Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 132

Ein hybrider, paralleler Volumen-Renderer für die Darstellung großer Datensätze auf hochauflösenden Displays

Sven Schnaible

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Thomas Ertl
Betreuer/in:	Dipl.-Inf. Christoph Müller
Beginn am:	07.05.2014
Beendet am:	06.11.2014
CR-Nummer:	I.3.2, I.3.7

Kurzfassung

Die Universität Stuttgart besitzt eine Großleinwand, die mit einem Cluster von zehn Knoten betrieben wird. Zur Berechnung der Bilddaten steht ein weiteres Cluster mit 64 Knoten zu Verfügung. Solche Systeme benötigen speziell angepasste Software. In dieser Arbeit wurde eine COM-Komponente entwickelt, mit der die Volumenvisualisierung einfach parallelisiert werden kann. Darauf aufbauend wurden ein System mit statischer Lastverteilung und ein System mit einer Job-Schlange entwickelt und miteinander verglichen. Beide Systeme sind in der Lage eine hybride Aufteilung – also des Bildraums und des Objektraums – vorzunehmen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Unter Volumen-Rendering versteht man eine Menge von Techniken, um ein zweidimensionales Bild aus einem dreidimensionalen Skalarfeld zu erzeugen. Beispiele für solche Techniken sind Raycasting [Sab88], Splatting [Wes90] und der Shear-Warp-Algorithmus [LL94]. Eines der größten Anwendungsgebiete für Volumen-Rendering ist die Medizin, denn viele Datensätze sind Computertomographien (CT) oder Magnetresonanztomographien (MRT). Durch den wachsenden Fortschritt der Geräte steigt dementsprechend auch die Auflösung und Größe der Datensätze, weswegen zumeist auf Rechnernetzwerke zurückgegriffen werden muss, um die Daten verarbeiten zu können. Auch die Auflösung der Ausgabegeräte steigt rapide. Während einzelne Projektoren und Bildschirme mittlerweile bis zu 10 Millionen Pixel[1] darstellen können, können mehrere dieser Geräte zu sogenannten „Tiled Displays“ (oder „Powerwalls“) zusammengeschaltet werden [NSS+06].

Die Universität Stuttgart besitzt eine solche Powerwall in ihrem Visualisierungslabor. Die Projektionsleinwand (Abbildung 1.1) hat eine Größe von etwa $6 \times 2,25$ Metern und wird von 10 Projektoren (5 je Stereo-Kanal) mit je einer Auflösung von 4.096×2.400 Pixel beleuchtet. Dies ermöglicht die Darstellung von etwa 44 Millionen Pixel. Die Projektoren werden von einem Cluster mit 10 Knoten und jeweils 4 Videoausgängen mit Bilddaten versorgt. Da jedoch die Rechenkapazität der Display-Knoten nicht zur Berechnung der Bilddaten ausreicht, steht ein weiteres Cluster mit 64 Knoten zur Verfügung, welches über ein Hochgeschwindigkeitsnetzwerk (InfiniBand) verbunden ist.

Da solche Powerwalls meist eine speziell angepasste Software benötigen, wurden in dieser Arbeit ein hybrides Volumen-Rendering-System entwickelt. Hybrid bedeutet, dass die Systeme eine Aufteilung sowohl im Bildraum als auch im Objektraum vornehmen können. In dieser Arbeit wurden zwei Modelle zur Lastverteilung implementiert und miteinender verglichen. Das eine Modell verteilt die Arbeit statisch auf die vorhandenen Render-Knoten, das zweite Modell verwendet eine Job-Schlange, um die Arbeit gleichmäßiger zu verteilen.

Um das Volumen-Rendering zu kapseln, wurde außerdem eine wiederverwendbare COM-Komponente (im Folgenden auch Viridian genannt) entwickelt, die sowohl für OffScreen- als auch für OnScreen-Volumen-Rendering verwendet werden kann.

[1] Der DLA-SH7NL von JVC kann eine Auflösung von 4.096×2.400 Pixel darstellen.

Abbildung 1.1.: Die Projektionsleinwand der Zielarchitektur mit Volumen-Rendering eines menschlichen Schädels

Gliederung

Diese Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Hier werden die grundlegenden Techniken für diese Arbeit beschrieben.

Kapitel 3 – Stand der Forschung: In diesem Kapitel wird der Stand der Forschung vorgestellt.

Kapitel 4 – Implementierung: Hier werden die implementierten Systeme vorgestellt.

Kapitel 5 – Ergebnisse und Diskussion: In diesem Kapitel werden die Ergebnisse der Arbeit präsentiert.

Kapitel 6 – Zusammenfassung und Ausblick: Hier werden die Ergebnisse der Arbeit zusammengefasst und Anknüpfungspunkte vorgestellt.

2. Grundlagen

In diesem Kapitel werden die theoretischen und technischen Grundlagen dieser Arbeit erläutert.

2.1. Theoretische Grundlagen

In diesem Abschnitt werden die theoretischen Grundlagen dieser Arbeit beschrieben.

2.1.1. Volumen

Ein Volumen ist ein dreidimensionales Feld, welches eine Funktion ist, die jedem Punkt aus dem $\mathbb{R}^3$ einen Wert zuweist. In der Regel sind diese Daten jedoch diskretisiert, sodass die Werte auf einem Gitter angeordnet sind. Üblicherweise stammen solche Daten aus der Medizin, z.B. aus der Computertomographie, aber auch aus der Technik, Geologie und Biologie. Die dreidimensionalen Daten werden dabei oft aus zweidimensionalen Röntgenbildern mit Hilfe des Feldkamp-Algorithmus rekonstruiert [FDK84]. Ein Voxel gibt hierbei die Absorption des untersuchten Materials an.

Die Volumenvisualisierung befasst sich mit Techniken, die ein Volumen auf die Bildebene projizieren. Dabei erfreut sich in den letzten Jahren gerade die direkte Volumenvisualisierung steigender Beliebtheit. Bei der direkten Volumenvisualisierung wird das Volumen ohne Umweg über eine geometrische Zwischenrepräsentation dargestellt. Ein Beispiel für einen Algorithmus mit einer geometrischen Zwischenrepräsentation ist der *Marching Cubes* [LC87], bei dem das Volumen durch Polygone angenähert wird.

2.1.2. Volumen-Rendering-Integral

Wie in [EHK$^+$04] beschrieben wird das Volumen als eine Wolke aus Teilchen mit den Eigenschaften Emission, Absorption und Streuung aufgefasst. In den meisten Fällen wird jedoch die Streuung vernachlässigt, um die Komplexität einzuschränken. Abbildung 2.1 verbildlicht den Effekt von Emission und Absorption. Nur ein Teil c' des Lichts c, das an Punkt $t = d$ emittiert wird, erreicht das Auge an Punkt $t = 0$. Ist die Absoption κ entlang des Sichtstrahls konstant, dann ergibt sich folgender Wert für c':

$$c' = c \cdot e^{-\kappa d} \tag{2.1}$$

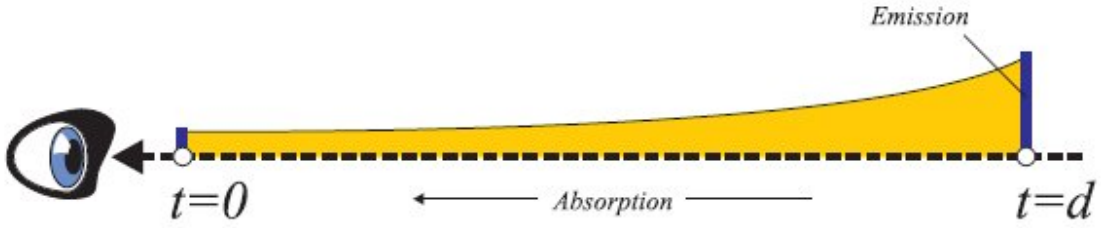


Abbildung 2.1.: Ein Teil des Lichts, das an Punkt $t = d$ emittiert wird, wird auf dem Weg zum Auge absorbiert. Quelle: [EHK+04]

Wenn die Absorption allerdings von der Position auf dem Strahl abhängt, muss über die Absorptions-Koeffizienten entlang der Strecke $(0, d)$ integriert werden:

$$c' = c \cdot e^{-\int_0^d \kappa(\hat{t})d\hat{t}} \tag{2.2}$$

Das Integral über die Absorptions-Koeffizienten wird auch *optische Tiefe* genannt:

$$\tau(d_1, d_2) = \int_{d_1}^{d_2} \kappa(\hat{t})d\hat{t} \tag{2.3}$$

Bisher wurde Licht nur von einem Punkt emittiert. Soll die gesamte Mege an Licht berechnet werden, die das Auge aus der Richtung des Sichtstrahls erreicht, dann muss über dessen gesamte Länge integriert werden:

$$c' = \int_0^\infty c(t) \cdot e^{-\tau(0,t)}dt \tag{2.4}$$

Um die *optische Tiefe* τ (Gleichung 2.3) mit einem Computer berechnen zu können, wird sie mit der Riemannsumme approximiert:

$$\tau(0, t) \approx \hat{\tau}(0, t) = \sum_{i=0}^{\lfloor t/\Delta t \rfloor} \kappa(i \cdot \Delta t)\Delta t \tag{2.5}$$

Δt gibt dabei den Abstand zwischen den Abtastpositionen an.

Die Summe kann durch eine Multiplikation von Potenzen ersetzt werden:

$$e^{-\hat{\tau}(0,t)} = \prod_{i=0}^{\lfloor t/\Delta t \rfloor} e^{-\kappa(i\cdot\Delta t)\Delta t} \tag{2.6}$$

Nun wird die *Opazität A* als

$$A_i = 1 - e^{-\kappa(i\cdot\Delta t)\Delta t} \tag{2.7}$$

eingeführt und die Gleichung 2.6 umgeschrieben zu:

$$e^{-\hat{\tau}(0,t)} = \prod_{i=0}^{\lfloor t/d \rfloor} e^{1-A_i} \tag{2.8}$$

Die Opazität A_i approximiert die Absorption auf dem i-ten Abschnitt des Strahls. Die emittierte Farbe des i-ten Strahlabschnitts kann durch

$$C_i = c(i \cdot \Delta t)\Delta t \tag{2.9}$$

approximiert werden.

Fügt man die Approximation der Opazität und der Farbe zusammen, erhält man folgende Approximation für das Volumen-Rendering-Integral:

$$\hat{C} = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1}(1 - A_i) \tag{2.10}$$

Wobei n die Anzahl der Strahlabschnitte angibt.

2.1.3. Alpha Blending

Die Summe aus Gleichung 2.10 kann iterativ mit Alpha-Blending berechnet werden. Dabei unterscheidet man zwischen den Richtungen front-to-back und back-to-front, in denen die Werte zusammengesetzt werden. Für die Richtung back-to-front wird die Formel

$$C'_i = C_i + (1 - A_i)C'_{i+1} \tag{2.11}$$

iterativ für i von $n-1$ bis 0 angewandt. $0 \leq A_i \leq 1$ bezeichnet die Opazität an der Position i. $(1 - A_i)$ bezeichnet die Transparenz und gibt die Lichtdurchlässigkeit an. Ein neuer Wert C'_i wird durch die Farbe C_i und die Opazität A_i an Position i und dem Wert C'_{i+1} der vorherigen Iteration berechnet. $C'_n = 0$.
Für die Richtung front-to-back besteht eine Iteration aus zwei Berechnungen:

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \tag{2.12}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \tag{2.13}$$

Es ist zu bemerken, dass hier neben der zusammengesetzten Farbe auch noch ein Opazitätswert zwischengespeichert werden muss.

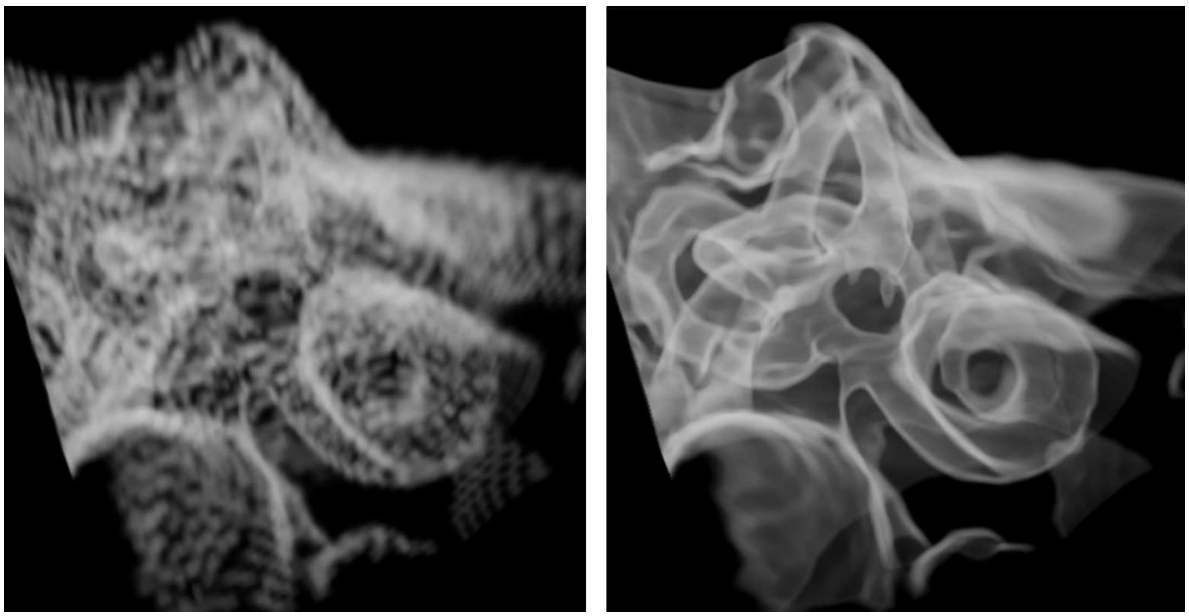


Abbildung 2.2.: Unterschied zwischen prä-interpolativer (links) und post-interpolativer Transferfunktion (rechts). Quelle: [EHK+04]

2.1.4. Transferfunktion

Wie bereits erwähnt, besteht ein Volumen in der Regel nur aus skalaren Werten. Im Anwendungsfall der Computertomographie stellen diese die Dichte bzw. Absorption dar. Blutgefäße haben dabei die geringsten Werte, danach kommt Weichgewebe, und Knochen haben die größten Werte. Um diese Stoffe farblich unterscheiden zu können, werden, mit Hilfe einer Transferfunktion, jedem Skalarwert optische Eigenschaften wie Farbe und Opazität zugewiesen. Dadurch können sogar einzelne Bestandteile (wie Weichgewebe) vollständig ausgeblendet werden.

Auf der technischen Ebene unterscheidet man hier zwischen der prä-interpolativen und der post-interpolativen Transferfunktion. Die prä-interpolative Transferfunktion wird auf die Voxel angewendet. Anschließend wird zwischen den Farbwerten und der Opazität interpoliert. Bei der post-interpolativen Transferfunktion wird zuerst zwischen den Skalarwerten interpoliert und danach die Transferfunktion angewendet. Letzteres liefert deutlich bessere Ergebnisse, wie in Abbildung 2.2 zu sehen ist. Für Details wird auf [EHK+04] verwiesen.

Um die Bildqualität weiter zu verbessern, kann in der Transferfunktion der Gradient entlang des Sichtstrahls berücksichtigt werden. Mit einer solchen zweidimensionalen Transferfunktion können zusätzlich Übergänge zwischen Stoffen (wie Muskelgewebe und Knochen) farbig hervorgehoben werden.

2.1.5. Raycasting

Das Raycasting [Sab88][EHK+04] zählt zu den direkten Volumenvisualisierungs-Techniken. Für jedes Pixel des Bildes wird ein Sichtstrahl in das Volumen gesendet. In regelmäßigen Abständen wird das Volumen abgetastet und der Wert am Abtastpunkt durch trilineare Interpolation aus den 8 nächsten Voxeln bestimmt. Auf diesen Wert wird die Transferfunktion (siehe 2.1.4) angewandt, um den Farbwert sowie Opazität zu erhalten. Anschließend werden die Abtastpunkte durch Alpha Blending (siehe 2.1.3) miteinander kombiniert, um den endgültigen Farbwert des Pixels zu erhalten.

Es bestehen einige Möglichkeiten, den klassischen Raycasting-Algorithmus zu beschleunigen. Die beiden wichtigsten sind *Early ray termination* und *Empty space skipping* [RGW+03].
Bei *Early ray termination* ist die Idee, dass entlang des Sichtstrahls unter Umständen irgendwann soviel Opazität angesammelt wurde, dass die nachfolgenden Abtastpunkte nur sehr gering oder gar nicht mehr zum endgültigen Farbwert beitragen. Der Sichtstrahl kann also vorzeitig abgebrochen werden.
Empty space skipping macht sich zunutze, dass Volumendaten oft Bereiche ohne sichbares Material enthalten. Diese Bereiche können daher von den Sichtstrahlen übersprungen werden, ohne die Bildqualität negativ zu beeinflussen. Weiteres dazu in Kapitel 3.

2.1.6. Isoflächen

Als Isoflächen werden Flächen bezeichnet, die nebeneinander liegende Punkte mit dem gleichen Wert verbinden. In einer Computertomographie, in der der Wert den Absorptionsgrad ausdrückt, könnte eine solche Isofläche die Oberfläche eines Knochens oder eines Organs darstellen. Eine Isofläche erhält man, wenn man nur die Opazität 0 und 1 verwendet. Alternativ kann man bei Raycasting auch explizit die Stelle berechnen, an der der Isowert überschritten wird. Bei beiden Techniken ist ohne Beleuchtung allerdings nur die Silhouette des Objekts sichtbar (Abbildung 2.3 (links)). Berechnet man den Gradienten der Oberfläche, kann dieser für verschiedene Beleuchtungsmodelle verwendet werden (Abbildung 2.3 (Mitte und rechts)).

2.2. Technische Grundlagen

In diesem Kapitel werden die in dieser Arbeit verwendeten Technologien vorgestellt.

2.2.1. MPI

MPI ist ein Standard für ein Kommunikationsprotokoll in parallelen Systemen. Weit verbreitete Implementierungen sind unter anderem OpenMPI, MPICH und Microsoft MPI. Die auf der Zielarchitektur verwendete Implementierung ist Microsoft MPI.

Der MPI-Standard bietet folgende Operationen an:

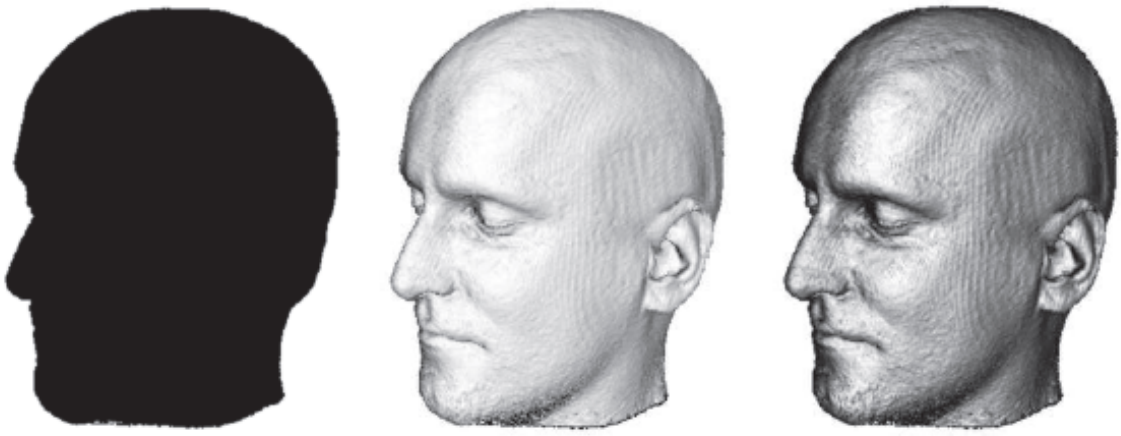


Abbildung 2.3.: Isofläche ohne Beleuchtung (links), mit diffuser Beleuchtung (Mitte) und spekularem Licht (rechts). Quelle: [EHK+04]

Punkt-zu-Punkt Kommunikation ist die grundlegendste Art der Kommunikation. Ein Prozess übermittelt eine Nachricht an einen anderen Prozess. MPI spezifiziert sowohl blockierendes als auch nichtblockierendes Senden und Empfangen. Bei blockierender Kommunikation warten beide Prozesse bis die Nachricht übermittelt wurde. Bei nichtblockierender Kommunikation wird die Nachrichtenübermittlung lediglich angestoßen. Die Prozesse können so weitere Berechnungen ausführen und mit einer weiteren Operation überprüfen, ob die Übermittlung abgeschlossen wurde. MPI erlaubt ausserdem die Markierung von Nachrichten. Damit kann der Empfänger angeben, auf welche Art von Nachricht er wartet.

Gruppenverwaltung: Prozesse sind in MPI in Gruppen zusammengefasst, wobei jedem Prozess in einer Gruppe ein eindeutiger Rang zugeordnet wird. Jede Gruppe besitzt einen Kommunikator der angegeben werden muss wenn innerhalb dieser Gruppe kommuniziert wird. Der Kommunikator der alle Prozesse einschließt heißt *MPI_COMM_WORLD*. MPI bietet Operationen an um die Schnittmenge, Differenz und Vereinigung von Gruppen zu bilden.

Globale Kommunikation bezeichnet Kommunikation bei der alle Prozesse einer Gruppe teilnehmen. MPI spezifiziert unter anderem die folgenden Operationen:

Broadcast: Ein Prozess schickt ein Datum an alle anderen Prozesse.

Scatter: Ein Prozess verteilt eine Liste von Daten auf den anderen Prozessen, wobei jeder Prozess nur einen Teil der Daten erhält.

Gather: Ein Prozess sammelt Daten von den anderen Prozessen zusammen und speichert sie in eine Liste.

Reduce: Eine spezielle Form des *Gather*, bei der die Daten zusätzlich durch eine angegebene Operation zu einem Datum reduziert werden. Beispiele für eine solche Operation sind

Addition, Multiplikation, Minimum und Maximum für Zahlenwerte, und Und- und Oder-Funktion für Wahrheitswerte.

2.2.2. Component Object Model (COM)

COM ist eine von Microsoft entwickelte Technologie zur Entwicklung von Softwarekomponenten die Programmiersprachen-unabhängig eingesetzt werden können. Es basiert auf dem Client/Server-Prinzip. Ein COM-Server kann COM-Komponenten erstellen die ein COM-Client über deren Schnittstelle (Interface) verwendet. Es gibt die folgenden Arten von COM-Servern:

In-process-Server: Wird eine COM-Komponente eines In-process-Servers erstellt, so wird der Server in den Prozess des Clients geladen. Zugriffe auf die Komponente sind daher sehr schnell, allerdings steht die Komponente nur einem Prozess zur Verfügung.

Local Server: Ein Local Server ist ein ausführbares Programm das bei Erstellen einer COM-Komponente gestartet wird. Die Komponente kann daher von mehreren COM-Clients verwendet werden, allerdings sind Zugriffe etwas langsamer.

Remote Server: Ein Remote Server kommt zum Einsatz wenn sich zwischen Client und Server in Netzwerk befindet. Da zur Kommunikation RPC (Remote Procedure Call) verwendet wird, sind Zugriffe deutlich langsamer.

Jede COM-Schnittstelle besitzt eine weltweit einzigartige Nummer, die sogenannte GUID um diese eindeutig identifizieren zu können.

2.2.3. Direct3D

Direct3D ist eine von Microsoft entwickelte API zur Darstellung von 3D-Graphiken. Eine weitere API mit nahezu identischer Funktionalität ist OpenGL. Da jedoch nur Direct3D das Ansprechen von mehreren Graphikkarten im Falle von Consumer-GPUs erlaubt, wurde in dieser Arbeit Direct3D verwendet. Die zum Zeitpunkt der Erstellung dieser Arbeit aktuellste Version ist Direct3D 11, welches unter anderem den Compute Shader einführte. Dieser erlaubt allgemeine parallele Datenverarbeitung auf der Graphikkarte und kann darum unter anderem für Raycasting eingesetzt werden. Da Direct3D als COM-Komponente entwickelt wurde, kann dieses mit jeder Sprache verwendet werden, die COM unterstützt.

2.2.4. Boost

Boost ist eine Sammlung von Bibliotheken für C++. Verwendet wurden daraus die Bibliotheken Boost.MPI, Boost.Serialization und Boost.Asio.
Boost.MPI ist ein C++-freundlicher MPI-Wrapper und erlaubt in Kombination mit Boost.Serialization das Versenden von benutzer-definierten Datentypen auf einfache Art.
Boost.Asio ist eine plattform-übergreifende Bibliothek für Netzwerkprogrammierung und wird verwendet um die Systeme aus Kapitel 4 über einen Computer steuern zu können, der nicht Teil des Hochgeschwindigkeitsnetzwerks ist.

Listing 2.1 Beispiel einer dat-Datei für das datraw-Format

```
ObjectFileName: head256.raw
TaggedFileName: ---
Resolution: 256 256 225
SliceThickness: 1 1 1
Format: UCHAR
```

2.2.5. datraw

Datraw ist ein Datei-Format für Volumendaten und bezeichnet auch eine C-Bibliothek, mit der die Volumendaten geladen und gespeichert werden können. Ein Volumendatensatz in diesem Format besteht aus einer dat-Datei, die Metadaten über das Volumen enthält, und einer oder mehreren raw-Dateien, die die rohen Volumendaten enthalten. Die Metadaten, die in der dat-Datei angegeben sind, sind unter anderem die Größe des Volumens, die Anzahl der Komponenten pro Voxel, der Datentyp dieser Komponenten und der Abstand der Voxel.
Listing 2.1 zeigt beispielhaft den Inhalt einer dat-Datei. Das Volumen hat eine Auflösung von $256 \times 256 \times 225$ und verwendet ein Byte pro Voxel. Der Abstand der Voxel ist 1.

3. Stand der Forschung

Dieses Kapitel stellt Forschungsergebnisse in den Bereichen hardware-basiertes Volumen-Rendering, k-d-Bäume, *parallel composition*, paralleles Volumen-Rendering und *scheduling* vor.

3.1. Hardware-basiertes Volumen-Rendering

Die wichtigsten Operationen für das Volumen-Rendering sind die Interpolation und Zusammensetzung von Werten – zwei Aufgaben, die die Grafikkarte hervorragend ausführen kann. Engel et al. [EHK+04] erläutern dazu mehrere Ansätze des *Texture Mapping*. Die Idee ist, das Volumen in Scheiben aufzuteilen, welche dann auf die Bildebene projiziert werden. Die Scheiben, über die das Volumen dargestellt wird, werden Proxy-Geometrie genannt. Es wird zwischen mehreren Varianten dieser Proxy-Geometrie unterscheiden. Die größten Unterschiede finden sich in der Ausrichtung der Scheiben. Bei der ersten Variante werden die Scheiben am Volumen ausgerichtet (siehe Abbildung 3.1). Bei der zweiten Variante werden die Scheiben anhand der Blickrichtung ausgerichtet (siehe Abbildung 3.2). Die zweite Variante liefert ein besseres Bild, war aber bei den ersten GPUs noch nicht möglich.

Die technische Entwicklung erlaubt seit einigen Jahren auch die effiziente Implementierung von Raycasting auf der Grafik-Hardware. Möglich wird das durch dynamische Schleifen und Bedingungen. Stegmaier et al. [SSKE05] stellen dazu ein System vor, das unter anderem Transluzenz, transparente Isoflächen, Brechung und Reflexion unterstützt. Das System ist so flexibel gehalten, dass andere Shader einfach hinzugefügt werden können.

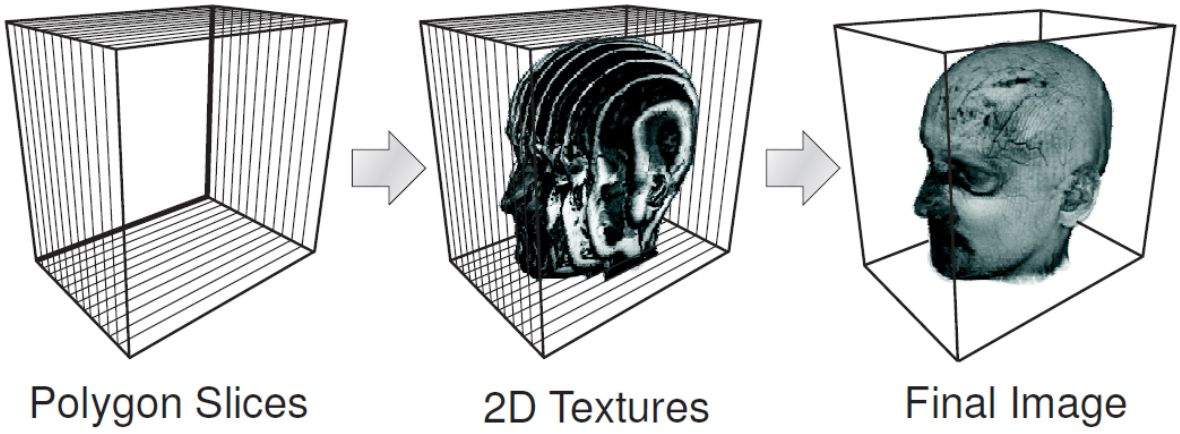


Abbildung 3.1.: Proxy-Geometrie am ausgerichtet. Quelle: [EHK+04]

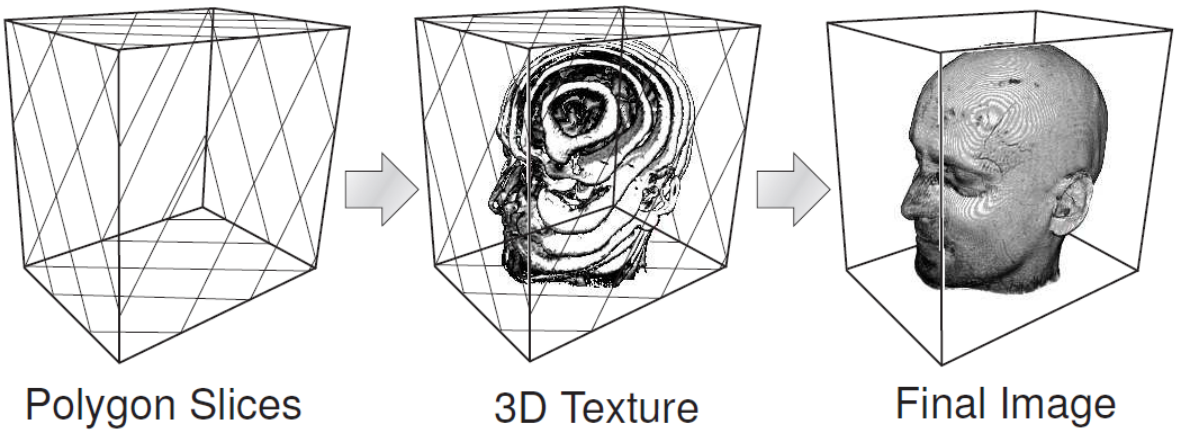


Abbildung 3.2.: Proxy-Geometrie an der Blickrichtung ausgerichtet. Quelle: [EHK+04]

3.2. k-d-Bäume

Ein k-d-Baum ist eine Datenstruktur zur Speicherungen von Punkten aus dem $\mathbb{R}^k$. Jeder innere Knoten des Baums teilt den Datensatz in zwei Bereiche. Die Blattknoten bilden die Datenpunkte. K-d-Bäume werden häufig zur Beschleunigung des Raycasting-Algorithmus verwendet. Durch die Baumstruktur können leicht Bereiche des Volumens übersprungen werden, die nur wenig oder gar nicht zum Bild beitragen. Mermitt et al. [MBFS06] präsentieren dazu einen k-d-Baum, der in den inneren Knoten den minimalen und maximalen Wert des zugehörigen Bereichs speichert. Sie präsentieren außerdem einen iterativen Algorithmus, mit dem sich der k-d-Baum parallel erneuern lässt. Damit kann der k-d-Baum auch für Volumendatensätze eingesetzt werden, die sich über die Zeit verändern.

K-d-Bäume werden auch von Marchesin et al. [MMD06] und Müller et al. [MSE06] verwendet, um ein Volumen so aufzuteilen, dass der Rechenaufwand gleichmäßig verteilt ist. Außerdem gibt der k-d-Baum direkt die Reihenfolge vor, in der die Bilder der Bricks zusammengesetzt werden müssen.

3.3. Parallel Composition

Immer wenn die Volumenvisualisierung parallelisiert wird und dabei Objektraumunterteilung zum Einsatz kommt, müssen die berechneten Bilder der Teilvolumen durch Image Compositing zusammengeführt werden, um das Bild des gesamten Volumens zu erhalten. Dabei bietet es sich an, die bereits vorhandenen Prozessoren zu verwenden. Es wurden dazu seit den Neunzigern mehrere Algorithmen vorgestellt.

Direct Send

Der einfachste ist das *Direct Send* [MPHK94][YWM08]. Dabei ist jeder Prozessor für einen Teilbereich des Bildes zuständig und erhält die entsprechenden Daten von den anderen Prozessoren. Diese

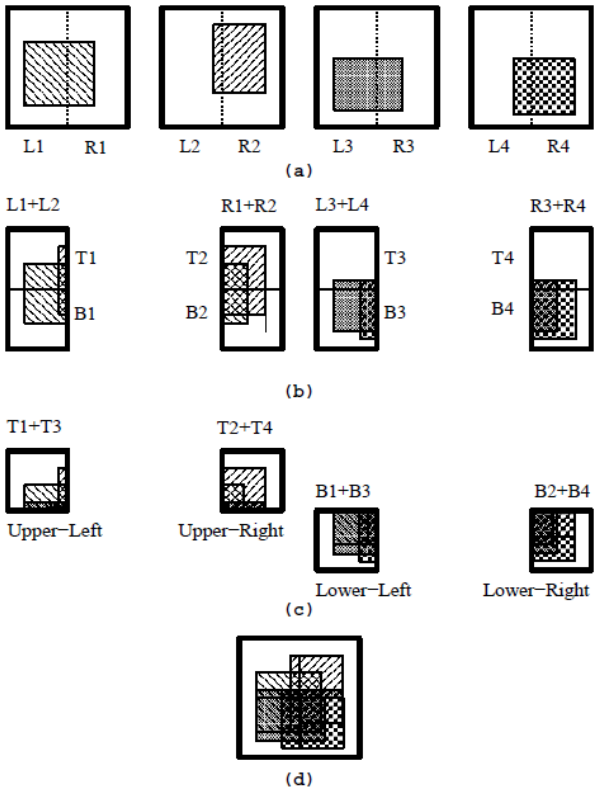


Abbildung 3.3.: *Parallel Composition* mit Binary Swap. Quelle: [MPHK94]

wiederum erhalten selbst die Daten für ihren Teilbereich. *Direct Send* funktioniert mit einer beliebigen Anzahl an Prozessoren. Da allerdings alle Prozessoren an alle anderen senden, kann es leicht zu Verklemmung kommen (d.h. alle warten, aber keiner sendet). Bei n Prozessoren werden mit dieser Methode $\mathcal{O}(n^2)$ Nachrichten versendet.

Binary Swap

Ma et al. [MPHK94] stellten 1994 den *Binary-Swap-Algorithmus* vor. Dieser zählt zu den Divide-and-Conquer-Algorithmen und strukturiert die verwendeten Prozessoren in einem vollständigen Binärbaum – weswegen die Anzahl der verwendeten Prozessoren auf eine Zweierpotenz festgelegt ist. Der Ablauf des Algorithmus ist in Abbildung 3.3 abgebildet. Zu Beginn hält jeder Prozessor das Bild des Teilvolumens, das er selbst berechnet hat. Die Hälfte davon – beispielsweise die rechte – gibt er an einen Partner-Prozessor ab, dieser wiederum gibt dem ersten Prozessor eine Hälfte – dementsprechend die linke – seines Bildes ab. Nun besitzt jeder Prozessor ein halbes Bild mit dem Inhalt von zwei Teilvolumen. Jedes Prozessor-Paar tut sich nun mit einem anderen Prozessor-Paar zusammen und wiederholt den Schritt, sodass jeder Prozessor nun ein Viertel des Bildes mit dem Inhalt von vier Teilvolumen besitzt. Bei 2^k Prozessoren wird dieser Schritt k-mal durchgeführt, sodass

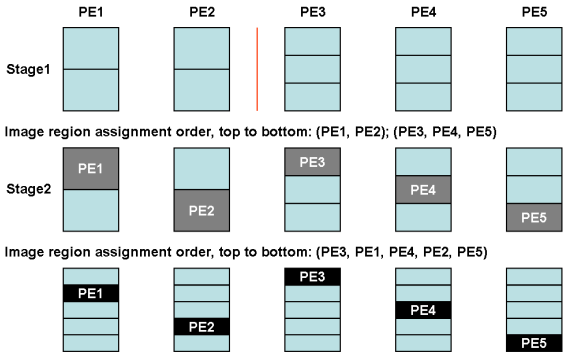


Abbildung 3.4.: Ablauf des 2-3 swap Algorithmus mit 5 Prozessoren. Die Gruppenaufteilung ist durch die senkrechte Linie markiert. Quelle: [YWM08]

jeder Prozessor am Ende ein 2^k-tel Bild des gesamten Volumens besitzt. Im Verlauf des Algorithmus werden $\mathcal{O}(n \log(n))$ Nachrichten versendet.

2-3 Swap

Mit *2-3 swap* [YWM08] stellen Yu, Wang und Ma eine Verallgemeinerung des *Binary-Swap-Algorithmus* vor. Ansatz des Algorithmus ist, dass sich jede ganze Zahl N mit $N > 1$ als Summe aus Zweien und Dreien darstellen lässt. Bei n (mit $2^{k-1} \leq n < 2^k$) gegebenen Prozessoren erstellt der Algorithmus einen Baum mit Höhe k. Jeder innere Knoten erhält entweder zwei oder drei Kinder. Diese Struktur stellt die Gruppen im späteren Verlauf des Algorithmus dar. Zu Beginn erhält jeder Prozessor innerhalb seiner Gruppe einen Index. Anschließend wird das Bild innerhalb jeder Gruppe gleichmäßig auf die Prozessoren verteilt. Das Bild wird dabei als eindimensionale Menge an Pixeln aufgefasst, sodass die Bereiche entsprechend der Ordnung der Prozessoren aufgeteilt werden können. Der Prozessor mit dem kleinsten Index in der Gruppe erhält den ersten Teil, der mit dem zweitkleinsten Index den zweiten Teil, usw. Anschließend werden die Gruppen anhand der Baumstruktur zusammengeführt. Die Ordnung der Indizes wird dabei verzahnt, wobei eine Gruppe mit mehr Prozessoren den Vorzug erhält. Dadurch ist gewährleistet, dass der Bildbereich, der einem Prozessor zugewiesen wird, eine Teilmenge des Bereichs aus dem vorherigen Schritt ist. Das Zusammenführen der Gruppen wird k-mal ausgeführt bis nur noch eine Gruppe existiert. Der Algorithmus ist dann abgeschlossen.
In Abbildung 3.4 wird ein Ablauf des Algorithmus mit 5 Prozessoren dargestellt. Gut zu erkennen ist das Verzahnen der Indizes.
Bei *2-3 swap* werden wie bei *Binary Swap* $\mathcal{O}(n \log(n))$ Nachrichten versendet, allerdings erlaubt die Verallgemeinerung die Verwendung einer beliebigen Anzahl von Prozessoren.

3.4. Paralleles Volumen-Rendering

Marchesin et al. [MMD06] und Müller et al. [MSE06] präsentieren Systeme für paralleles Volumen-Rendering mit Lastverteilung. Beide Arbeiten haben dabei dieselbe Grundidee und unterscheiden

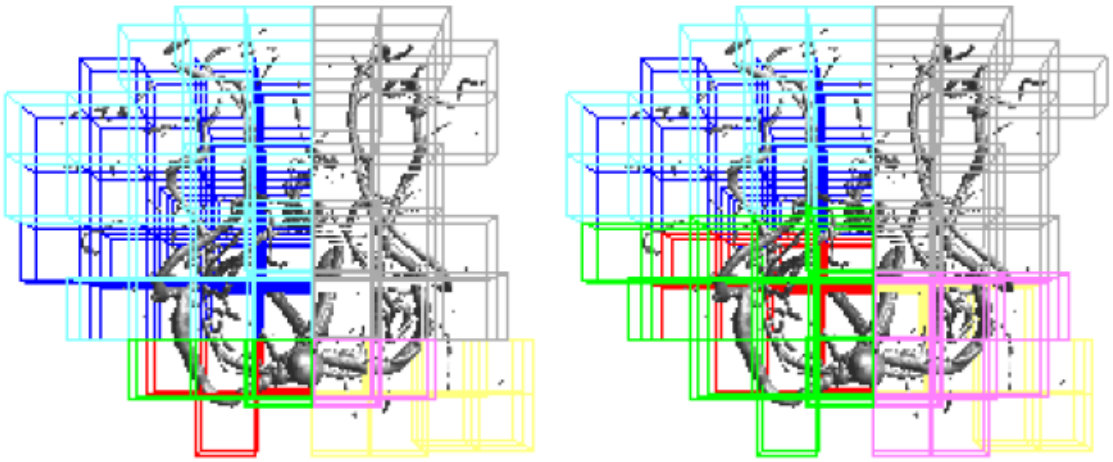


Abbildung 3.5.: Lastverteilung auf einem Aneurisma-Datensatz vor (links) und nach (rechts) der Lastverteilung. Quelle: [MSE06]

sich nur in wenigen Punkten: Zu Beginn wird das Volumen in viele Bricks aufgeteilt und auf den Render-Knoten verteilt. Zur Verwaltung der Bricks wird ein k-d-Baum verwendet, wobei die Blatt-Knoten die Bricks darstellen. Dadurch ist gewährleistet, dass die Subvolumen auf den Render-Knoten konvex sind. Als Metrik für den Rechenaufwand eines Subvolumen verwenden beide Arbeiten die Rechenzeit, die für das letzte Frame benötigt wurde.

Die Unterschiede in den Ansätzen finden sich vor allem darin, wie die Last ausbalanciert wird und wie die Volmendaten für die Bricks verwaltet werden. Während Müller et al. die Volumendaten über das Netzwerk versenden, verwenden Marchesin et al. ein mehrstufiges Caching-System. Abbildung 3.5 zeigt den Datensatz eines Aneurisma vor und nach der Lastverteilung. Die Einfärbung repräsentiert dabei die Zuteilung zu den Render-Knoten.

3.5. Scheduling

Neuberger et al. [NSS12] stellen eine C++-Bibliothek basierend auf MPI vor, die eine parallele Job-Schlange implementiert. Die Topologie berücksichtigt einen *Boss* und mehrere *Worker*. Der *Boss* initialisiert die Schlange mit einer Menge von Jobs, die von den *Workern* abgearbeitet werden. *Worker* können, nach Bedarf, einen Job in Teiljobs unterteilen und diese zur Schlange hinzufügen. Das erlaubt die einfache Parallelisierung von aufwendigen Berechnungen wie der Fakorisierung einer Zahl.
Diese Bibliohek lieferte die Idee und Grundlage für das System aus Abschnitt 4.3.

4. Implementierung

Wie bereits beschrieben, besteht die Zielarchitektur aus zehn Projektoren mit zehn dazugehörigen Display-Knoten, wobei jedes Paar aus Projektor und Display-Knoten für einen Ausschnitt des gesamten Bildes zuständig ist. Um die Rechenarbeit auf dem dahinterliegenden Cluster (mit 64 Knoten) verteilen zu können, müssen die Display-Knoten ihre Arbeit in kleinere Arbeitspakete aufteilen. Dies geschieht durch Bild- und Objektraumaufteilung.
Bei Bildraumaufteilung teilt der Display-Knoten einen Bildausschnitt in mehrere kleinere Ausschnitte auf. Der Display-Knoten muss dann nur die Teilbilder an der entsprechenden Stelle darstellen. Dies eignet sich gerade bei großen Bildern. Bei Objektraumaufteilung wird das Volumen aufgeteilt und für jedes Brick ein eigenes Bild gerendert. Diese Bilder müssen anschließend noch übereinandergelegt werden um das Bild für das gesamte Volumen zu erhalten. Dies eignet sich gerade für große Datensätze, die nicht komplett auf die GPU passen. Bei einer Kombination aus den beiden Aufteilungen kann die Objektraumaufteilung für jeden Ausschnitt der Bildraumaufteilung gesondert vorgenommen werden.

Für die Verteilung der Arbeitspakete auf dem Cluster werden in diesem Kapitel zwei Systeme vorgestellt. Das erste System weist jedem Renderer statisch einen Bildauschnitt und ein Brick zu. Das zweite System verwendet eine Schlange um die Arbeitspakete zu verteilen.
Doch zuerst wird die entwickelte Render-Komponente vorgestellt, die verwendet wird, um die Bilddaten zu berechnen.

4.1. Viridian

Viridian ist eine als In-process-Server entwickelte COM-Komponente, die Volumen-Rendering ausführt. Konfigurierbar sind unter anderem das zu rendernde (Sub-)Volumen, die Kameraparameter, der zu rendernde Ausschnitt auf der Bildebene, die Projektionsart sowie die Transferfunktion.

4.1.1. Interface

Wichtige Design-Merkmale sind die einfache Verwendung für On-Screen- und Off-Screen-Volumen-Rendering, sowie die Möglichkeit zur Bild- und Objektraumaufteilung. Die Abstraktion als COM-Interface erlaubt es, die Implementierung zu ändern, ohne dass der aufrufende Code angepasst werden muss. Außerdem kann die Komponente dadurch durch Sprachen wie C# und Visual Basic verwendet werden. Abbildung 4.1 zeigt eine Übersicht über die Schnittstellen und Datenstrukturen von Viridian, und wie diese verbunden sind. Nachfolgend werden die einzelnen Schnittstellen vorgestellt und ihre Funktionen erläutert.

Listing 4.1 Initialisierung von Viridian

```
ATL::CComPtr<IViridian> viridian;
viridian.CoCreateInstance(__uuidof(Viridian));
```

Listing 4.2 Verwendung von Viridian

```
ATL::CComPtr<IViridianEngine> engine;
viridian->CreateViridianEngine(&engine);
```

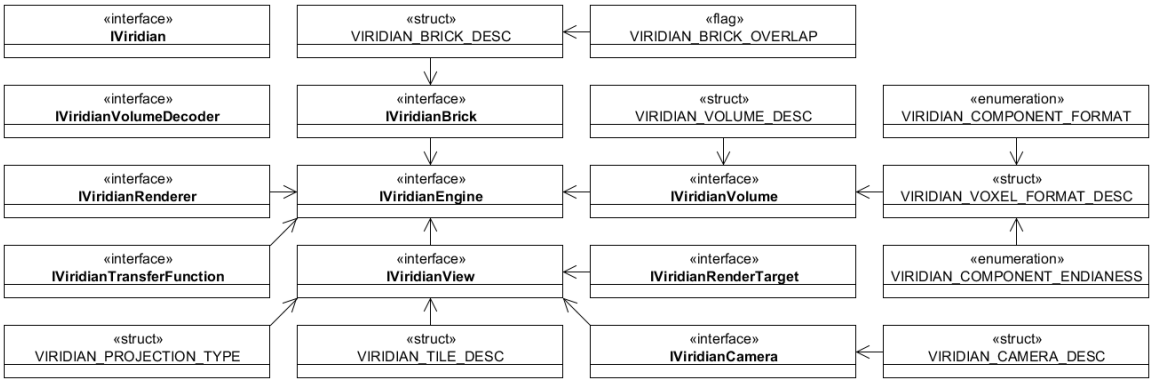


Abbildung 4.1.: Übersicht über Viridian

IViridian

Diese Klasse ist das Wurzelstück der Komponente und dient (ähnlich wie ID3D11Device in Direct3D 11) dazu, andere Klassen zu instanziieren. Code-Beispiel 4.1 zeigt wie Viridian selbst instanziiert wird. Code-Beispiel 4.2 zeigt die Verwendung von Viridian zur Instanziierung von anderen Klassen. ATL::CComPtr ist ein Objekt, das die Referenzzählung übernimmt und das verwaltete COM-Objekt automatisch freigibt sobald keine Referenzen mehr darauf verweisen.

IViridianEngine

Wie in Abbildung 4.1 erkennbar ist, ist ViridianEngine das Kernstück der Komponente, ähnlich wie ID3D11DeviceContext in Direct3D 11. ViridianEngine verwaltet die meisten Vorgänge in Viridian und ist die Schnittstelle, mit der der Benutzer am meisten interagiert.
Neben Gettern und Settern für den ViridianRenderer, die ViridianView, das ViridianBrick, die ViridianVolumes und die ViridianTransferFunction befinden sich hier die drei wichtigsten Methoden der gesamten COM-Komponente: *Render*, *FinishedRendering* und *Display*.
Render gibt das Rendering in Auftrag. Wenn die Implementierung dies unterstützt, kann der Renderprozess hier auch lediglich angestoßen werden. Mit *FinishedRendering* kann dann regelmäßig abgefragt werden, ob der Renderprozess abgeschlossen ist. So kann unter Umständen die CPU weitere

Befehle ausführen, während die GPU das Bild rendert. In der Referenzimplementierung (Unterabschnitt 4.1.2) wird dazu ein *ID3D11Query* verwendet. *Display* wird für OnScreen-Rendering verwendet und präsentiert das berechnete Bild auf dem Bildschirm.
ViridianEngine bietet mehrere Möglichkeiten zur Initialisierung an. Um die Verwendung für OnScreen-Rendering zu vereinfachen, kann dabei ein Fenster übergeben werden, in dem die berechneten Bilder dargestellt werden. Wird Viridian für OffScreen-Rendering verwendet, kann wahlweise ein Graphikadapter angegeben werden. Dadurch können auch Knoten mit mehreren Graphikkarten voll ausgenutzt werden.

IViridianRenderer

Der ViridianRenderer kapselt den eigentlichen Renderprozess und hängt daher sehr stark von der verwendeten Technologie und dem verwendeten Algorithmus ab.
Initialisiert wird der ViridianRenderer mit einer Shader-Datei. Dies erlaubt – je nach Algorithmus – das Ändern des Shadings bis hin zu anderen Render-Techniken, nur durch Angabe einer anderen Shader-Datei.

IViridianView

Ein ViridianView verbindet die folgenden vier Komponenten: Eine ViridianCamera, ein ViridianRenderTarget in das das berechnete Bild gespeichert wird, ein VIRIDIAN_TILE_DESC, das den Bildausschnitt des zu rendernden Bilds beschreibt, und ein VIRIDIAN_PROJECTION_TYPE, das die Projektionsart angibt. Unterstützt werden die folgenden Projektionsarten: perspektivisch, orthographisch und das linke und rechte Auge der Stereoprojektion.

IViridianCamera

Eine ViridianCamera beschreibt die Parameter einer Kamera. Diese sind:

apertureAngle: Der vertikale Öffnungswinkel der Kamera.

clipNear: Die Entfernung der vorderen Schnittebene zur Kamera.

clipFar: Die Entfernung der hinteren Schnittebene zur Kamera.

position: Die Position der Kamera.

lookAt: Der Punkt, auf den die Kamera ausgerichtet ist.

upVector: Der Richtungsvektor nach oben.

width: Die Breite der Bildebene.

height: Die Höhe der Bildebene.

disparity: Der Abstand der Augen bei Stereoprojektion.

IViridianRenderTarget

Ein ViridianRenderTarget ist ein Speicherbereich in den der IViridianRenderer das berechnete Bild speichert. Mit der Methode *GetData* werden die Bilddaten in ein Byte-Array geladen. Diese Methode wird für OffScreen-Rendering verwendet, um die Bilddaten zwischen Netzwerkknoten kommunizieren zu können.

IViridianVolumeDecoder

Die Schnittstelle ViridianVolumeDecoder bietet zwei Methoden an: *CreateVolume* und *LoadVolume*. *CreateVolume* nimmt eine Volumen-Datei entgegen und erstellt ein ViridianVolume mit den extrahierten Metadaten. *CreateVolume* lädt nicht die Volumendaten selbst. Diese werden erst bei Bedarf mit dem Aufruf von *LoadVolume* in den Arbeitsspeicher geladen. Durch Angabe eines ViridianBrick wird bestimmt, welcher Teil des Volumens geladen wird.

IViridianVolume

ViridianVolume beschreibt ein Volumen. Instanziiert und initialisiert wird die Klasse durch einen ViridianVolumeDecoder. Nach der Initialisierung enthält das Volumen nur Metadaten. Die konkreten Volumendaten werden erst bei Bedarf durch den ViridianVolumeDecoder geladen. Die Metadaten, die das Volumen enthält, sind: die Ausmaße des Volumens, Anzahl und Format der Komponenten pro Voxel und der Abstand der Voxel in alle drei Richtungen.

IViridianBrick

Ein ViridianBrick spezifiziert ein Subvolumen mit den Werten Ausmaß und Position im umschließenden Volumen. Außerdem wird gespeichert, in welche Richtungen sich das Brick mit anderen Bricks überschneidet. Dies wird benötigt, damit Abtastpunkte in Überschneidungsbereichen korrekt interpoliert werden können und somit Fragmente im Bild verhindert werden.

IViridianTransferFunction

Eine ViridianTransferFunction stellt eine Transferfunktion dar, wie sie in den Grundlagen (siehe Abschnitt 2.1) vorgestellt wird. Die Schnittstelle bietet drei Möglichkteiten zur Initialisierung der Transferfunktion an. Die erste Möglichkeit erstellt eine vordefinierte Transferfunktion und kann für Testzwecke verwendet werden. Die zweite Methode lädt die Daten aus einer Bilddatei. Die dritte Methode erstellt die Transferfunktion aus einem übergebenen Array. Somit kann die Transferfunktion auch programmatisch definiert werden.

4.1.2. Referenzimplementierung

In diesem Kapitel wird die Referenzimplementierung erläutert, die im Verlauf der Arbeit für alle Messungen verwendet wurde. Es wird dabei nur auf Klassen eingegangen, die relevante Logik besitzen. Die eingesetzten Technolgien und Bibliotheken sind C++, Direct3D und datraw (siehe Abschnitt 2.2).

D3D11Engine

Die D3D11Engine verwendet, wie der Name bereits sagt, Direct3D in der Version 11. Direct3D-spezifische Logik steckt vor allem in den Methoden *Initialize*, *InitializeWithAdapter* und *InitializeWithWindow*, welche sich folgendermaßen verhalten:

Initialize
: Ruft CreateDevice ohne einen DXGIAdapter auf, weswegen der erste von IDXGIFactory::EnumAdapters zurückgegebene DXGIAdapter für die Erstellung des ID3D11Device verwendet wird.

InitializeWithAdapter
: Es wird erwartet, dass der übergebene Parameter vom Typ DXGIAdapter ist. Ist das der Fall, wird das ID3D11Device mit diesem Adapter erstellt.

InitializeWithWindow
: Diese Methode iteriert über alle Devices und die zugehörigen Ausgabekanäle und wählt das Device, das den größten Teil des übergebenen Fensters enthält. Diese Methode erstellt außerdem eine ID3D11SwapChain für das Fenster. Dadurch können die berechneten Bilder mit einem Aufruf von *Display* direkt in dem Fenster dargestellt werden.

Ein weiteres interessantes Implementierungsdetail steckt in *Render* und *FinishedRendering*. Am Ende von *Render* wird ein ID3D11Query erstellt. Direct3D fügt dabei intern ein Kommando in die Liste der Befehle ein, die von der Graphikkarte ausgeführt werden müssen. In *FinishedRendering* wird das Query abgefragt, um herauszufinden ob das Kommando schon abgearbeitet wurde. Wenn das der Fall ist, kann der Benutzer sicher sein, dass der Renderprozess abgeschlossen ist.

D3D11ComputeShadeRenderer

Der D3D11ComputeShadeRenderer visualisiert die Volumen durch Raytracing und verwendet dazu den in Direct3D 11 eingeführten Compute Shader. Initialisiert wird D3D11ComputeShadeRenderer mit dem Pfad einer hlsl-Datei, die den Compute Shader enthält. Dieser wird zur Laufzeit kompiliert was es ermöglicht, den Shader zu ändern, ohne das Programm neu kompilieren zu müssen. Die Vorlage für einen Compute Shader, wie er von der Referenzimplementierung verwendet werden kann, befindet sich im Anhang (siehe Anhang A.2). Der Shader besitzt vier Slots für Volumendaten. Da diese keinen Datentyp angegeben haben, können hier unter anderem Skalardaten, vorberechnete Gradienten und Farbwerte verwendet werden.

D3D11UAVRenderTarget

Um eine beschreibbare Textur an den ComputeShader zu binden, muss ein Unordered Access View (ungeordneter Zugriffspuffer) für die Textur erstellt werden. Ein Unordered Access View erlaubt den ungeordneten Lese-/Schreibzugriff von mehreren Threads, also das was der ComputeShader benötigt.

Wird Viridian für OnScreen-Rendering verwendet, wird der BackBuffer der SwapChain direkt an die Pipeline gebunden und der Benutzer muss sich um nichts kümmern.

Wird Viridian jedoch für OffScreen-Rendering verwendet, ist es komplizierter, denn die berechneten Bilddaten liegen auf der GPU. Um die Bilddaten an andere Knoten im Netzwerk schicken zu können, müssen diese zuerst in den RAM geladen werden. Auf einen Unordered Access View kann jedoch nicht durch die CPU zugegriffen werden. Deswegen wird eine weitere Textur mit dem Attribut *D3D11_USAGE_STAGING* erstellt. Mit dem Befehl *ID3D11DeviceContext::CopyResource* wird das Bild aus dem Unordered Access View in die Staging-Textur kopiert. Anschließend erhält man mit dem Befehl *ID3D11DeviceContext::Map* Zugriff auf die Bilddaten und kann diese in den RAM kopieren. Dies alles übernimmt Viridian und der Benutzer muss lediglich *D3D11UAVRenderTarget::GetData* aufrufen um die Bilddaten zu erhalten. Der Grund weswegen die Staging-Textur benötigt wird ist, dass man verhindern will, dass sich die CPU und die GPU synchronisieren müssen und eine der beiden Ressourcen auf die andere wartet.

DatRawVolumeDecoder

Die Referenzimplementierung verwendet datraw als Dateiformat für Volumen. Die Bibliothek und der Aufbau der Volumendateien sind in Abschnitt 2.2 beschrieben. Die Aufteilung in eine Header-Datei und eine oder mehrere Dateien mit den Rohdaten unterstützt die Idee, dass die Volumendaten erst bei Bedarf geladen werden. *CreateVolume* lädt darum nur Metadaten aus der angegeben Dat-Datei und erstellt ein Volumen mit der Raw-Datei, die die Rohdaten enthält. Die konkreten Volumendaten werden erst dann durch die Methode *LoadVolume* geladen.

Da die datraw-Bibliothek leider kein partielles Laden von Volumendaten unterstützt, muss ein Umweg in Kauf genommen werden, wenn nur ein Teil des ganzen Volumens benötigt wird. Dazu lädt der DatRawDecoder das ganze Volumen und kopiert anschließend den benötigten Teil in einen separaten Puffer. Dieser Missstand ist zu vernachlässigen, wenn ein Volumen nur einmalig geladen werden muss. Anders sieht es hingegen aus, wenn mit zeitbasierten Datensätzen gearbeitet werden wird oder regelmäßig ein anderes Subvolumen benötigt wird. In diesem Fall müsste die Bibliothek angepasst oder auf eine andere Bibliothek zum Laden von Volumendaten ausgewichen werden.

4.2. Volumen-Renderer mit statischer Lastverteilung

Aufbauend auf Viridian wurde ein System erstellt, das speziell für die Zielarchitektur (siehe Kapitel 1) konzipiert ist. Die Verteilung der Last geschieht dabei statisch anhand der Konfiguration (siehe Anhang A.1).

4.2.1. Statische Verteilung

Zur Erinnerung: Die Zielarchitektur besteht aus zehn Display-Knoten, die die Darstellung der Bilder übernehmen, und optional bis zu 64 Render-Knoten[1] für die Berechnung der Bilder.

In dem hier vorgestellten System werden die Renderer statisch auf die Display-Knoten verteilt. D.h. ein Renderer wird bei Programmstart einem Display-Knoten zugeteilt, erhält von diesem einen Bildausschnitt und eventuell ein Brick und behält diese über die gesamte Programmausführung hinweg. Die Aufteilung der Renderer auf die Display-Knoten erfolgt gleichmäßig. Wenn es nicht aufgeht, werden die Display-Knoten mit kleinerem MPI-Rang bevorzugt. Bei n Display-Knoten und m Renderer erhält jeder Display-Knoten also etwa $\frac{m}{n}$ Renderer. Die Aufteilung der Renderer auf die Bildausschnitte innerhalb eines Display-Knotens erfolgt ebenfalls gleichmäßig. Wenn es nicht aufgeht, werden die Bildausschnitte oben links bevorzugt. Bei einer Bildraumaufteilung von p Ausschnitten pro Display und etwa $\frac{m}{n}$ Renderer pro Display, werden jedem Bildausschnitt etwa $\frac{m}{n \cdot p}$ Renderer zugeteilt. Innerhalb jedes Bildausschnitts wird das Volumen dann in so viele Bricks unterteilt, wie Renderer zur Verfügung stehen.

4.2.2. Architektur

Obwohl die Display-Knoten und Renderer unterschiedlichen Code ausführen, wird dieselbe ausführbare Datei gestartet. Der Grund dafür ist die Verwendung von MPI und die einfachere Verwaltung des Codes. Der Ausführungsmodus des Programms wird durch einen Kommandozeilenparameter angegeben. Zusätzlich zu den Ausführungsmodi Renderer und Display-Knoten gibt es noch den MasterDisplay-Knoten. Dieser ist ein spezieller Display-Knoten, der die Verwaltung aller Display-Knoten übernimmt. Es kann und muss darum genau ein MasterDiplay-Knoten existieren. Abbildung 4.2 zeigt eine Übersicht über das System mit den drei Ausführungsmodi und der Klasse Window, die ein Fenster auf einem Display-Knoten darstellt. Im Folgenden werden die einzelnen Komponenten des Systems näher beschrieben.

DisplayNode

Display-Knoten haben die Aufgabe einen zugeteilten Bildausschnitt auf einem Ausgabegerät darzustellen. Dazu stehen ihnen eine bestimmte Anzahl an Renderern zur Verfügung. Abhängig von der Konfiguration (siehe Abschnitt A.1) wird der Bildausschnitt des Displays in mehrere Fenster unterteilt und die Renderer auf diese verteilt. Abhängig von der Anzahl der Renderer wird ein entsprechendes Fenster erstellt. Diese sind:

[1] Hinweis: Es gilt zwischen den Bezeichnungen Render-Knoten und Renderer zu unterscheiden. Ein Render-Knoten bezeichnet einen physikalischen Knoten, während ein Renderer eine Programminstanz bezeichnet. Diese Unterscheidung ist wichtig, da unter Umständen mehrere Renderer auf einem Render-Knoten ausgeführt werden können.

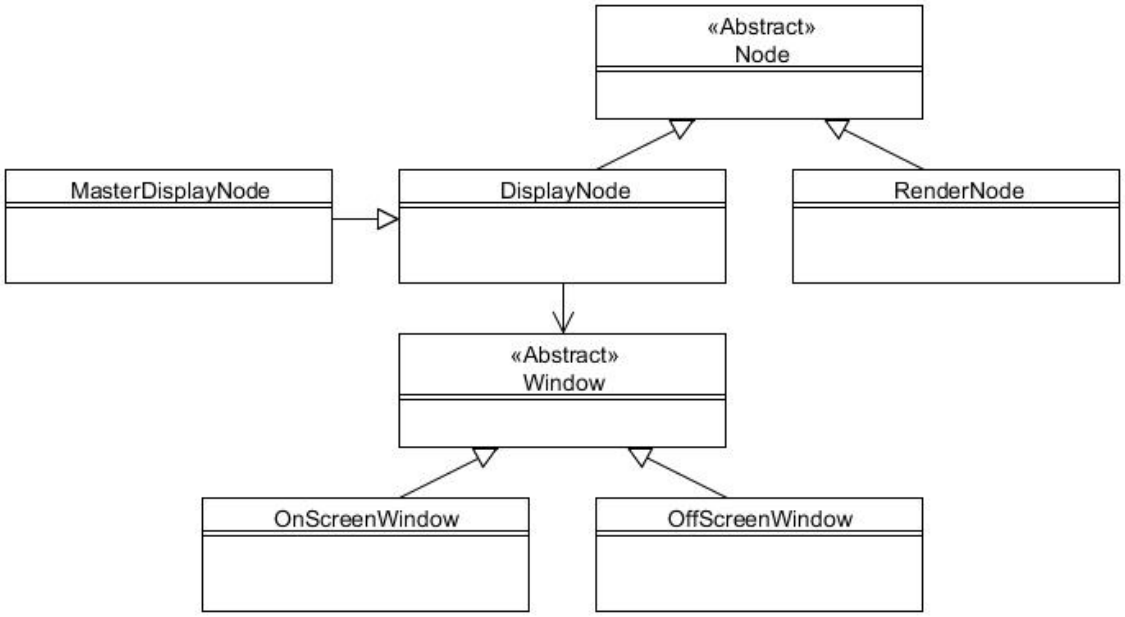


Abbildung 4.2.: Übersicht über den Volumen-Renderer mit statischer Lastverteilung

OnScreenWindow

Steht dem Fenster kein Renderer zur Verfügung, wird die Bildberechnung durch OnScreen-Rendering auf dem Display-Knoten selbst ausgeführt.

OffScreenWindow

Stehen dem Fenster ein oder mehrere Renderer zur Verfügung, wird das Volumen in mehrere Subvolumen, auch Bricks genannt, aufgeteilt, die dann auf die Renderer verteilt werden. So bekommt jeder Renderer im Cluster einen Bildausschnitt und einen Teil des Volumens (u.U. auch das ganze Volumen) zugeteilt, den er für den Rest der Programausführung rendert.
Da die Reihenfolge, in der die Bilder der Bricks in einem Fenster zusammengesetzt werden, von Bedeuteung ist, wird zur Verwaltung der Bricks ein k-d-Baum eingesetzt.

MasterDisplayNode

Der MasterDisplay-Knoten ist ein spezieller Display-Knoten, der, zusätzlich zu den Aufgaben eines regulären Display-Knotens, deren Koordinierung übernimmt. Außerdem versendet er vor jedem Frame die aktuellen Kameraparameter an alle beteiligen Knoten.
Wenn er so konfiguriert wurde (siehe Abschnitt A.1), wartet der MasterDisplay-Knoten auf eine eingehende Socket-Verbindung, über die er regelmäßig neue Kameraparameter erhält.

RenderNode

Bei Programmstart werden die vorhandenen Renderer gleichmäßig an die Display-Knoten verteilt. Von diesen bekommt jeder Renderer einen Bildausschnitt und einen Teil des Volumens (u.U. auch das ganze Volumen) zugewiesen. Zu Beginn jedes Frames erhält er die aktuellen Kameraparameter vom MasterDisplay, rendert das Bild und sendet es an den Display-Knoten, dem er zugeteilt wurde.

Da es sein kann, dass die Projektion des gegebenen Bricks überhaupt nicht oder nur wenig mit dem gegebenen Bildausschnitt überlappt (und das Bild damit leer ist), können einige Renderer ein Frame deutlich früher abgeschlossen haben als andere. Dadurch geht viel Rechenpotenzial verloren. Ein Ansatz, der versucht dieses Problem zu beheben, wird in Abschnitt 4.3 vorgestellt.

Vorschaufenster

Das Vorschaufenster ist ein eigenständiges Programm, das auf einem Rechner ausgeführt werden kann, der nicht Teil des Hochgeschwindigkeitsnetzwerks ist. Es stellt einen farbigen Würfel (stellvertretend für das Volumen) dar, welcher durch die Maus rotiert werden kann. Die Position der Kamera – relativ zum Würfel – wird regelmäßig über eine Socket-Verbindung an den MasterDisplay-Knoten gesendet. Somit kann die Ansicht des Volumens auf der Leinwand über einen Computer gesteuert werden.

4.3. Volumen-Renderer mit einer Job-Schlange

Der Volumen-Renderer, der in Abschnitt 4.2 vorgestellt wird, weist jedem Renderer statisch einen Bildbereich zu. Das hat den Nachteil, dass Renderer, die schneller mit der Berechnung fertig sind, auf langsamere Renderer warten müssen.
In diesem Kapitel wird ein System vorgestellt, das versucht dieses Problem durch eine Job-Schlange zu lösen. Die Display-Knoten geben dabei eine Liste mit Jobs an einen Scheduler, welcher diese an die Renderer verteilt.

4.3.1. Protokoll

Der Ablauf des Protokolls, das in diesem System verwendet wird, ist in Abbildung 4.3 dargestellt: Zuallererst sendet der Scheduler die benötigten Daten zur Initialisierung an die anderen Knoten. Diese sind die Pfade für die Shader- und Volumen-Datei für die Renderer und der Bildausschnitt für die Display-Knoten.

Zu Beginn jedes Frames sendet der Scheduler die aktuellen Kameraparameter per Broadcast an alle Knoten. Die Display-Knoten erstellen anschließend für jedes Fenster in Kombination mit jedem Brick einen Job und senden diese an den Scheduler. Dieser Schritt muss jedes Frame durchgeführt werden, da die Reihenfolge, in der die Bilder der Bricks übereinandergelegt werden müssen, von der Kameraposition abhängt.

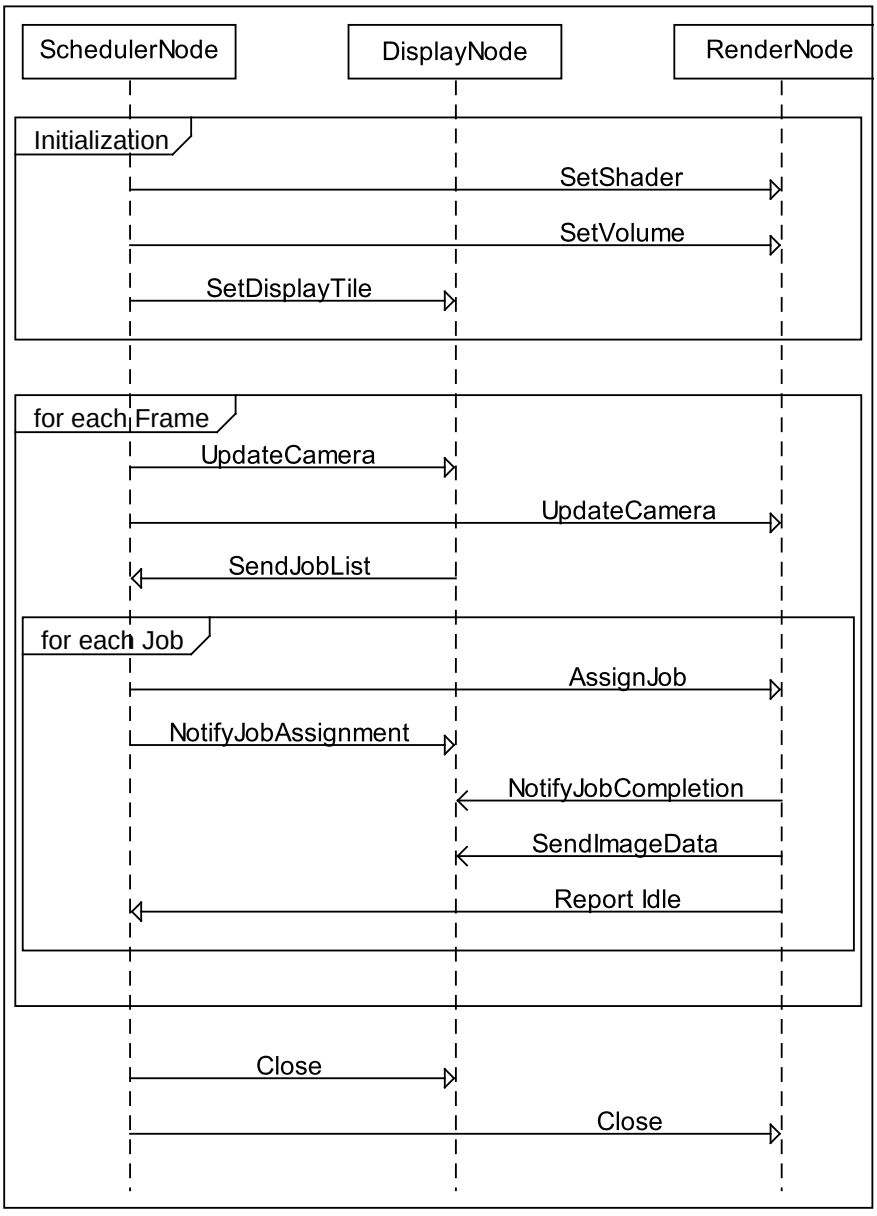


Abbildung 4.3.: Das Protokoll der Job-Schlange

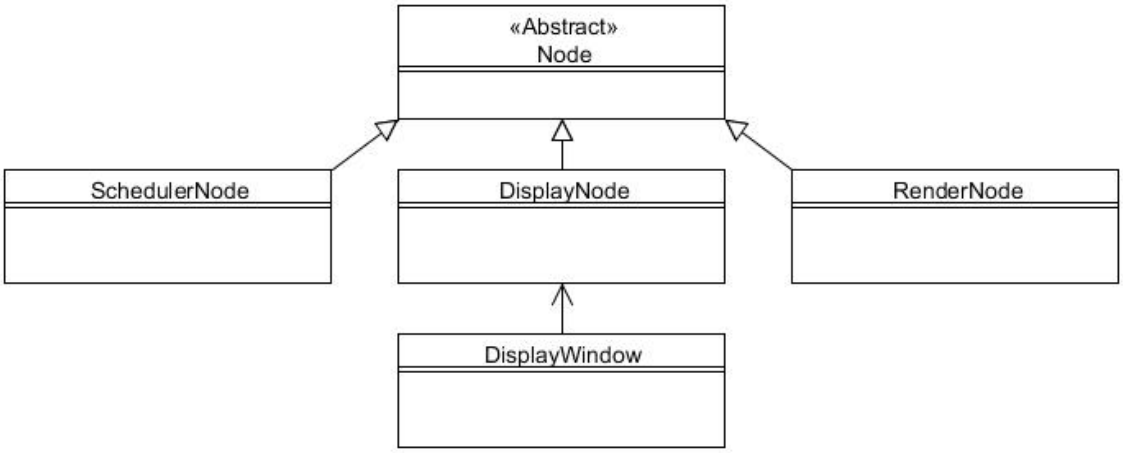


Abbildung 4.4.: Übersicht über den Volumen-Renderer mit einer Job-Schlange

Der Scheduler beginnt nun, die Jobs an unbeschäftigte Renderer zu verteilen. Wurde ein Job zugeteilt, wird anschließend der Display-Knoten, dem der Job gehört, über die Zuteilung informiert. Dies ist wichtig, da der Display-Knoten wissen muss, von welchem Renderer er das Bild zu erwarten hat. Hat ein Renderer ein Bild berechnet, informiert er den entsprechenden Display-Knoten asynchron und gibt dabei die Größe des Bildes an. Anschließend sendet er die Bilddaten – ebenfalls asynchron. Da der Renderer nun wieder Rechenkapazität verfügbar hat, meldet er dies dem Scheduler mit *ReportIdle*.

Sind alle Jobs verteilt und alle Renderer haben sich zurückgemeldet, weiß der Scheduler, dass das Frame abgeschlossen ist und ein neues Frame begonnen werden kann.

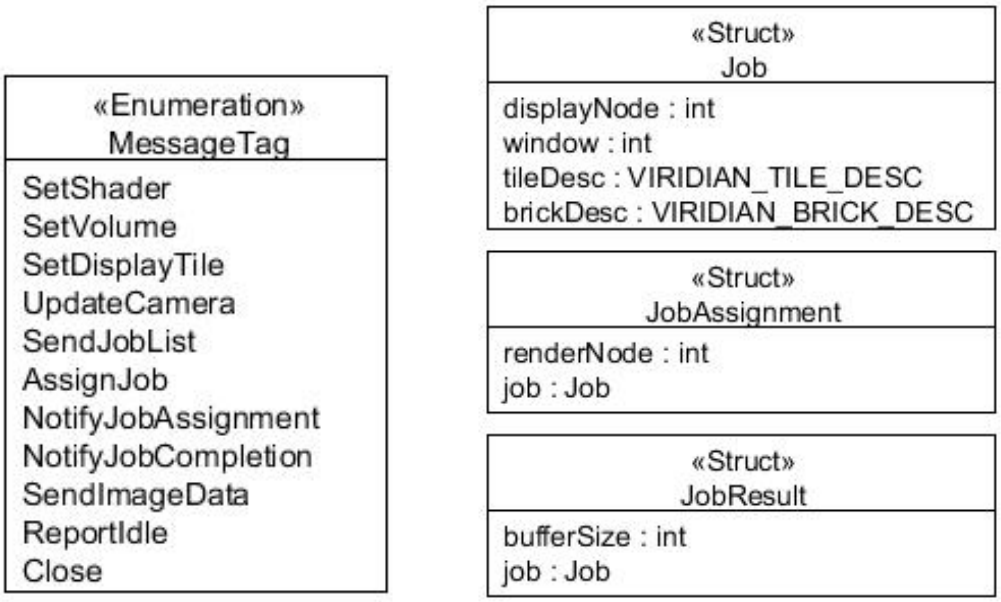
4.3.2. Architektur

Trotz des grundverschiedenen Ansatzes, die Rechenarbeit zu verteilen, hat der Aufbau viele Gemeinsamkeiten mit dem Aufbau des Systems aus Abschnitt 4.2: Auf allen Knoten wird dieselbe ausführbare Datei gestartet und der Ausführungsmodus über die Kommandozeilenparameter festgelegt. Die vorhandenen Ausführungsmodi sind Renderer und Scheduler- und Display-Knoten.

Die Implementierung der Job-Schlange verwendet MPI zur Kommunikation zwischen den Knoten. Das Protokoll verwendet die Markierungen aus Abbildung 4.5a um den Typ einer Nachricht zu spezifizieren. Abbildung 4.5b zeigt die Entitäten, die während der Kommunikation verwendet werden.

SchedulerNode

Der Scheduler ist dazu da, die Jobs zu verteilen und die anderen Knoten zu koordinieren. Es muss darum genau ein Scheduler existieren.



(a) Verwendete Markierungen **(b)** Verwendete Strukturen

Abbildung 4.5.: Verwendete Markierungen und Strukturen im Protokoll

Das Verhalten des Scheduler-Knotens ist in Listing 4.3 dargestellt: Nachdem die anderen Knoten initialisiert und die aktuellen Kameraparameter per Broadcast versendet wurden, werden die Job-Listen der Display-Knoten empfangen. Diese Listen werden in Zeile 10, unter Beibehalt der Ordnung innerhalb der Listen, verzahnt und zu einer einzigen Liste kombiniert. Diese Verzahnung erreicht, dass die Job-Listen der einzelnen Display-Knoten gleichmäßig abgearbeitet werden. Das ist von Vorteil, da die Display-Knoten selbst Zeit aufbringen müssen, um die berechneten Bilder der Jobs zusammenzufügen.

Anschließend wird kontinuierlich das folgende Schema durchgeführt: Wenn ein Job in der Job-Schlange wartet und ein Renderer verfügbar ist, dann wird der Job dem Renderer zugewiesen. Und wenn sich ein Renderer zurückmeldet, wird dieser an die Liste verfügbarer Renderer angehängt.

Dieses Schema wird solange durchgeführt, bis alle Jobs verteilt wurden und sich alle Renderer zurückgemeldet haben. Wenn das der Fall ist, ist das Frame abgeschlossen und das nächste kann begonnen werden.

DisplayNode

Ein DisplayNode ist für die Kommunikation auf dem entsprechenden Knoten zuständig. Das eigentliche Darstellen der Bilder wird von einem oder mehreren DisplayWindows ausgeführt. Diese Aufteilung hat zwei Effekte: Zum einen können mehrere Graphikkarten in dem Knoten zum Darstellen der Fenster verwendet werden, zum Anderen spiegelt sie die Aufteilung in Jobs wider, denn jedes

Listing 4.3 Verhalten des Scheduler-Knotens

```
jobSchlange = ∅
wartendeRenderer = Renderer

while (true) {
      Sende Kameraparameter

      for (display in Displays) {
            Erhalte Jobs von display
      }
      jobSchlange = Verzahne alle Jobs

      while (jobSchlange ≠ ∅ or |wartendeRenderer| < |Renderer|) {
            if (jobSchlange != ∅ && wartendeRenderer != ∅) {
                  job = jobSchlange.pop()
                  renderer = wartendeRenderer.pop()
                  Sende job an renderer
                  Informiere betreffendes Display über Jobzuweisung
            }
            else if (|wartendeRenderer| < |Renderer|) {
                  renderer = Warte bis sich ein Renderer zurückmeldet
                  wartendeRenderer.push(renderer)
            }
      }
}

Beende alle Knoten
```

Fenster erstellt pro Brick einen Job. Die Anzahl der Jobs eines Displays ist also die Anzahl der Fenster multipliziert mit der Anzahl der Bricks. Die Aufteilung in Fenster und die Anzahl der Bricks können durch die Konfiguration (siehe Abschnitt A.1) festgelegt werden.

Das Verhalten eines Display-Knotens ist in Listing 4.4 dargestellt: Nachdem die aktuellen Kameraparameter erhalten wurden, werden die Job-Listen der einzelnen Fenster gesammelt, verzahnt und an den Scheduler gesendet. Dieser Schritt muss zu Beginn jedes Frames neu ausgeführt werden, da die Reihenfolge, in der die Bilder der Bricks übereinandergelegt werden müssen, von der aktuellen Kameraposition abhängt. Anschließend wird für jedes Fenster eine leere Schlange erstellt. Dort werden später die Knoten angehängt, von denen Bilder erwartet werden. Ab dann wird solange das folgende Schema durchgeführt, bis die Anzahl der erhaltenen Bilder mit der Anzahl der gesendeten Jobs übereinstimmt: Wenn der Scheduler Informationen über eine Job-Zuweisung sendet, hänge den Renderer an die entsprechende Schlange. Überprüfe danach für jede Schlange, ob Ergebnisse bereitstehen. Wenn das der Fall ist, werden die Ergebnisse empfangen und an das entsprechende Fenster übergeben.

Wenn alle Bilder empfangen wurden, wird gewartet bis die Fenster die Bilder zusammengefügt haben. Anschließend synchronisieren sich die Display-Knoten, um das Bild auf allen Fenstern möglichst gleichzeitig zu erneuern.

Listing 4.4 Verhalten eines Display-Knotens

```
while (true) {
      Erhalte Kameraparameter

      for (fenster in Fenster) {
            Erhalte Job-Liste von fenster
      }
      jobListe = Verzahne Jobs in jobListe
      Sende jobListe an Scheduler

      erhalteneBilder = 0
      inBearbeitung[] = ∅ für alle Fenster

      while (erhalteneBilder < |jobListe|) {
            if (Erhalte Informationen über Jobzuweisung) {
                  inBearbeitung[Jobzuweisung.job.window].push(Jobzuweisung.renderNode)
            }

            for (fenster in Fenster) {
                  if (inBearbeitung[fenster].first() sendet Bild) {
                        bild = Erhalte Bild von inBearbeitung[fenster].first()
                        inBearbeitung[fenster].pop()
                        Übergebe bild an fenster
                        erhalteneBilder++
                  }
            }
      }

      for (fenster in Fenster) {
            Warte bis fenster mit dem Zusamensetzen fertig ist
      }

      Synchronisiere mit anderen Display-Knoten

      for (fenster in Fenster) {
            Erneuere Bild in fenster
      }
}
```

RenderNode

Ein Renderer hat die Aufgabe, Jobs entgegenzunehmen, abzuarbeiten und das berechnete Bild an das entprechende Display zu senden. Das Verhalten eines Renderers ist in Listing 4.5 dargestellt: Der Renderer wartet kontinuierlich auf eine Nachricht des Schedulers und reagiert entsprechend des Nachrichtentyps.

Ein großer Unterschied zu den Renderern aus Abschnitt 4.2 ist die Tatsache, dass sich das der Bildausschnitt und das Brick mit jedem Job ändern können. Wenn sich der Bildausschnitt ändert, müssen zwei Fälle unterschieden werden. Im ersten Fall haben der neue Bildausschnitt die selbe Größe wie der Bildausschnitt der vorherigen Jobs. Dann muss eine neue Textur erstellt werden. Im zweiten Fall haben die beiden Bildausschnitte die selbe Größe und die bestehende Textur kann wiederverwendet

Listing 4.5 Verhalten eines Renderers

```
while (true) {
      markierung = Warte auf eine Nachricht vom Scheduler
      switch (markierung) {

      case MessageTag::Close:
            Beende Programm

      case MessageTag::UpdateCamera:
            Erhalte Kameraparameter

      case MessageTag::AssignJob:
            Erhalte Job
            Lade Volumendaten, wenn nötig
            Starte den Renderprozess
            Warte bis das Bild des letzen Jobs gesendet wurde
            Warte bis der Renderprozess abgeschlossen wurde
            Sende Bild asynchron an den Display-Knoten
            Melde bei Scheduler zurück

      }
}
```

werden.
Wenn sich das Brick ändert, müssen die neuen Brickdaten auf jeden Fall von der Festplatte geladen werden. Bei einer Objektraumaufteilung von m Bricks geschieht das mit einer theoretischen Wahrscheinlichkeit von $\frac{1}{m}$. Die Kosten für das Neuladen werden in Kapitel 5 untersucht.

Außerdem ist zu bemerken, dass das Bild asynchron an den Display-Knoten gesendet wird. Dadurch kann der Renderer bereits den nächsten Job auf der GPU abarbeiten, während die CPU das Bild versendet.

5. Ergebnisse und Diskussion

In diesem Kapitel wird der Zugewinn durch das Cluster in Verwendung durch die Systeme aus Kapitel 4 bewertet. Dazu wurden Messungen durchgeführt, die anschließend diskutiert werden.

Für alle Messungen wurden – wenn nicht anders angegeben – direkte Volumenvisualisierung per Raycasting mit einer Transferfunktion und eine Schrittweite von einem Voxelabstand verwendet. Das verwendete Volumen (Abbildung 5.1) hat eine Auflösung von $256 \times 256 \times 225$.

5.1. Zielarchitektur

Die Spezifikation der Zielarchitektur ist in Tabelle 5.1 aufgelistet.

Display-Knoten	**Render-Knoten**
2 × Xeon X5650 @ 2,67 GHz HexaCore	2 × Xeon E5620 @ 2,4 GHz QuadCore
24 GB RAM	24 GB RAM
2 × Quadro 6000	2 × GeForce GTX 480, 1,5 GB RAM
Windows HPC Server 2008 R2 SP1	Windows HPC Server 2008 R2
2 × DDR InfiniBand (DualPort)	DDR InfiniBand

Tabelle 5.1.: Spezifikation der Zielarchitektur

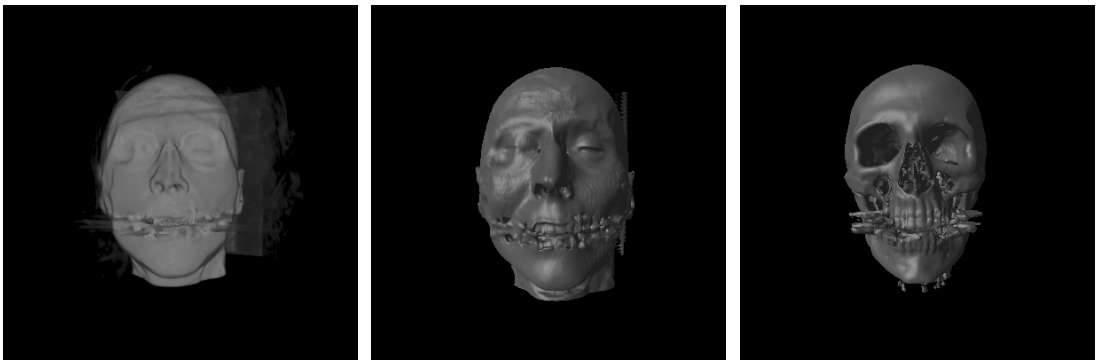


Abbildung 5.1.: Der verwendete Datensatz mit direkter Volumenvisualisierung (a) und als Isofläche mit den Werten 0,35 (b) und 0,45 (c)

Jeder der Render-Knoten besitzt zwei Graphikkarten. Deswegen können auf allen Render-Knoten zwei Instanzen von Viridian ausgeführt werden. Diese werden im Folgenden als Renderer bezeichnet.

Hinweis: In dem Zeitraum in dem die Messungen durchgeführt wurden, war das Cluster defekt und es konnte auf den Render-Knoten jeweils nur die erste Grafikkarte angesprochen werden. Außerdem waren vier Knoten komplett ausgefallen, sodass maximal 60 Renderer – von eigentlich 128 Renderern – verwendet werden konnten.

5.2. numthreads

Der Parameter *numthreads* besteht aus drei Werten (X, Y und Z) und legt fest mit wie vielen Threads der ComputeShader gleichzeitig ausgeführt wird – die Anzahl der Threads ist $X \cdot Y \cdot Z$. Der Parameter *numthreads* hat damit maßgeblichen Einfluss auf die Geschwindigkeit des Renderings. Die optimalen Werte für *numthreads* sind schwer vorherzusagen und unterscheiden sich je nach Grapfikkarte. Darum wurden in einem ersten Test nach guten Werten gesucht, die im weiteren Verlauf der Messungen verwendet wurden. Für diesen Test wurde OffScreen-Rendering mit statischer Lastverteilung und einer Bildraumaufteilung von 2×1 (also insgesamt zehn Renderer) verwendet. Als Werte für *numthreads* wurden (1, 1, 1), (8, 8, 1), (16, 16, 1), (24, 24, 1) und (32, 32, 1) gewählt. Die Ergebnisse sind in Tabelle 5.2 aufgeführt. Wie zu sehen ist, unterscheiden sich die Messergebnisse (außer für (1, 1, 1)) nur gering, weswegen für die weiteren Messungen die Werte (16, 16, 1) gewählt wurden.

numthreads	(1, 1, 1)	(8, 8, 1)	(16, 16, 1)	(24, 24, 1)	(32, 32, 1)
Anzahl der Threads	1	64	256	576	1024
FPS	0,26	5,13	5,13	5,13	5,03

Tabelle 5.2.: Durchschnittliche Bildrate für unterschiedliche Werte für *numthreads*

5.3. OnScreen-Rendering

Um den Zugewinn durch das Cluster bewerten zu können, wurden zuerst Messungen mit OnScreen-Rendering durchgeführt, d.h. alle Bilddaten werden auf den Display-Knoten berechnet.
Jeder dieser Display-Knoten besitzt zwei Graphikkarten mit jeweils zwei Ausgängen. Jedes Display ist dadurch in vier Bereiche geteilt. Wenn nun ein Vollbild-Fenster auf dem Display dargestellt wird, muss die Hälfte jedes Bildes von einer Graphikkarte auf die andere kopiert werden.
Es wurde untersucht, wie sich die Aufteilung eines Displays in mehrere Fenster auf die Rechenzeit pro Frame auswirkt. Die Ergebnisse sind in Tabelle 5.3 dargestellt. Die erste Zahl in der ersten Zeile gibt die horizontale, die zweite Zahl die vertikale Aufteilung an.

Wie der Tabelle zu entnehmen ist, verschlechtert das Umkopieren der Daten die FPS um etwa den Faktor 2. Eine weitere wichtige Beobachtung ist, dass eine Unterteilung in noch mehr Fenster nur geringe Auswirkungen auf die Rechenzeit hat.

Aufteilung	1×1	1×2	2×1	2×2	2×4	4×2
FPS	2,35	2,35	4,83	4,78	4,72	4,63

Tabelle 5.3.: Durchschnittliche Bildrate für OnScreen-Rendering unter Variation der Anzahl der Fenster

5.4. Statische Lastverteilung

In diesem Abschnitt wird die statische Lastverteilung genauer betrachtet.

5.4.1. Bildraumaufteilung

Tabelle 5.4 zeigt die durchschnittlichen FPS, wenn nur der Bildraum aufgeteilt wird. Die Aufteilung ist pro Display zu sehen, d.h. die Anzahl der Fenster mal fünf ergibt die Anzahl der verwendeten Renderer. Die Bildraten sind zusätzlich in Abbildung 5.2 als Diagramm dargestellt.
Wie zu sehen ist, steigt die FPS zuerst, fällt bei vielen Renderern aber wieder ab. Das liegt daran, dass – mit steigender Anzahl der Renderer – zwar die Kosten für das Rendering fallen, aber gleichzeitig die Kosten für die Netzwerkkommunikation und das Zusammensetzen der Bilder steigen. Die beste Bildrate wurde für 40 Renderer gemessen.

Aufteilung	2×1	2×2	4×2	6×2
Renderer	10	20	40	60
Pixel pro Renderer	4,9 Mio.	2,45 Mio.	1,23 Mio.	820 Tsd.
FPS	5,05	5,13	5,49	4,98

Tabelle 5.4.: Durchschnittliche Bildrate für statische Lastverteilung mit steigender Bildraumaufteilung

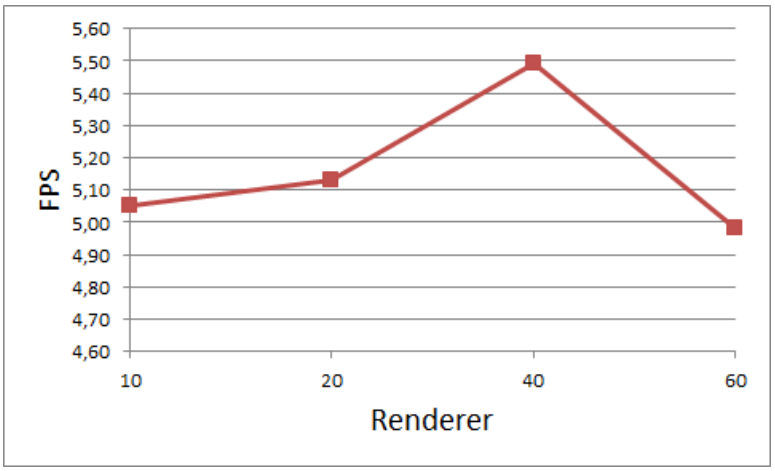


Abbildung 5.2.: Durchschnittliche Bildrate bei steigender Bildraumaufteilung

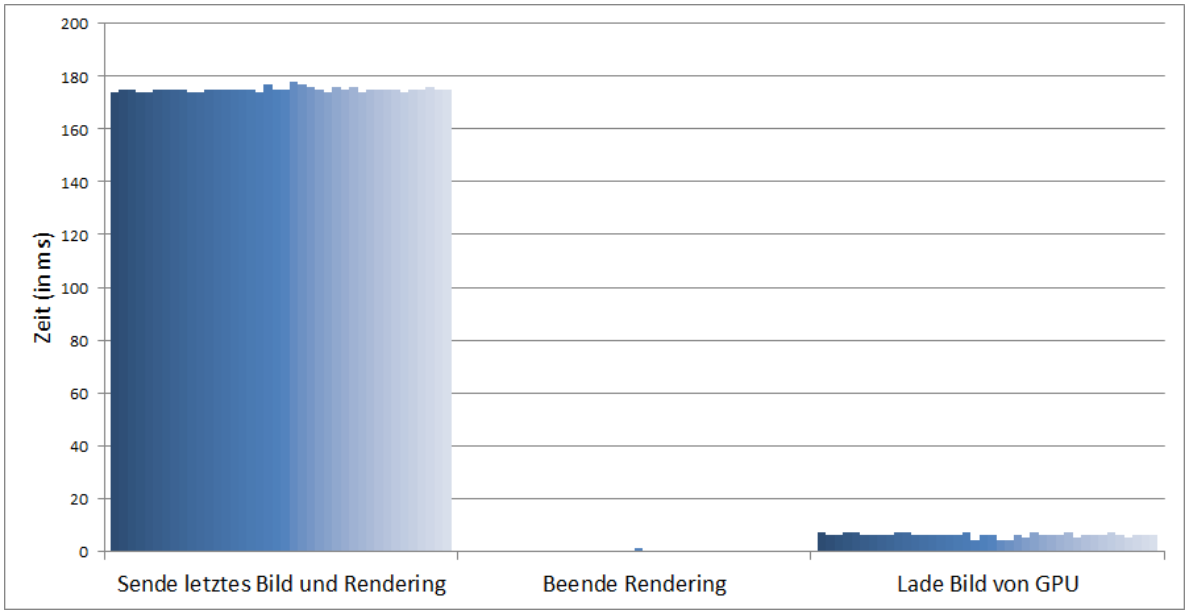


Abbildung 5.3.: Durchschnitttliche Zeitverteilung in den Renderern bei statischer Lastverteilung

Abbildung 5.3 gibt an, mit welchen Aktionen die Renderer die Zeit eines Frames verbringen. Der Ablauf ist wie folgt: Ein Renderer berechnet ein Bild und wartet währenddessen darauf, dass der Display-Knoten die Bilddaten empfangen hat. Wenn die Bilddaten versendet wurden, muss er eventuell noch warten warten bis die GPU die Bildberechnung abgeschlossen hat. Anschließend lädt er die neuen Bilddaten in den Arbeitsspeicher und sendet sie asynchron an den Display-Knoten. Direkt danach beginnt er das nächste Bild zu berechnen. Solange wartet er wieder bis das Bild gesendet wurde, etc.
Abbildung 5.4 gibt an, mit welchen Aktionen die Display-Knoten die Zeit eines Frames verbringen: Zuerst werden die Bilddaten von den Renderern empfangen, welche danach zusammengesetzt – also nebeneinander gelegt – werden. Anschließend synchronisieren sich die Display-Knoten um danach dann das zusammengesetzte Bild auf den Projektoren darzustellen. Abbildung 5.3 und Abbildung 5.4 zeigen die Zeitverteilung für die Bildraumaufteilung 4×2 (also 40 Renderer), allerdings ist die Zeitverteilung für die anderen Konfigurationen nahezu identisch. Es zeigt sich, dass die Renderer nahezu 100% der Zeit darauf verwendet wird, auf die Display-Knoten zu warten. Wieviel das eigentliche Rendering benötigt ist nicht erkenn- und messbar, doch offensichtlich sind die Display-Knoten der Flaschenhals für das System, da keine zusätzliche Zeit für das Rendering benötigt wird.
Betrachtet man die Display-Knoten sieht man, dass etwas mehr als die Hälfte der Zeit für das Empfangen der Bilder benötigt wird. Die größte Bremse für das System ist also das Netzwerk.

5.4.2. Objektraumaufteilung

Als nächsten wurde die Objektraumunterteilung untersucht (Tabelle 5.5). Es ist deutlich zu sehen, dass die Bildrate sinkt, wenn die Anzahl der Renderer steigt. Dies liegt daran, dass die Kosten für das Rendering auf den einzelnen Knoten weniger sinken also die Kosten für das Zusammensetzen

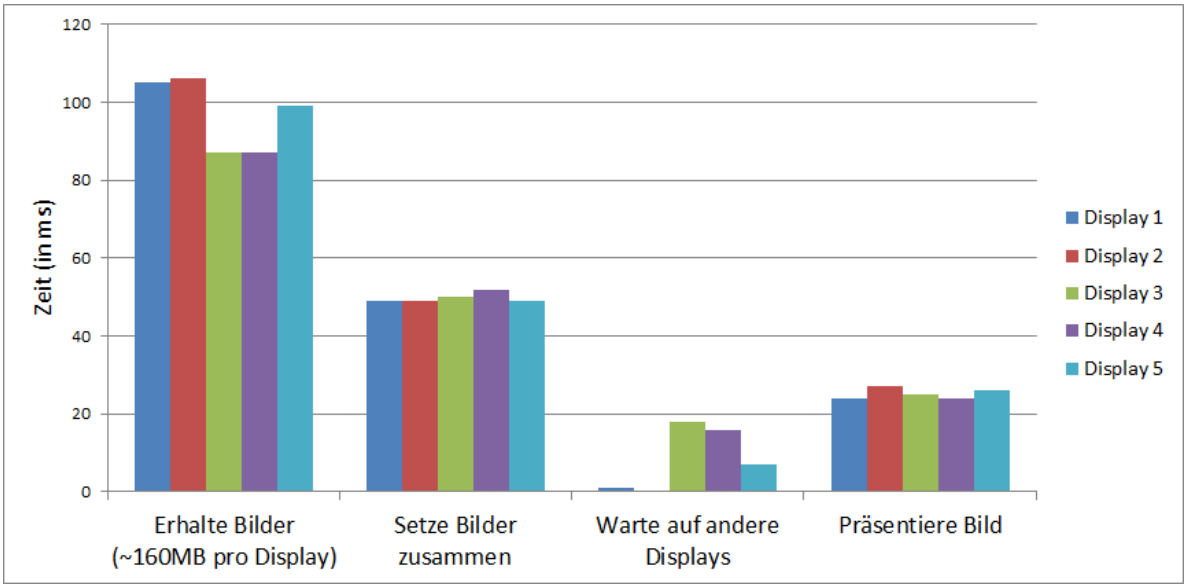


Abbildung 5.4.: Zeitverteilung auf den Display-Knoten bei statischer Lastverteilung

der Bilder steigen. Das Zusammensetzen bei Objektraumaufteilung ist außerdem teuerer als bei Bildraumaufteilung. Bei Objektraumaufteilung müssen die Bilder übereinander gelegt werden. Bei Bildraumaufteilung müssen sie nur nebeneinander dargestellt werden. Objektraumaufteilung sollte also nur verwendet werden, wenn es sich nicht vermeiden lässt, also wenn das Volumen so groß ist, dass es nicht auf die GPU passt.

Renderer	10	20	40	60
Bricks	2	4	8	12
FPS	5,09	2,66	1,37	0,88

Tabelle 5.5.: Durchschnittliche Bildrate für statische Lastverteilung mit steigender Objektraumaufteilung

5.4.3. Fazit

Wie die Messungen ergeben haben ist das Netzwerk der größte Flaschenhals für das System. Außerdem wurde erkennbar, dass die Objektraumaufteilung nur verwendet werden sollte, wenn dies wirklich nötig ist. Ein weiteres Problem des Systems ist, das die Renderarbeit mit statischer Lastverteilung ungleich verteilt ist. Im nächsten Abschnitt wird ein System untersucht, das versucht die Arbeit mit einer Job-Schlange gleichmäßig zu verteilen.

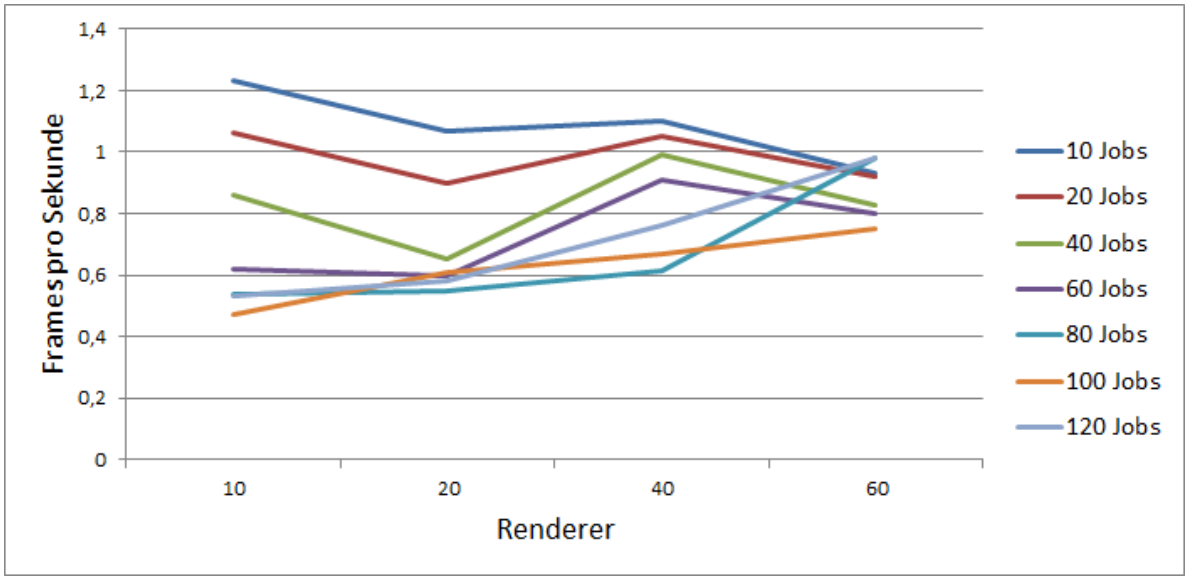


Abbildung 5.5.: Durchschnittliche Bildrate unter Variation der Renderer und Bildraumaufteilung

5.5. Lastverteilung mit Job-Schlange

Wie im vorherigen Abschnitt erwähnt, ist die Arbeit bei statischer Lastverteilung ungleich verteilt. Die Job-Schlange soll dem entgegenwirken. Wie in Abschnitt 5.4 wird dabei zunächst nur Bildraumaufteilung verwendet und anschließend zusätzlich die Objektraumaufteilung untersucht.

5.5.1. Bildraumaufteilung

In Abbildung 5.5 wurde die Bildraumaufteilung mit unterschiedlichen Anzahlen von Renderern und mehreren Bildraumaufteilungen untersucht. Die Bildraumaufteilungen und die entsprechende Anzahl von Jobs sind in Tabelle 5.6 aufgelistet. Wie man sieht ist das System mit $1,23$ FPS in der besten Konfiguration deutlich langsamer als das System mit statischer Lastverteilung ($5,49$ FPS in der besten Konfiguration).

Aufteilung	2×1	2×2	4×2	6×2	8×2	10×2	12×2
Jobanzahl	10	20	40	60	80	100	120

Tabelle 5.6.: Gesamtanzahl der Jobs bei verschiedenen Bildraumaufteilungen

5.5.2. Hybride Aufteilung

Mit zusätzlicher Objektraumaufteilung wurden nur sehr wenige Messungen (Tabelle 5.7) durchgeführt, da sehr schnell erkennbar war, dass sich die Bildrate dadurch sehr drastisch verschlechtert. Für die Messungen wurden 20 Renderer verwendet und eine Bildraumaufteilung von 2×1 und eine

Objektraumaufteilung in zwei, vier oder acht Bricks eingestellt.
Die dramatische Verschlechterung der Bildrate hat den folgenden Grund: Jedes Mal, wenn ein Renderer einen Job für ein Brick zugeteilt bekommt, das sich von dem letzen Brick unterscheidet, muss er die Daten dafür von der Festplatte laden – was viel Zeit in Anspruch nimmt (etwa 300 ms). Bei zwei Bricks geschieht dies mit einer theoretischen Wahrscheinlichkeit von 50%, bei vier Bricks mit 25%, usw. Das Laden von der Festplatte ist sehr teuer und sollte darum wenn möglich vermieden werden. Um das Neuladen von der Festplatte zu verhindern, könnte man die Bricks cachen, allerdings ist das bei sehr großen Datensätzen nicht möglich.

Aufteilung (Bricks)	2	4	8
FPS	0,74	0,45	0,27

Tabelle 5.7.: Job-Schlange mit Objektraumaufteilung

5.5.3. Fazit

Insgesamt kommt das System nicht an das System mit statischer Lastverteilung heran – nicht einmal an OnScreen-Rendering. Dies zeigt, dass die zusätzliche Kommunikation zwischen den Knoten nicht zu unterschätzen ist. Dennoch bestehen Verbesserungsmöglichkeiten für das System. Mehr dazu in Abschnitt 6.

6. Zusammenfassung und Ausblick

Zusammenfassung

In dieser Arbeit wurde eine COM-Komponente entwickelt, die in Abschnitt 4.1 vorgestellt wurde und gewöhnliches Volumen-Rendering durchführt. Wichtige Augenmerke waren die Wiederverwendbarkeit, die Erweiterbarkeit, die Verwendung für OnScreen- und OffScreen-Rendering und die hybride Aufteilung von Bild- und Objektraum.
Aufbauend auf dieser Komponente wurde ein System entwickelt, das die Bildberechnung mit statischer Lastverteilung parallelisiert. Die Ergebnisse zeigen, dass die Bildraumaufteilung der Objektraumaufteilung vorzuziehen ist. Probleme sind die ungleiche Lastverteilung auf den Render-Knoten und die Überlastung der Display-Knoten bei zu großer Aufteilung.
Um dem Problem der ungleichen Lastverteilung auf den Render-Knoten entgegenzuwirken, wurde ein zweites System entwickelt, das die Arbeit mit einer Job-Schlange verteilt. Die Messungen ergaben jedoch, dass das System nicht ausgereift genug ist und nicht an die Leistung der statischen Lastverteilung herankommt.

Ausblick

Wie die Ergebnisse in Kapitel 5 zeigen, bieten die entwickelten Systeme Raum für Verbesserungen. Einige werden hier kurz vorgestellt.

Beide Systeme aus Abschnitt 4.2 und Abschnitt 4.3 haben das Problem, dass das Zusammensetzen der Bilder auf den Display-Knoten viel Zeit in Anspruch nimmt.
Um den Display-Knoten Arbeit abzunehmen, könnte ein Weg gefunden werden, die Bilder auf den Render-Knoten zusammenzusetzen. Im Falle der statischen Lastverteilung könnte dies mit *Parallel Composition* geschehen, wie in Kapitel 3 vorgestellt.
Im Falle der Job-Schlange ist das nicht so einfach möglich. Eine Idee wäre, das System um eine neue Art von Job zu erweitern, in dem zwei Bilder zusammengefügt werden sollen. Allerdings müsste das auf eine Art geschehen, bei der sowohl die Kommunikation gering als auch die Arbeit weiterhin verteilt bleibt.

Ein weiterer Schwachpunkt der Job-Schlange ist das Neuladen des Volumens – und die damit verbundenen I/O-Kosten – wenn ein Render-Knoten ein neues Brick zugeteilt bekommt. Das Problem könnte durch Zwischenspeichern von Brickdaten im Arbeitsspeicher verringert werden. Dadurch müssten seltener Volumendaten von der Festplatte gelesen werden. Eine zweite Möglichkeit, die mit der ersten kombierbar ist, ist das intelligentere Verteilen von Jobs. Dies könnte zum Beispiel

dadurch erfolgen, dass ein Job bevorzugt einem Render-Knoten zugeteilt wird, der die entsprechenden Brickdaten bereits auf der Graphikkarte, oder zumindest im Arbeitsspeicher, hat.

Auch die Referenzimplementierung der COM-Komponente selbst bietet Raum für Optimierungen. Das größte Beschleunigungspotenzial hätte vermutlich eine Form des *Empty Space Skippings* in dem verwendeten Raycasting. Mermitt et al. [MBFS06] stellen einen k-d-Baum vor, der dafür geeignet wäre.

Verbesserungsmöglichkeiten gibt es nicht nur im Bereich der Rendering-Performance. Auch die Benutzersteuerung könnte erweitert und verbessert werden. So könnte das Vorschaufenster aus Kapitel 4 um eine Funktion erweitert werden, die die dynamische Modifikation der Transferfunktion erlaubt. Engel et al. [EHK+04] stellen dazu bereits einige Ideen vor.

A. Anhang

A.1. Konfiguration

Die Systeme aus Kapitel 4 können über zwei Stellen konfiguriert werden. Zum einen über Kommandozeilenparameter, zum anderen über eine Konfigurationsdatei.

Kommandozeilenparameter

Die folgenden Attribute können über die Kommandozeile als key/value-Paare angegeben werden:

--TYPE
: Dies ist ein Muss-Attribut und legt den Ausführungsmodus fest. Für das System mit statisher Lastverteilung aus Abschnitt 4.2 muss der Wert einer der folgenden sein: DISPLAY-MASTER, DISPLAY oder RENDERER.
Für das System mit einer Job-Schlange aus Abschnitt 4.3 sind die möglichen Werte SCHEDULER, DISPLAY und RENDERER.

--MODE
: Dieses Attribut ist optional und wird nur vom MasterDisplay-Knoten bzw. vom Scheduler ausgewertet. Der Wert muss einer der folgenden sein:

  ROTATING
  : Alle Eingabeparameter, wie Shader- und Volumen-Datei, werden mit Vorgabewerten belegt und die Kamera rotiert selbstständig um das Volumen. Dieser Modus kann für Messungen verwendet werden, da diese besser vergleichbar sind, wenn die Kamerabewegung gleichbleibend und deterministisch ist.

  CONTROLLED
  : Der MasterDisplay-Knoten bzw. der Schulder wartet auf eine eingehende Socket-Verbindung, über die er die Eingabeparameter und regelmäßig Kameraparameter erhält.

--CONFIG
: Dieses Attribut ist optional. Der Wert gibt eine Datei an, die ausgewertet wird. Die möglichen Einstellungen werden im nächsten Abschnitt beschrieben.

Listing A.1 zeigt beispielhaft, wie die Kommandozeile für den MasterDisplay-Knoten aussehen könnte.

Listing A.1 Beispiel der Kommadozeilenparameter

```
Renderer.exe --TYPE=DISPLAY-MASTER --TYPE=ROTATING --CONFIG=config.txt
```

Konfigurationsdatei

Die Konfigurationsdatei, die über die Kommandozeile angegeben wurde, wird als Menge key/value-Paaren (mit einem Paar pro Zeile) interpretiert. Das Trennzeichen ist das =-Zeichen. Die folgenden Parameter können festgelegt werden:

DISPLAY_WIDTH
: Gibt die Breite eines Displays in Pixeln an.

DISPLAY_HEIGHT
: Gibt die Höhe eines Displays in Pixeln an.

DISPLAY_OVERLAP
: Die Beamer in der Zielarchitektur überlappen in der Breite. Der Überlappungsbereich in Pixeln wird hier angegeben.

WINDOWS_PER_ROW
: Gibt an, in wieviele nebeneinander liegende Fenster jedes Display aufgeteilt werden soll.

WINDOWS_PER_COLUMN
: Gibt an, in wieviele übereinander liegende Fenster jedes Display aufgeteilt werden soll.

PROJECTION_TYPE
: Gibt die Projektionsart an. 0 entspricht perspektivischer Projektion, 1 entspricht orthographischer Projektion, 2 entspricht einer Stereo-Projektion.

VOLUME_FILE
: Gibt den Pfad des darzustellenden Volumens an.

SHADER_FILE
: Gibt die Shader-Datei an, die verwendet werden soll.

GPUS_TO_USE
: Gibt die Anzahl der GPUs an, die auf den Render-Knoten verwendet werden sollen.

ONE_DISPLAY_TEST
: Dieser Wert ist entweder 0 für falsch oder 1 für wahr und bewirkt, dass die Displays versetzt dargestellt werden. Die Einstellung wird verwendet, wenn zu Testzwecken mehrere Displays auf einem Bildschirm dargestellt werden.

BRICKS
: Dieser Wert ist nur für das System mit einer Job-Schlange relevant und gibt die Anzahl der Bricks an, in die das Volumen aufgeteilt wird.

Listing A.2 zeigt beispielhaft, wie die Konfigurationsdatei aussehen könnte.

Listing A.2 Beispiel der Konfigurationdatei

```
DISPLAY_WIDTH=2400
DISPLAY_HEIGHT=4096
DISPLAY_OVERLAP=300
WINDOWS_PER_ROW=2
WINDOWS_PER_COLUMN=2
PROJECTION_TYPE=0
VOLUME_FILE=volume.dat
SHADER_FILE=shader.hlsl
GPUS_TO_USE=2
ONE_DISPLAY_TEST=0
BRICKS=8
```

A.2. Vorlage für einen ComputeShader

```
#include "..\Viridian\gpustructures.h"

///------------------------------------------------------------
/// ReadWrite Texture - RGBA
///------------------------------------------------------------
RWTexture2D<float4> renderTarget;

///------------------------------------------------------------
/// Volume Textures
///------------------------------------------------------------
Texture3D textureSlot0 : register (t0);
Texture3D textureSlot1 : register (t1);
Texture3D textureSlot2 : register (t2);
Texture3D textureSlot3 : register (t3);
Texture2D transferFunction : register (t4);
SamplerState textureSampler : register (s0);

[numthreads(16, 16, 1)]
void main(uint3 threadID : SV_DispatchThreadID)
{
      renderTarget[threadID.xy] = float4(1.0f, 0.5f, 0.0f, 1.0f);
}
```

Literaturverzeichnis

[EHK+04] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, D. Weiskopf. Real-time Volume Graphics. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04. ACM, New York, NY, USA, 2004. (Zitiert auf den Seiten 6, 11, 12, 14, 15, 16, 19, 20 und 50)

[FDK84] L. A. Feldkamp, L. C. Davis, J. W. Kress. Practical cone-beam algorithm. *J Opt Soc Am*, S. 612–619, 1984. (Zitiert auf Seite 11)

[LC87] W. E. Lorensen, H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987. (Zitiert auf Seite 11)

[LL94] P. Lacroute, M. Levoy. Fast Volume Rendering Using a Shear-warp Factorization of the Viewing Transformation. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, S. 451–458. ACM, New York, NY, USA, 1994. (Zitiert auf Seite 9)

[MBFS06] G. Marmitt, R. Brauchle, H. Friedrich, P. Slusallek. Accelerated and Extended Building of Implicit kd-Trees for Volume Ray Tracing, 2006. (Zitiert auf den Seiten 20 und 50)

[MMD06] S. Marchesin, C. Mongenet, J.-M. Dischler. Dynamic Load Balancing for Parallel Volume Rendering. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '06, S. 43–50. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2006. (Zitiert auf den Seiten 20 und 22)

[MPHK94] K.-L. Ma, J. S. Painter, C. D. Hansen, M. F. Krogh. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, 1994. (Zitiert auf den Seiten 6, 20 und 21)

[MSE06] C. Müller, M. Strengert, T. Ertl. Optimized Volume Raycasting for Graphics-hardware-based Cluster Systems. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '06, S. 59–67. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2006. (Zitiert auf den Seiten 20, 22 und 23)

[NSS+06] T. Ni, G. S. Schmidt, O. G. Staadt, M. A. Livingston, R. Ball, R. May. A Survey of Large High-Resolution Display Technologies, Techniques, and Applications. In *Proceedings of the IEEE Conference on Virtual Reality*, VR '06, S. 223–236. IEEE Computer Society, Washington, DC, USA, 2006. (Zitiert auf Seite 9)

[NSS12] J. M. Neuberger, N. Sieben, J. W. Swift. An MPI Implementation of a Self-Submitting Parallel Job Queue. *CoRR*, abs/1204.4475, 2012. (Zitiert auf Seite 23)

[RGW+03] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, W. Strasser. Smart Hardware-accelerated Volume Rendering. In *Proceedings of the Symposium on Data Visualisation 2003*, VISSYM '03, S. 231–238. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003. (Zitiert auf Seite 15)

[Sab88] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *SIGGRAPH Comput. Graph.*, 22(4):51–58, 1988. (Zitiert auf den Seiten 9 und 15)

[SSKE05] S. Stegmaier, M. Strengert, T. Klein, T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-hardware-based Raycasting. In *Proceedings of the Fourth Eurographics / IEEE VGTC Conference on Volume Graphics*, VG'05, S. 187–195. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2005. (Zitiert auf Seite 19)

[Wes90] L. Westover. Footprint Evaluation for Volume Rendering. *SIGGRAPH Comput. Graph.*, 24(4):367–376, 1990. (Zitiert auf Seite 9)

[YWM08] H. Yu, C. Wang, K.-L. Ma. Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, S. 48:1–48:11. IEEE Press, Piscataway, NJ, USA, 2008. (Zitiert auf den Seiten 20 und 22)

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.

Stuttgart, 06.11.2014