Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Interaktives Raytracing auf CPUs für hochauflösende Remote-Displays

Benjamin Wolf

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Thomas Ertl
Betreuer/in:	M.Sc. Florian Frieß
Beginn am:	8. Juli 2020
Beendet am:	8. Januar 2021

Kurzfassung

In dieser Arbeit wird untersucht, ob die detaillierte Visualisierung von wissenschaftlichen Datensätzen auf hochauflösenden, gekachelten Anzeigen in Echtzeit realisierbar ist. Dafür wird ein System zur Anzeige von großen Datensätzen, die mittels CPU-basiertem Raytracing gerendert wurden, implementiert und vorgestellt. Damit das vorliegende System auch auf großen, hochauflösenden Displays interaktiv genutzt werden kann, wird *Foveated Rendering* eingesetzt. Hierbei werden die nicht-uniformen Eigenschaften der menschlichen Netzhaut nachgeahmt, um die Anzahl der benötigten Strahlen für das Raytracing zu reduzieren und dadurch Rechenaufwand zu sparen. Häufig wird für das CPU-basierte Raytracing ein separates Cluster benötigt, welches sich nicht notwendigerweise am selben Ort wie das Display befindet. Aus diesem Grund werden gerenderte Frames über eine Netzwerkverbindung an den Remote-Standort übertragen. Um die benötigte Bandbreite möglichst gering zu halten, werden die Bilddaten vor der Übertragung mit geeigneten Methoden der Videocodierung encodiert. Im Vergleich zu vollständigem Raytracing kann die Bildfrequenz um etwa 50 % gesteigert werden. Dadurch ist selbst auf hochauflösenden, gekachelten Anzeigen eine Darstellung nahezu in Echtzeit möglich.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API Programmierschnittstelle, engl. *application programming interface*. 13

CPU Prozessor, engl. *central processing unit*. 12

FPS Bilder pro Sekunde, engl. *frames per second*. 52

GOP Bildergruppe, engl. *group of pictures*. 30

GPU Grafikprozessor, engl. *graphics processing unit*. 12

HEVC High Efficiency Video Coding. 32

HMD Head-Mounted Display. 16

HPC Hochleistungsrechnen, engl. *high performance computing*. 11

MPEG Moving Picture Experts Group. 31

MPI Message Passing Interface. 37

OSPRay The Open, Scalable, and Portable Ray Tracing Engine. 13

RAM Arbeitsspeicher, engl. *random access memory*. 12

UDP User Datagram Protocol. 37

VISUS Visualisierungsinstitut der Universität Stuttgart. 13

VR Virtual Reality. 16

VRAM Grafikspeicher, engl. *video random access memory*. 12

1 Einleitung

Die Visualisierung und Analyse komplexer Datensätze aus verschiedenen Bereichen der Natur- und Ingenieurwissenschaften wird durch stetig wachsende Datenmengen erschwert. Zum einen werden technische Geräte wie Digitalkameras oder CT-Scanner fortwährend weiterentwickelt, um beispielsweise deren Auflösung zu erhöhen. Als Konsequenz ergibt sich daraus, dass die Menge an Daten pro Bild gleichermaßen ansteigt. Zum anderen führt das kontinuierliche Wachstum verfügbarer Rechenkapazitäten dazu, dass größere und aufwendigere Simulationen durchgeführt werden können, wodurch ebenfalls mehr Daten anfallen. Um große Datensätze passend aufbereiten zu können, werden geeignete Anzeigen benötigt. Konventionelle Desktop-Monitore erreichen aktuell bei einer 4K-Auflösung, was einer horizontalen Bildauflösung in der Größenordnung von 4000 Pixeln entspricht, ihr Limit. Besonders in Bereichen wie der Molekulardynamik, wo eine detaillierte Darstellung von mehreren tausend Atomen notwendig ist, werden solche Monitore meist nicht den Anforderungen gerecht. Um diesem Problem entgegenzuwirken, werden immer häufiger sogenannte Tiled Displays (dt. gekachelte Anzeigen), häufig auch als Powerwalls bezeichnet, zur Visualisierung großer Datensätze eingesetzt. Tiled Displays bestehen entweder aus einer Anordnung von mehreren herkömmlichen Monitoren oder werden durch die Kombination mehrerer Videoprojektoren aufgebaut. Beide Möglichkeiten bieten sowohl Vorteile als auch Nachteile. Die Nutzung von einzelnen Monitoren, die zu einer großen Anzeige zusammengeschaltet werden, hat den Vorteil, dass die Anschaffungs- und Instandhaltungskosten relativ gering sind. Allerdings entstehen bei diesem Ansatz sichtbare Kanten zwischen den einzelnen Bildschirmen, die speziell bei sehr detaillierten Darstellungen nicht wünschenswert sind. Im Gegensatz dazu steht der Einsatz von mehreren Videoprojektoren, deren Ausgabebilder nahtlos zu einem Gesamtbild zusammengefügt werden können. Nachteil eines solchen Systems sind die vergleichsweise hohen Anschaffungs- und Betriebskosten sowie das erschwerte Kalibrieren der Anzeige [MRE13].

Neben steigenden technischen Anforderungen wächst auch das Bedürfnis nach Remote-Arbeit (dt. Fernarbeit/Telearbeit). Mittlerweile bietet ein Großteil der Unternehmen die Möglichkeit des Homeoffice an, was es Arbeitnehmern gestattet, ihre Arbeit mit entsprechenden technischen Mitteln von zu Hause aus zu verrichten. Auch das Konzept des Screen-Sharings (dt. Bildschirmübertragung) kommt vermehrt zum Einsatz. Damit ist die Übertragung des Bildschirminhalts eines Computers an ein oder mehrere andere Geräte gemeint. Werden Video- oder Audiodaten übertragen und zur gleichen Zeit wiedergegeben, spricht man von *Streaming*. Dadurch wird es beispielsweise ermöglicht, Präsentationen an verschiedenen Standorten zu verfolgen, indem Stimme (Audio) und Folien (Video) des Referenten an diese Standorte übertragen werden. Man spricht in diesem Fall auch davon, dass Inhalte *gestreamt* werden. Für Computer mit geringer Rechenleistung ergibt sich mithilfe von Streaming die Möglichkeit, aufwendige Berechnungen an entfernte Standorte auszulagern. Rechnerverbunde, im Folgenden als *Cluster* bezeichnet, sind in der Lage, durch eine Vernetzung vieler Rechner sehr rechenintensive Aufgaben durchzuführen, was auch als Hochleistungsrechnen, engl. *high performance computing* (HPC) bezeichnet wird. Um auf Computern mit niedriger Leistung von HPC zu profitieren, können aufwendige Operationen von einem Cluster durchgeführt

werden, wobei die für die Berechnungen notwendigen Größen zuvor vom Quellcomputer an das Cluster gestreamt wurden. Die berechneten Ergebnisse werden schließlich vom Cluster zurück an den Quellcomputer gestreamt. Ein mögliches Einsatzgebiet ist die Erzeugung von fotorealistischen Darstellungen mittels eines Clusters.

In der Computergrafik kann die Generierung von Bildern aus gegebenen Rohdaten, nachfolgend *Rendern* genannt, durch den Einsatz von Raytracing umgesetzt werden. Dies ist ein Verfahren zur physikalisch korrekten Beleuchtungsberechnung von Objekten, mit welchem realistische 2D-Bilder eines dreidimensionalen Modells erzeugt werden können. Während frühere Computer noch nicht über ausreichend Rechenleistung verfügten, um den vergleichsweise aufwendigen Raytracing-Algorithmus in Echtzeit durchzuführen, bieten heutige Grafikkarten die Möglichkeit, Raytracing auch auf gewöhnlichen Rechnern, zum Beispiel bei Computerspielen, zu nutzen. Verantwortlich dafür ist der Grafikprozessor, engl. *graphics processing unit* (GPU), auf dem parallele Rechenleistung in hohem Maße zur Verfügung steht. Gemeint ist damit die simultane Ausführung mehrerer Operationen. Beispielsweise lässt sich in der Computergrafik ein Bild erzeugen, indem für jedes Pixel gleichzeitig die Farbe bestimmt wird. Dennoch erreicht auch der Speicher moderner GPUs bei sehr großen Datensätzen sein Limit. Herkömmlicher Arbeitsspeicher, engl. *random access memory* (RAM) ist im Vergleich zu Grafikspeicher, engl. *video random access memory* (VRAM) in deutlich höherem Maße verfügbar. Darüber hinaus sind HPC-Systeme oft nicht mit GPUs ausgestattet. Aus diesem Grund bietet es sich an, die Berechnungen für sehr große Datensätze auf dem zentralen Prozessor, engl. *central processing unit* (CPU) durchzuführen, da in Clustern viele CPUs zum Einsatz kommen [RZK+19].

Trotz stetig steigender technischer Möglichkeiten und wachsenden Rechenkapazitäten ist herkömmliches Raytracing vor allem für große hochauflösende Displays noch zu aufwendig. Aus diesem Grund werden immer häufiger die Eigenschaften der visuellen Wahrnehmung des Menschen in den Bilderzeugungsprozess miteinbezogen. Indem nur ein kleiner Teil eines Bildes, welcher für den Betrachter relevant ist, scharf dargestellt wird, kann die benötigte Rechenleistung deutlich reduziert werden, ohne einen bemerkbaren Qualitätsverlust hinnehmen zu müssen.

Für wissenschaftliche Bereiche, wo sowohl die detaillierte Darstellung von Datensätzen auf großen Anzeigen als auch die Aufteilung der Arbeit auf verschiedene Standorte immer wichtiger wird, müssen Systeme entwickelt werden, welche diesen beiden Anforderungen nachkommen.

In der vorliegenden Arbeit wird untersucht, ob die detaillierte Visualisierung von großen Datensätzen auf hochauflösenden, gekachelten Anzeigen in Echtzeit realisierbar ist.

Dafür wird ein System zum Rendern und Anzeigen von großen wissenschaftlichen Datensätzen entwickelt, welches die folgenden Anforderungen erfüllen soll. Die Verwendung des Systems soll **interaktiv** möglich sein. Das bedeutet, dass es eine Interaktion des Betrachters mit dem System gibt, die direkten Einfluss auf das angezeigte Resultat hat. Um einen hohen Detailgrad der erzeugten Bilder zu erreichen, soll **Raytracing** eingesetzt werden. Die für das Raytracing nötigen Berechnungen sollen **CPU-basiert** durchgeführt werden, um die zuvor beschriebenen Vorteile von HPC-Systemen nutzen zu können. Die erzeugten Bilder sollen **hochauflösend** sein, damit sie auch auf gekachelten Anzeigen scharf dargestellt werden können. Zudem soll das entwickelte System auf sogenannten **Remote-Displays** verwendet werden können. Damit sind Displays gemeint, die einen Großteil der Berechnungen des anzuzeigenden Inhalts an einen anderen Standort (Remote) auslagern.

Folgende Methoden werden verwendet, um den oben genannten Anforderungen nachzukommen. Eine Reihe von Datensätzen wird mittels CPU-basiertem Raytracing gerendert und auf einem Tiled Display angezeigt. Der Blickpunkt des Benutzers wird für das Rendern mitberücksichtigt, was auch als *Foveated Rendering* bezeichnet wird. Dadurch ergibt sich einerseits eine Interaktion des Betrachters mit dem System, andererseits kann Rechenaufwand eingespart werden. Die Applikation *The Open, Scalable, and Portable Ray Tracing Engine (OSPRay)* [WJA+17] wird für das Raytracing eingesetzt und in die bestehende Software *MegaMol* [GBB+19] integriert. Alle für das Rendern notwendigen Berechnungen werden auf einem Cluster ausgeführt, welches sich nicht am selben Ort wie das Display befindet. Aus diesem Grund werden gerenderte Bilder über ein Netzwerk vom Cluster an die Rechnerinfrastruktur des Displays verschickt, bevor sie angezeigt werden können. Vor dem Verschicken sorgt die Codierung der Bilddaten dafür, dass die benötigte Bandbreite gering bleibt. Für das Encodieren der Daten werden verschiedene Programmierschnittstellen, engl. *application programming interfaces* (APIs) eingesetzt. Dazu zählen die Software-Bibliotheken *x264* und *x265* sowie das Multimedia Framework *FFmpeg*. Nachdem codierte Bilddaten bei der Rechnerinfrastruktur des Displays eintreffen, werden sie von der Software *VV3* decodiert, um sie anzeigen zu können. Als Ausgabegerät für die erzeugten Bilder dient die sogenannte *Powerwall*, welche sich am Visualisierungsinstitut der Universität Stuttgart (VISUS) befindet. Die Powerwall ist eine großflächige Projektionsleinwand, welche die Anzeige von hochauflösenden Bildern ermöglicht.

Die Arbeit ist in folgender Weise gegliedert: Kapitel 2 beschreibt verwandte Arbeiten und stellt Unterschiede sowie Gemeinsamkeiten zu dieser Arbeit dar. In Kapitel 3 werden Grundlagen, die zum Verständnis des entwickelten Systems notwendig sind, erklärt. Kapitel 4 und Kapitel 5 erläutern das Konzept und die daraus resultierende Implementierung des Systems. Die Ergebnisse werden in Kapitel 6 aufgezeigt. Schließlich fasst Kapitel 7 die Resultate der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Verwandte Arbeiten

Schon seit einigen Jahrzehnten spielt Raytracing in der Computergrafik eine große Rolle, da es theoretisch eine physikalisch korrekte Beleuchtungsberechnung einer virtuellen Szene ermöglicht. Aufgrund des hohen Rechenaufwands wurde der Raytracing-Algorithmus in unterschiedlicher Weise erforscht, um Verbesserungsmöglichkeiten zu finden und ihn echtzeitfähig zu gestalten.

In Fachkreisen wird die Arbeit von Whitted [Whi80] aus dem Jahre 1980 häufig als der Grundstein für heutige Raytracing-Algorithmen angesehen. Darin wird beschrieben, wie globale Beleuchtungsinformationen in den Bilderzeugungsprozess miteinbezogen werden können. Eine spezielle Datenstruktur für jedes Pixel, ein „Strahlenbaum“, speichert jeweils einen Strahl für die erste angetroffene Oberfläche und von dort weitere Strahlen zu anderen Oberflächen und zu den Lichtquellen. Zudem wird für jeden Strahl eine bestimmte Intensität abgespeichert. Durch die Traversierung des Baumes lässt sich die Intensität und damit das Erscheinungsbild eines Pixels berechnen. Dadurch können Effekte wie Reflexion, Refraktion und Schatten exakt simuliert werden.

Moderne Raytracing-Algorithmen verwenden häufig spezielle Datenstrukturen, um den Aufwand der Schnittpunktberechnungen zu senken. Durch eine hierarchische Untergliederung der Objekte einer Szene lässt sich die Anzahl der benötigten Schnitttests und damit auch der benötigte Rechenaufwand deutlich reduzieren. Beispiele für Datenstrukturen, die eine solche Optimierung ermöglichen, sind Octrees [Gla84], BVHs (*bounding volume hierarchy*) [GPSS07] oder k-d-Bäume [FS05].

Das Buch *An Introduction to Raytracing* [Gla89] von Andrew Glassner liefert eine Einführung sowie zahlreiche Erklärungen zu Raytracing. Auch die Bücher von Shirley und Marschner [SM09] sowie Hughes et al. [HDM+13] bieten einen Überblick zum Thema Raytracing. Darüber hinaus werden in deren Arbeiten verschiedene Konzepte der Computergrafik vorgestellt. Sowohl der Raytracing-Algorithmus als auch die Grundlagen in der vorliegenden Arbeit sind an die genannten Werke angelehnt.

Echtzeit-Raytracing ist eine Herausforderung der Computergrafik. Moderne Grafikkarten bieten durch parallele Berechnungen die Möglichkeit, einfache Datensätze mittels Raytracing in Echtzeit zu rendern. Allerdings erreichen sie bei wissenschaftlichen Datensätzen, deren Größe, Komplexität und Varietät kontinuierlich anwächst, wegen des vergleichsweise geringen VRAM ihr Limit. Deshalb müssen, auch für HPC-Systeme, die über keine spezielle Rendering-Hardware verfügen, Alternativen in Betracht gezogen werden. OSPRay [WJA+17] ist ein CPU-Raytracing Framework, welches auf die wissenschaftliche Visualisierung ausgerichtet ist. Verschiedene Befehlssatzerweiterungen wie SSE4 oder AVX-512 werden von OSPRay unterstützt und beschleunigen die notwendigen Berechnungen. Zudem kann OSPRay auf HPC-Systemen eingesetzt werden und ermöglicht dadurch die Visualisierung von wissenschaftlichen Datensätzen mittels CPU-basiertem Raytracing.

Schon seit vielen Jahren wird erforscht, inwiefern das visuelle System des Menschen und die Eigenschaften der Fovea centralis in der Computergrafik miteinbezogen werden können. Werden Bilder nur in einem bestimmten Bereich, der durch den Blickpunkt des Betrachters festgelegt ist,

scharf dargestellt, so spricht man von *Foveated Rendering*. Durch Foveated Rendering wird es ermöglicht, den Aufwand für Raytracing-Algorithmen so weit zu reduzieren, dass diese in Echtzeit arbeiten können.

Einer der ersten Ansätze, die Blickrichtung direkt in den Rendering-Algorithmus miteinzubeziehen stammt von Levoy und Whitaker [LW90]. In ihrer Arbeit beschreiben sie ein System, das den Fixierungspunkt eines Benutzers mit einem Eye-Tracker aufzeichnet und anhand dessen ein Bild mit variabler Auflösung generiert. Als Rohdaten steht ein 3D-Array mit Voxel-Daten zur Verfügung, aus welchem mittels Raytracing ein Volumen gerendert wird. Dabei sind sowohl die Anzahl der Strahlen als auch die Anzahl der Samples entlang eines Strahls als Funktionen der lokalen Netzhautschärfe definiert.

In der heutigen Zeit erfreuen sich sogenannte Head-Mounted Displays (HMDs) wie Virtual Reality (VR)-Brillen immer größerer Beliebtheit. Siekawa et al. [SCMP19] präsentieren in ihrer Arbeit ein Raytracing-System, das komplexe Szenen auf VR-Geräten in Echtzeit rendert. Dafür wird eine nicht-uniforme Sample-Verteilung verwendet, welche angibt, wo Strahlen für das Raytracing generiert werden. Diese richtet sich nach den nicht-uniformen Eigenschaften des menschlichen visuellen Systems. Durch die Reduktion der verfolgten Strahlen kann die für das Rendern benötigte Berechnungszeit deutlich gesenkt werden. In einer Nutzerstudie wird gezeigt, dass die mit der Reduzierung der räumlichen Abtastung einhergehende Verschlechterung der Bildqualität im peripheren Bereich kaum wahrnehmbar ist.

Auch Weier et al. [WRK+16] präsentieren ein Raytracing-System für HMDs, welches in Echtzeit funktioniert. Ihr Ansatz besteht aus einer Kombination von Foveated Rendering basierend auf Eye-Tracking und dem Einbeziehen von vorherigen Frames (Reprojection Rendering). Um einen neuen Frame zu rendern, wird zunächst eine grobe Version des Bildes aus vorhandenen Werten aus dem G-Buffer konstruiert. Mögliche Fehler bei der Reprojektion sowie Bereiche, die in der Foveated Region liegen und daher kritisch für die Wahrnehmung sind, werden zum Resampling markiert.

Roth et al. [RWH+16] liefern in ihrer Arbeit eine detaillierte Studie zur wahrgenommenen Bildqualität bei Foveated Rendering. Dabei arbeiten sie Unterschiede zwischen statischen und dynamischen Szenen heraus.

Ein weiterer Ansatz für Foveated Rendering stammt von Guenter et al. [GFD+12]. Dabei werden mehrere Ebenen mit verschiedener Größe und Auflösung gerendert, die zu einem finalen Bild zusammengesetzt werden. Die Ebenen werden mit zunehmendem Radius, aber abnehmender Abtastrate um den Blickpunkt herum gerendert. Nach dem Rendern werden alle Ebenen mittels einer Blending-Funktion zu einem möglichst glatten Endbild zusammengefügt. Die Anzahl der Ebenen ist variabel, allerdings wurden mit drei Ebenen die besten Ergebnisse bezogen auf Qualität und Rechenaufwand erzielt. Zur Validierung der wahrgenommenen Bildqualität wurde eine Studie durchgeführt und ausgewertet.

Neben dem Rendering-Prozess können die Eigenschaften des menschlichen Auges zum Beispiel auch beim Codieren eingesetzt werden. Frieß et al. [FBB+20] präsentieren in ihrer Arbeit ein *Foveated Encoding* System. Die Qualität der Codierung wird dynamisch angepasst und variiert innerhalb des Bildes. Bereiche in der Peripherie werden stärker codiert, wodurch sich eine geringere Bildqualität, aber auch eine höhere Datenkompression ergibt. Bereiche in der Foveated Region werden hingegen schwächer codiert, um die Bildqualität möglichst hoch zu halten. Die Qualität

der Codierung wird über einen Qualitätsparameter festgelegt, der pro Makroblock anhand einer Funktion bestimmt wird. Diese Funktion ist abhängig von der Distanz eines Makroblocks zum Zentrum der Foveated Region.

Durch die eben genannten Verfahren kann die Anzeige von hochauflösenden Bildern und die Visualisierung von großen Datensätzen auf Tiled Displays realisiert werden. Darüber hinaus werden gekachelten Anzeigen in immer mehr Anwendungsszenarien verwendet, weshalb unterschiedliche Software-Systeme entwickelt wurden und zahlreiche laufende Forschungen zu Tiled Displays existieren.

Omegalib [FNM+14] ist ein Framework, welches die gleichzeitige Anzeige von mehreren Anwendungen auf gekachelten Anzeigen ermöglicht. Es ist sowohl für Endnutzer, die ein Programm zur Anzeige von Inhalten auf einem Tiled Display benötigen, als auch für Entwickler, welche die Software nach ihren Wünschen verändern und erweitern möchten, gedacht. Omegalib bietet eine einfache API, die auf unterschiedlichen Ebenen verwendet werden kann, um einen Kompromiss zwischen Komplexität und Kontrolle durch den Nutzer zu finden.

Das System *SAGE2* [MAN+14] ist einsetzbar für unterschiedliche Anzeigen, von Standard-Monitoren bis hin zu Tiled Display-Walls. Die Basis dieses Systems ist ein Browser, der zum Rendern und zur Benutzerinteraktion benutzt wird. Eine Cloud-basierte Infrastruktur dient für das Abrufen von Daten und die Echtzeitkommunikation zwischen Benutzern. Ein SAGE2-Server verwaltet verschiedene Clients (Input/Interaktion/Audio/Display) und ermöglicht dadurch die simultane Interaktion mehrerer Benutzer.

Die Cross Platform Cluster Graphics Library (CGLX) [DK11] ist ein Grafik-Framework, das für die Anzeige von OpenGL-Anwendungen auf hochauflösenden Tiled Displays oder Multiprojektor-Systemen verwendet werden kann. Eine grafische Benutzeroberfläche erleichtert das Setup von Anwendungen und bietet eine einfache Bedienung.

Ein Beispiel für eine gekachelte Anzeige liefern Müller et al. [MRE13]. In ihrer Arbeit beschreiben sie die sogenannte Powerwall und deren Rechnerinfrastruktur, welche das Rendern und Anzeigen von hochaufgelösten Stereo-Bildern ermöglicht. Das in der vorliegenden Arbeit entwickelte System nutzt die Powerwall als Anzeigegerät.

Das Softwaregerüst, welches der Powerwall zugrunde liegt, wird von Frieß et al. [FME20] beschrieben. Es dient der Verarbeitung von Pixelstreams, die decodiert und auf der Powerwall angezeigt werden können. Im Zuge dieser Arbeit wird es erweitert, sodass die Visualisierung der durch CPU-basiertes Raytracing generierten Streams ermöglicht wird.

3 Grundlagen

In diesem Kapitel werden Grundlagen beschrieben, die für das Verständnis des entwickelten Systems notwendig sind und einen Einfluss auf dessen Konzeption haben. Zunächst wird die visuelle Wahrnehmung des Menschen und der Bereich des schärfsten Sehens näher erläutert. Anschließend werden Konzepte der Computergrafik und Videocodierung genauer betrachtet. Durch Einbeziehen der Eigenschaften des menschlichen Sehens in den Bilderzeugungsprozess sowie Ausnutzen von Kompressionstechniken der Videocodierung lassen sich Aufwand und Speicherbedarf der nötigen Berechnungen deutlich reduzieren. Schließlich wird ein Überblick zur VISUS-Powerwall gegeben, welche im entwickelten System als Anzeige dient.

3.1 Visuelle Wahrnehmung

Licht kann beschrieben werden als die Ausbreitung von elektromagnetischen Wellen mit einer bestimmten Wellenlänge λ und ist damit eine Form von elektromagnetischer Strahlung. Trifft Licht auf das menschliche Auge, so kann es abhängig von der Wellenlänge wahrgenommen werden. Das elektromagnetische Spektrum setzt sich, wie in Abbildung 3.1 zu sehen ist, aus verschiedenen Arten von Strahlung zusammen. Das sichtbare Spektrum des Menschen bildet dabei mit Wellenlängen zwischen etwa 380 nm und 780 nm nur einen kleinen Teil des Gesamtspektrums. Gamma-, Röntgen- und UV-Strahlung ($\lambda < 380$ nm) kann ebenso wie Infrarotstrahlung und Radiowellen ($\lambda > 780$ nm) nicht vom menschlichen Auge wahrgenommen werden [HDM+13].

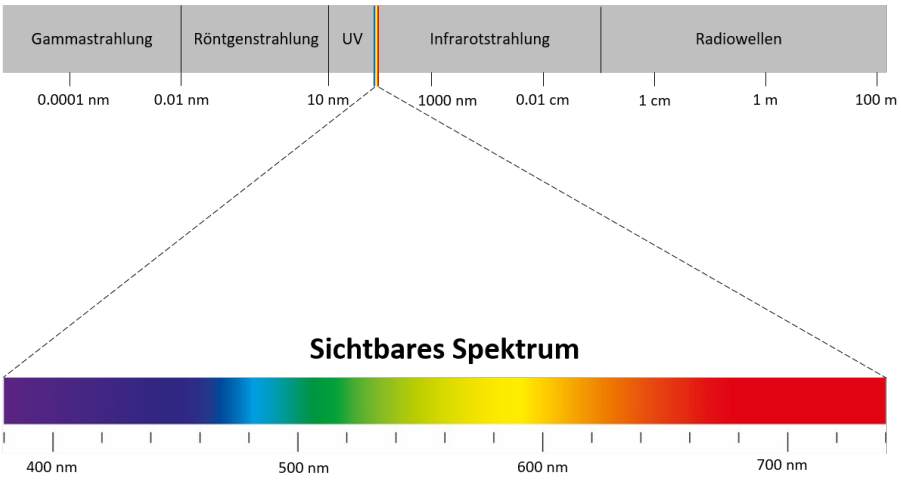


Abbildung 3.1: Das elektromagnetische Spektrum. Licht mit einer Wellenlänge von ungefähr 380 nm bis 780 nm ist für das menschliche Auge sichtbar. Wellenlängen außerhalb dieses Bereichs können vom Menschen nicht visuell wahrgenommen werden.

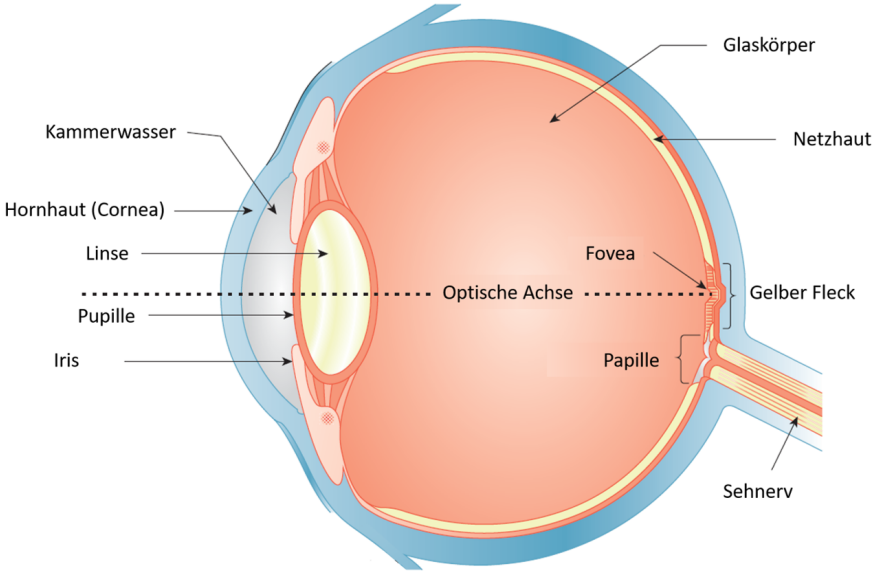


Abbildung 3.2: Aufbau des menschlichen Auges. Auf der Netzhaut (Retina) befindet sich der gelbe Fleck (Macula), in dessen Mitte die Fovea, der Bereich des schärfsten Sehens, liegt. Quelle: [HDM+13]

In Abbildung 3.2 ist der Aufbau des menschlichen Auges zu sehen, anhand dessen sich die visuelle Wahrnehmung vereinfacht darstellen lässt. Die Hornhaut (Cornea) ist zusammen mit der Linse dafür verantwortlich, einfallendes Licht auf die Netzhaut (Retina) zu fokussieren. Der Lichteinfall wird dabei durch die pigmentierte Regenbogenhaut (Iris) reguliert, indem der Pupillendurchmessers verändert wird. Die Pupille wird durch den inneren Irisrand gebildet. Auf der Netzhaut befinden sich Millionen lichtempfindlicher Zellen, die Fotorezeptoren, welche bei einfallendem Licht eine chemische Reaktion auslösen. Die entstehenden Signale werden über den Sehnerv an das Gehirn weitergeleitet, wo sie verarbeitet werden und den Eindruck eines wahrgenommenen Bildes erzeugen. Zwei Bereichen auf der Netzhaut kommt eine besondere Bedeutung zuteil. Die Fovea, welche inmitten des gelben Flecks (Macula) liegt, ist der Bereich des schärfsten Sehens. Die Austrittsstelle des Sehnervs (Papille), welche auch als blinder Fleck bezeichnet wird, ist dadurch charakterisiert, dass dort keine Fotorezeptoren vorhanden sind. Aus diesem Grund ist an dieser Stelle keine optische Wahrnehmung möglich. Dieses Phänomen von fehlenden Bildinformationen an einem bestimmten Punkt ist jedoch für den Menschen meistens nicht spürbar, da es durch die Verarbeitung des Bildes im Gehirn kompensiert wird [HDM+13].

Für die Wahrnehmung von sichtbaren Licht sind in unserem Auge zwei Arten von Rezeptoren verantwortlich. Neben den Stäbchen, die für das Dämmerungs- und Nachtsehen optimiert sind und keine Farbwahrnehmung ermöglichen, gibt es die für das Tagsehen verantwortlichen Zapfen. Bei den Zapfen wird zwischen drei verschiedenen Arten unterschieden, die jeweils auf bestimmte Wellenlängen und somit auf spezielle Farben unterschiedlich stark reagieren. Die S-Zapfen (von engl. *short*) nehmen Licht mit kurzer Wellenlänge (blau) verstärkt wahr, die M-Zapfen Licht mit mittlerer Wellenlänge (grün) und die L-Zapfen Licht mit langer Wellenlänge (rot). Durch eine Verschaltung der Signale verschiedener Zapfen wird die Wahrnehmung von farbigen Bildern ermöglicht. Anders als häufig angenommen, werden im Gehirn nicht die drei Grundfarben rot, grün

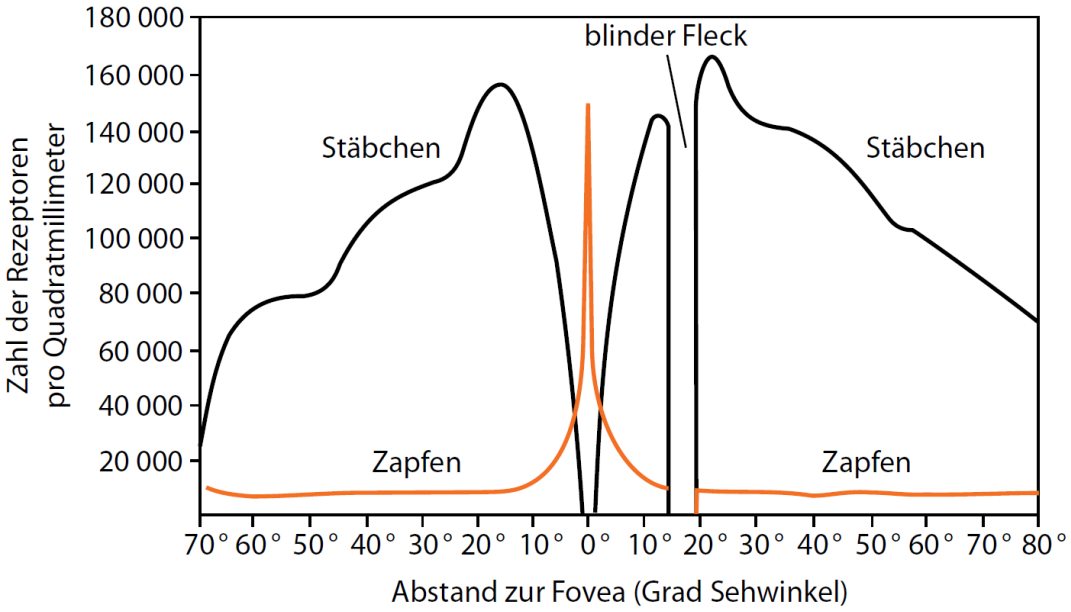


Abbildung 3.3: Nicht-uniforme Verteilung der Stäbchen und Zapfen über die Netzhaut. Im Bereich der Fovea erreicht die Verteilungsdichte der Zapfen ihr Maximum, allerdings gibt es dort keine Stäbchen. In der peripheren Retina ist hingegen die Anzahl der Zapfen gering, während sich dort viele Stäbchen befinden. Am blinden Fleck gibt es keine Fotorezeptoren. Quelle: [FM14]

und blau zu einer resultierenden, wahrgenommenen Farbe addiert. Es werden vielmehr Differenzen zwischen zwei Gegenfarben gebildet, die als ein achromatischer und zwei chromatische Kanäle einen Gesamtfarbeindruck ergeben. Insgesamt befinden sich auf der Netzhaut circa 120 Millionen Stäbchen und etwa sechs bis sieben Millionen Zapfen. In Abbildung 3.3 ist zu erkennen, dass die Fotorezeptoren nicht-uniform über die Netzhaut verteilt sind. Die Fovea centralis ist eine Einsenkung der Netzhaut inmitten des gelben Flecks und wird auch als Sehgrube bezeichnet. Im Bereich der Sehgrube, welche einen Durchmesser von etwa 1,5 mm hat, befinden sich keine Stäbchen, jedoch erreicht die Verteilung der Zapfen dort ihre höchste Dichte. Die Fovea centralis wird umgeben von der Parafovea und der Perifovea, welche ebenfalls zum gelben Fleck gehören. Am blinden Fleck befinden sich weder Stäbchen noch Zapfen. Durch die nicht-uniforme Verteilung der Fotorezeptoren auf der Netzhaut kann nur ein Bruchteil des Sichtfeldes, welches sich über einen Bereich von etwa 200° horizontal und 160° vertikal erstreckt, scharf wahrgenommen werden. Die hohe Dichte an Zapfen im gelben Fleck ermöglicht die Wahrnehmung von scharfen Bildern in einem Radius von etwa 10° um die Fovea centralis herum. Man spricht bei dieser Art der Wahrnehmung auch von *fovealem Sehen*. Periphere Bereiche können hingegen nur unscharf wahrgenommen werden [RL15]. Durch die rasante Weiterentwicklung von Rechnern und Anzeigegeräten wird es zunehmend wichtiger, die Eigenschaften des menschlichen Sehens in der Computergrafik zu berücksichtigen.

3.2 Computergrafik

In diesem Abschnitt werden Grundlagen der Computergrafik näher erläutert. Neben verschiedenen Farbräumen, die im entwickelten System Verwendung finden, werden auch das Konzept der Bildsynthese sowie der Raytracing-Algorithmus genauer betrachtet.

3.2.1 Farbmodelle und Farbräume

Ein Farbmodell ist eine abstrakte, mathematische Beschreibung, wie Farben als Tupel von Zahlen dargestellt werden können. Durch Anwendung eines Farbmodells mit konkreten Regeln zur Interpretation des abstrakten Modells entsteht ein Farbraum. In der Computergrafik wird versucht, mit Farbmodellen die menschliche Farbwahrnehmung zu modellieren. Es gibt zahlreiche verschiedene Farbmodelle, die sich in ihrer Komplexität und ihrem beabsichtigten Einsatzgebiet unterscheiden. Eines der bekanntesten Modelle ist das RGB-Farbmodell, welches aufgrund seiner Einfachheit als Grundlage vieler Monitore sowie zur Repräsentation von Bildern dienen kann.

RGB-Farbmodell

Die meisten heutigen Displays erzeugen ein Ausgabebild, indem für jedes Pixel Licht der drei Grundfarben **r**ot, **g**rün und **b**lau mit einer bestimmten Intensität überlagert wird, um dem menschlichen Auge einen Farbeindruck zu vermitteln. In ähnlicher Weise lassen sich Bilder Rechnerintern repräsentieren, indem für jedes Pixel des Bildes drei Werte, die RGB-Komponenten, abgespeichert werden. Durch additive Farbmischung der drei Komponenten entsteht für jedes Pixel eine bestimmte Farbe. Die einzelnen Farbkomponenten werden auch als Kanäle bezeichnet, wobei häufig 8 Bit pro Kanal und Pixel abgespeichert werden. Dadurch ergeben sich insgesamt $2^8 \times 2^8 \times 2^8 = 256 \times 256 \times 256 = 16777216$ verschiedene mögliche Farbeindrücke, die mit diesem Modell dargestellt werden können. Die Gesamtheit der auf diese Weise erzeugten Farben ist der RGB-Farbraum und lässt sich anschaulich als Würfel darstellen, wie in Abbildung 3.4 zu sehen ist. In manchen Fällen ist es wünschenswert, dass neben den drei Farbkomponenten noch eine weitere Information über die Undurchsichtigkeit eines Pixels abgespeichert wird, um beispielsweise transparente Bereiche in Bildern zu erhalten. Durch die Erweiterung des RGB-Farbmodells um einen weiteren Kanal, den Alphakanal, wird dies ermöglicht [HDM+13; SM09].

YUV-Farbmodell

Statt drei Farben wie beim RGB-Farbmodell, die zu gleichen Anteilen die resultierende Farbe bestimmen, gibt es beim YUV-Farbmodell eine Helligkeits- (Y) und zwei Farbkomponenten (U und V). Die Komponenten werden auch als Luminanz (Helligkeitssignal) und Chrominanz (Farb-

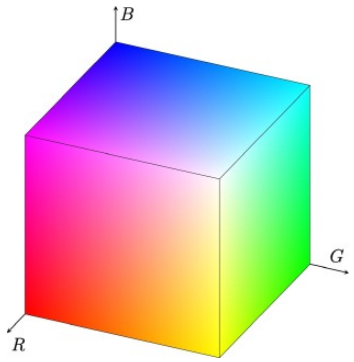


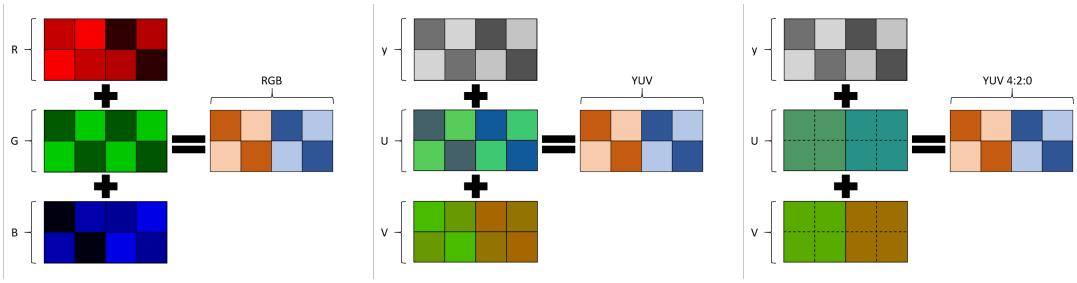
Abbildung 3.4: Der RGB-Farbraum kann anschaulich durch einen Würfel repräsentiert werden. Die Grundfarben rot, grün und blau stellen dabei jeweils eine der drei Koordinatenachsen dar. Quelle: [CAB15]

signal) bezeichnet. Der Luminanzwert lässt sich ebenso wie die beiden Chrominanzwerte direkt aus der RGB-Repräsentation einer Farbe berechnen. Dafür gibt es die folgende Umrechnungsmatrix [BB16]:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3.1}$$

Das YUV-Farbmodell ist näher an die in Abschnitt 3.1 beschriebenen Eigenschaften des visuellen Systems angelehnt. Da die helligkeitssensitiven Stäbchen in deutlich höherer Zahl als die farbsensitiven Zapfen vorkommen, werden Helligkeitsinformation stärker wahrgenommen als Farbinformationen. Diese Eigenschaft kann ausgenutzt werden, indem die Ortsauflösung der Chrominanz reduziert wird und dadurch Speicherplatz sowie benötigte Bandbreite bei einer Übertragung gespart wird. Diesen Vorgang nennt man auch Farbunterabtastung. Eines der gebräuchlichsten Formate, bei dem Farbunterabtastung zum Einsatz kommt, ist YUV 4:2:0. Die Zahlen beziehen sich dabei auf das Verhältnis zwischen Luminanz- und Chorminanzwerten. Ursprünglich bezog sich das 4:2:0 Format auf einen 4×2 großen Pixelblock, bei dem **4** Luminanzwerte pro Pixelreihe, allerdings nur **2** Chrominanzwerte in der oberen Pixelreihe und **0** Chrominanzwerte in der unteren Pixelreihe abgespeichert werden. Heutzutage ist mit einer Farbunterabtastung im 4:2:0-Format gemeint, dass für jedes Pixel eine Y-Komponente abgespeichert wird. Die U- und V-Komponenten für einen 2×2 großen Pixelwert erhält man, indem ein Farbmittelwert gebildet wird, der für alle vier Pixel gilt, aber nur einmal abgespeichert werden muss. Das YUV-Modell mit Farbunterabtastung kommt vor allem bei der Videocodierung zum Einsatz.

Abbildung 3.5 zeigt beispielhaft die Repräsentation eines 4×2 großen Pixelblocks in verschiedenen Farbmodellen.



(a) RGB **(b)** YUV **(c)** YUV 4:2:0

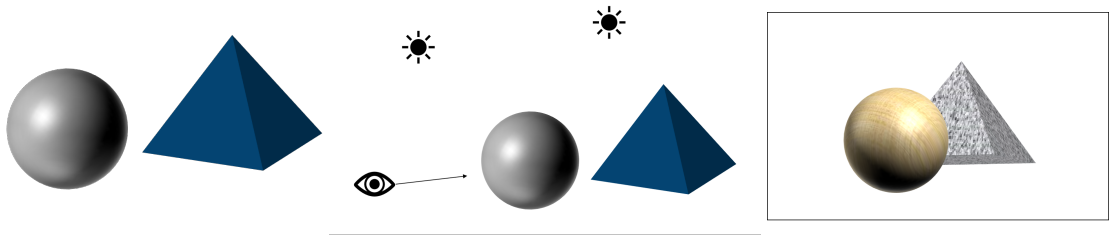
Abbildung 3.5: (a) RGB-Repräsentation eines Pixelblocks. Drei Kanäle für rot, grün und blau werden zu einem Gesamtbild addiert. (b) YUV-Repräsentation eines Pixelblocks. Eine Luminanz- (Y) sowie zwei Chrominanzkomponenten (U und V) werden abgespeichert. (c) Der selbe Pixelblock mit Farbunterabtastung im Format 4:2:0. Für einen 2×2 großen Pixelblock wird in der U- und V-Komponente jeweils nur eine Farbe abgelegt. Die Reduktion der Farbinformation hat kaum wahrnehmbare Auswirkungen auf das Gesamtbild.

3.2.2 Bildsynthese

Unter dem Begriff Bildsynthese versteht man in der Computergrafik das Erstellen eines Bildes aus gegebenen Rohdaten, welche zum Beispiel die geometrische Beschreibung eines 3D-Modells sein können. Ein *Modell* besteht aus einem oder mehreren Objekten, welche durch eine Menge von Punkten definiert sind. Die Punkte sind dabei durch verschiedene geometrische Formen wie Linien oder Dreiecke verbunden und ergeben so einen physischen Körper. Einem Objekt können verschiedene Eigenschaften wie die Oberflächenbeschaffenheit oder die Farbe zugeordnet werden. Werden neben einem Modell, das durch Objekte und deren Materialeigenschaften charakterisiert ist, auch Lichtquellen sowie die Position und Blickrichtung eines Betrachters definiert, spricht man von einer *virtuellen Szene*. Mit verschiedenen Bildsynthese-Techniken lässt sich aus einer Szene ein zweidimensionales Bild generieren. Dieser Vorgang wird auch als *Rendern* bezeichnet. In Abbildung 3.6a ist ein Beispiel für ein Modell zu sehen, welches aus zwei Objekten besteht. Abbildung 3.6b zeigt eine virtuelle Szene, die sich durch Ergänzen des Modells um einen Betrachter und dessen Blickrichtung sowie zwei Lichtquellen ergibt. Beim Bilderzeugungsprozess wird aus einer virtuellen Szene unter Berücksichtigung der Materialeigenschaften der einzelnen Objekte ein Bild gerendert (Abbildung 3.6c).

In der realen Welt entsteht ein wahrnehmbares Bild, indem Licht an verschiedenen Oberflächen reflektiert wird und schließlich in das menschliche Auge gelangt. In ähnlicher Weise muss man beim Rendern herausfinden, wie sich Licht in einer Szene verhält und welches Licht einen Betrachter erreicht und somit ein Bild auf dem Display ergibt. Die in Gleichung 3.2 gezeigte Rendering-Gleichung von Kajiya [Kaj86] liefert dafür einen Ansatz.

$$L(x,x') = g(x,x') \cdot \left(L_e(x,x') + \int_S b(x,x',x'') L(x',x'') \mathrm{d}x'' \right) \tag{3.2}$$



(a) Modell **(b)** virtuelle Szene **(c)** gerendertes Bild

Abbildung 3.6: (a) Ein Modell, das durch zwei Objekte definiert ist. Für jedes Objekt sind verschiedene Eigenschaften wie die Farbe oder Oberflächenbeschaffenheit abgespeichert. Sie beeinflussen das Erscheinungsbild dieses Objektes beim Rendern. (b) Wird ein Modell um einen Betrachter und dessen Blickrichtung sowie eine oder mehrere Lichtquellen erweitert, spricht man von einer virtuellen Szene. (c) Dieses Bild wird für den Betrachter sichtbar, wenn die virtuelle Szene unter Berücksichtigung der Materialeigenschaften, des Lichtflusses und der Blickrichtung gerendert wird.

Diese Formel charakterisiert die Lichtverteilung in einer virtuellen Szene. Sie gibt an, wie viel Licht den Oberflächenpunkt x von einem anderen Punkt x' erreicht, wobei ein dritter Oberflächenpunkt x'' berücksichtigt wird. Im Folgenden wird kurz die Bedeutung der einzelnen Bestandteile der Formel erklärt. Der Energiefluss $L(x,x')$ gibt an, wie viel Licht x von x' aus erreicht. In der selben Weise beschreibt der Term $L(x',x'')$, wie viel Licht x' von x'' aus erreicht. $g(x,x')$ ist ein geometrischer Term und beschreibt die gegenseitige Lage der Punkte x und x' in der Szene. Wenn zwischen x und x' ein undurchsichtiges Objekt liegt, hat dieser Term den Wert 0 und es kommt kein Licht von x' bei x an. Für den Fall, dass x' eine Lichtquelle ist, gibt der Emissionsterm $L_e(x,x')$ an, wie viel Licht von x' nach x abgestrahlt wird. Der Streuungsterm $b(x,x',x'')$ hängt mit der Intensität des gestreuten Lichts zusammen, welches x' von x'' aus erreicht und von dort in Richtung x reflektiert wird. S steht für die Gesamtheit aller Flächen in der virtuellen Szene.

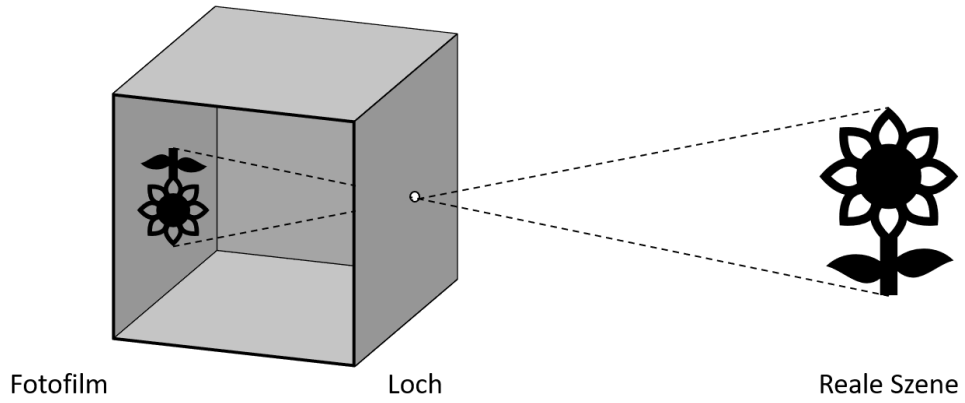
In der Praxis versuchen verschiedene Rendering-Techniken die Rendering-Gleichung so exakt wie möglich zu lösen, um eine realistische Beleuchtungsberechnung zu erhalten. Aufgrund von beschränkten Rechenkapazitäten und limitierter Berechnungszeit kann die Gleichung in den meisten Fällen nur approximiert werden.

Bei verschiedenen Rendering-Techniken unterscheidet man zwischen objektbasiertem (engl. *Object-Order Rendering*) und bildbasiertem (engl. *Image-Order Rendering*) Rendern. Beim Object-Order Rendering wird ein Objekt nach dem anderen betrachtet und bestimmt, auf welche Pixel des Ausgabebildes dieses Objekt einen Einfluss hat. Anschließend wird für diese Pixel die entsprechende Farbe berechnet. Objektbasiertes Rendern hat den Vorteil, dass Szenen mit wenigen Objekten ohne großen Aufwand gerendert werden können. Im Gegensatz dazu wird beim bildbasierten Rendern ein Pixel nach dem anderen betrachtet. Für jedes Pixel wird zunächst ermittelt, welches Objekt an dieser Stelle sichtbar ist und anschließend die Pixelfarbe für das finale Bild bestimmt. Ein Algorithmus zur Bildsynthese, welcher bildbasiert arbeitet, ist das sogenannte Raytracing.

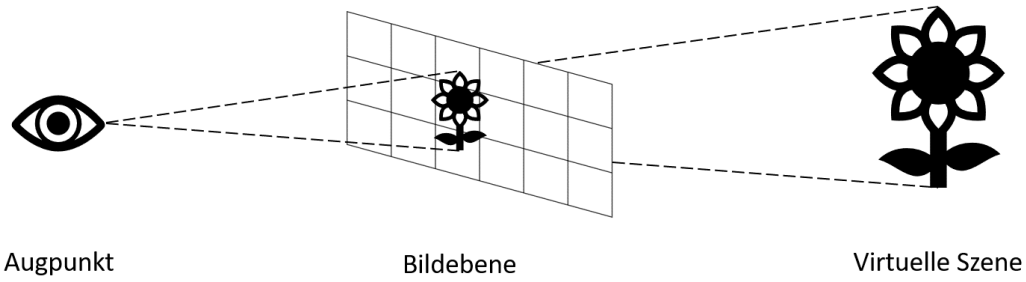
3.2.3 Raytracing

Der Begriff Raytracing stammt aus dem Englischen und bedeutet *Strahlenverfolgung*. Gemeint ist damit die Verfolgung von Lichtstrahlen in einer virtuellen Szene, um ein gegebenes Modell physikalisch korrekt zu rendern. Der Raytracing-Algorithmus ist an das Lochkamera-Modell angelehnt.

Die Lochkamera ist eines der einfachsten Modelle, um ein 2D-Abbild der (3D-)Realität zu erzeugen. Der Aufbau einer Lochkamera besteht, wie in Abbildung 3.7a zu sehen ist, aus einer lichtdichten, rechteckigen Box, die auf einer Seite ein kleines Loch hat. Auf die dem Loch gegenüberliegende Innenseite der Box wird ein lichtempfindlicher Fotofilm aufgelegt. Sobald Licht durch das Loch in die Box fällt, wird eine chemische Reaktion auf dem Fotofilm ausgelöst, wodurch ein spiegelverkehrtes Bild entsteht.



(a) Lochkamera



(b) Raytracing

Abbildung 3.7: (a) Schematischer Aufbau einer Lochkamera. Durch das Loch fallen Lichtstrahlen, die auf der Innenseite der Box, wo ein lichtempfindlicher Fotofilm aufgelegt wird, ein spiegelverkehrtes Bild erzeugen. (b) Beim Raytracing wird in ähnlicher Weise wie bei der Lochkamera ein Bild einer virtuellen Szene erzeugt, allerdings sind Loch und Fotofilm vertauscht. Man spricht stattdessen auch von Augpunkt und Bildebene.

Der Raytracing-Algorithmus basiert auf dem Prinzip der Lochkamera, allerdings sind Loch und Fotofilm vertauscht. Man spricht statt Loch und Fotofilm beim Raytracing auch von Augpunkt und Bildebene. Die Bildebene ist dabei ein Raster von Pixeln, welches den Ausschnitt des Bildes festlegt, der später auf dem Bildschirm sichtbar wird. Ähnlich wie bei der Lochkamera wird ermittelt, welche Lichtstrahlen auf den Augpunkt treffen und dadurch eine Abbildung in der Bildebene ergeben. Lichtstrahlen, welche auf das Auge treffen, sind äquivalent zu jenem Licht, das bei der Lochkamera das Loch passiert. Dieser Sachverhalt ist in Abbildung 3.7b vereinfacht dargestellt.

Lichtquellen emittieren Licht in Form von Millionen von Photonen pro Sekunde in alle Raumrichtungen. Eine Verfolgung des Lichtes von der Quelle zum Betrachter wäre daher zu aufwendig. Zudem hat ein Großteil der Lichtstrahlen keinen Einfluss auf das finale Bild, da die meisten Strahlen nicht auf den Augpunkt treffen. Stattdessen wird das Licht beim Raytracing vom Betrachter (Augpunkt) zur Lichtquelle zurückverfolgt. Dafür wird für jedes Pixel der Bildebene ein Strahl erzeugt, der als Halbgerade definiert ist, die sich anfangend im Augpunkt durch das entsprechende Pixel bis ins Unendliche erstreckt. Diese Art von erzeugten Strahlen wird im Folgenden als *Primärstrahlen* oder *Sichtstrahlen* bezeichnet.

Das prinzipielle Vorgehen beim Raytracing ist aufgeteilt in drei Schritte:

1. Erzeugung der Sichtstrahlen
2. Schnittberechnung
3. Schattierung/Beleuchtungsberechnung

Für die Erzeugung der Sichtstrahlen müssen Position und Blickrichtung des Betrachters sowie der Abstand zur Bildebene bekannt sein. Jeder Sichtstrahl hat seinen Ursprung im Augpunkt und erstreckt sich von dort in Richtung eines bestimmten Pixels. Mathematisch kann ein Sichtstrahl $\vec{s}$ beschrieben werden als

$$\vec{s} = \vec{o} + t \cdot \vec{p} \text{ mit } t \in \mathbb{R}.$$

Dabei steht $\vec{o}$ für die Position des Betrachters und $\vec{p}$ für den Vektor, der sich vom Augpunkt in Richtung des entsprechenden Pixels erstreckt.

Für jeden Sichtstrahl wird nach der Erzeugung bestimmt, welche Objekte in der Szene er schneidet. Dasjenige Objekt, dessen Schnittpunkt den geringsten Abstand zum Augpunkt hat, verdeckt gegebenenfalls andere Objekte und ist daher für den Betrachter in der Bildebene zu sehen.

Nach der Erzeugung eines Primärstrahls wird für das entsprechende Pixel eine Variable d angelegt, die eine Distanz abspeichert und mit ∞ initialisiert wird. Dieser Wert wird benötigt, um feststellen zu können, welches Objekt die kürzeste Distanz zum Betrachter in diesem Pixel hat. Außerdem wird eine Variable o benötigt, die das entsprechende Objekt abspeichert.

Anschließend wird für jedes Objekt in der Szene ein Schnitttest mit dem Sichtstrahl durchgeführt. Dabei wird berechnet, ob der Sichtstrahl das Objekt in einem oder mehreren Punkten, wie es zum Beispiel bei Kugeln der Fall sein kann, schneidet. Dieser Schritt erfordert den größten Rechenaufwand beim Raytracing, weshalb häufig spezielle Datenstrukturen zur Optimierung verwendet werden. Gibt es keinen Schnittpunkt zwischen dem Primärstrahl und einem Objekt, so ist dieses Objekt für den Betrachter an dieser Stelle des Bildes (Pixel) nicht sichtbar. In diesem Fall fährt die

Berechnung mit dem nächsten Objekt fort oder wird beendet, falls alle Objekte abgearbeitet wurden. Wenn der Schnitttest einen oder mehrere Schnittpunkte ergibt, wird der Abstand des Schnittpunktes zum Augpunkt berechnet. Ist der berechnete Wert kleiner als der momentan abgespeicherte Wert d, so wird die Variable d mit dem berechneten Wert überschrieben. Außerdem wird in der Variable o abgespeichert, welches Objekt in diesem Fall betrachtet wurde. So wird sichergestellt, dass nach Durchlauf aller Objekte dasjenige Objekt abgespeichert ist, welches die kürzeste Distanz zum Augpunkt hat und somit für den Betrachter sichtbar ist. Für den Fall, dass es mehrere Schnittpunkte für ein Objekt gibt, wird immer nur der Schnittpunkt mit der kürzesten Distanz zum Augpunkt berücksichtigt.

Für Sichtstrahlen, die kein einziges Objekt schneiden, wird im korrespondierenden Pixel eine festgelegte Hintergrundfarbe abgespeichert. Für alle anderen Pixel findet abhängig vom gespeicherten Objekt eine Beleuchtungsberechnung statt, um die finale Farbe des Pixels zu bestimmen.

Um das farbliche Erscheinungsbild eines Objektes zu bestimmen, muss berücksichtigt werden, in welcher Weise Licht auf dieses Objekt trifft. Möglich ist beispielsweise, dass ein Objekt direkt von einer Lichtquelle angestrahlt wird, über Reflexion an einem anderen Objekt indirekt beleuchtet wird oder im Schatten liegt, da es durch ein anderes Objekt verdeckt wird. Trifft Licht auf eine Oberfläche, so wird ein Teil des Lichtes absorbiert, während ein anderer Teil reflektiert wird. Verschiedene Materialien reflektieren das auf sie treffende Licht in unterschiedlicher Weise. Deshalb spielt es für die Beleuchtungsberechnung zudem eine Rolle, aus welchem Material ein Objekt besteht.

Aus diesem Grund werden nach dem Schnitttest mit den Sichtstrahlen an einem gefundenen Schnittpunkt weitere Strahlen, auch Sekundärstrahlen genannt, erzeugt.

Pro Lichtquelle in der Szene wird ein Strahl generiert, der seinen Ursprung im gefundenen Schnittpunkt hat und in Richtung der Lichtquelle verläuft. Auf diese Weise kann bestimmt werden, ob ein Objekt in Bezug auf eine bestimmte Lichtquelle im Schatten liegt, weshalb man diese Strahlen auch Schattenstrahlen nennt.

In gleicher Weise wie bei den Primärstrahlen wird auch bei den Schattenstrahlen ein Schnitttest mit allen Objekten durchgeführt. Schneidet ein Schattenstrahl auf dem Weg vom Ausgangspunkt (Schnittpunkt) zur Lichtquelle kein anderes Objekt, so wird die Oberfläche an dieser Stelle direkt beleuchtet. Andernfalls liegt der Ausgangspunkt in Bezug auf die betrachtete Lichtquelle im Schatten, da zwischen der Lichtquelle und dem Ausgangspunkt ein anderes Objekt liegt. Ausnahmen bilden lichtdurchlässige Objekte wie beispielsweise Glas. Abhängig davon, ob ein Schattenstrahl ein anderes Objekt schneidet oder nicht, wird eine bestimmte Lichtintensität für diesen Strahl abgespeichert.

Man unterscheidet bei der Reflexion von Licht zwischen spekularen bzw. perfekt spiegelnden, glänzenden bzw. imperfekt spiegelnden und diffusen Oberflächen. Des Weiteren gibt es Materialien, in die Licht eindringt und an einer anderen Stelle wieder austritt. Dieser Vorgang wird auch Transmission genannt. Um diese Eigenschaften bei der Beleuchtungsberechnung zu berücksichtigen, werden weitere Sekundärstrahlen an einem Schnittpunkt erzeugt. Abhängig von der Beschaffenheit des Objektes werden diese als *Reflexionsstrahlen* oder *Transmissionsstrahlen* bezeichnet. Im Folgenden wird das Prinzip der Reflexions- und Transmissionsstrahlen der Einfachheit halber nur für perfekt spiegelnde Oberflächen erläutert, da Licht an diesen nur in eine Richtung abgelenkt wird und somit nur ein Sekundärstrahl verfolgt werden muss. Auch glänzende und diffuse Materialien reflektieren auf sie treffendes Licht, allerdings in verschiedene Raumrichtungen, weshalb dort mehr Sekundärstrahlen erzeugt werden müssen. Das Grundprinzip ist jedoch für spiegelnde, glänzende, diffuse und transmittierende Materialien gleich.

Trifft Licht mit einem bestimmten Einfallswinkel auf eine spekulare Oberfläche, so wird es mit dem gleichen Ausfallswinkel reflektiert. Für den Raytracing-Algorithmus folgt daraus, dass an einem Schnittpunkt des Sichtstrahls mit einem perfekt spiegelnden Objekt ein Reflexionsstrahl erzeugt wird.

Die Transmissions- und Reflexionsstrahlen werden in gleicher Weise wie ein Primärstrahl verfolgt und können gegebenenfalls andere Objekte schneiden, an denen weitere Sekundärstrahlen erzeugt werden. Auf diese Weise ergibt sich eine Rekursion, die theoretisch unendlich lange fortgesetzt werden kann. In der Praxis wird deshalb meistens eine Rekursionstiefe angegeben, die festlegt, an wie vielen unterschiedlichen Schnittpunkten Sekundärstrahlen erzeugt werden.

Die finale Farbe eines Pixels ergibt sich aus der Aufsummierung mehrerer Komponenten. Dazu gehören die Farbe, die das Objekt am Schnittpunkt mit dem Primärstrahl hat, sowie die Intensitäten und Farben der Sekundärstrahlen.

In Abbildung 3.8 ist das Prinzip des Raytracing-Algorithmus vereinfacht dargestellt und wird für die Bestimmung der Farbe eines Pixels mithilfe des zugehörigen Primärstrahls erklärt.

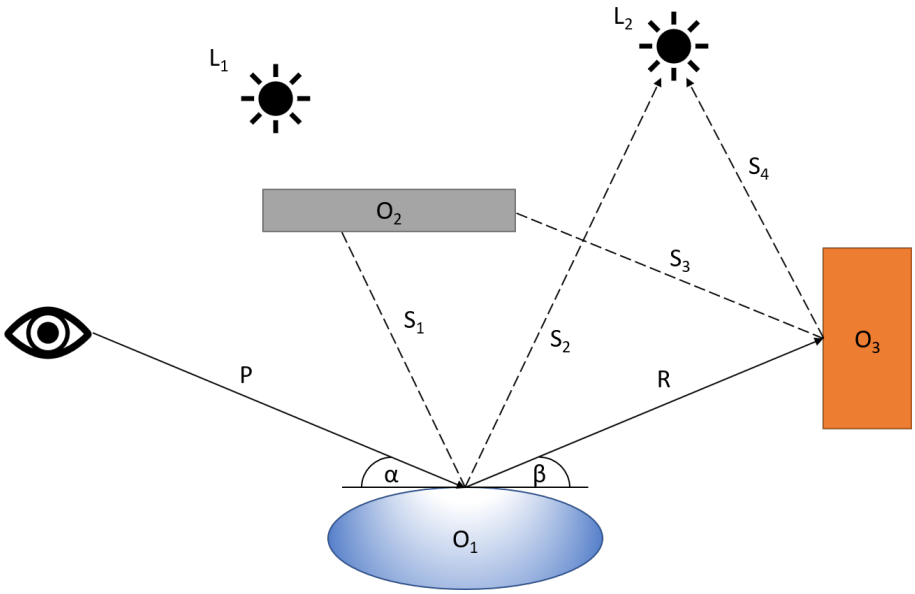


Abbildung 3.8: Vereinfachtes Prinzip des Raytracing-Algorithmus. Der Primärstrahl P schneidet das Objekt O_1, dessen Oberfläche perfekt spiegelnd ist, in einem Winkel α. Vom Schnittpunkt aus werden zwei Schattenstrahlen S_1 und S_2 in Richtung der beiden Lichtquellen L_1 und L_2 ausgesandt. Außerdem wird ein Reflexionsstrahl R mit einem Ausfallswinkel β erzeugt, wobei $\alpha = \beta$ gilt. Dieser schneidet das Objekt O_3, welches eine diffuse Oberfläche hat und aufgrund der vereinfachten Darstellung kein Licht reflektiert. Am Schnittpunkt des Reflexionsstrahls R mit dem Objekt O_3 werden weitere Schattenstrahlen S_3 und S_4 erzeugt. Die Objekte O_1 und O_3 liegen in Bezug auf die Lichtquelle L_1 im Schatten und werden von der Lichtquelle L_2 direkt angestrahlt. Die finale Farbe für das zum Primärstrahl P gehörende Pixel ergibt sich durch Aufsummierung verschiedener Komponenten wie den Oberflächenfarben von O_1 und O_3 sowie mehreren Licht- und Schattenanteilen.

Trotz der stetig wachsenden technischen Möglichkeiten ist Raytracing für komplexe Szenen und hochaufgelöste Bilder mit einer großen Anzahl an Pixeln noch zu aufwendig, um es in Echtzeit zu nutzen. Aus diesem Grund werden häufig die in Abschnitt 3.1 beschriebenen Eigenschaften der menschlichen Wahrnehmung in den Bilderzeugungsprozess miteinbezogen, was auch als *Foveated Rendering* bezeichnet wird.

3.3 Videocodierung

Die Videocodierung verfolgt das Ziel, bewegte Bilder zu komprimieren, um Speicherplatz bei möglichst gleichbleibender Bildqualität einzusparen. Bei der Videokompression kommen vor allem folgende zwei Prinzipien zum Einsatz. Neben der *Redundanzreduktion*, welche von Ähnlichkeiten zwischen mehreren Bildern profitiert, wird die *Irrelevanzreduktion* eingesetzt, mithilfe derer Eigenschaften des menschlichen Sehens ausgenutzt werden. Man unterscheidet dabei zwischen verlustfreier und verlustbehafteter Kompression.

Bei der Redundanzreduktion werden gleichbleibende Teile eines Bildes nicht erneut codiert. So kann vor allem in statischen Bereichen, wo keine Bewegung stattfindet und das Bild sich deshalb nicht ändert, viel Speicherplatz gespart werden. Für dynamische Bereiche, wo sich Objekte bewegen, wird mithilfe eines Bewegungsvektors abgespeichert, in welche Richtung sich dieses Objekt bewegt. Ein Decoder kann aus diesen Informationen das Ausgangsbild exakt rekonstruieren, weshalb man bei der Redundanzreduktion auch von verlustfreier Kompression spricht.

Im Gegensatz dazu gehen bei der Irrelevanzreduktion Informationen verloren, weshalb diese als verlustbehaftet bezeichnet wird. Bei dieser Art der Kompression werden Bildinformationen, die durch das menschliche Auge kaum wahrnehmbar und somit irrelevant für einen Betrachter sind, weggelassen. Häufig geschieht dies über eine Farbunterabtastung, da das visuelle System auf Farben nicht so empfindlich reagiert wie auf Helligkeit. Oft wird das in Unterabschnitt 3.2.1 beschriebene 4:2:0-Format zur Abtastung verwendet, um hohe Kompression bei subjektiv fast gleicher Bildqualität zu erreichen.

Für die Kompression werden unterschiedliche Standards eingesetzt, welche sich in Kompressionsrate und Qualität unterscheiden. Zwei der bekanntesten und am häufigsten eingesetzten Standards sind H.264 sowie H.265, welche auch im entwickelten System Verwendung finden. Sie arbeiten beide nach einem ähnlichen Prinzip, welches im Folgenden grob erläutert wird.

Grundsätzlich werden Einzelbilder, auch *Frames* genannt, nicht einzeln codiert, sondern als eine Bildergruppe, engl. *group of pictures* (GOP). Bei der Codierung wird unterschieden zwischen folgenden drei Arten von Frames:

- I-Frame
- P-Frame
- B-Frame

Ein **I**ntra-Frame, auch Keyframe (dt. Schlüsselbild) genannt, enthält alle Bildinformationen und kann unabhängig von anderen Frames decodiert werden. Bei der Codierung von I-Frames findet keine Redundanzreduktion statt, da alle Informationen codiert werden. Ein **P**-Frame nutzt die Bildinformationen des vorhergehenden Frames, um eine Vorhersage des aktuellen Frames zu

treffen. Man spricht deshalb auch von einem ***p**redicted picture*, was übersetzt *vorhergesagtes Bild* bedeutet. **B**-Frames nutzen zusätzlich auch die Informationen des nachfolgenden Frames für eine Vorhersage und werden deshalb als ***b**idirectional predicted picture* bezeichnet. Am Anfang einer GOP muss immer ein I-Frame stehen, damit ein Decoder genügend Bildinformationen hat, um mit dem Decodiervorgang beginnen zu können.

In Abbildung 3.9 ist ein Beispiel für eine GOP mit sechs Frames zu sehen. Am Anfang der Bildergruppe steht immer ein I-Frame, der alle Bildinformationen enthält. Zwischen zwei I-Frames können beliebig viele P- und B-Frames stehen, allerdings wird häufig eine feste Größe angegeben, die den Abstand zwischen zwei I-Frames bestimmt.

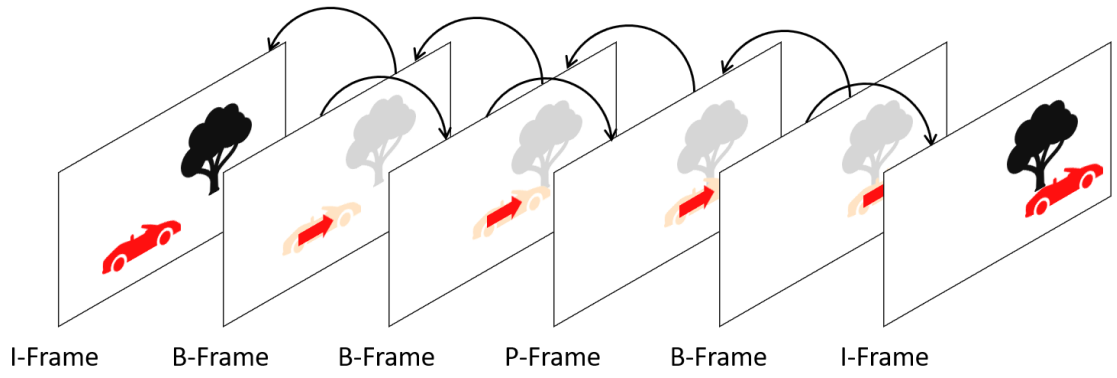


Abbildung 3.9: Beispiel einer Bildergruppe (GOP). Am Anfang einer GOP steht immer ein I-Frame, der alle Bildinformationen enthält. Zwischen zwei I-Frames können beliebig viele P- und B-Frames stehen, die ihre Informationen aus den vorhergehenden und nachfolgenden Frames beziehen (schwarze Pfeile). Im Beispiel bewegt sich ein Auto durch eine statische Szene, die durch den Baum repräsentiert ist. Für die Bewegung des Autos werden in den P- und B-Frames Bewegungsvektoren abgespeichert (rote Pfeile). Da sonst keine Änderungen an der Szene stattfinden, wird für den restlichen Teil des Bildes eine Redundanzreduktion ausgeführt.

Für die Codierung eines Frames erfolgt eine Aufteilung des Bildes in kleinere Bereiche, die sogenannten Makroblöcke, welche separat codiert werden. Die Größe eines Makroblocks kann dabei zwischen 4×4 und 64×64 Pixeln variieren, wobei Länge und Breite des Makroblocks immer eine Zweierpotenz sind. Abhängig von der Art des Frames wird ein Makroblock entweder direkt codiert (I-Frame) oder es findet ein Vergleich mit dem entsprechenden Makroblock des vorangegangenen Frames (P-Frame) bzw. des vorangegangenen und des nachfolgenden Frames (B-Frame) statt und es werden die Unterschiede zwischen diesen Makroblöcken codiert. Wenn keine Veränderungen in einem Makroblock auftreten, findet für diesen keine zusätzliche Codierung statt [Fis16].

3.3.1 H.264 / MPEG-4 AVC

Der H.264-Standard [WSBL03] ist Teil des MPEG-4-Standards und wurde von der Moving Picture Experts Group (MPEG) entwickelt. Daher wird er auch als *MPEG-4 AVC* bezeichnet, wobei AVC für *Advanced Video Coding* steht, was übersetzt *Erweiterte Videocodierung* bedeutet. Die

Funktionsweise beruht, wie oben beschrieben, auf der Aufteilung des zu codierenden Bildes in Makroblöcke und der Unterscheidung zwischen I-, P- und B-Frames. H.264 wurde für HD-Videos entwickelt und findet deshalb zum Beispiel bei TV-Bildern oder DVDs Anwendung.

3.3.2 HEVC / H.265 / MPEG-H Teil 2

High Efficiency Video Coding (HEVC) beziehungsweise der H.265-Standard [SOHW12] ist der Nachfolger des H.264-Standards und wird deshalb auch *MPEG-H Teil 2* genannt. Im Vergleich zu seinem Vorgänger bietet H.265 bei ähnlicher Funktionalität einige klare Verbesserungen. Eine Veränderung der inneren Datenstrukturen ermöglicht eine verdoppelte Kompressionsrate bei gleichbleibender Qualität im Vergleich zum H.264-Standard. Dadurch ist H.265 auch für Videos jenseits der Full HD Auflösung, beispielsweise mit 4K oder 8K, einsetzbar. Allerdings ist H.265 bezogen auf die benötigte CPU-Leistung sehr rechenintensiv. H.265 benutzt statt Makroblöcken eine veränderte Datenstruktur, die sogenannten Coding Tree Units (CTU), welche sich in der Funktionsweise kaum unterscheiden, aber deutlich mehr Flexibilität bieten.

3.4 VISUS-Powerwall

In diesem Abschnitt wird die sogenannte Powerwall, welche als Anzeige für das entwickelte System dient, näher beschrieben. Die Powerwall ist ein hochauflösendes, gekacheltes Display mit einer Größe von etwa $6 \times 2{,}2$ m. Insgesamt zehn Projektoren mit einer 4K-Auflösung sorgen dafür, dass auf der Powerwall zwei Bilder mit einer Auflösung von 10800×4096 Pixeln abgebildet werden, wodurch der Eindruck von Stereoskopie entsteht. Das Ausgabebild der einzelnen Projektoren wird, wie in Abbildung 3.10 zu sehen ist, mithilfe von Spiegeln auf die Projektionsleinwand abgelenkt.

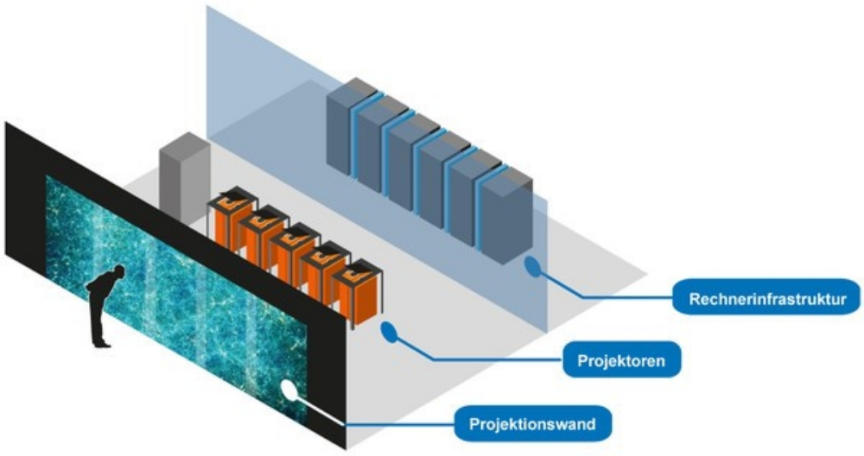


Abbildung 3.10: Skizze der VISUS-Powerwall. Ein Betrachter steht vor der $6 \times 2{,}2$ m großen Projektionsleinwand, welche von hinten durch zehn (5×2) Projektoren angestrahlt wird. Diese werden durch eine gesonderte Rechnerinfrastruktur mit Bilddaten versorgt. Quelle: VISUS[1]

[1] https://www.visus.uni-stuttgart.de/institut/visualisierungslabor/

Die Projektoren sind dabei nicht wie üblich im Landscape-Modus (dt. Querformat), sondern im Porträt-Modus (Hochformat) ausgerichtet. Jeweils zwei Projektoren - einer für das linke und der andere für das rechte Auge - sind für einen Bereich von 2400×4096 Pixel des Gesamtbildes verantwortlich. Durch die Anordnung der Projektoren und das daraus resultierende Überlappen der einzelnen Bereiche ergibt sich, wie in Abbildung 3.11 zu sehen ist, die Gesamtauflösung von 10800×4096 Pixel. Es entsteht jeweils ein etwa 300 Pixel breiter Streifen, wo sich zwei Bereiche überlappen. Die paarweise Verwendung der Projektoren ermöglicht die Anzeige von stereoskopischen oder dreidimensionalen Bildern mit Tiefeneindruck.

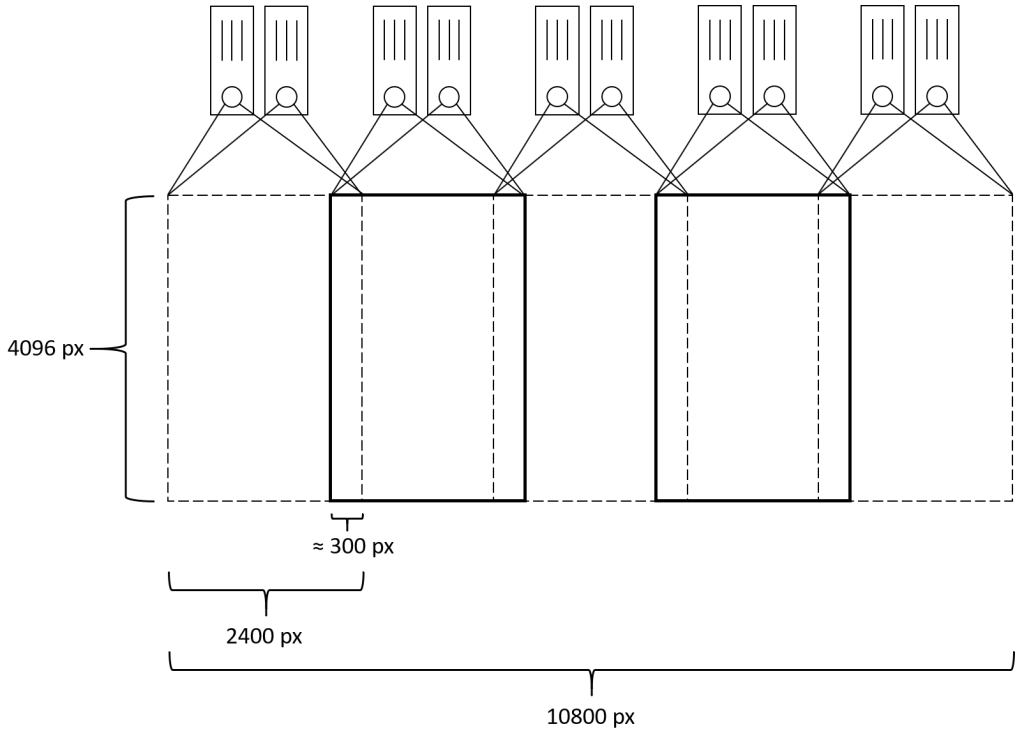


Abbildung 3.11: Schematischer Aufbau der Powerwall-Projektoren. Jeweils zwei vertikal ausgerichtete Projektoren zeigen einen 2400×4096 Pixel großen Bereich des Gesamtbildes an. Durch Überlappen der fünf Bereiche (im Bild durch gestrichelte bzw. dick durchgezogene Rechtecke angezeigt) ergibt sich die Gesamtauflösung von 10800×4096 Pixel. Es entsteht jeweils ein etwa 300 Pixel breiter Streifen, wo sich zwei Bereiche überlappen.

Eine gesonderte Rechnerinfrastruktur, das Grafik-Cluster, ist mit den Projektoren verbunden und stellt deren Input bereit. Zu diesem Grafik-Cluster gehören ein Trackingsystem, welches in Unterabschnitt 3.4.1 näher beschrieben wird, sowie ein Display-Cluster und ein Rendering-Cluster. Allerdings wird das letztere nicht mehr verwendet. Das Display-Cluster besteht aus 20 Display Knoten - zehn pro Auge - die jeweils mit zwei Intel® Xeon® E5-2640 v3 Prozessoren, 256 GB RAM und einer NVIDIA® Quadro® 6000 GPU ausgestattet sind. Jeder Projektor wird von zwei Display-Knoten mit Bilddaten versorgt. Alle Knoten sind über DDR InfiniBand miteinander verbunden, um eine Hochgeschwindigkeitsübertragung auf kurze Distanz mit geringer Latenz zu ermöglichen.

3.4.1 Eye-Tracking

Mit Eye-Tracking bezeichnet man die Blickerfassung und das Aufzeichnen von Blickbewegungen einer Person mithilfe spezieller Geräte. Es besteht die Möglichkeit, die Blickbewegungen mehrerer Betrachter vor der Powerwall zu verfolgen. Dazu wird mithilfe von verschiedenen Geräten, die im Folgenden als *Rigid Body* bezeichnet werden, der Blickpunkt eines Benutzers erfasst. Ein Beispiel für einen Rigid Body ist eine spezielle Brille, an der eindeutig unterscheidbare und Infrarotlicht reflektierende Marker angebracht sind, die eine genaue Bestimmung des Blickpunktes ermöglichen. Der Blickpunkt eines Betrachters lässt sich mathematisch beschreiben als der Schnittpunkt $\vec{s}$ des Vektors $\vec{d_0}$ mit der Powerwall. Dabei wird angenommen, dass $\vec{d_0}$ in Blickrichtung zeigt und normiert ist. Der Schnittpunkt $\vec{s}$ kann über folgende Formel berechnet werden:

$$\vec{s} = (\vec{p}_{cur} + \delta \cdot \vec{d_0}) - \vec{o}_{pw} \tag{3.3}$$

Dabei bezeichnet $\vec{p}_{cur}$ die aktuelle Position des Rigid Bodys, δ die momentane Distanz zur Powerwall und $\vec{o}_{pw}$ den Koordinatenursprung der Powerwall. Die Tracking-Software *Motive* des Herstellers *OptiTrack* sorgt dafür, dass die berechneten Bewegungsdaten über eine Netzwerkverbindung an eine Anwendung zur weiteren Verwendung gestreamt werden. Aus einem gegebenen Blickpunkt lässt sich ein Blickfeld berechnen, was im Folgenden auch als *Foveated Region* bezeichnet wird, da es sich auf den Bereich bezieht, der durch die Fovea im Auge scharf wahrgenommen wird. Das Blickfeld ist definiert durch vier Punkte auf der Powerwall, die sich analog zum Schnittpunkt $\vec{s}$ in Gleichung 3.3 berechnen lassen. Für die Berechnung des Blickfeldes werden der horizontale Öffnungswinkel α sowie der vertikale Öffnungswinkel β, durch die das Blickfeld definiert ist, angegeben. Die vier Schnittpunkte $\vec{s_i}$ mit $i \in \{1, 2, 3, 4\}$, welche das Blickfeld ergeben, lassen sich analog zu Gleichung 3.3 berechnen, wobei die Blickrichtung $\vec{d_0}$ ersetzt wird durch einen Vektor $\vec{d_i}$. Dieser zeigt jeweils zu einem der vier Eckpunkte des Blickfeldes und wird für $i \in \{1, 2, 3, 4\}$ anhand folgender Formel berechnet:

$$\vec{d_i} = \vec{d_0} \pm (\delta_\alpha \cdot \vec{r}) \pm (\delta_\beta \cdot \vec{u}) \tag{3.4}$$

Dabei gibt $\vec{u}$ die momentane Aufwärts-Orientierung („**U**p“) und $\vec{r}$ die momentane Seitwärts-Orientierung („**R**ight“) des Rigid Bodys an. Die Parameter δ_α und δ_β sind die halbierten Ausdehnungen des Blickfeldes in horizontaler beziehungsweise vertikaler Richtung im normierten Abstand entlang der Blickrichtung und ergeben sich über:

$$\begin{aligned} \delta_\alpha &= \tan(\frac{\alpha \cdot \pi}{180}) \\ \delta_\beta &= \tan(\frac{\beta \cdot \pi}{180}) \end{aligned} \tag{3.5}$$

Abbildung 3.12 veranschaulicht die Parameter der Blickfeldberechnung.

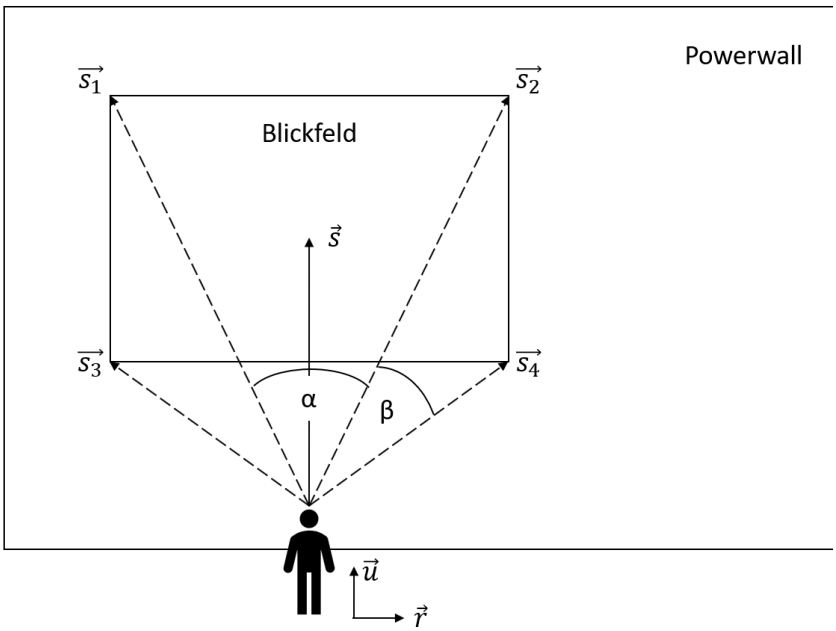


Abbildung 3.12: Ein Betrachter fixiert einen Punkt $\vec{s}$ auf der Powerwall. Das Blickfeld des Betrachters ist definiert über einen horizontalen Öffnungswinkel α und einen vertikalen Öffnungswinkel β. Anhand diesen Parametern sowie mit der Orientierung des Rigid Bodys ($\vec{u}$ und $\vec{r}$) lassen sich die Schnittpunkte $\vec{s_i}$ mit $i \in \{1, 2, 3, 4\}$ bestimmen, welche das Blickfeld ergeben.

4 Konzept

Im Folgenden wird das Konzept des entwickelten Systems erläutert. Es wird beschrieben, welche Vorüberlegungen in die Konzeption mit einfließen und wie das fertige Programm ablaufen soll.

4.1 Vorüberlegungen

Das System soll die Visualisierung von Datensätzen auf der Powerwall ermöglichen. Dazu sollen die Datensätze mittels CPU-basiertem Raytracing gerendert werden, wobei das Blickfeld des Betrachters mit in die Berechnungen einbezogen wird. Es muss berücksichtigt werden, dass die Berechnungen nicht zwangsläufig auf dem Grafik-Cluster der Powerwall durchgeführt werden, da unter Umständen ein entferntes Cluster für das Raytracing eingesetzt wird. Aus diesem Grund muss eine Netzwerkübertragung implementiert werden, die das Versenden von gerenderten Bildern ermöglicht. Die Übertragung von Nachrichten beruht im entwickelten System auf dem User Datagram Protocol (UDP). Im Vergleich zu anderen Netzwerkprotokollen hat UDP den Vorteil, dass es nur geringe Übertragungsverzögerungen gibt. Allerdings arbeitet UDP ungesichert und nicht-zuverlässig, wodurch nicht gewährleistet ist, dass verschickte Nachrichten in der richtigen Reihenfolge geschweige denn überhaupt beim Empfänger eintreffen. Für die Implementierung ist jedoch die Performanz des Übertragungsprotokolls wichtiger als dessen Sicherheit, da es für einen Betrachter kaum wahrnehmbar ist, wenn stellenweise Einzelbilder verloren gehen. Für das Rendern der Bilder sind in einem Cluster mehrere Knoten verantwortlich, die über das Message Passing Interface (MPI) miteinander kommunizieren. MPI bietet für verteilte Computersysteme die Möglichkeit, bei parallelen Berechnungen Nachrichten auszutauschen. Sobald ein Bild fertig gerendert wurde, muss es an das Display-Cluster der Powerwall übertragen werden, um dort angezeigt werden zu können. Da die Übertragung von unkomprimierten Bildern einen zu hohen Datenaufwand zur Folge hätte, müssen die Bilder vor dem Verschicken encodiert werden, um die benötigte Bandbreite möglichst gering zu halten. Daraus resultiert, dass im Display-Cluster der Powerwall eine geeignete Möglichkeit zur Decodierung der eintreffenden Bilder vorhanden sein muss. Abbildung 4.1 zeigt vereinfacht den Ablauf des Programms.

Abbildung 4.1: Vereinfachter Ablauf des Programms. Zunächst wird das Blickfeld eines Betrachters erfasst, um es dann bei den Raytracing Berechnungen zu berücksichtigen. Gerenderte Bilder werden encodiert, bevor sie über ein Netzwerk übertragen werden und mittels geeigneter Software decodiert und angezeigt werden können.

4.2 Programmablauf

Das System wird für einen oder mehrere Betrachter vor der Powerwall, deren Blickbewegungen mittels Eye-Tracking aufgezeichnet werden, entwickelt. Das Trackingsystem erfasst, wie in Abschnitt 3.4.1 beschrieben, den Fixationspunkt des Betrachters. Über eine Schnittstelle, die mit dem Trackingsystem interagiert, wird daraus das Blickfeld, die *Foveated Region*, berechnet. Die Informationen über das Blickfeld des Benutzers werden über eine Netzwerkverbindung an das Cluster übertragen, wo alle nötigen Berechnungen zum Rendern durchgeführt werden. Die Rechnerinfrastruktur im Cluster nutzt die erhaltene Foveated Region für das Raytracing und erzeugt ein Bild, das im Bereich der Foveated Region hochaufgelöst und außerhalb (Peripherie) niedrig aufgelöst ist. Die einzelnen Knoten des Clusters rendern dabei jeweils nur einen Teil des Bildes und kommunizieren über MPI. Das fertig gerenderte Bild wird, erneut über eine Netzwerkverbindung, an die Rechnerinfrastruktur der Powerwall zurückgeschickt. Um die benötigte Bandbreite möglichst gering zu halten, wird das Bild zunächst codiert. Sobald ein codierter Frame bei den Powerwall-Rechnern eintrifft, wird dieser durch geeignete Funktionen decodiert. Die decodierten Frames können schließlich auf der Powerwall angezeigt werden. Abbildung 4.2 zeigt die genannten Schritte in einem Ablaufdiagramm.

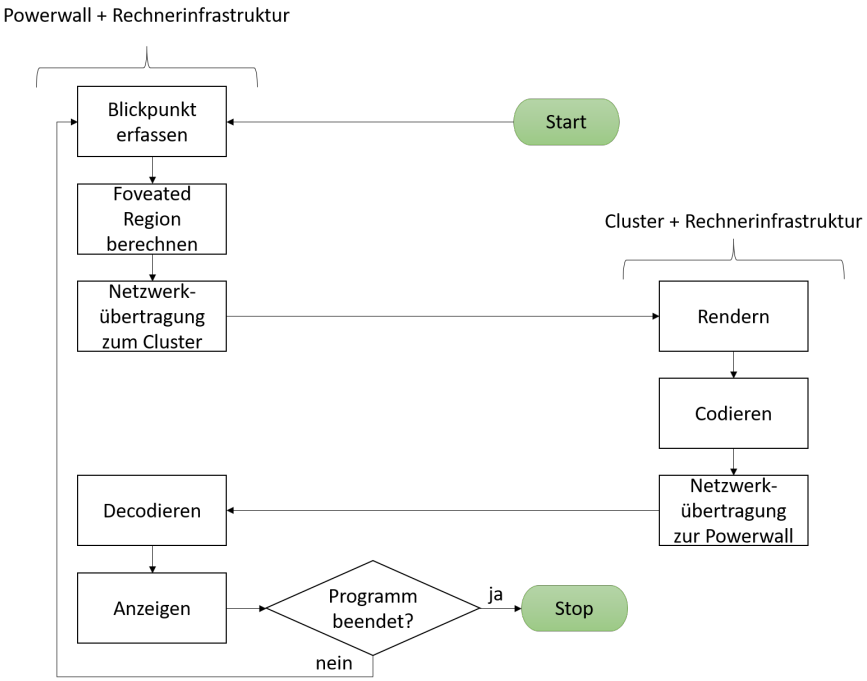


Abbildung 4.2: Ablaufdiagramm des Programms. Zunächst wird der Blickpunkt eines Betrachters vor der Powerwall erfasst, mithilfe dessen die Foveated Region berechnet wird. Diese wird von der Rechnerinfrastruktur der Powerwall über eine Netzwerkverbindung an das Cluster übertragen, wo sie das Rendern beeinflusst. Gerenderte Daten werden codiert und anschließend an die Powerwall zurückgeschickt, wo sie decodiert und angezeigt werden können. Im fertigen System laufen die drei Schritte zur Blickpunkterfassung parallel zu den zwei Schritten zur Decodierung und Anzeige auf der Rechnerinfrastruktur der Powerwall ab.

5 Implementierung

In diesem Abschnitt wird auf die Implementierung des Systems näher eingegangen. Wie es mittlerweile bei den meisten Implementierungen üblich ist, wird auch dieses System nicht von Grund auf neu entwickelt, sondern verwendet bestehende Software-Lösungen und erweitert diese zum Teil. Im Folgenden werden zunächst die verwendeten Frameworks und deren Funktionsumfang beschrieben. Anschließend wird der Programmablauf erklärt, welcher sich grob in die folgenden Schritte untergliedern lässt:

- Blickerfassung
- Rendern
- Codierung
- Netzwerkübertragung
- Decodierung und Anzeige

5.1 Verwendete Frameworks

Für die einzelnen Schritte des Programmablaufs werden unterschiedliche Frameworks und Software-Bibliotheken genutzt, die bereits verschiedene vorgefertigte Funktionen enthalten. Im Folgenden wird beschrieben, welche bestehenden Software-Lösungen für die Implementierung verwendet und erweitert werden.

5.1.1 MegaMol

MegaMol [GBB+19] ist ein System zur interaktiven Visualisierung von großen wissenschaftlichen Datensätzen. Ursprünglich wurde es für die Visualisierung und Analyse partikelbasierter Datensätze, die zum Beispiel aus Molekulardynamik-Simulationen stammen, entwickelt. Durch die Weiterentwicklung bietet MegaMol inzwischen größere Flexibilität an und kann daher auch für andere Arten der Visualisierung genutzt werden.

MegaMol funktioniert mit Modulen und Aufrufen (*engl. Modules and Calls*), welche sich zu einem azyklischen Modulgraphen kombinieren lassen. In Abbildung 5.1 ist beispielhaft ein Modulgraph dargestellt, welcher gleichzeitig als Basis für das entwickelte System dient. Ein Modul, welches für die Netzwerkkommunikation verantwortlich ist, fordert über einen Call codierte Bilddaten von einem anderen Modul, dem Encoder, an. Dieser fordert wiederum gerenderte Bilder über einen weiteren Call vom Encoder an.

Abbildung 5.1: Beispiel eines MegaMol Modulgraphen. Das Modul *UdpClient* fordert über einen Call encodierte Daten vom Modul *Encoder* an. Dieses fordert wiederum gerenderte Daten über einen Call vom *Renderer* an.

Der Funktionsumfang eines Moduls wird durch den Entwickler festgelegt und kann sich von kleineren Teilaufgaben bis hin zu einem gesamten Rendering-Vorgang erstrecken. Im Normalfall wird die Funktion eines Moduls möglichst feingranular gehalten, um die Wiederverwendbarkeit dieses Moduls in einem anderen Kontext zu gewährleisten. Mithilfe von Calls können Module Daten von anderen Modulen anfordern. Ein Aufruf geht dabei immer von der Senke, die Daten anfordert, zur Datenquelle, welche die Daten bereitstellt. Im Modulgraph bedeutet das, dass ein Call stets von einem Modul an ein weiter rechts stehendes Modul ausgehen muss. Die Kanten des Graphen stellen daher einen Daten- oder Informationstransport dar.

Durch die große Flexibilität kann MegaMol auch zum Rendern, Codieren und Netzwerk-basierten Verschicken von Datensätzen beziehungsweise Bildinformationen eingesetzt werden. Aus diesem Grund dient es als Software-Basis für die Implementierung. Alle Raytracing Berechnungen werden von OSPRay durchgeführt, was bereits in MegaMol integriert ist und die Möglichkeit bietet, CPU-basiertes Raytracing durchzuführen.

5.1.2 OSPRay

OSPRay [WJA+17] ist ein Framework, welches für CPU-basiertes Raytracing optimiert wurde. Es wird im Zuge der Implementierung erweitert, sodass es auch *Foveated Rendering* unterstützt, indem das Blickfeld eines Betrachters in die Raytracing Berechnungen miteinbezogen wird.

5.1.3 FFmpeg

FFmpeg ist ein freies Software-Projekt zur Verarbeitung von Video-, Audio- und anderen Multimediadateien. Zum Funktionsumfang von FFmpeg gehören neben einigen Kommandozeilen-Tools auch mehrere Software-Bibliotheken, die zum Beispiel das Codieren und Decodieren von Videodateien ermöglichen [Kor12].

5.1.4 x264 und x265

x264 ist eine Software-Bibliothek, die das Codieren von Videos in den H.264/MPEG-4 AVC Standard ermöglicht. *x265* ist der Nachfolger der x264-Bibliothek und bietet Funktionen zum Codieren von Videos in den neueren H.265-Standard an. Beide Bibliotheken wurden von VideoLAN entwickelt und sind frei verfügbar.

5.1.5 VV3

VV3 [FME20] ist eine Software, welche Remote-Visualisierung auf großen hochauflösenden Displays ermöglicht. Neben der Anzeige von Bildern auf der Powerwall bietet VV3 auch Funktionen zur Decodierung von Bildern an.

5.2 Blickerfassung

Das Blickfeld eines Betrachters wird wie in Unterabschnitt 3.4.1 beschrieben berechnet. Die ermittelten Werte werden an die Rechnerinfrastruktur übertragen, welche für das Rendern verantwortlich ist.

5.3 Rendern

Für das Rendern ist ein eigenes MegaMol-Modul, der Renderer, verantwortlich. Dieser erhält als Eingabe Rohdaten eines Modells, dessen visuelle Darstellung berechnet werden soll. Dafür wird Raytracing eingesetzt, wobei alle nötigen Schnittpunktberechnungen von OSPRay durchgeführt werden. Allerdings wäre vollständiges Raytracing bei einer Auflösung von 10800×4096 Pixeln (entspricht ca. 44 Millionen Pixeln) zu aufwendig. Stattdessen wird das Rendern aufgeteilt in mehrere Schritte.

Zunächst wird eine niedrig aufgelöste Version des Modells mittels Raytracing gerendert, welche im Folgenden als Support-Image bezeichnet wird. Dies wird dadurch erreicht, dass für den Raytracing-Algorithmus ein sehr grobes Raster zu Grunde gelegt wird. Im Vergleich zum finalen Bild (10800×4096 Pixel) hat das Support-Image in x- und y-Richtung nur jeweils ein Achtel der Pixel (1350×512). Dadurch verringert sich die Anzahl der für das Raytracing ausgesendeten Primärstrahlen um ein 64-faches.

Im nächsten Schritt wird ein kleiner Bereich des Bildes, die Foveated Region, gerendert. Um dem Betrachter den Eindruck zu vermitteln, er sähe ein vollständig hochaufgelöstes Bild, genügt es, den Bereich, auf den sich der Betrachter fixiert, scharf darzustellen. Das zuvor bestimmte Blickfeld des Betrachters wird in die Berechnung miteinbezogen. Innerhalb dieses Bereiches wird jedem Pixel über die Funktion

$$p = (1 - d)^3, \text{ mit } d \in [0, 1]$$

ein Wert p zugeordnet, der angibt, wie hoch die Wahrscheinlichkeit ist, dass die Farbe dieses Pixels mittels Raytracing bestimmt wird. Dabei steht d für die normalisierte Distanz des Pixels zum Mittelpunkt der Foveated Region. Es wird eine Hyperbelfunktion gewählt, da sie die in Abschnitt 3.1 beschriebene Verteilung der Zapfen auf der Netzhaut annähert. Je größer der Abstand eines Pixels zum Fixierpunkt des Betrachters ist, desto geringer ist die Wahrscheinlichkeit, dass für dieses Pixel ein Primärstrahl erzeugt wird. Außerhalb der Foveated Region wird eine minimale Wahrscheinlichkeit von p_{min} für alle Pixel angenommen.

Für jedes Pixel wird jeweils anhand einer zufälligen Zahl zwischen 0 und 1 entschieden, ob für dieses Pixel ein Schnitttest für den Raytracing-Algorithmus durchgeführt wird oder nicht. Ist die dem Pixel zugeordnete Wahrscheinlichkeit p größer als die zufällig generierte Zahl, so wird dieses Pixel gerendert, andernfalls findet kein Schnitttest statt und das Pixel erhält zunächst eine festgelegte Hintergrundfarbe. Alle Pixel, die auf diese Weise gerendert wurden, können direkt in den finalen Framebuffer übernommen werden. Der Framebuffer ist ein spezieller Speicher, welcher einer digitalen Kopie des Monitorbildes entspricht. Um eine Unterscheidung zwischen gerenderten und nicht gerenderten Pixeln vornehmen zu können, wird jedem gerenderten Pixel ein Alpha-Wert $\alpha > 0$ zugeordnet, während nicht gerenderte Pixel einen Alpha-Wert $\alpha = 0$ haben. In Abbildung 5.2 ist sehr gut erkennbar, wie sich die Wahrscheinlichkeit, dass ein Pixel gerendert wird, anhand der gewählten Hyperbelfunktion verteilt. Für weiße Pixel wurde die Farbe nicht mittels Raytracing bestimmt und muss aus dem Support-Image rekonstruiert werden. Je größer der Abstand zum Blickpunkt ist, desto geringer wird die Wahrscheinlichkeit, dass ein Pixel gerendert wird und desto dichter werden weiße Bereiche außerhalb des Blickfeldes.

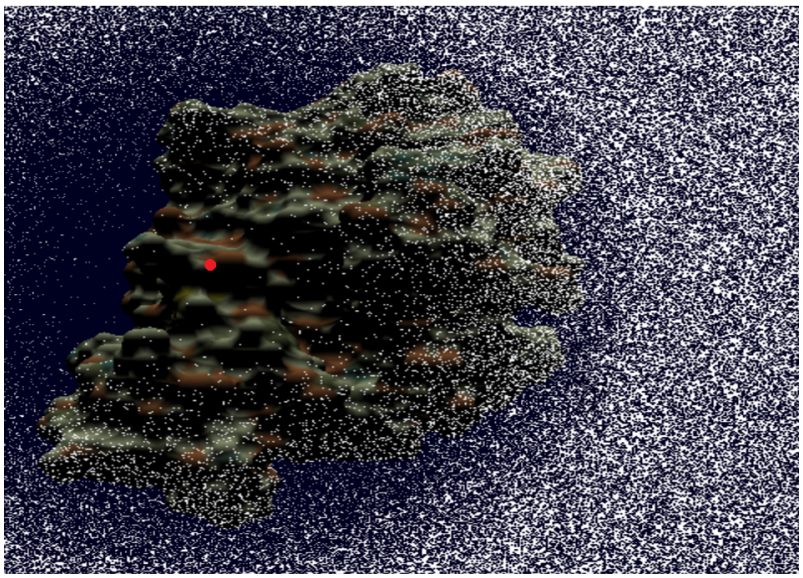
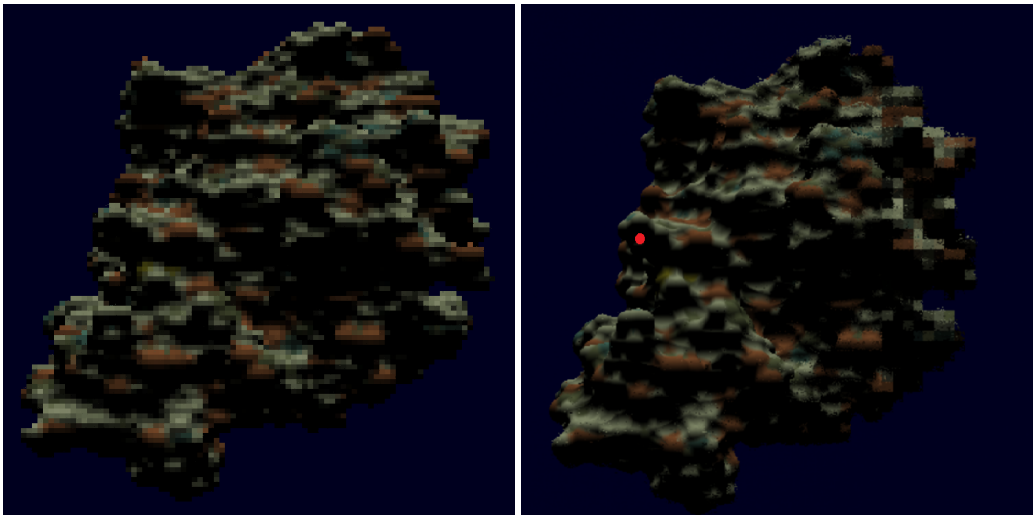


Abbildung 5.2: Wahrscheinlichkeit, dass die Farbe eines Pixels beim Foveated Rendering mittels Raytracing bestimmt wird. Weiße Pixel bedeuten, dass dort kein Primärstrahl erzeugt wurde und die Farbe aus dem Support-Image rekonstruiert werden muss. Weiße Bereiche werden mit zunehmendem Abstand zum Blickpunkt (rot) dichter.

Zuletzt muss für jedes Pixel, welches im vorherigen Schritt nicht berücksichtigt wurde, die Farbe bestimmt werden. Dies geschieht über das zuvor generierte Support-Image, welches beispielhaft in Abbildung 5.3a zu sehen ist. Die Bestimmung, ob ein Pixel seine finale Farbe bereits erhalten hat, geschieht anhand des Alpha-Wertes. Ist dieser 0, so wurde die Farbe dieses Pixels bisher nicht berechnet. In diesem Fall wird die Pixel-Farbe aus dem korrespondierenden Pixel des Support-



(a) Support-Image **(b)** Gerendertes Bild

Abbildung 5.3: (a) Ein Support-Image mit deutlich geringerer Auflösung als das finale Bild. (b) Ein fertig gerendertes Bild, bei dem die Foveated Region um den Blickpunkt (rot) herum scharf dargestellt wird. Die Farbinformationen außerhalb der Foveated Region werden zu einem großen Teil aus dem Support-Image rekonstruiert und sind deshalb unscharf.

Images übernommen. Jedes Pixel P des finalen Bildes kann mit Hilfe von Koordinaten in der Form $P = (x, y)$ repräsentiert werden. Das zugehörige Pixel $S(P)$ des Support-Images erhält man über folgende Gleichung:

$$S(P) = \left(\left\lfloor \frac{x}{8} \right\rfloor, \left\lfloor \frac{y}{8} \right\rfloor\right) \tag{5.1}$$

Der Farbwert, welcher im Support-Image für $S(P)$ eingetragen ist, wird für das Pixel des finalen Bildes übernommen. Schließlich erhält man ein Bild ähnlich dem in Abbildung 5.3b, wobei für jedes Pixel RGBA-Werte im Speicher abgelegt sind.

Ändert sich im Vergleich zum vorangegangen Frame weder die Blickrichtung des Benutzers noch der betrachtete Bildausschnitt, so wird die Qualität des Bildes immer weiter erhöht, indem mehr Primärstrahlen für den Raytracing-Algorithmus erzeugt werden. Dafür wird der Parameter p_{min}, welcher die Wahrscheinlichkeit angibt, dass die Farbe eines Pixels mittels Raytracing bestimmt wird, schrittweise erhöht. Sobald $p_{min} \geq 1$ gilt, wird für alle Pixel ein Primärstrahl beim Raytracing erzeugt und anhand dessen die Farbe bestimmt. Das daraus resultierende Bild hat für die gesamte Auflösung die höchstmögliche Qualität. Wenn dieser Zeitpunkt erreicht ist und keine Änderungen auftreten, wird nicht mehr weiter gerendert, sondern lediglich der zuletzt gerenderte Frame erneut verwendet, da er sich ohnehin nicht von einem neu gerenderten Frame unterscheiden würde. Erst bei einer Änderung des Blickpunktes oder des betrachteten Bildausschnitts wird p_{min} auf seinen ursprünglichen Wert zurückgesetzt und der Renderer beginnt wieder mit der Berechnung neuer Bilder. Diese Art des Renderns wird auch als *Progressive Rendering* bezeichnet.

5.4 Codierung

Die Codierung wird, ähnlich wie das Rendern, von einem separaten MegaMol-Modul durchgeführt. Dieses Modul erhält vom Renderer die Größe sowie den Startwert eines zuvor berechneten Frames. Mit dem Startwert ist hier eine Speicheradresse gemeint, ab welcher fortlaufend die jeweiligen Farbwerte der einzelnen Pixel im Speicher stehen. Jedes Pixel wird durch 32 Bit = 4 Byte repräsentiert, wobei jeweils 8 Bit = 1 Byte auf einen der vier Farbkanäle (RGBA) fallen. Abbildung 5.4 veranschaulicht diesen Sachverhalt.

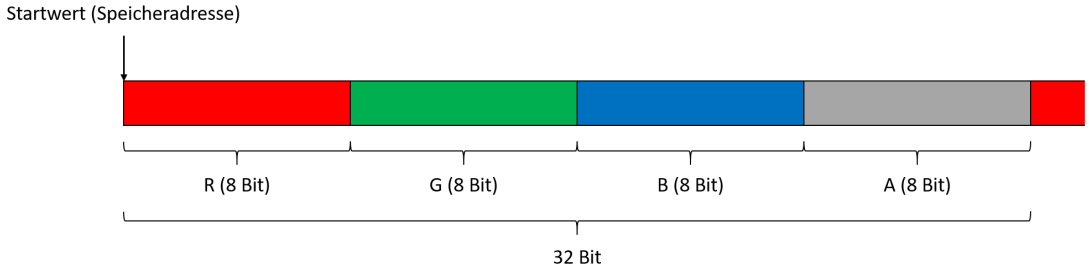


Abbildung 5.4: Ein Pixel wird durch vier Farbkanäle (RGBA) mit jeweils 8 Bit repräsentiert. Der Startwert eines Frames bezieht sich auf die Speicheradresse des ersten Pixels dieses Frames.

Insgesamt benötigt ein Bild der Größe 10800×4096 Pixel demnach

$$10800 \times 4096 \times 4 \text{ Byte} = 176947200 \text{ Byte} \approx 177 \text{ MB}$$

Speicherplatz. Für eine Echtzeit-Anwendung ergibt sich daraus eine benötigte Bandbreite von über 5 GB/s. Durch eine geeignete Codierung der Farbinformationen kann der Speicherbedarf deutlich reduziert werden. Die Codierung besteht aus zwei Schritten.

Zunächst werden die vorhandenen RGBA-Farbwerte in das YUV-Farbmodell konvertiert. Durch Farbunterabtastung mit einem Verhältnis 4:2:0 wird der Speicherbedarf bereits um etwa die Hälfte reduziert. Die Software-Bibliothek *Libswscale*, welche zum Umfang von FFmpeg gehört, bietet Funktionen zur Konvertierung zwischen verschiedenen Farbräumen an.

Die konvertierten Farbinformationen werden im zweiten Schritt durch geeignete Funktionen codiert. Dabei kommen die in Abschnitt 3.3 beschriebenen Methoden der Reduktion zum Einsatz. Sowohl die von FFmpeg bereitgestellte Bibliothek *Libavutil* als auch die freien Software-Bibliotheken *x264* und *x265* stellen vorgefertigte Funktionen zur Verfügung, die eine solche Codierung ermöglichen. Insgesamt wird der Speicherbedarf durch die Codierung so weit verringert, dass eine Netzwerkübertragung der Bilder problemlos möglich ist.

5.5 Ringpuffer

Sowohl für gerenderte Frames als auch für encodierte Frames wird zur Speicherung jeweils ein sogenannter Ringpuffer verwendet. Dieser ist eine Datenstruktur mit einer festen Anzahl an Speicherplätzen und einem Lese- sowie einem Schreibindex. Bei der Erzeugung eines Ringpuffers

werden Lese- und Schreibindex mit dem Wert 0 initialisiert und zeigen somit auf den ersten freien Speicherplatz des Puffers. Über einen Schreibzugriff kann dem Puffer ein Element hinzugefügt werden. Der Speicherplatz, an den das Element geschrieben wird, ist dabei durch den Schreibindex festgelegt. Nach einem erfolgreichen Schreibzugriff wird der Schreibindex um 1 inkrementiert oder auf 0 zurückgesetzt, wenn an den letzten Speicherplatz des Puffers geschrieben wurde. Sollte der Schreibindex bei einem Schreibzugriff auf einen belegten Speicherplatz zeigen, so kann dieser entweder überschrieben werden oder der Schreibzugriff wird abgebrochen und die zu schreibenden Informationen werden verworfen. Bei einem Lesezugriff wird ein Element aus dem Ringpuffer entnommen, sofern dieser nicht leer ist. Der Ringpuffer ist genau dann leer, wenn Lese- und Schreibindex identisch sind. Andernfalls befindet sich mindestens ein Element im Puffer. Im Anschluss an einen erfolgreichen Lesezugriff wird der Leseindex analog zum Schreibindex um 1 erhöht.

Im entwickelten System schreibt das Renderer-Modul gerenderte Frames in einen Ringpuffer. Aus dem selben Ringpuffer liest der Encoder die gerenderten Frames und codiert diese anschließend. Nach der Codierung schreibt das Encoder-Modul die codierten Bildinformationen in einen anderen Ringpuffer, aus welchem sie von einem weiteren Modul gelesen werden, um sie schließlich über eine Netzwerkverbindung übertragen zu können. Durch die Verwendung von Ringpuffern können mehrere Module unabhängig voneinander Operationen durchführen, was unter anderem die parallele Ausführung von Rendern und Encodieren ermöglicht. In Abbildung 5.5 ist dieses Prinzip beispielhaft dargestellt.

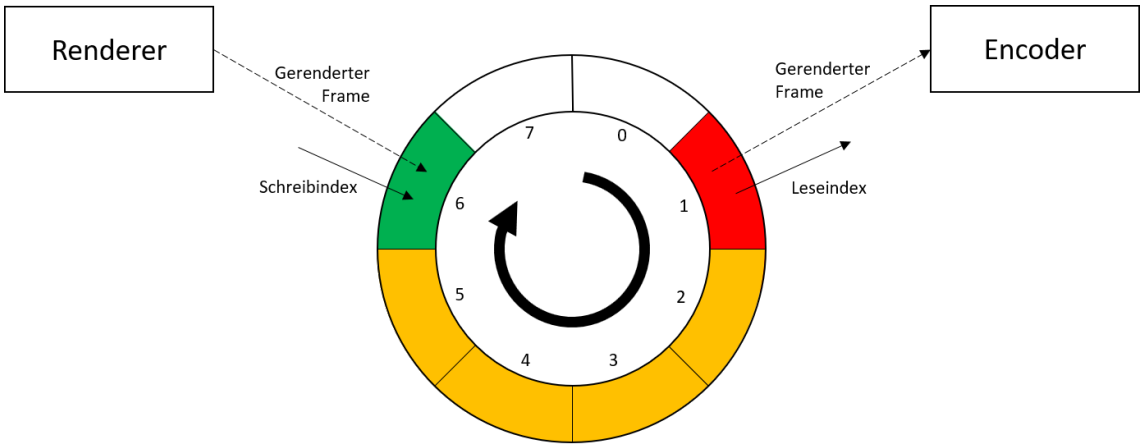


Abbildung 5.5: Prinzip des Ringpuffers. Der Renderer schreibt einen neu gerenderten Frame (grün) in den Ringpuffer. Der Schreibindex gibt dabei vor, an welche Stelle geschrieben wird. Parallel dazu entnimmt der Encoder das durch den Leseindex festgelegte Element (rot), welches sich zugleich am längsten im Puffer befindet. Nach erfolgreichen Zugriffen wird der jeweilige Index um 1 erhöht. Weiße Elemente stehen für freie, gelbe für bereits belegte Speicherplätze.

Besonders bei Modulen mit unterschiedlichen Ausführungsgeschwindigkeiten bietet es sich an, Ringpuffer zu verwenden. Dadurch wird verhindert, dass ein Modul auf den Ausführungsabschluss eines anderen Moduls warten muss, um mit der nächsten Operation fortfahren zu können. Sollte ein Ringpuffer bei einem Schreibzugriff voll sein, so wird der Zugriff abgebrochen und die Bildinformationen werden verworfen. Die Anzahl an Speicherplätzen in einem Ringpuffer wird so groß gewählt, dass möglichst selten Informationen verworfen werden, aber dennoch kaum negative Auswirkungen auf die Performanz des Systems entstehen.

5.6 Netzwerkübertragung

Die gerenderten und encodierten Bilder werden über eine Netzwerkverbindung an das Display-Cluster der Powerwall gestreamt. Für die Netzwerkübertragung werden verschiedene Nachrichtentypen benötigt, die sich in Größe und Funktion unterscheiden. Die Übertragung wird mit UDP realisiert. Deshalb muss berücksichtigt werden, dass die maximale Größe einer UDP-Nachricht etwa 64 kB beträgt. Wenn ein codiertes Einzelbild diese Größe überschreitet, kann es nicht mit einer einzelnen Nachricht verschickt werden. In diesem Fall wird das Bild in mehrere *Slices* (dt. Scheiben) unterteilt, die jeweils 56000 Byte groß sind. Das Bild wird dann in Form von mehreren Teilnachrichten verschickt, wobei für jede Teilnachricht sichergestellt ist, dass sie die maximale Nachrichtengröße nicht überschreitet. Im häufigsten Fall treffen beim Display-Cluster Nachrichten ein, die Informationen über einen einzelnen Frame beinhalten. Dazu gehören neben der Größe des Frames und den konkreten, encodierten Farbwerten auch eine Frame-Nummer, ein Zeitstempel (Timestamp) und eine Sequenz-Nummer. Die beiden letzteren sind notwendig, um sicherzustellen, dass die einzelnen Bilder in der richtigen Reihenfolge angezeigt werden. Außerdem enthält jede Frame-Nachricht eine Stream-ID, um sie dem richtigen Stream zuordnen zu können. Für den Fall, dass ein Bild nicht mit einer einzelnen Nachricht, sondern über mehrere Teilnachrichten verschickt wurde, dient die Frame-Nummer dazu, die einzelnen Teile des Bildes dem richtigen Gesamtbild zuzuordnen. Dann wird zusätzlich eine Slice-Nummer, die Anzahl der Slices und ein Offset in dieser Nachricht mitgesendet, um die codierten Teilbilder richtig zusammensetzen und decodieren zu können. In Abbildung 5.6 ist der Aufbau einer solchen Frame-Nachricht skizziert.

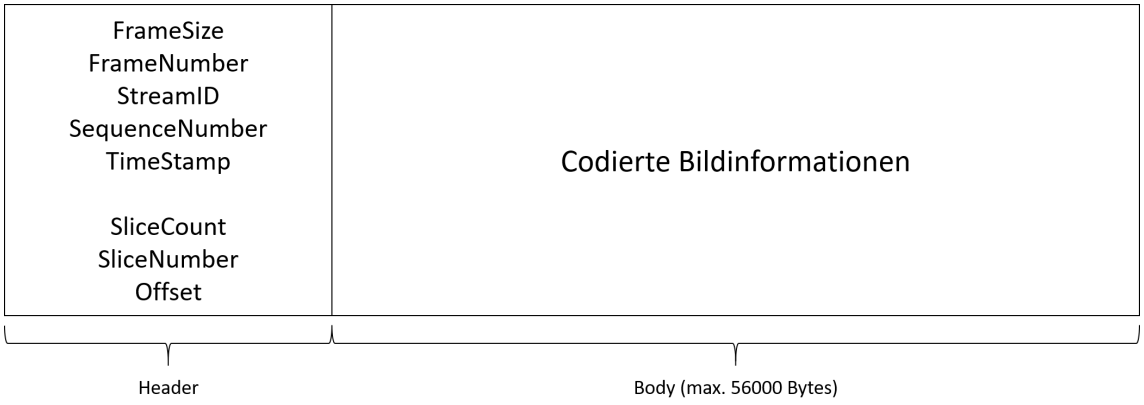


Abbildung 5.6: Schematischer Aufbau einer Frame-Nachricht. Im Header stehen für die Übertragung und Decodierung notwendige Informationen. Die codierten Bildinformationen stehen in einem separaten Teil der Nachricht und sind maximal 56000 Bytes groß.

Weitere Nachrichtentypen sind Stream-Beschreibungs-Nachrichten (Stream-Deskriptor) sowie KeepAlive-Nachrichten. Eine Stream-Beschreibungs-Nachricht muss beim Starten des Systems vom Client (Cluster, auf dem gerendert und codiert wird) verschickt werden, um dem Server (Display-Cluster der Powerwall) mitzuteilen, in welcher Form künftige Nachrichten eintreffen werden. Diese Nachricht enthält den Namen sowie die ID des entsprechenden Streams. Außerdem beinhaltet sie Informationen über das Videoformat, welches für die Codierung eingesetzt wurde. Um fehlerfrei decodieren zu können, muss der Decoder dasselbe Videoformat benutzen wie der Encoder. Eine KeepAlive-Nachricht teilt dem Empfänger mit, dass seit der letzten Nachricht keine Veränderungen aufgetreten sind. Diese Art von Nachrichten sind wichtig, um dem Server zu signalisieren, dass

die Verbindung noch aktiv ist. Des Weiteren wird dadurch der Datendurchsatz gesenkt, da ein unveränderter Frame nicht erneut gesendet werden muss. Es besteht auch die Möglichkeit, auf das Senden von Bereichen des Bildes, in denen sich nichts mehr ändert, zu verzichten.

5.7 MPI-Kommunikation

Das Rendern sowie das anschließende Codieren und Verschicken der Daten kann sowohl von einem einzelnen Rechner als auch von mehreren Knoten in einem Cluster durchgeführt werden. Wird ein Cluster verwendet, so rendert jeder Knoten nur einen Teil des Bildes und codiert diesen. Damit die Knoten untereinander kommunizieren können, wird MPI eingesetzt. Es muss für jeden Knoten bekannt sein, welchen Bildausschnitt er rendern und codieren muss. Zudem müssen alle Knoten das aktuelle Blickfeld des Benutzers kennen, um den Rendervorgang daran anpassen zu können. Diese Informationen erhalten alle Knoten von einem speziellen Knoten, der als *MPI Boss* deklariert wird. Der MPI Boss verarbeitet alle Nutzereingaben und gibt diese an die anderen Knoten weiter, wenn sich beispielsweise der Bildausschnitt oder das Blickfeld des Benutzers ändert. Jeder Knoten inklusive des MPI Bosses rendert und codiert anschließend einen Teil des Bildes. Für die Netzwerkübertragung gibt es zwei verschiedene Varianten, die in Abbildung 5.7 und Abbildung 5.8 schematisch dargestellt sind.

In Variante 1 sendet jeder Knoten die codierten Bildinformationen selbst an das Zielsystem. Auf dem Zielsystem läuft die Software VV3, welche die ankommenden Teilbilder zu einem Gesamtbild zusammenfügt, decodiert und sie anschließend anzeigt.

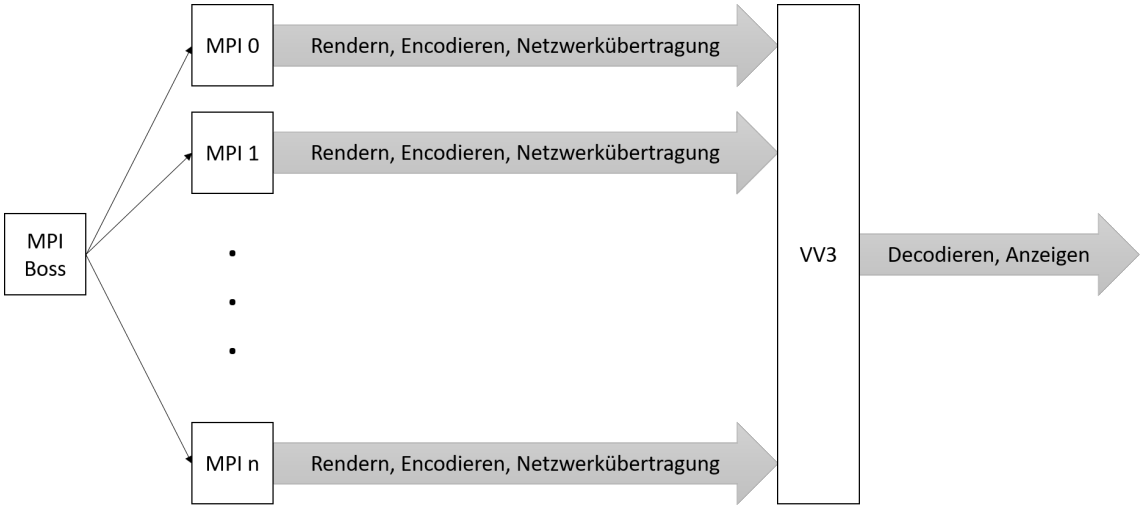


Abbildung 5.7: MPI-Variante 1: Jeder Knoten rendert, encodiert und verschickt einen Teil des Bildes an das Zielsystem.

Bei Variante 2 ist der MPI Boss für die Übertragung aller Teilbilder an das Zielsystem verantwortlich. Dafür müssen zunächst alle Knoten ihre codierten Teilbilder über MPI an den MPI Boss senden. Dieser verschickt diese anschließend über UDP weiter an das Zielsystem, wo sie zu einem Gesamtbild zusammengesetzt, decodiert und angezeigt werden. Diese Variante hat den Vorteil, dass nur ein Knoten des Clusters eine Netzwerkschnittstelle anbieten muss, mithilfe derer eine Übertragung an einen Remote-Standort möglich ist.

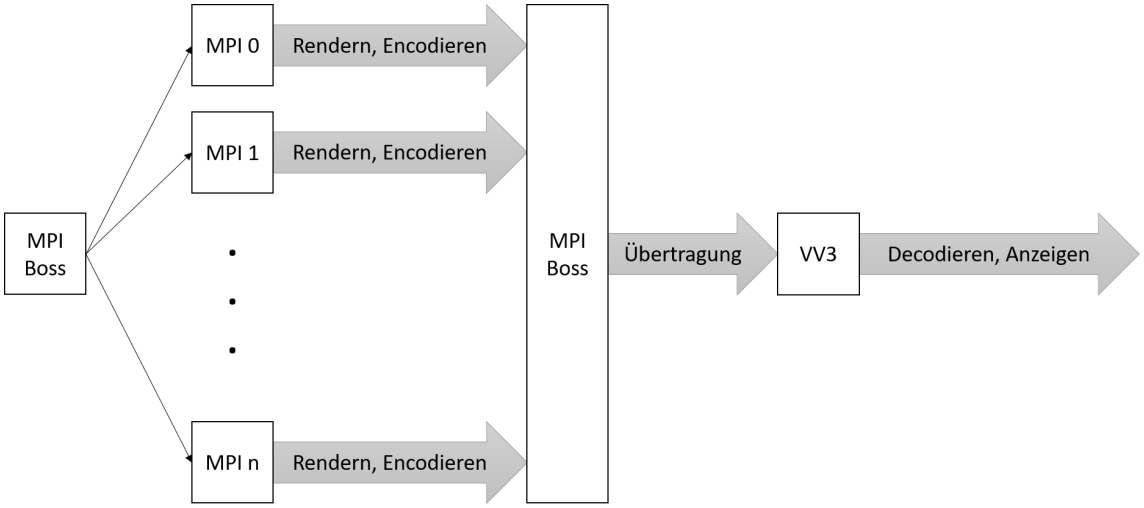


Abbildung 5.8: MPI-Variante 2: Jeder Knoten rendert und encodiert einen Teil des Bildes und sendet die codierten Bildinformationen an den MPI Boss. Dieser verschickt alle Teilbilder an das Zielsystem.

Bei der Implementierung wurde zunächst die zweite Variante eingesetzt, was jedoch nicht so gut funktionierte wie vermutet. Aus diesem Grund wurde zusätzlich die erste Variante implementiert, welche deutlich performanter ist als die zweite und deshalb beim Testen des Systems verwendet wird.

5.8 Decodierung und Anzeige

Die über das Netzwerk verschickten Nachrichten werden von der Software VV3 verarbeitet. Sobald eine Nachricht bei VV3 eintrifft, wird anhand des Nachrichtentyps entschieden, wie die weitere Verarbeitung dieser Nachricht abläuft.

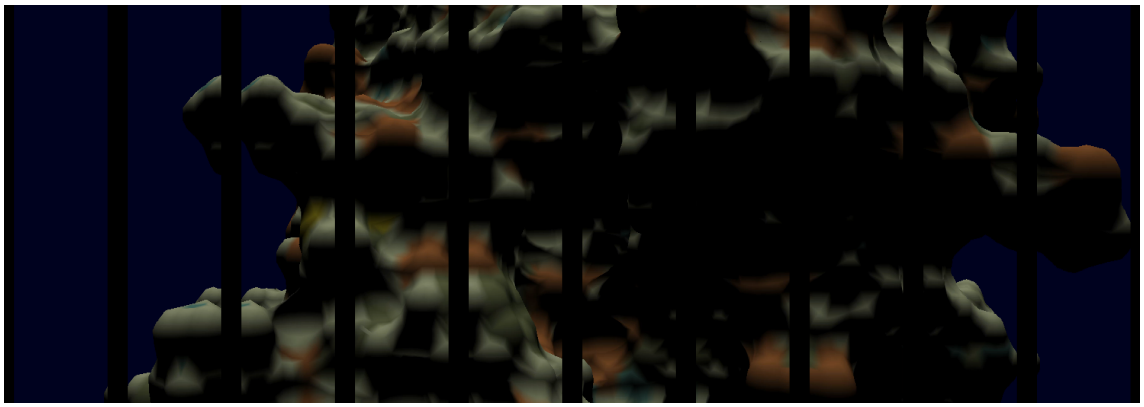
Zunächst erhält VV3 beim Start des Systems eine Stream-Deskriptor-Nachricht, die Auskunft über verschiedene Informationen wie Bildgröße und Videoformat gibt. Alle für das Decodieren notwendigen Datenstrukturen werden anhand der Werte im Deskriptor initialisiert. Nach der Initialisierung wartet VV3 auf eingehende Frame-Nachrichten, die codierte Bildinformationen enthalten.

Da eine Übertragung über UDP keine Garantie dafür gibt, dass die Nachrichten in der richtigen Reihenfolge eintreffen, werden eingehende Nachrichten zunächst anhand ihres Zeitstempels sortiert. Wurde ein Frame mit einer einzelnen Nachricht verschickt, so kann dieser anschließend direkt decodiert werden. Wenn ein Frame über mehrere Teilnachrichten verschickt wurde, so muss zunächst gewartet werden, bis alle Teile angekommen sind. Die einzelnen Teile werden anhand ihrer Slice-Nummer und ihres Offsets zu einem Gesamtbild zusammengefügt, welches anschließend decodiert werden kann.

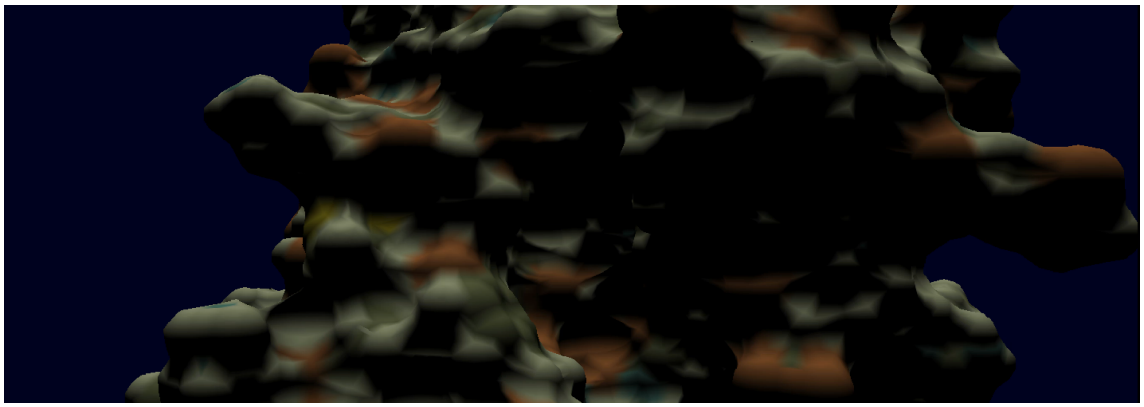
Für die Decodierung werden Funktionen der *Libavutil*-Bibliothek von FFmpeg verwendet. Im Anschluss an die Decodierung erfolgt eine Farbraum-Konversion. Die decodierten Farbwerte werden, wieder mithilfe von *Libswscale*, aus dem YUV-Farbformat in das ähnliche NV12-Format konvertiert.

VV3 ermöglicht es, Bilder im NV12-Format auf der Powerwall anzuzeigen. Dazu müssen diese abschließend in eine spezielle Datenstruktur kopiert werden, ehe ein Ausgabebild auf der Powerwall erscheint.

In Abbildung 5.9 ist ein Bild zu sehen, welches sich aus mehreren Teilbildern zusammensetzt und auf der Powerwall angezeigt werden kann. Dieses Bild wurde über die gesamte Bildgröße mit der höchsten Auflösung gerendert.



(a) Gerenderte Teilbilder



(b) Vollständiges Bild auf der Powerwall

Abbildung 5.9: (a) Wenn der Rendervorgang auf einem Cluster durchgeführt wird, entstehen mehrere Teilbilder, die zu einem Gesamtbild zusammengesetzt werden müssen. An den Rändern werden Bereiche zum Teil doppelt gerendert, um die einzelnen Teilbilder zu einem glatten Gesamtbild zusammenfügen zu können. (b) Durch Zusammenfügen der einzelnen Teilbilder entsteht dieses Bild auf der Powerwall.

6 Ergebnisse

Das entwickelte System wurde mit verschiedenen Einstellungen getestet. Da eine Berechnung auf einem externen Cluster nicht möglich war, wurde das Display-Cluster der Powerwall in zwei Parts aufgeteilt. Zehn Knoten wurden für das Rendern, Encodieren und die Netzwerkübertragung benutzt, während die anderen zehn Knoten für das Decodieren und Anzeigen der Bilder verantwortlich waren. Abbildung 6.1 zeigt das Grafik-Cluster der Powerwall, welches teilweise zum Testen des entwickelten Systems genutzt wurde.

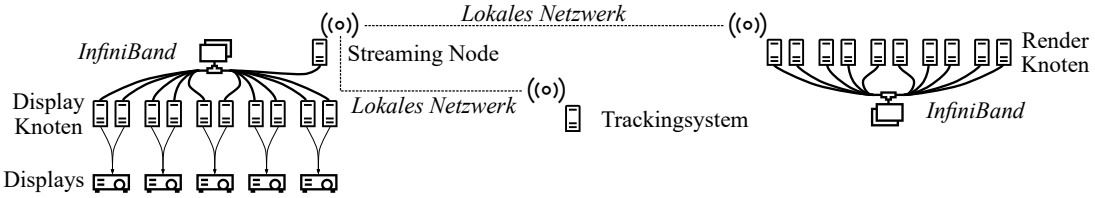


Abbildung 6.1: Das Grafik-Cluster der VISUS-Powerwall. Auf der linken Seite ist eine Skizze des Display-Clusters zu sehen. Die Display Knoten sind über InfiniBand miteinander verbunden und liefern den Input für die Projektoren der Powerwall. Das Trackingsystem (mittig) und das Rendering-Cluster (rechts) sowie das Display-Cluster sind über ein lokales Netzwerk miteinander verbunden. Das Rendering-Cluster wird nicht mehr eingesetzt.

Um die Qualität und die Performanz verschiedener Ansätze vergleichen zu können, wurden folgende Szenarien getestet:

- Vollständiges Raytracing
- Support-Image
- Foveated Rendering

Beim vollständigen Raytracing wurden keine Techniken zur Aufwandsreduktion eingesetzt und für jedes Pixel ein Primärstrahl erzeugt. Die Szenarien *Support-Image* und *Foveated Rendering* funktionieren beide wie in Abschnitt 5.3 beschrieben, allerdings wird bei ersterem kein Blickfeld angegeben. Dadurch wird effektiv nur das Support-Image gerendert, was vor allem für detaillierte Darstellungen keine ausreichend hohe Qualität bietet. Deshalb wird im Folgenden nur vollständiges Raytracing mit Foveated Rendering verglichen.

Um die beiden Szenarien geeignet vergleichen zu können, wurden Zeitmessungen für die einzelnen Schritte des Programmablaufs durchgeführt. Zum einen wurden Zeiten für das Rendern, Konvertieren und Codieren eines Frames sowie die Gesamtzeit vom Beginn der Bilderzeugung bis zur Netzwerkübertragung gemessen. Der Schritt *Konvertieren* bezeichnet dabei die Konvertierung eines

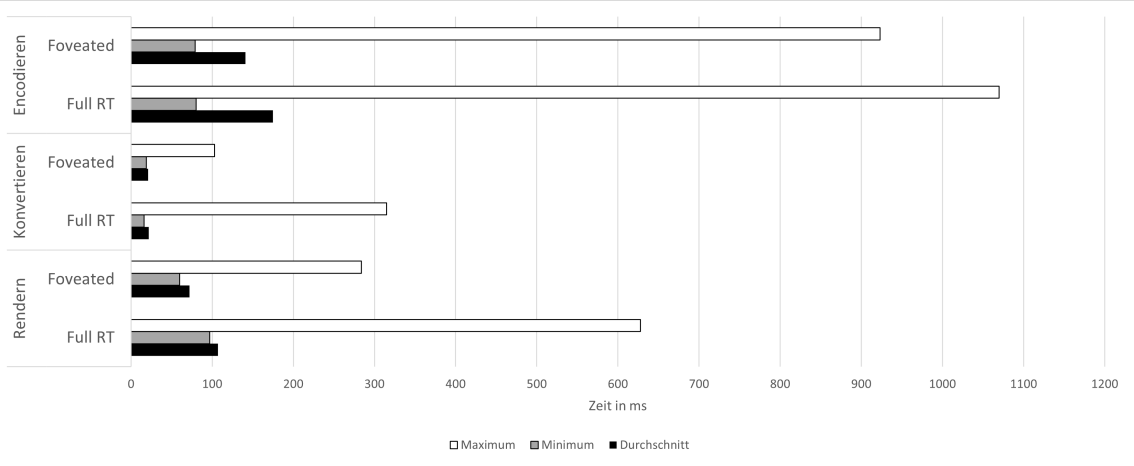


Abbildung 6.2: Überblick über die gemessenen Zeiten der Schritte *Rendern*, *Konvertieren* und *Encodieren*. Es werden die Szenarien *Vollständiges Raytracing* (Full RT) und *Foveated Rendering* im Vergleich dargestellt. Die durchschnittlichen Zeiten werden durch schwarze Balken, die minimalen und maximalen Werte durch graue bzw. weiße Balken repräsentiert. Da die Durchschnittswerte stets nahe dem Minimum liegen, stellen die maximalen Werte Ausreißer dar.

gerenderten Frames vom RGB-Farbmodell in das YUV-Modell. Zum anderen wurden Zeiten für die Schritte, die zur Decodierung und Anzeige der Bilder nötig sind, aufgezeichnet. Dazu gehört das Umsortieren von ankommenden Nachrichten, das Zusammenfügen von einzelnen Slices zu einem Gesamtbild, die Decodierung und das Kopieren der decodierten Farbwerte in eine spezielle Datenstruktur, um diese schließlich anzuzeigen.

In Abbildung 6.2 ist zu sehen, dass sowohl das Rendern als auch die Codierung beim Foveated Rendering deutlich schneller ablaufen als beim vollständigen Raytracing. Das liegt daran, dass nur ein kleiner Teil des Bildes in hoher Qualität dargestellt wird, während der restliche Teil des Bildes niedrig aufgelöst ist. Mit einer Zeit von etwa 72 ms pro Einzelbild kommt der Renderer beim Foveated Rendering auf eine Bildrate von etwa 14 Bildern pro Sekunde, engl. *frames per second* (FPS). Damit ist er um mehr als 50 % schneller als der Renderer beim vollständigen Raytracing, welcher nur auf etwa 9 FPS kommt. Die Codierung benötigt beim vollständigen Raytracing durchschnittlich 175 ms pro Einzelbild. Damit wird für diesen Encoder eine Bildrate von knapp 6 FPS erreicht. Mit etwa 7 FPS ist der Encoder beim Foveated Rendering geringfügig schneller. Allerdings wird die Echtzeitnutzung des Systems durch die Geschwindigkeit beim Encodieren stark eingeschränkt, da für Echtzeitanwendung Bildraten jenseits von 30 FPS charakteristisch sind. Die Konvertierung läuft bei beiden Szenarien ungefähr gleich schnell ab. Für den gesamten Prozess vom Start der Bilderzeugung bis zur Netzwerkübertragung wurde eine durchschnittliche Zeit von etwa 2,3 s bei der Foveated-Variante und circa 2,9 s beim vollständigen Raytracing gemessen. Dieser hohe Wert rührt daher, dass der Renderer und der Encoder mit unterschiedlichen Geschwindigkeiten arbeiten. Dadurch kann es passieren, dass ein gerenderter Frame nicht direkt codiert werden kann, da noch vorhergehende Frames codiert werden müssen. Für die Bildwiederholrate der Anzeige hat dies keine Auswirkung, allerdings werden Einzelbilder etwa zwei Sekunden nachdem sie gerendert wurden erst verschickt. Um diesem Problem entgegenzuwirken, könnte man die Codierung auf einen oder mehrere separate Knoten auslagern.

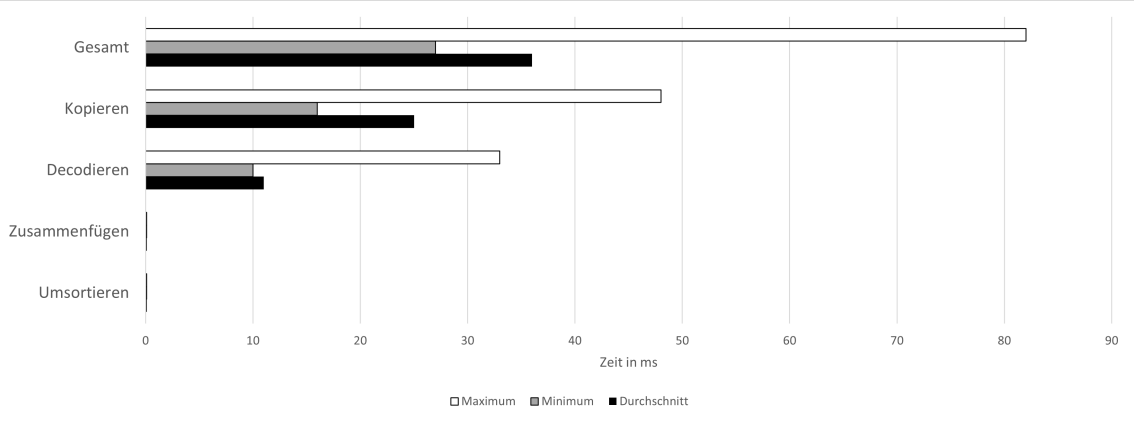
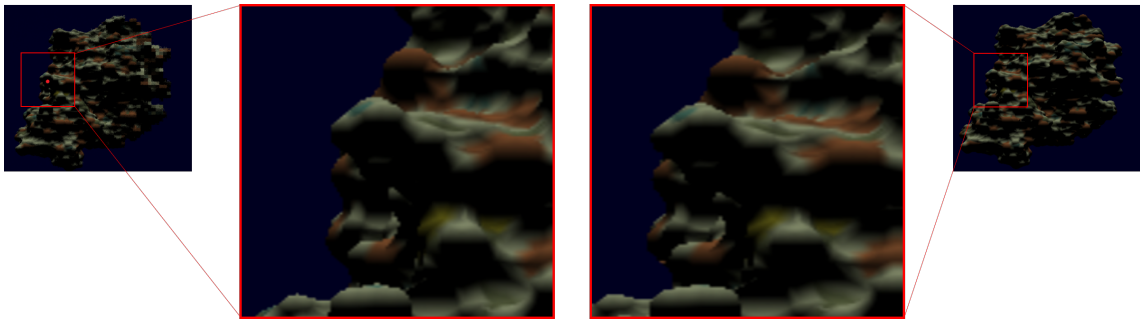


Abbildung 6.3: Überblick über die gemessenen Zeiten der Schritte, die zur Decodierung und Anzeige der gerenderten Bilder notwendig sind. Es findet kein Vergleich zwischen vollständigem Raytracing und Foveated Rendering statt, da die Decodierung und Anzeige unabhängig vom Szenario gleich lange dauert. Die durchschnittlichen Zeiten werden durch schwarze Balken, die minimalen und maximalen Werte durch graue bzw. weiße Balken repräsentiert. Da die Durchschnittswerte stets nahe dem Minimum liegen, stellen die maximalen Werte Ausreißer dar.

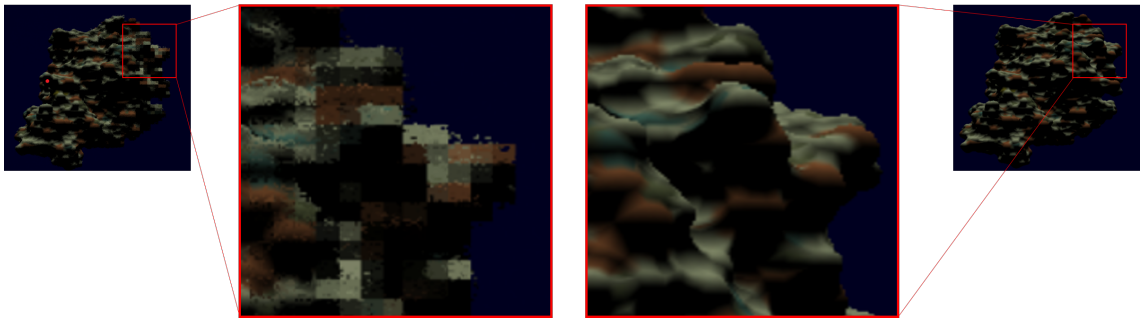
In Abbildung 6.3 sind die gemessenen Zeiten für die Schritte, welche zur Decodierung und Anzeige der gerenderten Frames nötig sind, grafisch dargestellt. Das Umsortieren von ankommenden Nachrichten und das Zusammenfügen einzelner Teilbilder zu einem Gesamtbild dauert jeweils deutlich weniger als 1 ms und kann daher vernachlässigt werden. Der Decoder braucht zum Decodieren eines Einzelbilds im Durchschnitt 11 ms und kommt somit auf etwa 90 FPS. Der gesamte Vorgang, bis ein Bild angezeigt wird, dauert durchschnittlich 36 ms, da decodierte Bildinformationen zunächst in eine Datenstruktur kopiert werden müssen, bevor sie angezeigt werden können, was circa 25 ms in Anspruch nimmt. Insgesamt können Bilder mit einer Bildrate von etwa 30 FPS angezeigt werden, wodurch dieser Part des Systems echtzeitfähig ist.

Betrachtet man das System als Ganzes, so lässt sich feststellen, dass durch Foveated Rendering der Rechenaufwand beim Raytracing deutlich gesenkt werden kann. Dennoch ist das System nur bedingt für die Nutzung in Echtzeit einsetzbar, da das CPU-basierte Raytracing und die Codierung mit deutlich weniger als 30 FPS ablaufen. Mögliche Verbesserungen werden in Kapitel 7 genannt und kurz erläutert.

Die Qualität der gerenderten Bilder ist sowohl bei vollständigem Raytracing als auch bei Foveated Rendering sehr hoch. Obwohl beim Foveated Rendering ein großer Teil des Bildes niedrig aufgelöst dargestellt wird, gibt es für einen Betrachter kaum merkliche Unterschiede im Vergleich zu vollständigem Raytracing. In Abbildung 6.4 werden die Bildqualitäten beider Ansätze gegenübergestellt.



(a) Foveated Region



(b) Peripherie

Abbildung 6.4: Foveated Rendering (links) und vollständiges Raytracing (rechts) im Vergleich. (a) Die Qualität ist beim Foveated Rendering im Bereich des Blickfeldes etwa gleich hoch wie beim vollständigen Raytracing. (b) Während beim vollständigen Raytracing die Qualität des gesamten Bildes hoch ist, sind beim Foveated Rendering Bereiche in der Peripherie nur unscharf. Dieser Qualitätsverlust ist jedoch für das menschliche Auge kaum wahrnehmbar.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein System vorgestellt, welches CPU-basiertes Raytracing für hochauflösende Anzeigen ermöglicht. Das System wurde auf der Powerwall und dem Grafik-Cluster des VISUS getestet. Die Qualität der gerenderten Bilder ist selbst auf großen, gekachelten Anzeigen wie der Powerwall sehr hoch.

Der Bilderzeugungsprozess wurde durch die Verwendung von *Foveated Rendering* optimiert. Mithilfe von Eye-Tracking wird das Blickfeld eines Benutzers bestimmt und in den Bilderzeugungsprozess miteinbezogen. Dabei werden die nicht-uniformen Eigenschaften der menschlichen Netzhaut ausgenutzt, um dem Betrachter den Eindruck eines vollständig scharfen Bildes zu vermitteln, obwohl nur ein kleiner Teil, die *Foveated Region*, scharf dargestellt wird. Innerhalb dieser Region wird für jedes Pixel anhand einer Funktion, welche die Verteilung der farbempfindlichen Zapfen auf der Netzhaut annähert, bestimmt, ob ein Schnitttest für den Raytracing-Algorithmus durchgeführt wird. Pixel, für die kein Schnitttest stattfindet, sowie Pixel außerhalb der Foveated Region erhalten ihre Farbe aus einem zuvor gerenderten Support-Image, welches deutlich niedriger aufgelöst ist als das finale Bild. Dadurch kann die Anzahl der insgesamt erzeugten Primärstrahlen und damit der Aufwand für das Raytracing deutlich reduziert werden. Allerdings verschlechtert sich infolgedessen auch die Qualität des Bildes mit zunehmendem Abstand vom Blickpunkt. Für einen Betrachter erscheint trotz der überwiegend niedrigen Bildqualität eine Darstellung, die subjektiv als hochauflösend empfunden wird. Durch den Einsatz von Foveated Rendering kann beim Rendern eine Bildrate von 14 FPS erreicht werden, was einer Steigerung von mehr als 50 % gegenüber vollständigem Raytracing entspricht.

Es wurde eine Netzwerkübertragung über UDP implementiert, damit einerseits eine Auslagerung der für das Rendern notwendigen Berechnungen an einen Remote-Standort ermöglicht wird und andererseits gerenderte Frames vom Remote-Standort zurück an die Rechnerinfrastruktur der Anzeige übertragen werden können. Um die benötigte Bandbreite möglichst gering zu halten, wurde eine Codierung der gerenderten Frames vorgenommen. Der implementierte Encoder erreicht eine Bildrate von etwa 6 - 7 FPS und schränkt daher die Echtzeitfähigkeit des Systems ein.

Damit eine parallele und unabhängige Ausführung von Rendern und Encodieren möglich ist, wurden Ringpuffer zur Speicherung der Bilddaten implementiert. Dadurch müssen nur selten Bildinformationen verworfen werden. Allerdings entsteht eine Verzögerung bei der Netzwerkübertragung von encodierten Frames, welche sich jedoch nicht negativ auf die Bildwiederholrate der Anzeige auswirkt.

Die Decodierung sowie die Anzeige der gerenderten Bilder funktioniert in Echtzeit.

Obwohl im Vergleich zu vollständigem Raytracing eine Steigerung der Bildfrequenz von über 50 % erzielt werden konnte, ist das System nur bedingt für die Echtzeitanwendung geeignet. Durch verschiedene Verbesserungen, zum Beispiel beim Encodieren, kann eventuell die Echtzeitfähigkeit erreicht werden.

Das System ist für die Remote-Anwendung geeignet und bietet die Möglichkeit, Berechnungen auf ein externes Cluster auszulagern, da sowohl eine Cluster-interne Kommunikation über MPI als auch eine Netzwerkübertragung der Daten über UDP implementiert wurde.

Ausblick

Eine mögliche Verbesserung des Systems kann erreicht werden, indem nicht nur die Bildqualität beim Rendern, sondern auch die Qualität beim Encodieren der gerenderten Bilder an das Blickfeld des Benutzers angepasst wird. Durch diese Technik, welche auch als *Foveated Encoding* bezeichnet wird, kann zusätzlich Rechenaufwand und Speicherbedarf eingespart werden. Eine weitere Möglichkeit der Verbesserung besteht in der Auslagerung der Codierung auf separate Cluster-Knoten. Dadurch wird die Latenz beim Rendern gesenkt, da nicht gewartet werden muss, bis ein gerendertes Bild encodiert wurde. Werden die Berechnungen des entwickelten Systems auf einem externen Cluster mit mehr Rechenleistung ausgeführt, so kann die Performanz unter Umständen weiter gesteigert werden. Durch die kontinuierliche Weiterentwicklung und Verbesserung von CPUs und dem damit einhergehenden Anstieg der verfügbaren Rechenleistung wird es immer wahrscheinlicher, dass Systeme wie das in dieser Arbeit vorgestellte in Zukunft auch für große, gekachelte Anzeigen in Echtzeit arbeiten können.

Literaturverzeichnis

[BB16] W. Burger, M. J. Burge. „Color Images“. In: *Digital Image Processing: An Algorithmic Introduction Using Java*. London: Springer London, 2016, S. 291–328. ISBN: 978-1-4471-6684-9. DOI: 10.1007/978-1-4471-6684-9_12. URL: https://doi.org/10.1007/978-1-4471-6684-9_12 (zitiert auf S. 23).

[CAB15] V. Chernov, J. Alander, V. Bochko. „Integer-Based Accurate Conversion between RGB and HSV Color Spaces“. In: *Comput. Electr. Eng.* 46.C (Aug. 2015), S. 328–337. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2015.08.005. URL: https://doi.org/10.1016/j.compeleceng.2015.08.005 (zitiert auf S. 23).

[DK11] K.-U. Doerr, F. Kuester. „CGLX: A Scalable, High-Performance Visualization Framework for Networked Display Environments“. In: *IEEE transactions on visualization and computer graphics* 17 (März 2011), S. 320–32. DOI: 10.1109/TVCG.2010.59 (zitiert auf S. 17).

[FBB+20] F. Frieß, M. Braun, V. Bruder, S. Frey, G. Reina, T. Ertl. „Foveated Encoding for Large High-Resolution Displays“. In: *IEEE Transactions on Visualization and Computer Graphics* (2020), S. 1–1. DOI: 10.1109/TVCG.2020.3030445 (zitiert auf S. 16).

[Fis16] W. Fischer. „Videocodierung (MPEG-2, MPEG-4/AVC, HEVC)“. In: *Digitale Fernseh- und Hörfunktechnik in Theorie und Praxis: MPEG-Quellcodierung und Multiplexbildung, analoge und digitale Hörfunk- und Fernsehstandards, DVB, DAB/DAB+, ATSC, ISDB-T, DTMB, terrestrische, kabelgebundene und Satelliten-Übertragungstechnik, Messtechnik*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, S. 133–186. ISBN: 978-3-642-53896-4. DOI: 10.1007/978-3-642-53896-4_7. URL: https://doi.org/10.1007/978-3-642-53896-4_7 (zitiert auf S. 31).

[FM14] S. Frings, F. Müller. „Sehen“. In: *Biologie der Sinne: Vom Molekül zur Wahrnehmung*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 117–188. ISBN: 978-3-8274-2273-6. DOI: 10.1007/978-3-8274-2273-6_7. URL: https://doi.org/10.1007/978-3-8274-2273-6_7 (zitiert auf S. 21).

[FME20] F. Frieß, C. Müller, T. Ertl. „Real-time High-resolution Visualisation“. In: *Vision, Modeling, and Visualization*. Hrsg. von J. Krüger, M. Niessner, J. Stückler. The Eurographics Association, 2020. ISBN: 978-3-03868-123-6. DOI: 10.2312/vmv.20201195 (zitiert auf S. 17, 41).

[FNM+14] A. Febretti, A. Nishimoto, V. Mateevitsi, L. Renambot, A. Johnson, J. Leigh. „Omegalib: A multi-view application framework for hybrid reality display environments“. In: *2014 IEEE Virtual Reality (VR)*. 2014, S. 9–14. DOI: 10.1109/VR.2014.6802043 (zitiert auf S. 17).

[FS05] T. Foley, J. Sugerman. „KD-Tree Acceleration Structures for a GPU Raytracer“. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS ’05. Los Angeles, California: Association for Computing Machinery, 2005, S. 15–22. ISBN: 1595930868. DOI: 10.1145/1071866.1071869. URL: https://doi.org/10.1145/1071866.1071869 (zitiert auf S. 15).

[GBB+19] P. Gralka, M. Becher, M. Braun, F. Frieß, C. Müller, T. Rau, K. Schatz, C. Schulz, M. Krone, G. Reina, T. Ertl. „MegaMol – a comprehensive prototyping framework for visualizations“. In: *The European Physical Journal Special Topics* 227.14 (März 2019), S. 1817–1829. ISSN: 1951-6401. DOI: 10.1140/epjst/e2019-800167-5. URL: https://doi.org/10.1140/epjst/e2019-800167-5 (zitiert auf S. 13, 39).

[GFD+12] B. Guenter, M. Finch, S. Drucker, D. Tan, J. Snyder. „Foveated 3D Graphics“. In: *ACM SIGGRAPH Asia*. Nov. 2012. URL: https://www.microsoft.com/en-us/research/publication/foveated-3d-graphics/ (zitiert auf S. 16).

[Gla84] A. S. Glassner. „Space subdivision for fast ray tracing“. In: *IEEE Computer Graphics and Applications* 4.10 (1984), S. 15–24. DOI: 10.1109/MCG.1984.6429331 (zitiert auf S. 15).

[Gla89] A. S. Glassner, Hrsg. *An Introduction to Ray Tracing*. GBR: Academic Press Ltd., 1989. ISBN: 0122861604 (zitiert auf S. 15).

[GPSS07] J. Gunther, S. Popov, H. Seidel, P. Slusallek. „Realtime Ray Tracing on GPU with BVH-based Packet Traversal“. In: *2007 IEEE Symposium on Interactive Ray Tracing*. 2007, S. 113–118. DOI: 10.1109/RT.2007.4342598 (zitiert auf S. 15).

[HDM+13] J. F. Hughes, A. van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. Feiner, K. Akeley. *Computer Graphics: Principles and Practice*. 3. Aufl. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 978-0-321-39952-6 (zitiert auf S. 15, 19, 20, 22).

[Kaj86] J. T. Kajiya. „The Rendering Equation“. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), S. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://doi.org/10.1145/15886.15902 (zitiert auf S. 24).

[Kor12] F. Korbel. *FFmpeg Basics Multimedia handling with a fast audio and video encoder*. Dez. 2012. ISBN: 978-1479327836 (zitiert auf S. 40).

[LW90] M. Levoy, R. T. Whitaker. „Gaze-directed volume rendering“. In: *Proceedings of the 1990 symposium on Interactive 3D graphics, I3D ’90, Snowbird, UT, USA, March 1990*. Hrsg. von M. Zyda. ACM, 1990, S. 217–223. DOI: 10.1145/91385.91449. URL: https://doi.org/10.1145/91385.91449 (zitiert auf S. 16).

[MAN+14] T. Marrinan, J. Aurisano, A. Nishimoto, K. Bharadwaj, V. Mateevitsi, L. Renambot, L. Long, A. Johnson, J. Leigh. „SAGE2: A New Approach for Data Intensive Collaboration Using Scalable Resolution Shared Displays“. In: Okt. 2014. DOI: 10.4108/icst.collaboratecom.2014.257337 (zitiert auf S. 17).

[MRE13] C. Müller, G. Reina, T. Ertl. „The VVand: A Two-Tier System Design for High-Resolution Stereo Rendering“. In: *CHI POWERWALL 2013 Workshop*. 2013 (zitiert auf S. 11, 17).

[RL15] M. Ramamurthy, V. Lakshminarayanan. „Human Vision and Perception“. In: Jan. 2015. DOI: 10.1007/978-3-319-00295-8_46-1 (zitiert auf S. 21).

[RWH+16] T. Roth, M. Weier, A. Hinkenjann, Y. Li, P. Slusallek. „An analysis of eye-tracking data in foveated ray tracing“. In: *2016 IEEE Second Workshop on Eye Tracking and Visualization (ETVIS)*. 2016, S. 69–73 (zitiert auf S. 16).

[RZK+19] T. Rau, S. Zahn, M. Krone, G. Reina, T. Ertl. „Interactive CPU-based Ray Tracing of Solvent Excluded Surfaces“. In: *Eurographics Workshop on Visual Computing for Biology and Medicine*. Hrsg. von B. Kozlíková, L. Linsen, P.-P. Vázquez, K. Lawonn, R. G. Raidou. The Eurographics Association, 2019. ISBN: 978-3-03868-081-9. DOI: 10.2312/vcbm.20191249 (zitiert auf S. 12).

[SCMP19] A. Siekawa, M. Chwesiuk, R. Mantiuk, R. Piórkowski. „Foveated Ray Tracing for VR Headsets“. In: *MultiMedia Modeling*. Hrsg. von I. Kompatsiaris, B. Huet, V. Mezaris, C. Gurrin, W.-H. Cheng, S. Vrochidis. Cham: Springer International Publishing, 2019, S. 106–117. ISBN: 978-3-030-05710-7 (zitiert auf S. 16).

[SM09] P. Shirley, S. Marschner. *Fundamentals of Computer Graphics*. 3rd. USA: A. K. Peters, Ltd., 2009. ISBN: 1568814690 (zitiert auf S. 15, 22).

[SOHW12] G. J. Sullivan, J. Ohm, W. Han, T. Wiegand. „Overview of the High Efficiency Video Coding (HEVC) Standard“. In: *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (2012), S. 1649–1668. DOI: 10.1109/TCSVT.2012.2221191 (zitiert auf S. 32).

[Whi80] T. Whitted. „An Improved Illumination Model for Shaded Display“. In: *Commun. ACM* 23.6 (Juni 1980), S. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882 (zitiert auf S. 15).

[WJA+17] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, P. Navratil. „OSPRay - A CPU Ray Tracing Framework for Scientific Visualization“. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), S. 931–940. DOI: 10.1109/TVCG.2016.2599041 (zitiert auf S. 13, 15, 40).

[WRK+16] M. Weier, T. Roth, E. Kruijff, A. Hinkenjann, A. Pérard-Gayot, P. Slusallek, Y. Li. „Foveated Real-Time Ray Tracing for Head-Mounted Displays“. In: *Comput. Graph. Forum* 35.7 (Okt. 2016), S. 289–298. ISSN: 0167-7055 (zitiert auf S. 16).

[WSBL03] T. Wiegand, G. J. Sullivan, G. Bjontegaard, A. Luthra. „Overview of the H.264/AVC video coding standard“. In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (2003), S. 560–576. DOI: 10.1109/TCSVT.2003.815165 (zitiert auf S. 31).

Alle URLs wurden zuletzt am 06. 01. 2021 geprüft.

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