Paralleles Raytracing

— Praktikumsbericht —
Praktikum Paralleles Programmieren
Sommersemester 2014

Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Dennis Steinhoff (6250975)
	Alexander Koglin (6218451)
E-Mail-Adresse:	dennissteinhoff@live.de
	alexanderkoglin@hotmail.de
Betreuer:	Dr. Julian Kunkel
	Nathanael Hübbe

Hamburg, den 24.10.2014

Inhaltsverzeichnis

1 Einleitung

Raytracing bietet die Möglichkeit, durch Aussenden von Strahlen die Sichtbarkeit von dreidimensionalen Objekten in einem Raum von einem festen Punkt aus darzustellen. Dabei ist der Ursprung der Strahlen gerade dieser Punkt von dem der Raum betrachtet wird. Dazu wird der eventuelle Schnittpunkt der Strahlen mit den Objekten im Raum berechnet und so die Position der Objekte bestimmt. Als Erweiterung dessen lassen sich Objekte simulieren, die das Licht reflektieren und brechen können.
Nicht wie in der Natur vorkommt und wie man sich zuerst denken könnte, werden die Strahlen nicht von der/den Lichtquelle/n ausgesendet, sondern von der Bildebene, da nur die wenigsten Strahlen der Lichtquelle auch wirklich ins 'Auge' treffen [2]. Dementsprechend ist Raytracing an sich bereits optimiert. Unrealistischer wird es dadurch nicht, da wir ja auch bei der Netzhaut nur Objekte sehen, deren Lichtstrahlen das Auge treffen.
Integrieren der Lichtintensität und Farbe beim Auftreffpunkt der Strahlen ermöglicht, relativ realistische Bilder zu erzeugen. Dies hat allerdings auch seinen Preis, obwohl die Integration bereits Monte-Carlo-getreu diskretisiert wird.
Dem Raytracing selbst liegt der Code smallpt [1] von Kevin Beason zugrunde. Dieser beinhaltet den Aufbau der Cornell-Box und den seriellen Ablauf des Programms. Es handelt sich dabei um Pathtracing [10], einer speziellen Form von Raytracing. Im Unterschied zum diffusen Raytracing wird bei Pathtracing die sogenannte Rendergleichung, die dich Beleuchtungsdichte beschreibt, durch zufällig generierte Strahlen, die in einer Halbkugel um den Auftreffpunkt ausgesendet werden, auf allen Oberflächen gelöst und somit die globale Beleuchtung simuliert. Dadurch sind weiche Schatten möglich, was allerdings sehr viel Rechenleistung benötigt. Bei diffusen Raytracing werden optional weiche Schatten separat berechnet. Auch bei sehr kleinen Lichtquellen ist das Problem, dass die meisten ausgesendeten Strahlen von der Oberfläche wohl nicht die Lichtquelle treffen. Es wird sogenanntes Sampling verwendet. Je mehr Samples pro Pixel verwendet werden, desto mehr Strahlen pro Pixel werden ausgesendet. Wie man anhand der nachfolgenden Bilder erkennen kann, beeinflusst die Samplezahl die Bildqualität ziemlich stark. Ab einer bestimmten Samplezahl jedoch bemerkt man keine (großen) Unterschiede mehr. Das Bildrauschen ist dann fast verschwunden.

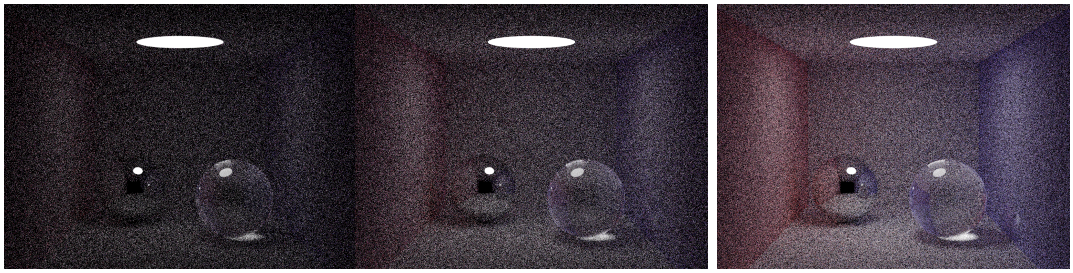

Abbildung 1: Szene mit unterschiedlichen Samples: 4, 8 und 16

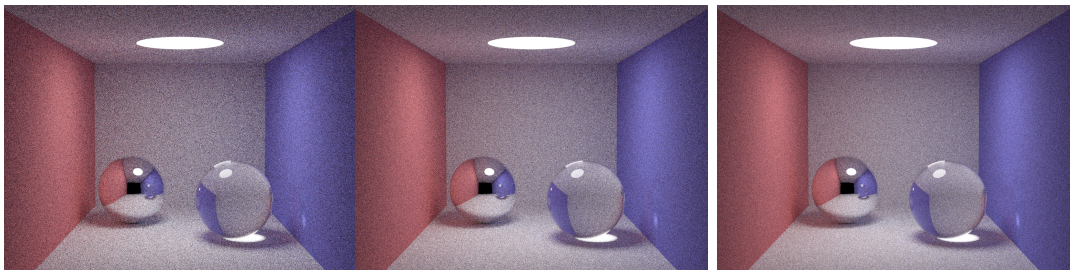

Abbildung 2: Szene mit unterschiedlichen Samples: 100, 148 und 200

Bei jedem Auftreffpunkt wird versucht, das darauf scheinende Licht zu approximieren. Je nachdem wie stark es gespiegelt oder gebrochen wird, trägt dieser Anteil gewichtet in die Gesamthelligkeit des Bildpunktes und der Farbe bei. Für eine genauere technische Erklärung ist die Präsentation von Dr. David Cline hilfreich [2].

2 Modellierung

In unserer Simulation werden nur kugelförmige Objekte dargestellt. Diese bieten die Möglichkeit, Objekte mit großer, gerader Oberfläche wie etwa Wände darzustellen, indem genügend große Kugeln betrachtet werden und die Kugeloberfläche auf Ausschnitten nahezu gerade ist. Die Anzahl beschreibbarer Szenen ist dadurch stark eingeschränkt. Betrachtet wird im Folgenden die Cornell-Box (nach der Universität, wo sie erfunden wurde, benannt), bestehend aus einer reflektierenden Kugel und einer licht-brechenden Kugel. Sie befinden sich in einem Raum mit verschiedenfarbigen Seitenwänden. Die anderen Wände sind farblos. Zudem befindet sich eine Lichtquelle an der Decke.

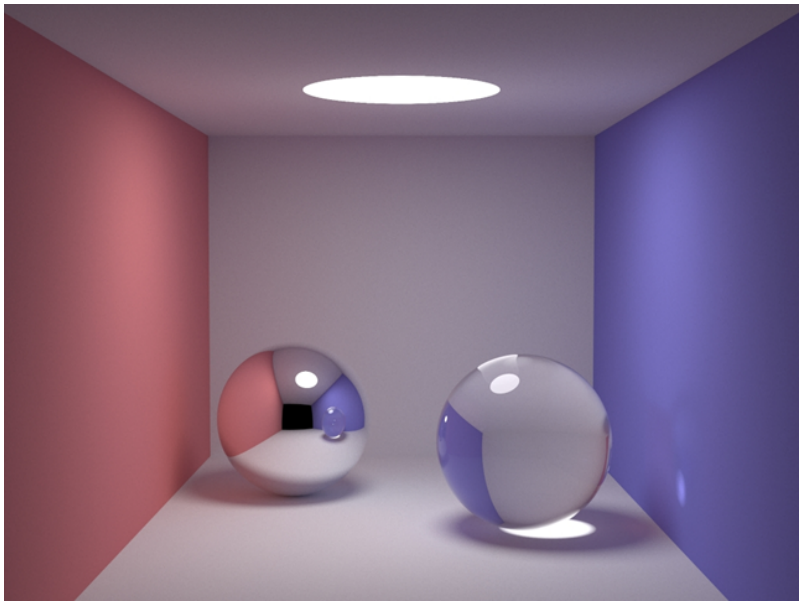

Abbildung 3: Cornell-Box, ausschließlich bestehend aus Kugeln

So lässt sich erkennen wie die blaue Farbe der rechten Wand an der linken Kugel reflektiert wird und an der rechten Kugel an der Wand-abgewandten Seite sichtbar wird.

Die Kamera wird als Strahl implementiert durch zwei Vektoren, wobei einer davon den Ursprung der Kamera darstellt und der Zweite die Richtung. Es werden vom Ursprung der Kamera alle Stahlen ausgesendet, die es dann zu verfolgen gilt und daraufhin den eventuellen Schnittpunkt der Strahlen mit den Kugeln.

3 Implementierung

Die Implementierung erfolgt in C++. Für die Kugeln werde Objekte definiert, bestehend aus einem double Wert für den Radius, einem Vektor für die Position, einem Vektor für die Strahlung, einem Vektor für die Farbe und dem Materialtyp. Der Materialtyp kann dabei entweder diffus, spiegelnd oder lichtbrechend sein.

Für das betrachtete Bild werden neun dieser Objekte erzeugt, für die sechs Wände, für die zwei Kugeln im Raum und für die Lichtquelle an der Decke.

Vom festen Ursprung der Kamera starten nun die Strahlen und zwar eine bei Programm-Aufruf festgelegte Anzahl pro Pixel. Diese werden nun abhängig davon, was für eine Materialtyp die Kugel, die Sie getroffen haben, hat entsprechend reflektiert oder gebrochen. Wie bereits oben erwähnt, findet auch bei diffusen Oberflächen bei Pathtracing eine gewisse Reflektion des Strahls statt.

Jeder Strahl berechnet den Farbwert für den entsprechenden Pixel. Je mehr Strahlen pro Pixel (Samples) genutzt werden, desto genauer wird der Farbwert für jedes Pixel und somit auch das Gesamtbild.

Es werden fünf for-Schleifen durchlaufen, die erste für Zeilen der Pixel im Bild, die zweite für die Spalten der Pixel im Bild, die dritte für 2x2 Sub-Pixel-Zeilen, die vierte für 2x2 Sub-Pixel-Spalten und die fünfte für die Strahlen pro Pixel (Samples), wobei diese wegen der Sub-Pixel durch vier geteilt werden. Die Sub-Pixel Zeilen bzw. Spalten erhöhen die Genauigkeit der Farbwerte, indem der Startpunkt des Strahls in der Bildebene variiert wird. Die Samples erhöhen die Genauigkeit, indem der Strahl variiert wird.

Durch eine höhere Anzahl der Pixel im Bild und Strahlen pro Pixel erhört sich der Rechenaufwand, weshalb es sinnvoll ist das Programm zu parallelisieren. Dies ist insbesondere so, weil der Geschwindigkeitszuwachs durch automatische Compileroptimierungen mit -O3 zwar beindruckend ist (Faktor vier: 102s vs. 398s bei 50 Samples), aber dennoch nicht genug.

Die serielle Version, die mit -ftree-vectorize, also der automatischen Vektorisierung durch den Compiler, kompiliert wurde, ist im Vergleich zum Original nicht schneller, weder bei 50 noch bei 100 Samples. Dies war nicht sonderlich überraschend [8], aber dennoch enttäuschend.

Die Analyse mit gprof war pathologisch, da die einzige Funktion, die von der Main-Funktion aufgerufen wird, die radiance() ist und somit dort 99% der Zeit verbracht wird. Innerhalb dieser wird aber die Schnittpunktsberechnung und die Berechnung der Lichtintensität inklusive Spiegelung und Lichtbrechung ausgeführt. Die Schnittpunktsberechnung ist wegen der geringen Anzahl an Objekten und nur einer Objektart ziemlich effizient.
Somit ist die Intensitätsberechnung sehr aufwändig. Dies liegt an der rekursiven Strahlaussendung, um globale Beleuchtung zu ermöglichen.

Auf der Webseite von smallpt werden desweiteren explicit.cpp und forward.cpp [1] bereitgestellt. Bei dem ersten Programm gibt es keine rekursiven Strahlen auf diffusen Oberflächen, sondern es wird explizites Light Sampling angewendet. Hierbei werden nur Raumwinkel in Richtung der Lichtquelle(n) betrachtet. Dadurch wird die benötigte Samplezahl stark reduziert, auch weil mehr Licht pro Sample erlangt wird, und die Berechnung dauert für vier Samples gerade einmal 12 Sekunden statt der 102 Sekunden bei 48 Samples.
Beim letzten Programm wurde die radiance-Funktion, welche mehr oder weniger die einzige nichttriviale Funktion ist, die direkt von der Main-Funktion aufgerufen wird, optimiert. Hier wird die gesamte Rekursion entfernt; es werden bei 48 Samples 17

Sekunden gebraucht. Allerdings ist das Ergebnis auch miserabel: Man kann zwar die Szene halbwegs erkennen, doch bilden sich Artefakte.

4 Parallelisierung

Im Folgenden wird ein Bild der Größe 1024x768 Pixel betrachtet. Die Parallelisierung erfolgt mit Hilfe von OpenMP oder MPI.

4.1 OpenMP

Eine einfache Möglichkeit der Parallelisierung bietet OpenMP, indem man die for-Schleife über Zeilen parallelisiert.

```
96  #pragma omp parallel for schedule(dynamic, 1) private(r)       // OpenMP
97  for (int y=0; y<h; y++){                                      // Loop over image rows
```

Abbildung 4: OpenMP-Pragma

Es ergibt sich folgender Speedup:
Das serielle Programm benötigt 102 Sekunden bei 50 Strahlen pro Pixel, bei Parallelisierung durch OpenMP mit 8 Threads werden nur noch 14,1 Sekunden benötigt.

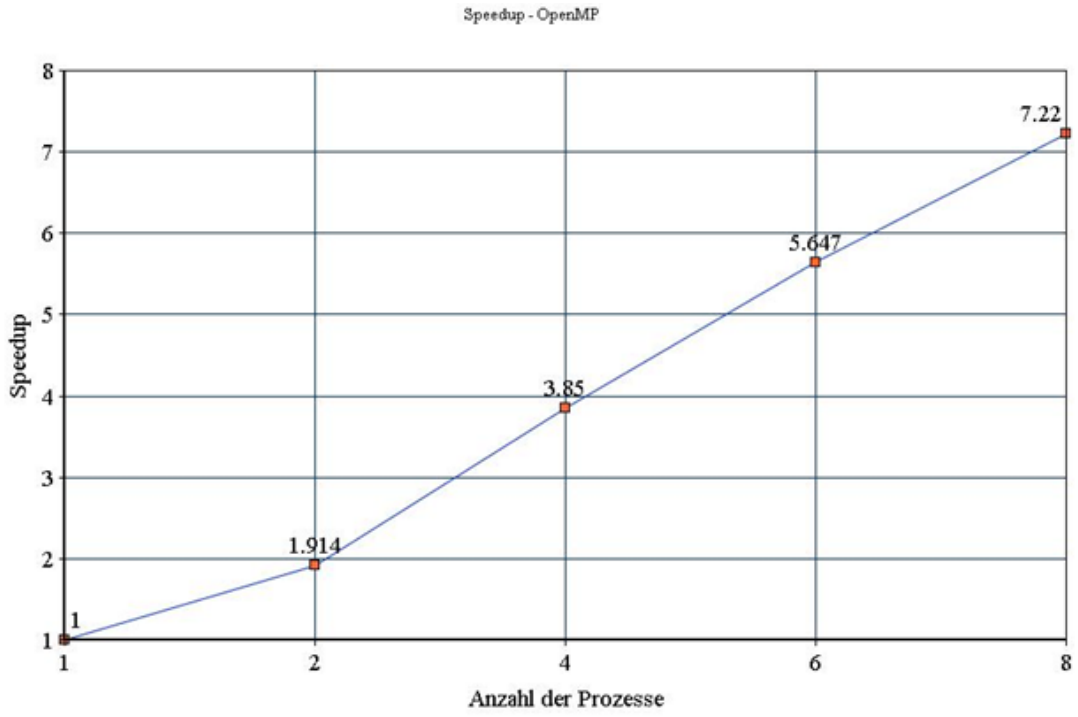

Abbildung 5: Speedup

Eine sogenannte hybride Parallelisierung bestehend aus MPI und OpenMP ist möglich. Dabei wird innerhalb eines MPI-Prozesses ein paralleler OpenMP-Abschnitt ausgeführt. Wir haben dies anhand eines Beispiels getestet, aber nicht beim Raytracing verwendet, da keine Erhöhung der Parallelität notwendig ist, bzw. dies durch die Erhöhung der Prozessanzahl auf einem Knoten erledigen lässt, obwohl dies mehr Arbeitsspeicher benötigt, was aber vernachlässigbar ist.

4.2 MPI

Bei MPI können wir mithilfe von SLURM [3], das ein Jobskript benötigt, unser Programm als viele Prozesse auf unterschiedlichen Knoten ausführen. Schreibt man `echo "‘date‘"` einmal vor und hinter den Ausführungsbefehl in das Skript, so kann die Zeit gemessen werden [5].
Hierbei ist es möglich [9], die Option `--byslot` zu benutzen (standardmäßig), um die Prozesse erstmal auf einen Knoten zu platzieren, bis die maximale Anzahl (auch per Argument festlegbar) erreicht ist; danach wird der nächste Knoten befüllt. Mit `--bynode` werden die Prozesse auf alle verfügbaren Nodes gleichmäßig aufgeteilt.
Unser Programm wird mit mpic++ komiliert und mit mpiexec oder mpirun (im Jobskript) aufgerufen.
Ein Vampir-Trace war nicht möglich, obwohl bereits `VT_BUFFER_SIZE=300M` und `VT_MAX_FLUSHES=6` gesetzt wurden.

4.2.1 Aufteilung des Bildes in feste Blöcke

Die Pixel des Bildes werden in Blöcke eingeteilt, die dann von den einzelnen Prozessen abgearbeitet werden, dabei hängt die Größe der Blöcke von der Anzahl der zur Verfügung stehenden Prozesse ab.
Wir benutzen Blöcke, die die gesamte Bildbreite einnehmen. Somit kann über die Zeilennummer ein Block adressiert werden.

```
143   int quotient, rest, max_proc;
144   quotient=h/world_size;
145   rest=h%world_size;
```

Abbildung 6: Blockberechnung

Das Zusammenführen über MPI_Reduce erfolgt durch eine selbst-definierte Funktion [4]:
Da die Blöcke disjunkt sind, liegt es nahe, dass wir die Daten einfach addieren. Dies spart dem Master das Einordnen der Zeilendaten.

```
138   MPI_Op_create((MPI_User_function *)&myProd, 1, &myOp );
```

Abbildung 7: Erstellen einer Reduktionsoperation

Die Funktion ist definiert durch:

```
79  void myProd( Vec *in, Vec *inout, int *len, MPI_Datatype *dptr ) {
80      int i; Vec c;
81
82  for (i=0; i< *len; ++i) {
83      c.x = inout->x + in->x;
84      c.y = inout->y + in->y;
85      c.z = in->z + inout->z;
86      *inout = c; in++; inout++;
87  }
88
89  }
```

Abbildung 8: benutzerdefinierte Reduktionsfunktion

Wir müssen aufgrund der Vektoren allerdings eine eigene Vektordatenstruktur in MPI definieren:

```
MPI_Datatype ctype;

/* Vektor*/
MPI_Type_contiguous( 3, MPI_DOUBLE, &ctype );
MPI_Type_commit( &ctype );
```

Abbildung 9: Vektordatenstruktur

Diese wird dann beim Aufruf von MPI_Reduce verwendet.
Allerdings steigt hierdurch der Speicherbedarf unnötig; Nur 1/N-tel wird von einem Prozess benutzt. Es ergibt sich folgender Speedup.

Bei 50 Strahlen pro Pixel benötigt das serielle Programm 102 Sekunden und das nach Aufteilung in Blöcke bei 8 Prozessen nur 18 Sekunden.

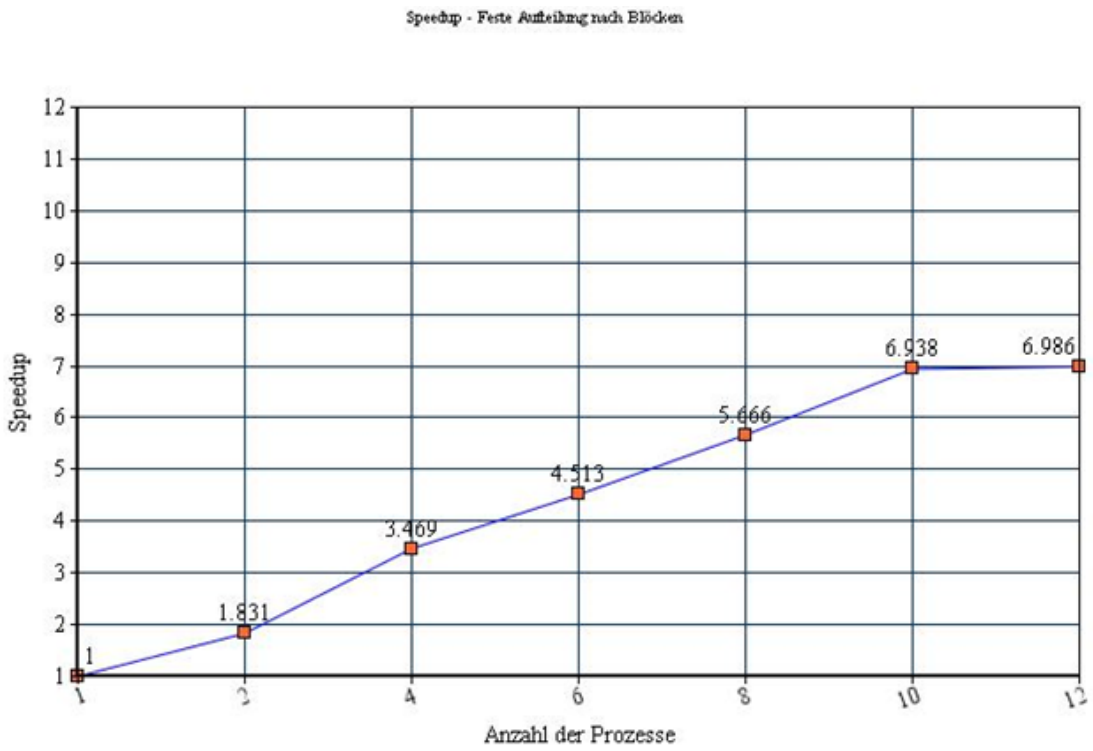

Abbildung 10: Speedup der festen Blockaufteilung

4.2.2 Aufteilung des Bildes in Zeilen

Die Pixel-Zeilen des Bildes werden den einzelnen Prozessen zugeordnet. Je mehr Prozesse zur Verfügung stehen, desto weniger Zeilen muss ein Prozess bearbeiten. Beim Aufruf von MPI_Reduce wird dieselbe Funktion wie bei der Aufteilung nach Blöcken verwendet.
Es ergibt sich folgender Speedup: Bei 50 Strahlen pro Pixel benötigt das serielle Programm 102 Sekunden und das nach Aufteilung in Zeilen bei 8 Prozessen nur 15,6 Sekunden.

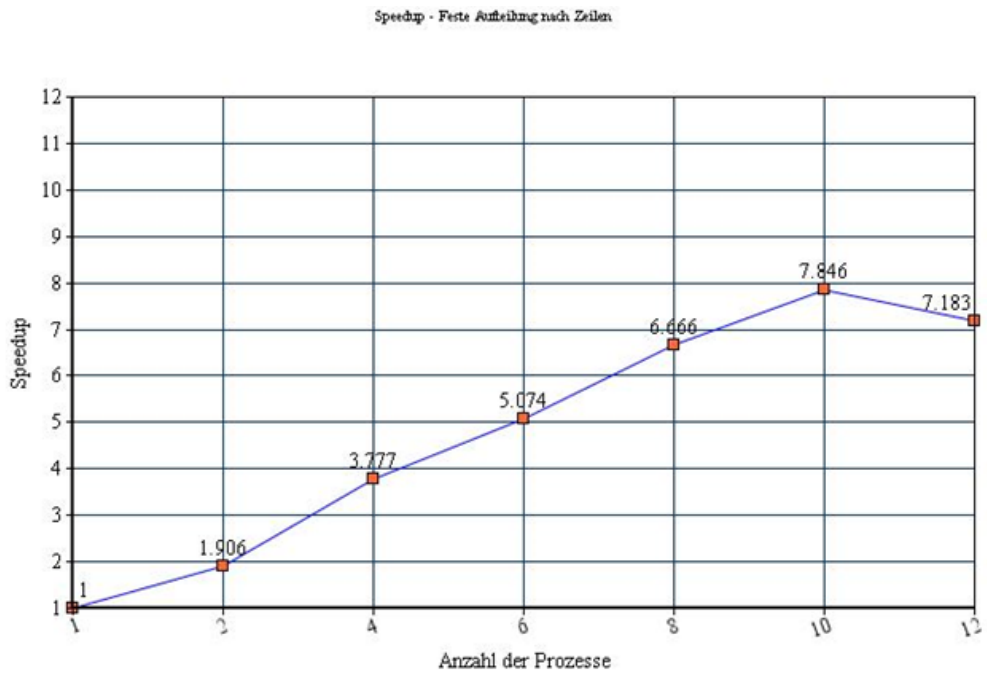

Abbildung 11: Speedup der festen Zeilenaufteilung

Am nachfolgenden Diagramm können wir die Ausführungszeiten der einzelnen Zeilen ablesen und vergleichen. Hierzu haben wir die Routine clock_gettime im seriellen Programm benutzt. Die Zeiten variieren zwischen neun und vierzehn Millisekunden. Dies ist insofern überraschend, weil der Unterschied einer rechenintensiven Zeile gerade einmal zusätzliche 50 Prozent beträgt.

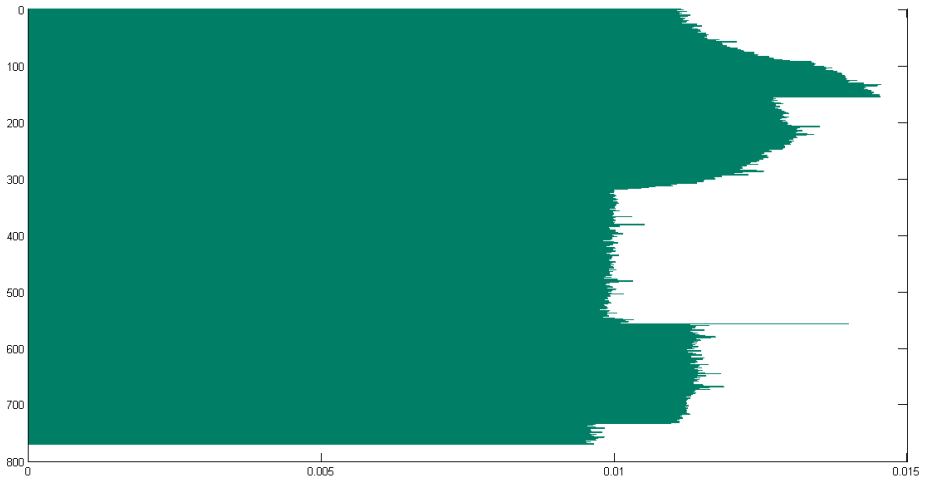

Abbildung 12: Benötigte Zeit in Sekunden für die einzelnen Zeilen

Man würde erwarten, dass die Zeilen mit der Licht reflektierenden und der brechenden Kugel bedeutend mehr Rechenleistung verlangt. Betrachten wir nun folgende Gegenüberstellung:

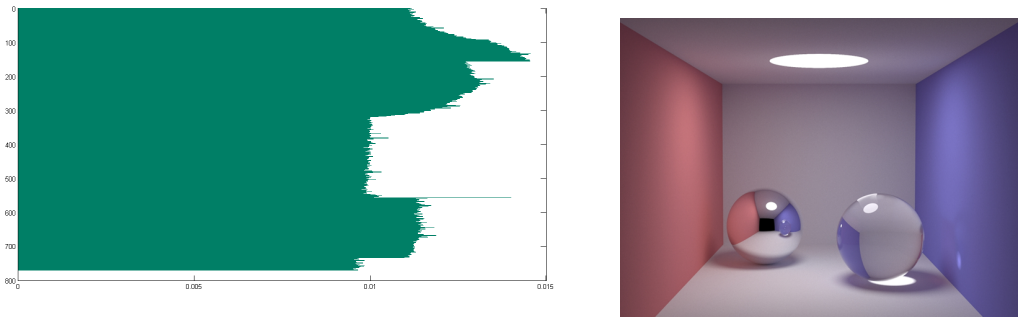

Abbildung 13: Benötigte Zeit in Sekunden für die einzelnen Zeilen

Wir sehen, dass nicht nur die Zeilen mit den Kugeln mehr Rechenleistung benötigen, sondern auch das erste Drittel des Bildes. In diesem Bereich sind es fünf verschiedene Kugeln, während im nicht rechenintensiven Teil nur drei Kugeln sichtbar sind.
Eben diese Bereich sind der Grund, warum die Aufteilung in Blöcke mehr Zeit benötigt als die Aufteilung nach Zeilen, wo ja die Bereiche relativ gemeinsam berechnet werden.

Wir könnten nach einem Durchlauf sofort die einzelnen Zeilen den Prozessen derart zuordnen, dass jeder Prozess gleich viel Arbeit bekommt und damit alle nahezu gleichzeitig fertig werden. Vor diesem Durchlauf ist es sehr, sehr schwierig, den Aufwand abzuschätzen; es ist quasi eine Blackbox. Es ist aber nicht Sinn der Sache, im Nachhinein eine bessere Performance zu erzielen. Deshalb haben wir dies nicht weiter verfolgt.

4.2.3 Warteschleife

Die Warteschleife stellt eine Art Load-Balancing zur Laufzeit dar.
Wir haben zwei potentielle Möglichkeiten der Umsetzung betrachtet:
Zum einen gibt es die Master-Slave-Warteschleife, zum anderen die Knoten-zu-Knoten-Kommunikation.
Bei der Knoten-zu-Knoten-Kommunikation teilt ein Prozess allen anderen mit, welche Zeile er als nächstes bearbeitet. Wenn somit ein Prozess eigentlich mit der vor der Ausführung zugewiesenen Arbeit, z.B. Aufteilung nach Zeilen oder Blöcken, fertig ist, schaut er sich an, wo die anderen gerade sind und übernimmt von hinten nach vorne einen Teil der Arbeit des langsamsten Prozesses.
Da hier eine Menge Kommunikation notwendig ist, haben wir diesen Ansatz später nicht fortgesetzt.

Bei der Master-Slave-Warteschleife dient ein Prozess als Master und die übrigen als Slaves. Der Master verteilt die Pixel bzw. wegen der Granularität die Zeilen an die Slaves und diese sind für die eigentliche Berechnung zuständig. Wenn ein Slave fertig ist, fordert er individuell neue Arbeit an. Es gilt somit das „First Come First Serve“-Prinzip.
Da die Ergebnis-Vektoren eines Slaves disjunkt zu Vektoren anderer Slaves sind, können wir abermals die obige Reduce-Funktion anwenden, um am Ende alle Bilddaten vom Master durch Addieren zusammenführen zu lassen.

Die jeweiligen count Parameter von MPI_Send und MPI_Recv der Slaves sind auf 1 begrenzt, so dass immer nur die Bereit-Meldung von einer Zeile vom Slave gesendet und daraufhin die Adresse zu einer neuen empfangen werden.

Es ergibt sich folgender Speedup.
Bei 50 Strahlen pro Pixel benötigt das serielle Programm 102 Sekunden und das mit Implementierung einer Warteschleife bei 8 Prozessen (inkl. Master) nur 16,5 Sekunden auf einem Knoten. Bei neun Prozessen ergibt sich: 15,3 Sekunden.

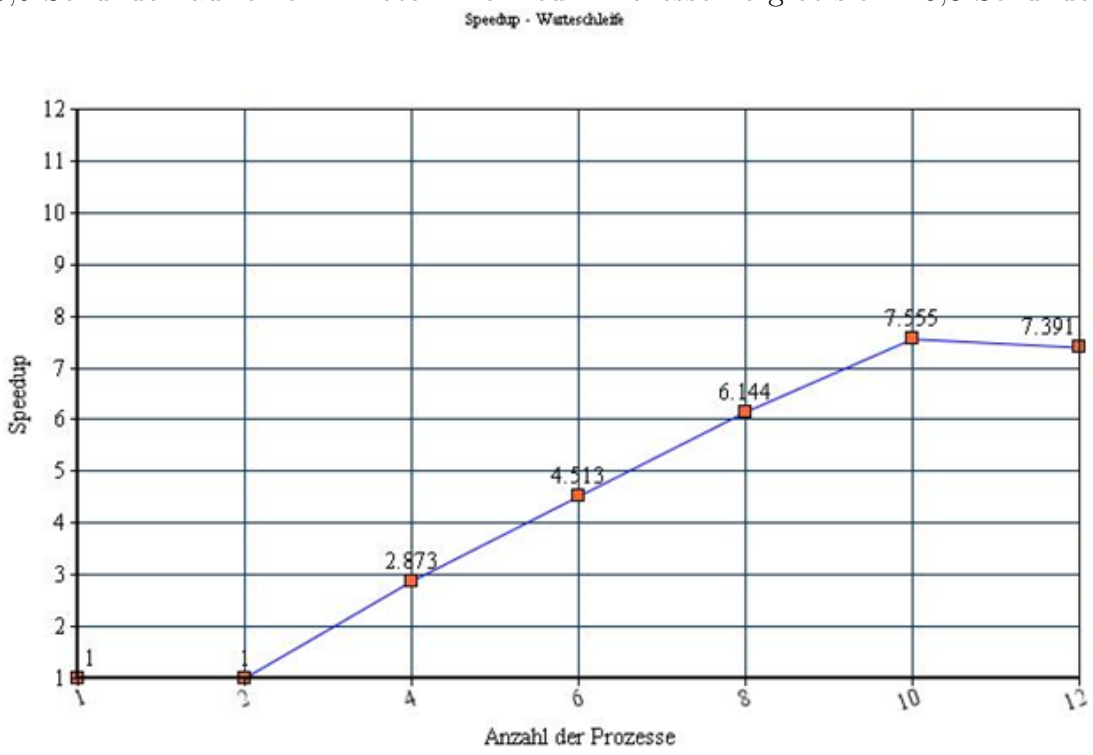

Abbildung 14: Speedup-Diagramm der Warteschleife

Obwohl der Masterprozess eigentlich mitgezählt werden muss, kann man im Vergleich mit den anderen Parallelisierungsschemata, bei denen ja alle Prozesse

direkt zur Berechnung beitragen, hier einen Prozess abziehen.

Parallelisierung	Anzahl Threads					
	2	4	6	8	10	12
Block	55,7	29,4	22,6	18,0	14,7	14,6
Zeile	53,5	27,0	20,1	15,3	13,0	14,2
Warteschleife	53,5	27,2	19,1	15,2	13,0	12,8

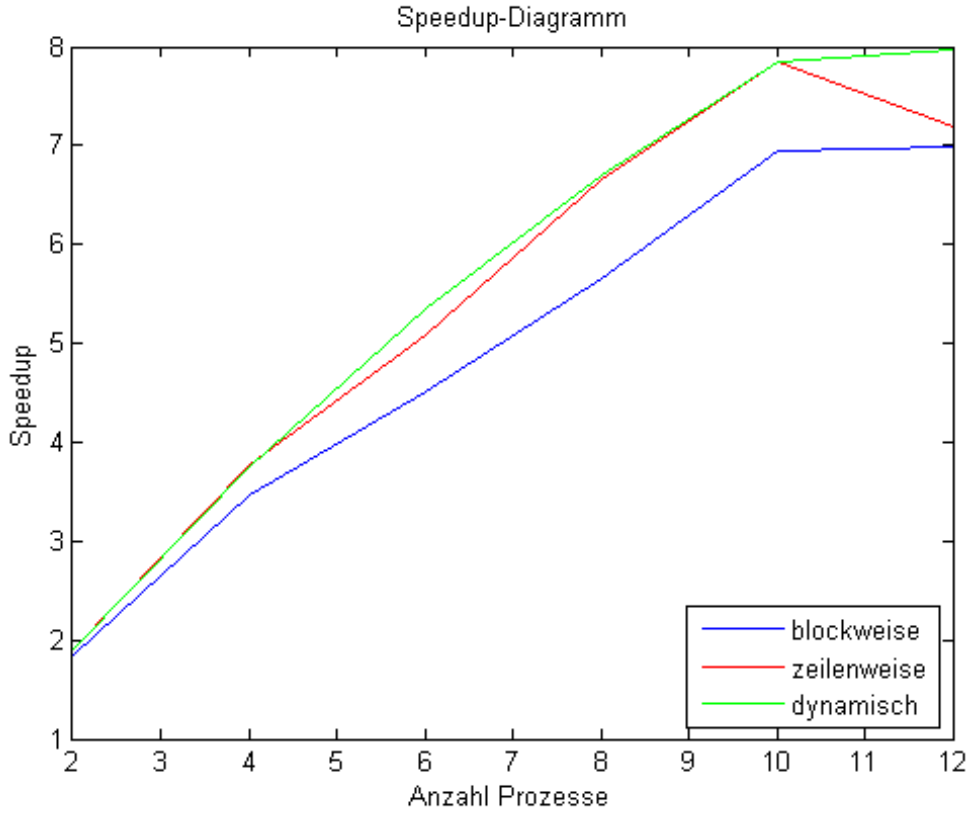

Abbildung 15: Speedup im Vergleich, wobei bei bei der Warteschleife nur die Slaves gezählt werden.

Wenn man nur die effektiv zur Berechnung beitragenden Prozesse zählt, so erzielt die Warteschleife einen höheren Speedup. Dies ist auch kein großes Hindernis, schließlich ist es möglich, dass wir den Master-Prozess nicht alleine auf einem Knoten auführen, sondern zusammen mit einem Slave unterbringen.

Der Vorteil der Warteschleife ist, dass sie bei allen Szenen gute Ergebnisse liefert. So können bestimmte Zeilen sehr viel Rechenleistung in Anspruch nehmen und gerade diese Zeilen bei den beschriebenen festen Aufteilungen größtenteils von demselben Prozess bearbeitet werden.

Insbesondere hat die Warteschleife ihre Vorteile bei heterogener Hardware-Leistung. Dies trifft in gewisser Weise auch auf Cluster zu, auf denen mehrere Benutzer gleichzeitig Programme ausführen lassen können (bei Slurm mit `\#SBATCH --share`) [7]. Unsere erste Implementation der Warteschleife hat noch Schwächen:

1. Der Slave-Prozess hat keine Arbeit, wenn er den Master nach einer berechneten Zeile benachrichtigt hat und um eine weitere bittet. Insbesondere bei mehreren Knoten ist dies wegen der langen Kommunikationswege ineffizient. Ein Test mit acht Slaves auf ebenso vielen Nodes brachte allerdings nur eine Verschlechterung im Vergleich zu einem Node um nur 0,2 Sekunden.

2. Es wurde immer bei jedem Prozess das ganze Bild alloziiert, weshalb der Arbeitsspeicher nicht optimal Ausgelastet wird: Es wird immer nur $\frac{1}{N-1}$-stel des reservierten Speichers beschrieben. In diesem Szenario hat dies zwar keine großen Nachteile, doch wollen wir im Allgemeinen keine Ressourcen verschwenden. Vorteilhaft ist, dass wir nur die Einzelbilder addieren müssen, damit wir das Gesamtbild erhalten. Sonst müsste der Master die Zeilen der Slaves zuordnen/in die richtige Reihenfolge bringen.

3. Die Granularität ist nicht perfekt.

Durch Berechnen von mehreren Zeilen kann das Problem der Granularität gelöst werden. Die Kommunikationslatenz kann behoben werden, indem per MPI_Isend und MPI_Recv vor dem Berechnen der letzten Zeilen die Fertig-Meldung gesendet wird, sodass dann hoffentlich (da es sich um eine Blackbox handelt) die Antwort mit neuen Zeilen rechtzeitig zum Ende des Berechnens eintrifft.

Wenn schon die abwechselnde Aufteilung nach Zeilen bereits relativ gut ist und die Slaves lange auf neue Arbeit warten, könnte man folgende Alternative umsetzen:
Wie oben wird mit MPI_Isend und MPI_Recv vor dem Berechnen der letzten Zeilen gearbeitet.
Anstatt immer nur ein paar Zeilen zuzuweisen, wird ein Vektor mit den zu berechnenden Zeilen vom Master an die Slaves übermittelt. Per MPI_Isend wird die jeweilige Iterationsgesamtzahl an den Master gesendet. Mit MPI_Recv kann der Master dann die Vektoren dynamisch so verändern, dass jeder Slave letztendlich gleich lange braucht.

4.2.4 Zufälliges Pixel

Es wird ein Seed mit Pixeln definiert, damit der Zufallszahlengenerator bei jedem Prozess unterschiedlich initialisiert wird. Jeder Prozess 'würfelt' sich einen zufälligen Pixel und übernimmt die Berechnung für diesen.

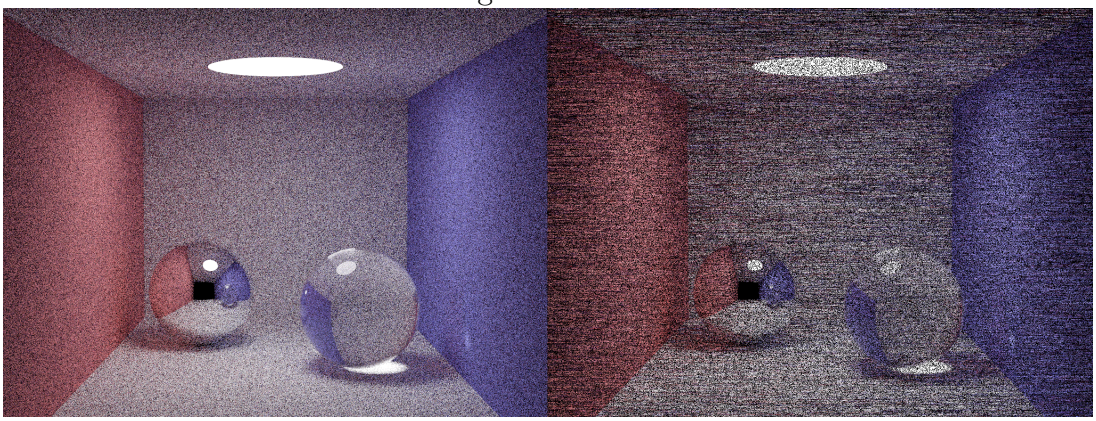

Abbildung 16: Gegenüberstellung: Zeilenversion links, Zufallspixelversion rechts

Anhand der beiden Bilder lässt sich ein Vergleich anstellen. Bei gleicher Ausführungszeit ist das von Zufallszahlen generierte Bild nicht nur weniger ausgefüllt, was zu erwarten war, da die Berechnung der Zufallszahlen ebenfalls Zeit benötigt, sondern man erkennt auch horizontale Streifen, was nicht für einen qualitativ guten Generator spricht.

```
168   /* initialize random seed: */
169   srand (time(NULL)+rank);
170   unsigned short  y;
171   unsigned short  x;
```

Abbildung 17: Random-Seed mithilfe der Prozessnummer

Beim Aufruf von MPI_Reduce wird bei diesem Parallelisierungsschema eine etwas andere Funktion aufgerufen, da man nun nicht einfach die Einzelbilder addieren kann, weil zwei Prozesse dasselbe Pixel berechnet haben können.

```
79  void myProd( Vec *in, Vec *inout, int *len, MPI_Datatype *dptr ) {
80      int i; Vec c;
81
82  for (i=0; i< *len; ++i) {
83
84   if ((inout->x)!=0 && (in->x)!=0)
85   {      c.x= inout->x;}
86  else { c.x = inout->x + in->x;
87   }
88
89   if ((inout->y)!=0 && (in->y)!=0)
90   {      c.y= inout->y;}
91  else { c.y = inout->y + in->y;
92   }
93
94
95   if ((inout->z)!=0 && (in->z)!=0)
96   {      c.z= inout->z;}
97  else { c.z = inout->z + in->z;
98   }
99      *inout = c; in++; inout++;
100  }
101
102 }
```

Abbildung 18: veränderte Reduktionsfunktion

Es muss geprüft werden, ob der x, y bzw. z Wert des Vektors noch 0 ist, also also zum ersten Mal berechnet wird oder nicht.

5 Video

Durch Einführung eines weiteren Terminalarguments, das die Position der reflektierenden Kugel und den Dateinamen beeinflusst, können wir per Shell-Skript (for i in ‘seq 0 100‘; do ...; done) eine Sequenz aus Einzelbildern erstellen, deren Hintereinanderausführung eine animierte Sinusbewegung der Kugel ergeben.
Mithilfe des Befehls mogrify (mogrify -format jpg *.ppm) aus ImageMagick können wir eine Batch-Konvertierung aller ppm-Bilddateien in Jpegs durchführen. Wir verschieben die Jpegs in einen Ordner und erstellen ein Archiv mit tar (tar -cf archiv.tar jpeg/). Über Scp laden wir dieses herunter. Dies ist nötig, da auf dem Server leider kein ffmpeg installiert ist, das eine Videodatei aus einer Sequenz aus Einzelbildern erstellen kann. Dieses haben wir letztendlich nicht verwendet, sondern ImageJ, wo wir verschiedene Frameraten gewählt haben. Die entstandenen Videos haben wir schließlich mit VLC in H.264 konvertiert.

6 Ausblick/Erweiterungen

Die Verwendung von SSE lässt einen großen Geschwindigkeitszuwachs erwarten.
Bei einer hybriden Parallelisierung aus MPI und OpenMP kann letzteres z.B. die Berechnung der Spaltenelemente innerhalb eines MPI-Prozesses parallelisieren.
Allerdings könnte man auch die MPI-Prozesszahl pro Knoten erhöhen; dann würde eine feinere Aufteilung der Zeilen den gleichen Rechen-Effekt haben, wohl aber mehr Arbeitsspeicher auslasten.
Der Intel C++-Compiler (icc) [6] soll gegenüber gcc besseres Loop-Unrolling haben und damit besser vektorisieren können.

Man kann recht einfach andere aus Kugeln aufgebaute Szenen verwenden. Auf der Webseite [1] von smallpt stehen andere Szenen bereit.

7 Fazit

Wir haben die grundlegende Funktionsweise von Raytracing zur Bildsynthese aus einer 3D-Szene kennengelernt.
Das Raytracing lässt sich sinnvoll parallelisieren. In dieser Ausarbeitung wurden mehrere Möglichkeiten der Parallelisierung vorgestellt: Zum einen ist dies die Parallelisierung mit OpenMP, welche einen fast linearen Speedup aufweist. Zum anderen ist es mit MPI die Aufteilung nach Zeilenbereichen (Blöcken), die abwechselnde Aufteilung der Zeilen und schließlich die Master-Slave-Warteschleife, bei der die Slaves nach beendigter Abarbeitung der Aufgaben den Master nach weiteren Aufgaben (Zeilen) fragen. Diese hat auch bei heterogener Hardware-Leistung Vorteile.

Von den MPI-Möglichkeiten bietet die Methode der Master-Slave-Warteschleife den stärksten Speedup. Es handelt sich um eine for-Schleifen-Parallelisierung. Der Speedup ist nicht komplett linear mit zunehmender Prozessanzahl, aber durchaus zufriedenstellend.
Im Allgemeinen ist die Master-Slave-Warteschleife ein sehr mächtiges Werkzeug für alle Anwendungen bei denen nicht unbedingt eine Kommunikation zwischen den durch die Slaves repräsentierten Bereichen oder Objekten existiert.

Die vorliegende Arbeit kann erweitert werden, beispielsweise durch Hinzunahme anderer Objekt wie etwa Dreiecken oder Zylindern. Dies erfordert allerdings neben den neuen Objektklassen an sich auch die Implementation der zugehörigen Gleichungen für die Abfrage des Schnittpunktes/der Schnittpunkte.

Weitere Optimierungsmöglichkeiten liegen darin, etwa die for-Schleife über die Spaltenelemente noch mit OpenMP zu parallelisieren oder etwa die Verwendung von SSE auf die Samples oder die Spaltenpixel.

Alles in allem haben wir bei der Parallelisierung des Raytracing gewinnbringende Techniken wie beispielsweise die Erstellung einer benutzerdefinierten Reduktionsfunktion und eines Datentyps kennengelernt.

Literatur

[1] Beason, K. : *smallpt: Global Illumination in 99 lines of C++*. https://docs.google.com/file/d/0B8g97JkuSSBwUENiWTJXeGtTOHFmSm51UC01YWtCZw/. Version: 2010. – [online; aufgerufen am 9-Juni-2014]

[2] Cline, D. : *smallpt: Global Illumination in 99 lines of C++ - presentation by Dr. David Cline*. https://docs.google.com/file/d/0B8g97JkuSSBwUENiWTJXeGtTOHFmSm51UC01YWtCZw/. Version: 2012. – [online; aufgerufen am 9-Juni-2014]

[3] DKRZ: *Batch Jobs (SLURM)*. https://www.dkrz.de/Nutzerportal-en/doku/cluster-wizard/batch-jobs. – [online; aufgerufen im April 2014]

[4] forum.org mpi: *106. User-Defined Reduction Operations*. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node107.htm. Version: 2009. – [online; aufgerufen im Juni 2014]

[5] HPC2N, U. U.: *Example job submission files*. https://www.hpc2n.umu.se/batchsystem/examples_scripts. – [online; aufgerufen im April 2014]

[6] Intel: *Intel C and C++ Compilers*. https://software.intel.com/en-us/c-compilers. Version: 2014. – [online; aufgerufen am 12-September-2014]

[7] Laboratory, L. L. N.: *sbatch*. https://computing.llnl.gov/linux/slurm/sbatch.html. Version: 2012. – [online; aufgerufen im April 2014]

[8] Novillo, D. : *Parallel Programming with GCC*. http://www.airs.com/dnovillo/Papers/rhs2006.pdf. Version: 2006

[9] OpenMPI: *FAQ:Running MPI jobs*. https://www.open-mpi.org/faq/?category=running#mpirun-scheduling. Version: 2014

[10] Wikipedia: *Path Tracing*. http://de.wikipedia.org/wiki/Path_Tracing. Version: 2014. – [online; aufgerufen am 7-May-2014]

Abbildungsverzeichnis