Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Wahrnehmungsorientiertes Volumen-Rendering

Ruben Bauer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Thomas Ertl
Betreuer/in:	Valentin Bruder M.Sc., Dipl.-Inf. Christoph Schulz, Dr. Steffen Frey
Beginn am:	12. Mai 2018
Beendet am:	12. Oktober 2018

Kurzfassung

Diese Arbeit handelt um Wahrnehmungsorientiertes Volumen-Rendering. Das menschliche Auge ermöglicht dem Menschen seine visuelle Wahrnehmung, welche in foveales und peripheres Sehen unterteilt werden kann. Das foveale Sehen ist detailliert, scharf und farbig und befindet sich im Zentrum des visuellen Wahrnehumgsbereiches, während das periphere Sehen sich außerhalb des Zentrums befindet und unschärfer und weniger farbig ist. In dieser Arbeit wird speziell diese Eigenschaft des menschlichen Sehapparates im Zusammenhang mit Volumen-Rendering untersucht. Dafür werden unterschiedliche Ansätze betrachtet, die Bildqualität im peripheren Bereich zu senken, um die Performanz des Volumen-Renderings zu erhöhen und gleichzeitig die Qualität der Darstellung zu erhalten oder zu verbessern. Die Daten zur Ermittlung des fovealen beziehungsweise peripheren Bereichs werden mit einem Eye-Tracking Gerät gemessen und fließen direkt in die Darstellung ein.

Inhaltsverzeichnis

1 Einleitung

Einleitung zu dieser Arbeit schreiben.

Gliederung der Arbeit kurz beschreiben.

2 Grundlagen

Der erste Abschnitt des Kapitels beschäftigt sich mit verwandten Arbeiten zum Thema Wahrnehmungsorientiertes Volumen-Rendering. Der zweite Abschnitt handelt über die Grundlagen des menschlichen Sehapparates. Hier werden die Fähigkeiten und Limitierungen der visuellen Wahrnehmung des Menschen diskutiert. In Abschnitt drei, wird die Funktionsweise von Raytracing erläutert, welches ein grundlegender Algorithmus, der für diese Arbeit zugrunde liegender Implementierung ist. Zusammenhängend mit Raytracing wird im Abschnitt vier, die Verwendung des Raytracers für das Volumenrendering erläutert. Abschnitt fünf diskutiert die Auswahl des für diese Arbeit zugrunde liegenden Eyetrackers und dessen Verwendung für die Erfassung des fovealen und peripheren Bereichs. Aus Performanzgründen kann es hilfreich sein für Berechnungen auf einer GPU, die Architektur der GPU zu betrachten und unter Umständen Algorithmen für eine bessere Effizienz anzupassen. Diese Thematik wird in Abschnitt sechs behandelt.

Beschreibung der Grundlagen überarbeiten

2.1 Related Work

Wahrnehmungsorientiertes Volumenrendering ist kein absolut neues Arbeitsgebiet und ist schon Teil einiger wissenschaftlicher Arbeiten gewesen

Beispiele

. Da diese Arbeit auf zwei trennbare Aspekte beruht, unterteile ich den Related Work Abschnitt in zwei Teile. Der erste Abschnitt bezieht sich auf Arbeiten im Bereich des wahrnehmungsorientierten Renderings mit dem Ziel, die Performanz einer Anwendung zu steigern. Die Ansätze hier beziehen sich meist darauf, dass die Qualität der Darstellung im peripheren Bereich der visuellen Wahrnehmung, gesenkt wird und so für die Berechnung eines Bildes weniger Rechenleistung aufgewendet werden muss.

Was genau ist mit Performanz gemeint?

Der zweite Abschnitt bezieht sich auf Arbeiten zu wahrnehmungsorientiertem Volumenrendering, mit dem Ziel, die Qualität der Darstellung zu erhöhen. Dabei werden vor allem Ansätze zur geschickten Anpassung von Parametern einer Transferfunktion vorgestellt, die dem Betrachter ein insgesamt besseres Verständnis der Volumendaten ermöglichen soll.

2.1.1 Performanz- und Wahrnehmungsorientiertes Volumenrendering

In einem Paper von Marc Levoy und Ross Whitaker *Gaze-Directed Volume Rendering* [LW90] erforschten diese Methoden, wie Eyetracking-Daten in Rendering-Algorithmen eingesetzt werden könnnen. Das Ziel ihrer Forschung war es, dem Nutzer einen Arbeitsplatz für Echtzeit-Volumen-Rendering, mit der Illusion eines hochauflösendes Bildes über den gesamten Bildschirm, zu präsentieren, welches durch Eyetracking unterstützten Verfahren einen geringeren Berechnungsaufwand als herkömmliche Volumen-Renderer hat. Dafür präsentierten sie eine Implementierung eines Ray-Tracers für Volumendaten, in welcher mit Hilfe der Eyetrackingdaten der Blickfokus auf dem Bildschirm berechnet wurde und um diese Position herum, die Anzahl der Strahlen und die Samples pro Strahl, abhängig von der Distanz zum Blickfokus, angepasst wurden. Die Implementierung basiert darauf, dass der Detaillgrad der visuellen Wahrnehmung des Menschlichen Auge nur in einem kleinen, zentralen Bereich, der Fovea, am höchsten ist und zu den Rändern des Blickfelds hin, der periphere Bereich, stark abnimmt. Ausgehend davon, berechneten sie in dem von dem Nutzer fokussierten Bereich, das Bild mit der vollen Auflösung und einer hohen Abtastrate der Strahlen. In dem restlichen Bereich reduzierten sie die Auflösung des Bildes und abhängig von dem Abstand eines Pixels zum Blickfokus, die Abtastrate eines Strahls. In ihrer Implementierung verwendeten sie 2D und 3D mip maps, einen Eye Tracker und die Pixel-Planes 5 rendering engine, ein hoch paralleles Raster Display System. Um die geringere Abtastung in der Peripherie gut nutzen zu können, wurde aus den 3D Volumendaten eine 3D mip map erstellt. Die 3D mip map wird durch den Ray-Tracing Algorithmus abgetastet und durch trilineare Interpolation eine 2D mip map erstellt. Die 2D mip map ist Grundlage für die Erstellung des endgültigen Bildes. In ihren Ergebnissen konnten sie die Rendering Kosten für ein teilweise hochauflösendes Bild im Vergleich zu einem vollständig hochauflösenden Bildes, um bis den Faktor fünf, senken.

In einer Arbeit von Guenter et. al. *Foveated 3D Graphics* [GFD+12], wurde ähnlich zu dem Paper von Levoy und Whitaker, der Abfall der visuellen Auflösung des Auges außerhalb des visuellen Zentrums, der Fovea, ausgenutzt, um eine beschleunigte Berechnung des Bildes zu erhalten. Im Gegensatz zu der davor genannten Arbeit, ist das darunterliegende System hier kein Ray-Tracer für Volumendaten, sondern eine Grafikpipeline für die Rasterung von 3D Szenen. Das Bild wird hier aus drei Teilbilder zusammengesetzt, welche jeweils mit unterschiedlichen Auflösungen berechnet wurden und wie Schichten übereinander gelegt werden. Die drei Schichten sind: innere Schicht, mittlere Schicht und äußere Schicht. Die innere Schicht hat ungefähr die Größe des fovealen Bereichs auf dem Bildschirm und wird in der maximalen Auflösung berechnet. Ihr Mittelpunkt ist der Blickfokus des Betrachters. Die mittlere Schicht ist ein bisschen größer als die innere Schicht und wird mit einer niedrigeren Auflösung berechnet. Sie wird ebenfalls auf den Blickfokus des Betrachters zentriert. Die äußerste Schicht überdeckt den gesamten Bildbereich und wird in der niedrigsten Auflösung berechnet. Um die Schichten zusammensetzen zu können, werden die mittlere und äußere Schicht jeweils zur nativen Auflösung des Bildschirms, die gleiche Auflösung wie die innere Schicht, interpoliert. Scharfe Kanten zwischen den Schichten werden dadurch vermieden, dass diese sich leicht überlappen und spezielle Blend-Masken verwendet werden, um die verschiedenen Schichten glatt übereinander zu blenden. In ihrer Arbeit nahmen sie sich auch das Problem an, dass das starke Unterabtasten, um bis zu dem Faktor sechs in jede Dimension, in der mittleren und äußeren Schicht zu störenden und sich bewegenden Artefakten führen kann. Um dies entgegen zu wirken, verwendeten sie drei Antialiasing Techniken: Hardware Multi-Sample Antialiasing (MSAA), *Temporal Reprojection* und *whole frame jitter sampling*. Um die Blickposition zu erfassen nutzten sie den Tobii Tx 300 Eye Tracker mit einer Worst-Case Latency von 10 ms. Das Bild wurde auf einem Computer mit Intel Xeon CPU (E5640

mit 2.67 GHz) und einer NVidia GeForce GTX 580 GPU berechnet. Dargestellt wurde es auf einem 23 Zoll 1920×1080 LCD Monitor mit einer Bildwiederholungsrate von 120 Hz. Mit ihrem System und Techniken erlangten sie eine Performanz-Verbesserung von einem Faktor zwischen fünf und sechs auf einem Monitor mit HD Auflösung. Dabei erreichten sie eine Darstellungsqualität, die vergleichbar mit dem Rendern eines hochauflösenden Bildes über den gesamten Bildschirm ist.

2.1.2 Wahrnehmungsorientiertes Volumenrendering zur Qualitätssteigerung

R. Englund und T. Ropinski stellten in ihrem Paper *Quantitative and Qualitative Analysis of the Perception of Semi-Transparent Structure in Direct Volume Rendering* [ER] verschiedene Techniken, zur Verbesserung der Wahrnehmung von komplexen volumetrischen Daten, vor. In einer Studie mit über 300 Teilnehmern untersuchten sie, wie diese Techniken zur verbesserten Wahrnehmung von Volumen beitragen und verglichen die verschiedenen Ansätze miteinander. Dabei mussten die Teilnehmer jeweils kleine Aufgaben absolvieren, so dass Rückschlüsse darauf geschlossen werden können, wie sehr eine gewisse Technik dem Teilnehmer bei der Erkennung von Form und Tiefe eines Objekts in einem Volumen beiträgt. Um eine bessere direkte Erforschung der Volumendaten später ermöglichen zu können, wurden Techniken, die automatisch Rendering-Parameter angepasst haben hier ausgelassen. Nur wenn der Nutzer selbst interaktiv sein kann, also die Parameter des Volumen-Renderings, wie die Transferfunktion und Kamera selbst anpassen kann, ermöglicht dies eine direkte Erforschung. In ihrer Studie haben sie sechs Techniken ausgewertet. Darunter *Direct Volume Raytracing* (DVR) als grundlegende Technik und *Depth Darkening* wobei Tiefeneffekte ähnlich zu *Ambient Occlusion* dadurch hervorgerufen werden, dass tiefere Objekte dunkler gezeichnet werden. In ihrer Auswertung kamen sie unter Anderem dazu, dass Techniken, die natürliche Lichteffekte in den Volumendaten simulieren, deutliche Vorteile gegenüber die anderen getesteten Ansätze gezeigt haben.

Anders als zu den untersuchten Techniken von Englund und Ropinski [ER] präsentieren Aidong Lu et. al. in ihrem Paper *Volume Composition Using Eye Tracking Data* [LME06] eine Methode für automatisierte Parameterauswahl bei der Betrachtung von Volumendaten mit Hilfe eines Eye Tracking Gerätes. Das Ziel, welches sie mit dieser Methode verfolgen ist es, die mühsame Nutzerinteraktionen bei der Auswahl der Parameter zu vereinfachen und damit die Nutzbarkeit des Darstellungssystems zu verbessern. Volumendaten können sehr komplex sein und daher ist es oft schwierig herauszufinden, was der Nutzer in dem Volumen betrachten möchte und was dahingehend hervorgehoben werden soll. Trotzdem ermöglichen Eigenschaften des Volumen-Renderings, wie konstante Größen, Formen und Positionen der Objekte eine automatische Anpassung der Parameter. Um die Bereiche, die für den Nutzer von Interesse sind, zu bestimmen, wird ein Eyetracker zur Hilfe genommen. Dieser misst die Augenbewegungen und die Blickposition auf dem Bildschirm. Es wird zwischen zwei Hauptsächlichen Augenbewegungen unterschieden: Sakkade und Fixation. Eine Sakkade ist eine schnelle Augenbewegung von einem Punkt zu einem anderem. Bei einer Fixation ruht das Auge auf einem Punkt. Die Punkte, die fixiert werden, sind meist für den Nutzer von Interesse. Da die Blickposition durch den Eye Tracker nur auf einer 2D-Ebene bestimmt werden kann, wird durch ein konstantes Rotieren des Volumenobjekts und der parallelen Aufzeichnung der Augenbewegung versucht, die 3D Position des fixierten Objektes zu rekonstruieren. Aus den Eyetracking und Volumendaten bestimmen sie mit Hilfe mehrerer Clustering-Methoden gewichte für die einzelnen Voxel des Volumens und berechnen so die Wichtigkeit der Objekte innerhalb des Volumens für den Nutzer. Entsprechend dieser Ergebnisse wurden die Render Parameter angepasst,

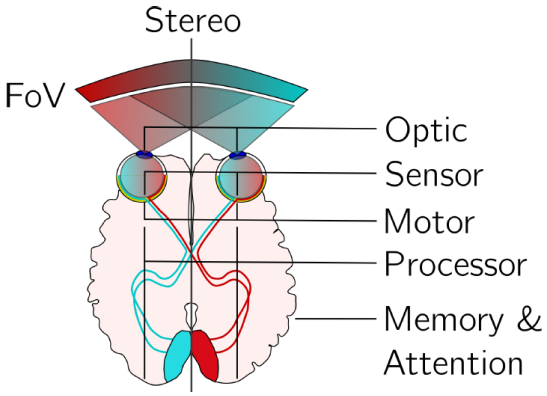

Abbildung 2.1: Modell des visuellen Wahrnehmungsapparates des Menschen aus [**doi:10.1111/cfg.13150**]

um die für den Nutzer am interessantesten Objekte hervorzuheben und anzuzeigen. Aidong Lu et. al. kamen zu dem Schluss, dass die präsentierte Methode den Aufwand für den Nutzer, die Render Parameter anzupassen, signifikant reduzieren kann. Trotzdem kann ein solcher regelbasierter Ansatz nicht mit einer manuellen Einstellung der Render-Parameter mithalten.

Related Work nochmal anschauen. Zum Bsp. ob aus der richtigen Perspektive geschrieben wurde.

2.2 Sehapparat

Das Auge ist der visuelle Sensor des Menschen und ermöglicht ihm das Sehen. Wahrnehmungsorientiertes Rendering nutzt gezielt Eigenschaften der visuellen Wahrnehmung des Menschen aus. Dafür ist es notwendig, ein gutes Verständnis des menschlichen visuellen Wahrnehmungsapparates zu besitzen. In diesem Abschnitt stelle ich einige Eigenschaften des visuellen Wahrnehmungsapparates vor, wie sie in [WSR+] vorgeführt werden. Abbildung 2.1 ist ein Modell des visuellen Wahrnehmungsapparates des Menschen. Das Modell unterteilt den Sehapparat des Menschen in Optik, Sensorik, Motorik, Verarbeitung, Speicherung und Aufmerksamkeit. Licht trifft auf die Augen und wird durch die Optik auf die Retina, die Sensorik, weitergeleitet. Hier wird der visuelle Input abgetastet und gefiltert. Dabei entstehen zwei Datenströme welche die Verarbeitung stereoskopischer Bilder über einen großen Blickwinkel mit unterschiedlichen räumlich unterschiedlicher Auflösungen ermöglichen. Die Retina (oder auch Netzhaut) ist mit dem visuellen Kortex verbunden. Die Signale werden über die visuellen Nerven komprimiert und zum visuellen Kortex transportiert. Dort werden sie von verschiedenen Bereichen im Gehirn verarbeitet. Speicherung und Aufmerksamkeit spielen dabei eine wesentliche Rolle.

Das visuelle Wahrnehmungssystem des Menschen hat wesentliche Limitierungen, die bei der Darstellung von Bildern gezielt genutzt werden können. Die Sehschärfe des ungleich auf der Retina verteilt und nur in in einem kleinen, zentralen Bereich ist die Sehschärfe maximal. Dieser Bereich wird Fovea oder auch Gelber Fleck genannt. Je weiter man sich von der Fovea nach außen hin entfernt, desto mehr nimmt die Sehschärfe ab. Der Bereich um die Fovea ist der periphere Bereich des Auges. Das Sichtfeld des visuellen Wahrnehmungsapparates des Menschen ist bei gerade

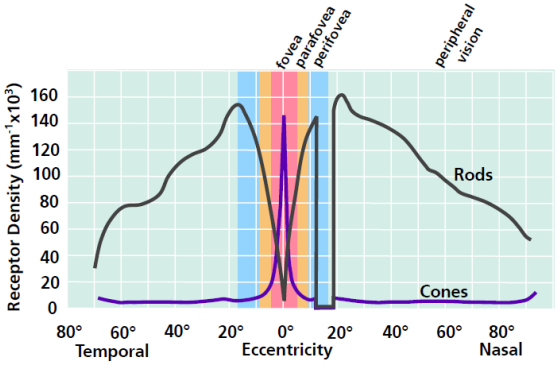

Abbildung 2.2: Verteilung der Photorezeptoren auf der Retina [**doi:10.1111/cfg.13150**]

gerichteten Blick horizontal bis circa 190 ° und mit Augenrotation bis zu 290 °. Visuelle Reize werden über das gesamte Sichtfeld wahrgenommen. Abhängig von dem zuständigen Bereich auf der Retina gibt es starke Unterschiede, wie die visuellen Reize über das Sichtfeld verteilt, verarbeitet werden. Durch das Verkleinern (Miosis) und Vergrößern (Mydriasis) der Pupille wird die Menge des einfallenden Lichts in das Auge gesteuert. Die Pupille nimmt dabei Größen zwischen 2 mm und 8 mm Durchmesser an.

Die Netzhaut (Retina) ist die photosensitive Schicht des Auges und besteht aus zwei Typen von Photorezeptoren, aus Zapfen und Stäbchen. Es sind circa 6×10^6 Zapfen und ungefähr 20 mal so viele Stäbchen auf der Retina verteilt. Stäbchen sind für die Helligkeitswahrnehmung verantwortlich. Zapfen sind für die Farbwahrnehmung verantwortlich. Man unterscheidet zwischen drei Zapfentypen: L-Zapfen für lange Wellenlängen, M-Zapfen für mittlere Wellenlängen und S-Zapfen für kurze Wellenlängen.

Die Fovea ist der Bereich um circa 5,2 ° um das Zentrum der Retina und besteht fast ausschließlich aus Zapfen. Die Anzahl der Zapfen nimmt aber nach außen hin stark ab. Der Bereich von circa 5,2 ° bis 9 ° wird als Parafovea bezeichnet. Der Bereich zwischen circa 9 ° bis 17 ° heißt Perifovea. Fovea, Parafovea und Perifovea sind für die zentrale Sicht verantwortlich. Alles außerhalb ist die periphere Sicht. Abbildung 2.2 zeigt die Dichteverteilung der Photorezeptoren auf der Retina. Die höchste Dichte der Stäbchen liegt bei circa 15 ° bis 20 ° um die Fovea herum. Ihre Anzahl verringert sich nach außen hin circa linear. Zapfen und Stäbchen sind sehr unterschiedlich auf der Reitna Verteilt. Beide folgen aber einem Poisson-Disc Verteilungsmuster. Die Dichte der Zapfen ist direkt mit der Sehschärfe verknüpft. Daher fällt die Sehschärfe auch nach außen hin stark ab. Bei 6 ° weg vom Zentrum beträgt die Sehschärfe schon nur noch ein Viertel der maximalen Sehschärfe. Die Sehschärfe hängt aber auch von der Kontraststärke der visuellen Reize ab. Dabei ist die Kontrastsensitivität von der Anzahl der auf den Reiz reagierenden neuronalen Zellen abhängig, welche ebenfalls nach außen hin stark abnimmt. Die Farbsensitivität ist von der Verteilung von Stäbchen und Zapfen abhängig. Die Zapfen zur Erkennung von grünem und rotem Licht sind vermehrt in der Fovea und eher weniger im peripheren Bereich verteilt. Von allen Zapfen sind lediglich neun Prozent zur Erkennung von blauem Licht. Diese sind vermehrt im peripheren Bereich verteilt als im Zentrum. Rezeptoren im Auge passen sich deutlich schneller an die Helligkeit als an die Dunkelheit an.

Das Auge ist dauerhaft in Bewegung. Sechs externe Muskeln ermöglichen es verschiedene Objekte von Interesse (OvI) in die Fovea zu bringen und zu fokussieren. Die wichtigsten Arten von Augenbewegungen sind: Sakkaden, Vestibular-Okularer Reflex, weiche Augenverfolgung *Smooth pursuit eye motion (SPEM)* und *coupled vergence-accommodation motion*.

Englische Begriffe richtig übersetzen.

Der Vestibular-Okulare Reflex passiert relativ schnell mit einer Latenz von 7 ms - 15 ms und ermöglicht auch bei schnellen Kopfbewegungen OvI zu fixieren. Der SPEM ermöglicht die weiche Verfolgung eines sich bewegendes Objektes. Bei der Wahrnehmung der Umgebung ist die Sakkade und Fixation die wichtigsten Eigenschaften des Auges. Eine Sakkade bezeichnet das schnelle springen von einem Ovi zu einem anderen. Dabei erreicht das Auge Geschwindigkeiten von bis zu 900 °/s. Die Sehsensivität ist während einer solchen Sakkade stark geschwächt (saccade suppression). Eine Fixation dauert zwischen 100 ms - 1.5 s. Sie tritt meist dann auf, wenn ein OvI genauer betrachtet wird und die Augen darauf ruhen. In natürlichen Szenen treten zwei bis drei Sakkaden pro Sekunde auf, mit jeweils durchschnittlich 250 ms Fixationszeit. Der räumliche Abstand zwischen den Fixierungen beträgt dabei circa 7 °. Ein Abstand von mehr als 30 ° wird als unangenehm empfunden und hat meist eine Kopfbewegung zur Folge. Auch während einer Fixierung macht das Auge wichtige kleine Bewegungen, sogenannte Tremor Bewegungen. Werden diese bewusst unterdrückt, resultiert dies in einem schwindenden Bild. Kleinere Augenbewegungen von bis zu 2.5 °/s haben kaum einen Effekt auf die Sehschärfe.

Das die Fovea einen sehr wichtigen Teil der visuellen Informationen liefert spiegelt sich auch darin wieder, dass über 30 Prozent des primären Sehverarbeitungsbereichs des Gehirns für die zentralen 5 ° des Sehfeldes, der Fovea, zuständig sind. Der periphere Bereich ist hier benachteiligt, liefert aber trotzdem einen großen und wichtigen Teil der Informationen. Besonders für das (frühe) Erkennen von Kontrasten, Objekten und Tieren. Aus dem peripheren Bereich werden auch wichtige kontextuelle Informationen geliefert. Dies ermöglicht unter Anderem die Vorverarbeitung von Informationen.

Die Aufmerksamkeit spielt eine wichtige Rolle in der Verarbeitung von visuellen Stimuli. Visuelles tunneling bezeichnet das längere Fokussieren auf einen bestimmten Punkt, wodurch ein Großteil der peripheren Informationen nicht mehr oder stark reduziert wahrgenommen werden. Die visuelle Auflösung reduziert sich circa linear für die ersten 20 °-30 °. Jedes lineare Modell für das visuelle Wahrnehmungssystem des Menschen ist aber immer nur eine Annäherung. Farben haben einen großen Einfluss auf die Wahrnehmung von Kontraste. Die Bewegungserkennung ist sowohl in der Fovea als auch im peripheren Bereich ähnlich oder genauso gut.

Grundlagen zum menschlichen Sehapparat überarbeiten: Teilweise sind Sätze zusammenhangslos aneinandergereiht. Hier eine bessere Verknüpfung finden.

2.3 Volumenrendering und Transferfunktion

Es ist nicht trivial, 3D Volumendaten auf einem 2D Bildschirm informativ zu präsentieren. In einem Volumen gibt es möglicherweise mehrere für den Betrachter interessante Objekte, welche sich durchaus gegenseitig überdecken können. Daher stellt sich die Frage, wie die Volumendaten auf dem Bildschirm projiziert werden und dabei für den Betrachter auch Informationen an verschiedenen

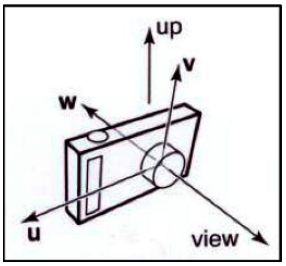

Abbildung 2.3: Illustration einer virtuellen Kamera im Raum [Dr 17]

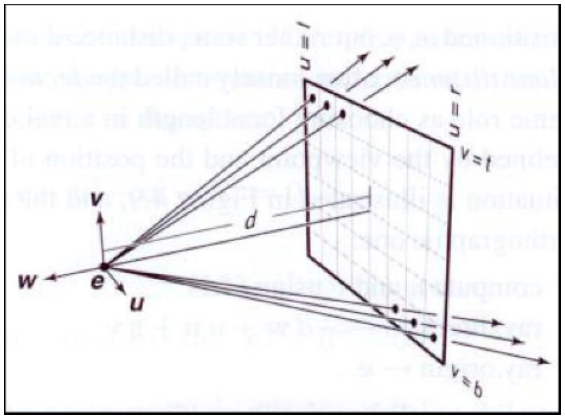

Abbildung 2.4: Illustration der virtuellen Kamera und der Bildebene [Dr 17]

Positionen innerhalb des Volumen sichtbar gemacht werden können. Raycasting ermöglicht in Verbindung mit einer Transferfunktion, ein Volumen auf eine 2D Ebene zu projizieren und dabei einzelne Bereiche des Volumens farbig hervorzuheben oder transparent erscheinen zu lassen.

2.3.1 Raycasting

Raycasting ist ein Verfahren um Volumendaten abzutasten und auf eine 2D Rastergrafik zu projizieren. Wie beim Raytracing werden beim Raycasting auch Sichtstrahlen verfolgt. Die Sichtstrahlen werden dabei ausgehend von einer virtuellen Kamera im Raum verfolgt. Die virtuelle Kamera, siehe Abbildung 2.3, ist im Raum an Position $\vec{e}$ positioniert, dies ist auch das Projektionszentrum, falls es keine orthogonale Projektion ist. Die Position $\vec{e}$ entspricht relativ der Position des Betrachters vor dem Bildschirm. Mit $\vec{u}$, $\vec{v}$ und $\vec{w}$ wird die Ausrichtung der Kamera im Raum eindeutig bestimmt. In einer Entfernung `d` von $\vec{e}$ aus in Richtung $-\vec{w}$ ist die Bildebene 2.4 Positioniert. Die Größe der Bildebene wird durch `l`, `r`, `b` und `t` bestimmt. Abhängig von der Größe des Bildschirms oder der Größe der gewünschten Rastergrafik, wird die Bildebene virtuell in die gewünschte Anzahl an Pixel in die Breite und Höhe unterteilt. Jeder dieser Teile entspricht nun einem Pixel der Rastergrafik. Nun kann für jeden Pixel ein Strahl, ausgehend von $\vec{e}$, in Richtung des entsprechenden Punktes auf der Bildebene, ausgesendet und verfolgt werden. Trifft ein Strahl Objekte in der Szene, wird dadurch ein Farbwert ermittelt, den der entsprechende Pixel annehmen kann und und die Szene dadurch auf den Bildschirm projiziert wird. Siehe auch Abbildung 2.5.

2.3.2 Volumenrendering

Ein Volumen kann mit Hilfe von Raycasting abgetastet und auf den Bildschirm projiziert werden. Hier wird wie beschrieben, für jeden Pixel ein Strahl in die Szene gesendet. Anstelle von verschiedenen Objekten gibt es nun ein Volumen welches aus vielen kleinen Voxeln besteht. Das Volumen ist ein dreidimensionales Bild, wobei statt 2D Pixel es aus 3D Voxel besteht. Ein Voxel ist eine Box in dem Volumen, mit einer Position, einer Ausdehnung in drei Dimensionen und einer Dichte. Wird ein

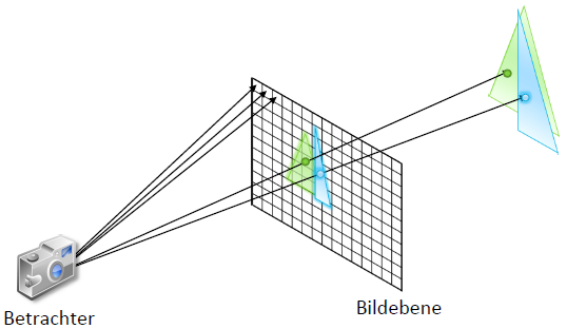

Abbildung 2.5: Illustration der Projektion von Objekten auf die Bildebene mit Hilfe von Raytracing. [Dr 17]

Strahl in das Volumen gesendet, so wird dieser in Intervallen abgetastet. Für jeden Abtastpunkt kann mit Hilfe einer Transferfunktion ein Farbwert ermittelt werden, welche zusammen einen Farbwert für den Pixel ergeben.

Anwendung von Raycasting zum Volumenrendering. Illustration von Rayasting und Volumenrendering mit mehreren Raysamples einfügen.

2.3.3 Transferfunktion

Die Transferfunktion ist ein mächtiges Werkzeug für die Visualisierung von Volumendaten. Mit Hilfe einer Transferfunktion können bestimmte Bereiche des Volumen verschieden stark ausgeblendet, oder auch farbig dargestellt werden. Die Transferfunktion bestimmt für einen Voxel, abhängig von seiner Dichte, einen Farb- und Opazitätswert. Dadurch können zum Beispiel Strukturen innerhalb des Volumens mit einer hohen Dichte farbig und kräftig hervorgehoben werden. Strukturen mit einer geringeren Dichte können transparent und weniger farblich dargestellt werden.

Funktion der Transferfrunktion im Volumenrendering erläutern. Unter Umständen ein Beispiel Bild mit einer Anwendung der Transferfunktion verwenden.

2.4 Eyetracking

Für einige wahrnehmungsorientierte Methoden ist es notwendig, den aktuellen Blickpunkt des Betrachters auf dem Bildschirm zu wissen und daher die Augenaktivitäten zu messen. In diesem Zusammenhang fällt meistens der Begriff `Eyetracking`. Aber was ist Eyetracking? Eyetracking ist das Messen von Augenaktivitäten eines Betrachters. In der Regel betrachtet der Betrachter dabei einen Bildschirm während seine Augenaktivität gemessen werden, wodurch Informationen über den Blickverlauf und die Fixationsdauer bestimmter Punkte auf diesem Bildschirm errechnet werden können. Diese Informationen ermöglichen es, Rückschlüsse über das betrachtete Bild zu ziehen, wie zum Beispiel, welche Objekte eines Bildes besonders interessant für den Betrachter sind. Eyetracking ermöglicht aber nicht nur das Messen von Daten um diese im Nachhinein auszuwerten, sondern auch das Nutzen solche Daten für interaktive Anwendungen in Echtzeit. Da in dieser Arbeit ein

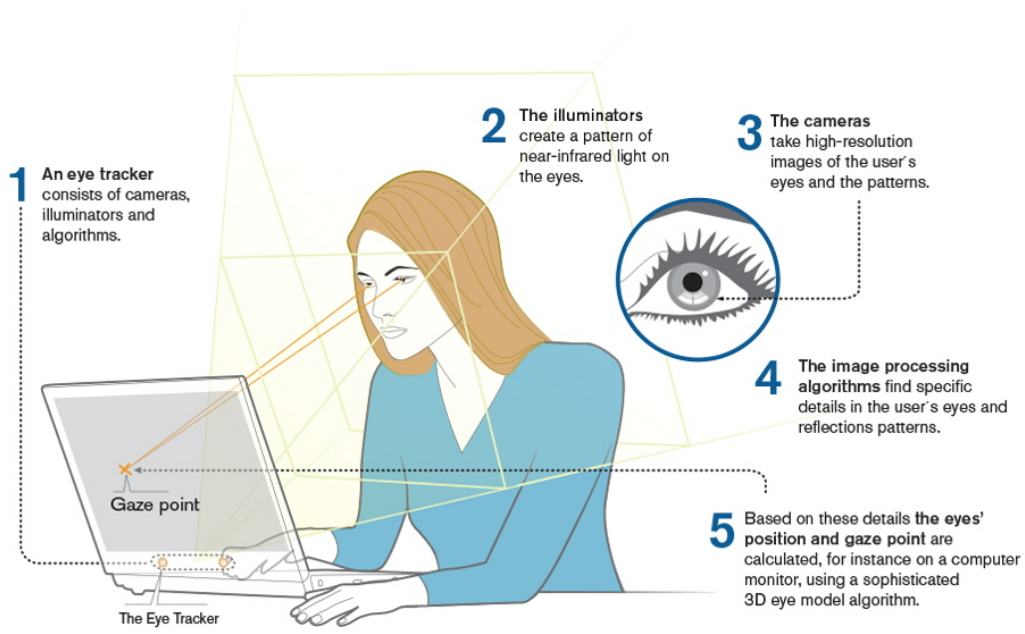

Abbildung 2.6: Illustration der Funktionsweise von Tobii Pro Eyetracker. [tobb]

externer Tobii Eye Tracker verwendet wurde, beziehen sich folgende Informationen hauptsächlich aber nicht ausschließlich auf Tobii Eyetracker. Die Firma Tobii AB schreibt auf ihrer Webseite zum Thema: Was ist Eyetracking?, dass Eye Tracking eine Technologie ist, die es ermöglicht, ein Gerät durch die natürliche Bewegung der Augen zu steuern [toba]. Dabei werden vier grundlegende Bestandteile des Eye Tracking, wie die Firma Tobii AB es realisiert, genannt. In der Regel benötigt man für das Messen von Augenaktivitäten zwei grundlegende Dinge, die auch in Abbildung 2.6 abgebildet sind: Eine Lichtquelle (2) und eine Kamera (3). Es gibt verschiedene Möglichkeiten, die Augenaktivitäten zu messen. Die am häufigsten genutzte Technik, welche auch von den Tobii Eyetrackern genutzt wird, ist *Pupil Centre Corneal Reflection (PCCR)*. Die Lichtquellen sind dabei auf die Augen gerichtet und erzeugen auf ihnen ein spezielles Reflektionsmuster. Die Kamera des Eyetrackers (3) nimmt mit einer hohen Abtastrate Bilder der Augen und ihrer Reflektionsmuster auf. Nun können spezielle Bildverarbeitungsalgorithmen (4) auf die erfassten Daten angewendet werden und die Reflektionspunkte auf den Bildern bestimmt werden. Anhand dieser Punkte und eines Modells des Auges kann die Blickrichtung der Augen, die Position der Augen im Raum und die Blickpunkte der Augen auf dem Bildschirm berechnet werden (4 + 5).

Die Position eines Auges im Raum wird als *Gaze origin* bezeichnet und wird bei Tobii Pro Eyetrackern im Nutzer Koordinatensystem angegeben, siehe Abbildung 2.7. Der Blickpunkt des Auges auf dem Bildschirm ist in der Regel das, was von größtem Interesse ist und wird als *Gaze point* bezeichnet. Der *Gaze point* wird im Aktiven Display Koordinatensystem angegeben, siehe Abbildung 2.8. Dieses Koordinatensystem hat den Ursprung an der Position oben links des aktiven Bildschirmbereichs, und der Punkt (1,1) ist an der Position rechts unten des aktiven Bildschirmbereichs. Die Umrechnung in Bildschirmkoordinaten funktioniert durch die Multiplikation mit

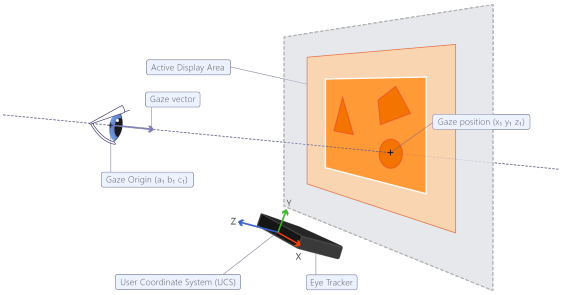

Abbildung 2.7: Nutzer Koordinatensystem [tobc]

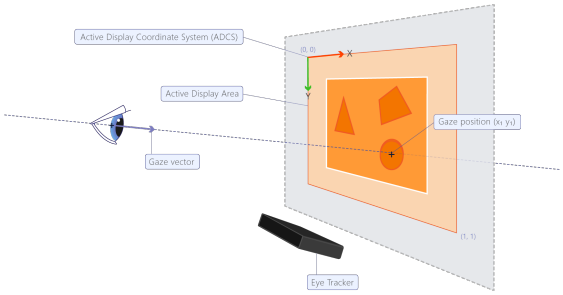

Abbildung 2.8: Aktives Display Koordinatensystem [tobc]

der Breite und Höhe des Bildschirms in Pixel. Die Koordinaten sind dann relativ zu dem aktiven Display. Dieses Display muss nicht unbedingt das einzige Display sein. Werden mehrere Bildschirme verwendet, muss möglicherweise ein Offset auf die Werte gerechnet werden, um die richtigen Bildschirmkoordinaten für den Bildschirm, auf dem die Anwendung dargestellt wird, zu erhalten.

In Abschnitt 2.2 wurden verschiedene Arten von Augenbewegungen angesprochen. Eine sehr wichtige Rolle spielen dabei Fixationen und Sakkaden. Bei der Analyse der visuellen Wahrnehmung des Menschen ergeben die Daten der Fixationen wichtige Informationen über Eigenschaften des visuellen Stimuli. Bei einer Fixation fixieren die Augen einen gewissen Punkt für eine vergleichsweise lange Zeit (100 ms - 1.5 s) und können dadurch viele Informationen des fixierten Objekts erfassen. Sakkaden hingegen sind kurze (20 ms - 40 ms) und schnelle Augenbewegungen zwischen zwei Fixationen. Für wahrnehmungsorientierte interaktive Anwendungen ist es daher wichtig, eine hohe Abtastrate und niedrige Latenz bei der Erfassung der Augenaktivitäten zu haben. Dies ermöglicht es, schnell zwischen einer Sakkade und Fixation zu unterscheiden und das nächste Bild, ohne eine merkbare Verzögerung und entsprechend des neuen Blickpunktes, darzustellen. So sollte bei der Methode des wahrnehmungsorientierten Renderings, wobei nur in dem fovealen Bereich das Bild mit hoher Auflösung dargestellt wird, ein Update ca. zwischen 5ms bis 60ms nach der Augenbewegung gestartet worden sein, um eine Bildveränderung nicht zu bemerken. Diese Zeit ist dabei abhängig davon, wie weit die unscharfe Fläche von der Fovea entfernt ist [WSR+].

Der in dieser Arbeit verwendete Tobii Pro Eyetracker hat eine Abtastfrequenz von 600 Hz. Dies entspricht einem Abtastintervall von 1.67 ms und ist vergleichsweise zu anderen aktuellen Eyetrackern recht hoch. Diese haben oft eine Abtastfrequenz von 60 Hz oder 120 Hz. Geräte mit Abtastraten im Frequenzbereich von 600 Hz und höher sind ausreichen schnell um Sakkaden messen zu können und haben eine entsprechend niedrige Latenz, welche es ermöglicht, eine Bildveränderung nicht wahrzunehmen.

Genaue Technische Daten des verwendeten Tobii Pro Eyetrackers nachschauen.

Hohe Abtastraten heißt aber auch viele Daten pro Sekunde. Für die Anwendung in dieser Arbeit ist es wichtig, den aktuellsten Blickpunkt des Auges zu kennen. Daher wird der letzte gemessene *gaze point* des Eyetrackers für das Berechnen des nächsten Bildes verwendet. Ein weiteres Problem bei vielen Messdaten ist es auch, dass es mehr fehlerhafte Messdaten gibt. Die Daten, die über die Tobii Pro SDK durch den Eyetracker geliefert werden, enthalten dafür einen Validity Code. Dieser Wert gibt an, ob eine Messung mit hoher Wahrscheinlichkeit richtige Daten enthält, oder nicht. Dadurch können wahrscheinlich fehlerhafte Daten früh abgestoßen werden.

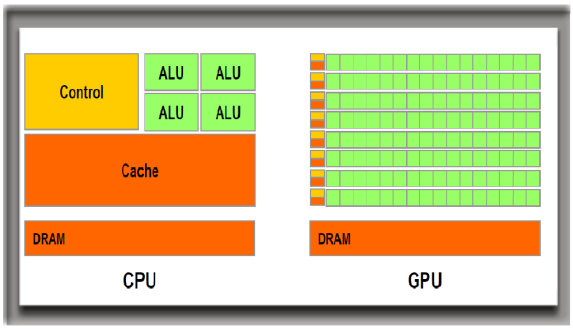

Abbildung 2.9: CPU vs. GPU. http://www.keremcaliskan.com/wp-content/uploads/2011/01/CPU-GPU-Structures1.png

2.5 GPU Architektur

Ein wichtiger Faktor, wenn es um interaktive Anwendungen geht, ist die Performanz. Viele Algorithmen haben essentielle Bestandteile welche ihre Komplexität nach unten beschränken. Die Einführung von Parallelität ist ein wichtiger Ansatz, um die Performanz von Anwendungen beziehungsweise ihrer Algorithmen weiter zu verbessern. Gerade Algorithmen in der Bildberechnung und -Verarbeitung eignen sich besonders um diese zu parallelisieren. In der Bildverarbeitung werden viele gleiche Berechnungen auf unterschiedlichen Daten ausgeführt. Graphics Processing Units (GPUs) oder auch Grafikkarten sind für diese Art von Berechnungen optimiert. Im Vergleich zu Central Processing Units (CPUs) besitzen GPUs eine deutlich höhere Anzahl an Prozessoren, siehe Abbildung 2.9. Aktuell besitzen Prozessoren meist vier oder acht Rechenkerne und eine Taktfrequenz von ca. 4 Ghz. GPUs hingegen haben mehrere tausend Berechnungseinheiten die gleichzeitig Berechnungen durchführen können, dafür aber bei ca. einem drittel der Taktfrequenz von CPUs.

Fast jeder Computer besitzt heutzutage hochparallele Einheiten, meist eine GPU. In den Anfängen waren GPUs sehr eng mit grafischen Berechnungen verbunden um die Berechnung von Farbwerten von Pixeln zu beschleunigen. Heute werden GPUs zur Beschleunigung von fast beliebigen Anwendungen genutzt. Programmierschnittstellen wie OpenCL oder CUDA ermöglichen die Nutzung von GPUs für allgemeine Berechnungen, oder auch General Purpose Computation on Graphics Processing Unit (GPGPU).

OpenCL

OpenCL (Open Computing Language) ist ein Standard für das allgemeine Programmieren für parallele CPU oder GPU-Platformen. Dabei stellt OpenCL eine Programmierschnittstelle für das koordinieren paralleler Berechnungen auf unterschiedlichen Platformen zur Verfügung, sowie eine Programmiersprache für die eigentliche Programmierung von Programmen, die auf parallelen Platformen wie einer GPU ausgeführt werden sollen.

Eine OpenCL Anwendung ist die Kombination des Codes eines Programms, das auf dem Host und den OpenCL Devices ausgeführt wird. Der Host ist dabei der Teil der Anwendung, der mit dem OpenCL Kontext über die OpenCL API kommuniziert. Ein Kontext ist die Umgebung, in

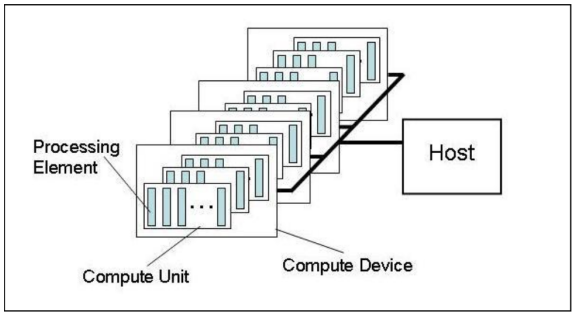

Abbildung 2.10: OpenCL Platform Modell ... ein Host, mehrere Compute Devices (ein Compute Device entspricht zum Beispiel einer GPU) mit je einer oder mehreren Compute Units. https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf

der ein OpenCL Kernel ausgeführt wird und weitere Eigenschaften definiert sind. Der Kernel ist eine Funktion die in einem OpenCL Programm geschrieben wurde und durch ein OpenCL Device ausgeführt wird. Ein Device entspricht meistens eine GPU oder CPU, die OpenCL implementiert.

OpenCL Platform Modell

Das Platform Modell von OpenCL ist eine Abstraktion davon, wie OpenCL die Hardware sieht. Die Beziehung zwischen seinen Einheiten und der Hardware ist dabei größtenteils von der verwendeten Hardware und ihrer Implementierung von OpenCL abhängig. Es besteht aus einem Host, welcher mit ein oder mehreren Devices verbunden ist. Diese sind unterteilt in Compute Units (CUs), welche weiter in Processing Elements (PEs) unterteilt sind. Berechnungen auf einem solchen Device werden in Processing Elements ausgeführt. Die Host-Anwendung gibt den Start des Kernel-Codes in Auftrag. Das OpenCL Device führt dann den Kernel Code auf den Processing Elements des Devices aus und hat dabei eine große Freiheit darüber, wie die Berechnungen auf den Processing Elements abgebildet werden. Falls die Processing Elements innerhalb einer Compute Unit die gleiche Folge von Befehlen ausführen, bezeichnet man ihren Befehlsfluss als konvergiert, sonst als divergiert.

OpenCL Ausführungsmodell

Der Host kann über Funktionen der OpenCL API mit einem Device über eine Command-Queue interagieren. Eine Command-Queue ist mit maximal einem Device verknüpft. Über die Command-Queue können Befehle zum starten eines Kernels, Transferieren von Daten zwischen Host und Device und Befehle zur Synchronisation ausgeführt werden. Ein Befehl durchläuft dabei immer sechs Zustände: Queued (Eingereiht), Submitted (Übermittelt), Ready (Bereit), Running (Ausführend), Ended (Beendet) und Complete (Abgeschlossen). Wird ein Kernel für die Ausführung übermittelt, wird für diesen ein Index-Raum definiert. Der Kernel selbst, die zugehörigen Parameter und die Parameter, die seinen Index-Raum definieren, definieren eine Kernel Instanz. Wird eine Kernel Instanz auf dem Device ausgeführt, wird für jeden Punkt in seinem Index-Raum eine Ausführung des Kernels gestartet. Eine solche Ausführung wird Work-Item genannt. Work-Items, die zu einer bestimmten Kernel Instanz gehören, werden von dem Device in Gruppen gehandhabt. Diese Gruppen heißen Work-Groups.

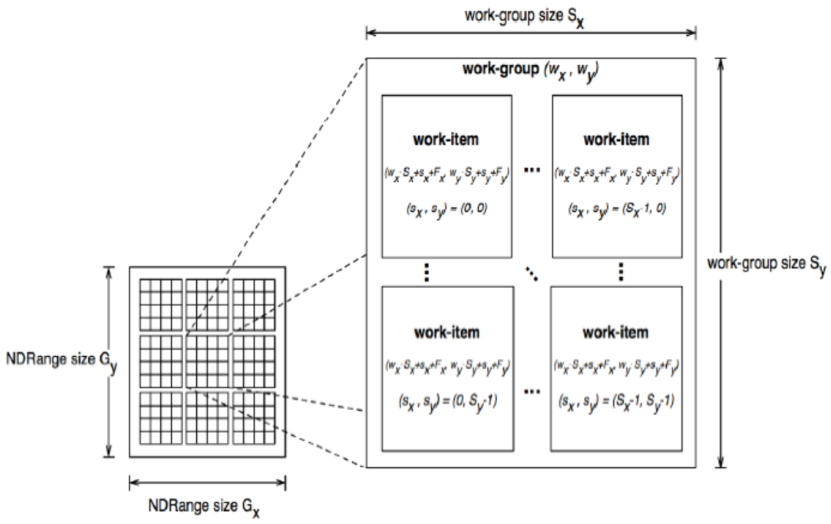

Abbildung 2.11: OpenCL NDRange-Mapping mit 2 Dimensionen. https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf

Der Index-Raum ist durch eine NDRange definiert. Dies ist ein N-dimensionaler Index-Raum, wobei N die Werte 1, 2 oder 3 annehmen kann. Eine NDRange wird dementsprechend durch drei Integer-Arrays der Länge N definiert: Die Ausdehnung des Index-Raums, ein Offset der Indices an dem sie starten und die Größe der Work-Groups, jeweils in N Dimensionen.

Jedes Work-Item hat ein N-Dimensionales Tupel, welches seine globale ID repräsentiert. Work-Groups erhalten ebenfalls N-Dimensionale Tupel als ID. Die Anzahl von Work-Groups ist abhängig von der definierten Größe von Work-Groups und der Größe des Index-Raums. Work-Items sind je einer Work-Group zugewiesen und haben eine loakle ID innerhalb ihrer Work-Group. Dadurch sind Work-Items eindeutig auf zwei verschiedene Arten definiert: Durch die globale ID beziehungsweise den globalen Index und durch die ID ihrer Work-Group zusammen mit ihrer lokalen ID in dieser Work-Group.

Wird die Ausführung einer Kernel-Instanz gestartet, werden die damit assoziierten Work-Groups in einen Work-Pool platziert und sind damit ausführungsbereit. Das Device entnimmt aus dem Work-Pool nach und nach Work-Groups und führt ein oder mehrere gleichzeitig aus. Da die Work-Groups aus dem Work-Pool in jeglicher Reihenfolge ausgeführt werden können, gibt es keinen sicheren Weg, verschiedene Ausführungen von Work-Groups zu synchronisieren.

Hardware Mapping

Ausrichtung der Bilder. Links in Fußnote oder Referenzen verschieben.

In der GPU Architektur gibt zwei große Konzepte, globalen Speicher und Streaming Multiprozessoren (SM). Globaler Speicher auf der GPU ist analog zu RAM für die CPU. Auf diesen kann von den Recheneinheiten der GPU und auch durch die Host-Anwendung auf der CPU zugegriffen werden.

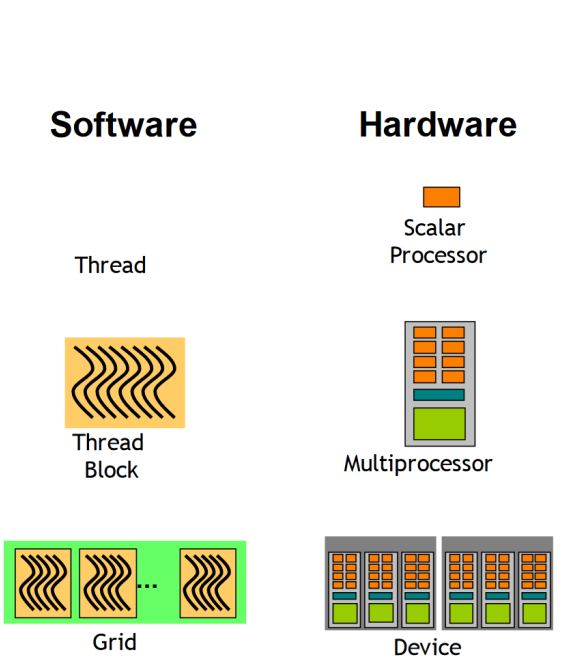

Abbildung 2.12: Ausführungsmodell NVIDIA GPU Architektur. http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf

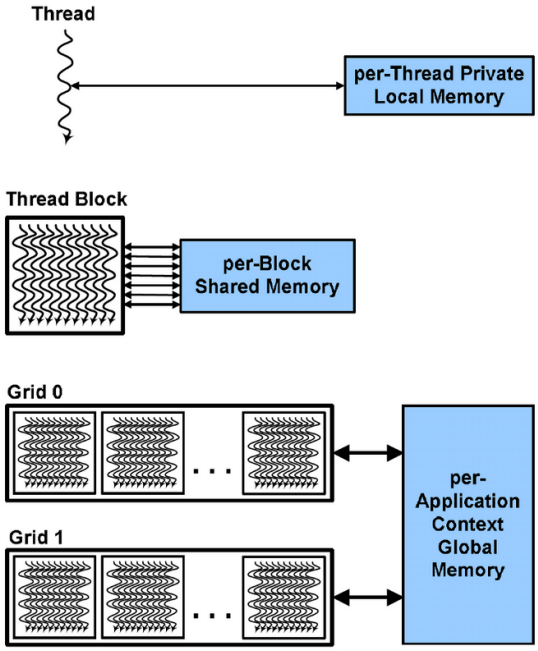

Abbildung 2.13: CUDA Thread- und Speicher Hierarchie. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

Eine GPU hat mehrere Streaming Multiprozessoren. Streaming Multiprozessoren führen die eigentlichen Berechnungen aus und haben je eigene Control Units, Register (lokaler Speicher), Ausführungs Pipelines und Caches. Ein Streaming Multiprozessor besitzt viele Skalare Recheneinheiten, die ähnlich zu einer ALU Berechnungen ausführen.

Im GPU Ausführungsmodell werden Threads von Skalaren Prozessoren ausgeführt. Ein Thread Block ist eine Gruppe von Threads und wird auf einem Multiprozessor ausgeführt. Dabei werden Threads innerhalb eines Thread Blocks ausschließlich innerhalb eines Multiprozessors ausgeführt. Es könnenn aber auch mehrere Thread Blöcke gleichzeitig auf einem Multiprozessor ausgeführt werden. Dies ist aber durch die verfügbaren Ressourcen limitiert.

Ein Thread Block besteht meistens aus 32-Threads und wird in einer NVIDIA Architektur Warp und in einer AMD Architektur Wavefront genannt. Ein solcher Thread Block wird physisch parallel auf einem Multiprozessor als Single Instruction Multiple Thread (SIMT) Ausführung ausgeführt. Die Ausführung eines Thread Blocks wird durch einen (Warp- / Thread-Block) Scheduler geregelt. Da Threads innerhalb eines Thread Blocks differenziert werden können, ermöglicht dies eine Single Instruction Mutlitple Data (SIMD) Ausführung. Ein Programm (Kernel) auf der GPU wird als Gitter aus Thread Blöcken gestartet.

OpenCL sieht die Hardware aus abstrakter Sicht und die exakte Beziehung des Platform Modells zu der Hardware wird von OpenCL nicht festgelegt. Das Platform Modell besteht aus einem Host, OpenCL Devices mit je mehreren Compute Units die jeweils ein oder mehreren Processing Elements enthalten. In der Realität entspricht der Host in der Regel dem Teil der Anwendung, der auf der CPU ausgeführt wird. Das OpenCL Device entspricht meist einer GPU und eine Compute Unit kann als Streaming Multiprozessor gesehen werden wobei die Processing Elements dementsprechend den Skalaren Prozessoren innerhalb eines Multiprozessors entsprechen.

Nach OpenCL entspricht eine Work-Group einer Menge von verwandten Work-Items, die alle innerhalb der selben Compute Unit arbeiten. Ein Work-Item kann von einem oder mehreren Processing Units als Teil einer Work-Group verarbeitet werden. Sie führen die selbe Kernel Instanz aus und haben gemeinsamen lokalen Speicher und Work Group Funktionen. Daher ist es naheliegend, das Work-Groups in Thread Blöcke unterteilt werden und alle Thread-Blöcke einer Work-Group auf dem selben Streaming Multiprozessor ausgeführt werden. In OpenCL können Work-Groups weiter in Sub-Groups unterteilt werden. Eine OpenCL Sub-Group ist eine implementationsabhängige Gruppierung von Work-Items innerhalb einer Work-Group. Diese sind eindimensional und haben, bis auf die letzte Sub-Group einer solchen Unterteilung, alle dieselbe Größe. Es ist ebenfalls naheliegend, dass eine solche Sub-Group einem Thread block entspricht.

Threads innerhalb eines Thread Blocks können also ausschließlich die selben Instruktionen ausführen. Müssen einige Threads innerhalb eines Thread Blocks aufgrund ihrer ID unterschiedliche Statements ausführen, resultiert dies darin, dass unter Umständen der gesamte Block alle Statements ausführt und am Ende der Berechnung die richtigen Ergebnisse auswählt.

Quelle dafür

Dies hat zur Folge, dass ein Thread Block der divergiert deutlich längere Ausführungszeiten hat, als ein konvergierender Thread Block. Bei der Implementierung eines OpenCL Kernels kann eine geschickte Strukturierung des Codes daher das gleiche Ergebnis mit besserer Performanz generieren.

Die Wahl der Work-Group Größe ist ebenfalls wichtig. Da Thread Blöcke aktuell meist eine Größe von 32 Threads besitzen und unter der Annahme, dass Work-Groups in Thread Blöcke unterteilt werden, macht es Sinn, die Größe einer Work-Group als vielfaches von 32 zu wählen. Ansonsten kann es passieren, dass Thread Blöcke zur Ausführung gestartet werden, bei denen eine große Anzahl an Threads kein Teil der Berechnung sind und Ressourcen dadurch verschwendet werden.

Ein weiteres Problem sind Speicher Konflikte bei der Ausführung eines Kernels. Lokaler Speicher von Multiprozessoren ist in Speicher Bänke unterteilt. Versuchen Threads eines Thread Blocks auf verschiedene Adressen innerhalb der selben Speicher Bank zuzugreifen, entstehen Speicher Konflikte, da eine Speicher Bank nur eine ihrer Adressen gleichzeitig adressieren kann. Ausführungen eines Thread Blocks mit Speicher Konflikten müssen serialisiert werden und dauern daher deutlich länger. Eine Ausnahme ist es, falls die Threads eines Blocks alle auf dieselbe Adresse innerhalb einer Speicher Bank lesend zugreifen. In diesem Fall wird der gelesene Wert an alle Threads gleichzeitig übertragen.

Quellen einfügen.

Bilder Referenzieren.

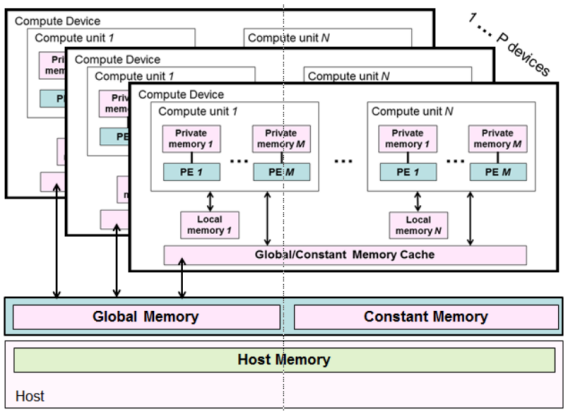

Abbildung 2.14: OpenCL Speicher Modell https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf

3 Entwurf

Die Implementierung dieser Arbeit beruht auf einem Visual Studio Projekt zum Volumen Rendering von Valentin Bruder. Der erste Abschnitt des Entwurfskapitels beinhaltet den Ausgangspunkt der Implementierung dieser Arbeit in Form einer Beschreibung des ursprünglichen Projekts. Dies umfasst die allgemeine Architektur der Anwendung und die Umsetzung des Volumenrenderings durch eine Implementierung eines Raycasters als OpenCL Kernel. Der zweite Abschnitt des Kapitels umfasst Überlegungen für die Erweiterung des Projektes, bezüglich des Ausgangspunktes wie er im ersten Abschnitt beschrieben wurde und dem Ziel dieser Arbeit, sowie die daraus entstandenen Arbeitspakete und die Ansätze für ihre Integration in das bestehende Projekt.

3.1 Projekt

Das Visual Studio Projekt, welches als Ausgangspunkt dient, ist eine auf QT basierende Anwendung. QT ist ein vollständiges Cross-Platform Softaware Development Framework, welches in c++ entwickelt wurde, und ermöglicht die einfache Erstellung von Anwendungen mit Benutzeroberflächen und bietet außerdem eine Vielzahl von Bibliotheken, die für eine leichtere und schnellere Entwicklung von Programmen genutzt werden können.

Die Hauptelemente für die QT Benutzeroberfläche sind QT Widgets. Widgets können Daten darstellen und Nutzereingaben erkennen. Außerdem stellt ein Widget selbst ein Container für weitere Widgets dar, welche in diesem gruppiert werden. Innerhalb eines Widgets können Elemente platziert werden, welche Informationen, mögliche Operationen oder Nutzereingaben repräsentieren. Für die bequeme Erstellung der grafischen Oberfläche bietet QT die Anwendung QT Designer. Der QT Designer ermöglicht es, Widgets und andere Bausteine der grafischen Oberfläche per Drag und Drop anzuordnen. Die durch den QT Designer definierte Oberfläche kann abgespeichert werden und wird von QT verwendet, um eine c++ Header File zu erstellen. Dies ermöglicht es die verschiedenen Elemente der grafischen Oberfläche an die Logik der Anwendung zu binden.

Quelle zu QT einfügen.

Das Projekt ist dementsprechend in c++ geschrieben und die grafische Oberfläche wurde mit Hilfe des QT Designers erstellt. Die Oberfläche selbst hat eine Menüleiste, die es unter Anderem ermöglicht, Volumendaten oder Transferfunktionen zu laden oder auch aktive Transferfunktionen zu speichern sowie das Erstellen eines Screenshots des zuletzt berechneten Bildes. Den Großteil der grafischen Oberfläche wird durch das Volumenrenderwidget ausgefüllt. Das Volumenrenderwidget ist ein QT OpenGL Widget und kann für das Darstellen von OpenGL Grafiken verwendet werden. Die berechneten Grafiken werden mit Hilfe des Volumenrenderwidgets dargestellt. Neben dem Volumenrenderwidget gibt es noch ein Widget, welches in drei weitere Widgets unterteilt ist, mit denen Parameter für das Volumenrendering gesetzt werden können. Das erste von ihnen ermöglicht

das variieren der Abtastrate im Bildraum, also die Anzahl der Strahlen, die ausgesendet werden, sowie das Setzen der allgemeinen Abtastrate der ausgesendeten Strahlen. Außerdem können hier weitere Rendering Parameter festgelegt werden, wie die Hintergrundfarbe oder ob Voxel beim Abtasten interpoliert werden. Das zweite Widget ist ein Farbenrad, welches für die einfache Auswahl der Farben einzelner Kontrollpunkte der Transferfunktion verwendet werden kann. Das dritte Widget ermöglicht schließlich das Setzen von Kontrollpunkten der Transferfunktion innerhalb eines Diagramms. Die x-Richtung gibt die Dichte, auf die sich ein Kontrollpunkt bezieht an. Die y-Richtung gibt seinen Opazitätswert an. Die Werte zwischen zwei Kontrollpunkten werden entweder linear oder quadratisch interpoliert. Daher gibt es immer mindestens einen Kontrollpunkt für den Dichtewert null und einen Kontrollpunkt für den Dichtewert eins.

Bitte auf Richtigkeit von diesem Teil prüfen, wie: Interpolation von Voxel und Einstellen der Transferfunkton.

Das Volumenrenderwidget ist ein OpenGL Widget und für die Darstellung der berechneten Bilder zuständig. Das QT Framework erlaubt es, das Widget in c++ Code mit Logik zu verknüpfen. Dafür existiert in dem Projekt eine Klasse VolumeRenderWidget, welche von QOpenGLWidget erbt. Die grundsätzliche Funktionalität zum Rendern in diesem Widget wird durch die Methode `paintGL()` ausgeführt. Innerhalb dieser Methode wird der Code geschrieben, der für die Darstellung des Bildes nötig ist. Das Darstellen auf dem Bildschirm beziehungsweise in dem Widget wird durch OpenGL realisiert. OpenGL zeichnet dabei aber lediglich eine durch einen OpenCL Kernel zuvor generierte Textur auf ein Fullscreen Quad. Das Management von OpenCL wird hier durch ein Objekt der Klasse `volumerendercl` geregelt. Innerhalb der `paintGL()` Methode wird eine Methode dieses Objekts zum Starten des OpenCL Kernels für den Raycast des Volumenrenderings aufgerufen. Das `volumerendercl` Objekt regelt die Handhabung der verschiedenen Parameter für den Kernel und die Unterteilung der übergebenen Anzahl an Strahlen in x- und y-Richtung, welche der Anzahl zu startenden Work-Items entspricht, in Work-Groups. Außerdem startet es den Kernel, synchronisiert einen gemeinsamen Stopp und speichert die benötigte Zeit der letzten Ausführung des Kernels. Die berechneten Werte der einzelnen Work-Items können bei der Ausführung des Kernels direkt in die OpenGL Textur geschrieben werden. Daher kann nach der Ausführung der aufgerufenen Methode des `volumerendercl` Objekts die Textur direkt gezeichnet werden. Mit Hilfe von QT Funktionen und der Information über die Ausführungszeit des Kernels werden anschließend noch ein paar Overlays gezeichnet. Unter Anderem eine Anzeige der ungefähren möglichen Anzahl an Bildern pro Sekunde der letzten Ausführungen, um die Ausführungsdauer des Kernels abschätzen zu können.

Raycaster

Der eigentliche Raycast passiert in einem OpenCL Kernel. Die OpenCL Objekte werden von dem `volumerendercl` Objekt gehandhabt, dessen Methoden innerhalb der `paintGL()` Methode aufgerufen werden. Das `volumerendercl` Objekt regelt auch die Übergabe der Parameter an den Raycast Kernel. Dies sind Parameter wie Volumendaten, Transferfunktionswerte und Strahlabtastrate zum lesen, sowie eine 2D-Textur zum schreiben für die berechneten Farbwerte der einzelnen Work-Items. Jedes Work-Item besitzt eine 2D-ID, die einer Position in der Ausgabetextur zugewiesen bekommt. Ein Work-Item ist für das Abtasten eines Strahls verantwortlich. Ein Strahl hat als Ursprung die Position der Kamera und entsprechend seiner ID, beziehungsweise Texturkoordinaten, wird seine Richtung bestimmt. Abhängig von der Abtastrate des Strahls, berechnet sich die Schrittgröße für das Abtasten.

Ausgehend von dem Schnittpunkt des Strahls mit der Bildebene wird nun in einer Schleife der Strahl schrittweise abgetastet. Dabei wird für jeden Schritt die aktuelle Position bestimmt, welche normiert und dann dafür genutzt wird, um in dem 3D-Volumen Objekt den Dichtewert für diese Position zu bestimmen. Mit dem Dichtewert und den Daten der Transferfunktion wird anschließend ein Farbwert berechnet. Dieser Farbwert wird mit den bisherigen gesammelten Farbwerten des Strahls verrechnet, so dass am Ende der Abtastschleife ein einziger Farbwert für den Strahl existiert. Der Farbwert wird zum Schluss an die entsprechende Texturkoordinate in der Ausgabetextur gespeichert.

Bild für das Abtasten des Strahls einfügen. Entweder hier oder bei Basics mit Volumenrendering und dann dorthin verweisen.

Der Raycast Kernel hat außer der grundlegenden Raycast Funktion noch weitere Eigenschaften, die die Performanz der Ausführung und die Qualität des Bildes verbessern. So kann *Empty Space Skipping* aktiviert werden, um größere Bereiche mit rein transparenten Voxeln zu überspringen. Dies wird mit Hilfe eines zuvor gröber Berechneten Volumen ermöglicht. Da das Volumen nur eine begrenzte Auflösung hat aber an einer beliebigen Position ein Wert aus dieser 3D-Textur abgerufen werden kann, muss angegeben werden, wie dieser Wert abhängig von den umliegenden Voxel bestimmt wird. Daher kann hier gewählt werden, dass beim Auslesen des Dichtewerts an einer bestimmten Position des Volumens, dieser interpoliert wird. Außerdem kann eine Orthografische Sicht des Volumens aktiviert werden, indem die Strahlen parallel ausgesendet werden und für solide Oberflächen gibt es die Möglichkeit, den Effekt der *Ambient Occlusion* darzustellen.

Beschreibung der Funktionsweise des Raycasters für das Volumenrendering in dem Projekt.

3.2 Arbeitspakete und Integration

Ausgehend von der in Abschnitt 3.1 beschriebenen Ausgangslage des Projekts, wurden einige Vorüberlegungen und Arbeitspakete erstellt, welche das Ziel hatten, das Projekt so zu erweitern, dass die Aspekte des wahrnehmungsorientierten Volumenrendering veranschaulicht und umgesetzt werden können. Im folgenden werden die für dieses Ziel entstandenen Vorüberlegungen und daraus erstellte Arbeitspakete aufgeführt und die dazugehörigen Ansätze zur Integration in das bestehende Projekt skizziert. Genauere Angaben zur Implementierung bestimmter Arbeitspakete werden im Kapitel 4 vorgestellt.

Einarbeitung in das Projekt

Das erste Arbeitspaket, welches nicht zu vernachlässigen ist, war die Einarbeitung in das Projekt beziehungsweise in die bestehende Implementierung. Dies erforderte das Einarbeiten in einige Grundlagen der c++ Programmierung sowie das Einarbeiten in den grundlegenden Umgang mit dem QT Framework. Da das Projekt ein Visual Studio Projekt ist und Programmierschnittstellen wie OpenCL oder auch QT verwendet, war eine kleine Einarbeitung in das richtige Verlinken der Bibliotheken mit dem Projekt auch Teil von diesem Arbeitspaket.

Simulieren der Blickposition

Um die Eigenschaften des visuellen Wahrnehmungssystems des Menschen auszunutzen, dass die Genauigkeit des Auges außerhalb des zentralen Bereichs stark abnimmt, ist es notwendig, die Blickposition beziehungsweise den fokussierten Punkt auf dem Bildschirm zu kennen. Ein Eyetracker kann dies messen und die Daten der Anwendung zur Verfügung stellen. Da die Einbindung eines Eyetrackers für die ersten Arbeitsschritte, wie das Implementieren erster Versuche, den Raycast Kernel wahrnehmungsorientiert umzuschreiben, nicht notwendig ist, sondern zum Teil auch erschwert, kann der Blickpunkt vorerst sehr gut mit der Maus simuliert werden. QT Widgets können auf Maus und Tastatureingaben reagieren. Daher war das erste Arbeitspaket das Erkennen von Mausbewegungen innerhalb des Volumerenderwidgets und das Abspeichern der letzten erkannten Mausposition in einer globalen Variable. Zusätzlich musste diese dem OpenCL Kernel, der den Raycast ausführt, zur Verfügung gestellt werden. Daher wurde die Mausposition dem OpenCL Kernel als Parameter übergeben.

Reduzierung der Strahlabtastrate im fovealen Bereich

Da die Mausposition dem Kernel nun zur Verfügung steht, können erste Implementierungsversuche für einen wahrnehmungsorientierten Raycast unternommen werden. Das ursprüngliche Projekt stellt zwei Parameter zur Verfügung, welche auf zwei verschiedene Arten die Ausführungszeit des Kernels beeinflussen. Die Abtastrate der jeweiligen Strahlen und die Anzahl der Strahlen in x- und y-Richtung. Da das Verändern der Anzahl an Strahlen in x- und y- Richtung das gesamte Bild betrifft und dies nicht einfach abhängig von dem Abstand zur Mausposition verändert werden kann, ist die Anpassung der Abtastrate einzelner Strahlen für den Anfang einfacher zu gestalten. Aus diesem Grund heraus entstand das nächste Arbeitspaket. Dieses umfasst die Verwendung der an den Raycast Kernel übergebenen Mausposition, um die Abtastrate der jeweiligen Strahlen, abhängig von der Distanz der Bildposition des Strahls zu der Position des Mauszeigers, zu reduzieren. Die Anpassung des Kernels hat zur Folge, dass für jeden Strahl abhängig seiner Distanz zum Mauszeiger eine eigene Abtastrate berechnet wird. Aufgrund dessen, dass wie im Abschnitt 3.1 die Work-Items den Texel zugeordnet sind und die Work-Groups quadratisch angeordnet sind, sollte dies bewirken, dass die Work-Items innerhalb der selben Work-Group eine ähnliche Abtastrate für ihren Strahl berechnen und die gesamte Work-Group früher terminieren kann. Da der Kernel erst beendet wird, wenn alle Work-Groups ihre Arbeit abgeschlossen haben, bewirkt eine schnellere Ausführung einzelner Work-Groups eine insgesamt schnellere Ausführung des Kernels. Dementsprechend bewirkt dies auch eine bessere Performanz beim Berechnen des Bildes. Die Mausposition wurde in Bildkoordinaten, bezüglich des Volumerenderwidgets dem Kernel übergeben. Da die Anzahl der gestarteten Work-Items den Ausmaßen des Volumerenderwidgets entspricht, entspricht auch die ID eines Work-Items einer Bildkoordinate. Daher konnte die Entfernung der Work-Items zum Mauszeiger einfach berechnet werden und die Abtastrate abhängig von dieser Distanz angepasst. Erste Tests der Implementierung haben eine Erhöhung der im Volumerenderwidget durchschnittlichen angezeigten Bilder pro Sekunde ergeben. Die Verminderung der Abtastrate machte sich nur bei dünnen Volumen visuell bemerkbar, da hier die einige Strahlen einen Teil des Volumen teilweise gar nicht abgetastet haben.

Reduzierung der Strahldichte im fovealen Bereich

Trotz der Reduzierung der Abtastrate der Strahlen, wird weiterhin die gleiche Anzahl an Work-Items gestartet. Dies ermöglicht eine weitere Möglichkeit, die Ausführungszeit des Kernels zu reduzieren, indem die Anzahl der Strahlen und somit auch die Auflösung des berechneten Bildes variiert wird. Weniger zu berechnenden Strahlen bedeutet hier auch weniger benötigte Work-Items und Work-Groups, die ausgeführt werden müssen. Weniger Work-Groups bedeutet dann auch eine geringere Ausführungszeit des Raycast Kernels.

Die erste Überlegung diesbezüglich war es, eine Art virtuelle Linse vor die Bildebene zu setzen, die die Strahlen so auf der Bildebene verteilt, dass an der Mausposition eine höhere Strahldichte existiert und diese mit größerem Abstand zur Mausposition abnimmt. So könnte bei einer geringeren Anzahl an Strahlen, an der Mausposition, die den Blickpunkt simuliert, trotzdem die maximale Auflösung erreicht werden. Die Bildpunkte die dann nicht direkt von einem Strahl abgedeckt werden, müssten interpoliert werden. Dieser Ansatz wurde aber verworfen, da die Implementierung der virtuellen Linse und der anschließenden Interpolation sich als zu aufwändig erwies und es deutlich einfacher umzusetzende Alternativen gibt.

Eine Alternative ist es, zwei Mal den Raycast Kernel hintereinander, mit jeweils unterschiedlichen Auflösungen, also einer unterschiedlichen Anzahl an Work-Items, zu starten. Dies wurde das nächste Arbeitspaket: Das Berechnen des Bildes in zwei verschiedenen Auflösungen und die anschließende Zusammenfügung beider Bilder zu einem. Für die Integration wurde wie in dem Arbeitspaket gefordert, das Bild in zwei verschiedenen Auflösungen nacheinander berechnet. Dafür konnte die bisherigen Funktionen des `volumerendercl` Objektes weiter verwendet werden. Es wurden aber zwei unterschiedliche OpenGL Texturen für die Ausgabe des Kernels verwendet. Zuerst mit nur einem viertel der Auflösung und anschließend mit der normalen Auflösung. Innerhalb eines festgelegten Quadrats um den Mauszeiger herum ist die Auflösung des Bildes normal und außerhalb davon hat diese nur ein viertel der normalen Auflösung. Bei der ersten Berechnung wird der Teil innerhalb von diesem Quadrat nicht berechnet und entsprechend wird bei der zweiten Berechnung der Teil des Bildes außerhalb des Quadrats nicht berechnet. Dadurch werden so wenige Bildpunkte wie möglich doppelt berechnet. Da das erste Bild nur mit einem viertel der Auflösung berechnet wurde und die ID der Work-Items nicht mehr mit den Bildkoordinaten übereinstimmte, wurde die Mausposition und die Ausmaße des Quadrates sowohl bei der Berechnung des Bildes, als auch bei der Zusammenfügung beider Bilder normalisiert.

Indize-Mapping

Das Ziel eines weiteren Ansatzes war es, statt nur zwei verschiedene Auflösungen zu verwenden, welche in Form eines Quadrats übereinander gelegt werden, nun drei verschiedene Bereiche des Bildes mit je unterschiedlichen Auflösungen zu berechnen. Ein äußerer Bereich und zwei innere Bereiche, die die Form von Ellipsen haben. Die drei Bereiche ergeben dann das gesamte Bild.

Die erste Überlegung für diesen Ansatz war es, wie bei der Umsetzung des vorherigen Arbeitspaketes, auch hier drei Mal eine Berechnung für das gesamte Bild zu starten, aber jeweils die Teile, die außerhalb des Bereichs der aktuellen Berechnung liegen, zu discarden also diese nicht zu berechnen. Ein früher erster Test von diesem Ansatz hat gezeigt, dass die Performanz deutlich schlechter war,

als bei der vorherigen Methode. Trotzdem zeigten die aufaddierten Ausführungszeiten der Kernels gute Werte. Ein mögliches Problem dafür ist, dass der Overhead für das Starten der Kernels bei drei verschiedenen Ausführungen doch einen Einfluss auf die Performanz nimmt.

Da bis zu drei Ausführungen des Raycast Kernels hintereinander nicht die gewünschte Performanz erbrachten, wurde dies verworfen und eine weitere Möglichkeit, das oben genannte Ziel zu erreichen, überlegt. In der zweiten Überlegung ging es nun darum, mit nur einem Raycast Kernel Aufruf eine Berechnung für das ganze Bild zu starten. Jedoch werden in den äußeren zwei Bereichen nur in x- und y-Richtung nur jede dritte beziehungsweise jede zweite Berechnung eines Work-Items ausgeführt. Die anderen Work-Items werden entsprechend früh discarded. Ein zweiter Kernel soll die fehlenden Bildpunkte durch bipolare Interpolation der umliegenden Bildpunkte ausfüllen, so dass ein einzelnes Bild mit normaler Auflösung entsteht. Hier hat ein erster Test, bei dem lediglich ein Raycast Kernel aufgerufen wurde aber in x- und y-Richtung jedes zweite Work-Item discarded wurde gezeigt, dass dies keinerlei Performanz bringt. Dies liegt daran, dass durch die Anordnung der Work-Groups keine Work-Group existierte, die nur aus Work-Items bestand, die discarded wurden. Da vermutlich eine Work-Group auf einen Thread-Block abgebildet wird und ein Thread-Block so lange ausführt, bis alle seine Thread terminiert sind, wurde bei diesem Ansatz zwar nur ein viertel der Berechnungen ausgeführt aber trotzdem konnte kein Thread-Block schneller seine Arbeit verrichten, wodurch die Ausführungszeit des Kernels gleich blieb.

Dies motivierte das nächste Arbeitspaket. Das Umsetzen des oben genannten Ziels, indem nur so viele Work-Items, wie benötigt, gestartet werden und das Abbilden dieser Work-Items auf die entsprechenden Bildpunkte. Mit der Motivation, dass so wenige Work-Items wie möglich innerhalb einer Work-Group auf die Ausführung anderer Work-Items warten müssen und die Thread-Blöcke möglichst konvergieren. Zusätzlich gehörte dazu, dass ein weiterer Kernel geschrieben wird, der die anschließende Interpolation der Bildpunkte ausführt.

Die Integration von diesem Arbeitspaket erfordert die Berechnung der mindestens benötigten Anzahl an Work-Items, das Unterteilen in einen x- und y-Wert für die Dimensionen der Work-Items, zur Ausführung durch den Kernel. Anschließend muss der Raycast-Kernel mit der zuvor berechneten Anzahl an Work-Items in x- und y-Richtung gestartet werden sowie danach der Kernel für die Interpolation. Der Interpolation Kernel wird aber mit den Dimensionen des zu berechnenden Bildes für die Dimension der Work-Items gestartet.

Auswahl des Eyetrackers

Erstellen von Messwerten

Darstellen der Messwerte

Vorüberlegungen zur Umsetzung und daraus entstandene Arbeitspakete beschreiben.

4 Implementation

Implementation der drei Ansätze.

5 Ergebnisse

Messbare Ergebnisse der Arbeit. (Performanzmessungen..).

Diskussion

Diskussion und Bedeutung der Ergebnisse.

6 Zusammenfassung und Ausblick

Hier bitte einen kurzen Durchgang durch die Arbeit.

Ausblick

...und anschließend einen Ausblick

Literaturverzeichnis

[Dr 17] D. G. R. Dr. Michael Krone. *Vorlesungsfolien in Computergraphik, Raytracing*. Institut für Visualisierung und Interaktive Systeme der Universität Stuttgart, 2016 / 2017 (zitiert auf S. 15, 16).

[ER] R. Englund, T. Ropinski. „Quantitative and Qualitative Analysis of the Perception of Semi-Transparent Structures in Direct Volume Rendering“. In: *Computer Graphics Forum* 37.6 (), S. 174–187. DOI: 10.1111/cgf.13320. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13320. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13320 (zitiert auf S. 11).

[GFD+12] *Foveated 3D Graphics*. ACM SIGGRAPH Asia, Nov. 2012. URL: https://www.microsoft.com/en-us/research/publication/foveated-3d-graphics/ (zitiert auf S. 10).

[LME06] A. Lu, R. Maciejewski, D. S. Ebert. „Volume Composition Using Eye Tracking Data“. In: *Proceedings of the Eighth Joint Eurographics / IEEE VGTC Conference on Visualization*. EUROVIS'06. Lisbon, Portugal: Eurographics Association, 2006, S. 115–122. ISBN: 3-905673-31-2. DOI: 10.2312/VisSym/EuroVis06/115-122. URL: http://dx.doi.org/10.2312/VisSym/EuroVis06/115-122 (zitiert auf S. 11).

[LW90] M. Levoy, R. Whitaker. „Gaze-directed Volume Rendering“. In: *Proceedings of the 1990 Symposium on Interactive 3D Graphics*. I3D '90. Snowbird, Utah, USA: ACM, 1990, S. 217–223. ISBN: 0-89791-351-5. DOI: 10.1145/91385.91449. URL: http://doi.acm.org/10.1145/91385.91449 (zitiert auf S. 10).

[toba] tobii. *What is Eyetracking?* Tobii AB. URL: https://www.tobii.com/tech/technology/what-is-eye-tracking/ (besucht am 12. 09. 2018) (zitiert auf S. 17).

[tobb] tobiipro. *How to tobii eye trackers work*. Tobii AB. URL: https://www.tobiipro.com/learn-and-support/learn/eye-tracking-essentials/how-do-tobii-eye-trackers-work/ (besucht am 12. 09. 2018) (zitiert auf S. 17).

[tobc] tobiisdk. *Eyetracking Common Concepts, Tobii Pro SDK*. Tobii AB. URL: http://developer.tobiipro.com/commonconcepts.html (zitiert auf S. 18).

[WSR+] M. Weier, M. Stengel, T. Roth, P. Didyk, E. Eisemann, M. Eisemann, S. Grogorick, A. Hinkenjann, E. Kruijff, M. Magnor, K. Myszkowski, P. Slusallek. „Perception-driven Accelerated Rendering“. In: *Computer Graphics Forum* 36.2 (), S. 611–643. DOI: 10.1111/cgf.13150. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13150. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13150 (zitiert auf S. 12, 18).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift