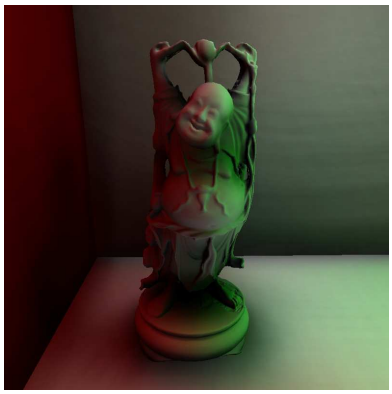
Lightcuts in CUDA mit Imperfect Shadow Maps

Arbesser Clemens*
Vienna University Of Technology

Mühlbacher Thomas†
Vienna University Of Technology


Zusammenfassung

Im Rahmen eines Praktikums haben wir uns im Wintersemester 2009 mit der globalen Beleuchtungsmethode Instant Radiosity und der Beschleunigungsstrategie Lightcuts beschäftigt. Der Fokus der Untersuchung liegt dabei auf der Fragestellung, ob eine Implementierung auf der Grafikkarte möglich bzw. sinnvoll ist. Dazu haben wir die beiden erwähnten Algorithmen sowohl in Software als auch unter Verwendung von CUDA entwickelt, einer von Nvidia entwickelten Architektur für GPGPU - Anwendungen. In diesem Dokument beschreiben wir einige Besonderheiten bei der Implementierung in CUDA sowie die dabei aufgetretenen Schwierigkeiten und stellen die Ergebnisse qualitativ und quantitativ gegenüber.

Inhaltsverzeichnis

*e-mail:e0625176@student.tuwien.ac.at
†e-mail:e0625075@student.tuwien.ac.at

1 Einleitung

Im Bereich der Computergraphik sind wenige Forschungsthemen derart aktuell wie die effiziente Erstellung von qualitativ hochwertigen, realitätsnahen Bildern. Es existieren derzeit dutzende von Lösungsansätzen für dieses Problem, das jedoch nach wie vor nicht als gelöst betrachtet werden kann. Schwierigkeiten ergeben sich vor allem bei der performanten Berechnung der globalen Beleuchtungssituation, d.h. der Simulation des physikalischen Lichttransports in einer Szene. Eine Möglichkeit, diese Berechnung durchzuführen, wurde 1997 von Alexander Keller in [Keller 1997] vorgestellt. Die Grundidee von *Instant Radiosity* ist es, zur Approximation des *ersten bounce*, d.h. der ersten Reflektion des Lichts in der Szene, ausgehend von den Lichtquellen Strahlen in die Szene zu schießen und an den Schnittpunkten mit der Szenengeometrie zusätzliche Lichter zu platzieren. Werden ausreichend viele solcher *virtueller* Lichter (VPLs, *V*irtual *P*oint *L*ights) erzeugt und wird jeder Pixel mit allen Lichtern beleuchtet, so konvergiert die Lösung gegen die korrekte Lösung (sofern Materialien etc. korrekt beachtet werden). Jedoch ist offensichtlich, dass die Performance von *Instant Radiosity* linear mit der Anzahl der VPLs skaliert, außerdem müssen, um sichtbare Artefakte (aufgrund der Diskretisierung der globalen Beleuchtung) zu vermeiden, sehr viele VPLs erzeugt werden - kurzum, es sind Strategien erforderlich, um die Verwendung von *Instant Radiosity* attraktiv zu machen.

Wir haben in den Monaten des Wintersemesters 2009/10 im Rahmen eines Praktikums versucht, eine derartige Strategie, den Algorithmus *Lightcuts*([Walter et al. 2005]), zu implementieren, wobei wir von der traditionellen Vorgehensweise abweichen. Wir möchten einerseits die Parallelität des Problems ausnützen (schließlich wird jeder Pixel separat berechnet) und andererseits das größte Bottleneck des ursprünglichen Algorithmus, nämlich die hohe Anzahl der (teuren) Shadowrays zwischen Pixeln und VPLs, umgehen. Um Ersteres zu erreichen war der Schluss naheliegend, die GPU zum Berechnen der Lightcuts zu verwenden. Die Verwendung von *Imperfect Shadow Maps*([Ritschel et al. 2008]) schließlich erlaubt es uns, das Schießen von Shadowrays zu vermeiden. Unsere Lösung beschränkt sich auf diffuse Materialien, eine Erweiterung auf leicht spiegelnde Oberflächen wäre möglich, ist für unsere Untersuchungen aber nicht erforderlich.

Wegen der Komplexität des Algorithmus und der Limitationen von gewöhnlichem Shader - Code verwenden wir CUDA, eine von Nvidia entwickelte Architektur für parallelisierbare Algorithmen, die es ermöglicht, 'C' Code auf neueren (Nvidia) Graphikkarten auszuführen und zu debuggen. In diesem Dokument beschreiben wir unsere Ergebnisse und führen einige quantitative und qualitative Betrachtungen durch, insbesondere der Entwicklung in CUDA betreffend. Wir setzen die Kenntnis von den beschriebenen Techniken voraus und beschränken uns in unserem Bericht auf jene Aspekte, die von den traditionellen Verfahren abweichen.

2 Lightcuts

2.1 Einleitung

Die Idee von Lightcuts ist, vereinfacht ausgedrückt, ähnliche Lichter zu clustern und beim Shaden gegebenenfalls ein repräsentatives Licht pro Cluster zur Beleuchtung zu verwenden (anstatt aller einzelnen Lichter bzw. VPLs). Dies hat zur Folge, dass die Anzahl der Lichter, die ausgewertet werden müssen, drastisch sinkt, es kann gezeigt werden, dass Instant Radiosity mit Lightcuts *sublinear* mit der Anzahl der Lichter skaliert. Damit das Ergebnis weiterhin von hoher Qualität ist, muss natürlich sichergestellt werden, dass die Verwendung von Clustern zur Beleuchtung im Endresultat keinen signifikanten Fehler einführt. Für Details verweisen wir auf das Originalpaper [Walter et al. 2005].

In der Praxis wird das Clustering in einem Vorverarbeitungsschritt berechnet, zur Laufzeit wird dann nur noch für jeden Pixel separat ein geeigneter *Lightcut* ausgewählt, d.h. ein Set von Lichtern, das (einzeln oder in einem Cluster) alle Lichter der Szene enthält. Der Vorverarbeitungsschritt erstellt, in einem Bottom - Up Verfahren, einen sog. *Lighttree*, d.h. einen Baum, der an der Wurzel einen alle Lichter der Szene beinhaltenden Cluster enthält, und an den Blättern die einzelnen Lichter bzw. VPLs.

In diesem Abschnitt beschreiben wir, wie wir die einzelnen Teile des Algorithmus implementiert haben, sowie ihre jeweilige Performance.

2.2 Die Erstellung von Virtual Point Lights

In einem ersten Schritt werden Flächenlichtquellen durch eine Anzahl an Punktlichtern ersetzt. Jedes Punktlicht (VPL) erzeugt wiederum jeweils eine gegebene Anzahl an VPLs. Dies geschieht denkbar einfach durch das Schießen von Strahlen in die Szene. Die Richtungen dieser Strahlen sind randomisiert und Cosinus - gewichtet, soll heißen, es werden eher Strahlen in Richtung der primären Lichtrichtung geschossen. Am jeweils ersten Schnittpunkt eines Strahls mit der Szenengeometrie werden neue VPLs erzeugt, deren Intensität entsprechend ihrer Entfernung zum Primärlicht geringer ist (Intensität nimmt quadratisch mit der Entfernung ab). Die Farbe der VPLs ist durch die Farbe des Primärlichts sowie des Materials, auf dem der Schnittpunkt liegt, bestimmt. Selbstverständlich kann dieses Vorgehen auch iterativ wiederholt werden, indem auch VPLs weitere VPLs erzeugen (sog. *second bounces*, *third bounces* usw.), wir haben uns in unserer Implementierung jedoch auf einen bounce beschränkt.

Das Standard - Verfahren zur Generierung von randomisierten, Cosinus - gewichteten Vektoren läuft wie folgt ab:

- 1. Generiere einen gleichverteilt zufälligen Punkt im Intervall $[0,1]x[0,1]$
- 2. Konvertiere diese Koordinaten der Einheitsbox zu Koordinaten im Einheitskreis (d.h. mit Mittelpunkt = Koordinatenursprung)

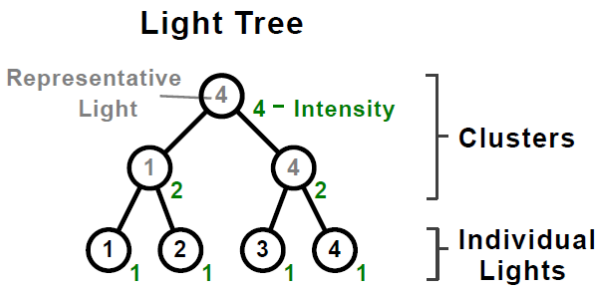


Abbildung 1: Ein Beispiel für einen Lighttree, entstanden durch Vereinigung der 4 Einzellichter in den Blättern. [Walter et al. 2005]

- 3. Projiziere die Koordinaten auf die Hemisphäre darüber
- 4. Die Koordinaten auf der Hemisphäre sind die gesuchten Vektoren

Um die Koordinatentransformation in Schritt 2 durchzuführen, verwenden wir den Ansatz von [Shirley and Chiu 1997], der für ebensolche Anwendungen äußerst nützlich ist.[1] Schritt 3 folgt *Malley's Method*, eine Beschreibung dieser findet man bspw. in [Cohen et al. 1993]. Klarerweise erhält man auf diese Weise nur Vektoren, die o.B.d.A. nach *oben* zeigen, eine abschließende Rotation zur Richtung, in die das Licht strahlen soll, ist also erforderlich. Die so erhaltenen Richtungen sind bereits ohne weitere Verarbeitung verwendbar.

2.3 Die effiziente Erstellung des Lighttrees

Nachdem alle Lichter erstellt wurden, wird, wie in der Einleitung bereits erwähnt, in einem nächsten Schritt ein sog. *Lighttree* erzeugt, ein binärer, im Allgemeinen nicht balancierter Baum. Zu diesem Zweck müssen Lichter zu Clustern von Lichtern vereinigt werden. Das *merging* von 2 Lichtern / Clustern funktioniert folgendermaßen:

- Wähle eines der Lichter / einen der Cluster als *repräsentatives Licht* aus[2]
- Berechne eine Bounding Box, die beide Lichter / Cluster beinhaltet
- Setze die Intensität des neuen Clusters gleich der Summe der Intensitäten seiner Kinder

Abbildung 1 zeigt ein Beispiel eines solchen Lighttrees, der durch Vereinigung von 4 Einzellichtern entstanden ist. Cluster müssen folgende Informationen speichern: ihre 2 Kinder, das repräsentative Licht, die Summe der enthaltenen Intensitäten und eine Bounding - Box.

Im Originalpaper läuft das Verfahren zur Erstellung des Lighttrees folgendermaßen ab:

- 1. Beginn mit jedem Licht als eigener Cluster
- 2. Vereine jene zwei Cluster, die entsprechend einer Metrik den kleinsten Cluster ergeben (greedy)
- 3. Wiederhole (2), bis nur noch ein Cluster übrig ist. Dieser ist die Wurzel des *Lighttrees*

Das von uns verwendete Ähnlichkeitsmaß zwischen Clustern wird gesondert im nächsten Abschnitt beschrieben. Schritt 2 verdient es jedoch, etwas genauer analysiert zu werden. Wie auch im originalen Paper erwähnt, kann dieser Schritt bei einer großen Anzahl von Lichtern sehr schnell zu sehr langen Berechnungszeiten führen, sofern die Suche nach dem kleinsten Cluster naiv[3] durchgeführt wird ($O(n^3)$ bei n Clustern). Zur Beschleunigung verwenden wir für diese Suche einen Kd - Tree, sowie eine Priority - Queue, die Cluster - Paare der Form $(Cluster, Cluster_{nearest})$ der Größe nach geordnet enthält. In unserer Implementierung verwenden wir nur die 3 Dimensionen der repräsentativen Lichtpositionen zur Erstellung des Kd - Trees, dies muss natürlich bei der Suche nach dem nächsten Nachbarn berücksichtigt werden. Um bei der Nearest - Neighbour Suche im Kd - Tree gewisse Zweige des Baums nicht auswerten zu müssen, muss an den Knoten ein *upper bound* gebildet werden, der angibt, ob es in den Kindern des Knotens ein besseres Ergebnis geben kann als das derzeitig Beste. Konkret bedeutet das bei Anwendung der in Abschnitt 2.4 vorgestellten Metrik, dass ein *upper bound* durch

$$upperBound = 1 + 1 + \frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}} \tag{1}$$

gegeben ist, wobei $\alpha_{\mathbb{C}}$ die diagonale Länge der gemeinsamen *bounding box* des fixen Clusters und des derzeitig besten Suchergebnisses ist und $\alpha_{\mathbb{S}}$ die Länge der Szenendiagonale. Die Schranke gilt, da bei der verwendeten Metrik die 3 Faktoren normierte Entfernung, Δcol und Lichtrichtungs-Unterschied jeweils im Wertebereich $[0;1]$ liegen und somit die Summanden 1 + 1 in Gleichung 1 den Worst-Case für Farb- und Richtungsunterschied darstellen. Für Implementierungsdetails verweisen wir auf [Mikšík 2007], das auch für andere Aspekte des Lightcuts - Algorithmus gute Anhaltspunkte liefert. Eine Implementierung einer Priority - Queue ist in C++ bereits in STL enthalten.

Das modifizierte Verfahren zur Erstellung des Lighttrees läuft nun so ab:

- 1. Erstelle einen Kd - Tree, der die einzelnen Lichter enthält
- 2. Finde für jeden Cluster denjenigen Cluster, der ihm, entsprechend einer Ähnlichkeitsmetrik, am ähnlichsten ist. Die Suche wird im Kd - Tree ausgeführt
- 3. Füge diese Cluster - Paare in eine Priority - Queue ein
- 4. Merge das oberste Cluster - Paar der Priority Queue, finde zu diesem neuen Cluster den nächsten anderen Cluster und füge dieses neue Cluster - Paar in die priority queue ein

[1] Für Details verweisen wir auf das Originalpaper

[2] Vorzugsweise randomisiert, jedoch mit der Intensität der Lichter gewichtet, sodass hellere Lichter / Cluster eher rep. Licht des Clusters werden als dunklere.

[3] Vergleich jedes Clusters mit jedem anderen

- 5. Wiederhole Schritt (4), bis in der Priority - Queue nur noch ein Paar übrig ist. Die Vereinigung der Cluster dieses letzten Paares ergibt die Wurzel des Lighttrees.

Folgende Feinheiten sind zu beachten: Auch Cluster, die soeben verschmolzen wurden, sind bereits implizit im Kd - Tree enthalten, da das repräsentative Licht des Clusters, der obigen Definition folgend, genau an der Stelle eines echten Lichts sitzt (nur die Intensität ist anders). Die Laufzeit dieses Verfahrens ist nur noch $O(n*log(n))$ bei n Clustern, was auch für viele Lichter schnell genug ist, nicht zuletzt da das Clustering bei statischen Szenen wiederverwendet werden kann.

2.4 Clustering - Kriterien

Bereits mehrmals angesprochen wurde die Notwendigkeit für die Definition einer Metrik, die 2 Clustern eine Ähnlichkeit zuordnet. Dieser Schritt ist für die Performance des Algorithmus **extrem** wichtig, da nur bei einer sinnvollen Metrik garantiert werden kann, dass bei Unterteilung eines Clusters in seine Kinder ein möglichst großer Genauigkeitsgewinn verbunden ist. Je besser das Clustering, desto geringer der Aufwand beim Ausrechnen des Lightcuts, da bereits nach wenigen Iterationen die erforderliche Genauigkeit erreicht ist. Im originalen Paper wird die Clustermetrik folgendermaßen definiert:

$$Clustersize = I_{\mathbb{C}} \left(\alpha_{\mathbb{C}}^2 + c^2 (1 - cos\beta_{\mathbb{C}})^2 \right) \quad (2)$$

wobei $I_{\mathbb{C}}$ die Summe der Intensitäten, $\alpha_{\mathbb{C}}$ die diagonale Länge der gemeinsamen *bounding box* und $\beta_{\mathbb{C}}$ der halbe Öffnungswinkel des gemeinsamen *bounding cone* der Licht-Hauptrichtungen der beiden Cluster ist. Die Konstante c steuert dabei, wie stark räumliche Ähnlichkeit bzw. direktionale Ähnlichkeit gewichtet werden sollen. Sowohl diese Konstante als auch die Berechnungsmethode bzw. die Bedeutung des *bounding cone* sind abhängig von der Art der verwendeten Lichter (omni - direktional, direktional, ...) und können im originalen Paper gefunden werden. Offensichtlich invalidiert bei Benutzung dieser Metrik bereits eine Änderung der Intensität der Lichter das Clustering. Um dies zu vermeiden, haben wir eine andere Metrik verwendet, die wie folgt definiert ist:

$$Clustersize = \frac{1}{3} \left(\frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}} + \Delta col + (0.5 - 0.5(\vec{dir}_A * \vec{dir}_B) \right) \quad (3)$$

Hierbei beschreibt Δcol die Farbdifferenz der beiden Lichter im Intervall $[0;1]$, $\alpha_{\mathbb{S}}$ die Länge der Szenen - Diagonalen und $\vec{dir}_A$ bzw. $\vec{dir}_B$ die Primärrichtungen der beiden Lichter (d.h. jene Richtungen, um die die Hemisphären zentriert sind). Durch die Normierung mit der Szenendiagonalen stellt man sicher, dass alle 3 Summanden den Wertebereich $[0;1]$ haben; die Multiplikation mit $\frac{1}{3}$ normiert das Ergebnis auf $[0;1]$. Gerade diese Gewichtung ist jedoch noch nicht gut genug begründet, eine qualitative Analyse wäre erforderlich um die Faktoren optimal zu wählen. Im nächsten Abschnitt stellen wir unter Anderem verschiedene Metriken gegenüber und vergleichen deren Einfluss auf die durchschnittliche Lightcut - Größe.

Tabelle 1: **Performance des Raycaster zur Erstellung von VPLs in der Szene vom Deckblatt**

Anzahl der erstellten VPLs	Durchschnittliche Zeit in sek.
100	0,15
1000	1,41
10000	14,1

Die Farbdifferenz wurde durch Konversion in den L^*C^*h Farbraum und Anwendung der sog. *CIE94* Methode berechnet.[4] Zwar gibt es bereits neuere und ausgereiftere Methoden zur Berechnung von Farbdifferenzen, *CIE94* ist aber für unsere Zwecke mehr als ausreichend und auch verhältnismäßig einfach zu implementieren. Wir haben die *CIE94* Methode jedoch insofern abgeändert, als dass wir den Luminosity Term aus der Gleichung entfernt haben. Durch die Einbeziehung der Farbdifferenz in die Metrik haben wir also 2 Fliegen mit einer Klappe geschlagen: Einerseits bleibt das Clustering auch für verschiedene Intensitäten gültig, und andererseits konnte die Qualität des Clusterings verbessert werden (siehe nächster Abschnitt).

2.5 Ergebnisse und Performance

Tabelle 2 zeigt die Qualität verschiedener Metriken bei unterschiedlichen Parametern. Zu jedem Eintrag wurden die Ergebnisse von 10 Versuchen gemittelt, die mittlere Streuung war in keinem Fall signifikant groß und wurde daher vernachlässigt.

In Tabelle 1 sind quantitative Statistiken zur Performance des verwendeten Raycaster zur Generierung von VPLs enthalten. Zum Einsatz kam ein 3 GHz Core Duo mit 3GB RAM.

Wie in der Tabelle ersichtlich ist, skaliert dieser Prozess linear mit der Anzahl der Lichter. Zur Beschleunigung haben wir einen einfachen bounding box Test (um nicht jeden Strahl mit allen Models schneiden zu müssen) implementiert. Da es sich hierbei um einen Vorverarbeitungsschritt handelt, ist die Performance mehr als ausreichend.

3 Imperfect Shadow Maps

3.1 Einleitung

Bei der Erzeugung hochqualitativer Bilder ist die Sichtbarkeitsberechnung stets eine der aufwändigsten und zeitintensivsten Aufgaben. Im ursprünglichen Lightcuts - Algorithmus wird die Verdeckung von Lichtquellen durch Schattenfühler (shadow rays) ermittelt, die ausgehend von den Lichtern mit der Szenengeometrie

[4] Siehe bspw. [Mandic et al. 2006]

Verwendete Metrik	Durchschnittliche Cutsize
$Clustersize = \frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}}$	68
$Clustersize = \frac{1}{2}\left(\frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}} + \Delta col\right)$	65
$Clustersize = \frac{1}{2}\left(\frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}} + (0.5 - 0.5(\vec{dir}_A * \vec{dir}_B)\right)$	60
$Clustersize = \frac{1}{3}\left(\frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}} + \Delta col + (0.5 - 0.5(\vec{dir}_A * \vec{dir}_B)\right)$	58

Tabelle 2: Qualität verschiedener Metriken, 110 Lichter

Verwendete Metrik	Durchschnittliche Cutsize
$Clustersize = \frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}}$	305
$Clustersize = \frac{1}{2}\left(\frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}} + \Delta col\right)$	265
$Clustersize = \frac{1}{2}\left(\frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}} + (0.5 - 0.5(\vec{dir}_A * \vec{dir}_B)\right)$	250
$Clustersize = \frac{1}{3}\left(\frac{\alpha_{\mathbb{C}}}{\alpha_{\mathbb{S}}} + \Delta col + (0.5 - 0.5(\vec{dir}_A * \vec{dir}_B)\right)$	220

Tabelle 3: Qualität verschiedener Metriken, 930 Lichter

geschnitten werden. Wie man sich vorstellen kann, ist diese exakte Lösung sehr teuer, besonders für komplexe Szenen. Die Autoren gehen davon aus, dass die Sichtbarkeitsberechnung ca. 50% der Rendering-Zeit ausmacht.

Eine andere Möglichkeit wäre, die Sichtbarkeit der Szene für jede Lichtquelle in gewisser Weise vorzuberechnen und abzuspeichern. Jedoch wird dieser Vorberechnungsschritt je nach Genauigkeit, Auflösung und Repräsentation ebenfalls viel Zeit in Anspruch nehmen. Ganz offensichtlich steigt der Speicherbedarf für vorberechnete Schatteninformation linear mit der Anzahl der Lichter, sofern man die Information für jedes Licht explizit speichert. Wir haben uns dennoch für letzteren Ansatz entschieden. Wir berechnen in einem Vorverarbeitungsschritt für jedes Punktlicht ein Rasterbild, welches für jede ausgehende Richtung auf der Hemisphäre die Entfernung zur nächsten Szenengeometrie enthält (Paraboloid Shadow Map [5]). Dazu rendern wir die Szene von jedem Licht aus gesehen in geringer Auflösung unter Verwendung einer parabolischen Projektion. Da bei der Rasterisierung Dreiecke und Linien jedoch linear interpoliert werden, dies auf der Halbkugel allerdings nicht mehr korrekt wäre, rendern wir anstelle der Dreicksrepräsentation eine Punktrepräsentation der Szene. Dies ist ohnehin naheliegend, da dadurch die Kosten des Shadow Map-Renderings drastisch reduziert werden und sich die dadurch entstehenden Fehler und Löcher aufgrund der hohen Anzahl an Punktlichtern in unserer Anwendung ziemlich gut gegenseitig aufheben. Diese Methode wurde 2008 in [Ritschel et al. 2008] unter dem Namen „Imperfect Shadow Maps“ vorgestellt.

3.2 Approximation der Szene durch eine Menge von Punkten

Wir generieren also beim Laden der Szene eine große Menge an Punkten, die wir zufällig auf den Dreiecken der Modelle verteilen. Dreiecke mit größerer Fläche erhalten entsprechend mehr Punkte. Zum Rendern der Imperfect Shadow Maps (ISMs) wird dann lediglich eine - von Licht zu Licht unterschiedliche - zufällige Teilmenge der Punkte auf das hemisphärische Blickfeld jedes Punktlichts projiziert. Dies dient zur Beschleunigung der ISM-Generierung und wird durch ein zufälliges Verwerfen von Punkten im Vertex-Shader-Programm erzielt. Konkret erreichen wir das Verwerfen durch Auswerten einer Noise-Funktion $N : \mathbb{R} \rightarrow [0,1]$ im Vertex-Programm, die als Parameter die Weltkoordinaten des aktuellen Punkts und die der Lichtquelle erhält. Ist das Bild der Noise-Funktion größer als ein gewisser Schwellwert, so setzen wir den Punkt hinter die Far-Clipping-Plane, wodurch der Punkt im Fragment-Programm ignoriert wird. Man könnte dieses Verfahren weiter beschleunigen, indem man die Punktrepräsentation der Szene schon vor der ISM-Generierung in eine zufällige Reihenfolge bringt und dann für jedes Licht eine Teilmenge fixer Größe ausgehend von einem zufälligen Start-Index in die Pipeline schickt. Somit würde bereits das Vertex-Programm deutlich seltener aufgerufen werden und man könnte sich das zufällige Verwerfen auf der Grafikkarte sparen.

3.3 Die Verwendung eines PullPush - Verfahrens zur Vermeidung von Artefakten

Durch das Rendern einer stark verkleinerten Punktmenge wird die ISM allerdings ziemlich viele Löcher aufweisen. Dies ist aufgrund der sehr geringen Auflösung der ISM (zwischen 32*32 bis maximal 128*128 Pixel) nur in begrenztem Ausmaß ein Problem. Um dem Problem weiterhin entschärfend entgegenzuwirken, haben wir ein Pull-Push-Verfahren, ähnlich dem von [Marroquim et al. 2007] bzw. [Grossman et al. 1998] implementiert, welches durch Resampling und Interpolation der Shadow Maps die Löcher zu füllen vermag. Dieses führen wir zur Gänze in CUDA durch. In der Pull-Phase wird ausgehend von der Shadow Map in Originalauflösung ein Downsampling um den Faktor 2 in beiden Dimensionen durchgeführt, indem aus je vier benachbarten pixels unter allen gültigen ein Mittelwert berechnet wird. Die Auflösung des resultierenden Mittelwertbilds beträgt ein Viertel der originalen ISM-Auflösung. Während der Push-Phase werden nun die Löcher aus dem origi-

[5] [Brabec et al. 2002]

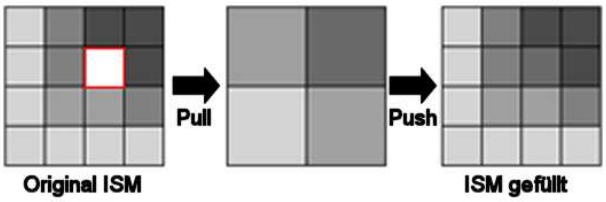


Abbildung 2: Füllung eines Loches durch Anwendung des PullPush Verfahrens

Abbildung 3: PullPush einer $128 * 128$ ISM mit 2 Iterationen

nalen Bild top-down mithilfe des gemittelten Bildes gefüllt. Dies funktioniert natürlich besonders gut für niederfrequente, zusammenhängende, am Besten konvexe Objekte. Abbildung 2 veranschaulicht diesen Vorgang, Abbildung 3 zeigt eine konkrete Anwendung bei der Füllung von Löchern einer ISM.

3.4 Ergebnisse und Performance

Somit können wir in einem einzigen Preprocess-Render-Schritt die Sichtbarkeit der Umgebung für alle orientierten Punktlichtquellen rudimentär ermitteln und durch den Pull-Push -Algorithmus zumindest für einigermaßen glatte, niederfrequente Geometrie zufriedenstellend rekonstruieren. Möchte man höherfrequente bzw. genauere Schatten rendern, kann man jedenfalls die Punktmenge vergrößern, um weniger stark auf die Pull-Push-Rekonstruktion angewiesen zu sein. Ein Erhöhen der Shadow-Map-Auflösung führt natürlich zu genauerer Speicherung der Visibility, allerdings auch zu größeren Löchern in der ISM.

Zum eigentlichen Shading der Szene kann man die Sichtbarkeit schließlich äußerst kostengünstig im Shadow-Map-Atlas nachschlagen. Pro Pixel führen wir pro Lichtquelle 5 Lookups durch, was natürlich eine große Zeitersparnis im Vergleich zur Verwendung von Schattenfühlern darstellt. Ein großer Nachteil des Verfahrens ist jedoch der hohe Speicheraufwand für Tausende von Shadow-Maps. Der Speicherplatz der Grafikkarte stellt eine klare obere Schranke für die höchstmögliche Anzahl der Lichter dar. Kann man bei Verfahren wie Instant Radiosity die ISMs problemlos auch nach und nach auf die Grafikkarte laden, um das Bild über mehrere Iterationen hinweg zu erzeugen, so würde dies bei Lightcuts die Sublinearität zerstören.

4 Implementierung in CUDA

4.1 Einleitung

CUDA ist eine von Nvidia entwickelte Software - Architektur, die es Entwicklern ermöglicht, die für parallele Aufgaben äußerst effiziente GPU für beliebige Algorithmen zu verwenden (GPGPU, General-purpose computing on graphics processing units). Die Entwicklung erfolgt dabei üblicherweise in C,[6] es existieren jedoch auch Anbindungen an andere Sprachen (Java, Matlab etc). CUDA bietet zudem die Möglichkeit, mittels eines Emulators GPU - Code auf der CPU auszuführen (zwecks Debugging, oder auch wenn auf dem Entwicklungs - PC keine CUDA enabled GPU vorhanden ist). Für gewöhnlich folgt die Entwicklung in CUDA folgendem Schema:

- Implementierung und Debugging eines „gold“ - Algorithmus auf der CPU
- Portierung dieses Algorithmus nach CUDA
- Debugging bis die Ergebnisse der beiden Algorithmen übereinstimmen
- Anwendung verschiedener Strategien zur Steigerung der Performance

Wir werden in den folgenden Abschnitten, ausgehend von unserem Projekt, auf bestimmte Aspekte der Implementierung in CUDA eingehen, für einen allgemeineren Überblick empfiehlt sich das Studium des „NVIDIA CUDA Programming Guide“, der Bestandteil des CUDA SDKs ist.

4.2 Dos and Don’ts

Wie eingangs erwähnt erfolgt die Entwicklung in C. CUDA bietet 2 Möglichkeiten, Code auf der GPU auszuführen: die Verwendung der "Driver - API"(low level) und die Verwendung der "C for CUDA: Runtime - API"(high level). In unserem Projekt haben wir letztere verwendet, da sie die Benutzung des Emulators erlaubt und einfacher zu handhaben ist.

- Ein Programm **muss** konsistent *eine* dieser Möglichkeiten verwenden, was manchmal leider problematisch ist, da für manche Anforderungen die runtime - API keine Unterstützung bietet. Beispielsweise ist es in „C for CUDA“ unmöglich, den derzeit freien Speicher auf der Grafikkarte abzufragen, hier muss man sich mit „educated guesses“ weiterhelfen - oder die Benutzung der „Driver - API“ in Betracht ziehen.
- Weiters empfiehlt es sich, eine klare Trennung zwischen GPU und CPU Code vorzunehmen (Aufspaltung in separate Dateien). Für konkrete Beispiele verweisen wir auf das CUDA SDK.

[6] Es stehen auch gewisse Funktionen von C++ zur Verfügung, für Details verweisen wir auf die API

- Häufig ist man versucht, bei der Portierung des gold - Algorithmus nach CUDA bereits einige Strategien zur Steigerung der Performance anzuwenden. Hiervon ist dringend abzuraten, da gerade diese Strategien zu Fehlern führen können, die nur bei der parallelen Ausführung auftreten (schwieriges und zeitaufwändiges Debugging).

- In diesem Zusammenhang sei auch darauf hingewiesen, dass die Verwendung des Emulators keine Garantie für die Lauffähigkeit des Codes auf der GPU bietet. Außerdem gibt es einige wichtige Unterschiede zwischen CUDA Emulation und CUDA:

  - Floating point Rechenoperationen liefern für gewöhnlich *nicht* die gleichen Ergebnisse
  - CUDA Emulation läuft sequentiell ab, einige Fehler treten jedoch erst bei der parallelen Ausführung auf
  - Bei Verwendung des Emulators können verschiedene Host - Funktionen (bspw. zwecks Debugging) aufgerufen werden, wie z.B. „printf“ u.Ä.

- Entwickelt man unter Windows, so ist darauf zu achten, dass der GPU Code nicht länger als einige Sekunden (abhängig von der verwendeten Windows Version) läuft, sofern die CUDA Berechnung auf jener GPU erfolgt, die auch für die Ausgabe am Bildschirm verantwortlich ist. Reagiert die GPU für eine gewisse Zeit nicht auf die Anfragen vom OS, so forciert Windows einen Neustart des Treibers, was zu einem Abbruch des CUDA Codes führt. Zur Lösung dieses Problems bietet es sich an, die Berechnung iterativ durchzuführen. Alternativ kann man auch durch Eingriffe in die Registry oder durch Verwendung spezieller Software den *timeout* beliebig einstellen.[7]

- Auf der GPU gibt es verschiedene, unterschiedlich schnelle und große Speicherbereiche. Dieser Punkt sollte in der Entwicklung eine zentrale Stellung einnehmen, da die Wahl der verwendeten Speicher für verschiedene Teile des Algorithmus mitunter gravierende Folgen für die Performance hat.

4.3 CUDA PullPush

Das in Abschnitt 3.3 vorgestellte PullPush Verfahren kann laufzeittechnisch sehr aufwändig werden, eine Portierung des Codes nach CUDA bietet sich an (aufgrund der parallelen Natur des Algorithmus). Die Implementation des eigentlichen Algorithmus ist beinahe deckungsgleich mit dem Gold - Algorithmus, eine Besonderheit unserer Lösung liegt in der Verteilung der Aufgaben auf die einzelnen GPU Threads. Wir haben dabei folgenden Ansatz gewählt: Jeder Thread berechnet nicht nur *ein* Mittel, sondern mehrere, je nach Größe der ISMs und der Anzahl der gleichzeitig laufenden Threads. Dadurch können wir die Größe des Thread - Gitters konstant halten. Diese Lösung skaliert auch für große Texturen gut.[8]

[7] Beispielsweise *NVIDIA Parallel Nsight*.

[8] Für beliebig große Texturen müsste man ggf. auch die einzelnen Levels iterativ behandeln, falls der Speicherplatz auf der GPU nicht ausreicht.

Die folgende Tabelle zeigt die Performance des Algorithmus für verschieden große Texturen:

Tabelle 4: **Performance des PullPush - Verfahrens in CUDA**

Auflösung der Textur	Durchschnittliche Zeit in sek.
1408x1408	0,011
2688x2688	0,033
3968x3968	0,08

Es zeigt sich, dass der Algorithmus, wie erwartet, linear mit der Anzahl der zu berechnenden Werte skaliert (da jeweils 4 Werte kombiniert werden), die Performance ist auch bei großen Texturen sehr hoch.

4.4 CUDA Lightcuts

Anders als beim bereits vorgestellten PullPush Verfahren ist die Implementierung von Lightcuts in CUDA problematisch, wodurch eine direkte Portierung des Codes von der CPU unmöglich gemacht wird. Die konkreten Probleme sind:

- CUDA unterstützt keine Rekursionen. Lightcuts und auch der Lighttree müssen in Arrays abgespeichert werden.

- Es gibt in CUDA keine Datenstruktur, um effizient auf das größte (bzw. kleinste) Element eines Sets zugreifen zu können (d.h. es gibt kein Äquivalent zu einer CPU priority queue). Der Grund dafür ist wohl, dass gerade solche Datenstrukturen in CUDA vermieden werden sollen, da die zugrundeliegenden Algorithmen für gewöhnlich sequentiell ablaufen. Da wir mehrmals pro Pixel den Cluster mit größtem Fehler aus dem aktuellen Lightcut finden müssen, lässt sich die Verwendung einer derartigen Struktur somit nicht vermeiden.

- Der Speicher der GPU ist äußerst begrenzt - diese Situation wird noch verschlimmert durch die Tatsache, dass dynamische Speicher - Allokationen in CUDA nicht möglich sind.[9] Dies führt dazu, dass für jeden Pixel bezüglich der Lightcut - Größe der „worst case“ angenommen werden muss, und dementsprechend $maxCutSize * anzahlThreads * sizeof(light)$ Speicher angelegt werden muss. Wir haben die Empfehlung des Originalpapers bezüglich der maximalen Cutsize übernommen (1000), die Größe eines Lichts ist in unserer Implementation 16 bytes - eine einfache Rechnung zeigt, dass für 32768 parallel ausgeführte Threads (2^{15}) bereits 500 Megabyte an Speicher benötigt wird, was für die meisten Grafikkarten zu viel ist. Offensichtlich kann man die Speicheranforderungen reduzieren indem man die Anzahl der gleichzeitig laufenden Threads reduziert, wodurch die Laufzeit jedoch entsprechend erhöht wird.

- Zusätzlich zu den Lightcuts müssen noch die ISMs und auch der Lighttree auf die Grafikkarte geladen werden, sowie Speicher für die Ergebnisbilder reserviert werden. **Keine** dieser

[9] Gemeint ist „global memory“, der nur in Host Code angelegt werden kann.

Strukturen kann je nach Bedarf geladen werden, sie alle müssen vollständig im Speicher der Grafikkarte angelegt sein, wodurch je nach verfügbarem Grafikkarten - Speicher die Anzahl der Lichter bzw. die Größe und Auflösung der ISMs stark eingeschränkt ist.

Zusätzlich zu diesen Problemen stellen sich noch weitere, CUDA spezifische Probleme:

- Für viele Performance - Strategien fanden wir keine Anwendungsmöglichkeit (bspw. die Verwendung von Shared Memory).
- Der Algorithmus „zwingt“ den Programmierer zu einer Vielzahl von Kontrollfluss - Anweisungen (control flow instructions) wie z.B. 'if', 'for', 'while' etc. Diese Anweisungen führen zu einer drastisch verschlechterten Performance. Grund dafür ist eine Aufspaltung des Kontrollflusses, was zu einer Erhöhung der Anzahl der Operationen führt - Details kann man dem CUDA Programming Guide entnehmen.
- Aufgrund des großen Speicherbedarfs musste für sehr viele Strukturen (Lightcuts, Lighttree, Ergebnisbild) der sog. „global memory“ verwendet werden, der mit Abstand langsamste Speicher auf der Grafikkarte, der zudem auch nicht gecached wird.

All dieser Probleme zum Trotz ist es uns gelungen, den Algorithmus in CUDA zu implementieren. Da wir das Shaden auf einzelne Threads verteilen und wir die Anzahl der gleichzeitig laufenden Threads kontrollieren können, ist die Auflösung des zu berechnenden Bildes (abgesehen von Speicherproblemen) keiner Limitation unterworfen (die Berechnung wird in unabhängige Batches von Threads = Pixeln aufgeteilt).

Den Richtlinien der CUDA API folgend, haben wir einige gängige Strategien zur Performancesteigerung verwirklicht:

- *coalesced global memory*: Durch konsistente Verwendung von CUDA-spezifischen Datentypen (*float2, float3, float4, int2, ...*) statt gewöhnlicher Arrays (*float[2], float[3]*, usw.) kann ein großer Geschwindigkeitsgewinn erreicht werden. Der Grund dafür ist, dass für Arrays alle Elemente einzeln angelegt und auch gelesen werden müssen, bei der Verwendung von *aligned memory* jedoch nicht. CUDA-Datentypen sind automatisch *aligned*, beim Einsatz von eigenen Datentypen kann durch Verwendung eines entsprechenden Suffixes (z.B.: __aligned(16)__) manuell ein Alignment erzwungen werden.
- *texture memory*: Da benachbarte Pixel auf benachbarte Felder der Positions - und Normal - Arrays zugreifen, bietet sich die Verwendung des *texture memory* an, der gecached ist. Hierbei ist zu beachten, dass 1D Texturen in CUDA einer starken Größenbeschränkung unterworfen sind[10], 2D Texturen jedoch nicht.

[10] Die genaue Grenze ist der CUDA API leider nicht zu entnehmen

Methode	Auflösung	#Lichter	Threads	Zeit[sek]
Radiosity HW	1000*x*1000	110	4096	0.526
			32768	0.189
Radiosity HW	1000*x*1000	420	4096	1.471
			32768	0.588
Radiosity HW	1000*x*1000	930	4096	3.069
			32768	1.208
Radiosity HW	1000*x*1000	2550	4096	7.718
			32768	3.137
Lightcuts HW	1000*x*1000	110	4096	5.733
			32768	6.182
Lightcuts HW	1000*x*1000	420	4096	31.392
			32768	35.609
Lightcuts HW	1000*x*1000	930	4096	62.454
			32768	67.241
Lightcuts HW	1000*x*1000	2550	4096	85.217
			32768	87.003

Tabelle 5: Gegenüberstellung der Performance von Instant Radiosity und Lightcuts in CUDA

- Schleifen, deren Durchläufe voneinander unabhängig sind und deren Anzahl von Iterationen zur Compile - Zeit bekannt ist, können mittels der Präprozessor - Anweisung *#pragma unroll x* „ausgerollt“ werden (der Parameter *x* stellt die Anzahl der Iterationen dar), was zu einer teilweise signifikanten Leistungssteigerung führen kann.

Weiters haben wir in CUDA zur effizienten Suche nach dem Cluster mit dem größten Fehler einen Heap implementiert, der gegenüber der naiven Suche eine deutliche Verbesserung darstellt, insbesondere bei einer großen Anzahl von Lichtern.

4.5 Ergebnisse und Performance

In diesem Abschnitt stellen wir die Ergebnisse unseres Praktikums vor. Der Schwerpunkt liegt dabei auf einem direkten Vergleich der Software - Lösungen mit den Hardware - Lösungen sowie einer qualitativen und quantitativen Analyse bezüglich der Skalierbarkeit und anderer wichtiger Parameter der einzelnen Methoden.

Abbildung 4 zeigt, wie die Implementierungen von Instant Radiosity mit und ohne Lightcuts in CUDA mit der Anzahl der Lichter skalieren. Tabelle 5 zeigt die entsprechenden Renderzeiten in tabellarischer Form und zusätzlich eine Gegenüberstellung der Renderzeit unter Verwendung von 4096 bzw. 32768 gleichzeitig ausgeführten Threads. Es zeigt sich, dass Instant Radiosity linear mit der Anzahl der Lichter skaliert, jedoch sehr stark von der Parallelität der Grafikkarte profitiert. Die Auswertung des Lichtcuts-Algorithmus ist vergleichsweise sehr aufwändig, sodass bei geringen Lichtanzahlen die Radiosity-Methode wesentlich schneller abläuft. Es lässt sich jedoch erkennen, dass der Anstieg der Renderzeit

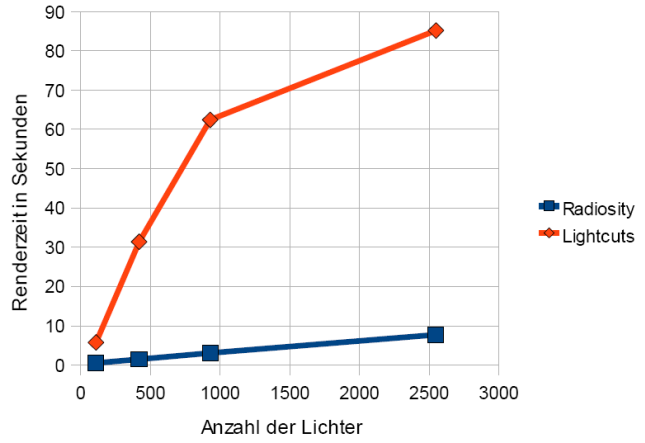


Abbildung 4: Renderzeiten der CUDA Implementierungen bei steigender Lichtanzahl, Auflösung = 1000x1000, 4096 concurrent Threads

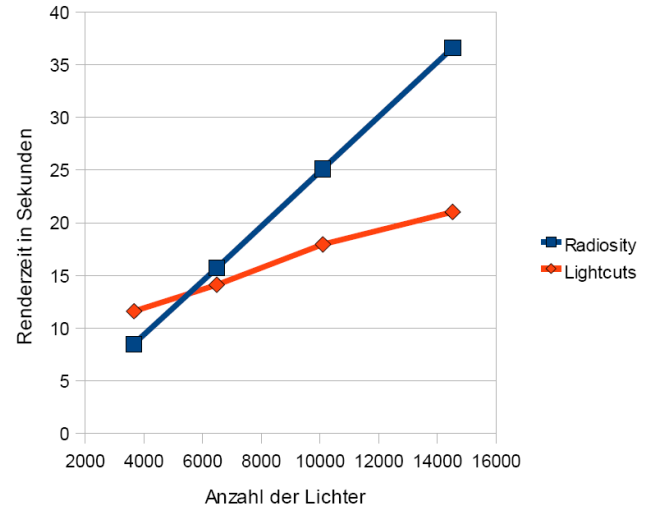


Abbildung 5: Renderzeiten der Software Implementierungen bei steigender Lichtanzahl, Auflösung = 50x50

bei Lightcuts mit steigender Lichtanzahl geringer wird, bei Radiosity hingegen gleich bleibt. Bei hohen Lichtanzahlen wird der Punkt erreicht werden, ab dem sich die Verwendung von Lightcuts zu lohnen beginnt. Aufgrund der bereits in Abschnitt 4.4 erwähnten Speicher - Limitationen der Grafikkarte konnten wir diese Stelle jedoch nicht erreichen.

Ein weiteres interessantes Ergebnis kam durch die Variation der gleichzeitig ausgeführten Threads zustande. Hierbei zeigt sich besonders eindrucksvoll, wie stark Instant Radiosity von CUDA profitieren kann. Dies ist nicht verwunderlich, sind doch die einzelnen Threads mit vergleichsweise „einfachen“ Aufgaben beschäftigt und - was extrem wichtig ist - der Kontrollfluss der Threads ist *ident*. Der Nvidia Programming Guide v.2.1 beschreibt diese Situation folgendermaßen[11]:

[11]Nvidia Programming Guide 2.1, s. 24

Methode	Auflösung	#Lichter	Zeit[sek]
Radiosity SW	50*x*50	3660	8.528
Radiosity SW	50*x*50	6480	15.715
Radiosity SW	50*x*50	10100	25.116
Radiosity SW	50*x*50	14520	36.589
Lightcuts SW	50*x*50	3660	11.613
Lightcuts SW	50*x*50	6480	14.124
Lightcuts SW	50*x*50	10100	17.956
Lightcuts SW	50*x*50	14520	21.038

Tabelle 6: Gegenüberstellung der Performance von Instant Radiosity und Lightcuts in Software

> A warp[12] executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a datadependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

In Instant Radiosity kann diese ideale Situation erreicht werden, bei Lightcuts kann davon aber keine Rede sein. Im Gegenteil, da die Auswahl der Lightcuts von Pixel zu Pixel unterschiedlich sein kann, kann der *worst case* eintreten, wenn nämlich die Ausführungspfade der einzelnen Threads völlig verschieden sind.

Abbildung 5 bzw. die dazugehörige Tabelle 6 zeigen, wie die Software - Lösungen von Instant Radiosity und Lightcuts mit der Anzahl der Lichter skalieren. Zu beachten ist, dass die Auflösung stark reduziert wurde, da die Berechnung eines 1000*x*1000 Bildes in Software sehr zeitaufwändig ist. Die Kurven zeigen einen ähnlichen Verlauf wie in Abbildung 4, die relative Performance ist jedoch gänzlich anders. In Software hat Instant Radiosity keinen Vorteil durch konsistente Ausführungspfade. Dementsprechend kann der Punkt der Überlegenheit von Lightcuts sehr viel früher erreicht werden.

5 Conclusio

Abschließend kann man festhalten, dass CUDA zwar durchaus die Implementierung eines derart komplexen Algorithmus wie Lightcuts erlaubt, alleine aufgrund der parallelen Natur der Aufgabe jedoch nicht auf Leistungssteigerungen geschlossen werden kann. Demgegenüber waren wir von der Geschindigkeit von Instant Radiosity mit CUDA positiv überrascht - Tabellen 5 und 6 stellen die unterschiedlichen implementierten Varianten von Instant Radiosity in Soft - und Hardware gegenüber. Nach wie vor gilt, dass Lightcuts auch in CUDA sublinear in der Anzahl der Lichter skaliert. Daher ist zu erwarten, dass auch die Implementierung von CUDA Lightcuts bei entsprechend vielen Lichtern einem einfachen Instant

[12]Anmerkung: Threads werden in Gruppen zu je 32 Threads zusammengefasst, die dann von einem Multiprocessor der GPU verarbeitet werden.

Radiosity Verfahren überlegen wird, durch den begrenzten Speicherplatz der GPU kann dies jedoch nicht ausgenutzt werden.

Der stetige Vormarsch parallel arbeitender Prozessoren wird viele bestehende Algorithmen obsolet machen. Zwar können manche auch parallel ausgeführt werden, um die Parallelität jedoch optimal nutzen zu können ist die Entwicklung von neuen Verfahren unabdinglich. Dennoch, für einfache Problemstellungen wie die von uns implementierte PullPush - Methode ist eine Parallelisierung einfach durchzuführen und liefert erstaunlich performante Ergebnisse. Architekturen wie CUDA unterstützen den Entwickler bei der Entwicklung derartiger Lösungen und werden in Zukunft vermutlich weiter an Bedeutung gewinnen.

6 Danksagungen

Wir möchten uns bei unserem Betreuer Paul Guerrero bedanken, durch dessen Unterstützung diese Arbeit möglich wurde.

Literatur

BRABEC, S., ANNEN, T., AND PETER SEIDEL, H. 2002. Shadow mapping for hemispherical and omnidirectional light sources. In *In Proc. of Computer Graphics International*, 397–408.

COHEN, M. F., WALLACE, J., AND HANRAHAN, P. 1993. *Radiosity and realistic image synthesis*. Academic Press Professional, Inc., San Diego, CA, USA.

GROSSMAN, WILLIAM, GROSSMAN, J. P., AND DALLY, W. J. 1998. Point sample rendering. In *Rendering Techniques '98*, Springer, 181–192.

KELLER, A. 1997. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 49–56.

MANDIC, L., GRGIC, S., AND GRGIC, M. 2006. Comparison of color difference equations. *International Symposium ELMAR 0*, 107–110.

MARROQUIM, R., KRAUS, M., AND CAVALCANTI, P. R. 2007. Efficient point-based rendering using image reconstruction. In *Proceedings Symposium on Point-Based Graphics*, 101–108.

MIKŠÍK, M. 2007. Implementing lightcuts. 113–120.

RITSCHEL, T., GROSCH, T., KIM, M. H., SEIDEL, H.-P., DACHSBACHER, C., AND KAUTZ, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, 1–8.

SHIRLEY, P., AND CHIU, K. 1997. A low distortion map between disk and square. *journal of graphics, gpu, and game tools 2*, 3, 45–52.

WALTER, B., FERNANDEZ, S., ARBREE, A., BALA, K., DONIKIAN, M., AND GREENBERG, D. P. 2005. Lightcuts: a scalable approach to illumination. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 1098–1107.