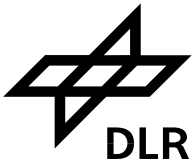


Automatisierte Umgebungserfassung in Mixed Reality

Bachelorarbeit

des Studienganges Informationstechnik an der
Dualen Hochschule Baden-Württemberg Mannheim

von

Jan Wulkop

17. September 2018

Bearbeitungszeitraum:	25.05.2018 bis 16.09.2018
Matrikelnummer, Kurs:	*1562662*, *TINF15ITIN*
Abteilung:	Interaktive Visualisierung
Ausbildungsfirma:	Deutsches Zentrum für Luft- und Raumfahrt e. V.
Betrieblicher Betreuer:	M.Sc. Sebastian Utzig
Betreuer der Dualen Hochschule:	Prof. Dr. Heinz-Jürgen Müller
Unterschrift Betrieblicher Betreuer	

Erklärung der Eigenleistung

Gemäß §5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Braunschweig, den 15. September 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Kurzfassung: In vielen Bereichen der modernen Forschung und Industrie wird international von mehreren Standorten aus an einem gemeinsamen Projekt gearbeitet. Mithilfe des Internets ist es bereits über weite Entfernungen hinweg möglich, gemeinsam miteinander zu kommunizieren und kollaborativ zusammenzuarbeiten. Dennoch befinden sich die jeweiligen Parteien in unterschiedlichen Umgebungen.

In dieser Arbeit wird eine Anwendung implementiert, welche die physisch-reale Umgebung eines Benutzer erfasst und in einer dreidimensionalen Rekonstruktion speichert. Sowohl Form, als auch farbiges Erscheinungsbild beliebig großer Umgebungen soll mithilfe moderner *Mixed Reality* Technik in Echtzeit aufgenommen und anderen über Netzwerk verbundenen Teilnehmern angezeigt werden. Diese Teilnehmer sollen sich unabhängig voneinander frei in der rekonstruierten Umgebung bewegen können. Eine mobile *Mixed Reality Brille*, die Microsoft HoloLens, soll dabei sowohl Ein- und Ausgabe, als auch die Verarbeitung der Daten übernehmen.

Mithilfe der integrierten Sensoren und Kameras wird die Umgebung vermessen und Fotos der Oberflächen aufgenommen. Diese Informationen werden in einer auf der HoloLens laufenden Anwendung verarbeitet. Die dreidimensionale Rekonstruktion kann mithilfe mehrerer Optimierungsverfahren in Echtzeit mit den begrenzten Hardwareressourcen der HoloLens dargestellt werden. Mehrere Teilnehmer können sich mit einer HoloLens verbinden, um die erfasste Umgebung zu empfangen.

Abstract: In many sectors of modern industry and research engineers coming from different locations are working together on a common project. Communication and collaboration over long distances are archived by using the Internet. Nevertheless, the respective parties are in different environments.

This project deals with the implementation of an application capturing the physical-real environment of a user. Both form and color appearance of arbitrarily large environments are to be recorded in real time using modern *Mixed Reality* technology. The reconstruction should be transmitted to other participants connected via network. These participants should be able to move freely in the reconstructed environment independently of each other. A mobile *Mixed Reality* device called Microsoft HoloLens should be used do reconstruct, process, visualize and distribute the environment to other participants.

The integrated sensors and cameras measures the environment and takes pictures of surfaces. This information is processed in an application running on the HoloLens. Using multiple optimization techniques the three-dimensional reconstruction can be displayed in real time. Multiple participants can connect to a HoloLens to receive and visualize the captured environment.

1 Einführung

Die Forschung und Entwicklung im Bereich der Luft- und Raumfahrt ist ein internationaler Prozess. Unabhängig vom Aufenthaltsort der Ingenieure wird an gemeinsamen Projekten gearbeitet. Dafür werden bereits heute Techniken der sogenannten *Telepräsenz* verwendet.[1] Mithilfe von Telepräsenztechniken kann eine Person, unabhängig vom realen Aufenthaltsort, eine andere Umgebung wahrnehmen und in dieser wahrgenommen werden. Ein klassisches, alltägliches Beispiel ist eine Videokonferenz.

Einige Probleme setzen dennoch die Anwesenheit der Experten an einem bestimmten Standort voraus. Ein Anwendungsfall ist beispielsweise die Wartung eines Bauteils. Besonders in der Luft- und Raumfahrt gelten hohe Anforderungen an die Qualität der Bauteile. In regelmäßigen Abständen wird ein Bauteil gewartet. Falls ein Fehler am Bauteil festgestellt wird, muss dieser durch Experten vor Ort begutachtet werden. Die Anreise der Ingenieure oder die Verfrachtung des Bauteils selbst sind sowohl zeitintensiv, als auch kostspielig.

Auch außerhalb der Luft- und Raumfahrt kommt Telepräsenz zum Einsatz, wenn die Präsenz eines Menschen nicht möglich oder zu gefährlich ist. Anstelle von Menschen werden Roboter eingesetzt, welche aus sicherer Entfernung gesteuert werden. Dabei wird mithilfe von Kameras die Umgebung aus Sicht des Roboters übertragen. Eine vom Roboter unabhängige Betrachtung der Umgebung ist nicht möglich.

In dieser Arbeit wird ein Konzept entwickelt, welches eine Umgebungserfassung mithilfe moderner Mixed Reality Technik realisiert. Ziel ist dabei, sowohl die Form, als auch die Oberflächenbeschaffenheit der umgebenden Objekte möglichst detailliert aufzunehmen und anzuzeigen. Benutzer können unabhängig ihrer eigenen Umgebung in diese Rekonstruktion eintauchen und sich frei innerhalb dieser bewegen. Am Beispiel der Bauteilwartung kann somit ein Experte auch aus der Entfernung am Entscheidungsprozess des Prüfers vor Ort mitwirken. Dies gelingt ihm auf Basis eines virtuellen Abbildes, welches Materialverformungen beinhaltet.

[1] Vgl. Mittelbach und Albu-Schäffer, 2018.

2 Motivation

Das Ziel dieses Projektes besteht in der Entwicklung, Implementierung und Evaluierung eines Softwarekonzeptes, um die physisch-reale Umgebung einer Person detailliert zu digitalisieren. Dieses setzt sowohl eine genaue Erfassung der Topologie, als auch eine möglichst hochauflösende Farbaufnahme der Oberflächen voraus.

Sowohl für die Erfassung der Umgebung, als auch für die anschließende Darstellung soll aktuelle, mobile Mixed Reality Technik in Form der Microsoft HoloLens verwendet werden. Über eine Unity3D Anwendung soll die Umgebung des Benutzers schrittweise automatisch erfasst und in einem Dreiecksmodell gespeichert werden. Diese Anwendung soll auf der HoloLens ausgeführt werden.

Über die Farbkamera an der Frontseite der HoloLens sollen in regelmäßigen Abständen Fotos der Umgebung aufgenommen werden. Diese Fotoaufnahmen müssen perspektivisch korrekt auf der rekonstruierten Oberfläche angezeigt und gespeichert werden. Bilder aus unterschiedlichen Perspektiven sind so zu kombinieren, dass die Farbinformation die Oberfläche des 3D Modells maximal abdeckt. Durch die Visualisierung der eingefärbten Rekonstruktion in Echtzeit soll dem Benutzer ein Eindruck über den aktuellen Detailgrad der erfassten Umgebung vermittelt werden.

Dabei sollen keine Annahmen über die Größe des zu rekonstruierenden Objektes gemacht werden. Daher muss sichergestellt werden, dass die Anwendung mit einer konstanten Bildwiederholrate auch bei sehr großen Rekonstruktionen ausführbar ist.

Ein Ziel ist es, die rekonstruierte Umgebung auf mehreren HoloLenses gleichzeitig angezeigt werden können. Um ein kollaboratives Arbeiten mehrerer Benutzer in der gleichen virtuellen Umgebung zu ermöglichen, müssen sowohl Farb- als auch Topologieinformationen des Raumes an andere HoloLenses über Netzwerk versendet werden. Dabei verbinden sich mehrere Anwendungen in einer Sitzung, in welcher eine HoloLens die Umgebung rekonstruiert und die Daten versendet. Die anderen Anwendungen in der Sitzung empfangen die Daten und zeigen anschließend das Modell samt Textur an.

Über eine intuitive Benutzeroberfläche sollen Anwender eine neue Sitzung starten oder einer bestehenden Sitzung beitreten können.

3 Stand der Forschung

Die Erfassung realer Objekte in Kombination aus Farbbildern und einer 3D-Rekonstruktion ist ein zentraler Gegenstand im Gebiet der Mixed Reality. Virtuelle Inhalte überlagern sich mit der realen Umgebung und interagieren mit dieser. Um reale Objekte an beliebigen Orten betrachten zu können, müssen diese in Form eines dreidimensionalen Abbildes rekonstruiert werden. Der Detailgrad dieser Modelle wird erhöht, indem zusätzlich zu der Form das Aussehen der realen Oberflächen anhand von Fotoaufnahmen erfasst wird. Das rekonstruierte Modell wird *texturiert*.

Einige Ansätze wie beispielsweise Pollefeys u. a., 1999, basieren auf der Verarbeitung mehrerer Fotoaufnahmen unterschiedlicher Perspektiven, die das zu rekonstruierende Objekt abbilden. Dieser Ansatz der Rekonstruktion ist im allgemeinen rechenintensiv und nicht echtzeitfähig.

Durch eine Kombination aus Tiefen- und Farbkamera konnten mit der *Microsoft Kinect* neue Ansätze erarbeitet werden. Alexiadis u. a., 2017 verwenden mehrere *Kinects*, um die Oberfläche eines menschlichen Körpers zu rekonstruieren. Sowohl die Form des Körpers, als auch die Oberflächentextur wird in Echtzeit erfasst. Dabei werden die *Kinects* in einem gleichen Abstand kreisförmig um den Körper platziert. Sowohl die gemessenen Entfernungen von Oberflächen zu den Kameras, als auch Farbinformationen der Kameras werden verarbeitet. Unterscheiden sich die aufgenommenen Farbwerte eines Punktes, welcher von mehreren *Kinect* Sensoren gleichzeitig erfasst wird, erfolgt eine Gewichtung entsprechend der Qualität der Farbaufnahme. Durch einen Vergleich eines aufgenommenen Farbfotos und der gerenderten Rekonstruktion aus der gleichen Perspektive, wurde die Qualität des rekonstruierten Objektes bewertet.

Auch Maimone und H. Fuchs, 2012 benutzten mehrere *Kinect* Sensoren, um Form und Textur größerer Räume in Echtzeit zu erfassen und darzustellen. Anhand von Daten fünf verschiedener *Kinect* Sensoren wurde experimentell ein Raum mit den Ausmaßen von 4,25 x 2,5 x 2,5 Meter rekonstruiert und texturiert.

Neben der *Kinect* Kamera, werden auch andere Tiefenkameras verwendet, um ein detaillierteres Modell der Umgebung zu erfassen. Garon u. a., 2016 verwenden eine externe Tiefenkamera in Verbindung mit der Microsoft HoloLens, um ein höher aufgelöstes Modell der Umgebung den HoloLens Anwendungen zur Verfügung zu stellen. Ähnlich wie in

Alexiadis u. a., 2017 und Maimone und H. Fuchs, 2012 werden die Daten der Sensoren auf einem externen Rechner verarbeitet. Eine den Rechenanforderungen des Anwendungsfalls angepasste Hardware wird für die Verarbeitung verwendet.

Die internen Hardwarekomponenten der HoloLens können nicht an den Anwendungsfall angepasst werden. Sowohl die Rechenleistung, als auch die Genauigkeit der Sensoren sind limitiert. In Dong und Hollerer, 2018 wird ein Verfahren vorgestellt, um eine begrenzte Umgebung ausschließlich mit Ressourcen der HoloLens zu rekonstruieren. Sowohl die Rekonstruktion, als auch die Texturierung wird in Echtzeit durchgeführt. Das dreidimensionale Modell wird über die integrierten Softwareschnittstellen der HoloLens einer Anwendung zur Verfügung gestellt. Die Echtzeitberechnung von Texturkoordinaten hingegen übernimmt ein eigener Algorithmus. Die bei der Texturierung benötigten Farbbilder werden einem Video entnommen und verarbeitet.

Dieser Ansatz eignet sich jedoch nur für begrenzt große Umgebungen. Alle Farbinformationen der Umgebung werden in einer globalen Textur mit einer Größe von 4096 x 4096 Pixeln gespeichert. Somit ist die Umgebungserfassung entsprechend der begrenzten Hauptspeicherressourcen optimiert, da ein konstanter Anteil des Hauptspeichers für die Textur verwendet wird. Durch die feste Texturauflösung verliert die Oberfläche mit zunehmender Größe der Rekonstruktion an Detailschärfe. Gleich viele Pixelinformationen werden für eine größere Oberfläche verwendet.

Steinbrucker u. a., 2013 thematisieren die Speicherung hochaufgelöster, dreidimensionaler Modelle und deren Texturen im Hauptspeicher. Ein Octree wird als Beschleunigungsstruktur verwendet, um dreidimensionale Daten und deren Texturen räumlich zu sortieren und effizient abzurufen.

Aufbauend auf einer 3D Rekonstruktion wird von Crowle u. a., 2015 ein Ansatz zum Versand von Texturen und Modellen in Echtzeit über Netzwerk vorgestellt. Durch Komprimierung konnte eine Übertragung dieser Daten in Bezug auf Latenzminimierung und Durchsatzsteigerung optimiert werden. Die Texturen wurden im klassischen JPEG Format nach Wallace, 1992 komprimiert und versendet. Um die Dreiecksmodelle zu komprimieren, wurde eine externe Bibliothek namens *OpenCTM* genutzt. Die komprimierten Daten wurden anschließend über das *Transmission Control Protocol* versendet.

Wulkop, 2018 behandelt ein Konzept zur Texturierung eines statischen 3D-Modells unter Verwendung mehrerer Fotoaufnahmen anhand eines realen Bauteils. Als Ein- und Ausga-

begerät wird dabei ausschließlich die HoloLens verwendet. Mittels einer Kombination aus verschiedenen Shaderprogrammen wird eine Textur für das Objekt erstellt und anhand der Farbinformationen aus der Fotoaufnahme beschrieben. Die Textur wird durch Aufnahme weiterer Fotos aus verschiedenen Perspektiven aktualisiert und in Echtzeit auf dem virtuellen Modell angezeigt. Dieser Ansatz ist jedoch nur für statische Modelle anwendbar und setzt Texturkoordinaten voraus. Außerdem wird zunächst nur die Texturierung eines einzigen Modells vorgestellt.

4 Grundlagen

Im folgenden Kapitel werden die Soft- und Hardwarekomponenten vorgestellt, welche im Rahmen dieses Projektes zum Einsatz kommen.

4.1 Microsoft HoloLens

Die HoloLens ist ein kabelloser und mobiler Computer, welcher von der Firma Microsoft entwickelt und im Jahr 2016 veröffentlicht wurde.[2] Wie in Abbildung 4.1 zu erkennen ist, unterscheidet sich die HoloLens sowohl im Aussehen, als auch in der Funktionalität von gewöhnlichen mobilen Computern.

Abbildung 4.1: Microsoft HoloLens

Die HoloLens stellt eine Kombination aus Computer und Anzeigegerät dar. Ähnlich wie bei einer gewöhnlichen Brille, ist die reale Umgebung des Trägers durch die transparenten Gläser wahrnehmbar. Digitale Inhalte werden auf zwei transparenten Bildschirmen vor den Augen des Trägers angezeigt. Beide Bildschirme können unabhängig voneinander angesteuert werden, sodass unterschiedliche Bildinhalte angezeigt werden. Durch eine perspektivische Verschiebung beider Bilder, entsteht eine räumliche Wahrnehmung virtueller Objekte, welche wie Hologramme erscheinen. Diese virtuellen Objekte werden gemeinsam mit der realen Umgebung vom Benutzer wahrgenommen.

Ein weiteres Alleinstellungsmerkmal der HoloLens äußert sich in der Möglichkeit, die virtuellen Objekte an einer definierten Position im Raum zu platzieren.[3] Der Träger der HoloLens kann sich um das virtuelle Objekt bewegen und es dabei aus unterschiedlichen Blickwinkeln betrachten.

[2] Vgl. Microsoft, 2018d.
[3] Vgl. Microsoft, 2018d.

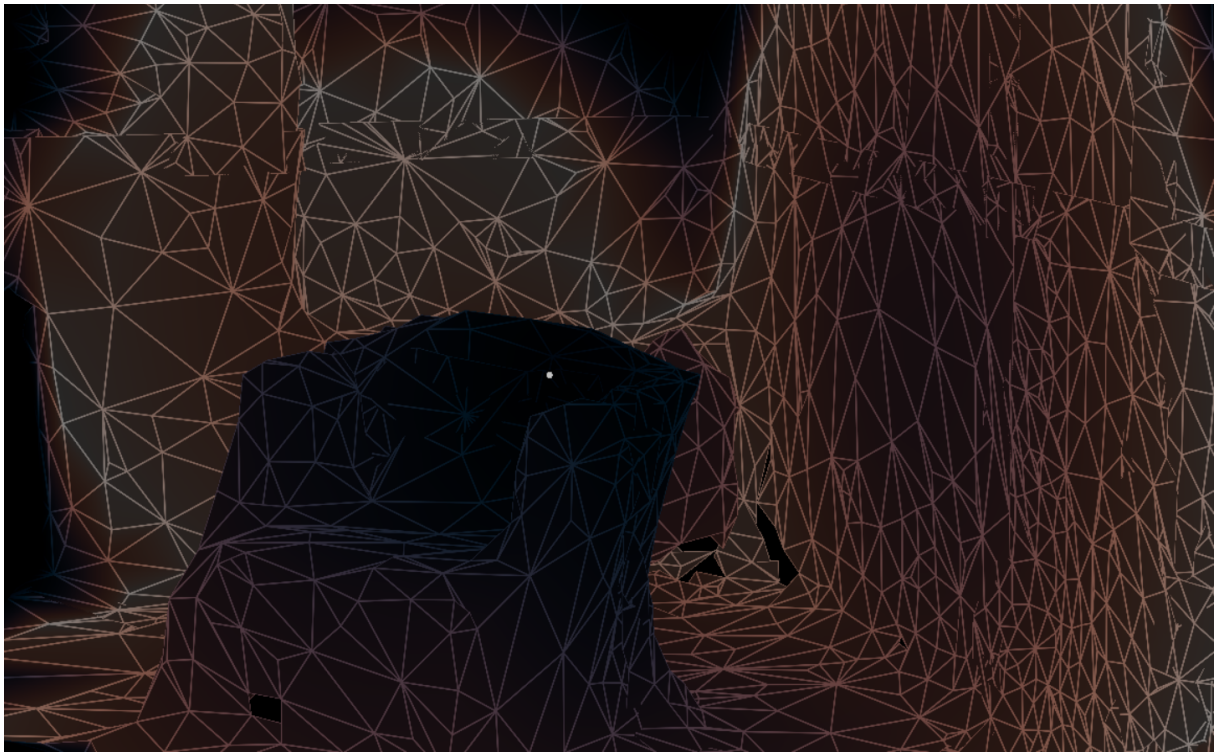


Abbildung 4.2: Dreiecksmodell der Umgebungsrekonstruktion

Um die virtuellen Objekte zu stabilisieren, erfasst die HoloLens Position und Rotation des Benutzers im Raum. Dabei rekonstruiert die HoloLens die physische Umgebung und speichert diese in einem dreidimensionalen Modell ab (siehe Abbildung. 4.2).

4.1.1 Hardwarespezifikationen

Im folgenden Abschnitt werden die Hardwarekomponenten der HoloLens näher beschrieben. Dabei wird zunächst die HoloLens ausschließlich als mobiler Computer betrachtet und die informationsverarbeitenden Komponenten näher untersucht. Im Anschluss folgt eine Betrachtung der Sensoren und der verfügbaren Kommunikationsschnittstellen.

Informationsverarbeitung

Die HoloLens ist ein mobiler Computer. Die Informationsverarbeitung ausgeführter Anwendungen kann direkt auf der Hardware der HoloLens erfolgen. Dazu besitzt die HoloLens

einen Prozessor (kurz CPU) mit integriertem Grafikchip, einen Hauptspeicher und persistenten Speicher.

Ein 32 Bit Intel Prozessor wird von der HoloLens als Hauptprozessor verwendet. Obwohl keine detaillierten Angaben über den Prozessor von Microsoft veröffentlicht wurden, konnten mithilfe des Analyseprogrammes *AIDA64 Mobile* nähere Hardwaredetails ermittelt werden.[4] Diesen Ergebnissen zufolge basiert der Prozessor auf der Intel Atom Reihe und besitzt vier Kerne mit einer Grundtaktfrequenz von 1,04 Gigahertz. In der CPU verarbeitet ein integrierter Grafikprozessor Informationen für die Bildsynthese. Der Grafikchip unterstützt eine Programmierung über die Programmierschnittstelle *DirectX* bis zur Version 11.2.[5]

Ein Coprozessor, die sogenannte *Holographic Processing Unit* wurde von Microsoft entwickelt, um die Berechnungen der CPU zu beschleunigen.[6] Nähere Details zu dieser Komponente sind jedoch nicht bekannt.

Die HoloLens verfügt insgesamt über zwei Gigabyte physischen Hauptspeicher.[7] Anwendungen dürfen jedoch maximal 900 Megabyte Speicher reservieren. Benötigt eine Anwendung mehr Hauptspeicher, wird diese automatisch geschlossen.[8] Um das Betriebssystem, die Anwendungen und deren Daten persistent zu speichern, verfügt die HoloLens über 64 Gigabyte Flash Speicher.

Sensoren

Um die eigene Position und Orientierung im Raum zu bestimmen, verwendet die Umgebungserfassung der HoloLens verschiedene Sensoren und Kameras. Lage und Beschleunigung der Brille werden mittels der *Inertial Measurement Unit* (kurz IMU) ermittelt.[9]

Die HoloLens verfügt über eine Tiefenkamera, vier sogenannte *environment understanding cameras* (umgebungsverarbeitende Kameras) und eine Farbkamera.[10] Die Tiefenkamera misst mittels Infrarotlicht Entfernungen zwischen Kamera und Objektoberflächen.

[4] Vgl. Rubino, 2016.
[5] Vgl. Microsoft, 2018j.
[6] Vgl. Microsoft, 2018d.
[7] Vgl. ebd.
[8] Vgl. Microsoft, 2018h.
[9] Vgl. Microsoft, 2018d.
[10] Vgl. Microsoft, 2018d.

Die Messungen werden zum einen für die Rekonstruktion der Umgebung, aber auch zur Erfassung von Handgesten eingesetzt.[11] Während der Rekonstruktion wird ausschließlich der Bereich der Umgebung erfasst, welcher sich in einem Winkel von 70 Grad mit einem Mindestabstand von 0,8 Meter und einem Maximalabstand von 3,1 Meter vor der Kamera befindet.[12] Die vier umgebungsverarbeitenden Kameras erfassen zusammen mit der IMU die Bewegungen des Benutzers in der rekonstruierten Umgebung.[13]

Mithilfe der integrierten Farbkamera können in Anwendungen Fotos und Videos aufgezeichnet werden. Die Auflösung der Aufnahmen kann dabei im Bereich von 896 x 504 Pixel bis 2048 x 1152 Pixel gewählt werden.[14] Abhängig von der gewählten Auflösung variiert der horizontale Winkel des Sichtfeldes zwischen 45 und 67 Grad.[15]

Kommunikationskanäle

Die HoloLens stellt ein eigenständiges System dar und ist nicht von anderen Rechnern abhängig. Zugunsten der Mobilität der HoloLens besitzt diese nur eine minimale Anzahl an Hardwareschnittstellen.

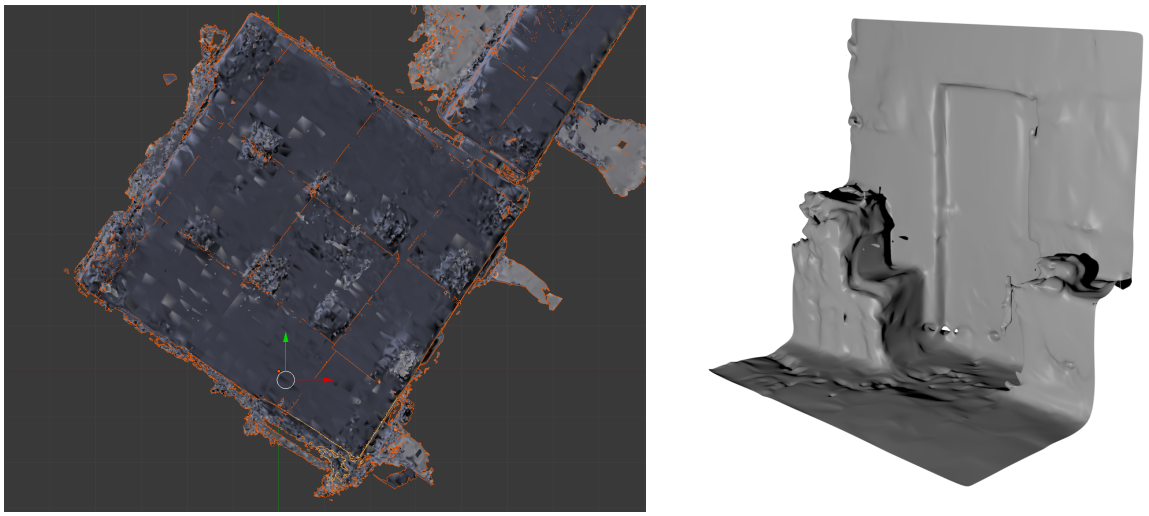
Andere Geräte können mit der HoloLens ausschließlich über Wireless LAN ac oder Bluetooth 4.1 kommunizieren.[16] Obwohl ein USB 2.0 Port vorhanden ist, wird dieser lediglich zur Stromversorgung, der Installation von Anwendungen oder der manuellen Datenübertragung verwendet.

4.1.2 Umgebungserfassung

Damit virtuelle Objekte mit realen Gegenständen in der Umgebung des Benutzers interagieren können, wird diese stückweise erfasst. Die resultierende Rekonstruktion wird dabei kontinuierlich aktualisiert.[17] Obwohl die Daten der Tiefenkamera auf Betriebssystemebene

[11] Vgl. Microsoft, 2018e.
[12] Vgl. Microsoft, 2018m.
[13] Vgl. Microsoft, 2018e.
[14] Vgl. Microsoft, 2018f.
[15] Vgl. ebd.
[16] Vgl. Microsoft, 2018d.
[17] Vgl. Microsoft, 2018l.

verarbeitet werden, ist es möglich die Raumrekonstruktion auch auf Anwendungsebene anzufragen.[18]



(a) Unterteilte Umgebung aus Vogelperspektive *(b)* Einzelner Block

Abbildung 4.3: Visualisierung der Umgebungsrekonstruktion ohne Textur

Wie in Abbildung 4.3 links zu erkennen ist, wird die Umgebung automatisch in einzelne Blöcke unterteilt. Die gemessenen 3D Punkte werden anhand deren Position im Raum gruppiert und ein Dreiecksmodell aus diesen Punkten generiert (siehe Abbildung 4.3 rechts).[19] Somit ist sichergestellt, dass lokale Änderungen in der Umgebung ausschließlich zu einer Änderung eines einzelnen Blockes führen und nicht zu einer Änderung des gesamten Modells.

Jedem Block wird automatisch eine Kennnummer zugewiesen, um diesen eindeutig zu identifizieren.[20] Sobald ein Bereich der Umgebung erfasst wird, welcher noch kein Teil der Rekonstruktion ist, wird ein neuer Block in der Anwendung erstellt. Dieser Block kann anhand der Kennung identifiziert werden. Das Dreiecksmodell eines Blocks wird durch die Umgebungserfassung kontinuierlich verändert. Sobald ein erstellter Block nicht mehr in der realen Umgebung erfasst werden kann, wird dieser Block aus der Anwendung entfernt.[21]

[18] Vgl. ebd.
[19] Vgl. Microsoft, 2018n.
[20] Vgl. Microsoft, 2018l.
[21] Vgl. Microsoft, 2018l.

Die Qualität der Rekonstruktion wird von unterschiedlichen Faktoren beeinflusst. Im Folgenden werden die vier wichtigsten Faktoren näher erläutert.

Bewegung des Benutzers: Die Tiefensensoren registrieren Veränderungen im Raum ausschließlich in einem eingeschränkten Erfassungsbereich. Es ist somit möglich, dass Teile des Raumes außerhalb des Erfassungsbereiches nicht oder nur fehlerhaft erfasst werden können, obwohl diese für den Benutzer sichtbar sind.[22]

Oberflächenbeschaffenheit: Die Entfernung von Objekten wird anhand der reflektierten Infrarotstrahlung an deren Oberfläche gemessen. Dunkle Oberflächen absorbieren die Infrarotstrahlung, sodass weniger Licht reflektiert und von der HoloLens erfasst wird. Dies führt dazu, dass solche Objekte nur erschwert erfasst werden können.[23]

Stark reflektierende Flächen sind für die Umgebungserfassung ebenfalls problematisch, wenn diese aus einem flachen Winkel betrachtet werden. Die Infrarotstrahlung wird entsprechend dem Einfallswinkel an der Oberfläche gespiegelt, sodass nur wenig Strahlung zurück in die Infrarotsensoren reflektiert wird.[24]

Bewegung in der Umgebung: Ähnlich wie die Bewegung des Benutzers, hat auch eine häufige Veränderung der Umgebung selbst einen Einfluss auf die Qualität der Rekonstruktion. Bewegliche Elemente, wie etwa Türen oder andere Personen, werden als Teil des Raumes erfasst. Verändert sich die Position dieser Gegenstände in der realen Umgebung, enthält die initiale Rekonstruktion veraltete Informationen.[25]

Andere Lichtquellen: Da die HoloLens für die Umgebungserfassung ein optisches Messverfahren verwendet, können andere Infrarotquellen einen Einfluss auf die Qualität der Rekonstruktion haben.[26]

[22] Vgl. Microsoft, 2018m.
[23] Vgl. ebd.
[24] Vgl. ebd.
[25] Vgl. ebd.
[26] Vgl. Microsoft, 2018m.

4.1.3 HoloLens Entwicklungsplattform

Alle Anwendungen, die für die HoloLens Plattform entwickelt werden (sog. *Mixed Reality Anwendungen*) basieren auf der Universellen Windows Plattform (kurz UWP). Grundsätzlich können alle UWP Anwendungen angepasst werden, sodass diese auf der HoloLens ausgeführt werden können.[27]

Für die Entwicklung solcher Anwendungen empfiehlt Microsoft entweder die Spieleengine *Unity* oder eigener Frameworks mit *DirectX* und weiteren Windows APIs.[28]

4.2 Unity

Unity ist eine plattformübergreifende Spiel-Engine.[29] Eine solche Engine stellt Softwareentwicklern eine Sammlung an Werkzeugen für die Entwicklung interaktiver Anwendungen zur Verfügung. Grundlegende Funktionen der Bild- und Physikberechnung, aber auch der Kommunikation von Anwendungen untereinander, werden abstrahiert. Dies ermöglicht die Entwicklung von Anwendungen, welche mit unterschiedlichen Hardwarekonfigurationen kompatibel sind. Kenntnisse auf hardwarenaher Ebene sind somit nicht mehr essentiell.

Sowohl die Entwicklung zwei-, als auch dreidimensionaler Programme ist möglich. Unity unterstützt den Export und die Ausführung der Applikationen auf über 25 Zielplattformen, wie beispielsweise der HoloLens.[30] Obwohl eine Verwendung der Unity-Engine besonders in der Videospielbranche verbreitet ist, wird die Engine auch in der Forschung oder in der Medizin verwendet.[31]

4.2.1 Softwareentwicklung

Unity ist eine Entwicklungsumgebung, welche die Ausführung und die Erstellung eigener Anwendungen mittels einer Benutzeroberfläche unterstützt. Über diese Oberfläche, auch Editor genannt, können grundlegende Projekteigenschaften definiert werden. Gleichzeitig

[27] Vgl. Microsoft, 2018b.
[28] Vgl. ebd.
[29] Vgl. Pluralsight, 2015.
[30] Vgl. Unity, 2018g.
[31] Vgl. Adams, 2014.

kann die Anwendung, eingebettet im Unity Editor, ausgeführt werden. Änderungen am Programmcode sind ohne Export der Anwendung sichtbar und können getestet werden.

Grundsätzlich basiert jede mit Unity entwickelte Software auf dem objektorientierten Ansatz der Softwareprogrammierung. Die einzelnen Komponenten der Anwendung werden entsprechend ihrer Aufgabe in einzelne Entitäten unterteilt. Um die Funktionalität einer Klasse nutzen zu können, muss ein Objekt dieser Klasse instantiiert werden. Klassen, welche direkt mit von Unity bereitgestellten Ressourcen interagieren, sind sogenannte *Komponenten*.[32] In diesen Komponenten wird das Verhalten und der Zustand von sogenannten *Game Objects* (im folgenden Spielobjekte genannt) bestimmt. Diese Spielobjekte sind die grundlegenden Entitäten einer Unity Anwendung, welche die Klasseninstanzen als Komponenten enthalten.

Jedem Spielobjekt wird eine Position, Rotation und Skalierung im virtuellen 3D-Koordinatensystem zugewiesen. Der Inhalt des Spielobjektes wird durch die angefügten Komponenten definiert. Spielobjekte werden innerhalb einer Unity Szene gespeichert und können hierarchisch geordnet werden. Mittels der Benutzeroberfläche des Unity Editors lässt sich diese Hierarchie der Spielobjekte verändern. Referenzen zwischen einzelnen Komponenten lassen sich ebenfalls im Editor zuweisen. Unity unterstützt die Verwendung der Programmiersprache C#.[33]

4.2.2 Rendering

Der Begriff des Renderings umfasst im Allgemeinen die Bildsynthese basierend auf einer virtuellen Szene. Das synthetisierte Bild wird in einer definierten Pixelauflösung auf einem Ausgabegerät angezeigt oder gespeichert. Im folgenden Abschnitt wird am Beispiel von Unity dieser Prozess näher beschrieben.

Kamera

Unity verwendet beim Rendering einer Szene eine virtuelle Kamera. Diese Kamera ähnelt in vielen Eigenschaften einer real existierenden Kamera. Sie wird dazu verwendet, eine

[32] Vgl. Unity, 2018a.
[33] Vgl. Unity, 2018a.

dreidimensionale Szene auf einer zweidimensionale Projektionsfläche abzubilden. Das resultierende Bild ist dabei zum einen von dem Inhalt der Szene, aber auch von den Parametern der Kamera abhängig. Über die sogenannten *extrinsischen* Parameter wird die Perspektive der Kamera auf die Szene beschrieben.[34] Position und Rotation der Kamera sind extrinsisch und unabhängig von der Art der Kamera.[35]

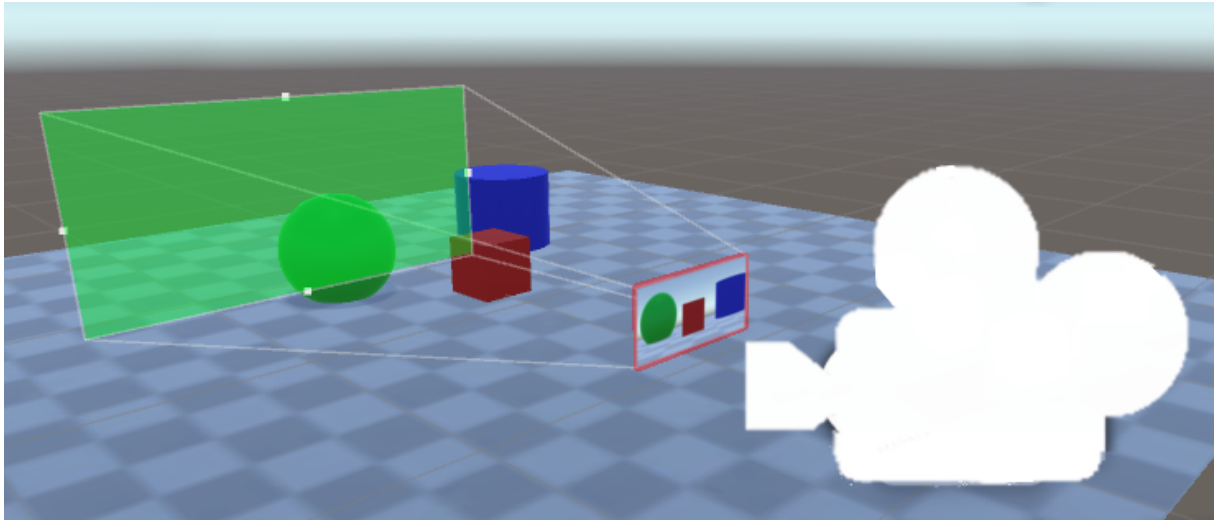


Abbildung 4.4: Konzept des Kamerafrustums

Sogenannte *intrinsische* Parameter sind spezifisch für einen Kameratyp, zum Beispiel Brennweite der Linse oder Format des Bildsensors.[36] In Abhängigkeit der extrinsischen und intrischen Parameter einer Kamera, lässt sich der Sichtbereich (auch Frustum genannt) der Kamera bestimmen. Dieser ist in Abbildung 4.4 mittels vier weißer Strahlen ausgehend von der Kamera hervorgehoben. Objekte in diesem Bereich sind bei der Bildsynthese sichtbar, solange sie nicht von anderen Objekten verdeckt werden. Über die extrinsischen Parameter lassen sich virtuelle Objekte aus dem globalen Koordinatensystem in das lokale System der Kamera transformieren.[37] Diese können daraufhin mittels der sogenannten *Projektionsmatrix* einem Bereich auf der zweidimensionalen Bildebene zugeordnet werden. Die Projektionsmatrix ergibt sich aus den intrinsischen Parametern einer Kamera.

[34] Vgl. Strobl, Sepp und S. Fuchs, 2018.
[35] Vgl. ebd.
[36] Vgl. ebd.
[37] Vgl. Strobl, Sepp und S. Fuchs, 2018.

Unity Shader

Ein Shader ist im Allgemeinen ein Programm, welches auf der besonderen Architektur eines Grafikchips ausführbar ist. Shader werden unter anderem verwendet, um aus einzelnen dreidimensionalen Geometriekoordinaten zusammenhängende Flächen zu bilden und diese Flächen in einzelne Fragmente zu unterteilen. Die Fragmente werden im Shaderprogramm eingefärbt und als Pixel auf einem Ausgabegerät angezeigt oder in einem Bild gespeichert.[38]

Diese vielen Rechenoperationen profitieren von der hocheffizienten Architektur eines Grafikchips. Dieser besteht üblicherweise aus vielen sogenannten *Shader Einheiten*. Durch eine große Anzahl an Rechenkernen können parallelisierbare Aufgaben effizient ausgeführt werden. Die Operationen, welche in den Shader Einheiten durchgeführt werden, können auf modernen Grafikchips durch Shaderprogramme definiert werden. Diese werden mit Unity in der Programmiersprache CG entwickelt.

Im folgenden Abschnitt werden die verschiedenen logischen Ebenen eines klassischen Shaderprogrammes näher erläutert.[39] Der zeitliche Ablauf dieser einzelnen Ebenen wird im folgenden *Renderpipeline* genannt. Außerdem wird der *Compute Shader* als zusätzliche Variante eines Shaderprogrammes vorgestellt.

Vertex-Shaderebene Die Ebene des Vertex Shaders ist für die Verarbeitung aller Knotenpunkte der darzustellenden Geometrie verantwortlich.[40] Die dreidimensionalen Koordinaten jedes Knotenpunktes werden zusammen mit der Projektionsmatrix und der Kameratransformation, sowie weiteren optionalen Parametern dem Shaderprogramm übergeben. In dieser Ebene des Shaderprogrammes wird üblicherweise die Projektion der dreidimensionalen Koordinaten auf die zweidimensionale Bildebene durchgeführt. Weitere Berechnungen können ebenfalls bereits in dieser Ebene vorgenommen werden.[41] In einem Durchlauf wird genau ein Kontenpunkt verarbeitet und die zweidimensionale Koordinate dieses Knotens zusammen mit weiteren optionalen Werten weitergereicht.[42]

[38] Vgl. Zucconi, 2018.
[39] Vgl. Microsoft, 2018i.
[40] Vgl. ebd.
[41] Vgl. Nischwitz u. a., 2011, S.50.
[42] Vgl. Microsoft, 2018i.

Geometry-Shaderebene In der Geometry-Shaderebene werden komplette geometrische Primitive bearbeitet. Je nach Anzahl der Knotenpunkte in einem Primitiv, werden entweder Dreiecke, Linien oder einzelne Knotenpunkte verarbeitet.[43] Der Geometry-Shaderebene werden die Informationen aller Knoten im Primitiv, welche in der Vertex-Shaderebene errechnet wurden, zusammen mit einer Kennnummer des Primitivs übergeben.[44]

Wird im Shaderprogramm keine Funktion für die Geometry-Shaderebene definiert, werden die eingehenden Werte unverändert an die nächste Ebene der Renderpipeline weitergeleitet. In einer Geometry-Shaderebene können die Werte eines Primitivs überschrieben werden. Das gesamte Primitiv kann verworfen oder in mehrere neue Primitive unterteilt werden.[45]

Rasterisierung Die in die Bildebene der Kamera projizierten Knotenpunkte bilden eine zusammengesetzte Fläche. Diese Fläche wird in einem konstanten Abstand abgetastet. In Abhängigkeit von der definierten Pixelauflösung des Renderergebnisses verändert sich dieser Abstand. Diese Fläche wird in einzelne, gleichmäßig große Flächenelemente (sogenannte Fragmente) unterteilt.[46] Dieser Prozess wird *Rasterisierung* genannt.

Ein Beispiel einer Rasterisierung ist in Abbildung 4.5 dargestellt. Die Größe eines Fragmentes entspricht der Größe eines Pixels im Renderergebnis.[47] Je höher die Auflösung des Renderergebnisses ist, desto mehr Fragmente werden rasterisiert. Jedes Fragment hat spezifische Eigenschaften, wie die Position im Bildbereich, die Entfernung des projizierten Fragmentes zur Kamera und eine Farbe.[48] Diese Werte werden für jedes Fragment eines Primitivs ausgerechnet. Dazu werden die Werte aus den Knotenpunkten des geometrischen Primitivs ausgelesen und linear interpoliert.[49] Für jeden Pixel, der von dem Primitiv abgedeckt wird,

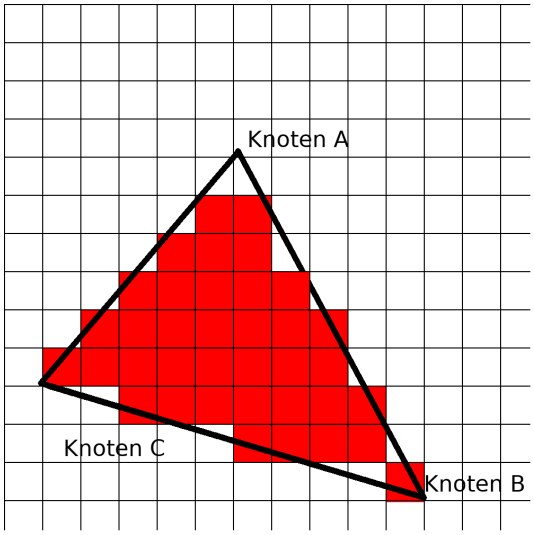


Abbildung 4.5: Rasterisierung eines Dreiecks

[43] Vgl. Microsoft, 2018c.
[44] Vgl. ebd.
[45] Vgl. ebd.
[46] Vgl. Fernando und Kilgard, 2003, S.11.
[47] Vgl. ebd., S.11.
[48] Vgl. ebd., S.11.
[49] Vgl. Fernando und Kilgard, 2003, S.11.

wird ein Fragment mit der entsprechenden Farbe erzeugt. Im weiteren Verlauf des Renderprozesses können andere Primitive diesen Farbwert überschreiben, falls sich ihr Fragment näher am Betrachter befindet.

Wie in Abbildung 4.5 zu erkennen ist, werden mit dem standardmäßigen Rasterisierungsverfahren Unitys einige Fragmente nicht generiert, obwohl Teile dieser vom Primitiv abgedeckt werden. Für jedes Fragment wird überprüft, ob die Mitte des Fragmentes innerhalb des Primitivs liegt. Ist dies nicht der Fall, wird das Fragment mitsamt Farbe verworfen und nicht angezeigt. Dies führt dazu, dass im rastersierten Bild an diesen Stellen (wie beispielsweise an Knoten A) keine Farbwerte geschrieben werden, obwohl die Fragmente teilweise angeschnitten werden.

Ein Lösungsansatz für dieses Problem ist eine sogenannte konservative Rasterisierung.[50] Ein konservativ rasterisiertes Primitiv ist in Abbildung 4.6 dargestellt. Ab der Version 11.3 der Grafikschnittstelle Direct3D kann ein bereits implementierter Algorithmus für die konservative Rasterisierung verwendet werden.

Für Grafikchips, welche diesen Schnittstellenstandard nicht erfüllen, kann ein eigener Algorithmus implementiert werden: Die Grundidee der konservativen Rasterisierung besteht in der künstlichen Vergrößerung des Primitivs für die Rasterisierung.[51] Zentriert um jeden Knotenpunkt wird eine quadratische Fläche mit den Maßen eines Pixels überlagert. Dieses Quadrat ist in der Abbildung 4.6 gelb hervorgehoben. Die Postion der Knotenpunkte wird so verändert, dass die Kanten zwischen den Knotenpunkten dieses Quadrat berühren, aber nicht schneiden.[52] Die neuen Maße des Primitivs werden rasterisiert und die ursprünglich verworfenen Fragmente (blau hervorgehoben) werden erzeugt.

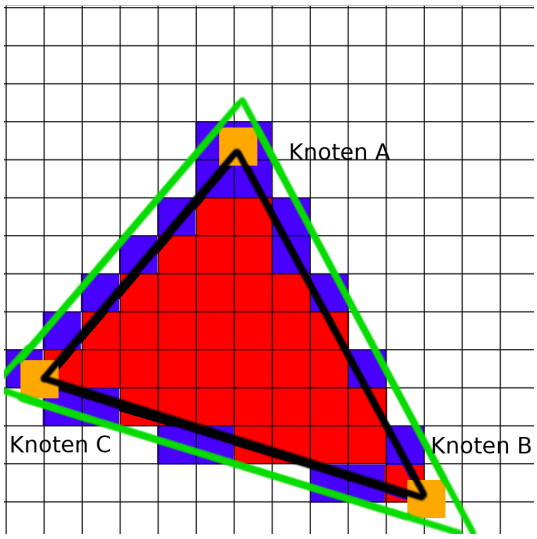


Abbildung 4.6: Konservative Rasterisierung eines Dreiecks

[50] Vgl. Story, 2014.
[51] Vgl. ebd.
[52] Vgl. Pharr und Fernando, 2005, S. 670.

Fragment-Shaderebene In der Fragment-Shaderebene wird der Farbwert eines Fragmentes auf Basis der im Vertex-und Geometry-Shaderebene errechneten und daraufhin interpolierten Werte ermittelt.[53] Farbwerte können an definierten Koordinaten aus Farbbildern ausgelesen und dem Fragment zugeordnet werden. Darüber hinaus können Lichtreflektionen anhand charakteristischer Eigenschaften der Modelloberfläche, wie etwa Unebenheiten im Material berechnet werden. Die Farbe des Fragmentes wird als Resultat dieser Berechnungen ausgegeben.

Compute Shader Die programmierbare Compute-Shaderebene unterscheidet sich von den anderen Shaderebenen. Während Vertex-, Geometry- und Fragment-Shaderebene üblicherweise für die Bildsynthese verwendet werden, dient ein Compute Shader zur Ausführung allgemeiner Berechnungen.[54] Diese Berechnungen werden parallel auf den vielen Prozessoren des Grafikchips ausgeführt. Die einzelnen Prozesse können synchronisiert werden.[55] Aus allen Prozessen kann auf einen gemeinsamen Speicherbereich zugegriffen werden.[56] Die Ergebnisse der Rechenoperationen können in eine Textur oder in ein Byte-Array geschrieben werden.

Texture Mapping

Texture Mapping ist eine Methode, um den Detailgrad einer Modelloberfläche zu erhöhen. Dazu werden Daten aus einem ein- oder mehrdimensionalen Speicherbereich, einer sogenannten *Textur* verwendet. Wie in Abbildung 4.7 gezeigt, kann durch eine Textur die Modelloberfläche mit alternativen Farben oder Erscheinungsmerkmalen angereichert werden.

[53] Vgl. Microsoft, 2018i.
[54] Vgl. Microsoft, 2018a.
[55] Vgl. ebd.
[56] Vgl. Microsoft, 2018a.

(a) Modell ohne Textur *(b)* Texturiertes Modell

Abbildung 4.7: Texture Mapping anhand eines dreidimensionalen Würfels

Bei der Darstellung des Würfels wird für jedes Fragment der hinterlegte Farbwert, an einer *Texturkoordinate* (auch *UV-Koordinate* genannt) ausgelesen. Der Wert eines Pixels der Textur (auch *Texel* genannt) wird der Fragment-Shaderebene übergeben und beeinflusst die resultierende Farbausgabe.

Es ist möglich, in einer Textur arbiträre Datenstrukturen binär zu speichern. Neben der Grundfarbe, können somit auch weitere Werte, wie etwa der Glanz einer Oberfläche gespeichert und im Fragmentshader verwendet werden.

Um während des Renderings eines Primitivs Werte aus einer Textur auszulesen, wird jedem Knotenpunkt mindestens eine UV-Koordinate zugewiesen. UV-Koordinaten statischer Modelle können bereits während der Modellierung generiert werden.[57] Auch eine dynamische Berechnung der UV-Koordinaten während der Laufzeit ist möglich.[58] Dieser Ansatz benötigt jedoch in Abhängigkeit des gewählten Algorithmus und der Anzahl der Primitive zusätzliche Rechenzeit.

Ein UV-Mapping eines Quaders wird in Abbildung 4.8 beispielhaft gezeigt.

Jedem Knotenpunkt des Modells ist mindestens eine UV Koordinate zugewiesen (rot hervorgehoben). Modelloberflächen werden jeweils durch mehrere Knotenpunkte definiert. Die Bildinformationen einer Modelloberfläche werden in der Textur (links) zwischen

[57] Vgl. Lengyel, 2011, S. 165.
[58] Vgl. Lengyel, 2011, S. 165.

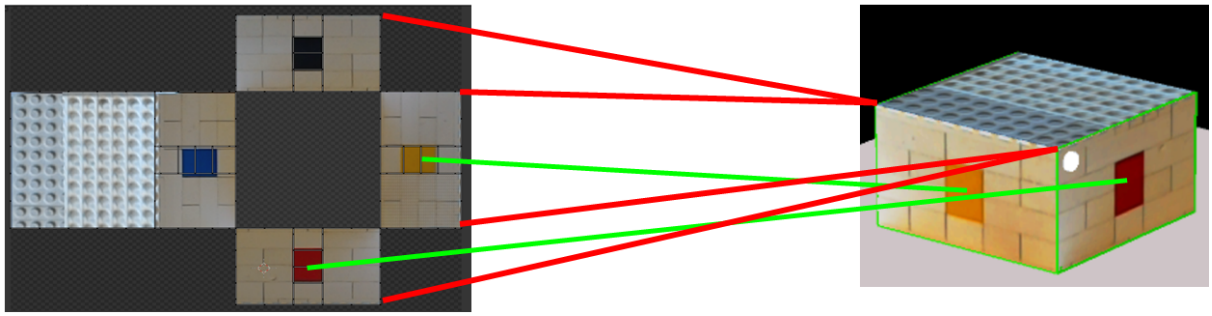


Abbildung 4.8: Beziehung zwischen dreidimensionalen Knotenpunkten und zweidimensionalen UV-Koordinaten

den UV-Koordinaten der Knotenpunkte gespeichert. Die Koordinaten der Knotenpunkte werden während der Rasterisierung interpoliert. So kann für jedes Fragment in der Fragment-Shaderebene der Wert an der interpolierten Koordinate aus der Textur ausgelesen werden (grün hervorgehoben).[59] Die Generierung von UV-Koordinaten wird *UV-Mapping* genannt.[60]

Im folgenden Abschnitt werden zwei spezielle Unity-spezifische Texturarten näher betrachtet.

Render Texturen Eine Render Textur unterscheidet sich durch ihre dynamischen Bildinhalte von gewöhnlichen zweidimensionalen Texturen. Während der Laufzeit können Texturinhalte effizient aktualisiert werden.[61] Standardmäßig werden dabei die Bildinformationen in einem abgegrenzten Speicherbereich des Grafikchips, dem Videohauptspeicher, abgelegt. Während der Bildsynthese kann ein Shaderprogamm sowohl lesend, als auch schreibend auf die Render Textur zugreifen.[62] Gerenderte Bilder einer virtuellen Kamera können statt auf dem Ausgabegerät in einer solchen Textur gespeichert werden.[63] Ein direkter Zugriff außerhalb eines Shaderprogrammes auf die Bildinformationen ist jedoch nicht möglich. Um eine Render Textur CPU seitig auszulesen, müssen die Bildinhalte aus dem Speicher des Grafikchips in den Hauptspeicher geladen werden.

[59] Vgl. ebd., S. 165.
[60] Vgl. Autodesk, 2014.
[61] Vgl. Unity, 2018c.
[62] Vgl. ebd.
[63] Vgl. Unity, 2018c.

Virtuelle Texturen Je nach Auflösung und Format der Textur variiert der Speicherbedarf. Eine Textur mit einer Auflösung von 4096 x 4096 Pixels und 32 Bit Farbinformation pro Pixel benötigt bereits 64 Megabyte Speicherplatz. Mit einer höheren Auflösung der Texturen steigt der Speicherbedarf quadratisch an. Die Größe des Grafikspeichers ist limitiert. Daher ist es möglich, dass besonders hoch aufgelöste Texturen (sogenannte *mega-textures*) nicht vollständig in den Grafikspeicher geladen werden können.

Sogenannte *Sparse Textures* ermöglichen das partielle Laden einer großen Textur in den Videohauptspeicher.[64] Die große Textur wird dazu von Unity automatisch in kleinere Segmente unterteilt. In der Regel werden dabei ausschließlich die Bereiche einer Textur geladen, welche zum aktuellen Zeitpunkt im Sichtbereich der Kamera liegen.[65]

Unity setzt für die Verwendung von Sparse Textures einen Grafikchip voraus, welcher die Programmierung über die Programmierschnittstelle Direct3D in der Version 11.2 unterstützt.[66] Da die HoloLens diesen Standard nicht erfüllt, können virtuelle Texturen nicht verwendet werden.

4.2.3 Netzwerkkommunikation

Um arbiträre Daten zwischen Anwendungen auf mehrerer HoloLenses auszutauschen, müssen diese über Netzwerk kommunizieren können. Im folgenden Abschnitt werden zwei Ansätze der Netzwerkkommunikation im Kontext der HoloLens und Unity näher betrachtet.

Sharing Service

Der sogenannte *Sharing Service* ist ein von Microsoft bereitgestellter Dienst, welcher auf einem zentralen Windows System ausgeführt wird. Dieser Dienst verwaltet die Kommunikation mehrerer Mixed Reality Anwendungen.[67] Diese können sowohl simple, als auch komplexe Datenstrukturen an andere Teilnehmer einer Sitzung versenden. Neben der Möglichkeit die Stimme des Benutzer an andere Teilnehmer zu übertragen, können

[64] Vgl. Unity, 2018d.
[65] Vgl. ebd.
[66] Vgl. ebd.
[67] Vgl. Microsoft, 2018k.

identische virtuelle Inhalte auf mehreren Anwendungen synchronisiert angezeigt werden.[68] Der Zustand dieser virtuellen Objekte kann synchronisiert werden. Basierend auf einer Client-Server-Architektur verteilt der Sharing Service die Nachrichten an die Teilnehmer.

Unity Networking

Für die Entwicklung verbundener Anwendungen werden in Unity mehrere Softwarekomponenten bereitgestellt. Da die Unity Engine primär für Videospiele eingesetzt wird, fokussieren sich diese Komponenten auf Mehrspieler-Szenarios. Eine Anwendung übernimmt üblicherweise die Rolle des *Hosts* und ist für die Verteilung der Daten an andere Teilnehmer (*Clients*) verantwortlich.

Um die Kommunikation zwischen den Teilnehmern einer Sitzung zu gewährleisten, können zwei unterschiedliche Schnittstellen verwendet werden: der *NetworkTransportAPI* (im Folgenden LLAPI genannt) und der *High Level API* (kurz HLAPI).[69]

LLAPI: Die LLAPI Schnittstelle verwaltet einen betriebssystemnahen Zugriff einer Unity Anwendung auf Netzwerkressourcen des Rechners (geringe Abstrahierung).[70] Dadurch können Entwickler einzelne Einstellungen der Netzwerkkommunikation ihren spezifischen Anforderungen anpassen. Die Daten werden als Byte-Array mithilfe des *User Datagram Protocol* (auch UDP genannt) übertragen.[71]

HLAPI: Die HLAPI basiert auf den Komponenten der LLAPI und abstrahiert diese. Sie ist somit im Vergleich zur LLAPI anwendungsnäher. Auch komplex aufgebaute Unity Datentypen können ohne weitere Verarbeitung versendet werden.[72] Funktionsaufrufe auf anderen Teilnehmern einer Sitzung (*Remote Procedure Calls*) durch den Host werden unterstützt. Ebenso lassen sich auch Unity Events anwendungsübergreifend verwenden.[73]

[68] Vgl. ebd.
[69] Vgl. Unity, 2018b.
[70] Vgl. ebd.
[71] Vgl. ebd.
[72] Vgl. ebd.
[73] Vgl. Unity, 2018b.

4.3 Externe Bibliotheken

Mithilfe von externen Bibliotheken ist es möglich, implementierte Softwarebausteine für eigene Anwendungen zu nutzen. Im folgenden Abschnitt werden Bibliotheken vorgestellt, welche verwendet werden, um sowohl die Entwicklung grundlegender Funktionen, als auch die Ausführung einer Mixed Reality Anwendung zu beschleunigen. Außerdem wird eine Bibliothek vorgestellt, welche den Zugriff auf einzelne Videobilder der HoloLens ermöglicht.

4.3.1 Microsoft Mixed Reality Toolkit

Das Mixed Reality Toolkit ist eine von Microsoft veröffentliche Zusammenstellung von Softwarekomponenten. Diese wird eingesetzt, um die Entwicklung von Mixed Reality Anwendungen zu vereinfachen.[74] Allgemeine Aufgaben von Mixed Reality Anwendung werden durch implementierte Lösungsbausteine vereinfacht und anhand von Beispielen den Entwicklern demonstriert.

Angepasste Komponenten für die Ein- und Ausgabe von Anwendungen, sowie zur gemeinsamen Betrachtung virtueller Inhalte über mehrere HoloLenses werden zur Verfügung gestellt. Darüber hinaus beinhaltet das Toolkit weitere Bausteine, die das Kompilieren der Anwendung, sowie die Einrichtung virtueller Szenen vereinfachen.[75]

Diese Komponenten können sowohl für die Entwicklung von Unity basierten, als auch für C++ basierte Anwendungen verwendet werden.[76] Die Bibliothek ist von Microsoft unter der MIT Lizenz veröffentlicht worden.[77]

4.3.2 HoloLensCameraStream

Das Unity Plugin *HoloLensCameraStream* ermöglicht Unity Anwendungen den Zugriff auf einzelne Videobilder der HoloLens Farbkamera.[78] Intrisische und extrinsische Parameter

[74] Vgl. Microsoft, 2018g.
[75] Vgl. ebd.
[76] Vgl. Ong, 2017.
[77] Siehe: `https://github.com/Microsoft/MixedRealityToolkit-Unity/blob/master/License.txt`
[78] Vgl. Vulcan, 2018.

der Farbkamera werden zum Zeitpunkt einer Bildaufnahme gespeichert und können in Programmklassen der Anwendung verwendet werden.[79] Das Plugin wurde durch die Firma Vulcan im Rahmen der Apache 2.0 Lizenz veröffentlicht. [80]

4.3.3 UnityOctree

UnityOctree ist ein Unity Plugin und wurde von dem Entwickler *Nition* entworfen.[81] Das Plugin ist eine konkrete Implementierung einer Beschleunigungsstruktur namens *Octree* für Unity.

Ein klassischer Octree ist eine hierarchisch aufgebaute Datenstruktur in Form eines Baumes und dient der Speicherung und effizienten Abfrage von Geometrien. Eine Octree-Struktur kann verwendet werden, um dreidimensionale Knotenpunkte oder Begrenzungsbereiche von Geometrien entsprechend ihrer Position und Maße räumlich zu sortieren.[82] Der initiale Knoten in einem klassischen Octree wird Wurzel genannt und ist ein Würfel.[83] Jeder Knoten im Octree besitzt per Definition entweder acht Kind-Knoten oder keine.[84]

Mit jeder weiteren Unterteilung eines Knotens halbiert sich die Größe entlang aller Achsen. Die räumliche Auflösung des Octrees verachtfacht sich.

In Abbildung 4.9 ist das zweidimensionale Pendant eines Octrees, ein sogenannter *Quadtree*, abgebildet. Knoten dieses Quadtrees besitzen per Definition entweder vier Kind-Knoten oder keine. Octree und Quadtree basieren auf der gleichen Grundidee:

Ein Quadtree besteht ausschließlich aus einem Wurzelknoten, wenn keine Geometrien in diesem

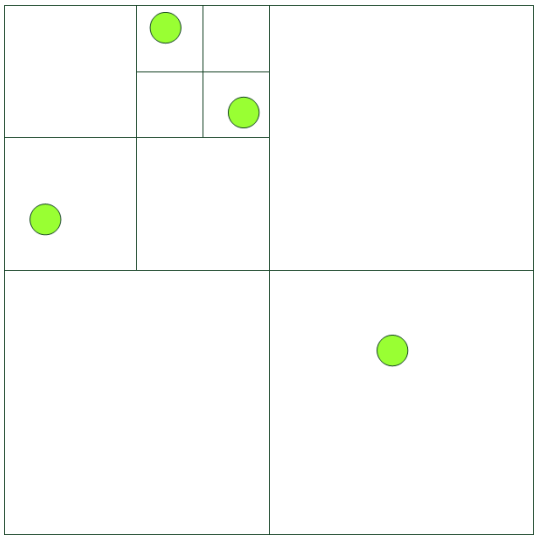


Abbildung 4.9: Aufbau eines Quadtrees mit vier Elementen

[79] Vgl. ebd.
[80] Siehe `https://github.com/VulcanTechnologies/HoloLensCameraStream/blob/master/LICENSE`
[81] Vgl. Nition, 2018.
[82] Vgl. Pharr und Fernando, 2005, S. 597.
[83] Vgl. ebd., S. 597.
[84] Vgl. Pharr und Fernando, 2005, S. 597.

gespeichert sind. Ein Knoten eines klassischen Quadtrees teilt sich, sobald mehrere Elemente innerhalb dessen Volumens referenziert sind. Innerhalb eines Quadranten befindet sich im klassischen Quadtree maximal ein Element (in Abbildung 4.9 grün markiert).

Diese hierarchische Ordnung ermöglicht eine rekursive Abfrage, um etwa die Sichtbarkeit der Geometrien aus der Perspektive der virtuellen Kamera zu testen. Falls der Bereich eines Eltern-Quadranten nicht sichtbar ist, sind auch die Elemente innerhalb der Kind-Knoten nicht sichtbar und müssen nicht einzeln getestet werden.[85] Dies kann zu einer Minderung der benötigten Rechenleistung führen.

Das Unity Plugin bietet Funktionen zur Erstellung und Verwaltung eines Octrees, sowie zur Durchführung von Sichtbarkeitstests der im Octree referenzierten Elemente.[86] Arbiträre Datentypen können an ein Volumen oder Knotenpunkt gebunden im Octree hinterlegt werden.[87]

Das Plugin wurde unter der 2-Klausel BSD Lizenz veröffentlicht[88]

[85] Vgl. Lengyel, 2011, S. 231.
[86] Vgl. Nition, 2018.
[87] Vgl. Nition, 2018.
[88] Siehe `https://github.com/Nition/UnityOctree/blob/master/LICENCE`

5 Implementierung

Um die HoloLens als Ein- und Ausgabegerät zu verwenden, wird eine auf der HoloLens lauffähige Anwendung entwickelt. Diese Anwendung basiert auf der Unity Engine. Im Rahmen dieser Arbeit wird dabei ausschließlich die Unity Version 2017.3.1 verwendet. In dem folgenden Kapitel werden einzelne Kernkonzepte der Implementierung näher erläutert.

5.1 3D-Rekonstruktion

Im folgenden Abschnitt wird beschrieben, wie von der HoloLens bereitgestellte 3D-Rekonstruktion der Umgebung geladen und der Unity Szene hinzugefügt werden kann. Das erzeugte Dreiecksmodell wird kritisch bezüglich seiner Auflösung, der Datenmenge und der Größe der Dreiecke untersucht. Darüber hinaus wird sowohl die Bildwiederholrate, als auch die Speicherauslastung in Abhängigkeit von der Größe der Rekonstruktion untersucht und im Rahmen der Anwendung optimiert. Es wird ein Verfahren zur Erstellung von Texturkoordinaten in Echtzeit vorgestellt, um eine Texturierung der rekonstruierten Umgebung zu ermöglichen. Sowohl Vor- als auch Nachteile dieses Verfahrens werden näher erläutert.

5.1.1 Abfrage der Rekonstuktionsdaten

Digitale Gegenstände der Unity Anwendung sollen mit Objekten der realen Umgebung interagieren können. Daher wird ein virtuelles Abbild der Umgebung erfasst und als Modell der Unity Szene hinzugefügt. Sowohl Unity, als auch Microsoft stellen verschiedene Softwareschnittstellen bereit, um Anwendungen mithilfe der Tiefensensordaten eine Echtzeitrekonstruktion der Umgebung zur Verfügung zu stellen.

Spatial Mapping Renderer und Spatial Mapping Collider

Der Spatial Mapping Renderer und der Spatial Mapping Collider sind abstrahierte Komponenten. Grundlegende Eigenschaften dieser Komponenten lassen sich durch Variablen oder direkt über die Benutzeroberfläche des Unity Editors konfigurieren.

Sowohl die Aktualisierungsrate, als auch die Qualitätsstufe des generierten Dreiecksmodells kann eingestellt werden. Ist eine Abstufung von hoher, mittlerer und geringer Qualität vorgegeben. Messbare Aussagen über diese Qualitätsabstufungen existieren jedoch nicht.

Darüber hinaus lassen sich die Form und die Größe des Bereiches definieren, in dem die reale Umgebung erfasst werden sollen. Obwohl die Sensoren der HoloLens nur in einem kegelförmigen Erfassungsbereich ihre Umgebung registrieren, kann zwischen einem kugelförmigen Volumen oder einem Quader ausgewählt werden.

Die Angabe eines Eltern Spielobjekts ist notwendig, um die generierten Dreiecksmodelle in die Szene zu laden. Um die Darstellung der Raumrekonstruktion zu verändern, kann dem Spatial Mapping Manager ein Material übergeben werden. Dieses Material wird für die Visualisierung der rekonstruierten Blöcke verwendet.

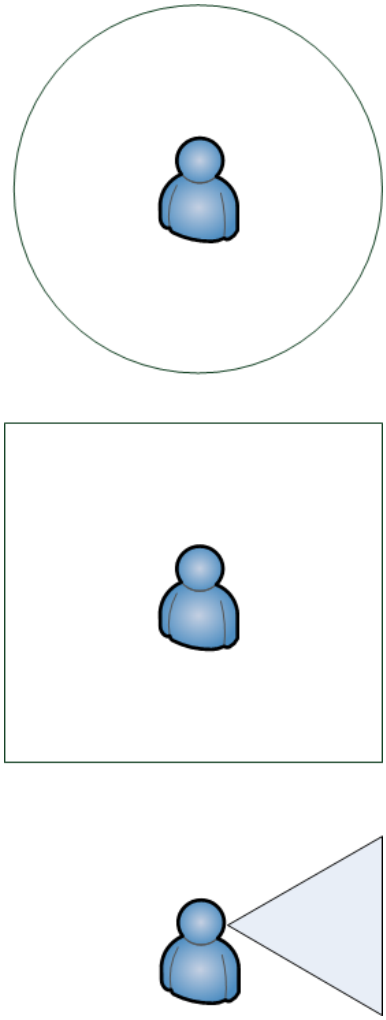


Abbildung 5.1: Kugel-, Quader- und Kegelförmiger Erfassungsbereich

Surface Observer

Sowohl der Spatial Mapping Renderer, als auch der Spatial Mapping Collider basieren auf einer Klasse namens *SurfaceObserver* und abstrahieren dessen Funktionalitäten.[89]

Die Schnittstelle des SurfaceObservers bietet präzisere Konfigurationsmöglichkeiten und mehr Kontrolle über den Ablauf der Rekonstruktion. Die Rekonstruktion durch den SurfaceObserver ist ein zweistufiger Prozess.[90] Durch Aufruf der *Update* Methode wird überprüft, ob sich die Umgebung in einem definierten Erfassungsbereich verändert hat.

Sobald eine Veränderung im Raum gemessen wird, erfolgt ein Aufruf einer vordefinierte Methode. Dieser Methode werden Informationen bezüglich der Art der Änderung, die Position des aktualisierten Blocks und eine Identifikationsnummer dieses Blocks übergeben.

[89] Vgl. Unity, 2018e.
[90] Microsoft, 2018l.

Anhand der Parameter lässt sich feststellen, ob ein zuvor unbekannter Abschnitt der Umgebung erfasst, ein bestehender Teil aktualisiert oder ein Teil der Umgebung entfernt wurde.

Um das Dreiecksmodell des Raumes zu erstellen, wird von der SurfaceObserver Schnittstelle die Methode *RequestMeshAsync* bereitgestellt. Dieser Methode werden mehrere Parameter übergeben: Die gewünschte Auflösung des Dreiecksmodells, die Identifikationsnummer des Blockes im Raum und einen Datencontainer, in welchem das Dreiecksmodell gespeichert wird. Die Bereitstellung des gewünschten Blocks wird asynchron ausgeführt.[91] Dieser Prozess kann mehrere Bildwiederholungen dauern.

Sobald das Dreiecksmodell fertig generiert ist, kann dieses der Unity Szene hinzugefügt und verwendet werden.

Research Mode

Seit der Veröffentlichung von Windows 10 RS4 im April 2018 wird ein direkter Zugriff auf die Tiefenkamera aus einer Anwendung ermöglicht.[92] Die Tiefendaten werden in einem kontinuierlichen Datenstrom der Anwendung zur Verfügung gestellt. Diese Tiefeninformationen mehrerer Bilder müssen zunächst in ein Dreiecksmodell umgewandelt werden, um sie einer Unity Anwendung bereitstellen zu können. Da diese Umwandlung jedoch nicht der Fokus dieses Projektes ist, wurde der Research Mode nicht näher betrachtet. Stattdessen wurde sich für die Nutzung der SurfaceObserver Schnittstelle entschieden, da nur diese die notwendige Flexibilität sowohl in der Konfiguration, als auch in der Verarbeitung der Daten bietet.

Eine eigene Komponente *SurfaceObserverManager* verwaltet den Zugriff auf die SurfaceObserver Schnittstelle und verarbeitet die einzelnen Blöcke der Raumrekonstruktion. Während der Initialisierung der Anwendung wird zunächst die SurfaceObserver Schnittstelle eingerichtet. In einem variablen Zeitintervall wird anschließend die *Update* Methode des SurfaceObservers aufgerufen,um Veränderungen in der Umgebung zu erfassen.

[91] Ebd.
[92] Vgl Microsoft, 2018e.

Da sich der Benutzer im Lauf der Anwendung fortbewegt, muss sich der Erfassungsbereich der Schnittstelle daran anpassen. Ausschließlich Änderungen im aktuellen Sichtfeld des Benutzers sollen erfasst werden. Obwohl eine Beschränkung des Erfassungsbereiches auf den Sichtbereich von der Schnittstelle unterstützt wird, war diese Funktionalität jedoch fehlerhaft und nicht zuverlässig. Stattdessen werden die Änderungen in einem quaderförmigen Volumen vor dem Benutzer erfasst. Der Erfassungsbereich ist in Abbildung 5.2 grün hervorgehoben.

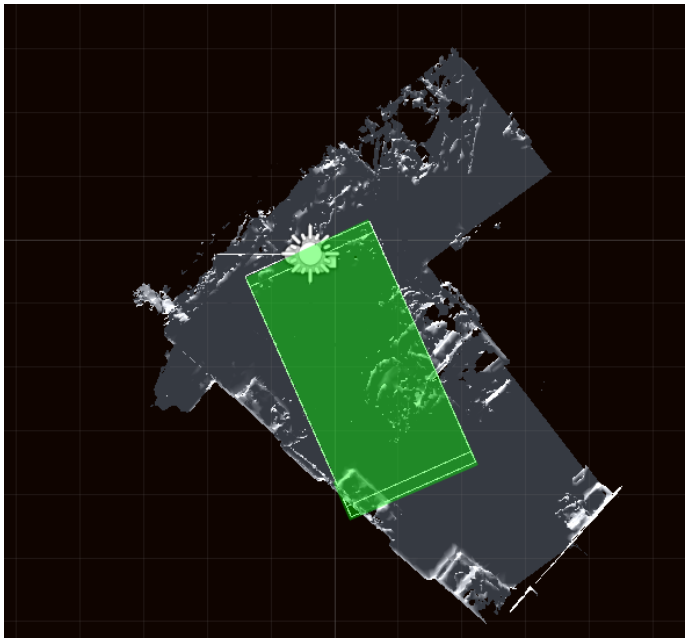


Abbildung 5.2: Erfassungsbereich des SurfaceObservers vor dem Benutzer

Der SurfaceObserverManager reagiert auf die drei unterschiedlichen Ereignisse (dargestellt in Abb. 5.3) des SurfaceObservers: die Erfassung eines neuen Blocks, die Aktualisierung eines bestehenden Blocks und die Entfernung eines Blocks.

Sobald der SurfaceObserver einen neuen Block erfasst, wird ein neues Spielobjekt instanziiert. Ein neues Material mit einer weißen Textur erstellt, um die Darstellung des Blocks zu initiieren.

Anschließend wird die Methode *RequestMeshAsync* aufgerufen, welche das entsprechende Dreiecksmodell generiert und in dem Spielobjekt speichert. Nachdem ein Block erstellt wurde, wird sein Spielobjekt zusammen mit der Identifikationsnummer in einer Tabelle registriert. Somit kann mithilfe der Kennnummer des Blockes das dazugehörige Spielobjekt identifiziert werden.

Um die Dreiecksmodelle eines bestehenden Blockes zu aktualisieren, muss kein neues Spielobjekt erstellt werden. Über die Referenz kann das bereits erstellte Spielobjekt mit dem veralteten Dreiecksmodell aus der Tabelle geladen werden. Mithilfe der Methode *RequestMeshAsync* wird das aktuelle Dreiecksmodell geladen, welches das veraltete Modell überschreibt.

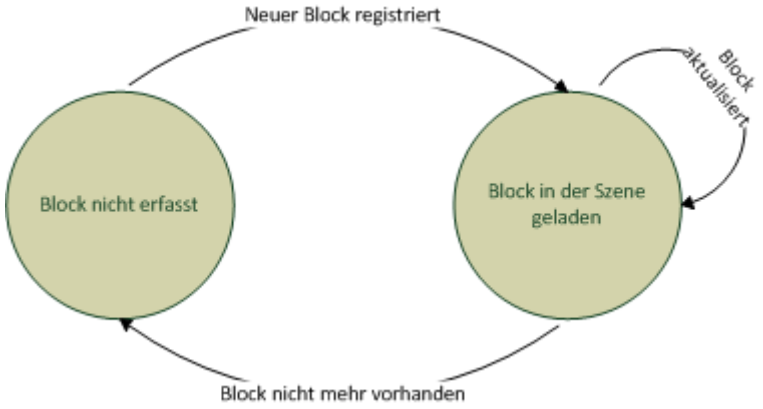


Abbildung 5.3: Lebenzyklus eines Blockes der Rekonstruktion

5.1.2 Parametrisierung des Dreiecksmodells

Die Umgebung des Benutzers wird blockweise der Unity Szene hinzugefügt. Im folgenden Abschnitt werden die Eigenschaften eines solchen Blocks und des darin enthaltenen Dreiecksmodell näher untersucht.

Auflösung: Während der Umgebungsrekonstruktion haben mehrere Faktoren Einfluss auf die Qualität des Dreiecksmodells. Neben den bereits vorgestellten Umweltfaktoren, wird auch softwareseitig die Qualität der Rekonstruktion beeinflusst.

Der SurfaceObserver Schnittstelle wird in der Methode *RequestMeshAsync* die maximale Anzahl an 3D-Knotenpunkten pro Kubikmeter übergeben. Dieser Wert ist nur variierender Knotenpunktdichte aussagekräftig, da nicht garantiert wird, dass diese Knotenpunktdichte im Dreiecksmodell tatsächlich erreicht wird. Je geringer die Knotendichte ist, desto weniger Speicher- und Rechenleistung wird für zur Darstellung des Dreiecksmodells benötigt. Gleichzeitig sinkt allerdings auch der Detailgrad des Modells. Um ein möglichst genaues Abbild der Umgebung zu erstellen, wird in der offiziellen Unity Dokumentation eine Knotendichte von mindestens 1000 Knoten pro Kubikmeter empfohlen.[93]

Topologie der Dreiecke: Jedes Dreiecksmodell besteht aus einer Liste von dreidimensionalen Knotenpunkten und einer Liste mit Indizes, welche auf diese Knotenpunkte verweisen. Jeweils drei aufeinanderfolgende Indizes in der Liste definieren den Aufbau

[93]Vgl. Unity, 2018f.

eines Dreiecks. Im folgenden Abschnitt wird anhand der Abbildung 5.4 der interne Aufbau des Dreiecksmodells näher betrachtet.

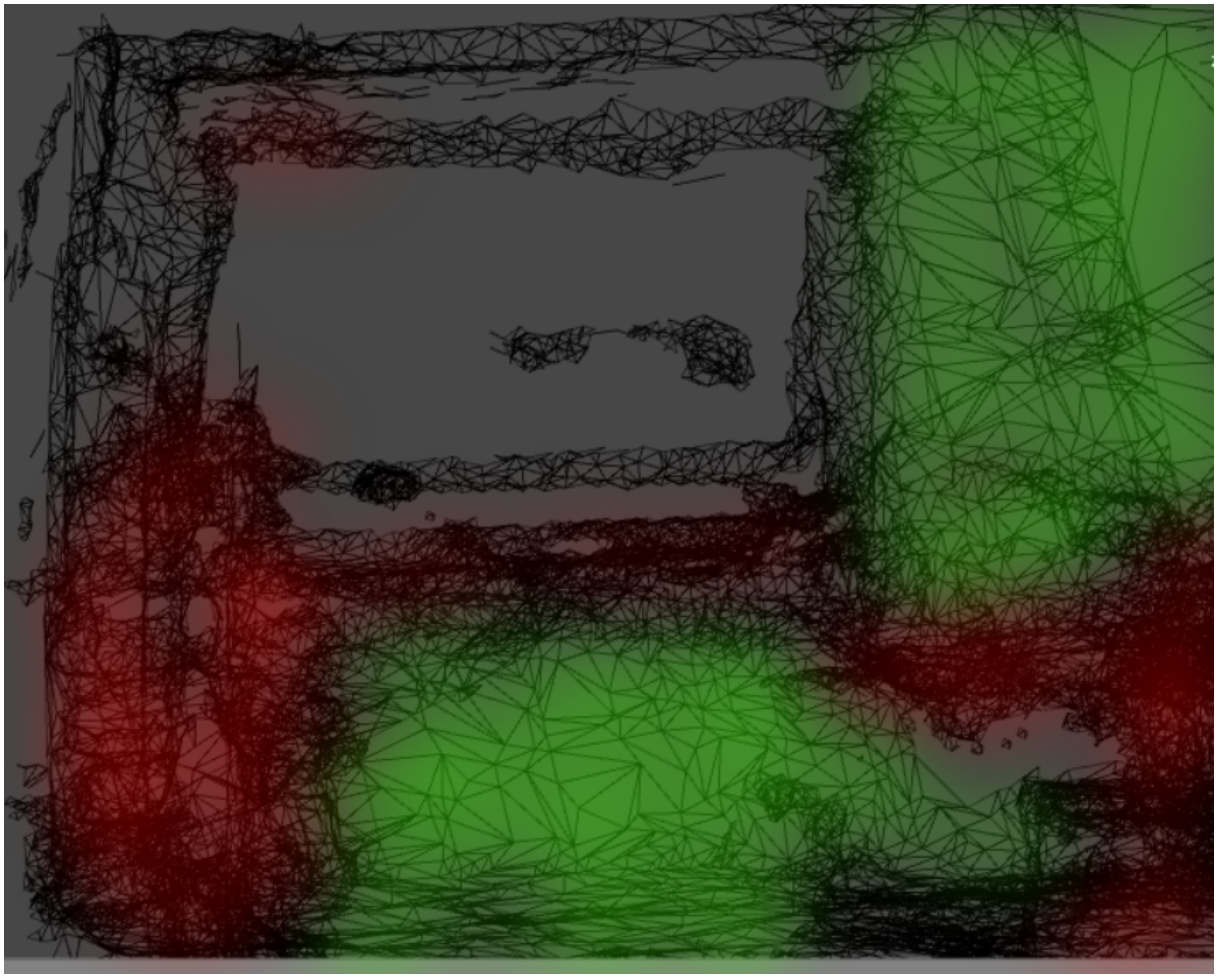


Abbildung 5.4: Dreiecksmodell mit variierender Knotenpunktdichte

Knotenpunktdichte: Ebene Oberflächen eines Raumes (grün markiert), wie etwa Wände oder der Boden, zeichnen sich durch große Flächen aus. Eine Ebene lässt sich bereits mit drei Knotenpunkten aufbauen. Werden nun von den HoloLens Sensoren mehrere Punkte einer Oberfläche in einer Ebene erfasst, können diese in einer Ebene zusammengefasst und die Anzahl der Knotenpunkte reduziert werden. Im Gegensatz dazu ist die Dichte der Knotenpunkte an Kanten oder Unebenheiten (rot markiert) deutlich höher, um den Detailgrad zu erhöhen.

Größe der Dreiecksflächen: Aufgrund der inhomogenen Knotendichte variiert auch die Fläche der einzelnen Dreiecke. In Bereichen mit einer hohen Knotendichte, ist die Entfernung zwischen diesen Punkten geringer und somit auch die einzelnen Dreiecksfläche kleiner. Bei einem Block mit insgesamt 98012 Knotenpunkten und einer maximalen Knotenpunktdichte von 5000 Knotenpunken pro Kubikmeter variieren die Flächeninhalte zwischen $5 * 10^{-9} m^2$ und $0,06 m^2$. Im Durchschnitt ergab sich ein Flächeninhalt von $7,35 * 10^{-5} m^2$.

UV-Koordinaten und Oberflächennormalen: UV-Koordinaten und Oberflächennormalen sind weitere Parameter, die optional pro Knotenpunkt definiert werden können. Die Oberflächennormale ist der Vektor, welcher senkrecht zu der Dreiecksfläche steht. Dieser Vektor wird üblicherweise für die Beleuchtungsberechnung an einer Oberfläche verwendet.

Durch eine UV-Koordinate wird ein dreidimensionaler Knotenpunkt auf einen zweidimensionalen Bildbereich abgebildet. Die Koordinaten können verwendet werden, um dreidimensionalen Knotenpunkten Werten aus einer zweidimensionalen Textur zuzuordnen.

Die von der HoloLens bereitgestellten Dreiecksmodell enthalten weder Oberflächennormalen, noch UV-Koordinaten. Während sich die Normalen anhand der Knotenpunkte neu errechnen lassen, stellt die Generierung von UV-Koordinaten eine komplexe Aufgabe dar. Diese Problematik und mögliche Lösungsansätze werden in Abschnitt 5.1.4 näher erläutert.

5.1.3 Optimierung der Rechen- und Speicherauslastung

Für eine allgemeine Umgebungserfassung sollen keine Annahmen über die Ausmaße der zu rekonstruierenden Umgebung gemacht werden. Größere Umgebungen, wie etwa ganzer Werkshallen, sollen ebenso rekonstruiert werden, wie kleinere Bauteile. Dies ist besonders in Hinblick auf die begrenzten Ressourcen der HoloLens eine Herausforderung. Sowohl Hauptspeicher, als auch Rechenkapazitäten sind daher effizient zu verwenden.

Um potentielle Optimierungsmöglichkeiten herauszuarbeiten, wurde zunächst die durchschnittliche Bildwiederholrate in Abhängigkeit der sichtbaren 3D Kontenpunkte gemessen.

Als Testobjekt wurden nacheinander Kugeln mit unterschiedlich vielen Knotenpunkten der Szene hinzugefügt. Die Kugeln wurde in Relation zur HoloLens so platziert, dass diese den Bildausschnitt maximal ausfüllen. Insgesamt wurde der Test mit sechs verschieden Kugeln durchgeführt. Die Gesamtanzahl der Knotenpunkte wurde hierbei in einem Bereich von 54192 bis 216768 verändert. Drei verschiedene Shaderprogrammen wurden getestet: der Standard Unity Shader, der für die HoloLens angepasste Vertex Lit Shader des Mixed Reality Toolkits und ein simpler Gourand Shader[94]. Zur Messung der Bildwiederholrate wurde das Projekt im Release-Modus kompiliert und auf die HoloLens übertragen. Mittels des in Visual Studio integrierten Grafikdebugger wurde die Bildwiederholrate über 3600 Bildberechnungen hinweg gemessen. Das arithmetische Mittel der Bildwiederholrate ist in Diagramm 5.5 abgebildet.

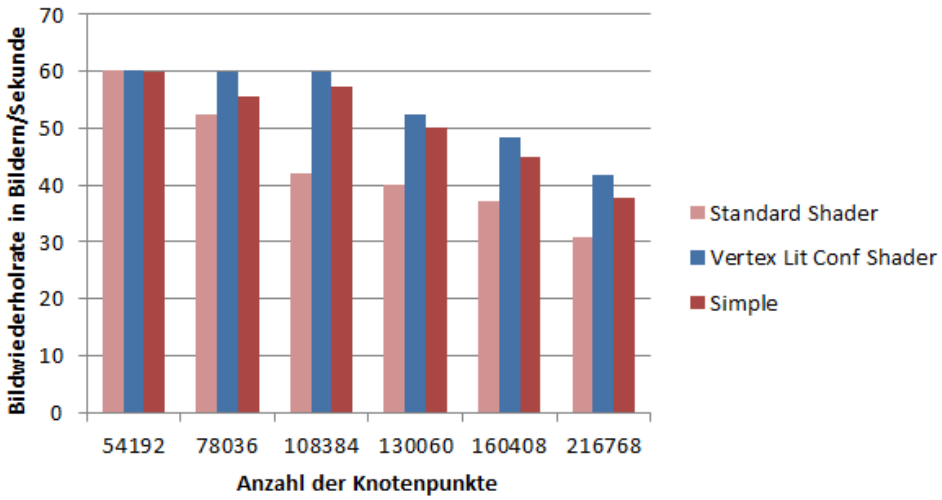


Abbildung 5.5: Mittlere Bildwiederholrate in Abhängigkeit von der Anzahl der Knotenpunkte und dem verwendeten Shader

Die durchschnittliche Bildwiederholrate nimmt mit zunehmender Anzahl der Knotenpunkte ab, da die benötigte Rechenzeit pro Bild zunimmt. Mit dem standardmäßig eingestellten Shader von Unity wurde die geringste Bildwiederholrate im Vergleich zu den zwei Alternativen gemessen. Die durchschnittliche Rate des Vertex Lit Shader ist im Schnitt 7,35 % höher, als die des simplen Gourand Shaders Shaderprogrammes. Insgesamt lässt sich erkennen, dass die Anzahl der sichtbaren Knotenpunkte nicht größer als 100.000 sein sollte, da sonst keine stabile Bildwiederholrate von 60 Bildern pro Sekunde erreicht wird.

Anhand dieser Messung wird deutlich, dass die HoloLens nur eine begrenzte Anzahl an

[94]Gouraud, 1971, Vgl.

Primitiven gleichzeitig darstellen kann. Ein einzelner Block kann allerdings aus mehr als 100.000 Dreiecken bestehen und mehrere Megabyte Speicherplatz reservieren. Da der HoloLens nur eine limitierte Hauptspeicherkapazität zur Verfügung steht, ist es nicht möglich, beliebig viele Blöcke gleichzeitig zu erfassen. Dieses steht jedoch im Konflikt mit der Voraussetzung, dass beliebig große Umgebungen erfasst und angezeigt werden können.

Um diese Begrenzung einzuhalten, werden die Blöcke auf dem Flash-Speicher ausgelagert, die für den Benutzer nicht sichtbar sind. Blöcke, welche sich hinter oder direkt neben dem Benutzer befinden, werden nicht angezeigt und haben keinen Einfluss auf den Rest der Szene. Diese Blöcke belegen unnötig Speicher- und Rechenkapazität und werden daher ausgelagert. So kann der gewonnene Speicherplatz anderweitig verwendet werden.

Pro Bildwiederholung muss die Sichtbarkeit jedes Blockes überprüft werden. Dreiecksmodelle, die das Sichtfeld verlassen, werden auf den Flash-Speicher ausgelagert. Damit festgestellt werden kann, ob ein Block sichtbar ist, wird die Position und Größe des Blocks in Form einer sogenannten *Bounding Box* gespeichert. Eine *Bounding Box* ist ein Quader, welcher mit minimalen Ausmaßen das gesamte Objekt umhüllt. Anhand dieser *Bounding Box* kann auch überprüft werden, ob ausgelagerte Blöcke in das Sichtfeld gelangen. In diesem Fall wird das entsprechende Dreiecksmodel von dem Flash-Speicher gelesen und dem entsprechenden Spielobjekt erneut zugewiesen.

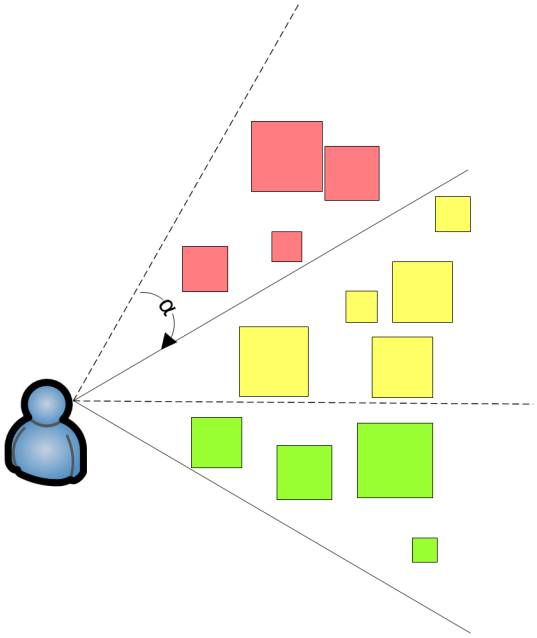


Abbildung 5.6: Veränderung der sichtbaren Objekte im Bildbereich

In Abbildung 5.6 wird dieser Prozess dargestellt. Der sichtbare Bereich während der aktuellen Bildberechnung ist als durchgezogene Linie darstellt, der Sichtbereich des vorherigen Bildes als gestrichelte Linie. Innerhalb einer Bildberechnung hat sich die Blickrichtung um einen Winkel α verändert, sodass einige Blöcke (rot hervorgehoben) ausgelagert werden müssen. Andere Blöcke verbleiben im Speicher (gelb hervorgehoben) und neue Blöcke müssen in den Speicher geladen werden (grün hervorgehoben).

Octree

Eine mögliche, naive Implementierung überprüft innerhalb einer Bildwiederholung für jeden Block, ob dessen *Bounding Box* sichtbar ist. Der Aufwand dieses Algorithmus skaliert mit der Anzahl der insgesamt rekonstruierten Blöcke linear. Da jeder Block überprüft werden muss, beträgt die Komplexität für die Abfrage von n Blöcken $\mathcal{O}(n)$.

Ein anderes Konzept nutzt eine Optimierung mittels einer Beschleunigungsstruktur namens *Octree*. Durch eine hierarchische Ordnung wird über rekursive Abfragen vom Wurzelknoten bis zu den Blättern die Sichtbarkeit überprüft. Obwohl die Komplexität der Abfrage durchschnittlich geringer ist[95], muss beachtet werden, dass die Erzeugung und Änderung eines Octrees zusätzliche Rechenzeit beansprucht. Im Fall einer Aktualisierung der Raumrekonstruktion, müssen neue *Bounding Boxes* dem Octree hinzugefügt werden. Ebenso können sich die Maße bestehender Blöcke verändern, was ebenfalls in einer Umstrukturierung des *Octrees* resultiert.

Sowohl der iterative Sichtbarkeitstest, als auch der Octree basierte Ansatz wurden implementiert.

Um einen Octrees in Unity zu verwenden, wurde die Octree Implementierung *UnityOctree* eingebunden. Die Kennung eines rekonstruierten Blocks wird im Octree zusammen mit der *Bounding Box* hinterlegt. In der „Update“ Methode wird einmal pro Bildwiederholung eine von der Octree Implementierung bereitgestellte Funktion zur Überprüfung der Sichtbarkeit aufgerufen. Diese Funktion gibt Referenzen auf die Kennnummern aller sichtbaren Bounding Boxes im Bildbereich zurück.

Um zu bestimmen, welche Blöcke ge- und entladen werden müssen, wird untersucht, welche Blöcke innerhalb einer Bildwiederholung in den Sichtbereich eindringen und welche ihn verlassen.

Dazu werden die sichtbaren Kennnummern des n-ten Bildes zwischengespeichert, um im (n+1)-ten Bild die Differenz der beiden Listen zu bestimmen. Anhand der Kennnummern werden die rekonstruierten Blöcke aus dem Register gelesen. Das Dreiecksmodell wird in eine Datei geschrieben, beziehungsweise aus einer Datei gelesen.

Im folgenden Abschnitt wird die iterative Implementierung der Sichtbarkeitstests mit der Octree basierten Implementierung verglichen. In zwei verschiedenen Szenarien wird

[95] Vgl. Saona-Vázquez, Navazo und Brunet, 1999.

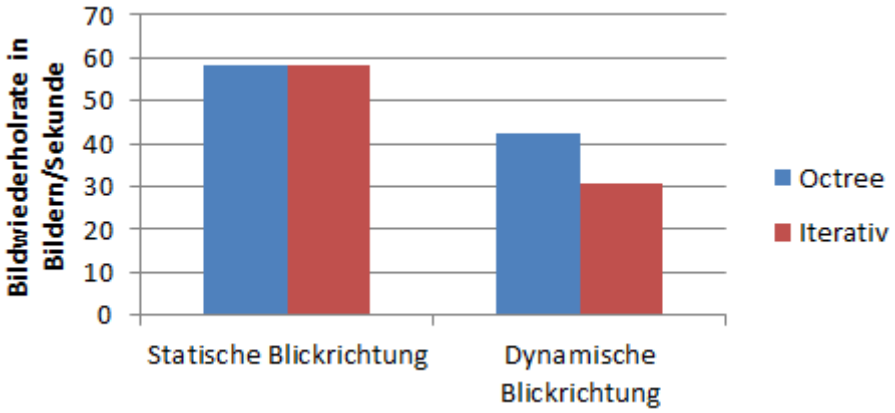


Abbildung 5.7: Durchschnittliche Bildwiederholrate in Bilder/Sekunde für iterative und Octree basierte Sichtbarkeitsabfragen

die durchschnittliche Bildwiederholrate der durchschnittliche Speicherverbrauch und der maximaler Speicherverbrauch untersucht: Während des ersten Tests verändert sich die Blickrichtung nicht. Im zweiten Szenario ist zwar die Position der HoloLens konstant, die Blickrichtung wird aber in einem Bereich von 0 bis 90 Grad verändert. Dadurch variiert die Anzahl der gleichzeitig sichtbaren Blöcke zwischen 0 und 16.

Beide Szenarien wurden von der selben Position im identischen Raum mit 159218 Knotenpunkten über 3600 Bildwiederholungen getestet. Die initiale Kantenlänge des Octrees betrug fünf Meter, die minimale Kantenlänge der Unterteilungen des Octrees betrug 2 Meter.

Wie in Diagramm 5.7 zu erkennen ist, konnte sowohl mit dem iterativen Sichtbarkeitstest, als auch mit der Octree basierten Lösung eine stabile Bildwiederholrate von nahezu 60 Bildern pro Sekunde bei statischen Bildausschnitt festgestellt werden. Sobald die HoloLens bewegt wird und Blöcke ge- und entladen werden, sinkt die Bildwiederholrate bei beiden Tests. Dies lässt sich auf zusätzliche Festplattenzugriffe zurückführen. Die durchschnittliche Bildwiederholrate der iterativen Implementierung ist um etwa 12 Bilder pro Sekunde geringer. Durch die Verwendung eines Octrees konnte die durchschnittliche Bildwiederholrate um 38% erhöht werden.

In Abbildung 5.8 ist die durchschnittliche und die maximale Hauptspeicherauslastung, sowohl mit statischer, als auch mit dynamischer Blickrichtung dargestellt. Unter Verwendung des Octrees beträgt die durchschnittliche Speicherauslastung 151 MB und die

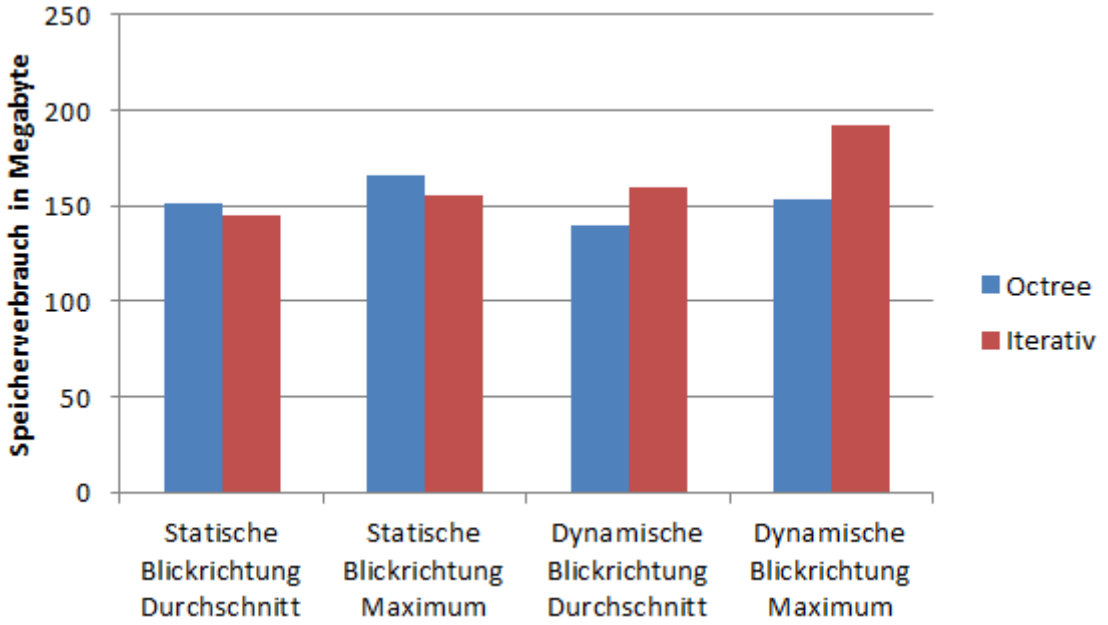


Abbildung 5.8: Durchschnittliche und maximale Speicherauslastung für iterative und Octree basierte Sichtbarkeitsabfragen

maximale Auslastung 166 MB bei statistischer Blickrichtung. Der iterative Ansatz reserviert durchschnittlich 145 MB und maximal 156 MB und ist somit für ein statisches Blickfeld ressourcenschonender. Die Analyse der Speicherauslastung für eine dynamische Blickrichtung ergab, dass der iterative Ansatz deutlich mehr Speicher reserviert.

Caching

Wie aus dem Diagramm 5.7 hervorgeht, ist die durchschnittliche Bildwiederholrate abhängig von der Blickrichtungsänderung des Benutzers.

Ändert sich die Blickrichtung innerhalb einer Bildberechnung, verändert sich die Liste der aktuell sichtbaren Blöcke. Je größer der Winkel der Blickrichtungsänderung ist, desto stärker variieren auch die Elemente in der Liste.

Unterscheiden sich die Listen der n-ten Bildberechnung und der (n-1)-ten Bildberechnung in m Elementen, müssen m Blöcke der Rekonstruktion von der Festplatte geladen, beziehungsweise auf die Festplatte ausgelagert werden. Diese zwei Prozesse benötigen in Abhängigkeit der Größe des Dreiecksmodells Rechenzeit, wodurch die Bildwiederholrate sinkt.

Ein möglicher Ansatz, die Anzahl der Zugriffe auf den Flash-Speicher zu minimieren, ist in Abbildung 5.9 dargestellt. Blöcke, welche sich in einem bestimmen Radius um den Benutzer befinden, werden nicht auf den Flash-Speicher ausgelagert. Stattdessen verbleiben die Blöcke im Hauptspeicher (grün schraffiert hervorgehoben), auch wenn diese nicht sichtbar sind. Dies führt dazu, dass bei einer starken Änderung der Blickrichtung diese Blöcke nicht neu geladen werden müssen, sondern ohne Verzögerung sichtbar sind. Gleichzeitig steigt allerdings auch die Anzahl der im Hauptspeicher geladenen Blöcke und somit der Speicherverbrauch. Daher soll das Dreiecksmodell in geringerer Qualität zwischengespeichert werden.

Für diesen Ansatz wurde die Klasse „SurfaceObserverManager“ um einen zweiten SurfaceObserver erweitert. Dieser erfasst dabei ausschließlich die Umgebung der HoloLens innerhalb eines definierten Radius. Sobald dieser SurfaceObserver eine Änderung in diesem Bereich erfasst, wird das Dreiecksmodell mit einer reduzierten Knotendichte angefragt und zur Szene hinzugefügt.

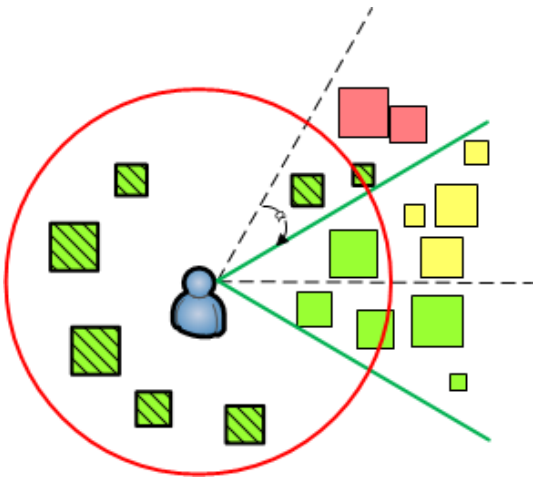


Abbildung 5.9: Zwischenspeichern von Blöcken in der Umgebung

Wenn ein Block der Rekonstruktion nicht mehr sichtbar ist, wird überprüft, ob dessen „Bounding Box“ den kugelförmigen Bereich schneidet. Ist dies der Fall, wird der Block nicht ausgelagert, sondern bleibt in der Szene erhalten. Im Abbildung 5.10 sind die geladenen Blöcke einer einer rekonstruierten Umgebung sichtbar. Der Erfassungsbereich des zweiten SurfaceObservers ist rot hervorgehoben, der Erfassungsbereich des ersten ist grün markiert.

Die durchschnittliche Bildwiederholrate, sowie die durchschnittliche und maximale Hauptspeicherauslastung wurden gemessen.

In gleicher Umgebung wurde dazu die Anwendung mit und ohne zweiten SurfaceObserver auf der HoloLens ausgeführt und über 3600 Bildberechnungen hinweg die Speicherauslastung und die Wiederholrate aufgezeichnet. Als Radius wurde für den zweiten Durchlauf eine Entfernung von 1,5 Metern gewählt. Die Testergebnisse sind in den Diagrammen in Abbildung 5.11 visualisiert. Der durchschnittliche Speicherverbrauch stieg mit der Nutzung des zweiten SurfaceObservers von 138 MB auf 154 MB, da mehr Blöcke gleichzeitig im

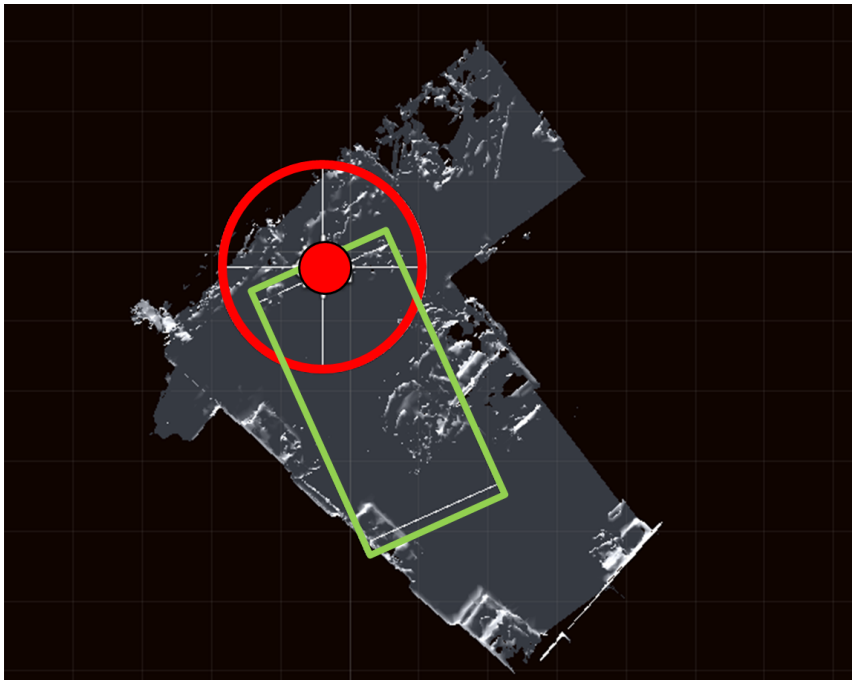
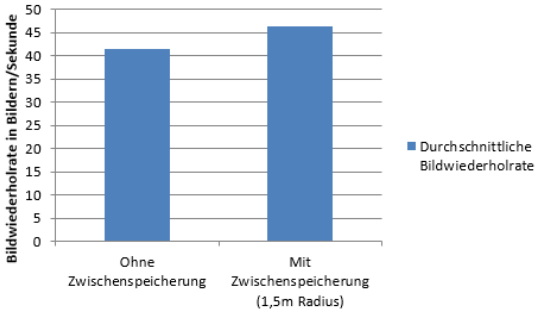
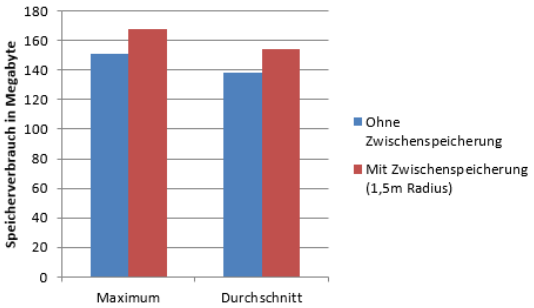


Abbildung 5.10: Visualisierung der Erfassungsbereiche zweier SurfaceObservers

Hauptspeicher geladen werden. Die Speicherauslastung ist somit um 11,59% gestiegen. Die durchschnittliche Bildwiederholrate stieg von 41,4 Bildern pro Sekunde auf 46,3 Bilder pro Sekunde. Dies entspricht einer Erhöhung um 11,84%.



(a) Durchschnittliche Bildwiederholrate



(b) Durchschnittlicher und maximaler Speicherverbrauch

Abbildung 5.11: Einfluss des Zwischenspeicherung von Blöcken in einem Radius von 1,5 Metern auf Speicherverbrauch und Bildwiederholrate

Asynchrone Serialisierung und Speicherung der Modelle

Wenn ein Block ausgelagert oder geladen werden muss, greift die Anwendung auf das Dateisystem des Flash-Speichers zu. Um komplexe Datenstrukturen in einer Datei zu speichern, müssen die Daten so vereinfacht werden, dass die Informationen geordnet in einem sequentiellen Datenstrom an das Dateisystem übertragen werden können. Dieser Prozess nennt sich *Serialisierung* und ist umkehrbar, sodass sich aus einem gelesenen Datenstrom komplexe Strukturen wieder aufbauen lassen (*Deserialisierung*).

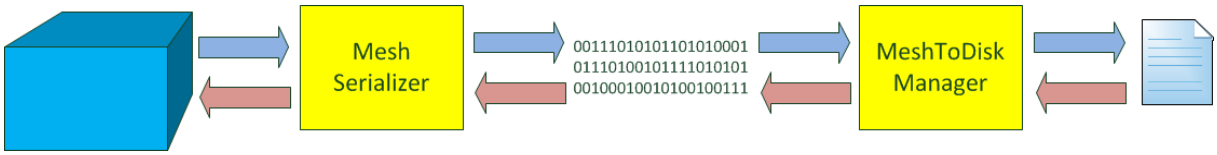


Abbildung 5.12: Speichern und Lesen eines Dreiecksmodells in bzw. aus einer Datei

Die Serialisierung und Deserialisierung eines Dreiecksmodells ist in Abbildung 5.12 dargestellt. Die blauen Pfeile symbolisieren den Ablauf der Speicherung des Dreiecksmodells in eine Datei. Entlang der roten Pfeile wird ein Dreiecksmodell aus den Informationen einer Datei geladen. Sowohl die Serialisierung/Deserialisierung als auch der Zugriff auf das Dateisystem sind zwei langsame Prozesse.

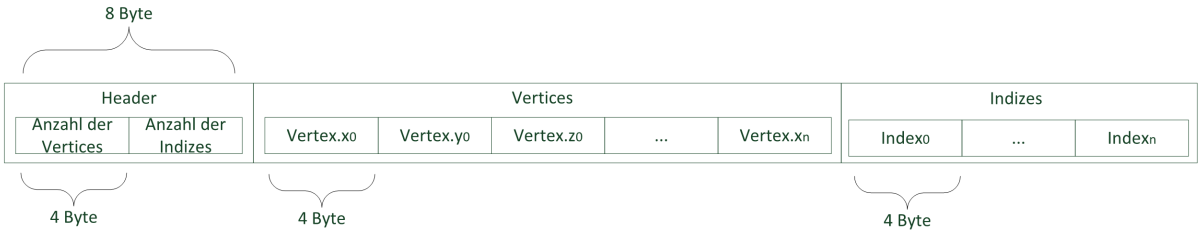


Abbildung 5.13: Binärformat des serialisierten Dreiecksmodells

Bisher wurde die vom HoloToolkit bereitgestellte Klasse *SimpleMeshSerializer* verwendet. Mittels dieser Klasse wird ein Dreiecksmodell entsprechend des im folgenden vorstellten Formates (siehe Abbildung 5.13) serialisiert: In den ersten 8 Bytes wird die Anzahl der Knotenpunkte, sowie die Anzahl der Indizes des Dreiecksmodells als Ganzzahl mit jeweils 4 Bytes gespeichert. Anschließend werden die X, Y und Z Komponenten der Knotenpunkte mit jeweils einer 4 Byte Gleitkommazahl aneinandergereiht. Die Indizes werden als Liste von Ganzzahlen hinter den Knotenpunkte gespeichert.

Es ergibt sich eine Gesamtgröße des Byte-Arrays von $(8 + (3 * 4 * n) + (4 * m))$ Bytes für ein Dreiecksmodell mit n Knotenpunkte und m Indizes. Da sowohl durch alle Knotenpunkte als auch durch alle Indizes während der De-/Serialisierung iteriert werden muss, ergibt sich eine Komplexität von $\mathcal{O}(n + m)$.

Bisher wurde diese Methode immer synchron aufgerufen. Bei der sequentiellen Bearbeitung dieser Prozesse wird die Bildsynthese verzögert. Da diese Prozesse in Abhängigkeit von der Größe des Dreiecksmodells bis zu einer Sekunde in Anspruch nehmen können, verlängerte sich die benötigte Rechenzeit innerhalb eines Bildes, wodurch die Bildwiederholrate sank.

Die Verzögerung durch diese rechenintensiven Prozesse muss minimiert werden. Eine Möglichkeit hierfür besteht darin, die De-/Serialisierung in einem separaten Ausführungsstrang (auch *Thread* genannt) auszulagern. Die Berechnungen eines externen Threads können im Hintergrund ausgeführt werden. Die Verzögerung der Bildsynthese durch langsame Operationen kann dadurch verhindert werden.

Dazu wurde eine Klasse „CustomMeshSerializer“ implementiert, welche die Dreiecksmodelle im gleichen Format wie der „SimpleMeshSerializer“ serialisiert und deserialisiert.

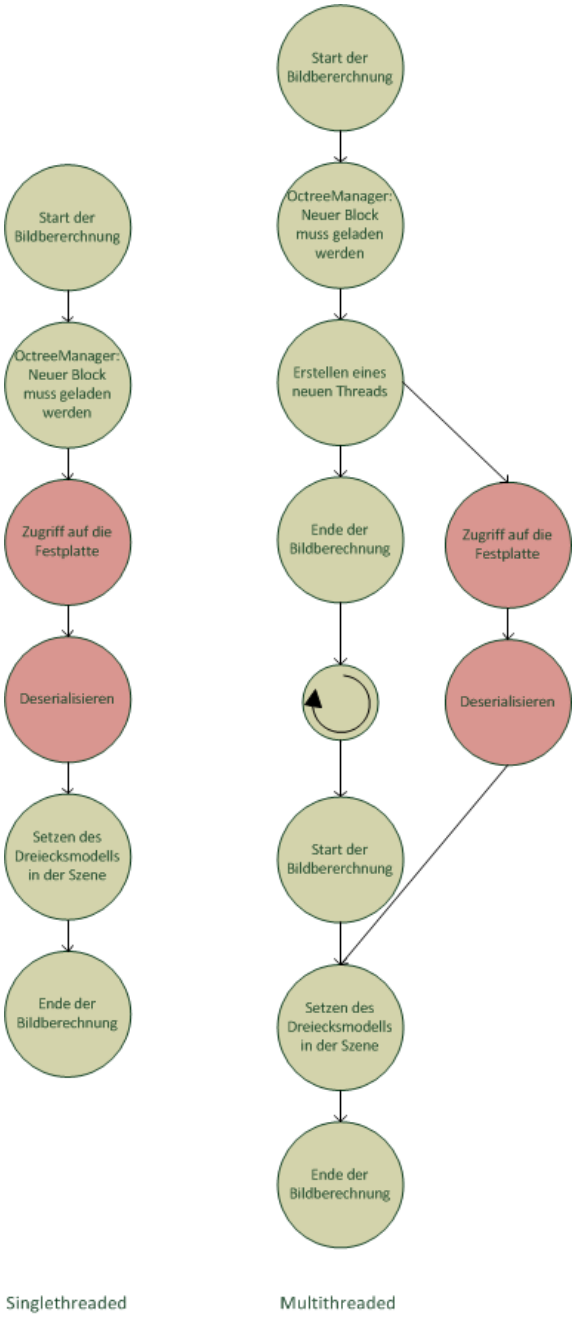


Abbildung 5.14: Ablauf des Ladens von Modellen im Hauptthread (links) und im externen Thread (rechts)

Die Methoden dieser Klassen können in eigenen Threads ausgeführt werden. Wird eine Methode aus dem Hauptthread in einem separaten Thread gestartet, führt dies zu einer unabhängigen Ausführung der beiden Threads. Sobald ein Dreiecksmodell ausgelagert

werden soll, muss dieses Modell erst serialisiert werden, bevor die Daten auf den Flash-Speicher geschrieben werden können. Es besteht eine zeitliche Abhängigkeit zwischen dem Schreiben und Lesen der Dateien und der Serialisaierung/Deserialisierung. Diese beiden Prozesse müssen daher im selben Thread ausgeführt werden.

In Abbildung 5.14 wird der Ablauf des Ladens eines Dreiecksmodells im Hauptthread dem Laden in mehreren Threads gegenübergestellt. Die rechenintensiven Operationen sind rot hinterlegt. Sobald ein Block aus dem Flash-Speicher in die Anwendung geladen werden muss, wird ein neuer Thread erzeugt. Diesem Thread wird die Kennung des entsprechenden Blocks übergeben. Darüber hinaus wird dem Thread mitgeteilt, welche Methode dieser nach Beendigung der Aufgabe im Hauptthread ausführen soll. Diese Methode wird im folgenden auch *Callback-Methode* genannt.

Sobald der Thread gestartet wird, erfolgt das Lesen und Deserialisieren der Daten. Die Knotenpunkte und Indizes des gelesenen Dreiecksmodells werden an die Callback-Methode übergeben. In dieser Methode wird anschließend das Modell aus den Dreiecksinformationen und dem entsprechenden Spielobjekt in der Szene übergeben. Da Änderungen an Geometrien der Unity Szene aus externen Threads nicht möglich sind, muss diese Callback-Methode im Hauptthread ausgeführt werden. Die Speicherung der Dreiecksmodelle verläuft im umgekehrter Reihenfolge. Knotenpunkte und Indizes werden aus dem entsprechenden Spielobjekt gelesen und an den Thread übergeben. Die Informationen werden in der Datei gespeichert. In der Callback-Methode des Hauptthreads wird das Dreiecksmodell gelöscht und der Hauptspeicher freigegeben.

5.1.4 Texturkoordinatengenerierung

Um den Detailgrad der erfassten Oberflächen in der Umgebung zu erhöhen, soll die Objektoberfläche durch eine Textur erweitert werden. Materialzustand und -farbe werden anhand von Fotoaufnahmen der realen Umgebung erfasst und mithilfe der Textur auf dem virtuellen Modell abgebildet. Um die korrekte Darstellung von Farbwerten auf einer Oberfläche sicherzustellen, wird ein UV-Layout vorausgesetzt.

Das UV-Layout eines Dreiecksmodells ist abhängig von der internen Struktur des Modells. Jedem Knotenpunkt wird eine Koordinate auf der Textur zugewiesen. Mittels dieser Koordinaten kann sowohl lesend, als auch schreibend auf die Textur zugegriffen werden. Der in der Textur hinterlegte Wert kann verwendet werden, um Farbinformationen auf

dem 3D-Modell darzustellen. Drei Knotenpunkte im Modell bilden eine Dreiecksoberfläche. Die UV-Koordinaten der drei Knotenpunkte sind ebenfalls im Dreieck angeordnet. Je mehr Texel dieses Dreieck in der Textur abdeckt, desto höher ist die lokale Auflösung der Bildinformationen dieses Dreiecks auf der Objektoberfläche.

Jedes Dreieck repräsentiert eine sichtbare Fläche der Umgebung. Um die gesamte Umgebung zu erfassen und zu visualisieren, muss für jeden Knotenpunkt in den Modellen eine UV-Koordinate generiert werden.

Generierung des UV-Layouts

Da mehrere Dreiecksmodelle sich in kurzer Zeit ändern können, muss die Generierung von UV-Koordinaten möglichst recheneffizient durchgeführt werden.

Auf Grundlage des von Dong und Hollerer vorgestellten Algorithmus wird pro Knotenpunkt eine UV-Koordinate errechnet. Im Gegensatz zu Dong und Hollerer wird für jeden Block der Rekonstruktion eine eigene Textur verwendet, damit die lokale Auflösung der projizierten Textur auch mit zunehmender Anzahl an Blöcken konstant bleibt. Die U- und V-Komponenten der Koordinaten eines einzelnen Blocks können sich über den gesamten Bereich einer lokalen Textur erstrecken. Unmittelbar nach der Rekonstruktion eines Blockes wird dieser an die UV-Layouterstellung weitergereicht.

Jeweils die Knotenpunkte dreier aufeinander folgender Indizes bilden ein Dreieck. Daher müssen für jedes Dreieck drei UV-Koordinaten errechnet werden.

In Abbildung 5.15 wird die Generierung von UV-Koordinaten anhand zweier Dreiecke eines Würfels demonstriert. Die Indizes der Knotenpunkte sind auf der linken Seite des Bildes rot hervorgehoben. Auf der rechten Seite ist das generierte UV-Layout abgebildet.

Die grundlegende Idee des verwendeten Lösungsansatzes besteht darin, für jedes Dreieck im Modell ein gleichschenkliges, rechtwinkliges Dreieck im UV-Layout zu erstellen. Zwei aufeinanderfolgende Dreiecke werden im UV-Layout so angeordnet, dass diese entlang deren Hypotenuse spiegelverkehrt zueinander liegen und gemeinsam ein Rechteck bilden. Durch dichtes Zusammenfügen der Rechtecke kann die verfügbare Fläche im UV-Layout effizient verwendet werden.

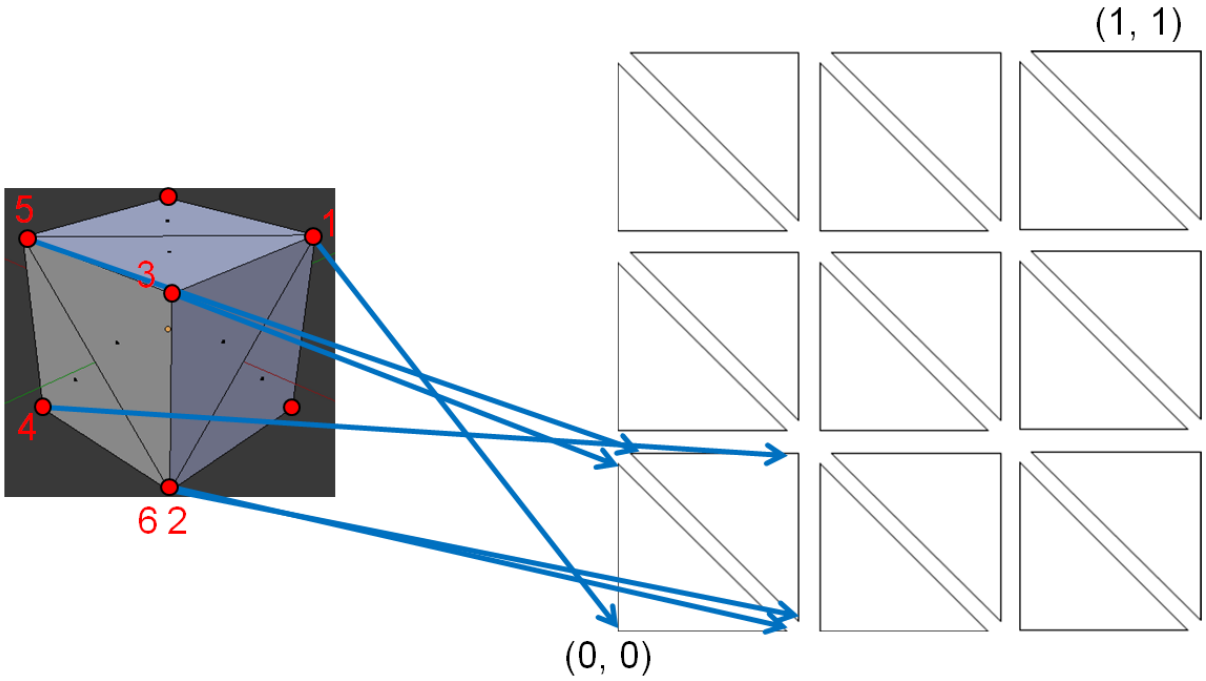


Abbildung 5.15: Generierung von UV Koordinaten anhand eines Würfels

Zunächst wird die Anzahl an Rechtecken entlang einer Dimension im UV-Layout errechnet. Dieser Wert ergibt sich aus der Dreiecksanzahl im rekonstruierten Block. Unter Verwendung einer quadratischen Textur lässt sich die Anzahl der Rechtecke n in Abhängigkeit der Anzahl der Indizes im Modell m über die folgende Formel ermitteln:

$$n = \lceil \sqrt{\frac{m}{6}} \rceil \tag{5.1}$$

Die Kantenlängen k werden in einem naiven Ansatz über den Kehrwert von n berechnet. Mit diesem Ansatz haben alle Rechtecke im UV-Layout eine identische Kantenlänge. In Abhängigkeit von der Anzahl der Dreiecke und der Texturauflösung ist es jedoch möglich, dass im Raster der Textur eine variierende Anzahl an Texeln pro Rechteck zur Verfügung stehen.

Um sicherzustellen, dass die Texel den Dreiecken gleichmäßig zugeteilt werden, wurde ein anderer Ansatz erarbeitet.

$$k = \frac{\lfloor \frac{1}{n} * t \rfloor}{t} \tag{5.2}$$

Der relative Anteil einer Rechteckkante ($\frac{1}{n}$) wird mit der Texturauflösung entlang einer

Dimension t multipliziert. Das Produkt ist die Länge eines Rechtecks in Texeln. Dieser Wert wird anschließend auf die nächste Ganzzahl abgerundet. Dieses Ergebnis wird mittels Division durch die Texturauflösung wieder normalisiert. Somit wird sichergestellt, dass die Kantenlänge eines Rechtecks immer ein ganzzahliges Vielfaches der Texelgröße ist.

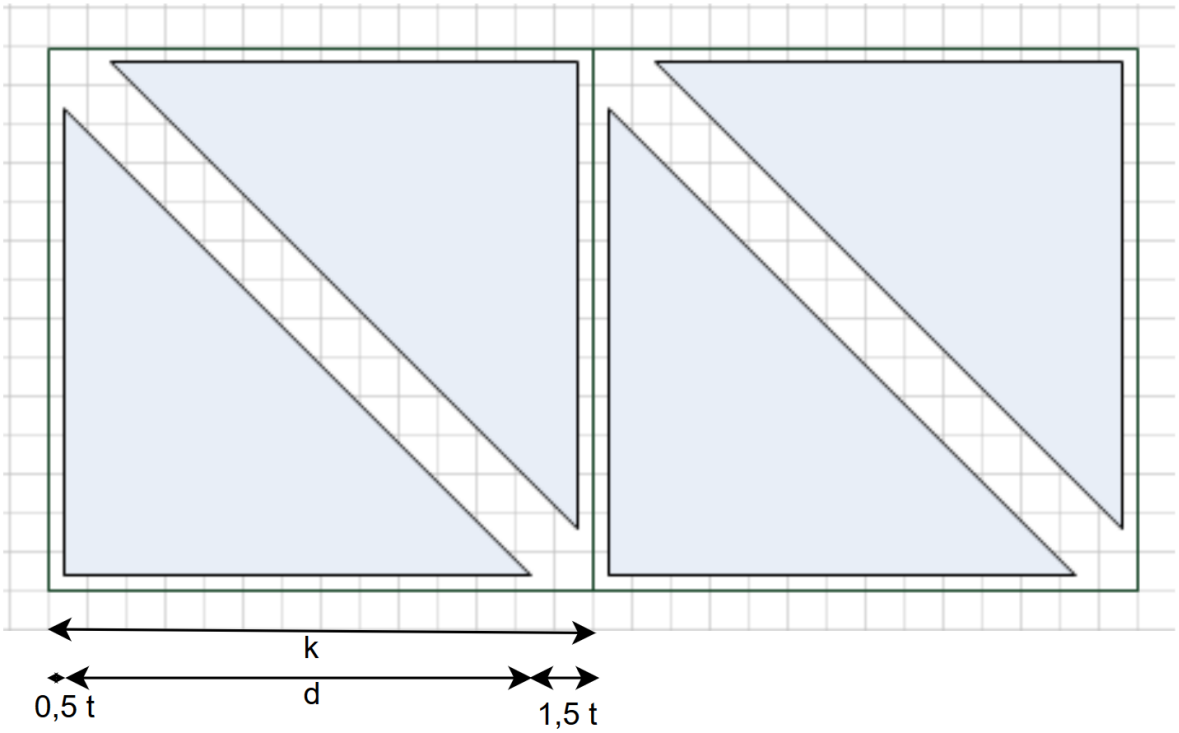


Abbildung 5.16: Abstände zwischen den Dreiecken im UV-Layout

In Abbildung 5.15 haben die einzelnen Dreiecke zueinander einen festen Abstand. Dieser wird von der konservativen Rasterisierung der Dreiecke im späteren Verlauf beansprucht. In Abbildung 5.16 sind diese Abstände deutlicher sichtbar. Ein Zelle des Gitters repräsentiert ein Texel der Textur.

Die Dreiecke werden so angeordnet, dass zwischen den Katheten zweier benachbarter Dreiecke eine halbe Texellänge Abstand zum Rand des Rechtecks gehalten wird. Zwischen den Hypotenusen benachbarter Dreiecke wird immer zwei Pixellängen Abstand entlang der X-Achse gehalten.

Die Länge eines einzelnen Texels l ergibt sich aus dem Kehrwert der Texturauflösung. Somit setzt sich die Länge eines Rechteckes k aus der Texellänge l und der Kantenlänge

des Dreiecks d zusammen:

$$k = 2 * l + d \tag{5.3}$$

Durch Umstellen der Formel 5.3 ergibt sich für die Kantenlänge eines Dreiecks:

$$d = k - 2 * l \tag{5.4}$$

Nach Berechnung dieser Daten wird über alle Dreiecke iteriert und die UV-Koordinaten für die drei Knotenpunkte berechnet.

Duplikation gemeinsamer Knotenpunkte

In einem Dreiecksmodell teilen sich benachbarte Dreiecke üblicherweise einen oder mehrere Knotenpunkte. Die Indexliste kann also an mehreren Stellen Duplikate enthalten. Anhand des Index wird einem Knotenpunkt eine UV-Koordinate zugewiesen. Sobald der gleiche Knoten mehrfach referenziert wird, werden zwei unterschiedliche UV-Koordinaten für einen Knoten errechnet. Eine vorher ausgerechnete UV-Koordinate eines Knotenpunktes wird von nachfolgenden UV-Koordinaten überschrieben. Dies führt dazu, dass diese Dreiecke nicht mehr gleichmäßig geformt sind und andere Dreiecke im UV-Layout überlappen.

Ein simpler Lösungsansatz besteht darin, noch vor der Generierung der UV-Koordinaten zu überprüfen, ob mehrere Dreiecke sich einen Knotenpunkt teilen. Dazu wird die Liste mit Inidizes auf Duplikate überprüft und für jeden doppelten Index ein neuer Knotenpunkt erzeugt. Obwohl die duplizierten Knotenpunkte identische Position haben, kann jedem Knotenpunkt eine eigene UV-Koordinate zugewiesen werden.

Sowohl die Überprüfung des Dreiecksmodell auf gemeinsame Knotenpunkte als auch die Darstellung der doppelten Knotenpunkte erhöht den Rechenaufwand.

Kritische Betrachtung des gewählten Algorithmus

Mit dem im vorherigen Abschnitt vorgestellten Algorithmus ist es möglich, Texturkoordinaten für dynamische Dreiecksmodelle effizient zu erstellen.

Da jedem Dreieck der Rekonstruktion eine gleich große Fläche im UV-Layout zugewiesen wird, variiert die lokale Auflösung der Dreiecke entlang der Oberfläche eines Blocks. Auch die lokale Auflösung innerhalb eines Dreiecks kann variieren und zu verzerrten Texturinhalten führen. Dieses Phänomen tritt besonders dann auf, wenn sich die Form des Dreiecks im Modell stark von dem rechtwinkligen, gleichschenkligen Dreieck im UV-Layout unterscheidet (siehe Abbildung 5.22 links).

Das generierte UV-Layout ist abhängig von dem internen Aufbau des Dreiecksmodells. Die Anzahl Position der generierten Dreiecke aus dem Dreiecksmodell variieren jedoch mit der Zeit. Somit wird bei Aktualisierung eines Blockes das gesamte UV-Layout neuberechnet. Die Textur der vorherigen Generation eines UV-Layouts ist nicht kompatibel mit einem neu erstellen UV-Layout. Während die Blöcke dynamisch aktualisiert werden, können Texturen nicht dauerhaft korrekt auf dem Modell angezeigt werden.

Um dieses Problem zu lösen, wurde die Anwendung in zwei Phasen ausgeführt:

Phase I: Während der ersten Phase werden in regelmäßigen Abständen die Dreiecksmodelle der Umgebung aktualisiert. Die UV-Layouts der Blöcke können sich dynamisch verändern. Sobald das Dreiecksmodell eines Blocks aktualisiert wird, kann eine Textur auf Basis des neu erstellen UV-Layouts auf dem Objekt bis zur nächsten Aktualisierung angezeigt werden. Dem Benutzer wird eine temporäre Vorschau der Textur auf dem Dreiecksmodell angezeigt.

Phase II: In der zweiten Phase werden keine Dreiecksmodelle aktualisiert. Die UV-Layouts werden nicht mehr verändert. Eine Textur auf Basis des konstanten UV-Layouts kann dauerhaft auf der Rekonstruktion angezeigt werden.

5.2 Fotoaufnahme

Die Oberflächendetails der Umgebung sollen möglichst genau auf der virtuellen Rekonstruktion angezeigt werden. Über die in der HoloLens integrierte Farbkamera werden reale Oberflächen fotografiert. Die Kamera unterstützt sowohl die Aufnahme einzelner Fotos

mit einer Auflösung von bis zu 2 Megapixel, als auch Videoaufnahmen mit einer Auflösung von 1408 x 792 Pixeln bei 30 Bildern pro Sekunde.[96]

Mithilfe des Unity Plugins *HoloLensCameraStream* ist es möglich, einzelne Bilder zusammen mit den intrinsischen und extrinsischen Parametern der Kamera aus einem Video zu entnehmen. Die Qualität des entnommenen Videobildes ist allerdings aufgrund der geringeren Auflösung und der Bewegung des HoloLens-Trägers schlechter als die einer Einzelaufnahme. Durch den Betrieb der Farbkamera wird die Bildwiederholrate der Anwendung auf maximal 30 Bilder pro Sekunde begrenzt.

Während der ersten Ausführungsphase der Anwendung werden die Farbinformationen aus dem Video entnommen. Die Qualität der Bildinhalte wird in dieser Phase vernachlässigt, da die Textur nur temporär verwendet werden kann. Unabhängig jeglicher Benutzereingaben wird das dreidimensionale Modell des Raumes generiert. Zu jedem Zeitpunkt kann auf das aktuelle Bild der Videoaufzeichnung zugegriffen werden.

In der zweiten Phase wird eine hohe Qualität der Farbaufnahmen benötigt, um die Oberflächen ausgewählter Objekte in der Rekonstruktion näher erfassen zu können. Daher kann der Benutzer während dieser Phase manuell Fotos aufnehmen.

5.3 Projektion der Bildinformationen

Um mithilfe von Bildinformationen der realen Umgebung die Modelloberfläche einzufärben, werden die Koordinaten des Modells werden in die Fotoebene projiziert, die Farbwerte ausgelesen und in einer Textur gespeichert.

Für jeden Block wird eine eigene Textur verwendet. Diese Textur speichert Bildinformationen entsprechend dem generierten UV-Layout. Während des Renderings eines Blockes werden die Farbwerte aus der Textur gelesen und dargestellt. Auf die Textur eines Blockes wird während der Laufzeit wird lesend und schreibend zugegriffen.

In den folgenden Abschnitten wird dargelegt, wie die Farbinformationen aus einer oder mehreren Fotos in eine Textur geschrieben werden können. Außerdem wird erläutert, wie diese Texturen effizient serialisiert und deserialisert werden.

[96] Vgl. Microsoft, 2018f.

5.3.1 Statisches Dreiecksmodell

Der in diesem Abschnitt vorgestellte Ansatz zur Projektion von Bildinformationen auf ein dreidimensionales Modell basiert auf den Erkenntnissen des Praktikumsberichtes T3000 von Wulkop (2018).

In dieser Arbeit wurde die Projektion von Farbinformationen aus einem aufgenommenen Bild anhand eines einzelnen statischen Dreiecksmodells behandelt. Das UV-Layout des Modells ist in dieser Betrachtung bereits bekannt. Die Vorgehensweise wird anhand eines Quaders demonstriert. Dieser ist in Abbildung 5.17 abgebildet.

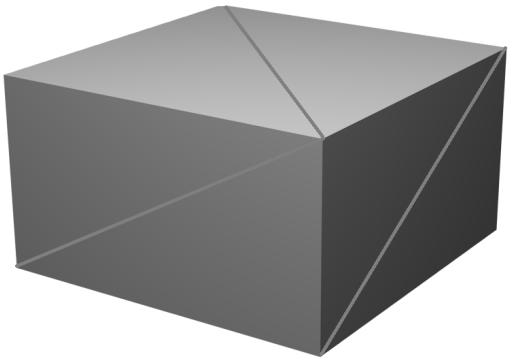
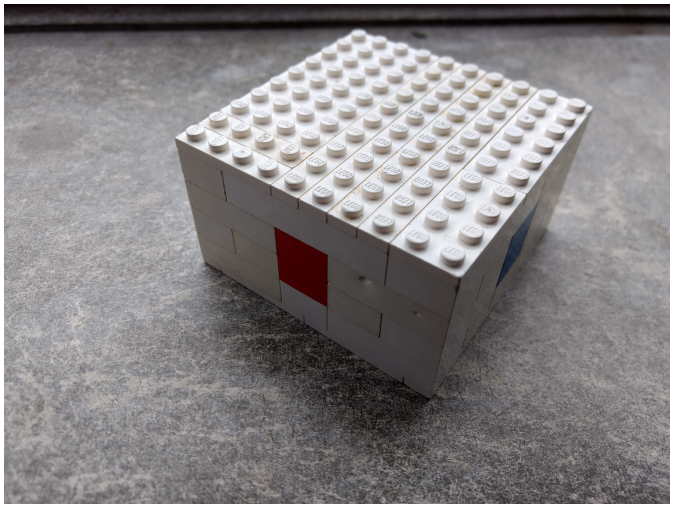


Abbildung 5.17: Dreiecksmodell eines Quaders

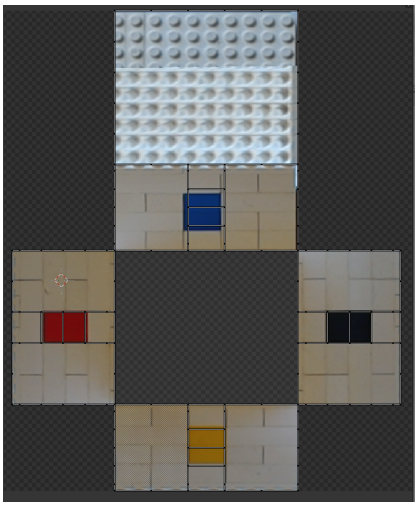
Den Oberflächen, die während der Bildaufnahme aus der Perspektive der Farbkamera sichtbar sind, werden Farbwerte aus dem aufgenommenen Foto zugeordnet. Für jeden Knotenpunkt wird dazu ausgerechnet, an welcher Koordinate im Foto der Knotenpunkt sichtbar ist. Das Renderergebnis wird in einer dedizierten Textur gespeichert und auf dem Quader angezeigt. Diese Operationen werden parallel auf dem Grafikprozessor durchgeführt.

Projektion der Knotenpunkte in die Fotoebene

Um die dreidimensionalen Knotenpunkte des Modells auf die Fotoebene zu projizieren, wird ein Shaderprogramm bestehend aus Vertex-, Geometry- und Fragement Shaderebene verwendet.



(a) Aufgenommenes Foto der HoloLens Farbkamera



(b) Generierte Farbtextur

Abbildung 5.18: Aufgenommenes Foto (links) und erstellte Farbtextur (rechts)

Sobald ein Farbfoto des Modells aufgenommen wurde (siehe Abbildung 5.18 links), wird zu dem kontinuierlich fortlaufenden Renderprozess ein zweiter Renderprozess für das gleiche Modell gestartet.

Im zweiten Renderprozess wird ein dediziertes Shaderprogramm verwendet. Das gerenderte Bild des zweiten Prozesses wird nicht auf den Bildschirmen der HoloLens ausgegeben, sondern stattdessen in der Textur des Quaders gespeichert. Eine generierte Textur des Quaders ist in Abbildung 5.18 auf der rechten Seite dargestellt. Im kontinuierlichen Renderprozess wird das Modell aus der Perspektive der Hauptkamera gerendert (siehe Abbildung 5.17). Während des zweiten Renderprozesses wird das Modell nicht in die Bildebene der Kamera projiziert. Stattdessen werden die Knotenpunkte in die zweidimensionale Ebene des UV-Layouts projiziert (siehe Abbildung 5.18). Die von der Fragment-Shaderebene errechneten Farbwerte werden berechnet und die Textur wird mit Farbinhalten beschrieben.

Für eine korrekte Projektion wird dem Shaderprogramm das aufgenommene Bild, die intrinsischen und extrinsischen Parameter der Farbkamera und die intrinsischen und extrinsischen Parameter der virtuellen Hauptkamera übergeben. In der Vertex-Shaderebene wird die UV-Koordinate eines Knotenpunkts ausgelesen und als Zielkoordinate für das Renderergbnis festgelegt. Die dreidimensionalen Koordinaten des Modells werden in das globale Koordinatensystem der Unity Szene transformiert. Anschließend werden diese Koordinaten anhand der externen Parameter der Farbkamera in das lokale Koordinatensystem

dieser Kamera transformiert. Da die internen Parameter der Farbkamera bekannt sind, ist eine Projektion der dreidimensionalen Koordinaten in die zweidimensionale Bildebene der Farbkamera möglich. Diese errechneten Koordinaten werden an die Geometry-Shaderebene weitergegeben.

In der Geometry-Shaderebene werden die errechneten Koordinaten auf der Textur verändert. Die Dreiecksfläche zwischen den Texturkoordinaten der drei Knotenpunkte eines Primitivs wird an allen Kanten um einen halben Pixel vergrößert. Diese Vergrößerung ist in Abbildung 5.19 dargestellt.

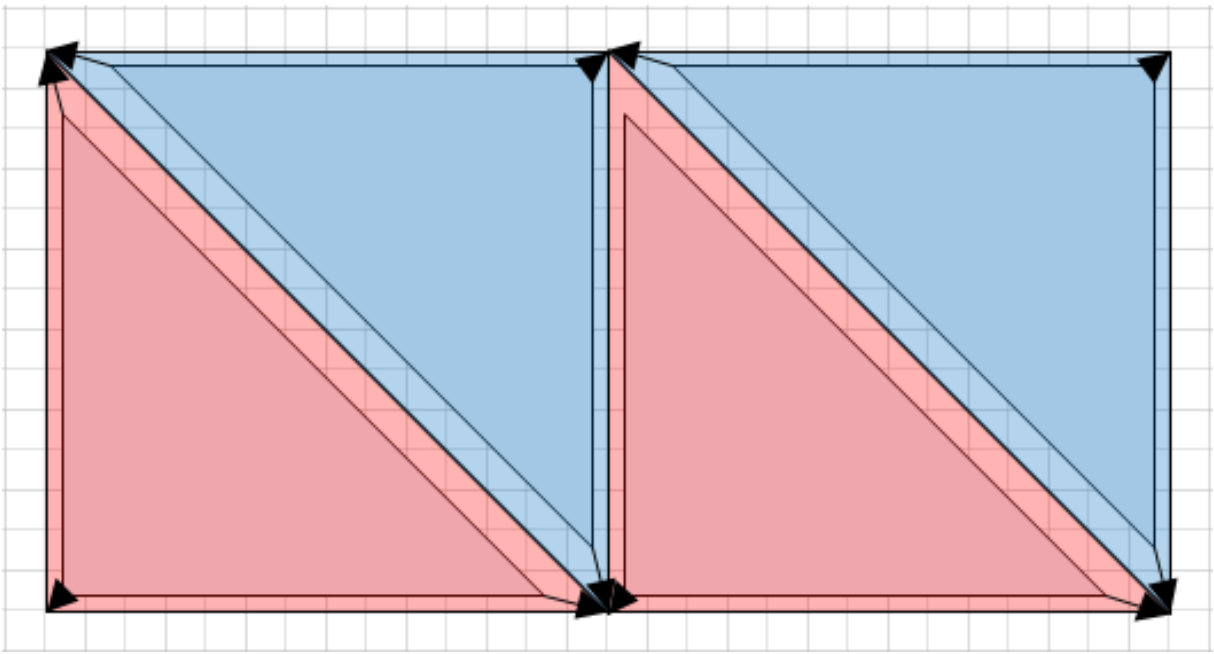


Abbildung 5.19: Vergrößerung der Primitive in der Geometry-Shaderebene

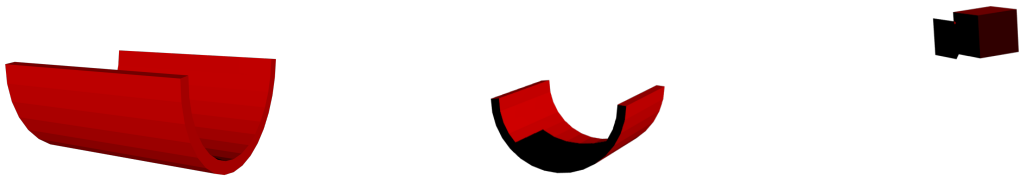
Anhand der drei Zielkoordinaten des Primitivs ist es möglich, zwischen einem oberen Dreieck (blau hervorgehoben) und einem unterem Dreieck (rot hervorgehoben) zu unterscheiden. In Abhängigkeit von der Orientierung des Primitivs werden die Zielkoordinaten verändert und die Dreiecksfläche vergrößert. Der Randbereich zwischen den Dreiecken im UV-Layout wird verwendet, um eine konservative Rasterisierung durchzuführen. Zusammen mit den Texturkoordinaten werden auch die in die Fotoebene projizierten Koordinaten skaliert.

Daraufhin werden die Primitive rasterisiert und die interpolierten Koordinaten jedes Fragments der Fragment-Shaderebene übergeben. Für jedes Fragment wird überprüft, ob sowohl X- als auch Y-Komponente der Koordinate im Bildbereich liegen. Eine Koordinate ist ungültig, sobald sich das rasterisierte Fragment außerhalb des auf dem Foto sichtbaren

Ausschnitts befindet. Fragmente, welche keine gültige Farbe aus dem Foto auslesen können werden maskiert. Ist eine Koordinate gültig, wird der in dem Foto hinterlegte Farbwert in die Rendertextur des Quaders geschrieben.

Sichtbarkeitstest der Fragmente

Ein Problem des im vorherigen Abschnitt beschriebenen Shaderprogrammes besteht darin, dass sich aus Sicht der Kamera verdeckende Fragmente identisch in die Fotoebene projiziert werden. Das hintere Fragment ist auf der Farbaufnahme nicht sichtbar. Diese Situation ist in Abbildung 5.20 dargestellt. Eine korrekte Texturprojektion schreibt die verdeckten Fragmente (schwarz hervorgehoben) nicht in die Textur, da das aufgenommene Foto keine sinnvollen Bildinhalte für diese Fragmente beinhaltet.



(a) Objekt aus Sicht der Kamera *(b)* Objekt und Kamera aus externer Sicht

Abbildung 5.20: Sichtbare Fragmente eines Objektes in roter Farbe hervorgehoben

Mittels eines sogenannten *Tiefentests* ist die Analyse von einander verdeckenden Fragmenten möglich. Anhand des Ergebnisses dieses Tests kann bestimmt werden, ob ein Fragment andere Fragmente verdeckt oder ob es selber verdeckt wird. Ein verdecktes Fragment wird ebenfalls maskiert.

Texturkomposition

Unter der Voraussetzung, dass das UV-Layout des zu texturierenden Objektes sich nicht verändert, können die Bildinformationen mehrerer Fotos, welche aus unterschiedlichen Perspektiven aufgenommen wurden, in einer Textur zusammengefasst werden.

Wenn eine neue Textur generiert wird, kann die vorherige Textur zusammen mit der Neuen in einem weiteren Schritt der Renderpipeline durch einem *Compute Shader* verarbeitet werden. Anhand der Maskierung kann zwischen gültigen und ungültigen Texturinhalten unterschieden werden. Gültige Texturinhalte der neueren Textur überschreiben und ergänzen Bildinhalte der Älteren.

5.3.2 Dynamisches Dreiecksmodell

Um die Texturprojektion für die Umgebungserfassung zu nutzen, muss zwischen den unterschiedlichen Texturprojektionen während der ersten und der zweiten Phase unterschieden werden.

Phase I: In der ersten Phase können sich die UV-Layouts der rekonstruierten Blöcke dynamisch verändern. Sobald ein Block der Umgebung aktualisiert wurde, wird das aktuelle Bild mitsamt intrinsischen und extrinsischen Parametern der Farbkamera aus dem Video entnommen. Die Bildinformationen auf diesem Bild werden in die Textur des aktualisierten Blocks projiziert. Aufgrund der dynamischen UV-Layouts können mehrere Texturen eines Blocks nicht zusammengefügt werden. Veraltete Texturinhalte werden überschrieben.

Phase II: In der zweiten Phase wird die rekonstruierte Umgebung nicht verändert und das UV-Layout der Blocke bleibt konstant. Während dieser Phase können durch den Benutzer gezielt Fotos aufgenommen werden, um die Qualität interessanter Oberflächen zu erhöhen. Mehrere Aufnahmen eines Blocks aus unterschiedlichen Perspektiven können zusammengefasst werden.

Zum Zeitpunkt einer Fotoaufnahme, werden die sichtbaren Blöcke ermittelt. Da jeder Block eine eigene Textur verwendet, wird die Texturprojektion für jeden sichtbaren Block einzeln durchgeführt.

In Abbildung 5.21 ist das Modell eines Ganges aus Perspektive des Benutzers dargestellt. Die Projektion eines Bildes aus dieser Perspektive hat Einfluss auf die Texturen aller sichtbaren Blöcke. Insgesamt 25 Texturprojektionen sind aus der abgebildeten Perspektive notwendig.

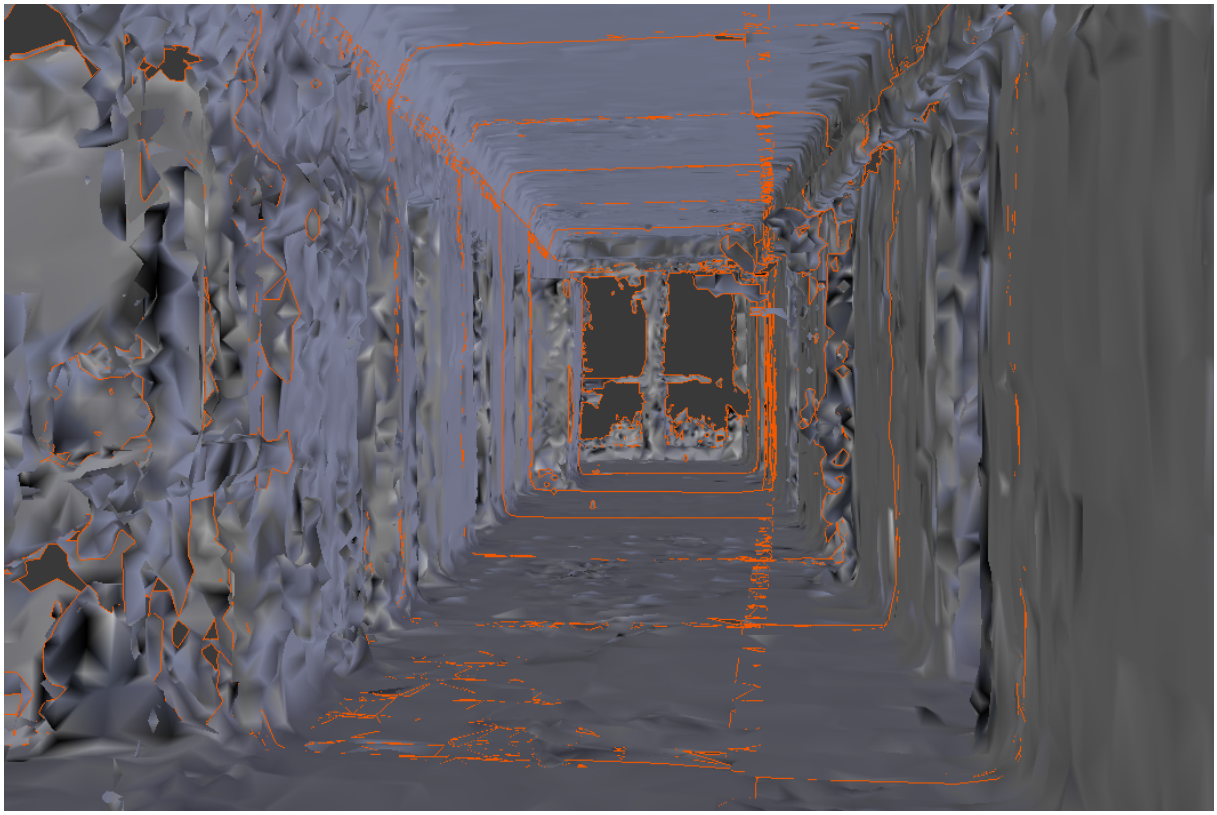


Abbildung 5.21: Rekonstrierte Blöcke aus Perspektive der HoloLens

5.3.3 Speichern und Laden

Bei Erfassung eines neuen Umgebungsblockes, werden Spielobjekt und Textur mit einer festen Auflösung erstellt. Da diese Textur die Bildinhalte der realen Umgehung speichern soll, beinhaltet sie pro Texel einen Rot-, einen Grün- und einen Blaukanal. Um möglichst viele unterschiedliche Farben darstellen zu können, werden acht Bits für jeden Kanal verwendet. Dies entspricht einer Datenmenge von drei Bytes für jeden Texel. Die Datenmenge einer gesamten Textur mit einer Texturauflösung von 1024 x 1024 Texeln beträgt somit drei Megabyte.

In Crowle u. a., 2015 wird die Größe der Texturinhalte mithilfe der verlustbehafteten JPEG Kompression reduziert. Ein solcher Ansatz kann jedoch nicht für diese Anwendung verwendet werden. Denn in Abhängigkeit der Anzahl der Dreiecke eines Blockes können bereits kleine Artefakte als sichtbare Flecken auf dem Modell erscheinen. Die Texturen werden daher unkomprimiert und verlustfrei gespeichert.

Aufgrund der limitierten Ressourcen der HoloLens kann nur eine begrenzte Anzahl an

Texturen gleichzeitig im Hauptspeicher liegen. Ein identischer Lösungsansatz, wie in Abschnitt 5.1.3 für die Daten der Dreiecksmodelle, wird ebenfalls für die Texturen verwendet. Während das Speichern der temporären Texturen in der ersten Anwendungsphase nicht notwendig ist, müssen alle Texturen während der zweiten Phase gespeichert und geladen werden können. Dabei wird die Textur eines Blockes zusammen mit dessen Dreiecksmodell von der Festplatte geladen oder auf die Festplatte geschrieben.

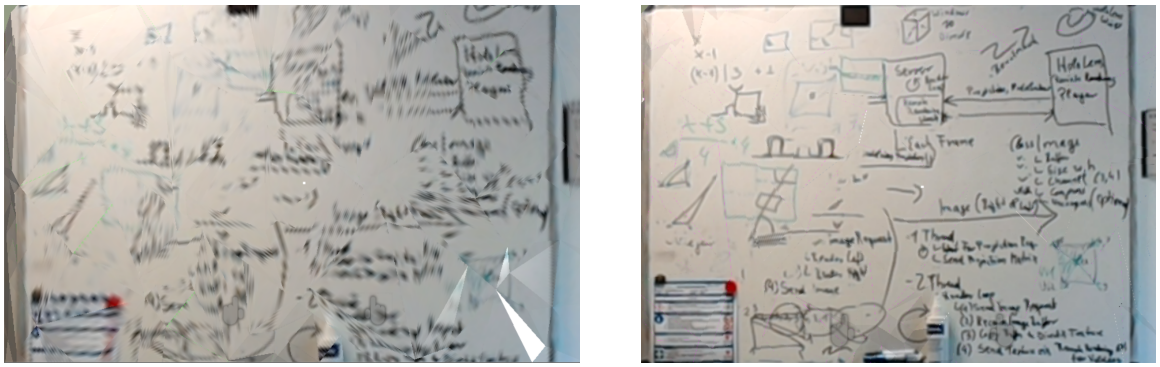
Die Texturinhalte müssen serialisiert und deserialisiert werden. Dazu werden mithilfe eines Compute Shaders die Texturinhalte aus der Rendertextur, welche sich im Video-Hauptspeicher des Grafikprozessors befinden, in eine dedizierte 2D-Textur kopiert. Die Bilddaten der 2D-Textur werden im Hauptspeicher hinterlegt. Auf diesen Bereich des Hauptspeichers kann direkt zugegriffen werden, um die Daten asynchron in eine Datei zu schreiben. Auf umgekehrtem Weg können die Rohdaten der 2D-Textur können in der Anwendung verwendet und in eine Datei geschrieben werden. Auch für die Deserialisierung wird ein Compute Shader eingesetzt, welcher die geladenen Bildinformationen in die Rendertextur des entsprechenden Blocks kopiert.

5.3.4 Benutzerdefinierte Texturqualität

Standardmäßig wird die Textur jedes Blockes mit einer konstanten Auflösung erstellt. Es ist jedoch möglich, dass der Benutzer die Oberflächen bestimmter Teile der Rekonstruktion detaillierter erfassen möchte. Die Textur dieser ausgewählten Blöcke soll höher aufgelöst sein.

Der Benutzer kann in der zweiten Phase der Anwendung einzelne Blöcke der Rekonstruktion auswählen. Bei Auswahl eines Blockes, wird die zugewiesene Textur verworfen und eine höher aufgelöste Textur dem Block zugewiesen. Obwohl das Dreiecksmodell des Blockes nicht verändert wird, ist eine Neuberechnung des UV-Layouts notwendig, da sich durch die höhere Auflösung die relative Größe eines einzelnen Texels verringert.

Dadurch werden während der Texturprojektion mehr Fragmente verarbeitet und mehr Bildinhalte aus der Fotoaufnahme können in der Textur gespeichert werden. Die benötigte Rechenzeit zur Verarbeitung der erhöhten Fragmentzahl steigt. Gleichzeitig vervierfacht sich die Größe der Textur bei einer Verdopplung der Auflösung von beispielsweise 1024 Pixel auf 2048 Pixel. Mehr Texel müssen serialisiert und deserialisiert werden.



(a) Texturauflösung von 1024 x 1024 *(b)* Texturauflösung von 4096 x 4096

Abbildung 5.22: Rekonstruierte und texturierte Umgebungsoberfläche

In Abbildung 5.22 ist ein texturierter Block einer Rekonstruktion dargestellt. Die Texturauflösung beträgt links 1024 x 1024 Pixel und rechts 4096 x 4096 Pixel.

5.4 Verteilung der Raumrekonstruktion und Texturen

Die reale Umgebung einer HoloLens wird erfasst, visualisiert und gespeichert. In dieser Umgebung soll kollaborativ mit mehreren Teilnehmer an einem Projekt gearbeitet werden. Dies setzt eine identische Rekonstruktion auf den Geräten aller Teilnehmer voraus. Die gesamte virtuelle Umgebung muss zwischen den Anwendungen über Netzwerk synchronisiert werden.

Eine HoloLens führt dabei die Umgebungserfassung aus (im folgenden *Host* genannt) und sendet sowohl die Dreiecksmodelle, als auch die dazugehörigen Texturen an andere passive Teilnehmer einer Sitzung (im folgenden *Clients* genannt). Die passiven Teilnehmer empfangen die Daten und zeigen die rekonstruierte Umgebung an.

Um Daten aus einer Unity Anwendung über das Netzwerk zu übertragen, können entweder der *Sharing Service* oder die Unity-internen Netzwerkkomponenten verwendet werden. Die HoloLens zeichnet sich durch ihre Mobilität und Unabhängigkeit von externen Geräten aus. Die Verwendung des Sharing Services setzt jedoch eine Instanz auf einen externen Computer voraus. Um die HoloLens Anwendungen unabhängig von externen Geräten miteinander zu verbinden, werden die Unity-internen Netzwerkkomponenten verwendet.

Aufbauend auf Bausteinen des Mixed Reality Toolkits wird dem Benutzer eine grafische Oberfläche beim Start der Anwendung gezeigt (siehe Abbildung 5.23). Der Benutzer kann als Client einer bestehenden Sitzung beitreten oder als Host eine neue Sitzung eröffnen. Wird eine neue Sitzung eröffnet, senden die Unity-Komponenten in regelmäßigen Abständen Broadcast Nachrichten, damit andere Geräte im gleichen Netzwerk die Sitzung entdecken können. In Abhängigkeit der aktuellen Ausführungsphase des Hosts werden unterschiedliche Daten an alle Teilnehmer versendet:

Abbildung 5.23: Benutzeroberfläche zur Verwaltung von Sitzungen

Phase I: In der ersten Ausführungsphase werden die erfassten Dreiecksmodelle an andere Teilnehmer übertragen. Sobald ein neuer Block erfasst oder ein bestehender Block aktualisiert wird, müssen auch andere Teilnehmer diese Änderung übernehmen. Die Dreiecksmodelle dieser Blöcke werden serialisiert und in eine Nachricht geschrieben.

Phase II: Nimmt ein Benutzer in der zweiten Ausführungsphase ein Foto auf, werden Bildinformationen des Fotos auf die Umgebungsoberfläche der Rekonstruktion projiziert. Die Texturen von Blöcken im Sichtbereich des Benutzers verändern sich und müssen an andere Teilnehmer übertragen werden. Durch eine verlustbehaftete Komprimierung (wie beispielsweise JPEG) kann zwar die Speichergröße der Texturen reduziert werden, jedoch verringern mögliche Farbartefakte die Qualität der Textur. Daher wird die Textur unkomprimiert serialisiert und in eine Nachricht geschrieben.

In Abhängigkeit vom Inhalt variiert die Größe der Nachricht. Sowohl ein serialisiertes Dreiecksmodell, als auch eine Textur kann mehrere Megabyte Speicherplatz reservieren. Besonders hochauflösende Texturen benötigen entsprechend mehr Speicherplatz.

Die Größe der Netzwerkpakete, in denen diese Daten übertragen werden, ist auf wenige Kilobytes beschränkt. Daher werden die Nachrichten, bevor diese versendet werden, in mehrere kleinere Nachrichten unterteilt. Diese Pakete werden in eine Warteschlange eingereiht.

Die Elemente der Warteschlange werden nacheinander einzeln über das Netzwerk versendet. Jeder Teilnehmer muss das Empfangen eines Pakets quittieren, um sicherzustellen, dass alle Pakete korrekt in der richtigen Reihenfolge von allen Teilnehmen empfangen werden.

Die gemessene Übertragungsrate zwischen einem Host und einem Client über Wireless Lan beträgt durchschnittlich 0,4 Megabyte pro Sekunde. Bei einer Texturauflösung von 1024 x 1024 Pixeln kann in 10 Sekunden eine Textur eines Blocks übertragen werden. Häufig werden die Texturen mehrerer Blöcke während einer Texturprojektion verändert. Mehr Elemente werden der Warteschlange hinzugefügt, als verarbeitet werden können. Die Speichergröße der Warteschlange steigt und Änderungen der Umgebung werden mit Verzögerung bei den Teilnehmern empfangen.

6 Evaluierung

Wie bereits im Kapitel der Implementierung gezeigt werden konnte, wird die Speicher- und Rechenauslastung der Umgebungserfassung sowohl von Umwelt-, als auch durch Softwarefaktoren beeinflusst. Dabei muss die Anwendung zwei Anforderungen erfüllen: Zum einen sollen keine Annahmen zur Größe der rekonstruierten Umgebung getroffen werden. Zum anderen soll die Umgebungserfassung dem Nutzer interaktiv präsentiert werden. Daher wurden während der Implementierung mehrere Optimierungsmöglichkeiten untersucht. Anhand der gemessenen Zwischenergebnisse konnte der Einfluss dieser Optimierungen auf Rechen- und Speicherauslastung der Anwendung festgestellt werden. Um eine optimale Bildwiederholrate und einen minimalen Speicherverbrauch zu erhalten, werden folgende drei Softwarefaktoren untersucht: die maximale Knotendichte der rekonstruierten Umgebung, der Caching-Radius und die Texturauflösung. Unterschiedliche Einstellungen dieser Parameter werden auf dem aktuellen Stand der Anwendung getestet. Sowohl die durchschnittliche Bildwiederholrate als auch der durchschnittliche und maximale Speicherbedarf wird gemessen. Alle Tests wurden dabei in einer gleichen Umgebung über eine Zeitspanne von 3600 Bildberechnungen hinweg durchgeführt.

Knotenpunktdichte: Es wird vermutet, dass die eingestellte maximale Knotendichte einen Einfluss auf die Speicherauslastung und Dauer der Bildberechnung hat. Je höher diese obere Grenze der Knotenpunktdichte ist, desto mehr Knotenpunkte werden potentiell in einem Block gespeichert und müssen verarbeitet werden. Die Bildwiederholrate und die Speicherauslastung werden mit unterschiedlichen Grenzwerten der Knotendichte gemessen. In jedem Test wird eine Umgebung bestehend aus bis zu 36 Blöcken stückweise erfasst. Blöcke in einem Radius von 1,25 Metern werden zwischengespeichert und eine Texturgröße von 1024 x 1024 Texeln wird verwendet. Die durchschnittliche Bildwiederholrate in Bildern pro Sekunde und der durchschnittliche und maximale Speicherverbrauch in Megabyte werden jeweils für eine Obergrenze von 100, 1000 und 5000 Knotenpunkten pro Kubikmeter gemessen. Die Ergebnisse dieser Messung sind in Abbildung 6.1 dargestellt. Die durchschnittliche Bildwiederholrate ist als Säule aufgetragen. Der durchschnittliche und maximale Speicherverbrauch (in Megabyte) ist zum Vergleich als Linie abgebildet.

Mit zunehmender Knotenpunktdichte nimmt die durchschnittliche Bildwiederholrate ab und der Speicherverbrauch zu. Gleichzeitig verändert sich auch die Qualität der rekonstruierten Umgebung.

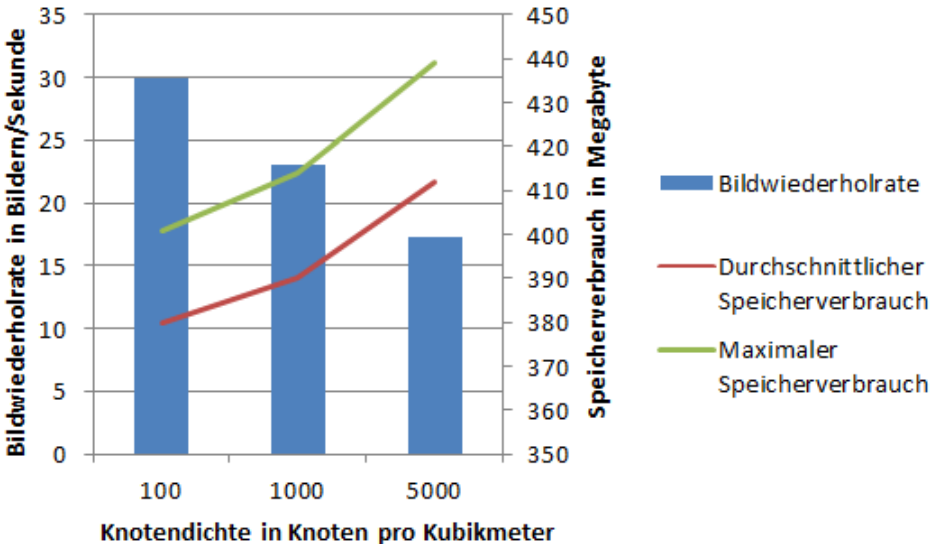


Abbildung 6.1: Durchschnittliche Bildwiederholrate in Bildern/Sekunde und durchschnittlicher und maximaler Speicherverbrauch in Megabyte in Abhängigkeit der Knotendichte

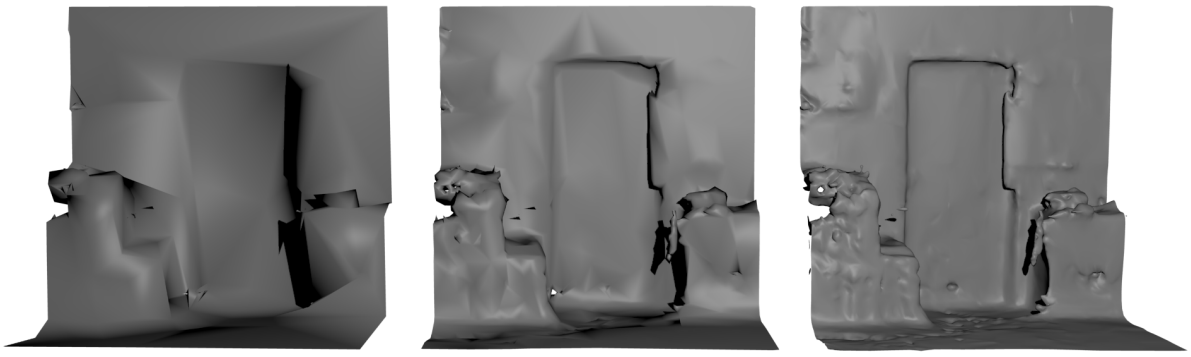


Abbildung 6.2: Rekonstruierter Block mit einer Knotendichte von 100 (links), 1000 (mitte) und 5000 (rechts) Knotenpunkten pro Kubikmeter

In Abbildung 6.2 sind drei verschieden Qualitätsstufen des gleichen Blocks dargestellt. Je höher die Knotenpunktdichte ist, desto besser ist die Qualität. Gleichzeitig steigt jedoch auch die Ressourcenauslastung. Mit der Verwendung einer Knotendichte von 2000 Knoten pro Kubikmeter wird ein Kompromiss zwischen Modellqualität und Ressourcenverbrauch eingegangen.

Radius: Durch den Radius des zweiten SurfaceObservers wird festgelegt ab welcher Entfernung Blöcke der Rekonstruktion auf den Flash-Speicher ausgelagert werden.

Je größer dieser Radius ist, desto mehr Blöcke bleiben potentiell im Hauptspeicher geladen, obwohl diese sich nicht im Sichtfeld befinden. Gleichzeitig sinkt potentiell die Anzahl der Zugriffe auf den Flash-Speicher. Daher wird vermutet, dass sowohl die Auslastung des Hauptspeichers, als auch die durchschnittliche Bildwiederholrate mit größerem Radius steigt.

Auch diese Hypothese wird mit einer Messung der Bildwiederholrate und der Speicherauslastung untersucht. Eine Umgebung wird mit einer maximalen Knotendichte von 2000 Knoten pro Kubikmeter und einer Texturauflösung von 1024 x 1024 Texeln erfasst. Nacheinander werden Messungen mit einem Radius von 0 Metern (kein Zwischenspeichern), 1 Meter, 2 Metern und 5 Metern durchgeführt.

Die Ergebnisse dieser Messungen sind in Abbildung 6.3 dargestellt. Wie vermutet, steigt die Speicherauslastung mit steigenden Radius, da mehr Blöcke gleichzeitig im Hauptspeicher geladen sind. Die Anzahl der Zugriffe auf den Flash-Speicher sinkt. Dadurch reduziert sich die benötigte Rechenzeit zum Laden der Dreiecksmodelle. Gleichzeitig sind jedoch zunehmend mehr Knotenpunkte in der Szene geladen. Auch wenn die entsprechenden Blöcke nicht sichtbar sind, erhöhen diese dennoch die benötigte Rechenzeit innerhalb einer Bildwiederholung. Bei einem Radius von zwei Metern ist die durchschnittliche Bildwiederholrate bei leicht ansteigender Speicherauslastung maximal.

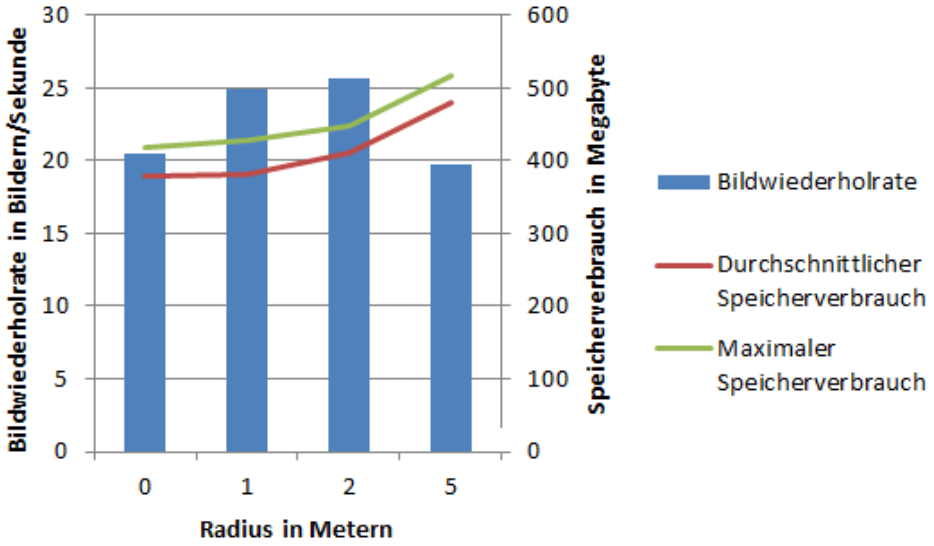


Abbildung 6.3: Durchschnittliche Bildwiederholrate in Bildern/Sekunde und durchschnittlicher und maximaler Speicherverbrauch in Megabyte in Abhängigkeit vom Radius, in dem Blöcke der Rekosntruktion im Hauptspeicher verbleiben

Texturauflösung: Auch die Auflösung der Textur, die für jeden Block der Rekonstruktion erstellt wird, beeinflusst die Bildwiederholrate und den Speicherverbrauch. In Abhängigkeit der Texturauflösung wird der Speicherbedarf bei einer Knotendichte von 2000 Knoten pro Kubikmeter und einem Caching-Radius von zwei Metern gemessen. Verschiedene Texturen mit einer Auflösung von 256 x 256, 512 x 512, 1024 x 1024 und 4096 x 4096 Texeln werden getestet. Die Messergebnisse sind in Abbildung 6.4 dargestellt. Mit zunehmender Texturgröße sinkt die Bildwiederholrate und der benötige Hauptspeicherplatz steigt. Eine Messung mit der Texturgröße von 4096 Texeln konnte nicht durchgeführt werden, da die Anwendung noch vor Berechnung aller 3600 Bilder mehr als 900 Megabyte Speicherplatz reserviert und automatisch geschlossen wird.

Je größer die Textur, desto mehr Bildinformationen müssen während der zweiten Ausführungsphase auf den Flash-Speicher geschrieben bzw. vom Flash-Speicher geladen werden. Die Texturprojektion dauert länger. Gleichzeitig steigt die Qualität der rekonstruierten Oberflächentextur. Die Qualitätsunterschiede zwischen einer Texturauflösung von 1024 x 1024 Texeln und 4096 x 4096 Texeln sind in Abbildung 5.22 sichtbar.

In Hinblick auf die Möglichkeit, einzelnen Blöcken eine höher aufgelöste Textur zuzuweisen, wird zugunsten des Speicherverbrauchs standardmäßig eine Texturauflösung von 1024 x 1024 Texeln verwendet.

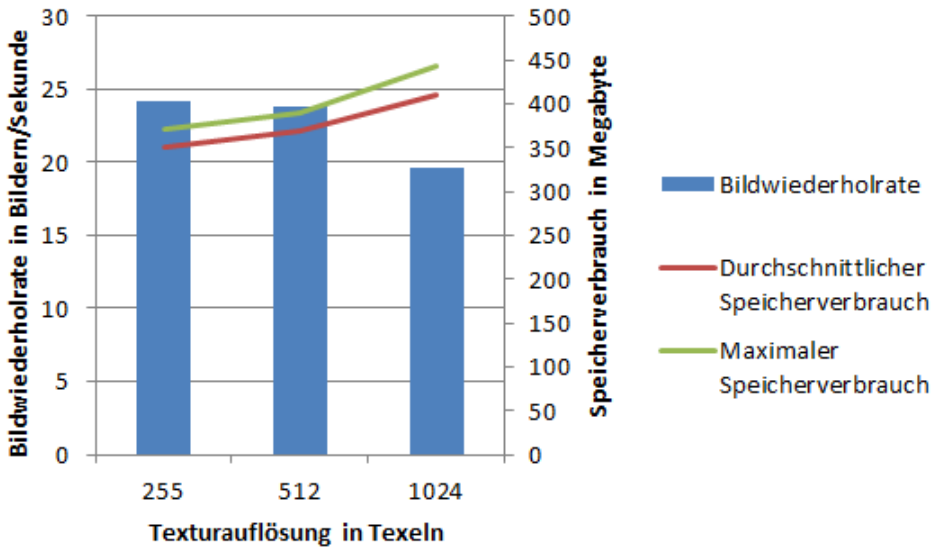


Abbildung 6.4: Durchschnittliche Bildwiederholrate in Bildern/Sekunde und durchschnittlicher und maximaler Speicherverbrauch in Megabyte in Abhängigkeit der Texturauflösung

7 Abschließende Diskussion und Fazit

Diese Arbeit beschreibt eine Anwendung, welche beliebig große Objekte und Umgebungen rekonstruiert darstellt. Erfassung, Verarbeitung und Visualisierung werden ausschließlich auf der Mixed Reality Brille Microsoft HoloLens ausgeführt.

Durch den Einsatz verschiedener Optimierungen ist es möglich, auch große Umgebungen in Echtzeit auf der HoloLens anzuzeigen. Sowohl die Topologie der Umgebung, als auch das die Materialerscheinung der Umgebungsoberflächen werden durch die Anwendung erfasst. Eine Beispiel einer rekonstruierten Wand ist in Abbildung 7.1 abgebildet. Andere Teilnehmer können sich über Netzwerk verbinden, sodass die rekonstruierte Umgebung auch auf anderen HoloLenses angezeigt werden kann.

Bezogen auf den Anwendungsfall einer virtuellen Bauteildokumentation, kann ein Benutzer das Bauteil erfassen. Das rekonstruierte Bauteil wird gleichzeitig an entfernte Begutachteter versendet. Der Gutachter empfängt die Rekonstuktion des Bauteils und kann sich um dieses bewegen und es so frei betrachten. Durch gezielte Fotoaufnahmen des Benutzers vor Ort werden mögliche Materialfehler oder Schäden sichtbar.

Abhängig von Umweltfaktoren, Form und Oberfläche der zu erfassenden Objekte variiert jedoch die Qualität der Rekonstruktion. Ungenauigkeiten während der Vermessung des Raumes können zu einem fehlerhaften 3D-Modell führen. Eine anschließende Texturprojektion auf Basis eines solchen fehlerhaften Modells verfälscht die Rekonstruktion weiter, da Fotoinhalte auf nicht existente oder falsch platzierte Oberflächen projiziert werden. Obwohl bereits die Nutzung des Hauptspeichers mithilfe von Caching und der Auslagerung nicht sichtbarer Umgebungsteile, ist ein Speicherüberlauf der Anwendung möglich. Durch eine langsame Netzwerkverbindung zu anderen Teilnehmern einer Sitzung kann ebenfalls der Speicherverbrauch der Anwendung kontinuierlich steigen.

Um den Prototypen zukünftig zu verwenden, müssen diese Probleme behoben werden. Unregelmäßigkeiten im 3D-Modell könnten durch Analyse und Glättung der Modelloberfläche vermieden werden. Darüber hinaus könnte die Qualität der Texturinhalte durch ein verbessertes UV-Layout optimiert werden. Aktuell wird jedem Oberflächendreieck innerhalb der Rekonstruktion unabhängig vom Flächeninhalt eine gleiche Anzahl an Texturinformationen zugeordnet. Besonders große Dreiecksflächen sind nur sehr gering aufgelöst. In einem optimierten UV-Layout würde großen Dreiecken mehr Texel zugewiesen

werden, als kleineren Dreiecken. Da die Flächeninhalte der einzelnen Dreiecke miteinander verglichen werden müssten, wäre dieser Algorithmus deutlich komplexer. Die Berechnung eines solchen UV-Layouts könnte zukünftig mit besserer Hardware in Echtzeit durchgeführt werden.

Viele dieser Probleme ließen sich mit einer leistungsfähigeren Hardware lösen. Es ist anzunehmen, dass kommende Generationen von Mixed Reality Geräten die Umgebung mithilfe verbesserter Sensoren und Kameras genauer erfassen werden und ein präziseres Modell zur Verfügung stellen. Dies bedeutet allerdings auch, dass mehr Daten gleichzeitig verarbeitet werden müssen. Auch zukünftig muss daher zwischen Qualität der Rekonstruktion und Ressourcenverbrauch abgewägt werden. Die vorgestellten Optimierungen sind somit auch zukünftig noch relevant und können eingesetzt werden.

Das kollaborative Arbeiten mehrerer Teilnehmer könnte zukünftig ein wichtiger Schwerpunkt werden. Mit der aktuellen Implementierung konnte eine Übertragung der Rekonstruktion bereits realisiert werden. Weitere Daten, wie etwa die Position und Orientierung der anderen Teilnehmer relativ zu der gemeinsamen betrachteten Umgebung, könnten übertragen werden. Diese Daten müssen mit möglichst geringer Latenz versendet werden. Die Übertragungsrate über das Netzwerk muss daher optimiert werden.

Gelingt sowohl eine Verbesserung der Rekonstruktionsqualität, als auch eine stabile Erfassung und Darstellung der Umgebung mit einer hohen Bildwiederholrate, dann kann der vorgestellte Prototyp auch in anderen Forschungsbereichen eingesetzt werden.

Denkbar wäre zum Beispiel ein Einsatz der Telepräsenz in der Raumfahrt. Eine Umgebung im Weltraum wird durch einen Roboter erfasst, während gleichzeitig die Experten auf der Erde die erfasste Umgebung begutachten können. Die Wissenschaftler können sich in der rekonstruierten Umgebung frei bewegen und diese erforschen, ohne auf teure bemannte Raumfahrtmissionen angewiesen zu sein.

Abbildung 7.1: Rekonstruktion einer Wand bei einer Texturauflösung von 1024 x1024 Texeln und einer maximalen Knotendichte von 2000 Knoten pro Kubikmeter

Literatur

[Ada14] Dan Adams. *No Limits - Unity in Cross Industry Development.* 05. 06. 2014. URL: `https://blogs.unity3d.com/2014/06/05/no-limits-unity-in-cross-industry-development/` (Datum: 14. 09. 2018).

[Ale+17] D. S. Alexiadis u. a. „An Integrated Platform for Live 3D Human Reconstruction and Motion Capturing". In: *IEEE Transactions on Circuits and Systems for Video Technology* 27.4 (04/2017), S. 798–813.

[Aut14] Autodesk. *Introduction to UV mapping.* 09. 09. 2014. URL: `https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2015/ENU/Maya/files/UV-mapping-overview-Introduction-to-UV-mapping-htm.html` (Datum: 14. 09. 2018).

[Cro+15] Simon Crowle u. a. „Dynamic Adaptive Mesh Streaming for Real-time 3D Teleimmersion". In: *Proceedings of the 20th International Conference on 3D Web Technology.* Web3D '15. Heraklion, Crete, Greece: ACM, 2015, S. 269–277.

[DH18] Samuel Dong und Tobias Hollerer. „Real-Time Re-Textured Geometry Modeling Using Microsoft HoloLens". In: 03/2018, S. 231–237.

[FK03] Randima Fernando und Mark J. Kilgard. *The Cg tutorial [Elektronische Ressource] : the definitive guide to programmable real-time graphics. - Includes bibliographical references and index / by Randima Fernando, Mark J. Kilgard.* Safari Books Online. Boston, Mass. Addison Wesley Professional 2003 2003, 2003.

[Gar+16] M. Garon u. a. „Real-Time High Resolution 3D Data on the HoloLens". In: *2016 IEEE International Symposium on Mixed and Augmented Reality (ISMAR-Adjunct).* 09/2016, S. 189–191.

[Gou71] H. Gouraud. „Continuous Shading of Curved Surfaces". In: *IEEE Trans. Comput.* 20.6 (06/1971), S. 623–629.

[Len11] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Third Edition.* 3rd. Boston, MA, United States: Course Technology Press, 2011.

[MF12] A. Maimone und H. Fuchs. „Real-time volumetric 3D capture of room-sized scenes for telepresence". In: *2012 3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON).* 10/2012, S. 1–4.

[Mic18a] Microsoft. *Compute Shader Overview*. 31. 05. 2018. URL: https://docs.microsoft.com/de-de/windows/desktop/direct3d11/direct3d-11-advanced-stages-compute-shader (Datum: 14. 09. 2018).

[Mic18b] Microsoft. *Development overview*. 21. 03. 2018. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/development-overview (Datum: 14. 09. 2018).

[Mic18c] Microsoft. *Geometry Shader Stage*. 31. 05. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/geometry-shader-stage (Datum: 14. 09. 2018).

[Mic18d] Microsoft. *HoloLens hardware details*. 21. 03. 2018. URL: https://developer.microsoft.com/en-us/windows/mixed-reality/hololens_hardware_details (Datum: 14. 09. 2018).

[Mic18e] Microsoft. *HoloLens Research mode*. 05. 03. 2018. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/research-mode (Datum: 14. 09. 2018).

[Mic18f] Microsoft. *Locatable camera*. 21. 03. 2018. URL: https://developer.microsoft.com/en-us/windows/mixed-reality/locatable_camera#device_camera_information (Datum: 14. 09. 2018).

[Mica] Microsoft. *Mixed-Reality-Toolkit Unity*. URL: https://github.com/Microsoft/MixedRealityToolkit-Unity (Datum: 14. 09. 2018).

[Mic18g] Microsoft. *Performance recommendations for HoloLens apps*. 21. 03. 2018. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/performance-recommendations-for-hololens-apps (Datum: 14. 09. 2018).

[Mic18h] Microsoft. *Pipeline Stages (Direct3D 10)*. 31. 05. 2018. URL: https://msdn.microsoft.com/en-us/library/bb205123(VS.85).aspx (Datum: 14. 09. 2018).

[Mic18i] Microsoft. *Rendering in DirectX*. 21. 03. 2018. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/rendering-in-directx (Datum: 14. 09. 2018).

[Micb] Microsoft. *Sharing*. URL: https://github.com/Microsoft/MixedRealityToolkit-Unity/tree/master/Assets/HoloToolkit/Sharing (Datum: 14. 09. 2018).

[Mic18j] Microsoft. *Spatial mapping*. 21. 03. 2018. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/spatial-mapping (Datum: 14. 09. 2018).

[Mic18k] Microsoft. *Spatial mapping design*. 21. 03. 2018. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/spatial-mapping-design (Datum: 14. 09. 2018).

[Micc] Microsoft. *SpatialSurfaceObserver Class*. URL: https://docs.microsoft.com/en-us/uwp/api/windows.perception.spatial.surfaces.spatialsurfaceobserver (Datum: 14. 09. 2018).

[MA] Elisabeth Mittelbach und Prof. Dr. Alin Olimpiu Albu-Schäffer. *Telepräsenz und On-Orbit-Servicing: DLR und ESA kooperieren in der Weltraumrobotik*. URL: https://www.dlr.de/dlr/desktopdefault.aspx/tabid-10212/332_read-10406/year-all/#/gallery/14858 (Datum: 14. 09. 2018).

[Nis+11] Alfred Nischwitz u. a. *Computergrafik und Bildverarbeitung*. Vieweg +Teubner Verlag und Springer Fachmedien Wiesbaden GmbH 2011, 2011.

[Nit] Nition. *UnityOctree*. URL: https://github.com/Nition/UnityOctree (Datum: 14. 09. 2018).

[Ong17] Sean Ong. *Beginning Windows Mixed Reality Programming*. Apress, 2017.

[PF05] Matt Pharr und Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[Plu15] Pluralsight. *Unity, Source 2, Unreal Engine 4, or CryENGINE - Which Game Engine Should I Choose?* 05. 03. 2015. URL: https://www.pluralsight.com/blog/film-games/unity-udk-cryengine-game-engine-choose (Datum: 14. 09. 2018).

[Pol+99] M. Pollefeys u. a. „Hand-held acquisition of 3D models with a video camera“. In: *Second International Conference on 3-D Digital Imaging and Modeling (Cat. No.PR00062)*. 10/1999, S. 14–23.

[Rub16] Daniel Rubino. *Microsoft HoloLens - Here are the full processor, storage and RAM specs*. 02. 05. 2016. URL: https://www.windowscentral.com/microsoft-hololens-processor-storage-and-ram (Datum: 14. 09. 2018).

[SNB99] Carlos Saona-Vázquez, Isabel Navazo und Pere Brunet. „The visibility octree: A data structure for 3D navigation“. In: 23 (10/1999), S. 635–643.

[Ste+13] F. Steinbrucker u. a. „Large-Scale Multi-resolution Surface Reconstruction from RGB-D Sequences“. In: *2013 IEEE International Conference on Computer Vision*. 12/2013, S. 3264–3271.

[Sto14] Jon Story. *Don't be conservative with Conservative Rasterization*. 19. 11. 2014. URL: `https://developer.nvidia.com/content/dont-be-conservative-conservative-rasterization` (Datum: 14. 09. 2018).

[SSF] Klaus Strobl, Wolfgang Sepp und Stefan Fuchs. *DLR CalDe and DLR CalLab*. URL: `http://www.dlr.de/rm/en/desktopdefault.aspx/tabid-3925/6084_read-9201/` (Datum: 14. 09. 2018).

[Unia] Unity. *Erfahrener Programmierer, aber neu bei Unity? Sie sind anderen bereits voraus.* URL: `https://unity3d.com/de/programming-in-unity` (Datum: 14. 09. 2018).

[Unib] Unity. *Multiplayer Overview*. URL: `https://docs.unity3d.com/Manual/UNetOverview.html` (Datum: 14. 09. 2018).

[Unic] Unity. *Render Texture*. URL: `https://docs.unity3d.com/Manual/class-RenderTexture.html` (Datum: 14. 09. 2018).

[Unid] Unity. *Sparse Textures*. URL: `https://docs.unity3d.com/Manual/SparseTextures.html` (Datum: 14. 09. 2018).

[Unie] Unity. *Spatial Mapping components*. URL: `https://docs.unity3d.com/2017.3/Documentation/Manual/windowsholographic-sm-component.html` (Datum: 14. 09. 2018).

[Unif] Unity. *SurfaceData.trianglesPerCubicMeter*. URL: `https://docs.unity3d.com/ScriptReference/XR.WSA.SurfaceData-trianglesPerCubicMeter.html` (Datum: 14. 09. 2018).

[Unig] Unity. *Unity*. URL: `https://unity3d.com/unity#platforms` (Datum: 14. 09. 2018).

[Vul] Vulcan. *HoloLensCameraStream for Unity*. URL: `https://github.com/VulcanTechnologies/HoloLensCameraStream` (Datum: 14. 09. 2018).

[Wal92] G. K. Wallace. „The JPEG still picture compression standard“. In: *IEEE Transactions on Consumer Electronics* 38.1 (02/1992), S. xviii–xxxiv.

[Wul18] J. Wulkop. „Texturprojektion von Fotografien auf virtuelle Objekte für Mixed-Reality Anwendungen“. Unpublished. 04/2018.

[Zuc] Alan Zucconi. *A Gentle Introduction to Shaders*. URL: `https://unity3d.com/de/learn/tutorials/topics/graphics/gentle-introduction-shaders` (Datum: 14. 09. 2018).