UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

NUMENUS

Adaptives Sampling Plug-In für RenderGin

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Alwin Wilbert

Erstgutachter:	Prof. Dr. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Dr. Oliver Abert (NUMENUS GmbH)

Koblenz, im Juni 2011

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.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Raytracing ist zwar mittlerweile eine alte Methode für Computergrafik, jedoch sind ihre Potentiale lange noch nicht erschöpft. Im Vergleich zu herkömmlicher Computergrafik (perspektivische Projektion) bietet Raytracing entscheidende Vorteile, die hauptsächlich in der vergleichsweise hohen physikalischen Korrektheit der Methode begründet sind. Die Schwächen liegen hingegen im immensen Rechenaufwand. Mit fortschreitender Zeit werden Prozessoren und andere Hardware-Komponenten jedoch immer leistungsfähiger, sodass der Punkt, an dem Raytracing die herkömmliche Computergrafik ersetzen kann, abzusehen ist. Um diesem Ziel näher zu kommen, ist es sinnvoll die Effizienz eines Raytracers zu maximieren. Eine Computergrafik muss vor allem echtzeitfähig sein, somit auch ein Raytracer. Ein Raytracer ist vergleichsweise so rechenintensiv, weil für jeden Pixel mindestens ein Strahl verschickt werden muss. Dieser muss gegen alle Objekte im Raum geschnitten werden. Hinzu kommen noch die Strahlen, die entstehen, wenn Strahlen an Objekten reflektiert werden (Rekursion).
Um diesen Rechenaufwand zu verkleinern und zusätzlich ein besseres Bild zu erzeugen, soll der adaptive Sampler den Raytracer unterstützen. Der adaptive Sampler soll während des Rendervorgangs den progressiven Fortschritt in der Bildgenerierung beobachten und Pixel von der weiteren Berechnung ausschließen, für die sich ein zusätzliches Verschießen von Strahlen nicht mehr lohnt. Also keine Änderung der Farbe eines Pixels durch das Verschießen von Strahlen zu beobachten ist. Anders als der reine progressive Raytracer verschießt der adaptive Sampler irgendwann keine Strahlen mehr, sondern hört mit dem Konvergieren des Bildes auf zu Rechnen. Der adaptive Sampler soll so dafür sorgen, dass schneller ein besseres Bild erzeugt wird und somit die Performanz gesteigert wird.
Insgesamt erwartet man sich vom adaptiven Sampler Vorteile bei der Berechnung von bestimmten Szenen. Unter anderem eine Verbesserung bei Szenen mit rein diffus beleuchteten Bildbereichen, sowie eine Verbesserung bei Szenen mit unterschiedlich rechenintensiven Bildbereichen.
Ein normaler Raytracer kann nicht beurteilen wie sinnvoll seine Schüsse sind. Er kann nur mehr Strahlen verschießen, in der Hoffnung das Bild damit effektiv zu verbessern. Es gibt jedoch viele Szenarien bei denen eine linear steigende Schussanzahl pro Pixel keine gleichmäßige Verbesserung im Bild erzeugt. Das bedeutet, dass Bereiche im Bild schon gut aussehen, währen andere noch sehr verrauscht sind. Man möchte nun Bildbereiche, die bereits konvergiert sind, in denen sich ein weiterer Beschuss also nicht mehr bemerkbar macht, ausschließen und die Rechenleistung dort nutzen wo man sie noch braucht.
Wichtig dabei ist, dass Pixel nicht ungewollt zu früh von der Berechnung ausgeschlossen werden, die nicht weit genug konvergiert sind. Der adaptive Sampler soll so lange arbeiten, bis jeder Pixel dauerhaft keine Änderungen mehr vorweist. Das bedeutet, dass die Wahrscheinlichkeit für eine signifikante Farbänderung eines Pixels durch verschießen eines Strahls (bei mehreren Lichtquellen in RendeGin mehrere Strahlen pro Pixel) klein genug ist.

Es wird zwar intern keine Wahrscheinlichkeit berechnet, jedoch bekommt der Raytracer eine Art Gedächtnis: Er speichert die Veränderungen im beleuchten Bild und deren Verlauf in eigenen Gedächtnisbildern. Man könnte es mit einem groben Langzeitgedächtnis und einem exakten Kurzzeitgedächtnis vergleichen. Das "Gedächtnis" für das alte Bild (Zustand des Bildes in der letzten Iteration über die Pixel) repräsentiert dabei das Kurzzeitgedächtnis. Es ist absolut genau. Das Langzeitgedächtnis wird von drei verschieden Bildern repräsentiert. Das erste gibt die Anzahl der verschossenen Strahlen pro Pixel an. Das zweite ist ein Wahrheitswertebild, das für jeden Pixel angibt, ob dieser noch in die Berechnung einbezogen werden soll. Das dritte Bild gibt an, wie oft jeder Pixel eine Farbänderung, die geringer als ϵ ist, vollzogen hat. ϵ ist dabei der geforderte Maximalabstand eines Pixels zu sich selbst (vor und nach dem Verschießen eines weiteren Strahls).
Mit diesen drei Bildern ist es möglich, zusätzliche quantitative Informationen zu den qualitativen Informationen des Vergleichs vom neuen und alten Bild zu berücksichtigen. Mit diesem "Gedächtnis" soll der adaptive Sampler seine Entscheidungen möglichst gut absichern können. Denn wir möchten ja einer Verbesserung hervorrufen und keine Fehler erzeugen. In dieser Arbeit kläre ich die Frage, ob die gewünschten Effekte eintreten und ob bei Integration in die bestehende Struktur von RenderGin ein Performazgewinn möglich ist.
Die Umsetzung eines adaptiven Samplers ist als Plug-In in der Software RenderGin von Numenus GmbH geschehen. RenderGin ist ein echtzeitfähiger, progressiver Raytracer, der sich durch seine Performanz auszeichnet. Die Bildgenerierung geschieht allein auf der CPU, die Grafikkarte wird lediglich zur Anzeige des erzeugten Bildes benötigt. Die Umsetzung und Programmierung des Plug-Ins ist in Microsoft Visual Studio 2010 geschehen unter Verwendung des RenderGin SDK der Numenus GmbH.

2 Motivation

Die Wahl dieses Themas erschien mir als eine sinnvolle, denn adaptives Sampling birgt die Möglichkeit zwei Fliegen mit einer Klappe zu schlagen. Einerseits konvergiert das Bild schneller und andererseits wird ein besseres Bild generiert, zumindest in der Theorie. Die Raytracing-Technologie konnte mich durch die Einfachheit der Technik, sowie ihren Realismus sofort begeistern. Zusätzlich ist das Potential von Raytracing noch lange nicht ausgeschöpft.
Für die endgültige Wahl des Themas war letztendlich aber auch ein konkretes Szenario verantwortlich: Oliver Abert, Gründer von NUMENUS und Begründer von RenderGin, zeigte mir anhand eines Modells eines Sportwagens, das in RenderGin geladen war, dass, obwohl die äußeren Teile des Autos nahezu keinerlei Fehler im Bild hatten, im Innenraum (dort wo nur diffuses Licht ankommt) das Bild immer noch sehr verrauscht war. Er erklärte mir, dass die Idee hinter dem adaptiven Sampler sei, die unnötig immer weiter verschossenen Strahlen der bereits konvergierten Bildregionen einzusparen, um die Rechenleistung in die Bereiche zu leiten,

die noch nicht konvergiert sind.
Es war direkt zu beobachten, dass der Bereich im Bild, der noch nicht konvergiert, also noch verrauscht war, in diesem Fall nur einen sehr geringen Teil des Bildes ausmachte. Das bedeutet, dass die meisten Strahlen völlig unnütz verschossen wurden. Dadurch, dass beispielsweise im Innenraum eines Fahrzeuges, nur diffuses Licht ankommt, werden solche Bereiche im Bild wesentlich bestrahlungsbedürftiger und benötigen mehr Samples zum konvergieren. Die Diskrepanz zwischen verrauschten und nicht verrauschten Bildregionen wirkt zusätzlich störend.
Der Adaptive Sampler erschien mir als ein einfacher und guter Lösungsansatz und somit als eine interessante Verbesserungsmöglichkeit für den Raytracer.

3 Begrifflichkeiten

Folgende Begrifflichkeiten werden verwendet:
Execution States - Zustände der Verarbeitungskette in RenderGin
Execution Chain - Alle Zustände zusammengefasst
RayGeneration - Zustand: Strahlgenerierung
Trace - Zustand: Strahlverfolgung
Shade - Zustand: Farbberechnung
Bilditeration - Einmaliges Durchlaufen aller Zustände für alle Pixel
Kriterien-Bild - Bild, das für jeden Pixel die Anzahl seiner ϵ-Unterschreitungen speichert
Bool-Bild - Bild mit einem Wahrheitswert pro Pixel
Zählbild - Bild, das für jeden Pixel die Anzahl seiner bisher verschossenen Strahlen speichert
Pack - Vier benachbarte Pixelpositionen im Bild, die mittels Intel SIMD-Befehlen parallel im jeweiligen Verarbeitungsschritt bearbeitet werden.

4 RenderGin

Für den adaptiven Sampler sind mehrere Teile des Raytracers wichtig. Einerseits das aktuell erzeugte Bild und andererseits die *RayGeneration* (Strahlgenerierung), *Trace* (Verfolgen) und *Shade* (Schattieren/Beleuchten) [AUGENBLICK2009]. Im RenderGin SDK ist das aktuell erzeugte Bild in einem Bildpuffer, der im Hauptspeicher des Computers gespeichert wird (siehe Abbildung 1). Um diesen Bildpuffer auszulesen muss man seine Startspeicheradresse abfragen und die im RGBA-Format gespeicherten Bytes nacheinander aus dem Speicher auslesen. Die RenderGin SDK beinhaltet drei Zustände, sogenannte *ExecutionStates* (siehe Abbildung 2). Diese beschreiben den Lebenszyklus eines erzeugten Strahls. Die *RayGeneragtion* ist der erste Zustand. In ihr werden 3D Rasterpositionen berechnet und von der Position der Kamera aus Strahlen zur jeweiligen Rasterpositionen erzeugt.
Diese Strahlen werden nun im zweiten Zustand *Trace* gegen alle Objekte im Raum

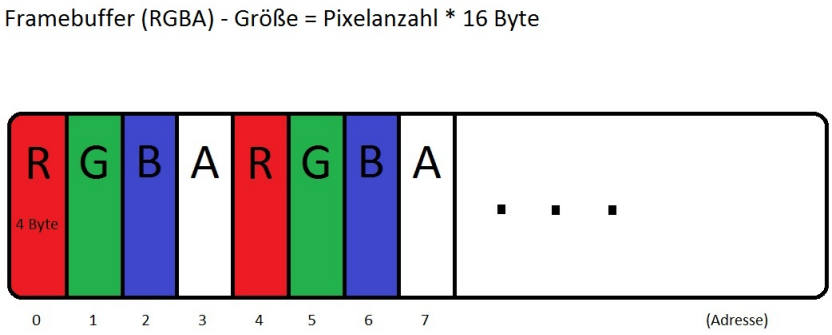

Abbildung 1: Speicheraufbau FrameBuffer

geschnitten, sodass durch Sortieren der Entfernung der Schnittpunkte zum Kameraursprung der erste Schnittpunkt gefunden wird.
Ist der vorderste Schnittpunkt gefunden, tritt der dritte Zustand das erste Mal in Kraft. *Shade* berechnet die Farbe des Strahls und anschließend wird ein reflektierter Strahl mit dem Ursprung vom aktuellen Schnittpunkt am Objekt innerhalb des Reflektionswinkels erzeugt und in den *Trace* Zustand versetzt, sodass der Strahl als sein reflektierter Rest erneut gegen alle Objekte geschnitten werden muss.
Die Farbe ergibt sich aus einer gewichteten Addition aller aufgesammelten Farbwerte. Der Strahl wechselt also so lange zwischen *Trace* und *Shade* hin und her, bis eine festgelegte Rekursionstiefe erreicht ist. Ist die Rekursionstiefe einmal er-

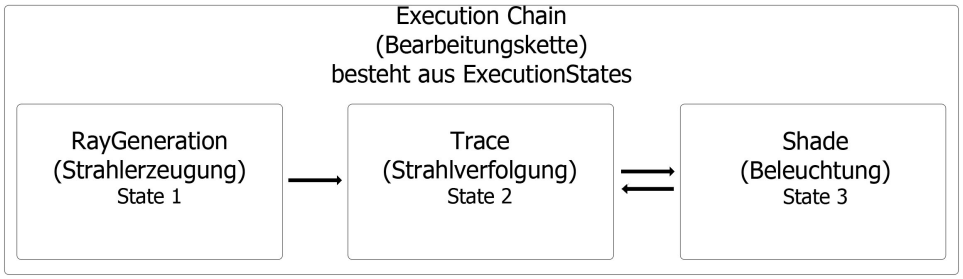

Abbildung 2: *Execution Chain* RenderGin

reicht, steht der aktuelle RGBA-Farbwert für einen Pixel fest. Dieser kann dann prozentual in den Bildpuffer (Siehe Abbildung 1) an die jeweilige Pixelposition addiert werden.
RenderGin arbeitet mit so genannten *Tiles*. Diese sind 32 * 32 = 1024 Pixel große Kacheln, aus denen ein ganzes Bild zusammengesetzt wird. Der Bildspeicher für einen *Tile* benötigt also 32 * 32 * 4 * 4 = 16384 (Byte)/ 4096 viele float-Werte um seinen Bildanteil zu speichern. Das *Tile* ist selbst aber noch in 16 * 16 = 256 sogenannte *RayPacks* (Strahlenpakete) unterteilt.

Diese Raypacks repräsentieren ein Bündel aus vier benachbarten Pixeln in einem *Tile*. Das *Pack* ist sozusagen die Recheneinheit von RenderGin. Sie sind das Bindeglied zu Intels *SIMD Technologie*. Diese ermöglicht es, innerhalb eines Speicherwortes vier verschiedene Werte zu speichern und diese parallel zu verarbeiten. RenderGin nutzt diese Technologie zur Parallelisierung der Rechenarbeit. Die

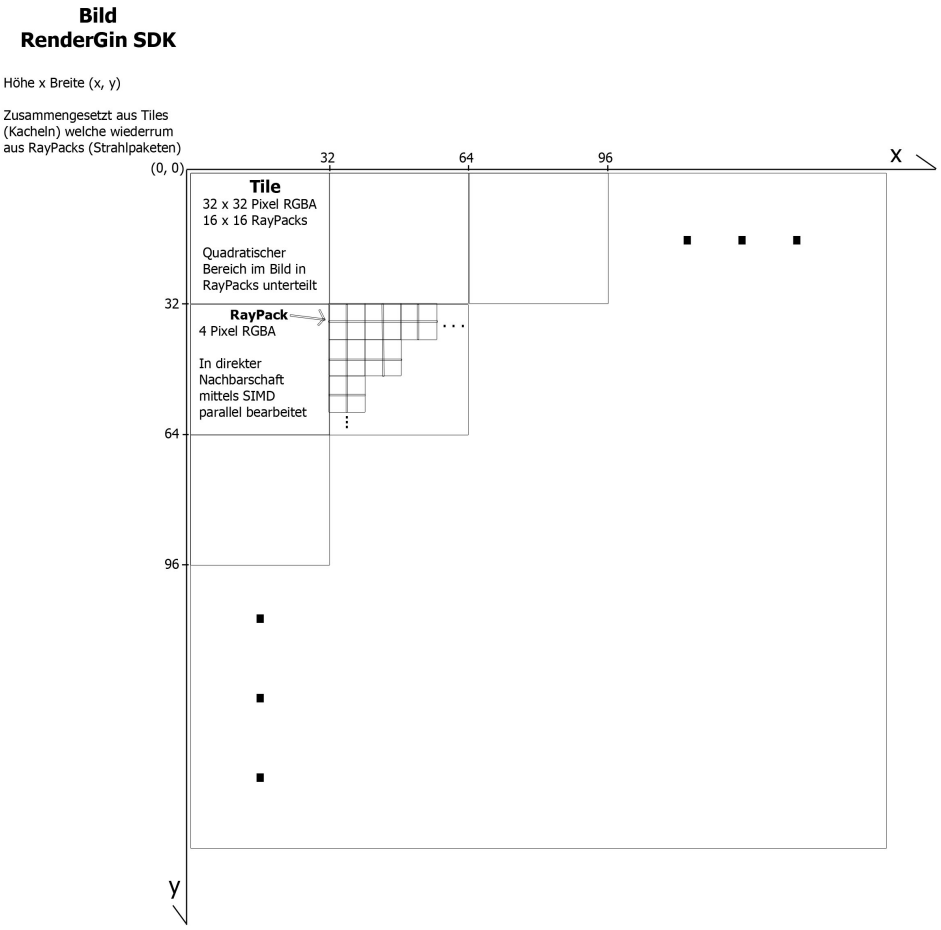

Abbildung 3: Bildaufbau RenderGin

Packs werden mit Hilfe von SIMD-Befehlen parallel in einem 64-bit-Wort verarbeitet. So können immer vier Strahlen auf einmal verarbeitet werden. Dies gilt für alle drei Zustände (siehe Abbildung 2). Das bedeutet, dass in allen Zuständen Paket weise iteriert wird, was wiederum bedeutet, dass im Vergleich zu einem Raytracer, der immer nur einen Strahl pro Speicherwort verarbeitet, nur etwa ein Viertel der Rechenvorgänge getätigt werden müssen.

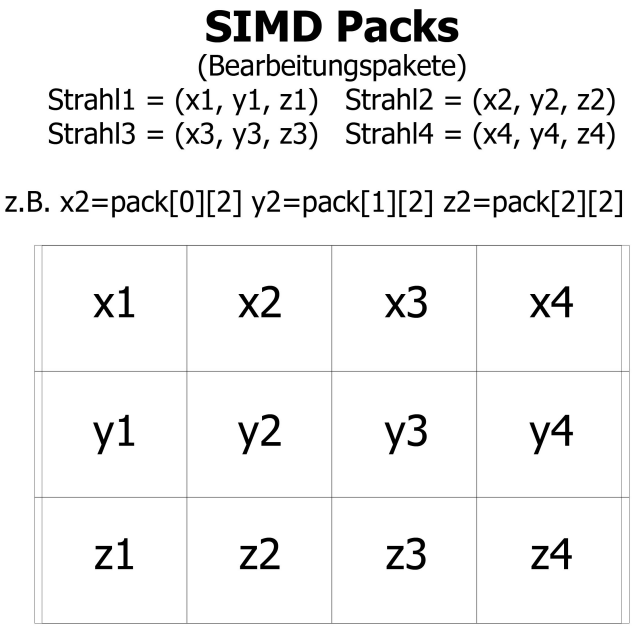

Abbildung 4: Pack (4 Strahlen [x,y,z])

RenderGin unterstützt Path-Tracing, was bedeutet, dass (ausser im Falle vollständiger Absorbtion) mit jedem verschossenen Strahl auch mindestens ein Zufalls-Reflektions-Strahl, erzeugt wird. In RenderGin ist aus Performanzgründen pro *Bilditeration* (für alle *Tiles* einmal Strahlen generiert, sowie *Trace* und *Shade*) nur eine Lichtquelle aktiv.
Wie schon erwähnt, ist RenderGin ein progressiver Raytracer, das bedeutet, dass der Farbwert eines Strahls nur prozentual (abhängig von der Anzahl der bisher verschossenen Stahlen für diesen Pixel) im Framebuffer vermerkt wird. Die Farbe eines Strahls wird in den Framebuffer addiert und anschließend wird der im Framebuffer nun befindliche RGBA-Wert wieder normiert. Dies passiert in jeder *Bilditeration* für jedes *Tile*.

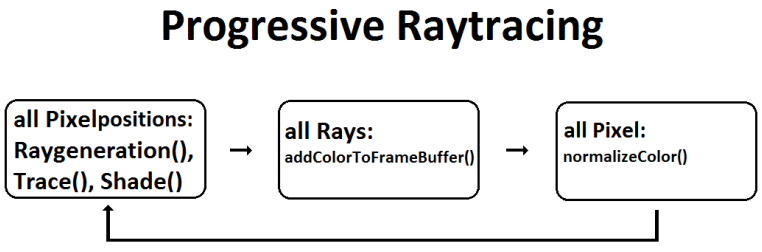

Abbildung 5: Progressiver Iterationsschritt

5 Das Konzept

Der adaptive Sampler soll in RenderGin als eine Reihe von Plug-Ins Realisiert werden. Ziel ist es, den Raytracer so zu gestalten, dass er nicht lohnenswerte Pixel auch nicht mehr weiter berechnet. Um sehen zu können, wo sich im Bild etwas verändert ist es sinnvoll ein Differenzbild zu berechnen. Dieses gibt für jeden Pixel den geometrischen Abstand zum vorherigen Pixel-Farbwert an. Dieses Differenzbild ist die Grundlage für die Bildauswertung und damit Grundlage des adaptiven Samplers [PBR2004].
Mit dem Differenzbild lassen sich verschiedene Heuristiken durchführen, die für adaptives Sampling interessant sind. Die Heuristiken können beliebig gewählt werden, sollten jedoch möglichst so gewählt sein, dass unerwünschte Effekte vermieden werden können.
Diese Effekte könnten beispielsweise sein, dass ein Pixel noch weiter berechnet wird, obwohl er es im Auge des Betrachters nicht weiter sollte. Noch schlechter ist der Fall, dass ein Pixel nicht weiter berechnet wird, obwohl er es im Auge des Betrachters noch sollte. Generell sind unerwünschte Effekte die unzureichende Berechnung von Pixeln oder Artefakte im Bild.

Die Vorgehensweise ist wie folgt gedacht:

1. Vor der Strahlgenerierung eine Maske berechnen, die für jeden Pixel seine Schussberechtigung speichert.

2. In der Strahlgenerierung nur die Strahlen berechnen, die eine Berechtigung haben. Alle anderen werden zu NULL-Rays.

3. In *Trace* nur die Strahlen verfolgen, die keine NULL-Rays sind.

4. In *Shade* nur die Pixel weiter berechnen, deren Strahlen keine NULL-Rays sind.

Im Falle, dass irgendwann alle Pixel konvergieren, also keine Änderungen mehr stattfinden, erzeugt, verfolgt und beleuchtet RenderGin nicht mehr. Es werden aber immer noch alle *Tiles* weiter überprüft, sodass der Render-Prozess nicht zum Erliegen kommt, jedoch weniger Leistung verbraucht.
Sobald sich Sicht oder die Szene verändert oder eine neuer Rendervorgang gestartet wird, werden alle Parameter und der Speicher des adaptiven Samplers auf den Initialzustand zurückgesetzt. Um die Information über den Bildabstand eines Pixels zu sich selbst über mehrere Iterationen des Raytracers berücksichtigen zu können, sind für den adaptiven Sampler zwei zusätzliche Bilder angelegt.
Das eine speichert, wie oft der Pixel schon geschossen wurde, und das andere Speichert, wie oft das gesetzte Kriterium erfüllt wurde. So können Ausreißer besser kontrolliert werden. Diese drei Kriterien (Abstand der Farbe des Pixels zu sich, Berechnungsanzahl pro Pixel und Anzahl der Kriteriumserfüllungen pro Pixel) sind die Basis der Auswertung und können jederzeit mit weiteren Kriterien kombiniert

werden.
Es ergeben sich die minimalen Parameter für den adaptiven Sampler: Epsilon (minimale Differenz des Pixel), n (minimale Schusszahl pro Pixel) und nEpsilon (minimale Kriteriumserfüllungen pro Pixel). Zu wissen wie oft ein Pixel ein gesetztes Kriterium erreicht hat, kann sinnvoll sein um Ausreißer zu eliminieren.

6 Die Umsetzung

Für die Umsetzung waren letztendlich vier zusätzliche Puffer nötig, um die Minimalfuntionalität für den adaptiven Sampler sicherzustellen.
Der erste enthält jeweils den Zustand des Bildes der letzten Iteration. Der zweite Puffer beinhaltet das Bild der Schussanzahl pro Pixel und der Dritte die Anzahl der Kriteriumserfüllungen pro Pixel. Der Letzte beinhaltet ein *Bool-Bild*, welches pro Pixel wahr oder falsch angibt.
Im Falle, dass dort *wahr* steht, soll der Pixel weiter berechnet werden. Im Falle, dass der Wert *falsch* ist, soll der minimale Rechenaufwand für diesen Pixel betrieben werden und dieser Pixel nicht weiter berechnet werden.
Die Auswertung des Bildes (Vergleich mit dem alten Bild), geschieht in der *RayGeneration*. Hier wird entschieden, ob ein NULL-Ray erstellt wird oder nicht. Die Zustände *Trace* und *Shade* bekommen nur über NULL-Rays mitgeteilt, dass ein Pixel/Strahl nicht weiter verfolgt oder beleuchtet werden soll.
Da RenderGin progressiv die Werte der Strahlen pro Pixel auf akkumuliert, muss in der Beleuchtung (*Shade*) sichergestellt sein, dass der alte Wert restauriert wird. Hierzu wird im Falle eines NULL-Rays einfach der Farbwert im FrameBuffer (Bild) durch den Farbwert im alten Bild ersetzt, sodass das Bild nicht abdunkelt wenn keine neuen Farbwerte in den Framebuffer addiert werden. Wenn dies einmal geschehen ist, also der Strahl ein NULL-Ray ist, wird die Berechnung der Farbe des zugehörigen Pixels fortan nur noch durch Zurücksetzen des alten Farbwertes realisiert.

Der adaptive Sampler hat 3 Zustände: (*Der erste und zweite Zustand sind initialisierende (vorbereitende) Zustände*)

1. Im ersten Zustand muss der adaptive Sampler warten, bis das erste Bild im Framebuffer (Bildpuffer) eingetragen ist. Das bedeutet, es müssen alle Pixel aktiv sein, sodass für jeden Pixel ein Primärstrahl erzeugt, verfolgt und beleuchtet werden muss.

2. Im zweiten Zustand des adaptiven Samplers kann nun, da bereits Farben im Framebuffer eingetragen sind, diese kopieren und den ersten Puffer (Speicher für den Speicherzustand des Framebuffers der letzten Iteration) damit füllen.

3. Im dritten Zustand haben wir den ersten Puffer gefüllt und können nun den Zustand des Bildes der aktuellen Iteration mit dem Zustand des Bildes der

letzten Iteration vergleichen und das Differenzbild berechnen. Anschließend wird der vierte Puffer (das *Bool-Bild*) überschrieben (vorher waren alle Werte im *Bool-Bild wahr*).
Dazu werden neben dem Differenzbild auch die Puffer zwei und drei hinzugezogen. Diese halten die Information über die Anzahl der bereits getätigten Schüsse pro Pixel und die Informationen über die Anzahl der Fälle, in denen die Differenz der Farben kleiner ist als Epsilon und die minimale Strahlanzahl pro Pixel verschossen wurde (erreichen des minimalen Kriteriums).
Mit Hilfe dieser Informationen (Differenzbild, Anzahl der verschickten Strahlen und Anzahl der Epsilonunterschreiungen nach dem Verschießen der minimalen Anzahl der Strahlen pro Pixel) kann nun entweder im *Bool-Bild* der Wert bei wahr belassen werden oder auf falsch gesetzt werden.

Diese Zustände sind nur von der Anzahl der bisher vollendeten Iterationen über alle Pixel abhängig (Anzahl der berechneten Frames). Im Falle, dass noch kein Bild fertig ist, ist der adaptive Sampler im Zustand eins. Im Falle, dass bereits ein Bild vorhanden ist, ist er im Zustand zwei und falls mehr als ein Bild gerendert wurde, ist er im Default-Zustand. Default bedeutet, dass alle Teilprozesse des adaptiven Samplers aktiv sind.

6.1 Multi-Threading

Da RenderGin Multi-Treading unterstützt, ist das Bild modular in *Tiles* (Kacheln) aufgeteilt die parallel in verschiedenen Threads abgearbeitet werden können. Aus diesem Grund ist die Rechenarbeit des adaptiven Samplers ebenfalls in *Tiles* aufgeteilt. Das bedeutet, dass seine Auswirkungen nur auf den Bereich eines von einem Thread ausgeführten *Tiles* beschränkt sind.
Der adaptive Sampler arbeitet also analog zur Bildberechnungsstruktur in RenderGin. Im Gegensatz zu einem globalen adaptiven Sampler (nicht auf den Bereich von Bildausschnitten (*Tiles*) beschränkt), konnte dieser Ansatz in die bestehende Struktur und Funktionsweise von RenderGin integriert werden.

6.2 Lichtquellen

Wie schon erwähnt benutzt RenderGin aus Performanzgründen pro Render-Parse (Renderdurchlauf über das gesamte Bild) nur eine Lichtquelle. Es sind also so viele Render-Parses durchzuführen, wie es Lichtquellen gibt, damit alle Pixel durch alle Lichtquellen gleichermaßen beeinflusst sind. Dies ist für den adaptiven Sampler sehr wichtig, da bei mehr als einer Lichtquelle davon auszugehen ist, dass zwei aufeinander folgende Render-Parses, aufgrund der für die Farbberechnung benutzten unterschiedlichen Lichtquellen, sehr verschiedene Ergebnisse erzeugen und damit eine große Differenz im Differenzbild erzeugen. Um dieses Flackern der Differenz zu unterbinden, wird der adaptive Sampler-Prozess nur aktiv, wenn gilt:

$$\#B \mod \#Lq = 0$$

Wobei gilt:
#B ist die Anzahl der absolvierten Render-Parses
#Lq die Anzahl der in Szene aktiven Lichtquellen

Es werden alle Pixel, die noch den Wert *wahr* im *Bool-Bild* haben, ohne Auswertung der Differenz erneut berechnet. Die Pixel werden erst wieder überprüft, wenn das Kriterium erfüllt ist und können nur in einem *Bilditeration* mit erfülltem Kriterium deaktiviert werden. So wird dafür gesorgt, dass die Farben der verschiedenen *Bilditerationen* mit verschiedenen Lichtquellen nur bemittelt in die Differenzbildung eingehen. Zusammengefasst wird der adaptive Sampler nur in Zuständen aktiv, wo das Bild von allen Lichtquellen in gleichem Maße beeinflusst wird.

6.3 Speicher

Der adaptive Sampler benötigt Speicher um den alten Zustand des Bildes zu speichern, sowie um sich zu merken wie oft er einen Pixel schon berechnet hat und wie oft ein Kriterium unterschritten wurde. Zusätzliche braucht er das *Bool-Bild* um abfragen zu können, ob ein Strahl berechnet werden soll. Der zusätzliche Speicheraufwand für den adaptiven Sampler beträgt in etwa:

$$\begin{gathered}\#P * (Palt + Pbw + Pba + Pk) \\ = \\ \#P * (16Byte + 1Bit + 4Byte + 4Byte) \\ = \\ \#P * 24\tfrac{1}{8}Byte\end{gathered}$$

Wobei gilt:
#P ist die Anzahl der Pixel der zu rendernden Szene
Palt ist der alte Pixel-Farbwert
Pbw ist der Bool-Wert des Pixels
Pba ist die Berechnungs-Anzahl des Pixels
Pk ist die Anzahl der Kriteriumsunterschreitungen des Pixels

7 Mathematik

Für den adaptiven Sampler sind verschiedene Berechnungen nötig. Berechnet werden müssen Differenzbild, *Bool-Bild*, die *Zählbilder* (Berechnungen pro Pixel , Kriteriumserfüllungen), Adressen, sowie Positionen im 3-dimensionalen Raum.

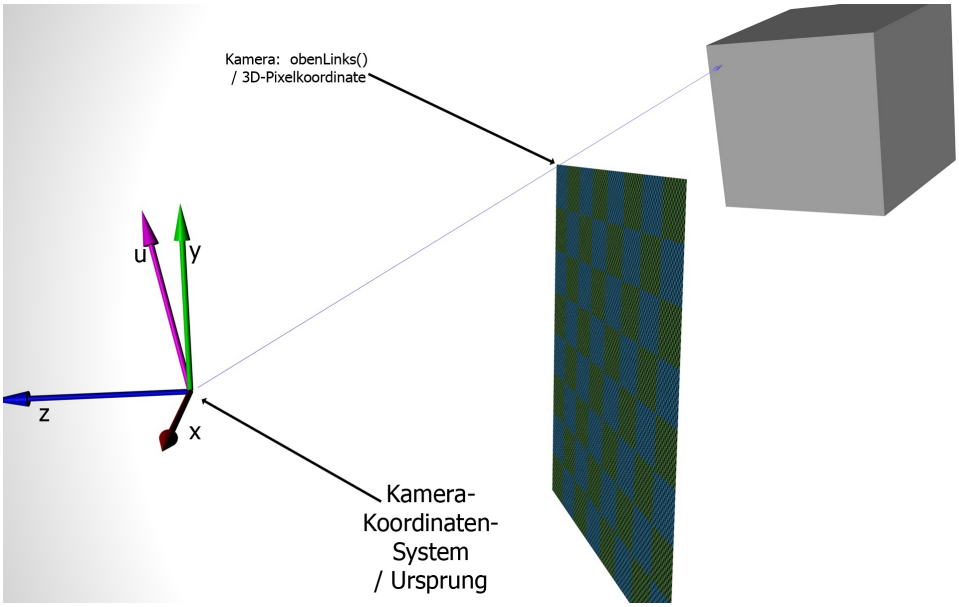

Abbildung 6: Primärstrahl (vom Kameraursprung zu 3D-Rasterposition ; schematisch)

7.1 Strahlgenerierung

In der *RayGeneration* müssen mittels der Kameraeigenschaften Strahlen generiert werden, die durch ihren Ursprung und ihre Richtung beschrieben sind. Hierfür sind vier Parameter der Kamera zu berücksichtigen ([CG2008]).
Der erste Parameter ist die Position der Kamera im 3-dimensionalen Raum.
Der zweite Parameter ist der Up-Vektor, er gibt an wo für die Kamera oben ist.
Der dritte Parameter ist die Blickrichtung der Kamera, ein Vektor der orthogonal auf der Bildebene steht.
Der vierte Parameter ist die 3D-Position des Pixels in der oberen, linken Ecke der virtuellen Bildebene.
Mit diesen Parametern kann die Berechnung der Strahlen durchgeführt werden [CG2008]:

1. Als erstes berechnen wir das Kamerakoordinatensystem der Kamera K: *($v_{BlickRichtung}(K)$ ist die negierte Z-Achse des Kamerakoordinatensystems, $v_{Oben}(K)$ ist der Up-Vektor der Kamera)*
   Z-Achse:
   $z = -1 * v_{BlickRichtung}(K)$
   $normalisiere(z)$
   X-Achse:
   $x = KreuzProdukt(v_{Oben}(K), z)$
   $normalisiere(x)$
   Y-Achse:
   $y = KreuzProdukt(z, x)$
   $normalisiere(y)$

2. Danach berechnen wir die Schrittvektoren, um über die 3-dimensionale Bildebene zu laufen:
   Schrittvektor v_X:
   $v_X = x/Bildbreite$
   Schrittvektor v_Y:
   $v_Y = -1 * y/Bildhhe$

   Analog ergibt sich der Schrittvektor für ein *Pack* durch Multiplikation der Schrittvektoren für einen Pixel (X, Y)mit zwei.

3. Anschließend wird die 3D-Position des *Tiles* berechnet *(obere linke Ecke des Tiles)*:

   $p_{Position}(Tile) = p_{ObenLinks} +$
   $2 * v_X * XpositionTile +$
   $2 * v_Y * YpositionTile$

   ObenLinks ist die 3D-Koordinate der linken, oberen Ecke der virtuellen Bildebene. XpositionTile und YpositionTile sind Skalare, die angeben wie oft der jeweilige Schrittvektor (X, Y) gegangen werden muss um an die gewünschte 3D-Position zu gelangen.

4. Nun, da alle vorbereitenden Berechnungen erledigt sind, können wir nun in *Packs* über das zu berechnende Tile laufen und die Strahlen generieren:
   (Hierfür ergeben sich die Parameter s und t, welche die x- und y-Position der Pakete repräsentieren)

   Berechnen der 3D-Paket-Position:
   $p_{Position}(Paket) = p_{Position}(Tile) + (2 * s * v_X) + (2 * t * v_Y)$

5. Wenn alle vorhergehenden Berechnungen erfolgt sind, kann nun für jedes Pixel im *Pack* ein Strahl (Position und Richtungsvektor) berechnet werden:
   (Rand() repräsentiert einen diskreten Zufallswert zwischen 0 und 1)

   Pixel 1 (oben links) - 3D-Rasterposition:
   $x_1 = x(p_{Position}(Paket) +$
   $Rand() * 2 * v_X +$
   $Rand() * 2 * v_Y)$
   $y_1 = y(p_{Position}(Paket) +$
   $Rand() * 2 * v_X +$
   $Rand() * 2 * v_Y)$
   $z_1 = z(p_{Position}(Paket) +$
   $Rand() * 2 * v_X +$
   $Rand() * 2 * v_Y)$

   Pixel 2 (oben rechts) - 3D-Rasterposition:
   $x_2 = x(p_{Position}(Paket) +$
   $(1 + Rand()) * 2 * v_X +$
   $Rand() * 2 * v_Y)$
   $y_2 = y(p_{Position}(Paket) +$
   $(1 + Rand()) * 2 * v_X +$
   $Rand() * 2 * v_Y)$
   $z_2 = z(p_{Position}(Paket) +$
   $(1 + Rand()) * 2 * X +$

$Rand() * 2 * Y)$

Pixel 3 (unten links) - 3D-Rasterposition:

$x_3 = x(p_{Position}(Paket) +$
$Rand() * 2 * v_X +$
$(1 + Rand()) * 2 * v_Y)$
$y_3 = y(p_{Position}(Paket) +$
$Rand() * 2 * v_X +$
$(1 + Rand()) * 2 * v_Y)$
$z_3 = z(p_{Position}(Paket) +$
$Rand() * 2 * v_X +$
$(1 + Rand()) * 2 * v_Y)$

Pixel 4 (unten rechts) - 3D-Rasterposition:

$x_4 = x(p_{Position}(Paket) +$
$(1 + Rand()) * 2 * v_X +$
$(1 + Rand()) * 2 * v_Y)$
$y_4 = y(p_{Position}(Paket) +$
$(1 + Rand()) * 2 * v_X +$
$(1 + Rand()) * 2 * v_Y)$
$z_4 = z(p_{Position}(Paket) +$
$(1 + Rand()) * 2 * v_X +$
$(1 + Rand()) * 2 * v_Y)$

Der Ursprung aller Strahlen ist der Kameraursprung. Nun kann aus Rasterposition und Kameraursprung die Strahlrichtung berechnet werden. Diese Richtung wird durch den Vektor von Kameraursprung zu 3D-Rasterposition eines Pixels beschrieben (die Subtraktion von 3D-Rasterposition und Kameraursprung).
Wenn das *Tile* komplett berechnet wurde, also alle Strahlen erstellt wurden, kann das *Tile* vom nächsten Zustand *Trace* weiterbearbeitet werden. Damit ist die Strahlengenerierung für ein *Tile* (Kachel) erledigt.

7.2 Jittering

Damit für einen Pixel nicht immer exakt die gleichen Strahlen generiert werden, ist es sinnvoll die 3D-Rasterpositionen der Strahlen zufallsbedingt innerhalb des Bereiches des Pixels (eine beschränkte Ebene im dreidimensionalen Raum) zu wählen. Damit ist es möglich, alle für einen Primärstrahl möglichen, diskreten Farben zu berücksichtigen und einen gemittelten Wert für die Farbe des Pixels zu erreichen.
Das sogenannte Streuen oder Stören (Jittering) des Pixels erzeugt ein deutlich besseres Bild als ohne Streuung der 3D-Rasterposition des Pixels. Zum Beispiel sorgt das Streuen der 3D-Rasterposition automatisch für Anti-Aliasing (Kantenglättung).

Dies geschieht auf Grund der Mittlung der Farbwerte der Primärstrahlen der verschiedenen 3D-Rasterpositionen eines Pixels. Da reflektierte Strahlen eine zufällige Richtung bekommen, werden diese durch das Streuen der 3D-Rasterposition eines Pixels nicht wesentlich beeinflusst.
Um diese Vorteile zu nutzen, wurde das Streuen (Jittering) auch in der Strahlgenerieung des adaptiven Samplers verwendet. Hierzu war es lediglich nötig die Rand()-Funktion zu implementieren, die eine beliebige rationale Zahl zwischen 0 und 1 erzeugt und diese wie oben im Abschnitt Strahlgenerierung beschrieben zu verwenden.

7.3 Auswertung

Befinden wir uns einmal im dritten und letzten Zustand des adaptiven Samplers, also mindestens zwei Render-Parses absolviert wurden, kann das Differenzbild berechnet werden. Ist dieses berechnet, kann, nach dem Verschießen der minimalen Anzahl der Strahlen pro Pixel und nach Erreichen der minimalen Anzahl der Erfüllungen des Kriteriums (Differenz kleiner als Epsilon), der Pixel von der weiteren Berechnung ausgeschlossen werden.
Das Ziel der Auswertung ist die Erstellung des *Bool-Bildes*. Hierfür ist eine Funktion implementiert, die den geometrischen Abstand zweier RGB-Farben wie folgt berechnet: *(A ist die Abstandsfunktion zweier RGB-Vektoren, F gibt die Farbe eines Pixels an, $B_n(P)$ ist das n-te Bild an der Stelle des Pixels P)*

$$A(F(B_{n-1}(P)), F(B_n(P)))) =$$
$$\| \begin{pmatrix} r_{n-1}(P) \\ g_{n-1}(P) \\ b_{n-1}(P) \end{pmatrix} - \begin{pmatrix} r_n(P) \\ g_n(P) \\ b_n(P) \end{pmatrix} \| = \| \begin{pmatrix} r_{n-1}(P) - r_n(P) \\ g_{n-1}(P) - g_n(P) \\ b_{n-1}(P) - b_n(P) \end{pmatrix} \|$$

In jeder Iteration über ein *Tile* wird nun der Zustand des Bildes im Einflussbereich des *Tiles* mittels dieser Funktion verglichen. Ist nun die minimale Strahlenanzahl pro Pixel erreicht und ist die Differenz der Farben klein genug, wird das *Kriterien-Bild* an der Position des Pixels erhöht. Ist die Farbdifferenz jedoch größer als gefordert, wird das *Kriterien-Bild* an der Position des Pixels verringert, sodass Pixel für jede Epsilonüberschreitungen zusätzliche Strahlen berechnen, verfolgen und beleuchten müssen.
Dies soll gewährleisten, dass Pixel mit häufiger Epsilonüberschreitung noch intensiver berechnet werden. Ist die Auswertung für alle Pixel im Tile geschehen und das *Bool-Bild* für das Tile gesetzt, kann die Strahlgenerierung beginnen. Die Auswertung verläuft wie folgt: *(ϵ ist der Schwellwert für die Farbdifferenz eines Pixels zweier aufeinander folgender Bilditerationen, $S(P)$ ist die Anzahl der verschossenen Strahlen für den Pixel P, n ist die minimale Anzahl der für einen Pixel zu verschießend Strahlen, $K(P)$ ist die Anzahl der Erfüllungen des ϵ-Kriteriums mit $S(P) > n$, n_ϵ ist die minimale Anzahl der Erfüllungen des ϵ-Kriteriums und*

$N_{links}(P), N_{rechts}(P), N_{oben}(P), N_{unten}(P)$ sind die Wahrheitswerte der Nachbarschaft des Pixels P aus dem Bool-Bild) den Strahlen, $K(P)$ ist die Anzahl der Erfüllungen des ϵ-Kriteriums mit $S(P) > n$, n_ϵ ist die minimale Anzahl der Erfüllungen des ϵ-Kriteriums und $N_{links}(P), N_{rechts}(P), N_{oben}(P), N_{unten}(P)$ sind die Wahrheitswerte der Nachbarschaft des Pixels P aus dem Bool-Bild)

1. Falls gilt:
   $A(F(B_{n-1}(P)), F(B_n(P)))) < \epsilon \wedge S(P) > n \Rightarrow$
   $K(P) = K(P) + 1$
   , sonst $K(P) = K(P) - 1$
2. Falls gilt:
   $K(P) > n_\epsilon \Rightarrow P = falsch$
3. Falls gilt:
   $N_{links}(P) = N_{rechts}(P) = wahr \vee$
   $N_{oben}(P) = N_{unten}(P) = wahr \Rightarrow$
   $P = wahr$

Nun wird nur noch bei den Teilen der Pixel ein regulär berechneter Strahl erzeugt, bei denen das *Bool-Bild* an der Position des Pixels den Wert *wahr* enthält. Alle anderen, also die Pixel deren *Bool-Bild* an der Stelle des Pixels den Wert *falsch* haben, bekommen NULL-Rays. Das bedeutet, dass der Richtungsvektor ein Nullvektor (Alle Komponenten des Vektors Null) wird. Der Nullvektor in der Richtung eines Strahls ist ein gut geeigneter Deaktivierungsmarker.
Für einen normalen Strahl kommt er ohnehin nicht in Frage, da er nicht fortgesetzt werden kann und auch in keinem Fall in einer normalen Strahlgenerierung erzeugt werden kann. Des weiteren ist er gut geeignet, weil der Zugriff auf Strahlen in allen Zuständen der *Execution Chain* von RenderGin möglich ist und die Information so gut von Zustand zu Zustand überführt werden kann. Ein NULL-Ray (R_0) hat also die folgende Gestalt:

$$R_0 = \left(\begin{pmatrix} x_{Ursprung} \\ y_{Ursprung} \\ z_{Ursprung} \end{pmatrix}, \begin{pmatrix} x_{Richtung} \\ y_{Richtung} \\ z_{Richtung} \end{pmatrix} \right) = \left(\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right)$$

Ist also die Strahlgenerierung abgeschlossen, können die Zustände *Trace* und *Shade* anhand der NULL-Rays erkennen, ob sie die Strahlen verfolgen und beleuchten sollen. Im Fall von *Shade* (dem Beleuchtungszustand) muss im Falle eines NULL-Rays jedoch noch der Farbwert des Pixels aus dem alten Bild zurück in den Framebuffer (Bildspeicher) kopiert werden, damit das Bild im Zuge der Farbnormierung nicht dunkler wird.

RenderGin addiert die Farben der Strahlen eines Pixels lediglich in den Bildspeicher, sodass die Farbe nach jeder Addition normiert wird. Dies führt dazu, dass eine solche Regelung nötig ist. Würde der Bildspeicher nicht in einem Post-Prozess weiter geändert, würde dieser Schritt entfallen. Anschließend ist dieser Pixel konvergiert. Er kann nicht, bevor ein neuer Rendervorgang gestartet wird, wieder in die Berechnung einbezogen werden. Gilt dies für alle Pixel ist das gesamte Bild konvergiert (es finden keine "Änderungen mehr statt) und der Raytracer legt seine Arbeit nieder.

7.4 Positions- und Adressberechnungen

Da wir das Bild in RenderGin mittels *Tiles* (Kacheln) und *Packs* (Paketen) aufbauen, ist es für den adaptiven Sampler sehr wichtig zu wissen, wo er sich gerade im Bild befindet. Hierzu muss die Tile-Position in Pixelkoordinaten, sowie die *Pack*-Position innerhalb des *Tiles* in eine eindimensionale Bildkoordinate umgerechnet werden. Diese eindimensionale Position dient der Zuordnung des jeweiligen Pixels im aktuellen *Pack* und Tile zur Position in den jeweiligen Puffern (Puffer für das alte Bild, Puffer für das *Bool-Bild*, Puffer für das Bild mit der Strahlenanzahl pro Pixel, sowie Puffer für die Epsilonunterschreitungen).
Je nach dem um welchen Puffer es sich handelt, muss die Position noch mit dem Faktor vier multipliziert werden, um die richtige Position zu erhalten. Dies ist der Fall, wenn es sich um das Format RGBA (float Rot, float Grün, float Blau, float Alpha) handelt. Da pro Pixel vier Float-Werte im Speicher stehen, ist die Position im Speicher auch vier mal so groß. Der Framebuffer und der Uploadbuffer (aktueller Bildspeicher und Bildspeicher des alten Bildes) haben dieses Format und sind dem entsprechend mit dem Faktor vier zu adressieren.
Für den adaptiven Sampler waren diese Berechnungen in zwei Zuständen der RenderGin Execution Chain nötig. Einmal in der Stahlgenerierung (*Raygeneration*) und in der Beleuchtung (*Shade*). In der Stahlgenerierung, wo die meiste Arbeit des adaptiven Samplers passiert, ist die Position für den Zugriff auf alle Puffer nötig. In der Beleuchtung nur für den Zugriff auf das alte Bild (Im Falle, dass ein NULL-Ray behandelt wird, muss die alte Farbe restauriert werden).
Da wir innerhalb der *Tiles* über *Packs* iterieren und mit Hilfe derer und SIMD-Befehlen vier Strahlen gleichzeitig behandeln, ist es sinnvoll die Adressfunktion so zu gestalten, dass für die Unterscheidung der vier Pixel im *Pack* ein Flag in die Funktion mit übergeben wird. Der Grund dafür ist, dass die oberen beiden Pixel im *Pack* im Bildpuffer nicht hintereinander liegen, sondern mindestens den Abstand der Pixelanzahl in der X-Richtung des Bildes haben.
Die Adressierungsfunktion lautet wie folgt: *(tileX ist die X-Pixelkoordinate des linkest, oberen Pixels des Tiles, TileY analog die Y-Pixelkoordinate, s,t sind die X- und Y-Koordinaten der Packs (Pakete) und width ist die Bildbreite in X-Richtung des zu rendernden Bildes)*

$$p_{Position}(tileX, tileY, s, t, flag, width) =$$

1. Fall (flag = 0):

$$p_{Position} = tileX + s * 2 + width * tileY + t * 2 * width$$

2. Fall (flag = 1):

$$p_{Position} = tileX + s * 2 + width * tileY + t * 2 * width + 1$$

3. Fall (flag = 2):

$$p_{Position} = tileX + s * 2 + width * tileY + t * 2 * width + width$$

4. Fall (flag = 3):

$$p_{Position} = tileX + s * 2 + width * tileY + t * 2 * width + width + 1$$

7.5 Zusätzliche Kriterien

Um die Entscheidung über die Deaktivierung eines Pixels nicht nur anhand der Farbdifferenz und zu fällen, ist in der Endphase der Bachelorarbeit noch ein weiteres, zusätzliches Kriterium hinzu gekommen. Dieses Kriterium besagt, dass, damit ein Pixel deaktiviert wird, gelten muss: *(a, b, c und d sind die Wahrheitswerte der direkt benachbarten Pixel aus dem Bool-Bild)*

$$(a \vee b = falsch) \wedge (c \vee d = falsch)$$

Somit ist sichergestellt, dass ein Pixel nicht deaktiviert werden kann, wenn dieser nur ein Ausreißer in einer homogenen Region (im *Bool-Bild*) ist. Sobald die Wahrheitswerte im *Bool-Bild* der Pixel in der Nachbarschaft eines beliebigen Pixels in X- oder Y-Richtung gleichermaßen wahr sind, muss der Pixel weiter berechnet werden.
Ohne dieses Kriterium ist es in manchen Szenen möglich, dass in inhomogenen Bildbereichen (gerendertes Bild) einzelne Pixel frühzeitig deaktiviert werden, und somit dort eine art ”Salz und Pfeffer Rauschen” entsteht.

7.6 Strahlverfolgung (*Trace*)

Trace ist der Zustand in dem die Strahlen verfolgt werden. Dafür muss der vorderste Schnittpunkt (sofern er existiert) mit der Geometrie der Szene gefunden werden

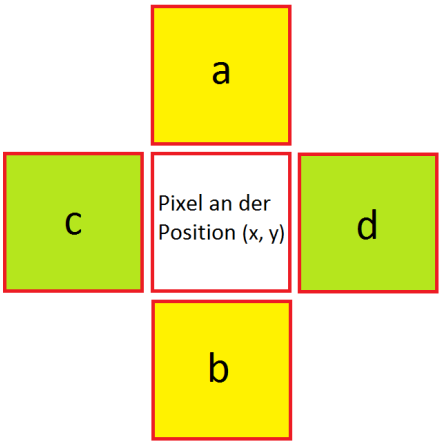

Abbildung 7: Ein beliebiger Pixel und seine Nachbarschaft

und im Falle keiner vollständigen Absorbtion des Strahls müssen Sekundärstrahlen emittiert werden.
RenderGin bietet von Haus aus die Möglichkeit ihre *Trace* Methode mit einer Maske zu versehen. Da wir über Pakete (*Packs*) iterieren, wird also eine Maske für vier Strahlen in die Funktion übergeben. Es werden nur die Pixel im Paket auf *wahr* gesetzt, die folgende Eigenschaft erfüllen: *(R_x, R_y und R_z sind die X-, Y- und Z-Koordinate der Richtung eines Strahls S)*

$$R_x(S) \neq 0 \vee R_y(S) \neq 0 \vee R_z(S) \neq 0$$

Nun berechnet die Trace-Methode von Rendergin nur noch die gewünschten Pixel im Paket. Falls diese Eigenschaft für alle Pixel im Paket zutrifft, wird die Trace Methode von RenderGin nicht mehr aufgerufen.

7.7 Beleuchtung (*Shade*)

Shade ist der Zustand, in dem ein Strahl eine Farbe zugewiesen bekommt. Hierzu werden mit Hilfe des Strahls, dem Schnittpunkt mit der Geometrie, Farbe des Schnittpunktes, den Materialeigenschaften des Schnittpunktes, sowie einer Lichtquelle (pro Iteration) die Farbe des Strahls im aktuellen Rekursionsschritt berechnet und in die (zusammengesetzte) Farbe des Primärstrahls einberechnet. Diese wird anschließend wieder in die aus vielen Primärstrahlen zusammengesetzte Farbe des Pixels einberechnet, zu dem die Strahlen gehören.
Im Falle, dass wir es jedoch mit einem NULL-Ray zu tun haben, wird die Position (1D im Bildspeicher) mittels der in Sektion 6.4 (”Positions- und Adressberechnungen”) beschriebenen Funktion berechnet. Nun wird der Pixel im Bild wie folgt gesetzt:

$$B_n(P) := B_{n-1}(P) \Leftrightarrow \begin{pmatrix} r_n(P) \\ g_n(P) \\ b_n(P) \end{pmatrix} := \begin{pmatrix} r_{n-1}(P) \\ g_{n-1}(P) \\ b_{n-1}(P) \end{pmatrix}$$

Nun ist sichergestellt, dass das Bild nicht abdunkelt, nachdem keine neuen Farbwerte in den Speicher addiert werden. Dies ist der letzte Schritt des adaptiven Samplers, bevor er die nächste Iteration über das Bild beginnt.

8 Programmablauf

Der adaptiver Sampler geht nach Starten eines neuen Rendervorgangs in RenderGin immer gleich vor (Abbildung 8):

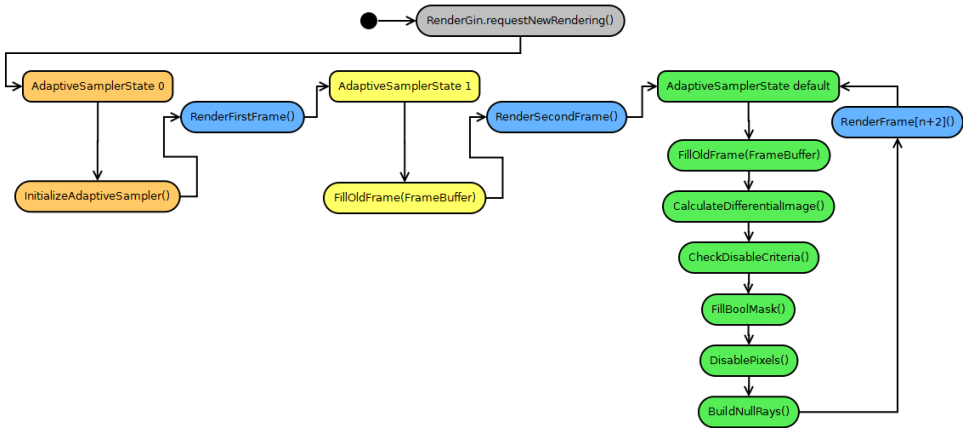

Abbildung 8: Schematisierter Programmablauf

Falls noch kein Bild berechnet wurde (AdaptihveSamplerState 0), wird der adaptive Sampler initialisiert. Wenn das erste Frame im Bildspeicher steht (AdaptiveSamplerState 1), wird dieses kopiert um es im nächsten Durchlauf (Iteration über das Bild) mit dem Folgezustand des Bildspeichers zu vergleichen.
Ab dem zweiten Frame (AdaptiveSamplerState default), vergleicht der adaptive Sampler nun Alt mit Neu (Bild) und merkt sich die Differenzunterschreitungen pro Pixel, und außerdem, wie oft für ihn ein regulärer Strahl verschossen wurde. Zusätzlich wird die Nachbarschaft des Pixels im *Bool-Bild* bezüglich ihrer momentanen Aktivität untersucht.
Sind alle Kriterien nun erfüllt, wird im *Bool-Bild false* eingetragen. Nun wird in der *RayGeneration* nur derjenige Strahl regulär erzeugt, der einen ”wahr” Wert im *Bool-Bild* haben. Für alle anderen werden NULL-Rays erzeugt, welche von der

Strahlverfolgung und Beleuchtung (*Trace* und *Shade*) entsprechend nicht oder gesondert behandelt werden. Der adaptive Sampler wird erst wieder neu gestartet, sobald wieder ein neuer Rendervorgang gestartet wird.

9 Test und Vergleich

In dieser Sektion sollen die erarbeiteten Plugins in verschiedenen Umgebungen getestet werden. Dabei gibt es zwei verschiedene Kriterien, nach denen der adaptive Sampler bewertet wird. Einerseits die Performanz und andererseits die Qualität des Raytracers.
Für die Testphase war eine Veränderung von RenderGin notwendig, die es erlaubt, Environment-Maps korrekt darzustellen. In Folge der Aktualisierung hat sich allerdings ein sporadischer Fehler eingeschlichen, der sich nur durch Multitasking (mehr als ein Render-Thread) erzeugen lässt. Vor der Aktualisierung, wahr jedoch kein solcher Fehler zu beobachten. Aus diesem Grund werden Tests bezüglich der Qualität nur mit einem Thread durchgeführt. In Performanztests können die Bilder jedoch mit mehreren Threads gerendert werden.
Getestet werden verschiedene Szenen (Audi A8, Dragon, Buddha, Menger Sponge) mit unterschiedlichen Materialien/Shadern (Phong, Fresnel, Torrance Sparrow, Oren Nayar, Spiegel) und verschiedenen Umgebungen/Environment Maps. Alle Tests werden bei einer Auflösung von $512 * 512$ Pixeln durchgeführt. Das Testsystem ist wie folgt konfiguriert: Prozessor: Intel i5 460 Mobile (2 Kerne, 4 Threads), Chipsatz: Intel HM55, RAM: 4096 MB DDR3 1333MHZ, Grafik: ATI Radeon 5470, Betriebssystem: Microsoft Windows 7 64 Bit Professionional, Testumgebung: Microsoft Visual Studio 2010 Ultimate. Für Vergleiche mit mehreren Threads werden analog zum Prozessor vier Threads verwendet. Die folgende Tabelle gibt an, welche Parameter für welche Shader (Materialien) verwendet werden.

Phong Shader	Diffuse Farbe	Spiegelnde Farbe	Ambiente Farbe	Glanzzahl	Transparenz
	$\begin{pmatrix}0.5\\0.2\\0.7\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\\1\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\\0\end{pmatrix}$	64	0

Torrence Sparrow	Glanzfarbe	Reflektionsindex	Absorbtionskoeffizient	Blinn-Exponent
	$\begin{pmatrix}0.8\\0.8\\0.8\\0.8\end{pmatrix}$	$\begin{pmatrix}0.16\\0.4\\1.6\\0\end{pmatrix}$	$\begin{pmatrix}4.4\\2.5\\1.9\\0\end{pmatrix}$	500

Oren Nayar	Diffuse Farbe	σ
	$\begin{pmatrix}0.8\\0.4\\0.4\\1.0\end{pmatrix}$	80

Fresnel Shader	Brechungsindex
	1.4

Spiegel	Reflektionsindex
	1

9.1 Parametervegleich

Diese Sektion beschreibt, wie sich die Parameter für den adaptiven Sampler auf das Bildergebnis auswirken. Alle verglichenen Bilder dieser Sektion sind konver-

giert, außer ihre Berechnungszeit geht über eine Stunde hinaus. Das bedeutet, der adaptive Sampler hat für alle Pixel entschieden, diese nicht weiter zu berechnen.
Es wird untexturiert getestet, mit einem Material und mit Material und Umgebung/Environment Map. Die folgende Bildreihe (neun Bilder) soll die Auswirkungen der Parameter des adaptiven Samplers auf das erzeugte Bild deutlich machen. Es werden pro Umgebung drei unterschiedliche Parametereinstellungen verglichen. Die Parameter sind in unten stehender Tabelle abgebildet.

Abbildung	ϵ	n	n_ϵ
9, 12, 15	0,1	3	1
10, 13, 16	0,001	5	5
11, 14, 17	0,0001	5	50

ϵ ist die zu unterschreitende Farbdifferenz. n ist die minimale Anzahl der Berechnungen (Strahlen) pro Pixel. n_ϵ ist die Anzahl der minimalen ϵ-Unterschreitungen.
Die erste Reihe (Erste drei Bilder der Reihe) wurden ohne Boden und ohne Material gerendert. Der Hintergrund ist weiß. In der zweiten Reihe (sechstes bis achtes Bilder der Reihe) ist der Drache wieder ohne Material, jedoch mit Umgebung (Environment Map) gerendert worden. Die dritte Reihe (siebtes bis neuntes Bild) zeigt den Drachen mit einem Phong-Shader (Material) und der bereits verwendeten Environment Map.
Zu beobachten ist, dass mit zu niedrig gesetzten Parametern unschöne schwarze Pixel entstehen können. Diese wurden zu früh nicht weiter berechnet, sodass eine Diskrepanz zu ihren Nachbarn entstand. Das Modell ist eine Drachenfigur aus dem ”Stanford 3D-Scanning Repository”. Er besteht aus 437,645 Vertices und 871,414 Dreiecken:

9.1.1 Beschreibung

In Abbildung 9, 12, 15 ist deutlich zu sehen, dass zwar der Hintergrund/Environment Map bereits ausreichend abgetastet ist. Das Modell ist jedoch noch sehr stark verrauscht (homogenes Rauschen und ”Salz und Pfeffer”-Rauschen). Viele Pixel sind zu früh von der Berechnung ausgeschlossen, sodass das ohnehin vorhandene Rauschen auf dem Modell von nahezu schwarzen Pixeln durchsetzt ist, welche zu früh von der Berechnung ausgeschlossen wurden.
Es sind also zwei Dinge zu beobachten. Einerseits ist das ϵ zu groß gewählt, sodass das Modell generell verrauscht ist (homogenes Rauschen). Zusätzlich ist zu beobachten, dass durch das zu klein gewählte n_ϵ eine Art ”Salz und Pfeffer”-Rauschen entsteht.
Man erkennt an diesem Fall schon gut, dass Einzelergebnisse (Veränderungen eines Pixel-Farbwertes durch einen einzelnen Strahl) nicht als Konvergenzkriterium ausreichen. Um sicher gehen zu können, dass ein Pixel-Farbwert wirklich konvergiert, darf n_ϵ nicht zu klein gewählt sein. In Abbildung 10, 13, 16 ist zu sehen, dass das Rauschen auf dem Drachen wesentlich weniger geworden ist. Insgesamt sind

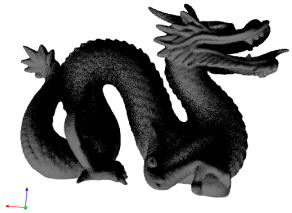

Abbildung 9

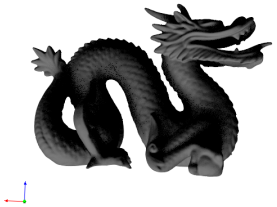

Abbildung 10

Abbildung 11

Abbildung 12

Abbildung 13

Abbildung 14

Abbildung 15

Abbildung 16

Abbildung 17

nur wenige Pixel zu sehen, die sich stark von ihrer Umgebung abheben. Die Verläufe sind weicher. Insegamt ein besseres Bild. Man sieht den Unterschied einerseits gut an den wesentlich weniger vorkommenden dunklen Pixel auf dem Modell und andererseits an den wesentlich weicheren Farbverläufen.
Details, die vorher nicht erkennbar waren, sind nun sichtbar. Was vorher stark verrauscht war, ist nun gut erkennbar. Das kleinere ϵ sorgt für weniger rauschen und das größere n_ϵ für weniger zu früh von der Berechnung ausgeschlossene Pixel. Insgesamt erhalten wir nun schon ein akzeptables Bild.
In Abbildung 11, 14, 17 sehen wir erneut eine Steigerung. Nun ist weder ein homogenes Rauschen mit bloßem Auge zu erkennen, noch "Salz und Pfeffer"-Rauschen zu sehen. Das Kriterium ist so hoch gewählt, dass selbst nach einer Stunde Rendern Abbildung 14 und 17 nicht konvergierten. Das Ergebnis ist eine makellose Drachenskulptur. Der Betrachter kann mit bloßem Auge kein störendes Rauschen mehr erkennen. Mit diesen Parametern sollte erfahrungsgemäß ein gutes Bild erzeugt werden können.

9.1.2 Parameteroptimierung

Um mit dem vorgestellten adaptiven Sampler ein sauberes Bild zu erzeugen , haben sich folgende Parametereinstellungen als gut erwiesen:
$n >= 5$, $n_\epsilon >= 50$, $\epsilon \in [0.0001, 0.0005]$.
Das Bild konvergiert dann zwar sehr langsam (abhängig von der Szene), es sind jedoch keine verrauschten Bereiche mehr im Bild. Setzt man ϵ höher als beschrieben, ergibt sich insgesamt ein verrauschteres Bild. Dieses Rauschen ist aber gleichmäßig verteilt. Setzt man jedoch n_ϵ niedriger, kann es (je nach Szene) zu unschönen, verrauschten Bereichen kommen, die zu früh von der Berechnung ausgeschlossen wurden. Gerade bei Environment Maps, wo viel Rauschen entsteht, kann dieser Effekt bei zu niedrigem n_ϵ entstehen.
Wichtig für den adaptiven Sampler ist, dass Pixel nur dann ausgeschlossen werden, wenn sie dauerhaft (über viele Iterationen hinweg) keine Veränderung mehr aufweisen und wenn der Pixel kein Ausreißer in einer homogenen Pixelregion der Boolmap ist. Aus diesem Grund rate ich davon ab n_ϵ zu klein zu wählen. Wenn man möchte, dass das Bild schneller konvergiert, sollte man ϵ so groß wählen, dass das Bildergebnis nicht zu verrauscht, aber ausreichend für die Anwendung wird. Dies ist inviduell von der darzustellenden Szene abhängig.
Abschließend empfiehlt sich, die Parameter nicht schlechter als in der zweiten gezeigten Einstellung zu setzten und nicht besser als die Parameter der dritten Einstellung.

9.2 Vergleich mit Augenblick

Um einen guten Vergleich mit Augenblick anstellen zu können, wird der adaptive Sampler einmal bezüglich der Performanz und andererseits bezüglich der Qualität getestet. Im Performanztest wird verglichen, wie schnell eine gewisse Frameanz

zahl erreicht ist. Im Qualitätstest wird gezeigt, wie sich die Bildqualität bei gleicher Strahlenanzahl verhält. Die Performanz wird mit einem und mit vier Threads getestet. Folgende Einstellungen für den adaptiven Sampler für alle Tests gesetzt: $\epsilon = 0,0001, n_\epsilon = 50, n = 5$.

9.2.1 Performanz

Zunächst vergleichen wir die Ergebnisse mit einem Thread. In Abbildung 19 bis 28 sind zwei verschiedene Szenen mit unterschiedlich vielen Iterationen (Frameanzahl) zu sehen. Die obere Reihe gehört jeweils zum adaptiven Sampler und die untere Reihe gehört zu RenderGin. Das Diagramm in Abbildung 18 zeigt, wie sich die Zahl der zu rendernden Frames auf die Renderzeit auswirkt. In Abbildung 19 - 23 ist ein Buddha mit Environment Map und Torrence Sparrow Shader (Gold Material) dargestellt. In Abbildung 24 - 28 ist der selbe Buddha mit Phong Shader und weißem Hintergrund abgebildet.
Grundsätzlich ist mit bloßem Auge kein wirklicher Unterschied der Bildqualität zwischen den Bildern des adaptiven Samplers und denen von RenderGin zu erkennen. Es sind Unterschiede da, jedoch ist nicht zu sagen, ob das eine besser als das andere ist. Es ist jedoch zu erkennen, dass der adaptive Sampler schneller am Ziel ist.
Zusätzlich zeigt sich, dass mit steigender Anzahl der Frames, die Diskrepanz der beiden Kurven größer wird. Dies ist natürlich offensichtlich, da RenderGin prinzipiell für jede Iteration über das Bild etwa gleich lange benötigt. Der adaptive Sampler braucht jedoch maximal pro Frame (etwa genau) so lang wie der Frame zuvor. Dadurch ist zu erklären, warum die Kurven auseinander gehen. An den Kurven der zweiten Szene (Abbildung 24 - 28) sieht man, dass die Diskrepanz größer als in der ersten Szene ist. Dies ist darauf zurück zu führen, dass die Szene generell weniger verrauscht ist (keine Environment Map) und zusätzlich der immer gleiche Hintergrund abgetastet wird. Dieser macht einen Großteil der Pixel im Bild aus. An dieser Szene kann man erkennen, wo die Potentiale des adaptiven Samplers liegen. Durch die große Anzahl von statischen Pixeln (immer die selbe Farbe), kann der adaptive Sampler früh viele Pixel von der Berechnung Ausschließen.
Im Bild selbst ist zu erkennen, dass Glanzbereiche in Bildern des adaptiven Samplers ausgeprägter sind als in den Bildern von RenderGin. Es ist aber nicht nachvollziehbar warum dies der Fall ist, da beim adaptiven Sampler pro Pixel maximal die gleiche Anzahl an Strahlen verschickt werden kann wie beim Berechnen mit der normalen Execution Chain von RenderGin. Dies gilt aber nur für die zweite Szene, denn in der ersten Szene ist das Phänomen nicht zu beobachten.

ein Thread	**100 Frames**	**300 Frames**	**1000 Frames**
Adaptives Sampling mit Environment Map	227,8 s	676,683 s	2030,39 s
Adaptives Sampling ohne Environmet Map		44,5915 s	85,7829 s
RenderGin mit Environment Map	230,077 s	682,364 s	2262,2 s
RenderGin ohne Environmet Map		65,3138 s	215,68 s

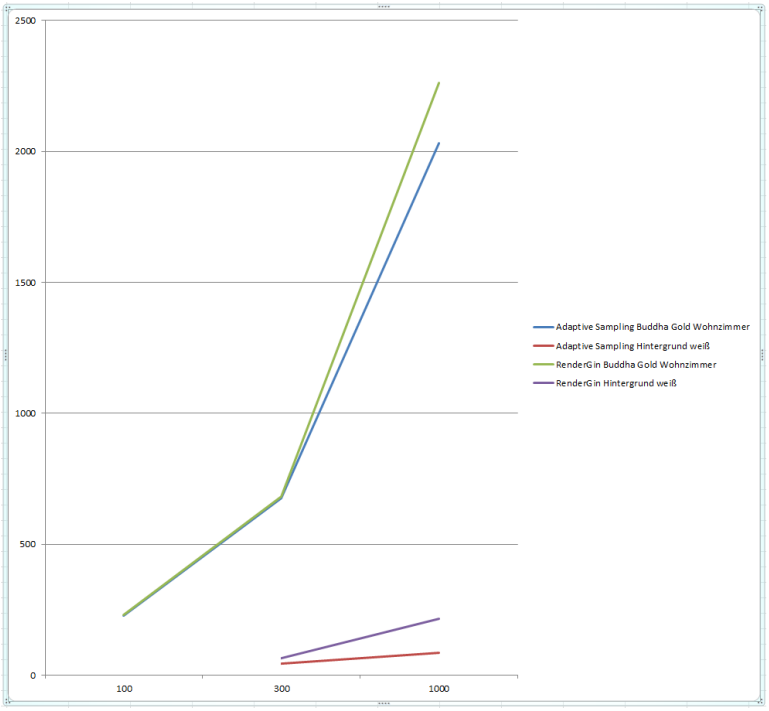

Abbildung 18: Diagramm: X-Achse Anzahl Frames, Y-Achse Zeit in Sekunden, **ein Thread**

Abbildung 19: 100 Frames

Abbildung 20: 300 Frames

Abbildung 21: 1000 Frames

Abbildung 22: 100 Frames

Abbildung 23: 300 Frames

Abbildung 24: 1000 Frames

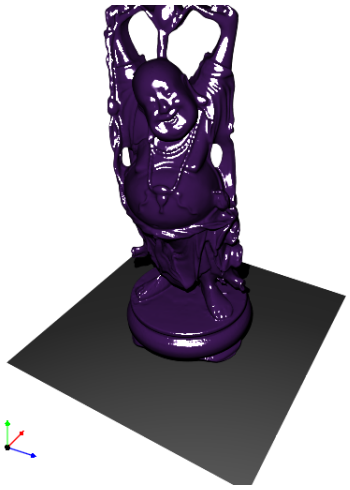

Abbildung 25: 300 Frames

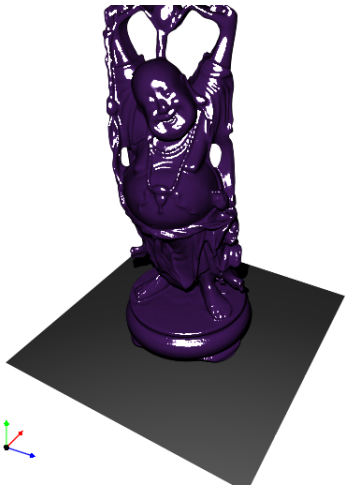

Abbildung 26: 1000 Frames

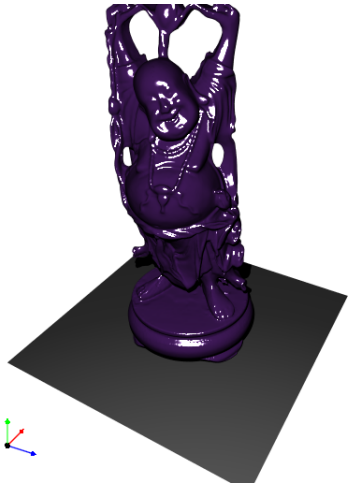

Abbildung 27: 300 Frames

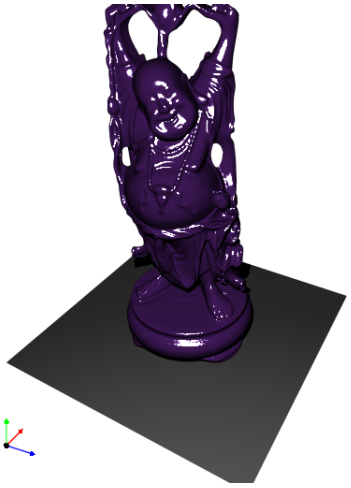

Abbildung 28: 1000 Frames

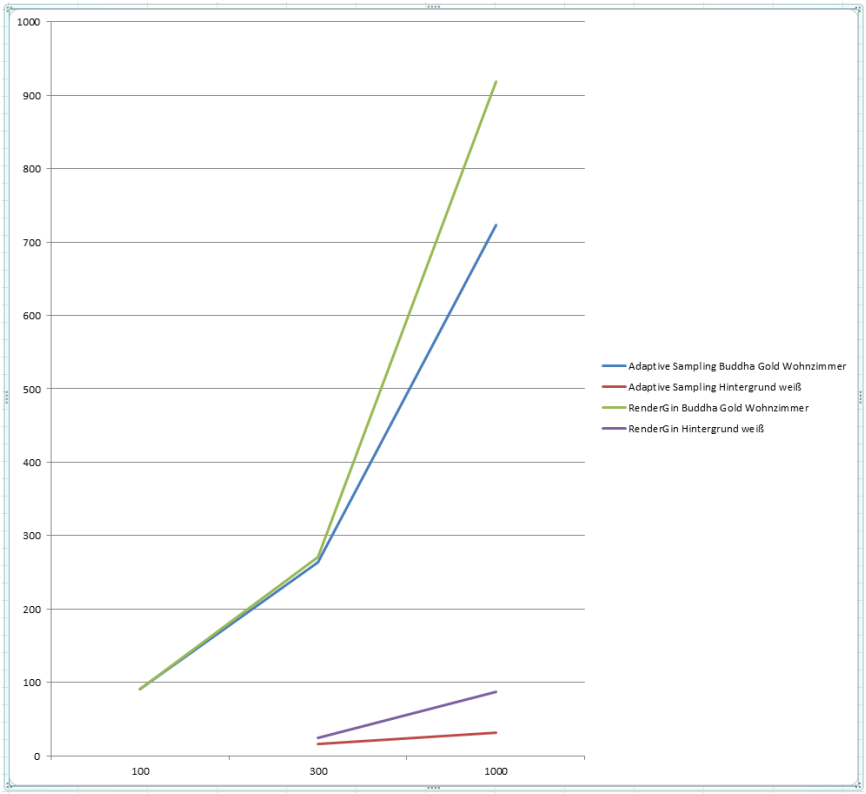

Abbildung 29: Diagramm: X-Achse Anzahl Frames, Y-Achse Zeit in Sekunden, **vier Threads**

Nun vergleichen wir die selbe Szenen mit vier Threads. Hier sind keine Bilder abgebildet, da diese mit den Bildern aus dem Test mit einem Thread vergleichbar sind. Die Ergebnisse fallen wie beim Test mit einem Thread aus. Die Geschwindigkeitsunterschiede zwischen dem adaptiven Sampler und RenderGin sind bei einem und vier Threads verhältnismäßig gering. In Abbildung 29 ist die Kurve zu den Bildern mit vier Threads dargestellt.

vier Threads	**100 Frames**	**300 Frames**	**1000 Frames**
Adaptives Sampling mit Environment Map	90,8656 s	263,696 s	723,119 s
Adaptives Sampling ohne Environmet Map		16,297 s	32,049 s
RenderGin mit Environment Map	90,4136 s	271,31 s	918,274 s
RenderGin ohne Environmet Map		24,549 s	86,945 s

9.2.2 Bildqualität

Nun wollen wir den adaptiven Sampler bezüglich der Bildqualität mit RenderGin vergleichen. Dazu rendern wir je ein Bild mit ca. 20.000.000 Strahlen und vergleichen die entstandenen Bilder vom adaptiven Sampler und von RenderGin. Zum

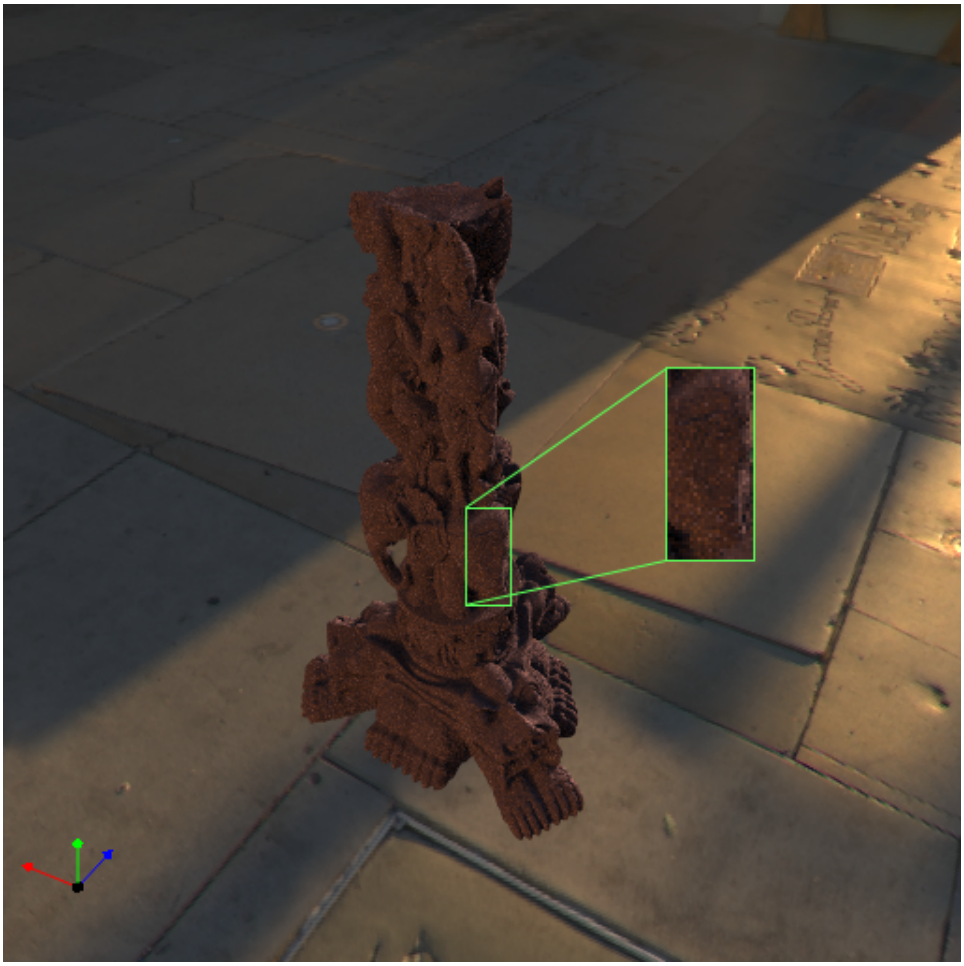

Abbildung 30: 20.000.000 Strahlen adaptives Sampling ($\epsilon = 0,0001, n_\epsilon = 50, n = 5$)

Test verwenden wir eine Szene mit einer Statue, die mit einem Diffusen Material belegt ist und in einer Environment Map als Umgebung platziert ist.
In Abbildung 30 ist das Bild des adaptiven Samplers, Abbildung 31 ist das zugehörige Bild von RenderGin. Für das Bild in Abbildung 30 hat der adaptive Sampler 15,167 Sekunden gebraucht und dabei 98 *Bilditerationen* absolviert (*Frames*). Das Bild in Abbildung 31 hat RenderGin 11,139 Sekunden gebraucht und dabei 77 *Bilditerationen* absolviert.
Es ist zu erkennen, dass das vom adaptiven Sampler produzierte Bild weniger Rauschen in homogenen Bildbereichen des in der Szene befindlichen Modells aufweist. Wie zu erwarten braucht der adaptive Sampler länger um die selbe Anzahl von Strahlen zu verschießen, da er mehr Bilditerationen bewältigen muss.
An Abbildung 32 und 33 wird eine Szene mit Schatten dargestellt. Es ist zu erkennen, dass der adaptive Sampler auch hier für weniger Rauschen Sorgt. Das Bild in Abbildung 32 (adaptiver Sampler) wurde in 3,611 Sekunden gerendert und

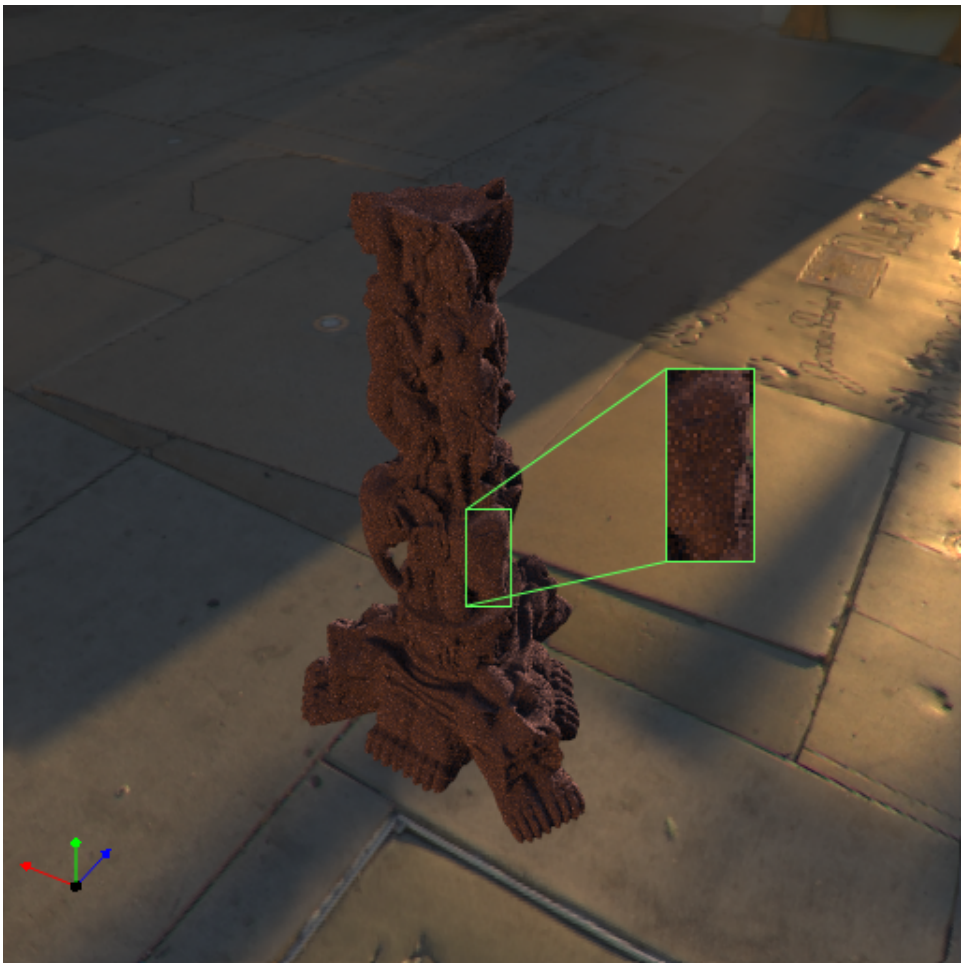

Abbildung 31: 20.000.000 Strahlen RenderGin

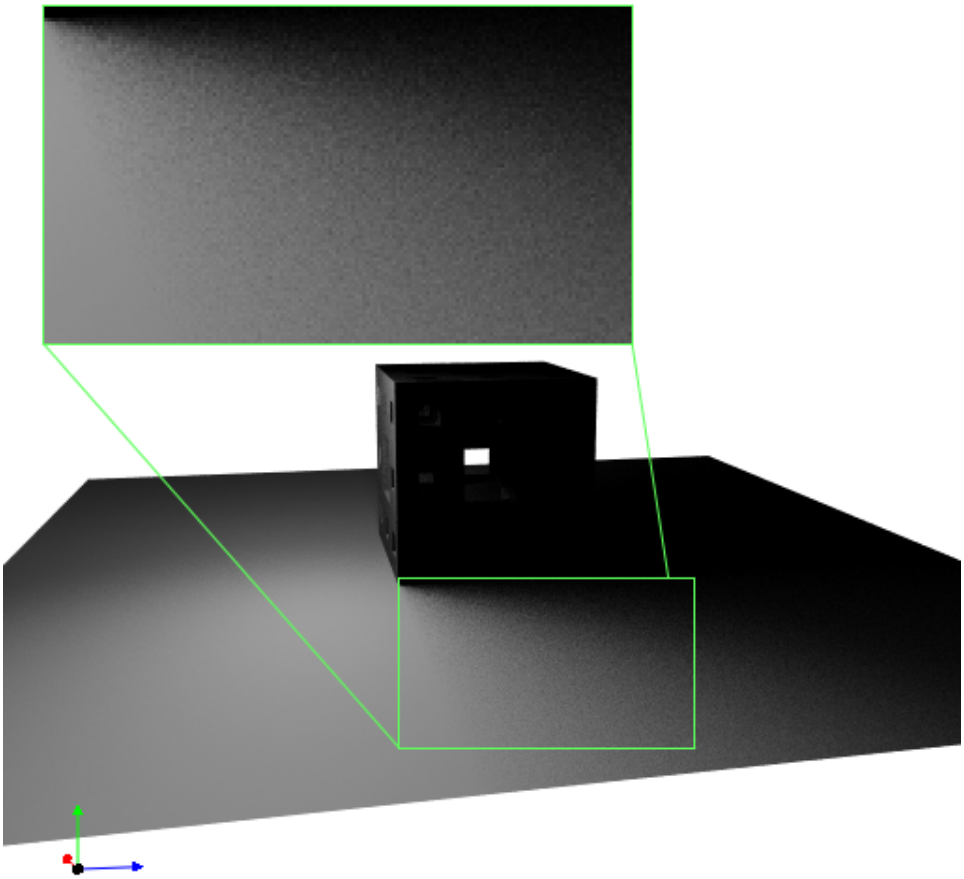

Abbildung 32: 20.000.000 Strahlen adaptives Sampling ($\epsilon = 0,0001, n_\epsilon = 50, n = 5$)

116 *Bilditerationen* durchlaufen. Das Bild in Abbildung 33 (RenderGin) hat 1,52 Sekunden gebraucht und dabei 77 *Bilditerationen* durchlaufen. Wie auch im Bild ohne Schatten ist eine Verbesserung des Bildes und speziell des diffusen Schattenbereichs zu erkennen. Sonst sind die gleichen Phänomene wie im Bild mit der Statue (Abbildung 30 und 31) zu erkennen.

10 Auswertung

Die Tests haben ergeben, dass der adaptive Sampler in der Lage ist, bei gleicher Strahlenanzahl ein besseres Bild zu erzeugen, sowie dass er in der Lage ist mit gleich vielen *Bilditerationen* schneller am Ziel zu sein, als die regulären *ExecutionStates* von RenderGin. Insofern kann diese Arbeit Aufschluss darüber geben, ob adaptives Sampling Vorteile für bestimmte Szenarien bietet. Besonders gut kann der adaptive Sampler immer dann arbeiten, wenn das Bild viele Pixel beinhaltet,

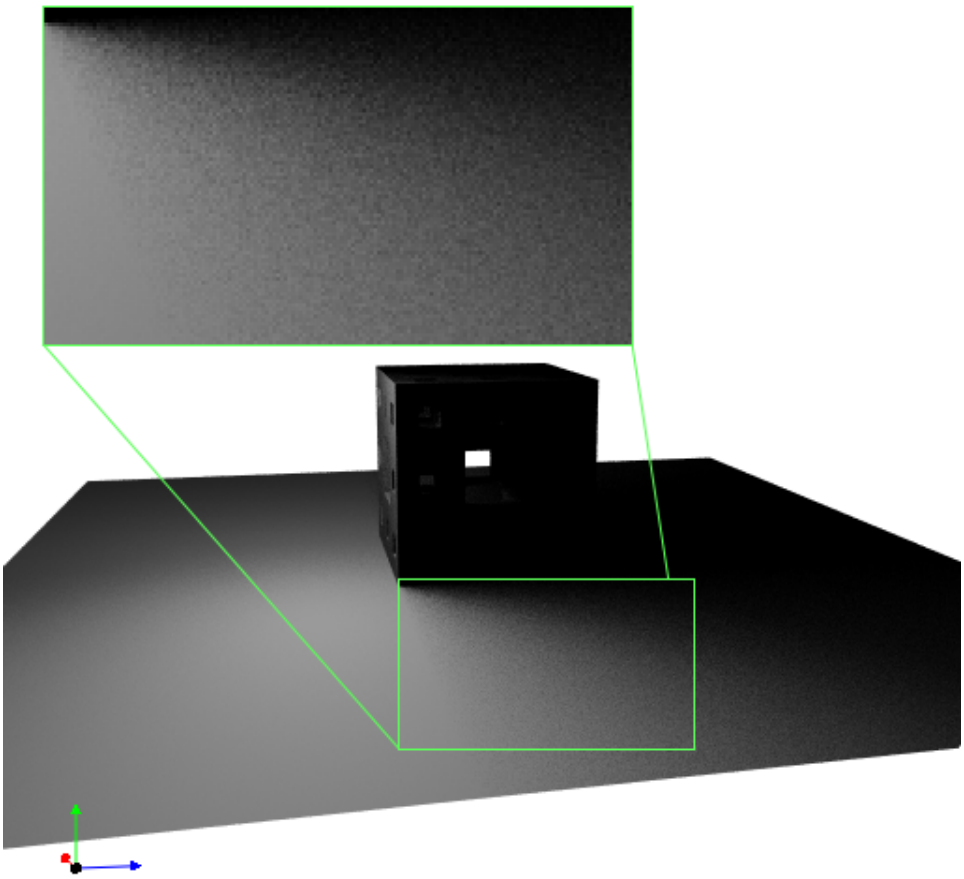

Abbildung 33: 20.000.000 Strahlen RenderGin

die keine Veränderung der Farbe über die *Bilditerationen* hinweg vorweisen.
Um beurteilen zu können, in welchen Szenarien konkret der Einsatz des adaptiven Samplers Sinn macht, müssten diesbezüglich zusätzliche Tests durchgeführt werden. Auch können die Parameter des adaptiven Samplers für bestimmte Szenen entsprechend angepasst werden, sodass sich je nach Anwendung ein maximaler Performanz- oder Qualitätsgewinn ergibt.

11 Fazit

Abschließend können wir sagen, dass alle Ziele vorerst erreicht wurden. Dies soll heißen, dass die Aspekte, die man sich vom adaptiven Sampler gewünscht hat, auch in den Ergebnissen der Tests wieder zu finden waren. Wie sinnvoll und in der Praxis bedeutsam diese Vorteile des adaptiven Samplers sind, muss sich in der Praxis zeigen. Auch eine anknüpfende Studienarbeit wäre denkbar und sinnvoll, da noch lange nicht alle Potentiale von adaptivem Sampling in dieser Arbeit verwirklicht und getestet wurden.
Sehr interessant ist auch die Erweiterung des Verfahrens auf Pathtracing. In dieser Arbeit war es leider nicht möglich auf diese spezielle Variante von Raytracing einzugehen. RenderGin unterstützt Pathtracing von Haus aus, sodass die Implementation des adaptiven Samplers als Pathtracer kein allzu großer Aufwand sein sollte. Natürlich bedeutet der adaptive Sampler zusätzliche Arbeit, jedoch zahlt sich diese nach Erreichen einer gewissen Anzahl an Bilditerationen in bestimmten Szenen wieder aus. Das Thema selbst lässt noch viel Raum für Weiterentwicklungen und weiterführende Forschungsarbeiten.

12 Schlusswort

Zuletzt möchte ich allen danken, die bei der Arbeit in irgendeiner Weise geholfen haben. Dies gilt besonders für die Mitarbeiter von Numenus, die mich immer mit ihren Ratschlägen unterstützt haben. Vielen Dank natürlich auch an den Geschäftsführer Oliver Abert, der diese Arbeit erst ermöglicht hat und mit seinen unkonventionellen Vorschlägen immer einen passenden Rat bei Problemen für mich hatte.

13 Anhang

13.1 Differenzbild

Das Differenzbild war im Stadium des Entwickelns sehr hilfreich. Abbildung 34 zeigt eine Szene mit zugehörigem (skaliertem) Differenzbild.

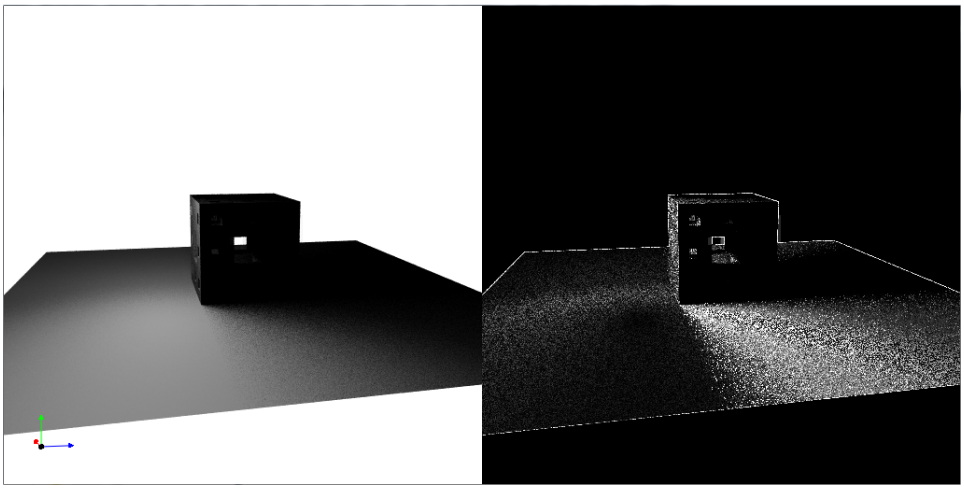

Abbildung 34: Gerenderte Szene mit zugehörigem Differenzbild (mit dem Faktor 100 skaliert)

13.2 Code

Abbildung 35 zeigt den Programmcode zur Berechnung der 3D-Rasterposition für die Strahlgenerierung (*RayGeneration*) für ein *Pack*. Abbildung 36 zeigt wie, nach Berechnung der Rasterpositionen und dem Berechnen aller für das adaptive Sampling nötigen Puffer, der erste Pixel im *Pack* seinen entsprechenden Strahl zugewiesen bekommt. Abbildung 37 zeigt den Code des Zustandes *Trace* und Abbildung 38 zeigt den Code des Zustandes *Shade* für den linken, oberen Pixel in einem *Pack*.

14 Quellen

[CG2008] Müller, Steffan: Computergrafik 1 Vorlesungsfolien. Foliensatz 19. Universität Koblenz: Institut für Computervisualistik. WS 2008/2009

[AUGENBLICK2009] Abert, Oliver: AUGENBLICK. Ein effizientes Framework für Echtzeit Ray Tracing. 1. Josef Eul Verlag 2009

[PBR2004] Matt Parr, Greg Humphreys: Pysically Based Rendering. 2. Morgan Kaufmann

```
240     //calculating the axis of the camera
241     z=-cam->direction();
242     z.normalize();
243     x=cross(cam->up(), z);
244     x.normalize();
245     y=cross(z, x);
246     y.normalize();
247
248     //stepvectors
249     vecPixX=x/width;
250     vecPixY=-y/height;
251
252     //basevector for viewplane
253     Vector3 topleft=Vector3(cam->topLeft()[0],cam->topLeft()[1],cam->topLeft()[2]);
254
255     //step-vectors for iterating viewplane-pixelpositions
256     Vector3 rayX=vecPixX*2;
257     Vector3 rayY=vecPixY*2;
258
259     //calculate 3D-TilePosition
260     Vector3 tilePosition=topleft+vecPixX*tile->position(0)+vecPixY*tile->position(1);
```

Abbildung 35: Code Rasterpositionsberechnung

```
262     //start adaptive sampling process when all lighsources are equaly taken in frame
263     if (AB::DATA->useHeadLight() || AB::DATA->frameCount()  % AB::DATA->activeLightSources().size
264
265     computeTileMask(tile);
266
267     for (unsigned int t = 0; t < Tile::PACKETS_VERT; ++t)
268     {
269         for (unsigned int s = 0; s < Tile::PACKETS_HORIZ; ++s)
270         {
271             // i is the 1D-postion in the actual tile
272             unsigned int i=t * 64 + s * 2;
273
274             //rays is the actual RayPack
275             RayPack* rays = tile->rayPack(s, t);
276
277             // creating 2 VecPack3 for information-storage for the rayPack
278             VecPack3 directions=VecPack3();
279             VecPack3 origins=VecPack3();
280
281             //get 3D-Postion of the Camera
282             Vector3 camPos=cam->position();
283
284             //calc 3D-position-vector for the pack
285             Vector3 Packpos=tilePosition + s * rayX  + t * rayY;
286
287             //setting directions if we want to shoot at pos(x,y)
288
289             //Pixel top-left
290             //framePosition is the 1D-position of the Frame
291
292             if(mBoolMask[getFramePosition(s,t,tile,0)/4]){
293
294                 //creating rays
295                 rays->origins()[0][0]=camPos[0];
296                 rays->origins()[1][0]=camPos[1];
297                 rays->origins()[2][0]=camPos[2];
298
299                 rays->directions()[0][0]=(Packpos + random() * vecPixX + random() * vecPixY)[0];
300                 rays->directions()[1][0]=(Packpos + random() * vecPixX + random() * vecPixY)[1];
301                 rays->directions()[2][0]=(Packpos + random() * vecPixX + random() * vecPixY)[2];
302
303                 //note that this pixel is shot one more time
304                 pixelState[getFramePosition(s,t,tile,0)/4]+=0.1;
305             }else
306             {
307                     //creating NULL-rays
308                     rays->origins()[0][0]=0;
309                     rays->origins()[1][0]=0;
310                     rays->origins()[2][0]=0;
311
312                     rays->directions()[0][0]=0;
313                     rays->directions()[1][0]=0;
314                     rays->directions()[2][0]=0;
315
316                 }
```

Abbildung 36: Code Strahlgenerierung

```
64      //get the tile
65      Tile* tile = reinterpret_cast<Tile*>(parent);
66
67      //loop all pixels
68      for (unsigned int t = 0; t < Tile::PACKETS_VERT >> tile->downscale(); ++t)
69      {
70          for (unsigned int s = 0; s < Tile::PACKETS_HORIZ >> tile->downscale(); ++s)
71          {
72              RayPack* rays = tile->rayPack(s, t);
73              HitPack* hits = tile->hitPack(s, t);
74
75              if( rays->directions()[0][0]!=0 || rays->directions()[0][1]!=0 || rays->directions()[0][2]!=0 || rays->directions()[0][3]!=0
76              ||  rays->directions()[1][0]!=0 || rays->directions()[1][1]!=0 || rays->directions()[1][2]!=0 || rays->directions()[1][3]!=0
77              ||  rays->directions()[2][0]!=0 || rays->directions()[2][1]!=0 || rays->directions()[2][2]!=0 || rays->directions()[2][3]!=0)
78              {
79
80                  Bool4 selection=Bool4(
81                      (rays->directions()[0][0]!=0 || rays->directions()[1][0]!=0 || rays->directions()[2][0]!=0),
82                      (rays->directions()[0][1]!=0 || rays->directions()[1][1]!=0 || rays->directions()[2][1]!=0),
83                      (rays->directions()[0][2]!=0 || rays->directions()[1][2]!=0 || rays->directions()[2][2]!=0),
84                      (rays->directions()[0][3]!=0 || rays->directions()[1][3]!=0 || rays->directions()[2][3]!=0)
85                      );
86                  hits->reset(selection);
87                  AUGENBLICK->trace(*rays, *hits, ls);
88              }
89
90          }
91      }
```

Abbildung 37: Code Strahlverfolgung

```
83      //loop all pixels
84      for (unsigned int t = 0; t < Tile::PACKETS_VERT >> tile->downscale(); ++t)
85      {
86          for (unsigned int s = 0; s < Tile::PACKETS_HORIZ >> tile->downscale(); ++s)
87          {
88
89              RayPack* rays = tile->rayPack(s, t);
90              HitPack* hits = tile->hitPack(s, t);
91              ColorPack colors;
92
93              //only shade if there is a pixel in the pack that should be further rendered therefore we ask for the case in that every direction is Null in the Pack
94              if(     rays->directions()[0][0]!=0 || rays->directions()[0][1]!=0 || rays->directions()[0][2]!=0 || rays->directions()[0][3]!=0
95                  ||  rays->directions()[1][0]!=0 || rays->directions()[1][1]!=0 || rays->directions()[1][2]!=0 || rays->directions()[1][3]!=0
96                  ||  rays->directions()[2][0]!=0 || rays->directions()[2][1]!=0 || rays->directions()[2][2]!=0 || rays->directions()[2][3]!=0)
97                  {
98
99                      //reoize colorpack
100                     colors.setZero();
101
102                     //shade the pack
103                     AUGENBLICK->shade(*rays, *hits, colors, Float4::one(), ls);
104
105                     //alternatively also write to new local structure
106                     simd_transpose4(colors[0], colors[1], colors[2], colors[3]);
107
108                 }
109
110             //decide ech pixel in the pack to take values from memory or not
111
112             float * dataOld=AB::DATA->backbone()->uploadBuffer()->data();
113
114             //decide pixel top-left
115             if(rays->directions()[0][0]!=0 || rays->directions()[1][0]!=0 || rays->directions()[2][0]!=0)
116             {
117                 tile->setImageData(s*2  , t*2, colors[0]);
118             }
119             else
120             {
121                 unsigned int framePosition=getFramePosition(s,t,tile,0);
122
123                 tile->setImageData(s*2,t*2, Float4(dataOld[framePosition], dataOld[framePosition + 1], dataOld[framePosition + 2], dataOld[framePosition + 3]));
124             }
```

Abbildung 38: Code Beleuchtung

```
160 void AB::ASRayGeneration::computeTileMask(Tile* tile)
161 {
162     switch(AB::DATA->frameCount())
163     {
164         //case 0 - the tile isn't shot yet
165         case 0: break;
166 
167         //case 1 - the tile has been shot one time, so we cam copy the old values for the tile
168         case 1: fillOld(tile);  break;
169 
170         //case default - the tile has been shot two or more times, so we can compare the old and the new Values of the tile
171         default: fillBoolMask(tile); fillOld(tile); break;
172     }
173 }
174 
175 unsigned int getFramePosition(unsigned int s,unsigned int t,Tile* tile, char flags)
176 {
177     //calculating position in Frame
178 
179     unsigned int position;
180 
181     switch(flags)
182     {
183     case 0: position=(tile->position(0) + s * 2)  + width * tile->position(1) + t * 2 * width ; break;
184 
185     case 1: position=((tile->position(0) + s * 2)  + width * tile->position(1) + t * 2 * width) + 1; break;
186 
187     case 2: position=((tile->position(0) + s * 2)  + width * tile->position(1) + t * 2 * width) + width ; break;
188 
189     case 3: position=((tile->position(0) + s * 2)  + width * tile->position(1) + t * 2 * width) + width + 1 ; break;
190     }
191     return position*4;
192 }
```

Abbildung 39: Code adaptives sampling 1

```
194 void AB::ASRayGeneration::fillOld(Tile* tile)
195 {
196     //iterating over tile
197     for(int y=0;y<32;y++)
198         {
199             for(int x=0;x<32;x++)
200             {
201 
202                 int position= (tile->position(0) + x  + width * tile->position(1) + y * width) * 4;
203 
204                 if(mBoolMask[position/4]==true)
205                 {
206 
207                 dataOld[position] = data[position];
208 
209                 dataOld[position+1] = data[position+1];
210 
211                 dataOld[position+2] = data[position+2];
212 
213                 }
214 
215             }
216         }
217 }
```

Abbildung 40: Code adaptives sampling 2

```
88  void AB::ASRayGeneration::fillBoolMask(Tile* tile)
89  {
90      //iterating over tile
91      for(int y=0;y<32;y++)
92          {
93              for(int x=0;x<32;x++)
94              {
95                  //1D-postion in tile
96                  int i=y*32+x;
97
98                  //1D-position in frame
99                  int position= tile->position(0) + x  + width * tile->position(1) + y * width;
100
101                 if(mBoolMask[position]==true)
102                     {
103                         //calculating difference between the old pixelvalue and the actual
104                         float difference=absDiffOf2Floats(dataOld,data,position*4);
105
106                         if(difference<epsilon && pixelState[position] >                    ace)
107                             {
108                                 mTrue[position]+=0.1;
109                             }
110                         else
111                             {
112                                 mTrue[position]-=0.1;
113                             }
114
115                         // if we reached an amount of Falling-Down-Epsilon-Cases for that Pixel
116                         if(mTrue[position] > minPixelStops)
117                             {
118                                 //note that we will not shoot a pixel anymore
119                                 mBoolMask[position]=false;
120
121                             }
122
123                     }
124             }
125         }
126
127     //iterating 2nd time over tile (outlier detection)
128     for(int y=0;y<32;y++)
129         {
130             for(int x=0;x<32;x++)
131             {
132                 //1D-postion in tile
133                 int i=y*32+x;
134
135                 //1D-position in frame
136                 int position= tile->position(0) + x  + width * tile->position(1) + y * width;
137
138                 if(mBoolMask[position]==true)
139                 {
140                         //boundary treatment
141                         if(position>width&&position<width*height-width&&AB::DATA->frameCount()>2+minPixelStops*10)
142
143                         //Ask the neighbourhood of the Pixel, wether they are true or false
144                         //if true this one must also be true (outlier detection)
145                         if(mBoolMask[position-1]==mBoolMask[position+1]==true ||
146                             mBoolMask[position-width]==mBoolMask[position+width]==true)
147                             {
148                                 mBoolMask[position]=true;
149                             }
150                 }
151             }
152         }
153 }
154
```

int position

Abbildung 41: Code adaptives sampling 3