaufbau basieren, zusammenfassend vorgestellt. Neben der Restrukturierung von BVHs beschreiben die Autoren eine Heuristik für die Unterteilung von Primitiven vor dem BVH Aufbau und die parallele Durchführung der Unterteilung. Die Variante mit optionaler Primitiv-Unterteilung erreicht im Vergleich zum SBVH Verfahren im Durchschnitt 90% Ray Tracing Performanz bei einem Bruchteil der Zeit für den Aufbau.

Für die Restrukturierung betrachten Karras und Aila eine Knoten-Nachbarschaft in Form eines kleinen binären Teilbaumes, den sie „Treelet“ nennen. Auf Basis wiederholt definierter Treelets werden Teile der BVH lokal neu strukturiert um die SAH Kosten zu minimieren, indem die inneren Knoten verworfen und für die gleiche Menge an Treelet-Blättern neugebildet werden. Die BVs der inneren Knoten werden also verändert, so dass deren Oberflächengröße reduziert wird, was sich direkt auf die SAH Kosten auswirkt. Das Verfahren kann in drei Stufen betrachtet werden: Initialer BVH Aufbau, Optimierung der Topologie, Nachverarbeitung. Der initiale BVH Aufbau wird nach einem Verfahren in (Karras 2012) durchgeführt, in welchem das LBVH Verfahren von Lauterbach et al. (siehe weiter oben)

durch den Einsatz eines binären Radix-Baumes weiterreichend parallelisiert wird. Die Unterteilungen werden damit anhand des räumlichen Medians eines Volumens durchgeführt. In den Blättern ist jeweils nur eine Dreiecks-Referenz enthalten. Da die Anzahl von Primitiven in Blattknoten Auswirkungen auf die Ray Tracing Performanz hat, werden einzelne Teilbäume in dem späteren Nachverarbeitungsschritt ggf. zu Blattknoten mit mehreren Primitiv-Referenzen zusammengelegt. Dies muss bereits während der Optimierung beachtet werden, daher wird die Abschätzung der SAH Kosten für beide Fälle ausgeführt und das Minimum verwendet. Gleichzeitig wird dadurch die Behandlung von inneren Knoten und Blättern vereinheitlicht.

Die Optimierung startet, indem der Baum parallel bottom-up traversiert und für jeden Knoten, dessen Teilbaum groß genug ist, ein Treelet mit fester Anzahl n an Treelet-Blättern erstellt wird. Die Restrukturierung der Treelet Topologie erfolgt durch einen hochgradig optimierten Algorithmus, indem nach dem Prinzip der dynamischen Programmierung Teilprobleme (für Teilmengen der Treelet-Blätter) gelöst werden, auf denen die Gesamtlösung (für die gesamte Menge der Treelet-Blätter) aufgebaut wird. Während der Betrachtung jeder Möglichkeit der Aufteilung der Blätter für jede Teilmenge werden die optimalen Kosten und die entsprechende Aufteilung gespeichert, um daraus zuletzt die optimale Lösung für das gesamte Treelet zu bilden und umzusetzen. Während bei der Traversierung ein Knoten pro Thread verarbeitet wird, geschieht die Optimierung durch einen Warp pro Treelet.

Bei der optionalen und vorausgehenden Dreiecks-Aufteilung werden, anders als bei Stich et al. (siehe weiter oben), nicht die Primitive, sondern deren Bounding Boxes gegebenenfalls mehrfach unterteilt. Zudem findet der Prozess anhand einer Heuristik nicht während, sondern vor dem BVH Aufbau statt. Die Heuristik basiert darauf, wie sehr die Oberfläche des BV eines Dreiecks durch Unterteilung reduziert werden kann, und ob das Dreieck eine „wichtige“ Achse schneidet. Wie beim initialen BVH Aufbau wird anhand des räumlichen Medians unterteilt, bezogen auf die „wichtigste“ Achse. Die Wichtigkeit einer Achse hängt davon ab, wie früh diese beim BVH Aufbau für die Unterteilung verwendet wurde.

Der Ray Tracing Prozess wird durch die optimierten Kernel von (Aila und Laine 2009, Aila, Laine und Karras 2012) mit diffusen Interreflexions-Strahlen ausgeführt. Die Ergebnisse der BVH aus den eigenen und weiteren (z.B. LBVH und SBVH) Aufbauverfahren setzen die Autoren in Relation zur Ray Tracing Performanz der BVH aus einem Greedy top-down Aufbau („Sweep SAH“), basierend auf (MacDonald und Booth 1990) und z.B. angewandt in (Wald, Boulos et al. 2007). Für den visuell präsentierten, detaillierten Vergleich sei auf (Karras und Aila 2013) verwiesen. Zusammenfassend zeigt sich, dass je nach Anzahl der zu verfolgenden Strahlen verschiedene Verfahren am besten abschneiden. Die BVHs mancher Aufbauverfahren sind also für kleinere Szenen bzw. weniger zu verarbeitende Strahlen besser geeignet, und andere Verfahren können im mittleren oder oberen Bereich

den größten Gewinn an Performanz verzeichnen. Das Treelet Verfahren erreicht zwischen ca. 7 Millionen und 25 Billionen Strahlen die besten Werte. Mit Primitiv-Unterteilung liegt der beste Bereich für das Treelet Verfahren bei 7 Millionen bis 60 Billionen Strahlen. Zudem zeigt sich in den Durchschnittswerten der Testszenen, dass die Entwicklung der SAH Kosten der verschiedenen Hierarchien häufig nicht mit der tatsächlichen Performanz übereinstimmt. Diese Diskrepanz der SAH in Verbindung mit manchen Verfahren wird im gleichen Jahr von Aila et al. (Aila, Karras et al. 2013) untersucht (siehe unten).

Aila et al. stellen fest, dass die SAH, gemessen an der tatsächlich erreichten Ray Tracing Performanz, oftmals abweichende Vorhersagen über die Qualität von BVHs macht (Aila, Karras et al. 2013). Im Fall von Bäumen aus Greedy top-down Aufbauverfahren, unterschätzt die SAH in der Regel die Qualität, während andere Verfahren teilweise schlechter abschneiden als per SAH abgeschätzt. Daher stellen die Autoren ergänzende Metriken vor, insbesondere für die Entwicklung von Aufbauverfahren, die vom Greedy top-down Ansatz abweichen und der Erklärung der tatsächlich gemessenen Qualität. Die beiden vorgestellten Deskriptoren sind jedoch nicht dazu geeignet, den Aufbauprozess zu steuern bzw. zu verbessern, sondern tragen zum Verständnis darüber bei, wodurch sich manche Bäume besser für Ray Tracing eignen. Die grundlegenden Annahmen der SAH besagen, dass die Strahlen zum einen zufällig und gleich verteilt sind und zum anderen außerhalb, mit großem Abstand zur Szene starten. Die zweite Annahme ist insbesondere für sekundäre Strahlen nicht erfüllt. Der erste Deskriptor, genannt EPO für „End-Point Overlap“, quantifiziert die Kosten durch Überschneidungen von Teilbaum-Volumen, also den Kostenpunkt, den Stich et al. in ihrem SBVH Verfahren reduzieren (Stich et al. 2009). Aila et al. berechnen die zusätzlichen Kosten, die durch Überschneidungen von Volumen entstehen durch

$$EPO := \sum_{n \in N} C_n \frac{A((S/Q(n)) \cap n)}{A(S)}, \tag{22}$$

wobei S die Menge aller Oberflächen der Primitive der Szene ist, $A(S)$ berechnet die Größe der gesamten Oberfläche einer Menge S, C_n sind die Kosten für die Verarbeitung des Knotens n, $Q(n)$ ist die Menge der Oberflächen, die zu dem Teilbaum des Knotens n gehören und $(S\, Q(n)) \cap n)$ ist demzufolge die Menge derjenigen Oberflächen, die nicht zum Knoten n gehören, aber dennoch darin liegen (durch Überschneidung mit einem anderen Knoten). Die Oberfläche dieser Menge relativ zur Oberfläche $A(S)$ sorgt als Faktor dafür, dass die berechneten Kosten durch Überlappungen, gegeben durch EPO, dort höher ausfallen, wo mit höherer Wahrscheinlichkeit Strahlursprünge oder Endpunkte sind. Der zweite vorgestellte Deskriptor bezieht die parallele (SIMT/SIMD) Ausführung der Traversierung eines

Baumes mit ein. Die Varianz in der Anzahl der Blattknoten, die von einem Strahl verarbeitet werden, führt zu geringerer Ausnutzung der SIMD Kapazitäten. Daher berechnet der zweite Deskriptor, LCV („Leaf Count Variability“) die Standard Abweichung der Anzahl von Blattknoten N_l, die von einem Strahl geschnitten werden (Gleichung 23) und wird durch Sampling der Strahlen abgeschätzt.

$$LCV := \sqrt{E[N_l^2] - E[N_l]^2} \tag{23}$$

Für die Tests verwenden die Autoren 22 Szenen und versenden Strahlen für diffuse Interreflexion, sowohl für eine serielle (ein Thread) als auch eine SIMT/SIMD Ausführung. Bei der seriellen Ausführung wird die Performanz für manche Szenen gut durch SAH beschrieben, bei anderen Szenen zeigen sich jedoch große Abweichungen. SAH zusammen mit EPO erreicht eine bessere Abschätzung. Die lineare Kombination erfordert eine Gewichtung von SAH und EPO durch einen Parameter $\alpha \in [0, 1]$, welcher szenenabhängig zu wählen ist. EPO wird stärker gewichtet für Szenen, bei denen die Strahlen einen relativ kleinen Teil traversieren. Das Erreichen besserer Abschätzungen durch die Kombination mit EPO zusammen mit dem Ergebnis, dass die SAH die Bäume aus Greedy top-down Verfahren unterschätzt, ist ein Hinweis darauf, dass diese Aufbauverfahren implizit Überschneidungen vermeiden, wo es darauf ankommt. Das erscheint naheliegend, da große Überschneidungen häufig mit größerer Oberfläche beider Volumen einhergehen, und die Oberfläche neben der Anzahl der Primitive die Basis für die zu optimierende Kostenfunktion bzgl. der Wahl der Unterteilung bei SAH basierten Verfahren ist (z.B. Gleichung 18). Die Autoren erklären die Überlegenheit der Bäume aus jenen Aufbauverfahren durch eben diese Minimierung der Worst Case Kosten durch die Wahl des lokalen Optimums für die Unterteilung eines Knotens. Dadurch tendieren die resultierenden Bäume dazu, balanciert zu sein und eher kleine Überschneidungs-Bereiche zwischen zwei Kindknoten aufzuweisen. Ein weiterer Aspekt dieser Bäume ist, dass sie nahezu optimale Ergebnisse bezüglich SAH für Szenen erreichen, die gleichmäßig tesseliert sind.

Im Verhältnis zeigen die Ausführungszeiten für die serielle und die SIMD Ausführung, dass die Bäume aus Greedy top-down Verfahren am besten für das parallele Ausführungsmodell geeignet sind. Aila et al. nehmen an, dass dies begünstigt wird durch die einheitlichere Größe der Blattknoten im Vergleich zu anderen Verfahren und durch weniger Überschneidungen von Blattknoten. Die Genauigkeit der Abschätzung durch SAH mit EPO fällt für die SIMD Ausführung schlechter aus. Durch lineare Kombination mit dem LCV Deskriptor (durch einen weiteren Parameter β), wird die Genauigkeit deutlich erhöht und liegt ungefähr bei den gleichen Korrelationswerten wie bei SAH mit EPO für die serielle Ausführung. Interessant ist zudem die Feststellung der Autoren, dass die verschiedenen Anordnungen der

Knoten im Speicher (durch verschiedene Aufbauverfahren), in den Tests nicht ins Gewicht fallen und auch durch zufällige Permutationen der Knoten im Speicher kaum Auswirkungen auf die Performanz festzustellen sind ($< 1\%$). Dies zeigt, entgegen der gängigen Annahmen, dass die Anordnung im Speicher auf GPUs (Stand 2013, Nvidia Kepler Architektur, GTX680) als Faktor für die Performanz zumindest in diesem Rahmen scheinbar kaum eine Bedeutung hat.

3.3 GPU Ray Tracing und BVH Traversierung

Bei der Traversierung von BVHs für GPU Ray Tracer wurden in zahlreichen Arbeiten verschiedene Ansätze und Stratgien entwickelt und untersucht. Man kann grob unterscheiden in Paket-basierte Verfahren und Traversierung einzelner Strahlen sowie in Stack-basierte Traversierung und „Stackless“ Verfahren, die keinen Stack bzw. einen deutlich kleineren Ersatz verwenden. Darüber hinaus wurden Ansätze mit dem Ziel entwickelt, die Effizienz bezogen auf das SIMT Ausführungsmodell für die Traversierung bzw. den Trace Prozess zu verbessern. Im Folgenden werden einige ausgewählte Verfahren vorgestellt, um einen Einblick in die verschiedenen Strategien zu bieten und natürlich auch die im Rahmen der vorliegenden Arbeit verwendeten Verfahren zu beschreiben.

In der Anfangszeit der CUDA Architektur beschreiben Günther et al. ihr Paket-basiertes Traversierungsverfahren für einen GPU Ray Tracer mit BVH (implementiert in einem CUDA Kernel) (Günther et al. 2007). Die vorgestellte Paket-Traversierung („Packet Traversal“) verwendet einen geteilten Stack pro Paket anstatt einen Stack pro Strahl. Die Leistung der GPU soll besser ausgenutzt werden durch ein paralleles, kohärentes Traversierungsverfahren für BVHs mit einem kohärenten Verzweigungsverhalten und weniger benötigten Registerplätzen. Der geteilte Stack wird per Shared Memory realisiert. Für die Ausführung ist ein Strahl auf einen Thread und ein Paket auf ein Warp abgebildet. Das gesamte „Paket“ wird Knoten für Knoten getestet, indem jeder Strahl des Pakets bestimmt, welchen Kindknoten er trifft bzw. aufgrund der Nähe als erstes verarbeitet, um dann durch eine „Parallel RAM sum reduction“ zu entscheiden, welcher Knoten von den meisten Strahlen als nächstes traversiert werden soll und welchen Knoten das gesamte Paket demzufolge als erstes besucht. Wenn beide Knoten jeweils von mindestens einem Strahl aus dem Paket getroffen wurden, kommt der zweite Knoten auf den geteilten Stack. Bei Erreichen eines Blattes werden die Strahlen des Pakets auf Schnittpunkte mit der enthaltenen Geometrie getestet. Wenn kein innerer Knoten mehr getroffen wird oder ein Blatt verarbeitet wurde, wird der nächste Knoten vom Stack genommen und seine Kindknoten werden traversiert oder die Verarbeitung terminiert, wenn der Stack leer ist.

Die binäre BVH wird durch ein Konstruktionsverfahren aufgebaut, das

ebenfalls durch die Autoren beschrieben wird. Es ist angelehnt an „Streamed Binning“ Verfahren für kd-Trees und das SAH-basierte BVH Verfahren in (Wald, Boulos et al. 2007). Gleichzeitig ist von Wald auf Basis des letztgenannten Verfahrens ein beschleunigtes Verfahren entstanden (Wald 2007), welches dem von Günther et al. sehr ähnelt und in Abschnitt 3.2 detailliert vorgestellt wird.

Die Ergebnisse von Günther et al. zeigen, dass ihr GPU Ray Tracer eine etwas höhere Performanz erreicht, als der bis dahin schnellste kd-Tree basierte Ansatz auf derselben GPU. Die Autoren sehen dies begründet in dem Algorithmus zur parallelen BVH Traversierung, der im Vergleich einfacher ist, weniger Register benötigt und so eine höhere Ausnutzung der GPU Leistung ermöglicht. Die Effizienz der Paket Traversierung hängt allerdings stark von der räumlichen Kohärenz der Strahlen ab, die sich in einem Paket befinden. Günther et al. zeigen, dass es vor allem an Objekt-Grenzen Divergenz innerhalb der Pakete gibt. Ergebnisse wurden nur für Primär- und Schattenstrahlen präsentiert.

Die Abbildung eines Strahls auf einen Thread und eines Pakets von 32 Strahlen auf einen Warp entspricht dem SIMT Ausführungsmodell. Ohne die explizite Paketformulierung würden 32 Strahlen auf der GPU ebenfalls parallel verarbeitet werden; allerdings könnten sie während der Traversierung voneinander divergieren, so dass verschiedene Pfade durchlaufen und ggf. einzelne Threads temporär deaktiviert werden (siehe dazu Abschnitt 2.4). Günther et al. hingegen lassen alle Strahlen eines Pakets ggf. alle Pfade durchlaufen, sobald zumindest ein Strahl den Pfad einschlägt, was insbesondere bei weniger kohärenten Strahlen zu redundanten Berechnungen führt. Wie gut dieser Paket-basierte Ansatz für die Ausführung auf der GPU geeignet ist, wird unter anderem in der nächsten beschriebenen Arbeit betrachtet.

Aila und Laine untersuchten 2009 die Effizienz verschiedener Ray Tracing Traversierungs-Strategien auf der GPU (mit Fokus auf Nvidia Architekturen), insbesondere im Hinblick auf die Nähe zum theoretischen (simulierten) Leistungslimit für die, zu dem Zeitpunkt, aktuellen GPU Architekturen (Aila und Laine 2009). Es geht um die Abbildung des Trace Prozesses, definiert als Traversierung einer BVH gefolgt vom Schnittpunkttest mit Primitiven, auf das SIMT-Ausführungsmodell der GPUs. Die dafür verwendeten Kernel sind in CUDA implementiert und hoch optimiert. Bei der Traversierung der BVH kommt ein Stack zum Einsatz. Die Messwerte der von Aila und Laine implementierten Kernel werden mit Werten aus einem ebenfalls entwickelten Simulator verglichen, der eine Obergrenze für die theoretische Leistung der verwendeten GPU (Nvidia GTX285) liefert. Die verglichenen Daten, für jede Methode jeweils simuliert und tatsächlich erreicht, umfassen die „SIMD efficiency“ als die relative Anzahl an tatsäch-

lich arbeitenden Threads und die Anzahl an Strahlen pro Sekunde. Es wird differenziert zwischen Primärstrahlen, Ambient Occlusion und Strahlen für diffuse Interreflexion. Die Autoren stellen in der Arbeit fest, dass vorherige Methoden um einen Faktor 1,5 bis 2,5 unter dem theoretischen Optimum liegen, und dies nicht auf die Speicherbandbreite zurückzuführen ist, sondern hauptsächlich an der Arbeitsverteilung der Hardware liegt. Die verschiedenen betrachteten Traversierungs-Strategien, „Packet Traversal", „*while-while*" und „*if-if*", werden im Folgenden zusammenfassend vorgestellt sowie zwei der vorgeschlagenen Ansätze für eine Verbesserung der SIMD-Effizienz bezogen auf die betrachtete GPU Architektur, „Speculative Traversal" und „Persistent Threads".

Für die Betrachtung der Paket-basierten Traversierung richten sich Aila und Laine nach der Methode von Günther et al. (2007), welche oben bereits beschrieben wurde. Die Methode erreicht bei Aila und Laine im Vergleich zu den Strategien für einzelne Strahlen (*while-while* und *if-if*) durchgehend deutlich schlechtere Messwerte, sogar für Primärstrahlen und ist am weitesten von den Simulationswerten entfernt. Der kohärente Speicherzugriff bei der Paket Traversierung kann nicht kompensieren, dass ggf. Strahlen viele Knoten besuchen, die sie nicht schneiden. Die *while-while* Methode (Listing 9) erreicht im Simulator die besten Werte, und die Messwerte sind näher an der Simulation.

Listing 9: Pseudocode *while-while* Traversierungs-Strategie („while-while trace()" nach Aila und Laine 2009)

```
while Strahl nicht terminiert
 while innerer Knoten
  traversiere weiter
 while Blattknoten enthält ungetestete Primitive
  führe Schnitttests durch
```

Bei der *if-if* Methode (Listing 10) sind die Messwerte, zur Überraschung der Autoren, sogar besser als bei *while-while*, wobei die Simulationswerte schlechter ausfallen und im Fall der Primärstrahlen sogar unter den Simulationswerten der Paket Traversierung liegen. Per *if-if* werden bessere Werte erreicht, wenn maximal vier Dreiecke in den Blattknoten gehalten werden. Die Autoren erklären diesen Vorsprung dadurch, dass *if-if* zu weniger besonders lang laufenden Warps führt.

Listing 10: Pseudocode *if-if* Traversierungs-Strategie („if-if trace()" nach Aila und Laine 2009)

```
while Strahl nicht terminiert
 if innerer Knoten
  traversiere weiter
 if Blattknoten enthält ungetestete Primitive
  führe Schnitttest durch
```

Da die Ausführungszeit einzelner Strahlen sich stark unterscheiden kann, ist es möglich, dass die Arbeitsverteilungs-Kapazitäten der GPU durch lang laufende Warps mit nur wenig aktiven Threads belegt werden. Aila und Laine versuchen dieses Problem zu lösen, indem sie die Arbeitsverteilung umgehen. Dafür starten sie genügend Threads, um die Kapazitäten einmal zu füllen und dann durch diese lang laufenden „Persistent Threads“ die Arbeit aus einem globalen Pool mit Hilfe eines Atomic Counters nachzuholen, bis der Pool leer ist. Damit soll Unterauslastung vermieden und gleichzeitig die Frage geklärt werden, ob die Arbeitsverteilung der GPU einen negativen Einfluss auf die Effizienz der Trace-Kernel hat. Aila und Laine erreichen mit diesem Ansatz der Arbeitsverteilung sowohl für die Paket Traversierung als auch für die *while-while* Methode eine deutlich höhere Performanz, wobei *while-while* weiterhin überlegen ist, insbesondere natürlich für inkohärente Strahlen. Während also die automatische Arbeitsverteilung der verwendeten GPU einen negativen Einfluss auf die Effizienz der Trace-Kernel hat, zeigt sich noch keine signifikante Limitierung der Leistung durch die Speicherbandbreite. Für Nvidias nachfolgende Architektur, Fermi, stellen Aila et al. später fest, dass durch eine bessere automatische Arbeitsverteilung dieses Vorgehen kaum mehr nützlich ist (Aila, Laine und Karras 2012). Für die darauf folgende Kepler-Architektur wird in derselben Arbeit unter gewissen Voraussetzungen ein Leistungsgewinn für inkohärente Strahlen beim Einsatz von Persistent Threads festgestellt.

Ein weiterer vorgestellter Ansatz für die potenzielle Verbesserung der SIMD-Effizienz ist die spekulative Traversierung („Speculative Traversal“). Die Divergenz, die bei der Traversierung entstehen kann, wenn Strahlen aus dem Warp bereits Blattknoten mit Primitiven gefunden haben, und andere noch weiter traversieren, führt dazu, dass die entsprechenden Threads ausmaskiert werden und sich im Leerlauf befinden, während die anderen Threads weiter traversieren. Gegebenenfalls müssen Strahlen, die ein Primitiv schneiden, an anderer Stelle wieder mit der Traversierung beginnen, um festzustellen, ob es sich um den vordersten Schnittpunkt handelt. Bei der spekulativen Traversierung nehmen die Strahlen, die bereits Blätter gefunden haben, weiterhin an der Traversierung teil, anstatt inaktiv zu bleiben bis die Traversierung anderer Strahlen abgeschlossen ist. Die Schnitttests werden also verzögert, wobei die Autoren für die Bewahrung der Einfachheit und zur Einschränkung der Anzahl evtl. umsonst besuchter Knoten eine Begrenzung auf ein verzögertes Blatt vorschlagen. Dieser Ansatz führt zu Verbesserungen, weniger bei Primärstrahlen, mehr bei Ambient Occlusion und entgegen der Simulationswerte ist keine Steigerung bei der diffusen Interreflexion vorhanden. Die Autoren sehen darin den ersten Hinweis auf eine Limitierung durch Speicherbandbreite aufgrund der redundanten Speicherzugriffe auf Knoten. Für spätere Architekturen stellen Aila et al. in (Aila, Laine und Karras 2012) kleinere Verbesserungen hauptsächlich für inkohärente Strahlen und große Szenen fest.

Darüber hinaus werden Möglichkeiten vorgestellt, welche potenziell zu einer Erhöhung der SIMD-Effizienz führen könnten, zu dem Zeitpunkt jedoch durch den Overhead für zusätzliche Berechnungen und aufwändige Operationen die Effizienz entweder nur gering verbessern oder verschlechtern. Eine Erweiterung des „Persistent Threads“ Ansatzes ersetzt terminierte Strahlen in regelmäßigen Abständen durch neue, die von der Wurzel starten. Dieser Ansatz brachte keine nennenswerte Verbesserung durch die zusätzlich nötigen Berechnungen und weniger Kohärenz der Speicherzugriffe. Außerdem wurde versucht eine Work Queue einzusetzen, um 32 zusätzliche Strahlen pro Warp (im Shared Memory) zu halten und zu geeigneten Zeitpunkten so auszutauschen, dass nach Möglichkeit immer 32 aktive Strahlen im Warp die gleiche Operation (Traversieren oder Schnitttests) durchführen. Dafür wird gezählt, wie viele der 64 Strahlen aktuell traversieren und wie viele darauf warten, einen Schnitttest auszuführen. Die größere Menge wird eingesetzt (durch Prefix Sum Operation zur Berechnung von SIMD Lanes und Speicherpositionen und Austausch der entsprechenden Strahlen), was sich als sehr aufwendig erwiesen hat, da auf der Architektur keine Instruktionen für Prefix Sum und das bedingte Zählen von Threads („population count“) vorhanden sind. Außerdem wurden weite Bäume mit einem Verzweigungsfaktor von 4, 8 und 16 getestet, welche jedoch zu Verschlechterungen der Performanz führten (gegenüber binären Bäumen).

In (Aila, Laine und Karras 2012) wird für die Folgearchitekturen hinzugefügt, dass die Ray Tracing Performanz bzgl. Traversierung und Schnitttests weiterhin mit der Rechenleistung steigt und noch nicht von der Speicherbandbreite limitiert wird.

In der hier vorliegenden Arbeit wurden im Rahmen des implementierten Ray Tracers per OpenGL Compute Shader für die Traversierung der binären BVH mit der *while-while* und der *if-if* Schleifenorganisation nach (Aila und Laine 2009) experimentiert. Die Strategien wurden sowohl mit einer Stack-basierten als auch „Stackless“ Traversierung (dazu später mehr) eingesetzt. Auf der Nvidia Maxwell Architektur (hier GTX970) hatten *if-if* Ansätze zumeist schlechtere Performanz als die *while-while* Organisation (weiteres in Abschnitt 5). Das könnte mit den Verbesserungen der Maxwell Architektur für eine vereinfachte, verbesserte Arbeitsverteilung (Nvidia 2015b) zusammenhängen.

Garanzha et al. strukturieren ihre CUDA Implementierung eines GPU Ray Tracers, wie in Abschnitt 3.1 beschrieben, hinsichtlich der Datenparallelität. Die BVH Traversierung basiert auf kohärenten Strahlenpaketen, die durch Frusta repräsentiert werden und verläuft breadth-first und ohne Stack (Garanzha und Loop 2010). Auch neben dem Traversierungs-Vorgang unterscheidet sich das Verfahren deutlich vom vorher betrachteten Paket-basierten Ansatz in (Günther et al. 2007), z.B. wird die Kohärenz der Strahlen

eines Pakets explizit durch Sortierung erzeugt und anstatt die einzelnen Strahlen eines Paktes mit BVs zu schneiden, wird ein Schnitttest mit dem Frustum durchgeführt.

Für die Zusammenfassung der Strahlen zu Paketen werden die Strahlen zunächst mit Hilfe von Hash-Werten sortiert. Das Generieren der Hash-Werte geschieht durch Quantifizierung des Ursprungs und der Richtung des Strahls, gefolgt von der Einordnung dieser Werte in ein Grid. Der Hash-Wert wird dem Strahl-Index als Schlüssel zugeordnet, und Strahlen mit gleichem Hash-Wert sind räumlich kohärent. Die Schlüssel-Wert Paare aus Hash und Strahl-Index werden komprimiert, per Radix Sort sortiert und wieder dekomprimiert. Räumlich kohärente Strahlen sind dann in aufeinanderfolgenden Speicherplätzen untergebracht, um die Divergenz beim Tracing Prozess zu minimieren. Für Komprimierung, Dekomprimierung und bei der Traversierung werden Präfixsummen-Operationen („parallel scan“) eingesetzt. Die Präfixsumme eines Arrays ergibt ein Ergebnis-Array, bei dem jedes Element die Summe der vorhergehenden Elemente des Eingabe-Arrays ist. Der „parallel segmented scan“ berechnet parallel die Präfixsumme für definierte Segmente des Eingabe-Arrays.

Für die Traversierung der BVH werden die Strahlenpakete durch Frusta repräsentiert, welche, wie erwähnt, statt der einzelnen Strahlen auf Schnitt mit den BVs getestet werden. Für jedes Frustum wird während der Traversierung eine räumlich sortierte Liste von den Blattknoten gebildet, die geschnitten wurden. Die binäre BVH wird auf der CPU erstellt, zwei Drittel der Baumebenen werden dann ersetzt durch eine Octo-BVH, um die Tiefe des Baumes zu verringern. Alle Knoten werden in breadth-first Ordnung gespeichert und die Kindknoten eines Knotens räumlich aufsteigend sortiert. Die breadth-first Traversierung beginnt bei der Wurzel. Auf jeder Ebene werden die Strahlen Frusta parallel mit allen Knoten der Ebene per AABB-Frustum Culling auf Schnitt getestet. Für die nächste Ebene der BVH werden die Strahlen neu geordnet. Durch die parallele breadth-first Verarbeitung wird kein Stack benötigt, da keine Verzögerung der Verarbeitung getroffener Nachbarknoten stattfindet. Allerdings wird für die Traversierung ein großer Speicherblock für „Work Queues“ reserviert, um die Frustum IDs, die Knoten IDs der aktuellen Ebene, die Listen der IDs der getroffenen Kindknoten und Zwischenergebnisse in Arrays zu verwalten. Am Ende der Traversierung werden die IDs der aktiven Frusta und der entsprechenden Bereiche von getroffenen Blattknoten durch einen Kompressions-Vorgang (mit den Frustum IDs anstatt Hash-Werten) herausgezogen, um lokalisierte Schnittpunkttests mit den Primitiven durchzuführen. Die aktiven Frusta werden in Mengen von maximal 32 Strahlen zerlegt (ein Warp), welche dieselbe Frustum ID teilen. Durch Verwendung der sortierten Blattliste eines Frustums werden die nächsten Primitive zuerst auf Schnitt mit den Strahlen getestet.

Nach (Aila und Laine 2009) verwenden die Autoren „Persistent Threads“

für die Verteilung der Arbeit. Ein Vergleich der Autoren mit einer Implementierung des „persistent speculative while-while“ Kernel nach (Aila und Laine 2009), der einen Stack verwendet und depth-first Traversierung durchführt, zeigt, dass das vorgestellte Verfahren für Primärstrahlen und Schattenstrahlen (weiche Schatten) schneller ist.

Wald et al. zeigen im Rahmen eines (CUDA) Path Tracers eine Möglichkeit, aus mehreren teilweise aktiven Warps weniger Warps zu generieren, die vollständig aktiv sind (Wald 2011). Das Ziel ist, wie bei (Aila und Laine 2009), die (SIMD-) Effizienz zu erhöhen, indem die 32 verfügbaren Thread Slots pro Warp vollständig ausgenutzt werden. Die Verdichtung der Warps wird durchgeführt, wenn die Auswertung für einen Strahl (ein Pfadsegment) abgeschlossen ist, und die ausgehenden Strahlen bestimmt sind. Die terminierten Threads eines Warps können zu diesem Zeitpunkt entfernt werden. Der Status von Pfaden wird in einem Array festgehalten und für die Verdichtung verwendet.

Die Autoren beschreiben zwei Versionen für den Verdichtungsprozess. Der erste arbeitet global, alle Pfade müssen nach einem Bounce im Speicher abgelegt und wieder gelesen werden, so dass keine Daten in den Registern verbleiben können. Die Verdichtung wird durch einen extra Kernel (aus der „CUDA Performance Primitives“ Bibliothek) durchgeführt. Durch die Aufteilung wird ein Teil des Kontrollflusses auf den Host (CPU) ausgelagert. Dieses Vorgehen erfordert die globale Synchronisation durch Barrieren, da die Pfade nach jedem Kernel in den Speicher geschrieben und auch wieder ausgelesen werden müssen. Die zweite Variante führt die Verdichtung nur für Threads innerhalb eines Thread Blocks aus, so dass alle Bounces in einem Kernels stattfinden, und keine globale Synchronisation benötigt wird. Dafür werden die Zustände der Pfade im Shared Memory gehalten. Da jedoch in einem Thread Block aufgrund einer hohen Anzahl genutzter Register zu wenige Threads für eine wirkungsvolle Verdichtung über mehrere Bounces gehalten werden, wird jedem Thread eine „Kachel“ (Tile) der Größe $64 \times M$ zugeordnet, so dass ein Thread mehrere Pfade verarbeitet. Zusätzlich werden auch die Daten für den Verdichtungsprozess im Shared Memory gehalten, wobei das Maximum für M durch die Datenmenge auf 8 festgelegt werden muss.

Die Ergebnisse zeigen, dass die Verdichtung zwar zu deutlich weniger Ausführungen der Kernel führt (potenziell bis zu dreimal weniger), zur Überraschung der Autoren allerdings nur wenig für die Performanz bringt (im besten Fall 16%). Der zweite Ansatz, der lokal im Thread Block arbeitet und dafür Shared Memory einsetzt, läuft sogar bis zu dreimal langsamer. Die weiteren Untersuchungen der Autoren geben Anhaltspunkte für die Ursachen dieser Diskrepanz zwischen Anzahl an Kernel Ausführungen und Ausführungszeit. Die Ausführungszeit für die verdichteten, vollen Warps steigt mit jedem Bounce deutlich an, was sich durch die sinkende

Kohärenz der zusammengefassten Strahlen und die damit einhergehende (Kontrollfluss-) Divergenz erklären lässt. Ohne Verdichtung verringert sich die benötigte Ausführungszeit pro Warp mit jedem weiteren Bounce deutlich. Die Autoren vermuten zudem, dass zusätzlich die Latenzen der weniger kohärenten Speicherzugriffe der verdichteten Warps nicht mehr ausreichend durch die Hardware versteckt werden können. Die deutliche Verlangsamung beim Thread Block-lokalen Ansatz wird auf die hohe Ressourcennutzung (Register, Shared Memory) zurückgeführt. Je mehr Ressourcen genutzt werden, desto weniger Warps können gleichzeitig auf dem Multiprozessor untergebracht werden, so dass die Kapazität für Hardware Multithreading und damit für das Verstecken von Latenzen geringer ausfällt (siehe auch Abschnitt 2.4). Für den Kernel (also pro Thread) werden 64 Register genutzt und damit ein Registerplatz mehr, als für die verwendete Nvidia Fermi Architektur pro Thread als Maximum definiert ist (Nvidia 2015a). Zusätzlich wird ein großer Anteil des verfügbaren Shared Memory verwendet. Wald et al. sehen in dem Verdichtungs-Verfahren Potenzial, wenn die Hardware entsprechend verbessert wird (Stand 2011).

Anzumerken ist, dass das Problem der Warp-internen Divergenz, welche durch die Verdichtung verstärkt wird (besonders für spätere Bounces) und damit einhergehend die größeren Latenzen inkohärenter Speicherzugriffe, bei einer Ressourcennutzung am Limit, auch auf aktuellerer Hardware (z.B. Nvidias Maxwell Architektur) zu geringerer Belegung der Multiprozessoren führen würde und damit weniger Möglichkeiten für das Verstecken von Latenzen gegeben sind (Nvidia 2015a, Nvidia 2015b). Das Limit bzgl. der Ressourcen fällt bei aktuelleren GPUs allerdings höher aus, als auf der von Wald et al. verwendeten Fermi Architektur.

Ein großer Stack-basierter Traversierungszustand pro Strahl kann auf hoch parallelen Architekturen großen Speicher- und Zugriffsaufwand mit hohen Latenzen verursachen. Für die Traversierung per GPU sind daher Verfahren interessant, die eine Traversierung ohne Stack erlauben („Stackless Traversal“).

Eines der ersten „Stackless“ Traversierungsverfahren für binäre BVHs wurde von Laine vorgestellt und basiert auf einem „Restart Trail“ (Laine 2010), angelehnt an Verfahren für die kd-Tree Traversierung ohne Stack. Der „Restart Trail“ kann als 32 Bit oder 64 Bit Integer Variable im schnellen Registerspeicher implementiert werden und speichert in einem Bit pro Baum-Ebene, welche Teile der Hierarchie bereits verarbeitet wurden, so dass ein Neustart von der Wurzel aus den Pfad zum noch nicht verarbeiteten Teilbaum finden kann. Dies erlaubt die depth-first Traversierung ohne Stack oder mit einem „Short Stack“ (einem sehr kleinen Stack), in beliebiger Reihenfolge (z.B. den näheren Knoten als Erstes, wenn beide getroffen wurden).

Ohne Stack wird ein Neustart immer dann ausgeführt, wenn ein Stack-

basiertes Verfahren einen Knoten vom Stack holen würde (Pop-Operation). Bei Einsatz eines Short Stacks geschieht dies erst, wenn die wenigen Stack Plätze erschöpft sind, der Short Stack also nicht ausreicht. Bei einem Neustart führt die „Spur“ zu dem Knoten, dessen Kinder als nächstes verarbeitet werden müssen. Eine 0 in der Spur `trace` bedeutet, dass der entsprechende Knoten noch nicht besucht wurde oder, dass beide Kindknoten des Knoten traversiert werden müssen und der Teilbaum des Näheren noch nicht komplett verarbeitet wurde. Eine 1 in `trace` bedeutet, dass für den Knoten nur einer seiner Kindknoten traversiert werden muss oder, dass beide Kindknoten traversiert werden müssen, aber der Teilbaum des Näheren bereits komplett verarbeitet wurde. Das erste Element ist immer 0 und dient als Markierung für die Terminierung des Verfahrens. Das zweite Element, also das zweite Bit, steht für die Wurzel.

Neben der Variable für den Restart Trail werden zwei weitere Variablen benötigt, `level` zeigt auf das Bit in der Spur, welches zum Elternknoten des aktuellen Knotens gehört (initial 0) und `popLevel` speichert die Ebene, zu welcher die letzte Pop-Operation geführt hätte. Zu Beginn werden alle Bits in `trace` auf 0 gesetzt. In einer *while-while* Schleifenorganisation wird zunächst über die inneren Knoten iteriert bis entweder kein Knoten mehr getroffen oder ein Blattknoten erreicht wird und `level` wird durch Inkrementierung (bitweise Shift nach links) entsprechend aktualisiert. Wenn beide Kindknoten getroffen werden und das Bit in `trail` an der Stelle `level` eine 0 ist, wird beim näheren Kindknoten weiter traversiert, und wenn ein Short Stack verfügbar ist, wird der andere Kindknoten auf den Stack gelegt. Wenn das Bit gleich 1 ist, wird beim weiter entfernten Kindknoten weiter traversiert, da der andere Teilbaum bereits verarbeitet wurde. Wenn nur ein Kindknoten getroffen wurde, wird geprüft ob `popLevel` der Ebene der Kinder des getroffenen Knotens entspricht. Wenn nicht, kommt an der Ebene eine 1 in `trace`, und es geht bei den Kindknoten weiter. Andernfalls wurden ursprünglich beide Kindknoten geschnitten und der nähere ist bereits verarbeitet. Da nun nur noch ein Kindknoten getroffen wurde, muss in der vorhergehenden Verarbeitung des näheren Knotens ein Primitiv geschnitten und der Strahl entsprechend gekürzt worden sein, so dass der weiter entfernte Knoten nicht mehr getestet wird. In dem Fall würde eine Pop-Operation stattfinden, hier bedeutet dies:

- `level` muss auf die Ebene des nächsten unverarbeiteten Knotens gesetzt werden, was dem nächsten Bit mit Wert 0 in `trace` oberhalb der aktuellen Ebene entspricht,
- `trace` wird aktualisiert, indem das Bit an der Stelle auf 1 gesetzt wird, da der nähere Teilbaum bereits verarbeitet wurde und
- die verbleibenden Bits an den Stellen größer als `level` werden auf 0 gesetzt, da der weiter entfernte Teilbaum noch nicht verarbeitet wurde,

- `popLevel` wird auf `level` gesetzt, da ein Stack Pop zu dem weiter entfernten Knoten auf dieser Ebene geführt hätte und bei einem Neustart der Traversierung somit klar ist, wann der Knoten erreicht ist.

Wenn das erste Bit in der Spur auf 1 gesetzt wurde, terminiert das Verfahren. Sonst kann nun der erste Knoten vom Stack weiterverarbeitet werden, sofern ein Short Stack verwendet wird. Wenn der Short Stack erschöpft ist, oder keiner verwendet wird, muss ein Neustart bei der Wurzel stattfinden. Auch wenn kein Knoten mehr getroffen oder ein Blattknoten verarbeitet wurde, werden die obigen Schritte (anstatt eines Stack Pop) durchgeführt. Sobald ein Blattknoten getroffen wird, werden alle enthaltenen Primitive auf Schnittpunkte getestet und ggf. der Abstand zu einem Schnittpunkt für die Kürzung des Strahls bei der weiteren Traversierung verwendet.

Von den Testergebnissen berichten die Autoren, dass im Vergleich zu einem Stack-basierten Verfahren durch den Neustart ungefähr 2,2 bis 2,4 Mal so viele Knoten besucht werden müssen. Bei Verwendung eines Short Stack mit einem Platz sind es noch 1,3 bis 1,4 Mal und bei drei Plätzen 5% bis 8% mehr Knoten. Zu dem Zeitpunkt konnten die gewonnenen Einsparungen bezüglich der Speicherzugriffe auf einen Stack jedoch nicht die zusätzlichen Kosten durch Neustarts und durch weitere Instruktionen für die Verwaltung der Spur kompensieren.

Ein weiteres Traversierungsverfahren ohne Stack, das zudem die kostenintensiven Neustarts vermeidet, wurde von Barringer und Akenine-Möller vorgestellt, in Varianten für die implizite Repräsentation eines (voll besetzten) Binärbaumes und für nicht voll besetzte Bäume („Sparse Trees“) mit Links zu den Elternknoten (Barringer und Akenine-Möller 2013). Dabei ist die Reihenfolge der besuchten Knoten dieselbe, wie beim Einsatz eines Stacks. Die impliziten Binärbäume werden breadth-first im Speicher abgelegt, so dass anhand der Adresse eines Knotens die Adresse zu den Kindknoten, dem Eltern- und dem Geschwisterknoten berechnet werden kann, und somit keine weiteren Pointer bzw. Indizes bei einem Knoten gespeichert werden müssen. Für nicht balancierte Bäume werden die „fehlenden“ Knoten von leeren Speicherplätzen repräsentiert. Ein Knoten auf Ebene d hat dann eine Adresse im Bereich $2^d - 1, ..., 2^{d+1} - 2$, wobei $d = 0$ für die Wurzel steht. Bei den nicht voll besetzten Bäumen lässt sich die Position der Verwandten eines Knotens nicht unbedingt berechnen, daher wird bei einem Knoten die Verbindung (Pointer oder Index) zu seinem Elternknoten gespeichert, um per „Backtracking“ in der Hierarchie zurück bis zu dem Teilbaum zu gelangen, der als nächstes verarbeitet werden muss.

Die Autoren beschreiben das Verfahren erst anhand einer festen Reihenfolge (der linke Knoten zuerst) und erweitern es dann für das Vorgehen in dynamischer Reihenfolge (der nähere Knoten zuerst). Ähnlich wie in (Laine 2010) (siehe oben) werden zwei Integer Variablen verwendet, um

den Traversierungszustand festzuhalten, und pro Baumebene wird ein Bit benötigt. Für die impliziten Bäume muss jedoch kein „Trail“ gespeichert werden, sondern die Anzahl der Knoten eines Levels (2^d) durch `levelStart` und den Index des aktuellen Knotens relativ zur Ebene durch `levelindex`. Für eine feste Reihenfolge, in welcher der linke Kindknoten immer als Erstes verarbeitet wird, reicht dies aus.

Der Index des ersten Knotens einer Ebene ergibt sich dann aus $levelStart - 1$, und der Index zum aktuellen Knoten kann somit berechnet werden durch $levelStart + levelindex - 1$. Bei der Traversierung zum linken Kindknoten werden die Variablen durch bitweise Shift Operationen nach links aktualisiert, so dass sie auf den linken Kindknoten der nächsten Ebene zeigen. Wenn ein Blattknoten verarbeitet oder der Kindknoten nicht getroffen wurde, wird `levelIndex` auch verwendet, um anhand der Anzahl der angehängten Nullen die entsprechende Anzahl an Ebenen im Baum aufzusteigen. Dafür werden die Nullen gezählt und beide Variablen durch eine Shift Operation um diese Anzahl nach rechts aktualisiert. Wenn der Aufstieg bei der Wurzel ankommt, also kein weiterer Knoten zu verarbeiten ist, terminiert das Verfahren.

Für eine dynamische Bestimmung der Reihenfolge wird darüber hinaus eine dritte Integer Variable, `swapMask` verwendet. Wenn ein Strahl auf einer Ebene beide Kindknoten trifft, und der rechte (da er näher ist) zuerst verarbeitet werden soll, setzen Barringer und Akenine-Möller eine Funktion ein, um die Indizes der Knoten einer Ebene n auszutauschen (Gleichung 24). Für einen Austausch auf mehreren Ebenen ist die Komposition der entsprechenden Anzahl an Funktionsanwendungen notwendig. In `swapMask` wird bitweise gespeichert, auf welcher Ebene die Traversierung nach rechts gegangen ist, um die Berechnung der Funktionskomposition zu vereinfachen (Gleichung 25).

$$f_n(levelIndex) = levelIndex + 2^n - 2(levelindex \wedge 2^n) \tag{24}$$

$$f_{comp}(levelIndex) = levelindex + swapMask - 2(levelindex \wedge swapMask) \tag{25}$$

Dadurch wird die Verwaltung des Traversierungszustands und die Berechnung des aktuellen Knotenindexes aufwändiger. Als Optimierung schlagen die Autoren vor, `swapMask` durch eine Variable für den dynamischen Index zu ersetzen und inkrementell zu aktualisieren.

Für Bäume, die nicht voll besetzt sind, oder wenn kein implizites Speicher-Layout für den Baum verwendet werden kann, zeigen die Autoren eine verallgemeinerte Variante des Verfahrens, die darauf basiert, das jeder Knoten zusätzlich den Index zu seinem Elternknoten speichert. Diese zusätzliche Verbindung erlaubt den Aufstieg in der Hierarchie auf die richtige Ebene entlang der Indizes zu den Eltern. Dadurch werden der dynamische Index und `levelStart` ersetzt. Anstatt über eine bestimmte Anzahl an Shift Operationen zur entsprechenden Ebene zu springen, findet also ein iteratives

Backtracking statt. Für den Traversierungszustand wird nur noch eine Integer Variable benötigt, welche pro Ebene in einem Bit speichert, ob ein Kindknoten getroffen wurde oder beide getroffen wurden und somit die Anzahl an Backtracking Stufen liefert. Zurück an dem Knoten angekommen, dessen Teilbaum bereits verarbeitet wurde, muss noch der Nachbarknoten erreicht werden, was laut Barringer und Akenine-Möller je nach Baumstruktur im Speicher durch Berechnung oder einen Round-Trip über den Elternknoten durchgeführt werden kann. Die Beschreibung dieser Variante durch die Autoren fällt vergleichsweise kurz aus

Für die Ergebnisse wurden zwei Testszenen mit Primärstrahlen und Ambient Occlusion Strahlen durch einen CPU Ray Tracer gerendert. Barringer und Akenine-Möller stellten fest, dass unter den vorgestellten Varianten diejenige für implizite Binärbäume mit inkrementeller Aktualisierung des dynamischen `levelIndex` die besten Werte erreicht. Anhand der Ergebnisdaten ist die Variante für nicht voll besetzte Bäume für die kleinere Szene fast genauso gut und für die komplexere Szene besser. Vergleiche mit dem Neustart-Verfahren in (Laine 2010) (siehe oben) zeigen, dass die Restart-Variante ohne und mit kleinem Short Stack in den Tests auf deutlich schlechtere Zeiten kommt, mit einem Short Stack für acht Elemente allerdings vorne liegt. Die besten Werte zeigt eine Traversierung mit vollständigem Stack.

Da die Tests mit einem CPU Ray Tracer durchgeführt wurden, liefert die Arbeit leider keine Erkenntnisse darüber, wie das Stackless-Verfahren im Vergleich zur Stack-basierten Traversierung auf der GPU abschneidet. Aufgrund der hohen Latenz für Zugriffe auf den globalen Speicher der GPU dürften Verfahren ohne einen Stack und mit kleinem Traversierungszustand im Registerspeicher sich vorteilhaft auf die Performanz auswirken.

Der im Rahmen der vorliegenden Arbeit implementierte GPU Ray Tracer verwendet unter anderem die Stackless Traversierung basierend auf einem Verfahren, welches dem obigen (nicht implizite Variante) sehr ähnelt und etwas später vorgestellt wurde: In (Áfra und Szirmay-Kalos 2014) wird ein weiterer Algorithmus für die BVH-Traversierung mit dynamischer Reihenfolge und ohne Stack vorgestellt. Der Fokus liegt auf der Traversierung von „Multi Bounding Volume Hierarchies“ (MBVHs), welche mehrere BVs pro Knoten speichern. Bei der Variante für binäre MBVHs lassen sich viele Parallelen zu der oben beschriebenen Variante für nicht voll besetzte Bäume aus (Barringer und Akenine-Möller 2013) erkennen. Áfra und Szirmay-Kalos beschreiben das Verfahren für MBVHs mit Verzweigungsfaktor von 4 und 2, wobei als bevorzugte Variante für die Traversierung auf GPUs (Nvidia Kepler Architektur) die binäre MBVH angegeben wird. Dementsprechend wird in der hier folgenden Beschreibung der Schwerpunkt gesetzt.

Die Reihenfolge der besuchten Knoten soll der einer Stack-basierten Traversierung entsprechen und der nähere getroffene Knoten wird als erstes verarbeitet. Wie im Algorithmus für „Sparse Trees“ aus (Barringer und

Akenine-Möller 2013) wird für jeden Knoten auch der Index (oder Pointer) zu seinem Elternknoten gespeichert, um das Aufsteigen im Baum (Backtracking) zu ermöglichen. Zusätzlich speichern Áfra und Szirmay-Kalos auch den Index zum Geschwisterknoten, damit kein Round-Trip über den Elternknoten notwendig ist. Das Backtracking wird durch einen „Bitstack" gesteuert, einer Bitmaske, die hier ebenfalls in Form einer Integer Variablen (32 oder 64 Bit) implementiert wird, wobei sich die Arbeitsweise mit diesem Stack-Ersatz im Detail unterscheidet, das Ziel aber das Gleiche ist. Die Bezeichnung wählen die Autoren, da die bitweise Operationen auf dem Bitstack analog zu den Operationen auf einem Stack sind, also „Push" (durch Shift nach links) und „Pop" (durch Shift nach rechts). Durch die Verwendung der Bitmaske als Stack ergeben sich gegenüber dem vorhergehenden Verfahren Vorteile bezüglich einer frühzeitigen Terminierung. Bei der binären Hierarchie speichert der Bitstack für jede besuchte Ebene, außer der Wurzelebene in einem Bit („Skip Code"), ob

- der Geschwisterknoten des aktuellen Knotens übersprungen und zum Elternknoten zurückgegangen werden soll durch eine 0 oder
- der Geschwisterknoten des aktuellen Knotens traversiert werden soll durch eine 1.

Initial wird der Bitstack auf 0 gesetzt, was einem leeren Stack entspricht und ein Kriterium für die Terminierung der Traversierung ist. Somit muss das Verfahren hier nicht erst zur Wurzel zurückkehren und kann ggf. frühzeitig terminieren, wenn kein Knoten mehr getroffen wird und kein Geschwisterknoten mehr zu testen ist, ohne redundante Backtracking Schritte durchführen zu müssen.

Die depth-first Traversierung beginnt ausgehend von der Wurzel mit der Verarbeitung der inneren Knoten in einer Schleife. Es werden jeweils die beiden Kindknoten des aktuellen Knotens getestet, und bei einem Treffer wird von rechts eine Null auf den Bitstack gelegt. Wenn nur einer der beiden Kindknoten getroffen wurde, fährt die Traversierung bei dem entsprechenden Knoten fort; wurden beide Kindknoten getroffen, wird zunächst der nähere zum aktuellen Knoten und das (least significant) Bit auf dem Bitstack wird auf 1 gesetzt (durch bitweise OR). Sobald der aktuelle Knoten ein Blattknoten ist, wird die Traversierung der inneren Knoten verlassen und die Primitive des Blattknotens werden auf Schnittpunkte getestet. Ein Schnittpunkt, der näher ist als ggf. zuvor gefundene Primitiv-Schnittpunkte, wird als bislang nächster Schnittpunkt aufgenommen, und der Strahl wird etsprechend gekürzt (Knoten, die weiter entfernt sind als die Schnittpunktdistanz, werden nicht weiter traversiert). Nachdem ein Blattknoten verarbeitet wurde, oder wenn bei der Verarbeitung innerer Knoten kein Treffer mehr gefunden wird, setzt das Backtracking ein. Wenn der Bitstack 0 ist, terminiert das Verfahren an dieser Stelle. Andernfalls wird iterativ jeweils das least

significant Bit des Bitstacks betrachtet: Bei einer 0 steigt der aktuelle Knoten zu seinem Elternknoten auf, und das betrachtete Bit wird vom Bitstack entfernt (Pop), um das nächste zu verarbeiten. Wenn auf dem Bitstack eine 1 erreicht wird, endet der Backtracking Prozess, der Geschwisterknoten des aktuellen Knotens wird zum Ausgangspunkt der weiteren Traversierung, und das Bit wird auf 0 gesetzt (Bit Flip durch bitweise XOR).

Das Verfahren wird mit dem ähnlichen Verfahren (für „Sparse Trees“) in (Barringer und Akenine-Möller 2013) verglichen. Wie oben angemerkt, wird bei beiden eine Bitmaske verwendet, um das Backtracking zu steuern, aber die Semantik der Bits und damit der Bitmasken unterscheidet sich. Die Vorteile der Methode von Áfra und Szirmay-Kalos sind die frühzeitige Terminierung (wenn der Bitstack 0 ist) gegenüber dem Backtracking bis zur Wurzel, ein „Bit Flip“ statt dem Inkrementieren der Bitmaske, was bei Einsatz eines 64 Bit Integer auf manchen Architekturen effizienter ist und die Möglichkeit der Skalierung auf ein höheres Verzweigungsverhältnis der Bäume, indem statt einem Bit drei Bits für die „Skip Codes“ eingesetzt werden. Die Autoren stellten fest, dass ihr Verfahren auf der GPU je nach Szene zwischen einem und elf Prozent schneller läuft.

Für MBVHs mit vier Kindknoten werden pro Knoten acht Indizes bzw. Pointer benötigt, vier für die Kinder, drei zu den Geschwistern und einer zum Elternknoten. Das Traversierungsverfahren wird erweitert, indem für jeden Geschwisterknoten durch jeweils ein Bit angezeigt wird, ob dieser noch zu traversieren ist oder nicht. Die Semantik bleibt dieselbe. Im Speicher müssen Knoten mit weniger als vier Kindern durch Padding mit leeren Knoten ergänzt werden, damit jeder Knoten drei zugreifbare Geschwisterknoten hat. Für die Berechnung des „Skip Code“ wird der Index des nächsten Kindknoten und eine vierstellige Bitmaske verwendet, welche für die vier Knoten angibt, ob ein Treffer (1) vorhanden ist. Das Bit für den nächsten Knoten wird entfernt, und die drei verbliebenen Bits werden so angeordnet, dass die Position der Ordnung der Knoten-Indizes entspricht. Da diese Variante für den Einsatz auf CPU und MIC beschrieben wird, sei für eine vollständige Beschreibung auf (Áfra und Szirmay-Kalos 2014) verwiesen.

Die GPU Implementierung basiert auf der *while-while* Schleifenorganisation nach (Aila und Laine 2009). Áfra und Szirmay-Kalos sprechen von einer „*while-if-while*“ Organisation, wobei das „*if*“ sich auf die Überprüfung des Knotentyps bezieht. Wenn es sich um einen Blattknoten handelt, wird dieser vollständig verarbeitet (alle Primitive im Blattknoten auf Schnittpunkt testen), so dass es im Sinne der Unterscheidung nach (Aila und Laine 2009) eher auf eine *while-while-while* Organisation hinausläuft. Das letzte *while* steht dabei für den Backtracking Prozess. Als Bitstack wird eine 64 Bit integer Variable verwendet. Für die Bewertung wird ein hoch optimierter Path Tracer (nur diffuse Interreflexion) eingesetzt, und die binären MBVHs wurden auf Basis des Verfahrens von (Stich et al. 2009) aufgebaut, mit 8 Dreiecken in den Blättern. Die GPU Implementierung (Nvidia Kepler GK110 Architektur) mit

binärem MBVH ist verglichen zur Stack-basierten Traversierung laut den Autoren bis zu 31% langsamer, was auf die komplexere Traversierungslogik und niedrigere SIMD Effizienz durch sofortige Verarbeitung von besuchten Blattknoten zurückgeführt wird. Die Größe des Traversierungszustands ist ungefähr bis zu 50 Mal kleiner als bei Einsatz eines Stacks.

3.4 Zum Vergleich von BVHs und kd-Trees

Der Vergleich zwischen BVHs und kd-Trees hinsichtlich der Ray Tracing Performanz zeigt in der Literatur je nach der zugrunde liegenden Architektur (CPU, GPU, spezifische GPU-Generation), den Testszenen, den Verfahren für Aufbau und Traversierung und der Kohärenz der Strahlen, Ergebnisse, die mal für kd-Bäume und mal für BVHs besser ausfallen (z.B. Havran 2000, Günther et al. 2007, Wald, Boulos et al. 2007, Stich et al. 2009, Zlatuška und Havran 2010). Im Folgenden wird auf zwei aktuellere Arbeiten mit Fokus auf der vergleichenden Betrachtung der Beschleunigungsdatenstrukturen eingegangen. Beide Arbeiten verwenden verschiedene GPUs, basierend auf Nvidias Kepler Architektur.

In (Santos et al. 2014) werden BVHs (SBVH, SAH-basiert, Median Cut), kd-Trees (Ropes, SAH, Median Cut), Octrees, Uniform Grids und Bounding Intervall Hierarchies und deren Strahlen-Traversierung auf der GPU (GTX780 Ti) (implementiert in CUDA) für vier Testszenen verglichen. Dafür wurden die Szenen jeweils mit Whitted-style Ray Tracing und Ambient Occlusion gerendert. Die Autoren kommen zu dem Ergebnis, dass kd-Trees und BVHs am schnellsten zu traversieren sind, wobei SAH kd-Trees mit „Ropes“ (in Blättern werden für jede der sechs Seiten zusätzliche Verbindungen zu den nahegelegenen anderen Knoten gespeichert, so dass viele Traversierungsschritte eingespart werden können (Havran et al. 1998)) bei den Testszenen am besten abschneidet. Gleichzeitig benötigt diese Variante im Vergleich zu BVHs in den meisten Fällen ein Vielfaches an Speicher. Unter den gezeigten Szenen stellt insbesondere die *Crytec Sponza* Szene für BVHs durch die ungleichmäßige Verteilung der Geometrie eine Herausforderung dar, während sich eine gleichmäßige Verteilung der Geometrie für BVHs vorteilhaft auswirkt. Für die BVH-Traversierung wird ein Stack-basiertes Verfahren eingesetzt. Für kd-Trees mit Ropes wird die Traversierung anhand der Ropes ohne Stack ausgeführt, bei kd-Trees ohne Ropes wird ein Stack eingesetzt. Für drei der vier Szenen liegt SBVH hinter dem SAH kd-Tree mit Ropes auf dem zweiten Platz, die SAH BVH ist für zwei Szenen vor dem SAH kd-Tree (ohne Ropes) auf dem dritten Platz, für die anderen beiden Szenen hinter kd-Trees auf dem fünften bzw. sechsten Platz. Santos et al. kommen zu dem Schluss, dass ein auf Ropes basierter kd-Tree bis zu 71% schneller sein kann als „BVHs“. Dazu sei angemerkt, dass die 71% aus dem Vergleich zwischen BVH aus Median Cut und kd-Tree aus Median Cut mit

Ropes resultieren. Davon abgesehen dokumentieren die Testdaten je nach Szene für Whitted Ray Tracing im Vergleich zur Referenz (SAH kd-Tree mit Ropes) für die SAH BVH bis zu 30% weniger Performanz und für SBVH bis zu 23%, wobei ein SAH kd-Tree mit Ropes in den gezeigten Testdaten ungefähr bis zu 10 Mal (SAH BVH) bzw. bis zu knapp 9 Mal (SBVH) so viel Speicher benötigt. Eine Ausnahme stellt die *Crytec Sponza* Szene dar, bei der SBVH einen um den Faktor (rund) 1,24 höheren Speicherbedarf aufweist.

Der Vergleich in (Vinkler et al. 2014) betrachtet SAH-basierte kd-Trees und BVHs hinsichtlich der Ray Tracing Performanz auf der GPU (GTX680). Auch der Aufbau der Hierarchien findet auf der GPU statt. Anders als in (Santos et al. 2014) werden weder kd-Tree Varianten mit Ropes noch SBVH betrachtet, sondern jeweils SAH-basierte Hierarchien mit hoher Qualität aufgebaut und nach Möglichkeit ähnliche Parameter für Aufbau und Traversierung eingesetzt. Beide Traversierungsverfahren verwenden die *while-while* Schleifenorganisation und einen Stack. Die Speicherorganisation ist einfach gehalten. Die Leistung wird für Primärstrahlen und für diffuse Strahlen in 16 Szenen bewertet. Vinkler et al. kommen zu dem Ergebnis, dass BVHs für einfache und moderat komplexe Szenen eine höhere Performanz erreichen als kd-Trees, während letztere für komplexe Szenen besser abschneiden. Vor allem die Überschneidungen der BVs in solchen Szenen führen bei BVHs zu einem Overhead bei der Traversierung. Bei kd-Trees stellen die Autoren fest, dass für manche Szenen deutlich weniger Traversierungsschritte und Primitiv-Schnittests durchgeführt werden als bei BVHs, was nicht immer zu einer schnelleren Traversierung führt, vor allem in großen, komplexen Szenen jedoch positive Auswirkungen auf die Performanz hat. Bei Szenen, in denen die Anzahl für beide Hierarchien ähnlich ausfällt, sind hingegen BVHs deutlich schneller. Anhand einer Szene, bei der die Anzahl für beide Hierarchien ungefähr gleich ist, führen Vinkler et al. einige Unterschiede auf, die zu diesem Ergebnis beitragen. Die kd-Tree Traversierung führt zwar weniger Floating Point Operationen durch als die BVH Traversierung, aber der prozentuale Anteil divergenter Verzweigungen ist beim kd-Tree höher, sowie auch die Anzahl an Speicherzugriffen. Auf dem Stack werden beim kd-Tree für einen Knoten 8 Byte abgelegt (Knoten und Distanz entlang des Strahls), bei der BVH sind es 4 Byte, was insgesamt einen deutlichen Unterschied in der Datenmenge für die Schreibzugriffe auf dem langsamen DRAM ausmacht. Die Menge an Daten, die aus dem Speicher gelesen wird, ist für beide ähnlich (ausgehend von der ähnlichen Anzahl an Traversierungsschritten und Primitiv-Schnitttests). Bei einer großen, komplexen Szene, in welcher die Traversierung des kd-Trees deutlich weniger Traversierungsschritte und Schnittests durchführt als für die BVH, fällt die Menge an gelesenem Speicher hingegen für BVHs deutlich größer aus, während die anderen Verhältnisse ungefähr gleich bleiben. Die Leistung ist in diesem Fall beim kd-Tree besser.

Zwei der Testszenen haben beide Arbeiten gemeinsam, *Crytec Sponza* und *San Miguel*. Die erhobenen Daten für SAH BVHs und SAH kd-Trees sind jedoch nicht direkt vergleichbar, da neben der Auflösung auch die gerenderten Ansichten auf die Szenen unterschiedlich sind. Zudem verwenden (Santos et al. 2014) eine stärkere GPU und führen Ray Tracing nach Whitted aus, während in (Vinkler et al. 2014) Daten für Primärstrahlen erhoben werden. Die Tendenz, dass bei der BVH Traversierung mehr Schritte und Primitiv-Schnitttests durchgeführt werden, ist bei Santos et al. (für Whitte Ray Tracing) weniger deutlich und nicht durchgehend ausgeprägt. Die Leistung in Millionen Strahlen pro Sekunde fällt bei Santos et al. höher aus, wobei ein Durchschnittswert eines Walkthrough mit 100 Frames angegeben wird. Bei Vinkler sind es Durchschnittswerte über 4 Ansichten der Szene. Bei Santos et al. ist der SAH kd-Tree für beide Szenen schneller, wobei der Unterschied in der *Sponza* Szene deutlicher ausfällt. Bei Vinkler et al. ist hingegen die SAH BVH in beiden Szenen etwas schneller (Primärstrahlen).

4 Konzept und Implementierung eines GPU Ray Tracers mit BVH

Der GPU Ray Tracer soll klassisches Ray Tracing (nach Whitted) durchführen, also harte Schatten, spiegelnde Reflexionen und Refraktion darstellen. Als geometrische Primitive werden Dreiecke eingesetzt, auf Hinzunahme anderer Primitivtypen wird verzichtet, um die erhöhte Divergenz durch die notwendige unterschiedliche Behandlung zu vermeiden. Somit soll der GPU Ray Tracer statische Szenen aus Dreiecken rendern. Während die meisten modernen GPU Ray Tracer aus Forschungsarbeiten in CUDA implementiert sind, wird hier GLSL im Rahmen von OpenGL Compute Shader Programmen verwendet (siehe Abschnitt 2.5). Zur Beschleunigung des Ray Tracers soll eine Bounding Volume Hierarchie zum Einsatz kommen. Dementsprechend müssen Verfahren zum Aufbau einer BVH und zur Traversierung implementiert werden. Die Traversierung als Teil des Ray Tracing Prozesses muss auf der GPU stattfinden, der Aufbau der BVH kann als Vorverarbeitungsschritt auf der CPU durchgeführt werden. Ein Hierarchie-Update zwischen einzelnen Frames ist nicht vorgesehen, da nur statische Szenen dargestellt werden sollen. Auf Antialiasing und Texturierung wird verzichtet, eine spätere Erweiterung diesbezüglich ist möglich. Die Programmteile für den Host werden im CVK-Framework der Arbeitsgruppe Computergraphik (Universität Koblenz Landau) integriert. Das CVK-Framework wird auch für das initiale Laden der Szenen verwendet (intern wird „Assimp“ eingesetzt), bevor die Szenendaten für die Verwendung im Ray Tracer weiterverarbeitet werden. Als Entwicklungsumgebung wird Microsoft Visual Studio 2015 eingesetzt.

Die Entwicklung wird in die folgenden drei Meilensteine aufgeteilt:

- GPU Ray Tracer (ohne BVH).
  - Konzept für die Strukturierung.
  - Implementieren des grundlegenden Ray Tracers.
- BVH Builder für den Aufbau von BVHs.
  - Auswahl des Aufbauverfahrens.
  - Implementieren des Aufbauverfahrens und Anpassungen für GPU Ray Tracer.
- GPU Ray Tracer mit integrierter BVH Traversierung.
  - Auswahl von Traversierungsverfahren.
  - Implementieren der Traversierungsverfahren und Anpassung des GPU Ray Tracers

In den folgenden Beschreibungen werden, angelehnt an GPGPU Programmiermodelle (wie in CUDA und OpenCL), die Begriffe *Host*, *Device* und *Kernel* verwendet. Die CPU wird hier auch als Host bezeichnet, dementsprechend ist ein „Host Programm“ ein Programm, dass auf der CPU ausgeführt wird (in C++ programmiert). Device steht hier für die GPU, und die Bezeichnung „Device Programm“ wird analog zu „Shader“ bzw. „Shader Programm“ für ein Programm verwendet, das auf der GPU ausgeführt wird. Ein Kernel ist hier ein Shader bzw. Device Programm, das vom Host aufgerufen und auf der GPU ausgeführt wird.

4.1 Strukturierung des GPU Ray Tracers

Wie in Abschnitt 3.1 beschrieben hat man bereits verschiedene Ansätze für die Strukturierung eines GPU Ray Tracers (bzw. Path Tracers) angewendet. Neben der Implementierung in einem einzigen (Mega)Kernel wurden Aufteilungen auf mehrere Kernel vorgestellt. Aufgrund der Beschaffenheit der GPU Architektur und des SIMT Ausführungsmodells (siehe Abschnitt 2.4) sowie der Erkenntnisse in (Laine et al. 2013), erscheint eine Aufteilung auf mehrere Kernel unter Berücksichtigung der Datenparallelität, des Kontrollflusses sowie der Ressourcennutzung vorteilhaft. In (Santos et al. 2014) hingegen, wird aufgrund der zusätzlichen Kosten durch Speicherzugriffe zwischen Kernel Aufrufen eine Lösung mit nur einem Kernel gewählt. Beide Arbeiten verwenden CUDA. Bei der Arbeit von Laine et al. ist zu berücksichtigen, dass komplexe Materialen eingesetzt werden, welche durch ihren hohen Berechnungsaufwand jeweils in einem eigenen Kernel implementiert sind. Darüber hinaus erfolgt aber auch eine Unterteilung für den gesamten

Path Tracer (nicht nur für die Auswertung der Materialien), und die Motivation für die Unterteilung betrifft vor allem Aspekte der zugrunde liegenden Hardware Architektur, wie auch in (Garanzha und Loop 2010). Dementsprechend wird in der vorliegenden Arbeit die Hypothese aufgestellt, dass auch im Rahmen eines klassischen Ray Tracers ein Gewinn an Performanz durch eine geeignete Unterteilung auf mehrere Kernel erzielt werden kann. Um dies zu prüfen wird der GPU Ray Tracer sowohl in unterteilter Form als auch in einem einzigen Kernel implementiert und verglichen.

Die Aufteilung des GPU Ray Tracers erfolgt durch Implementierung in vier Kernel (Compute Shader Programme), wie in Abbildung 9 dargestellt. Der *Ray Generation* Kernel erzeugt ausgehend von den Kameraparametern

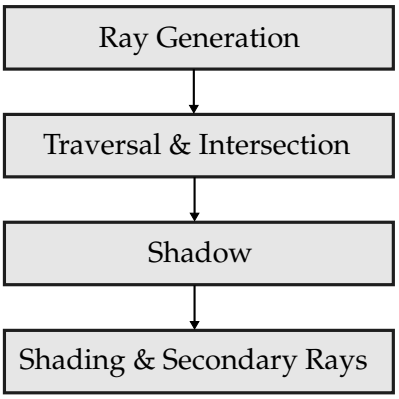

Abbildung 9: Unterteilung des GPU Ray Tracers auf vier Kernel: *Ray Generation* für das Erstellen und Speichern der Primärstrahlen, *Traversal & Intersection* für die Traversierung der BVH und Schnitttest mit Dreiecken, *Shadow* für Traversierung und Schnitttest mit Schattenstrahlen, *Shading & Secondar Rays* für Beleuchtung und Erstellen (und Verarbeiten) von Sekundärstrahlen für Reflexion und ggf. Refraktion.

die Primärstrahlen und speichert diese im Ray Buffer (als SSBO implementiert). *Traversal & Intersection* liest die Strahlen aus dem Ray Buffer, traversiert die BVH und testet ggf. den Strahl auf Schnittpunkt mit Dreiecken in einem getroffenen Blattknoten. Die Schnittpunkte, Distanz zum Schnittpunkt, Normale und der Index des Dreiecks (*HitData*) werden in den Hit Buffer (SSBO) geschrieben. *Shadow* liest die Schnittpunkte aus, berechnet und traversiert für jede Lichtquelle einen Schattenstrahl und schreibt die resultierenden Shadowflags (0 oder 1) in jeweils ein Bit eines vorzeichenlosen 32 Bit Integers, welcher schließlich im Shadow Buffer (SSBO) gespeichert wird. Diese Bitmaske enthält die Schatteninformation zu allen Lichtquellen für einen Bildpunkt. *Shading & Secondary Rays* berechnet die Beleuchtung für jeden Schnittpunkt anhand der Materialdefinition, erzeugt und verfolgt ggf. Reflexions- und Refraktionsstrahlen und schreibt die akkumulierte Ergebnis-

farbe in eine Bild Textur. In Abschnitt 4.7 wird der Ablauf der Ausführung detailliert beschrieben.

An dieser Stelle sei vorweggenommen, dass für den *Shading* Kernel versucht wurde, die sekundären Strahlen von einem Durchlauf (Bounce) zu speichern und für die weitere Verarbeitung wieder beim *Traversal* Kernel zu starten. Diese Variante lief jedoch deutlich langsamer, als der Ansatz alle verbleibenden Stufen für sekundäre Strahlen direkt im *Shading* Kernel durchzuführen. Der Overhead durch die wiederholte Rückgabe der Kontrolle an den Host über mehrere Kernel, und das vielfache Schreiben der sekundären Strahlen in und Lesen aus dem Speicher zwischen jedem Bounce hat sich in diesem Fall also nicht amortisiert. Das dürfte zum großen Teil daran gelegen haben, dass die Speicherstruktur, welche hier verwendet wurde, um die sekundären Strahlen in einem SSBO zu speichern, die Zuordnung der Strahlen zu Bildpunkten beibehalten hat, und somit große Lücken zwischen Einträgen entstehen können. Der Speicherzugriff erfolgt dann sehr inkohärent. Daher wurde die Variante mit Rückgabe der Kontrolle zum *Traversal* Kernel frühzeitig verworfen. Eine mögliche Lösung für dieses Problem wird im Abschnitt 6 angesprochen. Für Primärstrahlen, Schatten und die Trennung zu *Shading & Secondary Rays* wird die Aufteilung wie in Abbildung 9 beibehalten, und deren Effekt wird in den Ergebnissen in Abschnitt 5 dargestellt. Die weitere Beschreibung basiert auf dieser Aufteilung in vier Kernel, ohne Rücksprünge. Für die Vereinigung aller Stufen in einem einzigen Kernel werden alle Teile entsprechend in einem Compute Shader Programm zusammengefasst, es erfolgt dann nur ein Dispatch und keine Zwischenspeicherung von Strahlen oder Schnittpunkten.

4.2 Repräsentation der Daten

Die Daten für Dreiecke, Materialien, Lichtquellen, Strahlen, Schnittpunkte und AABBs werden in Strukturen organisiert, da sie in dieser Form direkt in SSBOs und somit im Compute Shader verwendet werden können, wohingegen die Definition durch Klassen in diesem Fall keinen Mehrwert bringen würde. Daher werden die Daten im Host Programm in der Regel nicht in Form von Klassenobjekten implementiert. Für bessere Lesbarkeit werden die Daten in Arrays von Strukturen organisiert, wobei angemerkt sei, dass durch Strukturen von Arrays eine effizientere Organisation und Verarbeitung möglich sein kann, wie z.B. in (Laine et al. 2013) festgestellt (siehe auch Abschnitt 3.1). Im Compute Shader sind die SSBOs in Form von Shader Storage Blocks nutzbar, wobei als Speicher-Layout *std430* verwendet wird. Dementsprechend werden Komponenten innerhalb der Strukturen nach Möglichkeit so angeordnet, dass kein Padding notwendig ist. Wenn jedoch nötig, wird durch explizites Padding ergänzt, entweder durch zusätzliche Variablen oder z.B. durch Verwendung von Vektoren mit vier Elementen anstatt drei. Die Struktur *Ray* enthält zum Beispiel für Ursprung und Rich-

tung eines Strahls jeweils einen Vektor mit vier Elementen, wobei im vierten Element ein Flag für den Strahl-Typ bzw. ein Gewichtungsfaktor gespeichert wird. Die Struktur der Daten für BVH Knoten, wird in Abschnitt 4.4 beschrieben.

4.3 Host Programm

Für die Initialisierung und den Aufruf des GPU Ray Tracers wird die Klasse *GPU_RayTracer* entwickelt. Sie bildet auch die Schnittstelle zum CVK-Framework und kapselt die Details für die Erstellung und das Binden aller spezifischen Shader Programme und Buffer Ressourcen sowie für die Ausführung der Ray Tracer Kernel und die Darstellung des berechneten Bildes auf einem bildschirmfüllenden Viereck. Die Klasse setzt *CVK::ShaderSet* ein, um das Erstellen, Laden des Shader Codes, Kompilieren, Binden und Verknüpfen für die Shader Programm Objekte durchzuführen. Für das Laden von Modellen und die Initialisierung der Szene sowie die Definition der Kamera und der Lichtquellen wird zunächst das CVK-Framework im Rahmen des `main` Blocks verwendet. Die Weiterverarbeitung der per CVK-Framework definierten Szene wird dann von *GPU_RayTracer* initiiert, indem der CVK Szenegraph (*CVK::Node*) an die eigene Klasse *GPU_RayTracerScene* übergeben wird. Diese ist zuständig für die Extraktion der Geometrie-Daten, Material-Daten, Lichtquellen und die Berechnung von AABBs für die Dreiecke sowie für die Überführung und Organisation der Daten in die Strukturen und Buffer Objekte (Abschnitt 4.2). Der Aufbau der BVH wird durch die Klasse *BVH_Builder* durchgeführt (Abschnitt 4.4). Der Aufruf der entsprechenden Methoden für Initialisierung und Vorverarbeitung (innerhalb des `main` Blocks der Anwendung) ist in Listing 11 aufgeführt.

Listing 11: Initialisierung und Vorverarbeitungsschritte für den GPU Ray Tracer

```
GPU_RayTracer gpu_rt = GPU_RayTracer(WIDTH, HEIGHT);
gpu_rt.init();  //initialisiere Shader Programme und Buffer
//Extraction der Szene-Daten und Organisation in Buffer
gpu_rt.transferSceneData(scene_node);
gpu_rt.bindRessources(); //Binde Buffer Objekte & Bild Textur
//BVH Aufbau und Überführung in Buffer
BVH_Builder bvhBuilder(sceneAABB, centroidBounds, triAABBs);
bvhBuilder.build();
bvhBuilder.bvhToSSBO();
//bvhBuilder.mbvhToSSBO(); //alternativ MBVH Oganisation
bvhBuilder.bvhBindRessources();
```

Der Aufruf der Ray Tracing Kernel wird innerhalb der Rendering Schleife durch `gpu_rt.trace_m(camPos, camView, projMat)` ausgeführt (`trace(...)` für die Variante mit nur einem Kernel), wobei die drei Parameter für Pointer auf die Kameraposition, die View Matrix und die Projektionsmatrix stehen

und für Vorberechnungen von Werten zur initialen Strahlen-Erstellung verwendet werden.

4.4 BVH Aufbau

Für den Aufbau einer binären BVH wird das (sequenzielle) Binned BVH Verfahren nach (Wald 2007) implementiert, welches in Abschnitt 3.2 ausführlich dargestellt ist. Das Verfahren wurde gewählt, da eine hohe Qualität (bezüglich SAH) der resultierenden Hierarchie erreicht wird, und die benötigte Zeit für den Aufbau gleichzeitig relativ gering ausfällt. Eine vereinfachte Darstellung der beteiligten Klassen ist in Abbildung 10 zu sehen. Wie bei

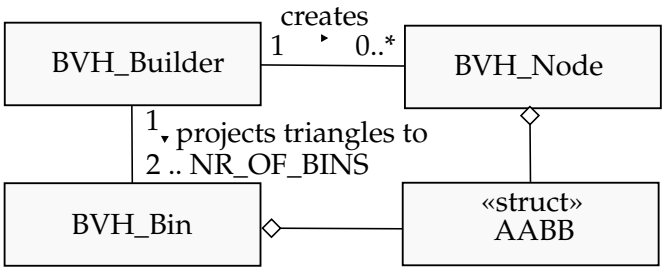

Abbildung 10: Vereinfachte Darstellung der Klassen für den BVH Aufbau.

Wald werden 16 Bins verwendet. Ein Versuch mit 32 Bins zeigte praktisch keine Verbesserung der Qualität. Die Bins werden entlang derjenigen Achse verteilt, für welche die Centroid Bounds des zu unterteilenden Knotens die größte Ausdehnung aufweisen. Zusätzlich wird eine Variante implementiert, in der die lokal beste Unterteilung für alle drei Achsen gesucht wird. Eine Anzahl von vier Dreiecken dient als Schwellenwert für die Erstellung eines Blattknotens. Durch die anderen Terminierungskriterien (SAH Kosten für den unterteilten Knoten fallen größer aus als für den Knoten vor Unterteilung oder Centroid Bounds sind zu klein für eine sinnvolle Unterteilung) können sich bei besonders fein tesselierten Bereichen auch mehr Dreiecke in einem Blattknoten befinden. Das Ergebnis aus dem Aufbauprozess ist ein Array mit BVH Knoten und ein Array mit entsprechend geordneten Indizes aller Dreiecke (Dreiecksliste). Die BVH Knoten werden für die Verwendung im Compute Shader letztendlich als Struktur organisiert und beinhalten zunächst jeweils folgende Komponenten: Die minimale und maximale Position als `vec4`, wobei das vierte Element jeweils ein Offset (Start bzw. Ende des Intervalls der enthaltenen Dreiecke) in die Dreiecksliste speichert. Darüber hinaus wird die ID des Knotens, der Typ (innerer Knoten oder Blatt), und die Indizes zu den Kindknoten, dem Elternknoten und dem Geschwisterknoten gespeichert. Inklusive Padding benötigt ein Knoten 64 Byte. Die Klasse *BVH_Builder* hat zudem Methoden für die Überführung des BVH in das Buffer Objekt (SSBO) und für die Liniendarstellung der Volumen

in OpenGL. Die Optimierung wurde mit Hilfe des Visual Studio Profilers durchgeführt. Der Aufruf der *BVH_Builder* Methoden für den Aufbau und die Überführung auf die GPU sind in Listing 11 aufgeführt.

Die SAH Kosten für den gesamten Baum, ohne die konstanten Kosten, werden nach dem Aufbau berechnet durch Gleichung 26 (nach MacDonald und Booth 1990, Aila, Karras et al. 2013). Dabei ist $S(root)$ die Oberflächengröße des Wurzel BVs, n ist ein Knoten, I ist die Menge der inneren Knoten, L die Menge der Blattknoten, $N(n)$ ist die Anzahl der Dreiecke in Knoten n und $S(n)$ ist die Oberflächengröße des Knotens. Die Oberflächengröße der AABBs wird dabei nach Gleichung 14 (siehe Abschnitt 2.3.1) angenähert.

$$SAH_{tree} = 1/S(root) \left(\sum_{n \in I} S(n) + \sum_{n \in L} N(n) \cdot S(n) \right) \tag{26}$$

Die für die Knoten gespeicherten Daten erlauben verschiedene BVH Repräsentationen im Speicher. Die hier verwendete „klassische“ BVH Variante verwendet die oben beschriebene Struktur. Darüber hinaus ist die Organisation als MBVH (M. Ernst und G. Greiner 2008) möglich, wobei hier für eine binäre MBVH in einem inneren Knoten die zwei BVs der beiden Kindknoten gespeichert werden, um die Lesezugriffe bei der Traversierung kohärent zu gestalten und die Anzahl an Zugriffen zu verringern. Dafür wird eine platzsparende Repräsentation auf Basis von (Aila und Laine 2009) in 64 Byte erreicht, indem die x und y Komponenten der minimalen und maximalen Position eines BVs zusammen in jeweils einem `vec4` pro Kindknoten gespeichert werden und die z Komponenten beider BVs zusammen in einem einzigen `vec4`. Die Ausdehnung des inneren Knotens selbst wird dann nicht benötigt. Zusätzlich werden die Indizes zu den Kindknoten, dem Elternknoten und dem Geschwisterknoten gespeichert. Eine Konvertierung in die MBVH Struktur und die Überführung in den Buffer kann ebenfalls durch *BVH_Builder* durchgeführt werden.

4.5 BVH Traversierung

Die BVH Traversierung wird an mehreren Stellen benötigt. Für Primärstrahlen im *Traversal* Kernel, für Schattenstrahlen in angepasster Form im *Shadow* Kernel und für Sekundärstrahlen aus Reflexion und Refraktion (und ggf. deren Schattenstrahlen) im *Shading* Kernel. Eine Stackless Traversierung (für binäre BVH) wurde auf Basis des Verfahrens nach (Áfra und Szirmay-Kalos 2014) (ausführlich in Abschnitt 3.3 beschrieben) implementiert. Das Verfahren wurde ausgewählt, da es für die BVH Einzelstrahl-Traversierung ohne Stack das schnellste veröffentlichte (Stand Anfang 2016) Verfahren zu sein scheint: Die Verfahren in (Laine 2010) und (Hapala et al. 2011) wurden von (Barringer und Akenine-Möller 2013) mit ihrem eigenen Verfahren verglichen und sind langsamer (außer Laine mit Short Stack für 8 Elemente). Die

Methode von Áfra und Szirmay-Kalos ist im Vergleich zu Barringet et al. wiederum schneller. Erst kürzlich wurde von Binder und Keller ein Verfahren vorgestellt, das noch performanter ist (Binder und Keller 2016). Leider ist die Arbeit erst relativ spät im Verlauf des Verfassens und Entwickelns der vorliegenden Arbeit veröffentlicht worden und wurde erst später gefunden, daher konnte es zeitlich nicht mehr mit einbezogen werden.

Áfra und Szirmay-Kalos verwenden als Ersatz für einen Stack ein 64 Bit Integer als „Bitstack“ und haben dadurch einen, um ein Vielfaches kleineren Traversierungsstatus, was für hochparallele Architekturen vorteilhaft ist. Sie stellten dennoch fest, dass die Stackless Traversierung langsamer ist als eine hoch optimierte Traversierung mit Stack. Ob das für die hier in GLSL implementierte Variante auf einer aktuelleren GPU Architektur ebenfalls zutrifft soll überprüft werden. Dafür wird zum Vergleich zusätzlich eine einfache Stack-basierte Traversierung umgesetzt. Darüber hinaus werden die Traversierungsmethoden, welche eine *while-while* Schleifenorganisationen haben, zum Vergleich auch in einer *if-if* Struktur (Abschnitt 3.3 und Aila und Laine 2009) realisiert, um festzustellen, ob sich bei einer der Varianten Vorteile ergeben. Ausgangspunkt für die grundlegenden Implementierungen und die folgenden Beschreibungen ist zunächst die *while-while* Struktur, wenn nicht anders angegeben.

Damit für die Implementierung der Stackless Traversierung in GLSL ein 64 Bit Integer als Bitstack verwendet werden kann, kommt die Erweiterung *GL_NV_gpu_shader5* zum Einsatz. Der Baum wird depth-first durchlaufen und gemäß der *while-while* Struktur werden zunächst die inneren Knoten traversiert, bis alle Strahlen eines Warps entweder einen Blattknoten gefunden haben oder terminieren. Dann prüft jeder Strahl, der einen Blattknoten getroffen hat, alle enthaltenen Dreiecke auf Schnittpunkt. Danach erfolgt das Backtracking über den Bitstack in einer weiteren Schleife oder ggf. die Terminierung, wenn der Bitstack null ist. Bei einem Schnittpunkt mit einem Dreieck wird der Strahl auf den Abstand gekürzt, so dass weiter entfernte Teilbäume nicht mehr traversiert werden müssen. Anders als bei Áfra und Szirmay-Kalos werden die Knoten aus dem globalen Speicher hier nicht über den read-only Textur Cache der GPU geladen (per CUDA *__ldg()* intrinsic möglich, ohne tatsächlich Texturen zu verwenden). Stattdessen wird der aktuelle Knoten (64 Byte) zu Beginn eines Traversierungsschritts aus dem Buffer in lokale Variablen geladen und verarbeitet. Für die klassische BVH Variante müssen zusätzlich über die Kindknoten die Grenzen der beiden Kind-BVs (nochmals 64 Byte) aus dem Speicher geholt werden. Bei Einsatz des MBVH sind diese Teil des Knotens, und es ist nur ein Lesezugriff auf 64 Byte notwendig. Wenn möglich werden statt Branching bedingte Zuweisungen eingesetzt, was kleine Verbesserungen der Performanz bringt. Das zeigt, dass die automatische Optimierung durch den GLSL Kompiler auch für einfache Verzweigungen nicht immer greift, und eine manuelle Umsetzung hilfreich sein kann. Da die Speicherorganisation der BVH, wie in

Abschnitt 4.4 beschrieben, in zwei Varianten realisiert wird, werden auch die Traversierungs-Methoden für das klassische BVH Layout und das MBVH Layout implementiert und über Präprozessor Direktiven eingesetzt.

Für die Stack-basierte Traversierung wird ebenfalls depth-first vorgegangen, wobei eine Variante mit Stack für 32 Elemente und eine mit größerem Stack für 64 Elemente betrachtet wird. Werden zwei benachbarte Kindknoten getroffen, setzt die Traversierung zunächst beim Teilbaum des näheren fort und die ID des weiter entfernten wird auf den Stack gelegt. Wurde ein Blattknoten verarbeitet oder kein Kind mehr getroffen, geht es mit dem ersten Element auf dem Stack weiter, oder das Verfahren terminiert, wenn der Stack leer ist. Es zeigt sich, dass die Performanz der Stack-basierten Traversierung mit der festgelegten Stack Größe zusammenhängt (weiteres in Abschnitt 5).

Die angepasste Traversierung für Schattenstrahlen terminiert zusätzlich schon beim ersten gefundenen Schnittpunkt mit einem Dreieck oder bei Überschreiten der Distanz zur Lichtquelle. Für die Traversierung durch Schattenstrahlen wird zudem eine weitere Möglichkeit der Schleifenorganisation getestet, die nicht in (Aila und Laine 2009) behandelt wurde. Die Idee ist es, die Verarbeitung der Dreiecke von ggf. getroffenen Blattknoten zu priorisieren, indem eine *if-while* Struktur eingesetzt wird, da die Traversierung beim ersten gefundenen Schnitt mit einem Dreieck, unabhängig vom Abstand, bereits terminiert. In jeder Iteration werden also innere Knoten getestet bis Strahlen aus dem Warp Blattknoten treffen. Die Verarbeitung aller Dreieck in getroffenen Blattknoten hat dann Priorität, so dass ggf. Schattenstrahlen frühzeitig terminieren können, wenn sie treffen oder kein Eintrag auf dem Bitstack ist, bevor die Traversierung von inneren Knoten durch die übrigen Strahlen fortgesetzt wird. So ergeben sich insgesamt zehn Varianten der Traversierung, die untersucht werden.

4.6 Beleuchtung

Als BRDF wird ein modifiziertes, normalisiertes Blinn-Phong Modell eingesetzt, ergänzt um die Schlick Approximation des Fresnel Reflexiongrades in RGB (Gleichung 27 nach Möller, Haines et al. 2008, Kapitel 7.6).

$$f(\vec{l}, \vec{v}) = \vec{c}_{diff} + \frac{m+8}{8\pi} \cdot \vec{c}_{spec}(\alpha_h) \cdot (cos\theta_h)^m \tag{27}$$

Dabei ist $\vec{l}$ die Lichtrichtung, $\vec{v}$ die Richtung zum Betrachter, $\vec{c}_{diff}$ die diffuse Farbe, $\vec{c}_{spec}(\alpha_h)$ ist der Fresnel Reflexionsgrad (bzw. Reflexionsfarbe) für den Winkel α_h zwischen Halbvektor $\vec{h} = \vec{l} + \vec{v}/|\vec{l} + \vec{v}|$ und $\vec{l}$ und m ist der Glanzexponent bzw. bestimmt der Parameter hier die „Rauheit“ der Oberfläche durch die Erscheinung der Glanzlichter (hohe Werte lassen die Oberfläche durch entsprechende Glanzlichter glatter erscheinen). Durch die Verwendung des Halbvektors ergeben sich Ähnlichkeiten zu

einem Mikrofacetten-Modell, so dass der Kosinusterm als NDF (Normal Distribution Function) interpretiert werden kann(Möller, Haines et al. 2008).

Die Schlick Approximation eines Fresnel Reflexionsgrades, ausgehend von der im Material definierten charakteristischen Reflexionsfarbe, kann berechnet werden als

$$\vec{c}_{spec}(\psi) \approx \vec{c}_{spec}(0°) + (1 - \vec{c}_{spec}(0°))(1 - cos\psi)^5, \tag{28}$$

wobei $\vec{c}_{spec}(\psi)$ die angenäherte Fresnel Reflexionsfarbe für Einfallswinkel ψ ist und $\vec{c}_{spec}(0°)$ für die charakteristische Reflexionsfarbe oder Glanzfarbe des Materials steht (Möller, Haines et al. 2008, Kapitel 7.5). Zudem wird anhand des Reflexions-Koeffizienten *ks* per Schlick Approximation ein Faktor für die Gewichtung von Beiträgen der Reflexionsstrahlen berechnet, der abhängig vom Einfallswinkel ist.

Der diffuse Term $\vec{c}_{diff}$ wird hier berechnet durch Gleichung 29.

$$\vec{c}_{diff} = kd \cdot \vec{mat}_{diff} \cdot cos\phi \tag{29}$$

Dabei ist kd der diffuse Reflexionskoeffizient, $\vec{mat}_{diff}$ die definierte diffuse Farbe des Materials und φ_i der Winkel zwischen Normale und Lichtrichtung. Das Ergebnis der obigen BRDF wird dann für jede Lichtquelle mit der definierten „Lichtfarbe“ multipliziert und aufaddiert (siehe auch Abschnitt 2.1). Die Berechnungen werden nur durchgeführt, wenn das Shadowflag für die jeweilige Lichtquelle an dem zu beleuchtenden Punkt 1 ist (Licht sichtbar).

4.7 Ablauf der Kernel Ausführung

Sobald das Ray Tracing in der Rendering Schleife vom Host Programm gestartet wurde (siehe oben), werden in *GPU_RayTracer* (`trace_m(...)`) zunächst Werte für die Strahlenberechnung bereitgestellt. Dann erfolgt der Dispatch des *Ray Generation* Kernels mit einer Anzahl von Work Groups, die abhängig ist von der Bildgröße und der Work Group Größe, die im Shader definiert und vorab bereits ausgelesen wurde (Listing 12). Die nächste Zweierpotenz der Bildgröße wird verwendet, damit die Division ein glattes Ergebnis liefert (denn die Work Group Größe wird als ein Vielfaches von 32 definiert, siehe Abschnitt 2.4).

Listing 12: Dispatch des ersten Kernels, analog für weitere Kernel

```
//workSizeX: nächste Zweierpotenz der Bildbreite
//workGroupSizeX: Größe der Work Groups des Kernels für x.
glDispatchCompute(m_workSizeX / workGroupSizeX,
                  m_workSizeY / workGroupSizeY, 1);
```

Die Work Group Größe im *Ray Generation* Kernel ist als 32×32 definiert (1024 Threads, 32 Warps pro Gruppe), da es sich um einen einfachen Kernel

handelt, der nur wenige Registerplätze benötigt. Höhere Werte wirken sich hier negativ auf die Performanz aus. Die Primärstrahlen werden dann für jeden Bildpunkt berechnet und im Ray Buffer (SSBO) gespeichert. Dabei entspricht die Position des jeweiligen Bildpunktes der eindeutigen ID von dem jeweils aktiven Work Item, und der korrespondierende Index eines Strahls wird als `offset` berechnet (Listing 13).

Listing 13: Bildpunkt Position und Strahl-Index

```
vec2 pixel = ivec2(gl_GlobalInvocationID.xy);
vec2 size = imageSize(framebuffer); //Größe Bild Textur
uint offset = pixel.y * size.x + pixel.x; //1D Offset
```

Sobald alle Work Items verarbeitet wurden, terminiert der Kernel, und die Kontrolle geht zurück an den Host, welcher durch eine Barriere sicherstellt, dass alle Strahlen in den Speicher geschrieben wurden, bevor der nächste Kernel aufgerufen wird. Eine solche Barriere wird nach jedem Dispatch gesetzt. Alle weiteren Kernel haben eine experimentell ermittelte Work Group Größe von 8×8 oder 4×16, da so die schnellste Ausführung erreicht wurde. Der *Traversal & Intersection* Kernel wird als Nächstes aufgerufen. Hier werden die Strahlen für das jeweilige Work Item anhand des Offsets ausgelesen und durch die BVH traversiert (siehe Abschnitt 4.5). Für die Schnittpunktsuche mit einem Dreieck kommt das Verfahren aus (Möller und Trumbore 1997) zum Einsatz. Wenn der nächstgelegene Schnittpunkt mit einem Dreieck gefunden wurde, werden die Daten (Schnittpunkt, Normale, Distanz, Dreiecks Index) im *HitData* SSBO gespeichert. Nach Terminierung des Kernels wird der *Shadow* Kernel ausgeführt. Für jeden Schnittpunkt aus dem *HitData* Buffer wird ein Schattenstrahl pro Lichtquelle traversiert und so ein Shadowflag ermittelt. Das Shadowflag kann 1 (Licht ist sichtbar) oder 0 (Licht ist verdeckt) sein. Zur Speicherung des Shadowflags kommt pro Bildpunkt ein 32 Bit Unsigned Integer als Shadowstack zum Einsatz, somit werden bis zu 32 Lichtquellen unterstützt. Ähnlich wie bei der Traversierung mit einem Bitstack (Áfra und Szirmay-Kalos 2014) wird das Shadowflag für jede Lichtquelle durch Bitoperationen, angefangen beim Least Significant Bit, im Shadowstack gespeichert.

Listing 14: Speichern eines 1 Bit Shadowflag für eine Lichtquelle

```
shadowStack <<= 1; //füge ein 0 Bit für Lichtquelle hinzu
//ändere Bit zu 1, wenn Lichtquelle sichtbar
shadowStack = (lVisible == 1) ? shadowStack | 1 : shadowStack;
```

Der 32 Bit Shadowstack für den Bildpunkt wird dann im Shadow Buffer (SSBO) gespeichert, dessen Größe von der Bildauflösung abhängt. Die Traversierung der Schattenstrahlen ist so angepasst, dass sie bereits bei dem ersten Treffer eines Dreiecks terminiert und keine Teilbäume traversiert,

die weiter entfernt sind als die Distanz zur Lichtquelle. Als Letztes wird der *Shading* Kernel ausgeführt, der neben der Beleuchtung auch die Verarbeitung von Sekundärstrahlen für Reflexion und Refraktion durchführt und somit der größte Kernel ist. Zunächst werden die Schnittpunkte der Primärstrahlen ausgelesen sowie die Materialeigenschaften der zugehörigen Dreiecke, die Eigenschaften der Lichtquellen und die Schattenflags aus dem Shadow Buffer. Anhand dieser Parameter wird ein Beleuchtungsmodell angewendet (siehe Abschnitt 4.6), um die Farbe für den zum Schnittpunkt korrespondierenden Bildpunkt zu berechnen. Für Strahlen, die auf reflektierende Materialien treffen, wird ein Reflexionsstrahl berechnet und weiterverfolgt; bei transmittierenden Materialien kommt neben dem Reflexionsstrahl auch ein Refraktionsstrahl hinzu. Pro Bounce (hier Iteration für Reflexions- und Refraktionsstrahlen) können also für jeden auftretenden Schnittpunkt bis zu zwei neue sekundäre Strahlen generiert werden, im Worst Case sind das für n Bounces 2^n Strahlen pro Bildpunkt. Für die iterative Verarbeitung wird eine Queue eingesetzt, um die Strahlen, die während einer Iteration neu entstehen, für die Verarbeitung in der nächsten Iteration zu speichern. Die maximale Größe dieser Queue ist 2^n. Für jeden sekundären Strahl muss die Traversierung und ggf. die Beleuchtung durchgeführt werden. Die resultierende Farbe wird über einen Gewichtungsfaktor, der mit jedem Bounce geringer wird, auf die Farbe des primären Schnittpunktes aufaddiert. Diese Verarbeitung terminiert, wenn entweder eine definierte maximale Anzahl an Bounces (`MAX_DEPTH`) erreicht wird, die Gewichtung der Beiträge unter einen Schwellenwert (`MINWEIGHT`) sinkt oder kein Material mehr getroffen wurde, das transmittiert und/oder reflektiert. Wenn der *Shading* Kernel terminiert, wird durch den Host mit einer Barriere für das `SHADER_IMAGE_ACCESS_BARRIER_BIT` sichergestellt, dass alle Schreibzugriffe auf der Bild Textur abgeschlossen sind, damit diese im nächsten Schritt auf einem Rechteck in Bildgröße dargestellt werden kann.

5 Ergebnisse und Diskussion

5.1 Testsystem

Das System, auf dem die folgenden Daten gesammelt werden, verfügt über eine Intel Core i5-2500K CPU, 8GB RAM, Nvidia GeForce GTX970 GPU mit 4GB GDDR5 DRAM. Als Betriebssystem wird Windows 7 Professional 64 Bit eingesetzt. Die Framerate (Frames Per Second, FPS) und die Zeit pro Frame in Millisekunden werden mit Nvidia Nsight Visual Studio Edition (Version 5.1) ermittelt. Der GPU Treiber ist Nvidia ForceWare 368.69.

5.2 Testszenen und Ergebnisse

Daten werden für vier Testszenen erhoben, deren Konfiguration in Tabelle 2 beschrieben wird. Für Szenen im OBJ-Format wurden die Material-Definitionen in MTL-Dateien manuell angepasst. Die gerenderten Ansichten der Szenen sind in Abbildung 11 dargestellt. Abbildung 12 zeigt zusätzlich für zwei der Szenen die Darstellung der BVHs.

Der Ray Tracer berechnet und verfolgt Primärstrahlen, Schattenstrahlen und zusätzlich zwei Bounces für Sekundärstrahlen (Reflexion und Refraktion). Die Anzahl der jeweiligen verfolgten Strahlen pro Frame ist für jede Szene in Tabelle 3 aufgeführt. Zusammen mit der benötigten Zeit für einen Frame in Millisekunden (ms/F) wird die Summe von Strahlen pro Frame (Rays/F) dazu verwendet, die Anzahl von Strahlen pro Sekunde (MRays/s) zu berechnen (Gleichung 30). Die Differenzierung der gezählten Strahlen pro Frame nach Primär- (nur von der Auflösung abhängig), Schatten- und Sekundärstrahlen gibt Anhaltspunkte für die Situation der Beleuchtung und Reflexion bzw. Refraktion der jeweiligen Szene. Bei *Dragon* ist allerdings nur der Boden reflektierend, und es gibt keinen geschlossenen Raum, daher haben die Sekundärstrahlen in dieser Szene einen geringeren Einfluss auf die Leistung als in den anderen Szenen. In der Regel werden zwei Lichtquellen verwendet, nur die *Mad Science* Szene wird durch drei Lichtquellen beleuchtet.

Die *Dragon* Szene ist bewusst anders aufgebaut als die anderen Szenen, es steht ein großes Objekt im Zentrum, das sehr fein trianguliert ist. *Crytec Sponza* und *Conference Room* (im Folgenden häufig zu *Conference* abgekürzt) zeigen Architektur, und beide haben eine unregelmäßige Aufteilung der Geometrie. Bei *Crytec Sponza* wurde zudem eine Glaskugel (aus Dreiecken) hinzugefügt. *Mad Science* hat viele kleinere Objekte, darunter viele transmittierende und reflektierende aus Glas; die Szene wurde in einen geschlossenen Raum (aus 12 großen Dreiecken) platziert. Die gewählte Auflösung beträgt 1024 $\times$ 768 Bildpunkte. Die Variante, welche gemäß den Ergebnissen in Tabelle 4 die beste Leistung erreicht, wird zudem verwendet, um die Bilder in 1920 $\times$ 1080 Bildpunkten zu rendern (Tabelle 9).

$$MRays/s = \frac{Rays/F}{ms/F} \cdot 1000 \tag{30}$$

Die Beleuchtung erfolgt, wie in Abschnitt 4.6 dargelegt, durch ein modifiziertes Blinn-Phong Modell mit angenähertem Fresnel Reflexionsgrad. In den Ergebnis-Tabellen handelt es sich bei allen Traversierungs-Varianten ohne den Zusatz „Stack“ um die Traversierung ohne Stack (Stackless), zudem sind Varianten mit MBVH Speicherorganisation der BVH mit dem Zusatz „MBVH“ gekennzeichnet. Die anderen Varianten verwenden eine BVH Organisation, die hier in der Beschreibung als „klassisch“ bezeichnet wird (siehe Abschnitt 4.4).

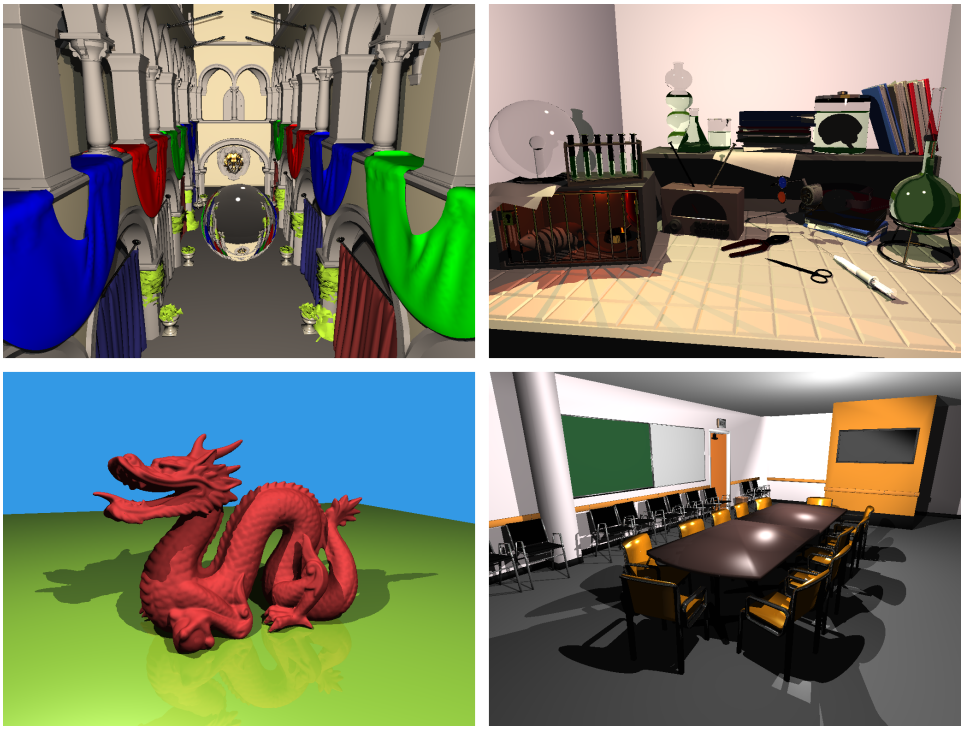

Abbildung 11: Gerenderte Testszenen: *Crytec Sponza*, *Mad Science*, *Stanford Dragon*, *Conference Room*.

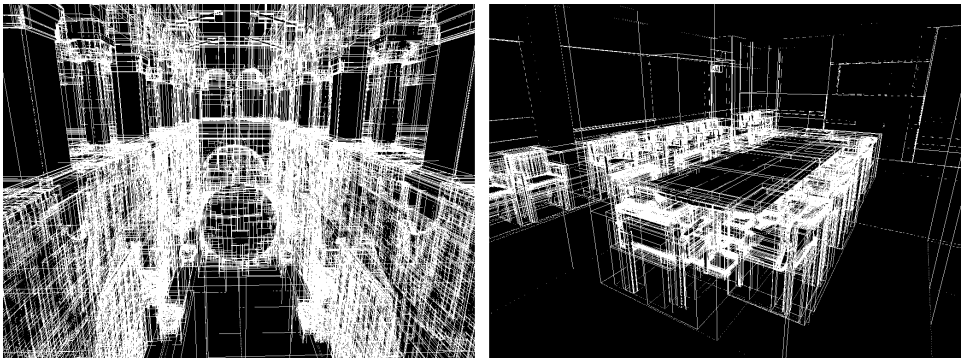

Abbildung 12: Darstellung der BVH für *Crytec Sponza* und *Conference Room*.

Tabelle 2: Konfiguration der Testszenen

Crytec Sponza	von Frank Meinl, Crytec
Dreiecke	262.267 zzgl. 431 (für Kugel)
Reflektierend:	Vorhangstangen, Fahnenstangen, Vasen, Ketten, Löwenkopf
Transmittierend:	Glaskugel vor der Kamera platziert
Lichtquellen:	2
Mad Science	von Dan Konieczka und Giorgio Luciano
Dreiecke	80.137 zzgl. 12 (für geschlossenen Raum um die Szene)
Reflektierend:	Käfigbox u. Gitter, Napf, Radio, Objekte aus Glas
Transmittierend:	viele Objekte aus Glas
Lichtquellen:	3
Stanford Dragon	Stanford University
Dreiecke	871.414 zzgl. 2 (für Boden)
Reflektierend:	Bodenfläche
Transmittierend:	-
Lichtquellen:	2
Conference Room	von Anat Grynberg und Greg Ward
Dreiecke	331.179
Reflektierend:	einige Oberflächen, Stühle, Stuhlgestelle
Transmittierend:	-
Lichtquellen:	2

Tabelle 3: Strahlen pro Frame, ermittelt für Primär- Schatten- und Sekundärstrahlen und deren Summe (Rays/F)

1024×768	**Primary rays/F**	**Shadowrays/F**	**Secondary rays/F**	**Rays/F**
Crytec Sponza	786.432	1.634.942	31.922	2.453.296
Mad Science	786.432	2.838.933	159.879	3.785.244
Dragon	786.432	1.702.746	363.515	2.852.693
Conference	786.432	1.745.818	86.477	2.618.727

In Tabelle 4 werden die Ergebnisse für alle getesteten Traversierungs-Varianten für den aufgeteilten Ray Tracer (vier Kernel) dargestellt. Tabelle 5 zeigt ergänzend vergleichende Ergebnisse zu zwei der Stack-basierten Varianten mit einem größeren Stack. Für die BVHs wurde beim Aufbau die Suche nach der lokal besten Unterteilung auf allen drei Achsen angewendet. Zusätzlich wurden auch Daten mit BVHs erhoben, für die beim Aufbau nur jeweils die Achse verwendet wurde, auf der die Centroid Bounds die größte Ausdehnung haben (siehe Abschnitt 3.2 und 4.4). Die Ergebnisse dazu sind in Tabelle 6 aufgeführt. Tabelle 7 zeigt die SAH Kosten für die BVHs aus beiden Varianten des Aufbauverfahrens. Ergebnisse der Einzelkernel Version des Ray Tracers werden in Tabelle 8 vorgestellt und zu Leistungswerten in Tabelle 4 in Bezug gesetzt.

Die Ergebnisse in Tabelle 4 zeigen, dass die MBVH Speicherorganisation mit *while-while* Traversierung ohne Stack (nach Áfra und Szirmay-Kalos 2014) durchgehend die besten Werte erreicht. Daher wird ***MBVH while-while*** als Referenz bzw. Optimum für relative Leistungswerte der anderen Varianten eingesetzt. Anders als bei (Aila und Laine 2009) ist die *if-if* Variante ohne und mit Stack sowohl für die klassische BVH Organisation als auch für MBVH für drei der vier Szenen am langsamsten. Die Differenz der relativen Leistung von der *if-if* Variante ohne Stack zum Optimum beträgt z.B. 13% bis 17%, im Vergleich zu *while-while* ist sie 5% bis 10% langsamer. Eine Ausnahme bildet die *Dragon* Szene, bei der *if-if* ohne Stack um 1% schneller ist als *while-while* und nur 6% vom Optimum entfernt. Die *if-while* Variante, die eigentlich speziell für die Traversierung von Schattenstrahlen gedacht war (siehe Abschnitt 4.5), ist für die klassische BVH Organisation um 1% bis 4% besser als *while-while* (für *Sponza*, *Mad Science* und *Dragon*). Für *Conference* ergibt sich eine kleine Verschlechterung von -1%. Auch eine Kombination aus *while-while* mit *if-while* für die Schatten (*w-w; i-w; w-w*) führt für die klassische BVH im Vergleich zu *while-while* nur zu kleinen Verbesserungen von 3% für drei der vier Szenen und 1% für *Conference*. Der Vergleich zwischen *while-while* und *MBVH while-while* zeigt, dass die MBVH Organisation hier, je nach Szene, zu einer Beschleunigung um 7% bis 9% führt. Betrachtet man alle Varianten ohne Stack (klassische BVH und MBVH) ist *MBVH while-while* zwischen 5% und 17% schneller als die anderen.

Die Stack-basierten Varianten sind hier durchgehend deutlich langsamer als die Methoden ohne Stack. Am besten unter den Stack-basierten Methoden ist *MBVH Stack(32) w-w*, welche jedoch im Vergleich zur korrespondierenden Variante ohne Stack, *MBVH while-while*, je nach Szene zwischen 53% und 68% langsamer ist. In der Sponza Szene erreichen die Methoden mit Stack relativ zu den anderen Methoden noch die besten Werte, sind aber dennoch über 50% langsamer als die korrespondierenden Stackless Varianten. Tabelle 5 zeigt zudem, dass die Verdoppelung der Stack-Größe auf 64 Elemente fast zu einer Verdoppelung der benötigten Zeit führt (43%

Tabelle 4: Ergebnisse für den in vier Kernel strukturierten Ray Tracer. Zu den getesteten Varianten werden die erreichte Framerate (FPS), die Zeit pro Frame in Millisekunden (ms/F), die Anzahl der Strahlen pro Sekunde in Millionen (MRays/s) und die Leistung relativ zum Referenzwert von *MBVH while-while* (Ref.) gezeigt.

BVH SAH 3 Axes	***Crytec Sponza***				***Mad Science***			
# Dreiecke	262.698				80.149			
1024 × 768	**FPS**	**ms/F**	**MRays/s**	**Ref.**	**FPS**	**ms/F**	**MRays/s**	**Ref.**
while-while	24,2	41,34	59,34	91%	22,7	44,1	85,83	92%
if-if	22,9	43,68	56,17	86%	21,4	46,67	81,11	87%
if-while	24,4	40,92	59,95	92%	23,5	42,53	89,00	95%
MBVH while-while	**26,5**	**37,67**	**65,13**	**100%**	**24,7**	**40,48**	**93,51**	**100%**
MBVH if-if	23,3	42,95	57,12	88%	21,6	46,35	81,67	87%
MBVH if-while	24,3	41,15	59,62	92%	22,8	43,91	86,20	92%
Stack(32) w-w	11,2	89,58	27,39	42%	7,4	134,08	28,04	30%
Stack(32) i-i	9,9	100,98	24,29	37%	6,9	145,95	25,94	28%
MBVH Stack(32) w-w	12,5	79,89	30,71	47%	8,1	122,75	30,84	33%
MBVH Stack(32) i-i	10,3	96,87	25,33	39%	7,0	143,11	26,45	28%
w-w; i-w; w-w:								
BVH	24,4	40,12	61,15	94%	23,4	42,80	88,44	95%
MBVH	25,7	38,9	63,07	97%	24,0	41,71	90,75	97%

BVH SAH 3 Axes	***Dragon***				***Conference***			
# Dreiecke	871.416				331.179			
1024 × 768	**FPS**	**ms/F**	**MRays/s**	**Ref.**	**FPS**	**ms/F**	**MRays/s**	**Ref.**
while-while	43,7	22,87	124,74	93%	44,3	22,56	116,08	92%
if-if	44,1	22,68	125,78	94%	40,0	24,96	104,92	83%
if-while	44,7	22,35	127,64	95%	44,0	22,72	115,26	91%
MBVH while-while	**47,0**	**21,28**	**134,06**	**100%**	**48,2**	**20,76**	**126,14**	**100%**
MBVH if-if	44,7	22,38	127,47	95%	40,7	24,59	106,50	84%
MBVH if-while	43,0	23,25	122,70	92%	43,3	23,11	113,32	90%
Stack(32) w-w	16,7	59,81	47,70	36%	13,9	71,95	36,40	29%
Stack(32) i-i	16,4	61,15	46,65	35%	12,3	81,28	32,22	26%
MBVH Stack(32) w-w	18,3	54,77	52,08	39%	15,4	64,96	40,31	32%
MBVH Stack(32) i-i	17,0	58,82	48,50	36%	12,7	78,75	33,25	26%
w-w; i-w; w-w:								
BVH	45,2	22,12	128,96	96%	45,0	22,23	117,80	93%
MBVH	46,3	21,60	132,07	99%	47,0	21,28	123,06	98%

Tabelle 5: Ergebnisse für die Stack-basierte Traversierung mit einer Stackgröße von 64 für *while-while* mit BVH und MBVH. Die Spalte „Cor." gibt die Leistung relativ zur korrespondierenden Methode in Tabelle 4 an, Spalte „Ref." gibt die Leistung relativ zur MBVH while-while Referenz in Tabelle 4 an.

BVH SAH 3 Axes	*Crytec Sponza*					*Mad Science*				
# Dreiecke	262.698					80.149				
1024 × 768	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**
Stack(64) w-w	6,4	156,77	15,65	57%	24%	4,0	250,24	15,13	54%	16%
MBVH Stack(64) w-w	7,2	139,62	17,57	57%	27%	4,4	227,36	16,65	54%	18%
BVH SAH 3 Axes	*Dragon*					*Conference*				
# Dreiecke	871.416					331.179				
1024 × 768	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**
Stack(64) w-w	9,0	111,28	25,64	54%	19%	7,5	134,21	19,51	54%	15%
MBVH Stack(64) w-w	9,8	101,75	28,04	54%	21%	8,3	120,98	21,65	54%	17%

bis 46% langsamer als bei 32 Elementen). Die Leistung nimmt also nahezu linear mit der Größe des Stacks ab. Mögliche Ursachen für diesen Effekt werden in Abschnitt 5.3 diskutiert.

Betrachtet man die relativen Leistungs-Differenzen der Stackless Varianten zum jeweiligen Optimum über die vier Szenen hinweg, fällt auf, dass diese sehr nah beieinander liegen, wobei die *Dragon* Szene bzgl. *if-if* herausfällt. Zum Beispiel ergibt sich für die *while-while* Variante zwischen den Szenen ein Unterschied von bis zu 2 Prozentpunkten (7% - 9% vom jeweiligen Optimum entfernt), bei *if-if* sind es, *Dragon* ausgenommen, bis zu 4 Prozentpunkte (13% - 17% vom Optimum entfernt), während sich *Dragon* bezüglich dieser Distanz um bis zu 11 Prozentpunkte von den anderen unterscheidet. Bei der besten Stack-basierten Methode ist die Varianz zwischen den Szenen bezüglich der Differenz zum Optimum mit bis zu 15 Prozentpunkten größer.

Zum Vergleich wurden auch BVHs aus der schnelleren Variante des Aufbauverfahrens erstellt, bei der nicht alle drei Achsen betrachtet werden, sondern nur diejenige, auf welcher die Centroid Bounds die größte Ausdehnung aufweisen. Die ermittelten Leistungswerte werden in Tabelle 6 dargelegt und zu den Leistungsdaten in Tabelle 4 in Bezug gesetzt. Die Daten zeigen sehr szenenabhängige Ergebnisse. Für die *Sponza* Szene führt die BVH aus diesem Aufbauverfahren sogar zu einer besseren Leistung um 12% bis 13%, bezogen auf die korrespondierenden Varianten mit BVH auf Basis der Unterteilungssuche auf allen drei Achsen. Bei der *Mad Science* Szene hingegen ist die Leistung um -8% bis -9% durchgehend schlechter. Bei der *Dragon* Szene ergeben sich sowohl bis zu 3% bessere Werte als auch um -2% schlechtere, insgesamt fällt der Unterschied für diese Szene also sehr gering aus. Für die *Conference* Szene werden um -10% bis -11% schlechtere Leistungswerte erreicht.

Tabelle 6: Ray Tracing (4 Kernel) Ergebnisse mit BVH aus dem Aufbauverfahren mit Unterteilungssuche auf der Achse mit der größten Ausdehnung der Dreiecks-Schwerpunkte (Centroid Bounds). Die Spalte „Cor." gibt die Leistung relativ zur korrespondierenden Methode in Tabelle 4 an. Spalte „Ref." gibt die Leistung relativ zur MBVH while-while Referenz in Tabelle 4 an.

SAH Centroid Extend	*Crytec Sponza*					*Mad Science*				
# Dreiecke	262.698					80.149				
1024 × 768	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**
while-while	27,0	37.01	66,29	112%	102%	20,6	48,51	78,03	91%	83%
MBVH while-while	29,8	33,56	73,10	112 %	112 %	22,6	44,15	85,74	92%	92%
MBVH if-if	26,0	38,42	63,85	112%	98%	19,6	50,98	74,25	91%	79%
MBVH if-while	27,4	36,51	67,20	113%	103%	20,9	47,85	79,10	92%	85%
SAH Centroid Extend	*Dragon*					*Conference*				
# Dreiecke	871.416					331.179				
1024 × 768	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**
while-while	44,5	22,47	126,96	102%	95%	39,5	25,33	103,38	89%	82%
MBVH while-while	48,3	20,65	138,14	103%	103%	43,2	23,14	113,17	90%	90%
MBVH if-if	43,8	22,85	124,84	98%	93%	36,1	27,72	94,47	89%	75%
MBVH if-while	44,4	22,52	126,67	103%	94%	38,4	26,05	100,53	89%	80%

Damit der Zusammenhang zwischen diesen Unterschieden und den SAH Kosten der BVHs aus den beiden Varianten des Aufbauverfahrens untersucht werden kann, werden die SAH Kosten für die Hierarchien berechnet. In Tabelle 7 werden die resultierenden Werte aufgeführt und verglichen. Die Berechnung der SAH Kosten wird mit den angenäherten Oberflächengrößen ohne Berücksichtigung von konstanten Kostenfaktoren nach Gleichung 26 (Abschnitt 4.4) durchgeführt. Zu beachten ist, dass die SAH Kosten nur szenenintern aussagekräftig sind. Bei der *Dragon* Szene sind die Kosten z.B. so niedrig, da die Bodenfläche zu einem sehr großen Wurzelknoten mit viel leerem Raum führt, so dass die Oberflächengröße des Wurzelknotens im Verhältnis zur Oberfläche des restlichen Baumes sehr groß ist. Die erhobenen Werte zeigen, dass die SAH Kosten wie erwartet durchgehend niedriger ausfallen, wenn alle drei Achsen für die Unterteilungssuche berücksichtigt werden. Für die Variante, in der nur die Achse mit der größten Ausdehnung der Centroid Bounds betrachtet wird, sind die SAH Kosten um Rund 2% bis 8% höher, bei der *Dragon* Szene sind es sogar rund 60%. Die geringste Differenz tritt bei der *Sponza* Szene auf. Bezüglich der SAH Kosten ist die Variante, welche drei Achsen betrachtet, also überlegen, und dennoch hat die Sponza Szene mit der anderen Variante bessere Leistungswerte (Weiteres dazu folgt in Abschnitt 5.3).

Tabelle 8 zeigt Leistungswerte für den Ray Tracer, der nicht auf mehrere Kernel unterteilt ist, sondern in einem einzigen Kernel zusammengefasst wurde. Als beste Work Group Größe wurde 8×8 bestimmt und eingesetzt. Die Distanzen von den Leistungswerten der verschiedenen Methoden zu

Tabelle 7: SAH Kosten und Anzahl der Knoten zu den BVHs auf Basis der Unterteilungssuche auf allen drei Achsen (BVH SAH 3 Axes) und der Unterteilungssuche entlang der Achse mit der größten Ausdehnung der Primitiv-Schwerpunkte (BVH SAH Centroid Extend).

	Crytec Sponza		*Mad Science*		*Dragon*		*Conference*	
# Dreiecke	262.698		80.149		871.416		331.179	
BVH SAH 3 Axes								
SAH cost	78,43	-	10,09	-	3,10	-	38,47	-
Nodes	157.465	-	46.197	-	571.627	-	111.067	-
BVH SAH Centroid Extend								
SAH cost	79,78	+1,7%	10,85	+7,5%	4,95	+59,68%	40,58	+5,48%
Nodes	159.011	+0,98%	47.019	+1,77%	578.845	+1,26%	105.695	-4,84%

Tabelle 8: Ergebnisse für die Einzelkernel Version des Ray Tracers. Die Leistung wird relativ zur korrespondierenden Methode aus Tabelle 4 gezeigt (Cor.) und zudem relativ zur Referenz aus Tabelle 4 (Ref.)

Single Kernel										
BVH SAH 3 Axes	*Crytec Sponza*					*Mad Science*				
# Dreiecke	262.698					80.149				
1024 × 768	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**
while-while	16,3	61,39	39,96	67%	61%	16,8	59,40	63,72	74%	68%
if-if	13,7	72,82	33,69	60%	52%	15,1	66,17	57,20	71%	61%
MBVH while-while	18,5	53,96	45,47	70%	70%	18,6	53,73	70,45	75	75%
	Dragon					*Conference*				
# Dreiecke	871.416					331.179				
1024 × 768	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**	**FPS**	**ms/F**	**MRays/s**	**Cor.**	**Ref.**
while-while	34,0	29,41	97,00	78%	72%	34,6	28,91	90,58	78%	72%
if-if	31,7	31,54	90,45	72%	67%	30,7	32,56	80,43	77%	64%
MBVH while-while	37,3	26,84	106,29	79%	79%	38,4	26,05	100,53	80%	80%

MBVH while-while dieser Version fallen in der Regel etwas größer aus, zeigen davon abgesehen die gleiche Tendenz wie für die Version mit Aufteilung auf mehrere Kernel. Daher werden in der Tabelle drei der Varianten ohne Stack aufgeführt. Die Leistung wird in Bezug gesetzt zur Leistung der korrespondierenden Methode des unterteilten Ray Tracers in Tabelle 4.

Die Einzelkernel Version zeigt deutlich schlechtere Leistungswerte als die Aufteilung des Ray Tracing Prozesses auf 4 Kernel sowohl im Vergleich mit der korrespondierenden Methode als auch in Bezug auf den Referenzwert. Die Variante *MBVH while-while* ist 20% bis 30% langsamer. Für *while-while* ergibt sich eine Differenz von 22% bis 33% zur korrespondierenden Methode.

Die insgesamt schnellste Variante, *MBVH while-while* ohne Stack mit dem in vier Kernel strukturierten Ray Tracer, wird zusätzlich für die Darstellung der Testszenen in einer Auflösung von 1920 × 1080 Bildpunkten eingesetzt (Tabelle 9). Die erreichte Framerate beträgt je nach Szene rund 13 bis 29

FPS, während die Leistung in Millionen Strahlen pro Sekunde durchgehend höher ist (um rund 24% bis 63%) als bei der geringeren Auflösung.

Tabelle 9: Aufgeteilter Ray Tracer mit *MBVH while-while* in 1920 × 1080.

BVH SAH 3 Axes	***Crytec Sponza***			***Mad Science***		
# Dreiecke	262.698			80.149		
1920 × 1080	**FPS**	**ms/F**	**MRays/s**	**FPS**	**ms/F**	**MRays/s**
MBVH while-while	12,6	79,42	80,67	12,9	77,51	123,45
BVH SAH 3 Axes	***Dragon***			***Conference***		
# Dreiecke	871.416			331.179		
1920 × 1080	**FPS**	**ms/F**	**MRays/s**	**FPS**	**ms/F**	**MRays/s**
MBVH while-while	29,0	34,49	218,86	23,8	42,06	156,12

5.3 Diskussion

Da bei der Traversierung eines inneren Knotens der BVH immer die beiden Kindknoten benötigt werden, bietet sich die MBVH Organisation auch für binäre BVHs an. Die Ergebnisse zeigen, dass die MBVH Organisation hier tatsächlich zu den besten Leistungswerten führt, was aufgrund der verringerten Lesezugriffe auf den globalen Speicher nicht überraschend ist. Für die Traversierung eines Knotens findet ein Lesezugriff auf 64 Byte statt, während die hier verwendete „klassische“ BVH Organisation weniger effizient aufgebaut ist und sowohl auf einen Knoten zu 64 Byte (inklusive Padding) als auch auf die BV Daten der beiden Kindknoten (nochmals 64 Byte) zugreifen muss. In Summe sind das 128 Byte.

Bezüglich der Traverserungs-Varianten zeigt sich in den Ergebnissen, anders als bei (Aila und Laine 2009), dass abgesehen von der Ausnahme bei der *Dragon* Szene, die *if-if* Variante größtenteils langsamer ist als *while-while*. Das könnte mit stetigen Verbesserungen der hardwareseitigen Arbeitsverteilung bei den späteren GPU Generationen zusammenhängen (Nvidia 2015b), denn Aila und Laine erklärten die damalige (2009) überraschende Überlegenheit der *if-if* Organisation mit den Defiziten der Hardware auf diesem Gebiet.

Die Ergebnisse, die im vorigen Abschnitt vorgestellt wurden, zeigen auch, dass die GLSL Compute Shader Implementation der Stack-basierten Traversierung um bis zu dreimal langsamer ist als das Äquivalent mit Bitstack. Während hier eine positive Tendenz für Traversierungsverfahren ohne Stack auf modernen GPUs erwartet wurde, ist die Differenz zu den Stack-basierten Verfahren überraschend hoch. Nicht zuletzt da häufig die Stack-basierte BVH-Traversierung auf GPUs, in CUDA implementiert, als schneller gilt (Áfra und Szirmay-Kalos 2014, Vinkler et al. 2014), obwohl der Speicheraufwand bei einem Traversierungs-Stack pro Strahl insgesamt sehr hoch ausfallen kann. Auch wenn die hier in GLSL implementierte

Traversierung mit Stack (als lokales Array mit konstanter Größe) nicht in dem Maße optimiert ist wie bei (Aila und Laine 2009, Aila, Laine und Karras 2012) im Rahmen der CUDA Kernel, ist die große Differenz zu den Verfahren ohne Stack nicht allein dadurch zu erklären.

Bei CUDA werden Arrays, die entweder dynamisch indexiert werden oder zu groß sind, in einem bestimmten Bereich des globalen Graphikspeichers (DRAM) gehalten, genannt „Local Memory“, für den gewisse Optimierungen (Caching) möglich sind. Anders als in CUDA gibt es in GLSL, nach bestem Wissen, keine Möglichkeit, explizit das Caching im L1 Cache für Zugriffe auf den globalen Speicher zu erzwingen. Wie Arrays behandelt werden, die im OpenGL Compute Shader lokal definiert sind, ist aus der GLSL Spezifikation nicht ersichtlich. Bei Ausführung auf einer Nvidia GPU mit CUDA Architektur liegt die Vermutung nahe, dass Ähnlichkeiten bezüglich der Nutzung der Hardware bestehen können. Das würde bedeuten, solange auf ein Array von konstanter Größe nur mit konstanten Indizes zugegriffen wird, und es klein genug ist, um in den Registerspeicher zu passen, kann es dort gehalten werden (Nvidia 2015a). Andernfalls wird es im globalen Speicher gehalten, wobei nicht klar ist, ob GLSL intern eine Äquivalenz zu dem Konzept des „lokalen Speichers“ innerhalb des globalen Graphikspeichers hat.

Die Ergebnisse in Tabelle 5 haben gezeigt, dass die Traversierung mit Stack bei Verdoppelung der allokierten Array Größe fast die Halbierung der Leistung zur Folge hat, obwohl die Zugriffe auf den Stack gleich bleiben. Wenn der Stack im Registerspeicher liegen würde, wäre die deutlich höhere Anzahl belegter Register und damit einhergehend ggf. geringere Kapazität für Hardware Multithreading ein möglicher Hinweis, allerdings müssten gleichzeitig die Zugriffe auf das Array dann deutlich schneller durchführbar sein. Da hier die Indexierung durch einen Stack Pointer in Form einer Integer Variablen geschieht, welche abhängig vom Traversierungsverlauf in- und dekrementiert wird, würde dies jedoch bedeuten, dass Arrays im Compute Shader auch dann im Register gehalten werden, wenn die Indexierung nicht konstant ist. Dass dies nicht der Fall ist, wird durch Experimente mit „Dummy-“ Arrays und Zugriffen ersichtlich, denn eine konstante Indexierung hat deutlich weniger negativen Einfluss auf die Leistung. Wenn das Array im globalen Speicher liegt und sich lediglich die Größe von 32 auf 64 Plätze ändert, während die Zugriffe gleich bleiben, erscheint die Halbierung der Leistung ebenfalls nicht schlüssig. Mit Hilfe weiterer Experimente war festzustellen, dass der Leistungseinbruch bereits ab dem ersten nicht konstant indexierten Zugriff auftritt, jedoch zahlreiche zusätzliche Zugriffe notwendig sind, um die Performanz noch weiter zu verschlechtern. Als Auslöser erwies sich dabei jeder Index in Form einer Variablen, die nicht explizit als `const` deklariert ist, auch die Schleifen-Kontroll-Variable einer simplen `for` Schleife, welche nur von Integer Literalen oder Werten von Konstanten abhängt und Variablen, denen explizit und unmittelbar vor dem Zugriff

konstante Werte zugewiesen werden. Der Compiler merkt also in diesem Zusammenhang auch in einfachsten Fällen nicht, wenn eine Indexvariable ausschließlich von konstanten Werten abhängt. Selbst ein einziger nicht konstanter Zugriff pro Thread initiiert einen Leistungseinbruch mit den gleichen Ausmaßen wie mehrere nicht konstante Zugriffe pro Traversierungsschritt. Dieses Verhalten erweckt die Vermutung, dass hier ein Problem besteht, dass sich auf OpenGL Compute Shader bzw. den GPU Treiber bezieht und bedarf weiterer Untersuchung, was jedoch aus dem Rahmen dieser Arbeit fällt.

Zu den BVHs aus den zwei Aufbau-Varianten zeigen die Ergebnisse den Effekt, dass BVHs mit höheren SAH Kosten dennoch in einzelnen Szenen bessere Leistung erzielen. Die BVHs aus dem Aufbau mit Unterteilungssuche auf Basis der Centroid Bounds anstatt der Suche auf allen drei Achsen führen je nach Szene sowohl zu schlechteren als auch zu besseren Leistungswerten. Die Unterteilungssuche auf allen drei Achsen soll Hierarchien mit besserer Qualität, bezogen auf die SAH Heuristik, erstellen, was durch die durchgehend geringeren SAH Kosten bestätigt wird. Dass die vereinfachte Suche mit Auswahl einer Achse auf Basis der Centroid Bounds für die *Sponza* Szene zu deutlich besseren Ergebnissen führt zeigt, wie z.B. in (Aila, Karras et al. 2013) dargelegt, dass die SAH Heuristik nicht alle Aspekte bewertet, von denen die Ray Tracing Performanz abhängt. Einer dieser Aspekte, welcher besonders hohe Auswirkungen auf die Performanz haben kann, ist die Überschneidung von BVs der Hierarchie. Bei der *Sponza* Szene ist die Geometrie nicht gleichmäßig verteilt, und es kommt bei der BVH vermehrt zu Überschneidungen der BVs (Santos et al. 2014). Die höhere Leistung für die Sponza Szene bei der BVH mit höheren SAH Kosten könnte damit zusammenhängen, dass in diesem Fall der Aufbau auf Basis der Centroid Bounds zu weniger Überschneidungen führt. Zur Überprüfung dieser Vermutung wurde im Rahmen dieser Diskussion die EPO Metrik („End-point overlap“) nach (Aila, Karras et al. 2013), in angenäherter Form berechnet (siehe auch Abschnitt 3.2). Angenähert bedeutet in diesem Fall, dass anders als bei Aila et al. anstatt der tatsächlichen Oberflächen der Dreiecke die BVs der Dreiecke zur Berechnung verwendet wurden. Im Falle einer Überschneidung wurden die BVs entsprechend geklippt. Die Verwendung der BVs anstatt der Primitive führt zu höheren Werten, die jedoch als grobe Annäherung für den Vergleich untereinander ausreichen. Die ermittelten angenäherten EPO-Werte zeigt Tabelle 10. Die EPO-Werte zeigen für die *Sponza* Szene, dass die Menge an zusätzlicher Arbeit durch Überschneidungen bei der BVH auf Basis der Centroid Bounds geringer ausfällt. Dies könnte die höhere Leistung bei geringfügig höheren SAH Kosten für die Sponza Szene erklären. Für *Mad Science* und *Conference* ist neben den SAH Kosten bei den BVHs aus dieser Variante des Aufbauverfahrens auch der EPO-Wert deutlich höher. Bei der *Dragon* Szene hingegen zeigt sich trotz deutlich höherem SAH-Wert um rund 60% eine geringfügige Verbesserung

Tabelle 10: Angenäherte EPO-Werte auf Basis der Dreiecks-BVs für BVHs aus beiden Varianten des Aufbauverfahrens: „3 Axes“ und „CB“ (Centroid Bounds). Spalte „Dif.“ gibt die relative Differenz der angenäherten EPO-Werte der BVHs beider Varianten an. Bei der *Dragon* Szene wurde die Bodenfläche entfernt.

Crytec Sponza			*Mad Science*			*Dragon* **o. Boden**			*Conference*		
3 Axes	**CB**	**Dif.**	**3 Axes**	**CB**	**Dif.**	**3 Axes**	**CB**	**Dif.**	**3 Axes**	**CB**	**Dif.**
92,96	90,01	-3,17%	3,11	5,14	65,41%	14,78	14,30	-3,22%	35,94	40,58	12,91%

der Leistung. Der EPO-Wert (*Dragon* mit Bodenfläche) zeigt hingegen eine geringe Erhöhung, wobei der Wert besonders niedrig ausfällt. Das liegt an der großen Bodenfläche, welche die Größe der Oberfläche der Wurzel bestimmt. Ohne die Bodenfläche ist der EPO-Wert für die Dragon Szene beim vereinfachten Aufbau, im Verhältnis zu *SAH 3 Axes*, geringer, was auch hier die Diskrepanz zwischen SAH Kosten und Leistungswerten zum Teil erklären könnte.

Die Ergebnisse zur Einzelkernel Version im Vergleich zur Mehrkernel Version bestätigen die Hypothese, dass auch für einen GPU Ray Tracer, der klassisches Ray Tracing durchführt, ein Leistungsgewinn durch eine geeignete Aufteilung erzielt werden kann (Abschnitt 4.1). Die Eigenschaften der GPU Architektur und das SIMT Ausführungsmodell (siehe Abschnitt 2.4) motivieren die Aufteilung, so dass letztendlich sowohl die Rückgabe der Kontrolle an den Host zwischen Kernel Ausführungen als auch die zusätzlichen Schreib- und Lesezugriffe auf den globalen Speicher kompensiert werden, und darüber hinaus für die schnellste Taversierungs-Variante ein Leistungsgewinn von bis zu 30% erzielt wird.

6 Fazit & Ausblick

Diese Arbeit hat sich mit dem Thema GPU Ray Tracing mit Beschleunigung durch BVHs beschäftigt und in diesem Rahmen insbesondere mit der Abbildung des Ray Tracing Prozesses auf die GPU Architektur, dem Aufbau und Einsatz einer Bounding Volume Hierarchie als Beschleunigungsstruktur und verschiedenen Möglichkeiten der BVH Traversierung. Neben der Recherche und Beschreibung von Forschungsarbeiten im Hinblick auf diese Themen wurden auf der Basis des gesammelten Wissens Verfahren ausgewählt und ein eigenes Konzept für einen GPU Ray Tracer mit BVH entwickelt. Dieses Konzept wurde schließlich mit OpenGL in C++ und Compute Shader Programmen in GLSL (hier auch als Kernel bezeichnet) umgesetzt und über mehrere Iterationen hinweg verbessert. Für diese Umsetzung beschäftigte sich die Arbeit vorab auch mit der Architektur moderner GPUs, dem Ausführungsmodell und deren Implikationen für die Programmierung für

GPUs. Die Erkenntnisse auf diesem Gebiet haben zusätzlich die Strukturierung des Ray Tracers in vier Kernel und die Implementierung beeinflusst. Der resultierende GPU Ray Tracer verwendet zur Beschleunigung eine vom Host Programm aufgebaute BVH, kann diese mit verschiedenen Varianten von Traversierungsverfahren mit und ohne Stack verarbeiten und so, basierend auf Primär- Schatten- und Sekundärstrahlen (Reflexion und Refraktion), Bilder von statischen Szenen in Echtzeit darstellen. Die Bedeutung des Begriffes „Echtzeit“ oder „Real-Time“ hat je nach Anwendungsgebiet eine etwas andere Ausprägung; hier soll er aussagen, dass die Bilder auf der hier verwendeten Hardware schnell genug berechnet werden, um eine flüssige Kamerabewegung in der Szene auszuführen. Für die kleinere Auflösung, 1024×768, werden je nach Szene zwischen 24 und 48 FPS erreicht, während bei 1920×1080 Bildpunkten noch zwischen 12 und 29 FPS angezeigt werden. Die Experimente mit verschiedenen Varianten der Traversierungsverfahren und der Organisation der BVH im Speicher hat gezeigt, dass die MBVH Organisation zusammen mit einem *while-while* Ablauf mit dem Stackless Verfahren nach (Áfra und Szirmay-Kalos 2014) im entwickelten Ray Tracer die besten Leistungswerte erreicht. Zudem wurde gezeigt, dass die Aufteilung des Ray Tracing Prozesses auf mehrere Kernel deutlich performanter läuft als eine Implementierung des gleichen Ray Tracers in nur einem Kernel.

In den Forschungsarbeiten zum Thema GPU Ray Tracing wurde in den letzten 9 Jahren meistens CUDA, seltener OpenCL und kaum, die erst im Jahr 2013 (in der Kernspezifikation) eingeführten, OpenGL Compute Shader (mit GLSL) für die Implementierung verwendet. Alle diese Möglichkeiten haben ihre Vor- und Nachteile, so können Compute Shader z.B. direkt aus dem OpenGL Kontext heraus eingesetzt werden, während CUDA und OpenCL einen Kontextwechsel und eine zusätzliche API benötigen. Andererseits ist die Unterstützung durch Tools für das Debugging und Profiling für CUDA und OpenCL sehr gut, während für OpenGL Compute Shader zu diesem Zeitpunkt weder für das Debugging noch das Profiling unterstützende Werkzeuge existieren. Dies, und die im Vergleich zu den genannten APIs bislang eingeschränkteren Möglichkeiten bestimmte Aspekte der GPU Architektur direkt zu nutzen, macht insbesondere die Optimierung von Compute Shader Programmen schwieriger. Dennoch hat sich die Erweiterung von OpenGL um Compute Shader hier als interessante und mächtige Möglichkeit erwiesen, um für Anwendungen im Graphik Kontext das Potential der GPU auch für darüber hinausgehende Berechnungen und Verfahren zu nutzen (ohne das Problem ggf. über Umwege auf Vertex und Fragment Shader abbilden zu müssen), wie z.B. für Ray Tracing auf der GPU innerhalb einer OpenGL-basierten Anwendung.

Naheliegende Erweiterungen des entwickelten GPU Ray Tracers für eine bessere Bildqualität sind Antialiasing, damit einhergehend stochastisches Ray Tracing (Distributed Ray Tracing) und die Unterstützung für texturierte

Oberflächen. Darüber hinaus wären auch prozedurale Materialien interessant. Zudem könnte eine Variante als Path Tracer implementiert werden oder eine Ergänzung um Photon Mapping stattfinden. Abgesehen von der Bildqualität ist die weitere Verbesserung der Leistung ein Thema für die Weiterentwicklung. Neben zusätzlichen Verfahren für den BVH Aufbau, wie z.B. SBVH, welches einen hybriden Ansatz zwischen BVH und kd-Tree darstellt und Hierarchien von sehr hoher Qualität (bei relativ langen Aufbauzeiten) erreicht, sind vor allem die Traversierungsverfahren bezüglich der Leistung interessant. Kürzlich wurde in (Binder und Keller 2016) ein Verfahren ohne Stack vorgestellt, welches schneller sein soll als das implementierte Verfahren aus (Áfra und Szirmay-Kalos 2014) und sich daher für die Implementierung anbieten würde. Darüber hinaus könnte auch durch andere Ansätze der Speicherorganisation der Daten noch Potenzial für Steigerungen der Effizienz ausgeschöpft werden, dies wäre zu bewerten.

Bezüglich der Sekundärstrahlen, deren Verarbeitung hier mit dem Shading Kernel zusammengefasst wurde, da die zusätzliche Aufteilung in diesem Fall die Leistung verschlechtert hat, wäre es möglich, die Speicherung der Strahlen im Buffer und deren Verarbeitung anders zu organisieren, um zu prüfen, ob eine weitere Unterteilung unter bestimmten Bedingungen doch die Performanz weiter verbessern kann. Eine Möglichkeit dazu wäre, bei einem Strahl auch die Koordinaten des zugehörigen Bildpunktes zu hinterlegen, um die Speicherung und Verarbeitung von der Indexierung auf Basis der Bildpunktkoordinaten (bislang als 1D Offset realisiert) zu entkoppeln, so dass die Strahlen unabhängig von der Zuordnung dicht in einem Buffer gespeichert werden, der nach Art einer Work Queue verarbeitet wird. So wäre, ähnlich wie in (Laine et al. 2013), eine Verdichtung der Speicherung und der Verarbeitung möglich.

Das Ray Tracing von dynamischen Szenen wäre zudem ein Feld für zahlreiche Erweiterungen, die andere Ansätze für den Aufbau und schnelle Aktualisierungen der Beschleunigungsstruktur erfordern. Insbesondere in diesem Zusammenhang sind zudem Ansätze interessant, welche die Leistungsfähigkeit der GPU bezüglich der Rasterisierung ausnutzen, um die primären Schnittpunkte zu finden und dann per Ray Tracing die Schatten, Reflexion und Refraktion zu ergänzen.

Neben der hier vorgestellten und realisierten Umsetzung mit OpenGL und Compute Shader Programmen wäre eine Umsetzung mit der Graphik und Compute API Vulkan (Khronos Group Inc 2016) interessant. Die API Spezifikation in Version 1.0 wurde in diesem Jahr veröffentlicht. Vulkan arbeitet auf einem niedrigeren Abstraktions-Level und verspricht höhere Performanz durch direktere Kontrolle der zugrundeliegenden Hardware. Allerdings bedeutet mehr Kontrolle und weniger Treiber-Overhead häufig auch eine Erhöhung des Aufwands bei der Entwicklung. Es wäre festzustellen, wie sehr sich eine Umsetzung in Vulkan bezüglich der erreichten Leistung unterscheidet.

Über die genannten Möglichkeiten hinaus wird es interessant, die weitere Entwicklung zu verfolgen. Wie werden sich OpenGL Compute Shader weiterentwickeln, wird es möglich, ähnlich wie in CUDA/OpenCL, in einem weiteren Umfang auf die Ressourcen der GPU zuzugreifen und für die Leistung relevante Daten, wie z.B. die Anzahl der genutzten Registerplätze, direkt abzufragen? Was wird sich in zukünftigen GPU Architekturen verändern, abgesehen von der stetig wachsenden Berechnungsleistung und der schnelleren Speicheranbindung? Und was wird möglich durch die Verbesserung vorhandener und die Entwicklung neuer Verfahren und Ideen für GPU Ray Tracing, Path Tracing und Beschleunigungsstrukturen? Ich bin gespannt darauf, wie die weiteren Entwicklungen auf diesen und auch anderen Gebieten der Computergraphik aussehen werden.

Abkürzungsverzeichnis

AABB Axis Aligned Bounding Box

AOS Array of Structures

API Application Programming Interface

BRDF Bidirectional Reflectance Distribution Function

BSP Binary Space Partitioning

BV Bounding Volume

BVH Bounding Volume Hierarchy / Bounding Volume Hierarchie

CB Centroid Bounds

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DRAM Dynamic Random Access Memory

EPO End-point Overlap

FPS Frames per Second

GDDR Graphics Double Data Rate

GLSL OpenGL Shading Language

GPGPU General Purpose Computation on Graphics Processing Unit

GPU Graphics Processing Unit

k-DOP k-Discrete Oriented Polytopes

kd-Tree k-dimensional Tree / k-dimensionaler Baum

LBVH Linear Bounding Volume Heirarchy

LCV Leaf Count Variability

MBVH Multi Bounding Volume Hierarchy

MIC Many Integrated Core Architecture

MRays/s Millionen Strahlen pro Sekunde

ms/F Millisekunden pro Frame

MSAA Multisample Anti-Aliasing

MTL	Material Library File (Wavefront)
OBB	Oriented Bounding Box
OBJ	Object File (Wavefront)
RGBA	Red Green Blue Alpha
SAH	Surface Area Heuristic
SBVH	Split Bounding Volume Hierarchy
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
SP	Streaming Prozessor
SOA	Structure of Arrays
SSBO	Shader Storage Buffer Object
UBO	Uniform Buffer Object

Literatur

Áfra, Attila T. und László Szirmay-Kalos (2014). "Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing". In: *Computer Graphics Forum*. Bd. 33, S. 129–140 (siehe S. 58, 60, 69, 73, 78, 83, 87, 88).

Aila, Timo, Tero Karras und Samuli Laine (2013). "On quality metrics of bounding volume hierarchies". In: *Proceedings of the 5th High-Performance Graphics Conference*. Hrsg. von Steve Molnar, Jens Krüger, Tim Purcell und Warren Hunt, S. 101–107. URL: http://research.nvidia.com/sites/default/files/publications/aila2013hpg_paper.pdf (siehe S. 45, 69, 85).

Aila, Timo und Samuli Laine (2009). "Understanding the efficiency of ray traversal on GPUs". In: *Proceedings of the Conference on High Performance Graphics*. Hrsg. von David Luebke, Philipp Slusallek, Stephen N. Spencer, David McAllister, Matt Pharr und Ingo Wald. New York, S. 145 (siehe S. v, 21, 34, 36, 44, 48, 49, 51–53, 60, 69–71, 78, 83, 84).

Aila, Timo, Samuli Laine und Tero Karras (2012). "Understanding the efficiency of ray traversal on GPUs–Kepler and Fermi addendum". In: *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02* (siehe S. 36, 44, 50, 51, 84).

Appel, Arthur (1968). "Some techniques for shading machine renderings of solids". In: *AFIPS '68 (Spring) Proceedings of the April 30 - May 2, 1968, spring joint computer conference*. Hrsg. von Unknown, S. 37–45 (siehe S. 6).

Arvo, James und David Kirk (1989). "A Survey of Ray Tracing Acceleration Techniques". In: *An Introduction to Ray Tracing*. Hrsg. von Andrew S. Glassner. London, UK, UK: Academic Press Ltd, S. 201–262 (siehe S. 13).

Bailey, Mike (2013). "Using GPU shaders for visualization, part 3". In: *IEEE Computer Graphics and Applications* 33.3, S. 5 (siehe S. 26, 28).

Barringer, Rasmus und Tomas Akenine-Möller (2013). "Dynamic stackless binary tree traversal". In: *Journal of Computer Graphics Techniques* 2.1, S. 38–49 (siehe S. 56, 58, 60, 69).

Bell, Brandon (2014). *GeForce GTX 980 Whitepaper* (siehe S. 22).

Bikker, Jacco (2007). "Real-time ray tracing through the eyes of a game developer". In: *2007 IEEE Symposium on Interactive Ray Tracing*, S. 1–10 (siehe S. 2).

Binder, Nikolaus und Alexander Keller (2016). "Efficient Stackless Hierarchy Traversal on GPUs with Backtracking in Constant Time". In: *Proceedings of High Performance Graphics*. HPG '16. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, S. 41–50. URL: https://diglib.eg.org/handle/10.2312/hpg20161191 (siehe S. 70, 88).

Bürger, Kai, Stefan Hertel, Jens Krüger und Rüdiger Westermann (2007). "GPU Rendering of Secondary Effects". In: *Vision, Modeling and Visualization 2007* (siehe S. 2).

Christensen, P. H., J. Fong, Laur und D. D. M. Batali (2006). "Ray Tracing for the Movie 'Cars'". In: *2006 IEEE Symposium on Interactive Ray Tracing*, S. 1–6 (siehe S. 2).

Christensen, Per H., George Harker, Jonathan Shade, Brenden Schubert und Dana Batali (2012). *Multiresolution radiosity caching for efficient preview and final quality global illumination in movies* (siehe S. 2).

Cook, Robert L. (1989). "Stochastic Sampling and Distributed Ray Tracing". In: *An Introduction to Ray Tracing*. Hrsg. von Andrew S. Glassner. London, UK, UK: Academic Press Ltd, S. 161–199. URL: http://dl.acm.org/citation.cfm?id=94788.94793 (siehe S. 9).

Cook, Robert L., Thomas Porter und Loren Carpenter (1984). "Distributed ray tracing". In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. Hrsg. von H. Christiansen. Bd. vol. 18, no. 3. Computer graphics. New York: ACM, S. 137–145 (siehe S. 9).

Dürer, Albrecht (1525). *Underweysung der Messung, mit dem Zirckel und Richtscheyt, in Linien, Ebnen und gantzen Corporen*. Bd. 4. Nuremberg (siehe S. 1).

Dutré, Philip, Kavita Bala und Philippe Bekaert (2006). *Advanced global illumination*. 2nd ed. Wellesley Mass.: AK Peters (siehe S. 1).

Friedrich, Heiko, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel und Philipp Slusallek (2006). "Exploring the use of ray tracing for future games". In: *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*. Hrsg. von Alan Heirich. New York, NY: ACM, S. 41–50. URL: http://www.johannes-guenther.net/RTG/RTG.pdf (siehe S. 2).

Garanzha, Kirill und Charles Loop (2010). "Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing". In: *Computer Graphics Forum* 29.2, S. 289–298 (siehe S. 21, 33, 34, 51, 65).

Glassner, Andrew S., Hrsg. (1989a). *An Introduction to Ray Tracing*. London, UK, UK: Academic Press Ltd (siehe S. 6).

— (1989b). *An Introduction to ray tracing*. London: Academic Press (siehe S. 9).

— (1989c). "An Overview of Ray Tracing". In: *An Introduction to Ray Tracing*. Hrsg. von Andrew S. Glassner. London, UK, UK: Academic Press Ltd, S. 1–31. URL: http://dl.acm.org/citation.cfm?id=94788.94789 (siehe S. 6–9).

Goldsmith, Jeffrey und John Salmon (1987). "Automatic Creation of Object Hierarchies for Ray Tracing". In: *IEEE Computer Graphics and Applications* 7.5, S. 14–20 (siehe S. 16, 17).

Günther, Johannes, Stefan Popov, Hans-Peter Seidel und Philipp Slusallek (2007). "Realtime Ray Tracing on GPU with BVH-based Packet Traversal". In: *IEEE Symposium on Interactive Ray Tracing*, S. 113–118 (siehe S. 21, 40, 47, 51, 61).

Haines, Eric (1989). “Essential Ray Tracing Algorithms: An Introduction to Ray Tracing”. In: *An Introduction to Ray Tracing*. Hrsg. von Andrew S. Glassner. London, UK, UK: Academic Press Ltd, S. 33–77 (siehe S. 11).

Hanrahan, Pat (1989). “A Survey of Ray-surface Intersection Algorithms: An Introduction to Ray Tracing”. In: *An Introduction to Ray Tracing*. Hrsg. von Andrew S. Glassner. London, UK, UK: Academic Press Ltd, S. 79–119 (siehe S. 11).

Hapala, Michal, Tomáš Davidovič, Ingo Wald, Vlastimil Havran und Philipp Slusallek (2011). “Efficient stack-less BVH traversal for ray tracing”. In: *Proceedings of the 27th Spring Conference on Computer Graphics*, S. 7–12 (siehe S. 69).

Havran, Vlastimil (2000). “Heuristic ray shooting algorithms”. Diss. Czech Technical University, Prag (siehe S. 61).

— (2007). “About the Relation Between Spatial Subdivisions and Object Hierarchies Used in Ray Tracing”. In: *Proceedings of the 23rd Spring Conference on Computer Graphics*. SCCG ’07. New York, NY, USA: ACM, S. 43–48. URL: http://doi.acm.org/10.1145/2614348.2614355 (siehe S. 12).

Havran, Vlastimil, Jiří Bittner und Jiří Zára (1998). “Ray tracing with rope trees”. In: *14th Spring Conference on Computer Graphics*, S. 130–140 (siehe S. 61).

Howes, Lee (2015). *The OpenCL Specification: Version 2.1*. Hrsg. von Khronos OpenCL Working Group (siehe S. 24).

Jensen, Henrik Wann und Niels Jørgen Christensen (1995). “Photon maps in bidirectional Monte Carlo ray tracing of complex objects”. In: *Computers & Graphics* 19.2, S. 215–224 (siehe S. 9).

Jensen, Henrik Wann und Per Christensen (2007). “High Quality Rendering Using Ray Tracing and Photon Mapping”. In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH ’07. New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/1281500.1281593 (siehe S. 9).

Kajiya, James T. (1986). “The rendering equation”. In: *ACM Siggraph Computer Graphics*. Bd. 20, S. 143–150 (siehe S. 4, 6, 9).

Karras, Tero (2012). “Maximizing parallelism in the construction of BVHs, octrees, and k-d trees”. In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, S. 33–37 (siehe S. 43).

Karras, Tero und Timo Aila (2013). “Fast parallel construction of high-quality bounding volume hierarchies”. In: *Proceedings of the 5th High-Performance Graphics Conference*. Hrsg. von Steve Molnar, Jens Krüger, Tim Purcell und Warren Hunt. New York, S. 89–99 (siehe S. 43, 44).

Kay, Timothy L. und James T. Kajiya (1986). “Ray tracing complex scenes”. In: *ACM SIGGRAPH Computer Graphics* 20.4, S. 269–278 (siehe S. 15).

Keller, A., L. Fascione, M. Fajardo, I. Georgiev, P. Christensen, J. Hanika, C. Eisenacher und G. Nichols (2015). “The path tracing revolution in the movie industry”. In: *Proc. ACM SIGGRAPH Courses*, S. 24–1 (siehe S. 2).

Khronos Group Inc (2016). *Vulkan 1.0.27: A Specification* (siehe S. 88).
Kirk, David B. und Wen-mei W. Hwu (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, MA: Elsevier Inc. (siehe S. 2, 3, 19–21).
Lafortune, Eric P. und Yves D. Willems (1993). "Bi-directional path tracing". In: *Proceedings of the Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*. Bd. 93, S. 145–153 (siehe S. 9).
Laine, Samuli (2010). "Restart trail for stackless BVH traversal". In: *Proceedings of the Conference on High Performance Graphics*, S. 107–111 (siehe S. 54, 56, 58, 69).
Laine, Samuli, Tero Karras und Timo Aila (2013). "Megakernels considered harmful: wavefront path tracing on GPUs". In: *Proceedings of the 5th High-Performance Graphics Conference*, S. 137–143 (siehe S. 21, 34, 64, 66, 88).
Lauterbach, C., M. Garland, S. Sengupta, D. Luebke und D. Manocha (2009). "Fast BVH Construction on GPUs". In: *Computer Graphics Forum* 28.2, S. 375–384 (siehe S. 42).
M. Ernst und G. Greiner (2008). "Multi bounding volume hierarchies". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, S. 35–40 (siehe S. 69).
MacDonald, J. David und Kellogg S. Booth (1990). "Heuristics for ray tracing using space subdivision". In: *The Visual Computer* 6.3, S. 153–166. URL: http://dx.doi.org/10.1007/BF01911006 (siehe S. 17, 19, 44, 69).
McGuire, Morgan und Michael Mara (2014). "Efficient GPU Screen-Space Ray Tracing". In: *Journal of Computer Graphics Techniques (JCGT)* 3.4, S. 73–85. URL: http://jcgt.org/published/0003/04/04/ (siehe S. 2).
Möller, Tomas, Eric Haines und Naty Hoffman (2008). *Real-Time Rendering*. 3rd ed. Wellesley Mass.: A.K. Peters (siehe S. 9, 11, 13, 14, 19, 71, 72).
Möller, Tomas und Ben Trumbore (1997). "Fast, Minimum Storage Ray/Triangle intersection". In: *Journal of Graphics Tools* 2.1, S. 21–28 (siehe S. 11, 73).
Müller, Stefan (2008). *Photorealistische Computergrafik*. Koblenz. URL: https://www.uni-koblenz-landau.de/de/koblenz/fb4/icv/agmueller/lehre/ws1415/pcg#folien (siehe S. 4, 5).
Nvidia (2015a). *CUDA C Programming Guide*. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model (siehe S. 21, 24, 26, 29, 54, 84).
— (2015b). *Tuning CUDA Applications for Maxwell* (siehe S. 20, 22, 26, 51, 54, 83).
Parker, Steven G., James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison et al. (2010). "Optix: a general purpose ray tra-

cing engine". In: *ACM Transactions on Graphics (TOG)*. Bd. 29, S. 66 (siehe S. 33, 34).

Pohl, Daniel (2009). *Quake Wars* Gets Ray Traced*. URL: https://software.intel.com/en-us/articles/quake-wars-gets-ray-traced/ (siehe S. 2).

Purcell, Timothy J., Ian Buck, William R. Mark und Pat Hanrahan (2002). "Ray Tracing on Programmable Graphics Hardware". In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. Hrsg. von Tom Appolloni. New York, NY: ACM, S. 703–712 (siehe S. 1, 3, 20, 31–33).

Rubin, Steven M. und Turner Whitted (1980). "A 3-dimensional representation for fast rendering of complex scenes". In: *ACM SIGGRAPH Computer Graphics*. Bd. 14, S. 110–116 (siehe S. 14).

Santos, Artur Lira dos, Veronica Teichrieb und Jorge Lindoso (2014). "Review and Comparative Study of Ray Traversal Algorithms on a Modern GPU Architecture". In: URL: http://hdl.handle.net/11025/11931 (siehe S. 61–64, 85).

Schmittler, Jörg, Daniel Pohl, Tim Dahmen, Christian Vogelgesang und Philipp Slusallek (2005). "Realtime Ray Tracing for Current and Future Games". In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05. New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/1198555.1198762 (siehe S. 2).

Scriba, Christoph J., Peter Schreiber und Jana Schreiber (2015). *5000 years of geometry: Mathematics in history and culture*. New York und London: Springer Basel (siehe S. 1).

Segal, Mark und Kurt Akeley (2013). *The OpenGL Graphics System: A Specification: (Version 4.3 (Core Profile) - February 14, 2013)* (siehe S. 3, 22).

— (2016). *The OpenGL Graphics System: A Specification: (Version 4.5 (Core Profile) - July 7, 2016)* (siehe S. 23, 24, 27, 29, 31).

Seymour, Mike (2012). *The Art Of Rendering (updated)*. URL: https://www.fxguide.com/featured/the-art-of-rendering/ (siehe S. 2).

Solid Angle S.L. (2016). *What is Arnold?* URL: https://www.solidangle.com/arnold/ (siehe S. 2).

Stich, Martin, Heiko Friedrich und Andreas Dietrich (2009). "Spatial splits in bounding volume hierarchies". In: *Proceedings of the Conference on High Performance Graphics 2009*, S. 7–13 (siehe S. 36, 40, 45, 60, 61).

Stone, Lawrence D. (1976). *Theory of optimal search*. Bd. 118. Elsevier (siehe S. 17).

Tabellion, Eric und Arnauld Lamorlette (2004). "An approximate global illumination system for computer generated films". In: *ACM Transactions on Graphics (TOG)* 23.3, S. 469–476 (siehe S. 2).

Vinkler, Marek, Vlastimil Havran und Jiří Bittner (2014). "Bounding Volume Hierarchies Versus Kd-trees on Contemporary Many-core Architectures".

In: *Proceedings of the 30th Spring Conference on Computer Graphics*. SCCG '14. New York, NY, USA: ACM, S. 29–36 (siehe S. 62, 63, 83).

Voica, Alexandru (2014). *Practical techniques for ray tracing in games*. URL: http://www.gamasutra.com/blogs/AlexandruVoica/20140318/213148/Practical_techniques_for_ray_tracing_in_games.php (siehe S. 2).

Wald, Ingo (2007). "On fast Construction of SAH-based Bounding Volume Hierarchies". In: *IEEE Symposium on Interactive Ray Tracing*, S. 33–40 (siehe S. 19, 37, 40, 41, 43, 48, 68).

— (2011). "Active thread compaction for GPU path tracing". In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, S. 51–58 (siehe S. 21, 36, 53).

Wald, Ingo, Solomon Boulos und Peter Shirley (2007). "Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies". In: *ACM Transactions on Graphics* 26.1 (siehe S. 13, 19, 36, 37, 39, 44, 48, 61).

Wald, Ingo, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker und Peter Shirley (2007). "State of the Art in Ray Tracing Animated Scenes". In: *STAR Proceedings of Eurographics 2007*. Hrsg. von D. Schmalstieg und J. Bittner, S. 89–116 (siehe S. 3, 13, 14).

Wald, Ingo, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin und Philipp Slusallek (2003). "Realtime ray tracing and its use for interactive global illumination". In: *Eurographics State of the Art Reports* 1.3, S. 5 (siehe S. 32).

Weghorst, Hank, Gary Hooper und Donald P. Greenberg (1984). "Improved Computational Methods for Ray Tracing". In: *ACM Trans. Graph.* 3.1, S. 52–69. URL: http://doi.acm.org/10.1145/357332.357335 (siehe S. 15).

Whitted, Turner (1980). "An improved illumination model for shaded display". In: *Communications of the ACM* 23.6, S. 343–349 (siehe S. 7, 9).

Wright, Richard S., Graham Sellers und Nicholas Haemel (2016). *OpenGL Superbible: Comprehensive Tutorial and Reference*. Seventh edition. New York: Addison-Wesley (siehe S. 23, 27, 29, 31).

Yoon, Sung-Eui und Dinesh Manocha (2006). "Cache-Efficient Layouts of Bounding Volume Hierarchies". In: *Computer Graphics Forum* 25.3, S. 507–516 (siehe S. 14).

Zlatuška, Martin und Vlastimil Havran (2010). "Ray tracing on a GPU with CUDA–comparative study of three algorithms". In: (Siehe S. 61).