THM TECHNISCHE HOCHSCHULE MITTELHESSEN | CAMPUS FRIEDBERG | IEM Informationstechnik-Elektrotechnik-Mechatronik

Verwendung der Unreal Engine für das Rendern hochauflösender fotorealistischer Ansichten

Studiengang Medieninformatik

Bachelorarbeit

vorgelegt von

Björn Crombach
geb. in Gießen

durchgeführt bei der Firma
mi360 GmbH, Lich

Referent der Arbeit:	Hans Christian Arlt, M.Sc.
Korreferent der Arbeit:	Prof. Dr. Cornelius Malerczyk

Friedberg, 2022

Danksagung

An dieser Stelle möchte ich mich bei meinem Referenten M.Sc Hans Christian Arlt und bei meinen Korreferenten Prof. Dr. Cornelius Malerczyk, für das Betreuen dieser Arbeit bedanken.

Weiterhin möchte ich mich bei meinem Vater Peter Crombach für das Korrekturlesen der Arbeit bedanken und bei meinem Bruder Adrian Crombach für das Bereitstellen des Programms zur Auswertung der Metriken.

Selbstständigkeitserklärung

Ich erkläre, dass ich die eingereichte Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Friedberg, Februar 2022

Björn Crombach

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Für Visualisierungen in den unterschiedlichsten Bereichen, wie beispielsweise in der Werbebranche, werden optisch ansprechende Bilder verwendet. Damit bekommt z. B. der Kunde einen besseren Eindruck von dem beworbenen Produkt. Dabei ist die Vielfalt der Bilder sehr groß durch die enorme Anzahl der Möglichkeiten, wo diese zum Einsatz kommen können. So reichen diese von quadratischen Bildern für Instagram über Bilder für große Werbedisplays bis hin zu hochauflösenden Bildern, welche für VR oder virtuelle Panorama-Touren eingesetzt werden. Je nach Zweck werden auch verschiedene Bildstile verwendet. So eignen sich realistische Bilder gut, um ein Produkt realitätsnah abzubilden, wohingegen ein stilisiertes Bild sich besser dafür eignet, um auf etwas aufmerksam zu machen. Solche Grafiken können auf verschiedenen Wegen erstellt werden. Beispielsweise durch Fotografie und Bildbearbeitung oder durch das Rendern von 3D-Modellen. Heutzutage sind 3D-Renderings in Zusammenspiel mit Bildbearbeitung aus dieser Branche nicht mehr wegzudenken. In vielen Fällen eignen sich 3D-Renderings besonders gut, um etwas zu visualisieren, was es noch nicht in der Wirklichkeit gibt, wie z.B eine Wohnung in einem noch nicht errichteten Neubau.

Für die Erstellung solcher 3D-Renderings werden Programme wie Blender, Maya oder 3ds Max eingesetzt. Dort werden 3D-Modelle erzeugt oder modifiziert und schließlich gerendert. Diese Programme verfügen über integrierte Render-Engines, welche aus der gegebenen 3D-Szenenbeschreibung ein 2D-Bild generieren können.

1.2 Problemstellung

Das Rendern von solchen 3D-Objekten oder vollständigen 3D-Szenen gestaltet sich häufig als ein zeitaufwendiger Prozess. Zusätzlich beeinflusst auch die Komplexität von 3D-Objekten, und damit auch der daraus zusammen gesetzten Szenen die Rechenzeit häufig erheblich. Zudem ergibt sich in der Praxis eine Tendenz zu immer höherer Auflösung der Bilder, wodurch eine weitere Zunahme an Rechenzeit verursacht wird. Ein Beispiel eines solchen Bildes

kann in Abbildung 1.1 gesehen werden. Oftmals ist deswegen die Rechenzeit für den Rendervorgang ein Problem. Man kommt leicht in den Bereich von Stunden oder Tagen für ein einzelnes Bild in finaler Qualität. Dementsprechend ist man hier auf der Suche nach anderen Ansätzen, um diesen Vorgang zu beschleunigen. Eine Möglichkeit wäre es, deutlich leistungsstärkere Hardware einzusetzen, dem sind aber in der Praxis wegen der hohen Kosten schnell Grenzen gesetzt. Weiterhin ist es nicht möglich, mit allen Softwarelösungen bestimmte leistungsstärkere Hardware zu nutzen. Zusätzlich verursachen immer komplexere Szenen und immer höhere Auflösungen erhebliche Mehrkosten für die Erstellung der Bilder. Die daraus resultierenden Kosten wirken einer stärkeren Verbreitung dieser ansonsten für Visualisierungen sehr vorteilhaften Technologie entgegen.

Neben den herkömmlichen Methoden gibt es seit einigen Jahren alternative Methoden, beispielsweise durch Game Engines, welche möglicherweise auch für die Erzeugung von fotorealistischen Bildern aus 3D Szenen genutzt werden können. Zu diesem erst vor wenigen Jahren relevanter gewordenen Thema gibt es nur wenig wissenschaftliche Literatur, sodass Bedarf an einem Vergleich solcher alternativen Methoden zur herkömmlichen Rendermethode besteht. Zusätzlich muss bei jeder in Betracht gezogenen Lösungvariante evaluiert werden, welche Vor- und Nachteile sich damit ergeben. Zudem stellt sich die Frage, an welchen Stellen Nachteile gegenüber dem herkömmlichen Verfahren entstehen und ob diese im Endergebnis vertretbar sind.

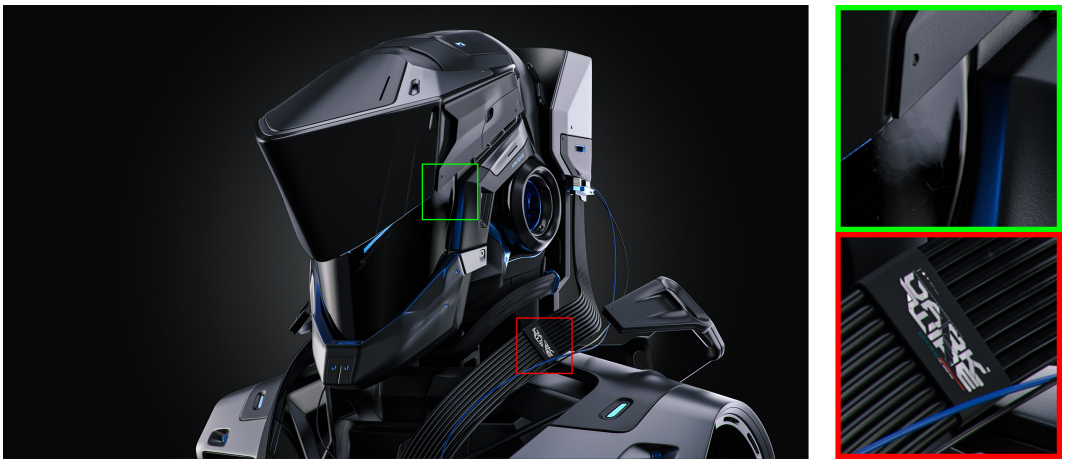


Abbildung 1.1: Zeigt ein Beispiel eines hochauflösenden 3D-Renderings, auf dem ein Roboter zu sehen ist. Rechts sind Bereiche aus dem Bild vergrößert, um die Auflösung hier besser darzustellen. (Quelle: Eigenes Rendering, modelliert anhand von Anleitung und Design von Mario Brajdich: https://www.artstation.com/artwork/8lk0Eq , Stand: 29. April 2022)

1.3 Zielsetzung

Ein interessanter Ansatz ist die Nutzung von Game Engines für das Rendern von Bildern oder sogar für die Umsetzung von interaktiven Visualisierungen, welche in Echtzeit lauffähig sind. Eine Game Engine ist eine Softwareumgebung zum Entwickeln von Computerspielen. Für Visualisierungen wird davon jedoch hauptsächlich die Render-Engine benutzt. Dieser Ansatz ist sehr vielversprechend, da Game Engines wie Unity oder Unreal Engine 4 in der Lage sind, fotorealistische Bildqualität auf anderen Wegen schneller zu erreichen als die herkömmlich genutzten Verfahren. Ein Beispiel für den Realismusgrad, der erreicht werden kann, ist in Abbildung 1.2 zu sehen.

Abbildung 1.2: Links ist ein in Unreal Engine 4 gerendertes Bild zu sehen und Rechts ein echtes Foto als Referenz. (Quelle: http://www.acidarts.fr/projects/schooldesk-photography.html , Stand: 10. März 2022)

Dieser Trend ist auch den Entwicklern der Game Engines bekannt, weshalb immer mehr Funktionen für andere Industriezweige wie beispielsweise die Filmindustrie oder die Architekturvisualisierung entwickelt werden. Da diese Art von Nutzung einer Game Engine in diesem Feld noch sehr neu ist, stellt sich hier die Frage, wie eine Game Engine überhaupt zu einem Ergebnis kommen kann, welches vergleichbar mit den herkömmlichen Verfahren ist.

Eine mögliche Lösung für das Problem des Renderns von hochaufgelösten Bildern wäre die Verwendung der Unreal Engine 4. Wie Abbildung 1.2 zeigt können damit fotorealistische Ergebnisse erreicht werden. In dieser Bachelorarbeit wird daher die Unreal Engine 4 mit einem gängigen Offline-Renderer, hier Autodesk Arnold verglichen. Der Vergleich sollte zeigen warum eine Game Engine im generieren von fotorealistischen Bildern schneller als ein

Offline-Renderer sein kann. Hierzu müssen die notwendigen Grundkonzepte der verwendeten Renderverfahren analysiert und verglichen werden, damit Gemeinsamkeiten und Unterschiede aufgezeigt werden können. Außerdem sollte ersichtlich werden, ob es Sinn macht anstelle von Autodesk Arnold, Unreal Engine 4 einzusetzen beim Rendern von hochaufgelösten Bildern. Es bietet sich an, anhand von praktischen Vergleichen die Unterschiede zu evaluieren. Um die Ergebnisbilder nicht allein visuell zu beurteilen, wäre die Verwendung einer objektiven Methode, also einer Metrik, sinnvoll. Mindestens sollte es möglich sein, den Grad des Rauschens rechnerisch zu bestimmen.

Kapitel 2

Grundlagen

2.1 Was ist eine Game Engine

Eine Game Engine ist ein Framework, welches Komponenten für die Spieleentwicklung bereitstellt. Diese Komponenten decken mehrere Bereiche ab, darunter sind beispielsweise Physik, Audio oder Grafik[1]. Hierdurch müssen Entwickler nicht jedes mal wichtige Bausteine wie eine Rendering Engine von Grund auf selbst entwickeln. Dies spart Zeit und vereinfacht den Prozess, sodass auch kleineren Studios oder gar Einzelpersonen die Entwicklung von hochwertigen Spielen ermöglicht werden kann. Häufig stellen diese Frameworks auch ihre eigenen Entwicklungsumgebungen bereit. Zudem erleichtern moderne Game Engines das entwickeln von Spielen bzw. Anwendungen für mehrere Zielplattformen. Früher wurden Spiele meist nur für eine oder wenige Zielplattformen entwickelt, was dazu führte, das sehr gezielt entwickelt und optimiert werden konnte. Heutzutage ist es keine Seltenheit das Spiele und Anwendungen auf möglichst vielen Zielplattformen gut lauffähig sein sollen.

Das Prinzip einer Game Engine ist keine neue entwicklung, schon in den 80er-Jahren gab es sogenannte Game creation systems welche dem Prinzip einer Game Engine sehr nahe kamen [Ste17]. Bekannte Beispiele hierfür sind das Pinball Construction Set[2] von Bill Budge aus 1983 oder das Adventure Construction Set[3] von Stuart Smith aus 1984 (Abbildung 2.1). Der Begriff Game Engine kam erstmals in 1991 auf, in Verbindung mit dem id Software Spiel DOOM (Rechts in Abbildung 2.1). In einer Pressemitteilung von 1993 wurde zusammen mit dem Spiel DOOM, das Konzept der Game Engine von id Software veröffentlicht [Low14].

Heutzutage gibt es Game Engines, die opensource sind wie Godot oder die in 2021 veröffentlichte Open3D Engine, Game Engines welche frei verfügbar und bis zu einem gewissen grad kostenlos sind, wie Unity oder Unreal Engine und Game Engines, welche von größeren Spieleentwicklern intern genutzt werden wie Frostbyte oder id Tech 7.

[1] https://usk.de/alle-lexikonbegriffe/game-engine/ , Stand: 17. März 2022
[2] /en.wikipedia.org/wiki/Pinball_Construction_Set , Stand: 17. März 2022
[3] https://en.wikipedia.org/wiki/Adventure_Construction_Set , Stand: 17. März 2022

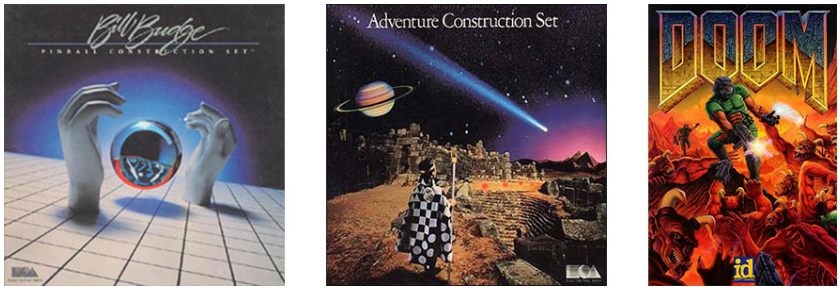


Abbildung 2.1: Coverart der Pinball und Adventure Construction Sets und DOOM. (Quellen: Links nach rechts: en.wikipedia.org/wiki/Pinball_Construction_Set , Stand: 17. März 2022 / en.wikipedia.org/wiki/Adventure_Construction_Set , Stand: 17. März 2022 / doomwiki.org/wiki/File:Doom.jpg , Stand: 17. März 2022)

2.1.1 Historie Unreal Engine

In 1995 begann Tim Sweeney, der Gründer von Epic Games, mit der Entwicklung der Unreal Engine. Diese wurde im Jahr 1998 zusammen mit dem Spiel Unreal veröffentlicht[4]. Zu ihrem Release nutze die Engine Softwarerendering, da dedizierte Grafikkarten noch nicht verbreitet waren. Eine weitere Besonderheit zu ihrer Zeit war das visuelle Toolset und die Möglichkeit der objektorientierten Programmierung. In 2002 wurde Unreal Engine 2 veröffentlicht zusammen mit dem Spiel Americas Army (ARMA). Diese Version der Engine war Single-threaded und unterstützte DirectX 7 Grafik, weiterhin bot sie die Möglichkeit, für Konsolen wie die Playstation 2 oder die Xbox zu entwickeln. In 2006 wurde die dritte Version der Engine veröffentlicht. Sie realisierte Multithreading mit bis zu 6 Threads, DirectX 9 Grafik und weitere visuelle Toolsets [SG09]. Bekannte spiele, die mit dieser Version realisiert wurden, sind Mass Effekt, Bioshock und Gears of Wars. Zu diesem Zeitpunkt umfasste die Engine bereits etwa 2,000,000 Zeilen C++ Code [SG09]. In 2014 wurde Version 4 veröffentlicht, diese konnte über ein Abo-Modell genutzt werden. Dies wurde jedoch nur ein Jahr später in 2015 geändert[5]. Seitdem ist die Unreal Engine für jeden bis zu einem gewissen Umsatz kostenlos verfügbar. Mit Version 4 wurde Support für neuere Plattformen wie die Playstation 4 und die Xbox One ermöglicht. Weiterhin wurden neue Features wie der Marketplace und Blueprints, welche visual scripting ermöglichen, eingeführt. In Version 4.22 wurde DirectX 12 Echtzeit Raytracing eingeführt. Heute ist die aktuellste Version 4.27, jedoch ist bereits eine Beta der Versuib 5 frei verfügbar.

Schon bei früheren Versionen der Unreal Engine wurde das Potential gesehen, sie für Visualisierungen einzusetzen [FK+04]. Diese Art von Nutzung wurde besonders mit der 4. Version interessant, wegen des hohen Realismusgrad der erreicht werden kann. Bereits im Jahr des Release gab es eindrucksreiche Echtzeit Visualisierungen mit der Unreal Engine 4[6]. Ein weiterer neuer Anwendungsfall ist virtuelle Produktion in der Filmindustrie[7].

[4] https://en.wikipedia.org/wiki/Unreal_Engine , Stand: 18. März 2022
[5] https://www.unrealengine.com/en-US/blog/ue4-is-free , Stand: 19. März 2022
[6] www.ronenbekerman.com/unreal-engine-4-and-archviz-by-koola/ , Stand: 18. März 2022
[7] https://www.filmundtvkamera.de/technik/der-gamechanger/ , Stand: 18. März 2022

2.2 Unreal Engine

In den folgenden Abschnitten wird auf grundlegende Funktionen und Eigenschaften der Unreal Engine eingegangen, diese sind notwendig für die in Kapitel 5 beschriebene Umsetzung.

2.2.1 Beleuchtung

In Unreal Engine sind Directional, Point, Rect, Spot und Skylights verfügbar. Zusätzlich haben alle Lichtquellen eine Mobility Einstellungen, welche sich drastisch auf die Lichtquelle auswirkt, hierbei gibt es Static, Stationary und Movable zur Auswahl. Wird ein Licht auf Movable eingestellt, wird es zu einer dynamischen Lichtquelle, welche in Echtzeit bewegt werden kann. Die Kalkulationen für Licht und Schatten erfolgen hierbei in Echtzeit und sind deswegen als teuer zu betrachten beim Entwickeln von Echtzeitanwendungen. Zusätzlich unterstützen dynamische Lichtquellen keine indirekte Beleuchtung. Die performanteste Einstellung für Echtzeitanwendungen ist Static. Lichter mit dieser Einstellung werden vorberechnet und in einer Textur gespeichert, diese Textur wird Lightmap genannt. Static hat jedoch den Nachteil, das die Lichtquelle nicht einfach bewegt oder verändert werden kann, eine Veränderung der Lichtquelle erfordert eine Neuberechnung der Lightmap. Stationary ist ein Kompromiss aus Static und Movable. Lichtquellen mit dieser Einstellung nutzen eine Lightmap für die indirekte Beleuchtung und berechnen die direkte Beleuchtung in Echtzeit. Zusätzlich sind bei dieser Mobility Einstellung Änderungen wie beispielsweise die Lichtfarbe oder Intensität in Echtzeit möglich, diese Änderungen gelten jedoch nur für den dynamischen Anteil des Lichts[8], siehe Abbildung 2.2. Außerdem können sich maximal bis zu 4 Stationäre Lichtquellen gleichzeitig in ihrem Beleuchtungsbereich überlappen. Wird dieses Limit überschritten, verhält sich die Lichtquelle mit dem kleinsten Radius wie eine dynamische Lichtquelle.

(a) Stationary Lichtquelle und Lightmap

(b) Änderung der Lichtfarbe

(c) Änderung der Lichtfarbe und Lichtintensität

Abbildung 2.2: Auswirkungen von nachträglichen Änderung der eingestellten Parameter an einer Stationary Lichtquelle.

[8] https://www.techarthub.com/lighting-companion-light-mobility-in-ue4-explained/ , Stand: 16. März 2022

Eine Lightmap ist eine Textur, welche vorberechnete Licht Informationen enthält (Abbildung 2.3). Sie wird auf die Base color der Objekte, welche die Lightmap nutzen, multipliziert. Hierdurch ist es möglich, Effekte wie die diffuse Reflexion (Indirekte Beleuchtung) und Area Shadows für Echtzeit Anwendungen umzusetzen. Diese Technik ist essenziell, damit Computerspiele oder andere Echtzeitanwendungen mit realistischer Beleuchtung in Echtzeit lauffähig sind. Da diese Lösung texturbasiert ist, wird ein UV-Layout für die Lightmap benötigt. Das Vorberechnen dieser Lightmaps ist in der Unreal Engine über den Lightmass Renderer möglich. Lightmass ist ein eigenständiges Programm, welches über den sogenannten Swarm Agent mit Unreal Engine kommuniziert. Dieser ermöglicht es ebenfalls, die Berechnung der Lightmaps auf ein Computernetzwerk aufzuteilen.

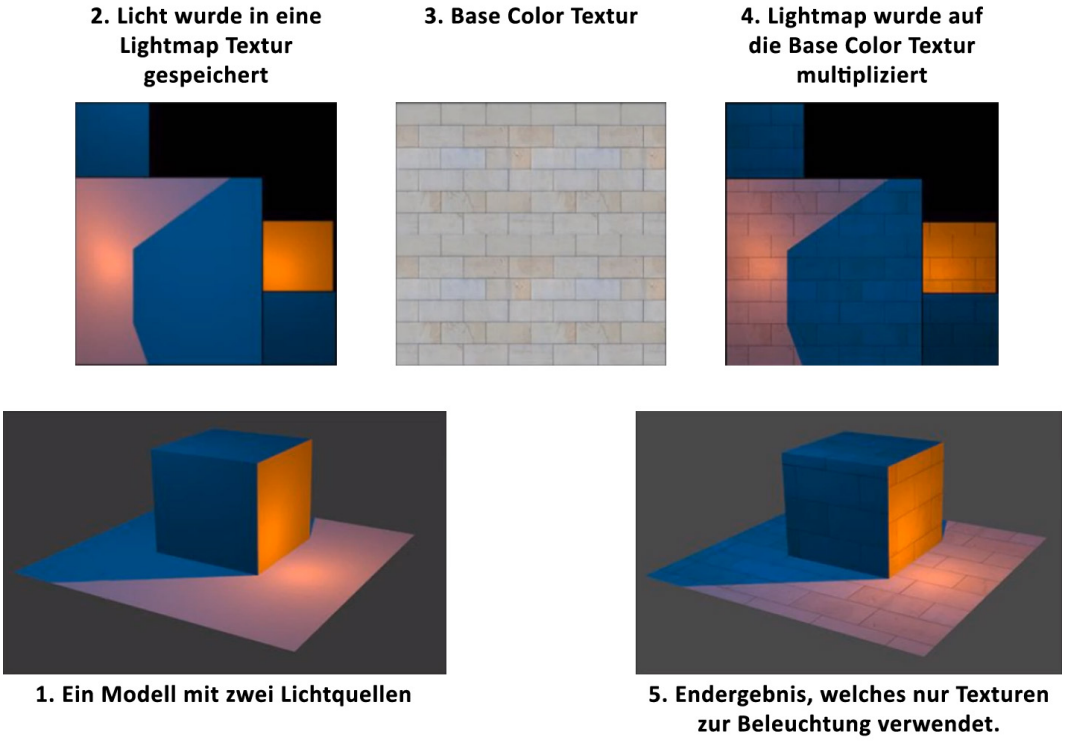


Abbildung 2.3: Zeigt das Prinzip wie eine Lightmap angewendet wird. (Quelle: Screenshot mit angepasstem Layout und Text aus dem folgenden Video: https://dev.epicgames.com/community/learning/courses/EGR/an-in-depth-look-at-real-time-rendering/YXn/static-lighting , Stand: 07. April 2022

2.2.2 Materialien

Materialien können über den Materialeditor (siehe Abbildung 2.4) erstellt und konfiguriert werden, dieser kann verglichen werden mit dem Hypershade Editor in Maya. Es können wie im Hypershade Editor Netzwerke aus Knoten, welche mathematische Funktionen bereitstellen oder Texturen verwalten, erstellt werden. Diese vom Benutzer erstellten Netzwerke werden dann Eigenschaften des Materials beeinflussen. Jeder Knoten besteht aus HLSL (High Level Shading Language) Code, dieser kann sichtbar gemacht werden durch kopieren und einfügen eines Knotens in einen Texteditor.

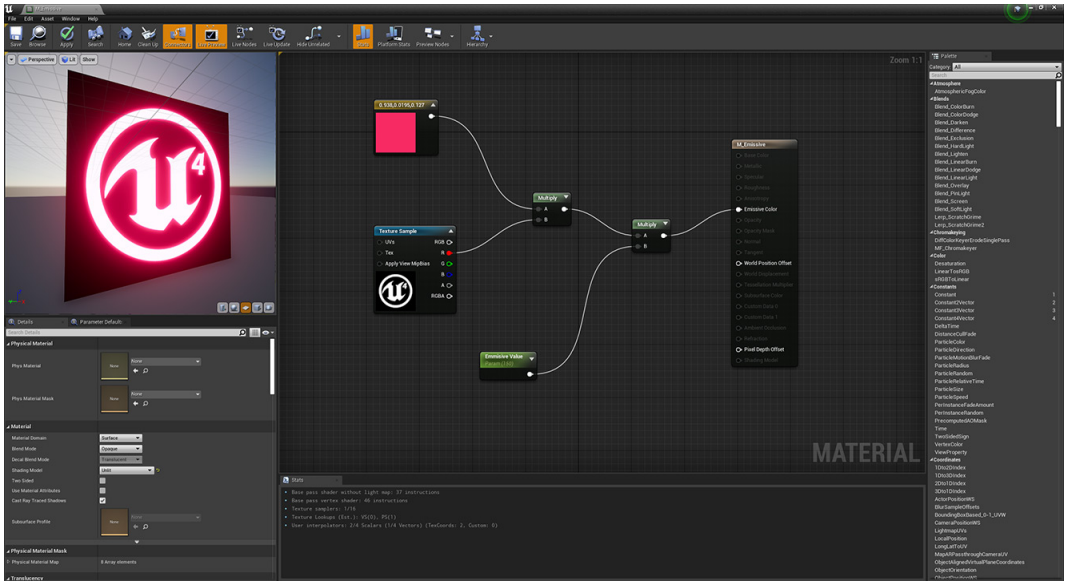


Abbildung 2.4: Zeigt die Oberfläche des Materialeditors mit einem einfachen Material. (Quelle: Eigener Screenshot aus der Unreal Engine 4.)

Materialien verwenden Pixel und Vertex Shader, diese sind sozusagen Programme, welche üblicherweise in HLSL geschrieben sind. Sie führen Kalkulationen pro Vertex bzw. pro Pixel aus. Der Ablaufprozess für Materialien in Unreal Engine lässt sich in fünf Abschnitte unterteilen, siehe Abbildung 2.5. Vorbereitete Shadertemplates welche in HLSL geschrieben sind, befinden sich in der Unreal Engine Installation, diese sind im Prinzip fertige Shader mit undefinierten Werten für Einstellungen wie beispielsweise ob das Objekt ein Metal ist oder nicht. Der Input für diese Werte geschieht durch den Benutzer mithilfe des Materialeditors. Es gibt verschiedene Shadertemplates mit Eigenschaften für unterschiedliche Einsatzzwecke[9], beispielsweise Blend Mode Einstellungen wie Translucent, Opaque oder Masked oder Shading Modell Einstellungen wie Default Lit, Subsurface oder Hair. Diese können im Materialeditor entsprechend ausgewählt werden und führen zu der Nutzung eines anderen

[9] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/MaterialProperties/LightingModels/ , Stand: 14. März 2022

Shadertemplates. Weiterhin kann eine Instanz[10] des Materials erstellt werden, welche Attribute, die der Benutzer vorher im Material spezifiziert hat, als Variable bereitstellt. Dies hat den Vorteil, dass durch eine Änderung der Instanz nicht das ganze Material neu kompiliert werden muss, somit lässt sich eine Änderung in Echtzeit schnell durchführen. Es gibt zwei Arten von Materialinstanzen, konstante Instanzen, welche vor der Laufzeit berechnet werden, und dynamische Instanzen, welche während der Laufzeit berechnet und somit auch verändert werden können. Das Materialinstanzensystem basiert auf dem Parent-Child Prinzip, aus einem Material können mehrere Instanzen generiert werden. Durch das Parent-Child Prinzip lassen sich effizient und einfach viele Variationen eines Materials erzeugen.

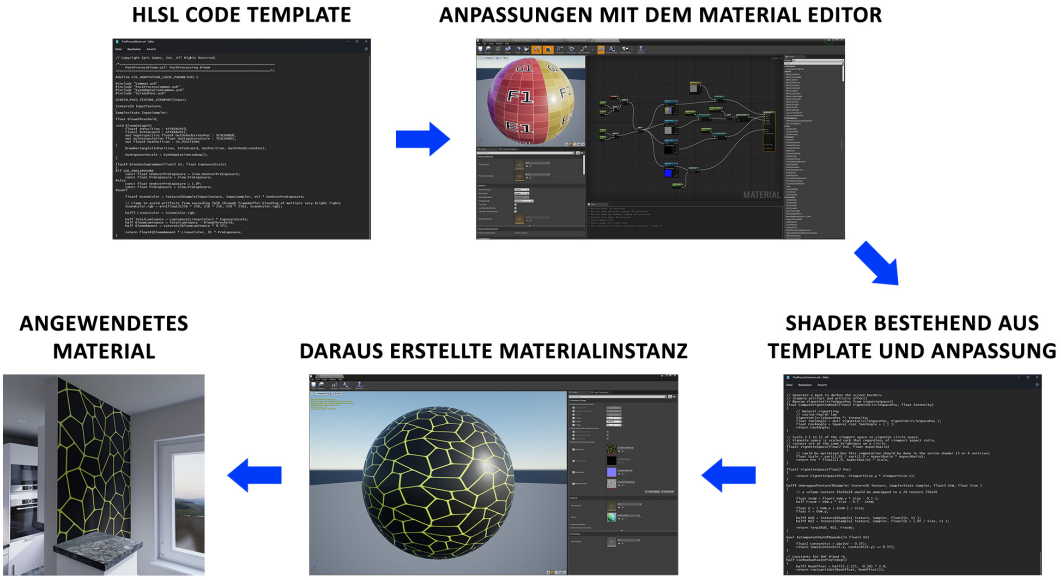


Abbildung 2.5: Zeigt den Ablauf wie ein Material in Unreal Engine erstellt und eingesetzt wird. (Quelle: Nachbau einer Grafik aus dem folgenden Video: https://dev.epicgames.com/community/learning/courses/2dy/materials-master-learning/jje/architecture , Stand: 07. April 2022)

Ein beliebter Workflow ist es, Materialien (auch Master Materialien genannt) anzulegen, welche ein breites Spektrum an Funktionen beinhalten wie beispielsweise Texture Sampler für die geläufigen Maps oder die Möglichkeit, Texturen zu wiederholen durch UV-Manipulationen [Ves14]. Dadurch müssen für die aller meisten Materialien, die im Verlauf des Projekts benötigt werden, nur Instanzen von diesen Master Materialien angelegt werden.

[10] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/MaterialInstances/ , Stand: 14. März 2022

2.2.3 Sequencer und Movie Render Queue

Zum Rendern von Bildern oder Videos in Unreal Engine ist die Verwendung des Sequencers (Abbildung 2.6) erforderlich. Über den Sequencer lassen sich Animationen mithilfe von Keyframes erstellen, zusätzlich kann der Sequencer über die render Movie Funktion diese Animationen als Video oder Bild Sequenz rendern. Hierfür muss eine Level Sequence angelegt werden und die notwendigen Actor wie beispielsweise die Kamera der Sequence hinzugefügt werden.

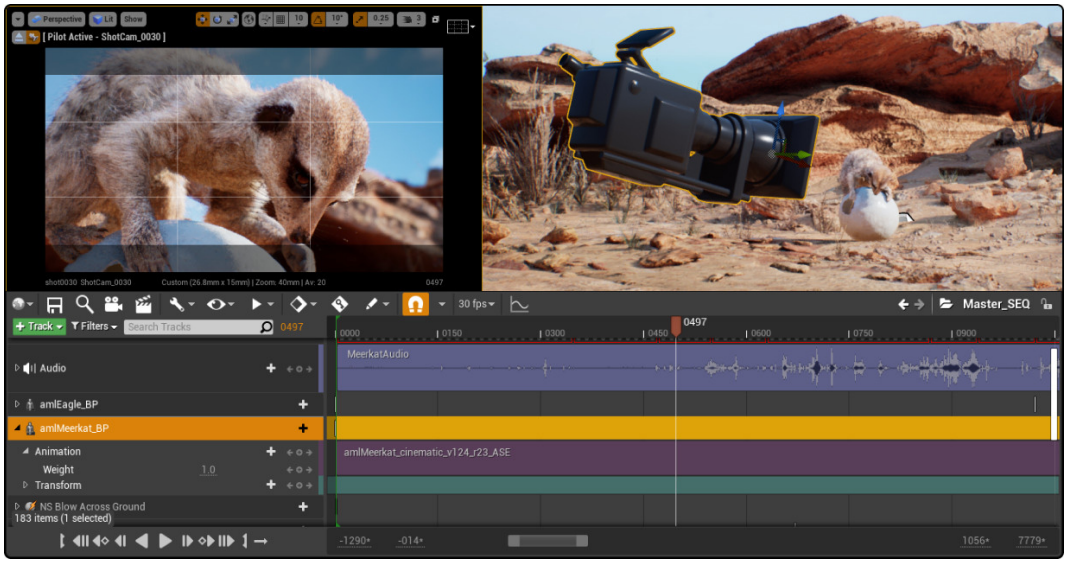


Abbildung 2.6: Zeigt im unteren Bildbereich die Benutzeroberfläche des Sequencers mit hinzugefügten Actors und einer Audiospur. (Quelle: https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/Sequencer/ , Stand: 15. März 2022)

Mit Version 4.25 wurde die Movie Render Queue als Plugin hinzugefügt und seit der Version 4.27 ist sie standardmäßig in der Engine aktiviert. Sie ist der Nachfolger[11] der Render Movie Funktion des Sequencers, welche aber immer noch als Legacy feature gekennzeichnet verfügbar ist. Durch sie können wie bei dem Sequencer Einzelbilder oder Bildsequenzen gerendert werden. Hierfür muss eine Level Sequence der Movie Render Queue hinzugefügt werden. Der größte Vorteil besteht darin, das sie ein qualitativ hochwertigeres Ergebnis erreichen kann durch die Verwendung von mehreren Samples, zudem bietet sie eine große Vielfalt an zusätzlichen Einstellmöglichkeiten. Weitere Vorteile sind die Möglichkeit, wie beispielsweise Bilder in 16 Bit Multilayer EXR Dateien zu speichern, sehr hochauflösende Renderings zu realisieren oder Batch Rendervorgänge von mehreren Level Sequences durchzuführen.

[11] https://docs.unrealengine.com/4.26/en-US/AnimatingObjects/Sequencer/Workflow/RenderAndExport/HighQualityMediaExport/ , Stand: 15. März 2022

Die Movie Render Queue ist so aufgebaut, das je nach Bedarf Einstellungen hinzugefügt werden können. Verändert man nichts an den Standardeinstellungen, wird das Ergebnis gleich sein mit den Ergebnissen der Render Movie Funktion des Sequencers. Ein Bild der Standard Einstellungen kann Links in Abbildung 2.7 gesehen werden. Eine Auflistung der verfügbaren Einstellungen ist in derselben Abbildung rechts zu sehen.

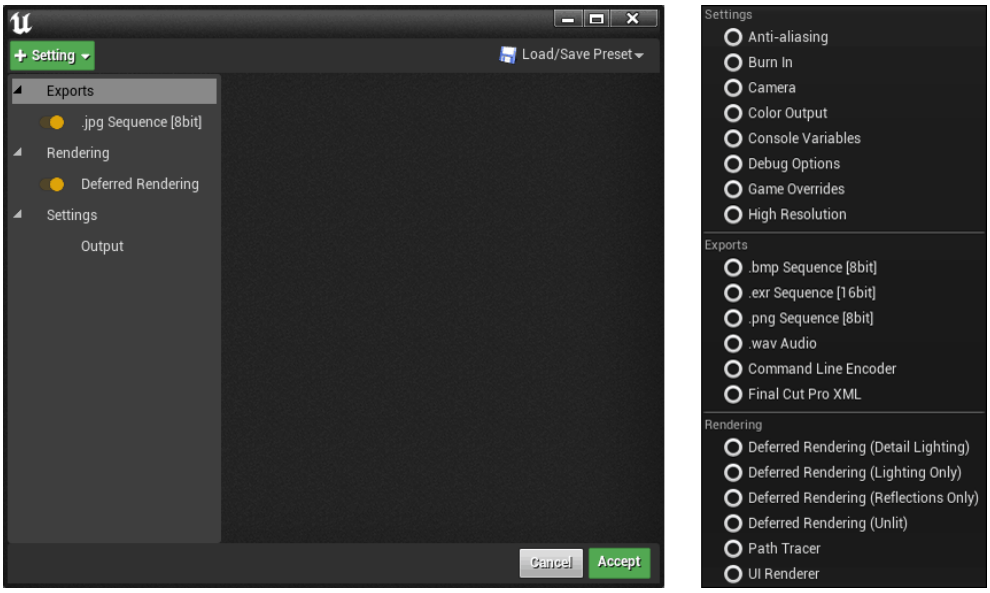


Abbildung 2.7: Links: die Default-Render-Einstellungen der Movie render Queue. Rechts: Auflistung der Einstellmöglichkeiten, welche hinzugefügt werden können. (Quelle: Eigener Screenshots aus der Unreal Engine 4.)

Eine sehr wichtige Einstellung, um die Bildqualität sichtbar zu verbessern, ist Anti-aliasing. über dieses Einstellungsmenü können Spatial oder Temporal Samples zu verwendet werden. Bei der Verwendung von Spatial Samples wird jeder Frame mehrmals gerendert mit leicht unterschiedlichen Kamerapositionen. Während dem Positionsoffset Vorgang vergeht keine Zeit. Mit einer größeren Zahl von Spatial samples lässt sich Rauschen reduzieren, Antialiasing verbessern und in Situationen, in denen motion blur mit sehr kurzer Dauer vorhanden ist, auch dieser verbessern. Temporal Samples nutzen die eingestellte Zeit, die der Kameraverschluss geöffnet ist, und teilen diese entsprechend in Zeitabschnitte ein, welche als einzelne Frames gerendert werden. Danach interpoliert die Engine die Zwischenschritte dieser Zeitabschnitte mithilfe des Motion blur der in der Engine implementiert ist. Bei diesem Vorgang vergeht weiterhin Zeit, daher der Name Temporal Samples[12]. Dieser Prozess führt zu einer sehr guten Darstellung von Motion blur, besonders bei rotierenden Objekten wie beispielsweise Autorädern, zusätzlich wird hierdurch wie bei Spatial Sampling das Antialiasing verbessert und Rauschen reduziert[13].

[12] https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/Sequencer/HighQualityMediaExport/RenderSettings/Reference/#anti-aliasing , Stand: 15. März 2022

[13] https://www.youtube.com/watch?v=XBuOotF-CxA , Stand: 16. März 2022

Um sehr hohe Auflösungen zu realisieren, gibt es die high Resolution Einstellungen. Diese teilen das Bild in mehrere niedriger aufgelöste Kacheln auf, welche dann einzeln gerendert und anschließend zusammengefügt werden zu einem hochaufgelösten Bild. Dieses Feature ist sehr nützlich, da ab einer Gewissen Szenen, Komplexität oder Render Auflösung es nicht mehr möglich ist, die Szene in hoher Auflösung zu rendern wegen unzureichendem Grafikspeicher. Jedoch hat dieses Feature auch Limitierungen, Screenspace Effekte wie convolution Bloom, Lens Flares und Screenspace Reflections werden nicht unterstützt[14].

Weiterhin können Konsolenkommandos[15] über die Movie Render Queue ausgeführt werden, welche zu aufwendig sind, um sie in Laufzeit zu verwenden. Diese Option ist besonders hilfreich für Szenen mit Raytracing, um die Samples der Raytracing Optionen noch weiter zu erhöhen.

2.3 Globale Beleuchtung

Globale Beleuchtung (GI) beschreibt Verfahren, welche zusätzlich zur direkten Beleuchtung der Objekte durch die Lichtquellen auch die indirekte Beleuchtung berücksichtigt, die sich durch Reflexionen an Objekten auf andere Objekte übertragen wird[16]. Verfahren, die globale Beleuchtung umsetzen, versuchen immer die Rendergleichung [Kaj86] zu lösen. Meist ist mit dem Begriff Globale Beleuchtung jedoch nur die diffuse Interreflexion gemeint. Die meisten Verfahren auf diesem Gebiet modellieren zusätzlich zur diffusen Interreflexion auch Spiegelreflexion mit Ausnahme des Radiosity Verfahrens. Da es rechenaufwendig ist, globale Beleuchtung zu berechnen, wurde früher und in Echtzeit Anwendungen die diffuse Interreflexion über ein sogenanntes ambient light oder auch umgebungslicht in der Rendergleichung abgebildet. Dieses hellt einheitlich die Szene auf, wodurch schwarze Flächen an dunklen Stellen verhindert werden, aber kein realistischer Eindruck entsteht. In den folgenden Abschnitten werden die Verfahren Raytracing, Pathtracing, Photonmapping und Radiosity erklärt, welche häufig verwendet werden, um globale Beleuchtung zu realisieren.

[14] https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/Sequencer/HighQualityMediaExport/RenderSettings/Reference/ , Stand: 16. März 2022

[15] https://digilander.libero.it/ZioYuri78/ , Stand: 16. März 2022

[16] https://hmn.wiki/de/Global_illumination , Stand: 03. April 2022

2.3.1 Raycasting und Raytracing

Ray Casting wurde erstmalig in 1968 von Arthur Appel in einer Veröffentlichung [App68] beschrieben, wo es darum ging, Schatten zu ermitteln für die Darstellung von Maschinen. Bei Ray Casting werden Strahlen durch die Bildebene in die Szene geschossen, bis sie auf ein Objekt auftreffen. Anschließend wird von dem Punkt, an dem der Strahl aufgetroffen ist, ein Strahl (Schattenstrahl) zur Lichtquelle geschossen, welcher überprüft, ob das Objekt direktes Licht abbekommt oder ob es von anderen Objekten verdeckt wird und somit im Schatten liegt. Raycasting wurde auch in vereinfachter Form in frühen First Person Shooter Computerspielen wie Wolfenstein 3D oder DOOM verwendet[17].

Das klassische Ray Tracing wurde in 1980 von Turner Whitted in seiner Veröffentlichung "An Improved Illumination Model for Shaded Display" [Whi05] beschrieben. Dabei werden Strahlen nach dem Ersten auftreffen auf einem Objekt weiter verfolgt. Strahlen können bei jedem auftreffen, reflektiert, absorbiert oder gebrochen werden[18]. Diese Strahlen werden in einer Baumstruktur verfolgt, welche rekursiv abgearbeitet wird. Bei jedem auftreffen wird wie bei Ray Casting mit einem Schattenstrahl überprüft, ob der Punkt direktes Licht abbekommt oder ob er verdeckt wird und im Schatten liegt. Da Lichtquellen als Punktlichtquellen angenommen werden und nur ein Schattenstrahl zur Lichtquelle eingesetzt wird, sind mit dieser Methode nur harte Schatten möglich.

In 1984 wurde dieses Verfahren von Robert L. Cook erweitert, welches dann unter dem Name stochastisches Raytracing [CPC84] bekannt wurde. Hierbei wurden anstelle von nur einem Strahl bei jedem auftreffen, mehrere Strahlen verwendet. Die Lichtquellen hatten dabei eine definierte Größe (Area Lights), weshalb es möglich wurde, weiche Schatten abzubilden. Durch das aussenden von mehreren Schattenstrahlen bei jedem auftreffen konnte ermittelt werden, wie viele dieser Strahlen die Lichtquelle treffen und wie intensiv der Schatten an dieser stelle sein muss. Zusätzlich wurden Effekte wie Motion Blur, Tiefenschärfe und matte Reflexionen mit dieser Technik möglich.

2.3.2 Pathtracing

Path Tracing ist ein auf Raytracing basierendes stochastisches Verfahren. Es nähert die in 1986 von James T. Kajiya beschriebene Rendergleichung [Kaj86] (2.1) mit Hilfe von Monte Carlo Integration an, weswegen es auch Monte-Carlo-Raytracing genannt wird. Die Rendergleichung beschreibt wie sich Licht im Raum ausbreitet und von Oberflächen gestreut wird. Bei Path Tracing werden mehrere Strahlen pro Pixel von der Kamera aus durch die Bildebene in die Szene geschossen, diese Strahlen werden Samples genannt. Die Strahlen werden bei jedem Auftreffpunkt reflektiert, absorbiert oder gebrochen und in der Szene weiter verfolgt. Anschließend werden all diese Samples zusammengefasst, um den Farbwert für

[17] https://lodev.org/cgtutor/raycasting.html , Stand: 05. April 2022

[18] https://developer.nvidia.com/blog/ray-tracing-essentials-part-1-basics-of-ray-tracing/ , Stand: 05. April 2022

den Pixel an dieser Stelle zu bilden. Werden zu wenige Samples verwendet, wird die Rendergleichung zu ungenau angenähert, was zu Bildrauschen führt. Ein visuelles Beispiel einer Monte-Carlo-Simulation mit zu wenig und mit ausreichend Samples kann in Abbildung 2.8 gesehen werden. In der Praxis verwendet man hier oftmals hunderte oder tausende Samples pro Pixel was zu einem enormen rechenaufwand führt. Path Tracing ist das Verfahren welches heutzutage in kommerziellen Offline Renderern wie Autodesk Arnold[19], in erweiterter Form zum Einsatz kommt.

$$L_o = L_e + \int_\Omega L_i \cdot f_r \cdot cos\theta \cdot d\omega \tag{2.1}$$

$$L_o(x,\, v) = L_e(x,\, v) + \int_\Omega L_i(x,\, \omega) \cdot f_r(x,\, \omega \rightarrow v) \cdot cos(\theta_x) \cdot d\omega \tag{2.2}$$

2.1 zeigt eine zusammengefasste Form der von James T. Kajiya aufgestellten Rendergleichung [Kaj86]. In 2.2 ist die Rendergleichung ausführlicher aufgeschrieben[20]. L_o beschreibt das ausgehende Licht von einem Punkt x in einer Richtung v. L_e beschreibt das Licht welches der Punkt x in die Richtung v selbst emittiert. $\int_\Omega$ sagt aus das die folgenden Terme über eine Hemisphere integriert werden. L_i ist der rekursive Term der das eingehende Licht an dem Punkt x in richtung v beschreibt. f_r beschreibt das Material bzw. die Oberflächenbeschaffenheit an dieser Stelle mittels einer BRDF. Der Lambert Term $cos(\theta_x)$ beschreibt die abschwächung und ausweitung des Lichts auf der Oberfläche bei ansteigendem Winkel.

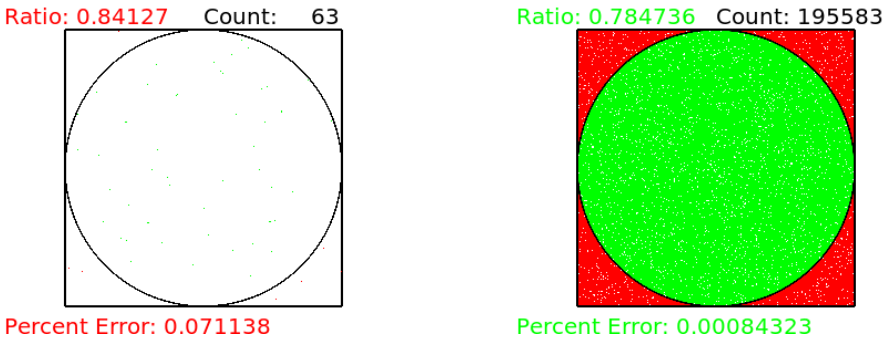


Abbildung 2.8: Bsp. einer Monte-Carlo Simulation mit 63 Samples und 195583 Samples. (Quelle: Ausschnitte aus dem .Gif von: https://www.algorithm-archive.org/contents/monte_carlo_integration/monte_carlo_integration.html , Stand: 06. April 2022)

[19] https://www.arnoldrenderer.com , Stand: 06. April 2022

[20] https://developer.nvidia.com/blog/ray-tracing-essentials-part-6-the-rendering-equation/ , Stand: 07. April 2022

2.3.3 Photonmapping

Photonmapping ist ein Verfahren, welches in 1995 von Hendrik Wann Jensen [Jen96] beschrieben wurde. Es arbeitet in zwei Abschnitten, zuerst werden Photonen von den Lichtquellen aus in die Szene emittiert, wo sie absorbiert, reflektiert oder refraktiert werden. Die Photonen werden in einer Datenstruktur festgehalten, welche sich global Photon Map nennt. Photonen, welche zu Kaustiken beitragen, werden in einer separaten Caustic Photon Map gespeichert, diese wird erstellt, in dem Photonen in Richtung von Spekularen Objekten emittiert werden. Die Photon Maps können für unterschiedliche Blickwinkel der Szene eingesetzt werden. Im zweiten Abschnitt wird ein Bild mittels Monte Carlo ray tracing gerendert, indem Strahlen aus der Kamera durch die Bildebene in die Szene geschossen werden. Sobald ein Strahl eine Oberfläche trifft, werden die Lichtintensitätswerte der benachbarten Photonen aus der Photon Map gemittelt und auf die Lichtintensität dieser Stelle angerechnet [YLS05]. Mit dieser Methode können besonders gut Kaustiken abgebildet werden, zusätzlich ist sie nicht vom Betrachtungswinkel abhängig. Die Nachteile dieses Verfahrens zeigen sich in langen Rechenzeiten für brauchbare Resultate und fleckigen Bildartefakten durch das Mitteln von Werten.

2.3.4 Radiosity

Mit dem Radiosity-Verfahren wird die Lichtausbreitung in einem Raum simuliert, damit lassen sich realistische Bilder erzeugen. Weiche Schatten und ausgedehnte Lichtquellen sind mit diesem Verfahren möglich. Das Radiosity-Verfahren wurde in 1984 in der Veröffentlichung "Modeling the Interaction of Light Between Diffuse Surfaces" [GTGB84] beschrieben. Die Szene inklusive der Lichtquellen wird durch sogenannte Patches beschrieben, welche einfache planare Polygone sind [DBB18]. Es wird angenommen, dass jeder Patch eine diffuse Oberfläche hat und dass der Reflexionsgrad und die Radiosity (Strahldichte) an jeder Stelle eines Patches den gleichen Wert hat. Berechnet wird die Radiosity B_i an dem Patch i, Sie berechnet sich aus der eigenen Abstrahlung E_i und der diffus reflektierten Strahlung R_i, die von allen anderen Patches kommt.

$$B_i = E_i + R_i \tag{2.3}$$

$$B_i = E_i + \rho_i \cdot \sum_{j=1}^{n} B_j \cdot F_{ij} \tag{2.4}$$

Die Radiosity-Formeln für die Gesamtzahl der Patches ergeben ein lineares Gleichungssystem, welches numerisch gelöst werden kann[21]. Als Ergebnis erhält man für alle Patches die Radiosity B_i. Diese Berechnung ist nicht abhängig vom Standpunkt des Betrachters. Zum Schluss wird die Szene aus der gewünschten Blickrichtung gerendert.

[21] https://www.cg.tuwien.ac.at/courses/CG1/textblaetter/11Radiosity.pdf , Stand: 06. Mai 2022

2.4 Lokale Beleuchtungsmodelle

Für eine realistische Darstellung von Objekten ist es notwendig die Beleuchtung dieser richtig zu berechnen. Lokale Beleuchtungsmodelle simulieren wie sich die Oberfläche von Objekten bei einfallendem Licht verhält. Somit sind Positionen, Orientierungen und Charakteristiken der Objektoberfläche und der Lichtquelle wichtig. Sie bilden so Farb- und Helligkeitswerte an der jeweiligen Stelle ab. Dazu verwendet jedes lokale Beleuchtungsmodell eine Bidirektionale Reflexionsverteilungsfunktion (BRDF).

Ein einfaches Modell ist das Lambert Modell [Lam60], welches in fast jedem 3D-Programm Anwendung findet. Es beschreibt nur komplett diffus reflektierende Oberflächen. Dabei wird das eingehende Licht so verteilt, dass das Flächenelement für den Betrachter aus allen Richtungen gleich hell erscheint[22] [HB97].

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = \frac{K_d}{\pi} \tag{2.5}$$

f_r beschreibt die BRDF, wobei θ_i, ϕ_i Zenit und Azimuth der Lichtquelle sind und θ_r, ϕ_r Zenit und Azimuth des Betrachters. K_d ist die diffuse Reflexionskonstante (diffuse Albedo), die beschreibt, wie viel Licht reflektiert wird. Sie liegt immer zwischen 0 und 1[23].

2.4.1 Cook Torrance Modell

Ein aufwändigeres, aber auch genaueres und vielseitigeres Modell, welches oft Anwendung findet ist das Cook Torrance Modell [CT82]. Mit ihm lässt sich das Reflexionsverhalten von Materialien unterschiedlicher Rauigkeit beschrieben, so können z.B. gut die Glanzlichter verschiedener Metalle und Kunststoffe dargestellt werden. Dem Modell liegt eine Aufteilung der Oberflächen in ebene Mikrofacetten zugrunde, die mehr oder weniger stark geneigt sind:

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = \frac{F_\lambda D G}{\pi(\vec{N} \cdot \vec{V})(\vec{N} \cdot \vec{L})} \tag{2.6}$$

Der Fresnelterm F_λ steuert den Reflexionsgrad, er ist abhängig vom Einfallswinkel bzgl. der Facettenebene und von der Wellenlänge. Die Funktion D beschreibt die Verteilung der Facettenneigungen relativ zur Gesamtfläche, somit beschreibt sie die Rauigkeit, und diese beeinflusst dann das Streuverhalten. Der geometrische Abschwächungsfaktor G berücksichtigt die gegenseitigen Beschattungen der Facetten. $\vec{N}$ ist der Normalenvektor der Fläche, $\vec{V}$ der Vektor in Richtung des Betrachters und $\vec{L}$ der Vektor in Richtung der Lichtquelle.

[22] https://www.irrlicht3d.org/papers/BrdfModelle.pdf , Stand: 07. Mai 2022
[23] https://graphics.tu-bs.de/upload/people/berger/PA_Schaefer.pdf , Stand: 08. April 2022

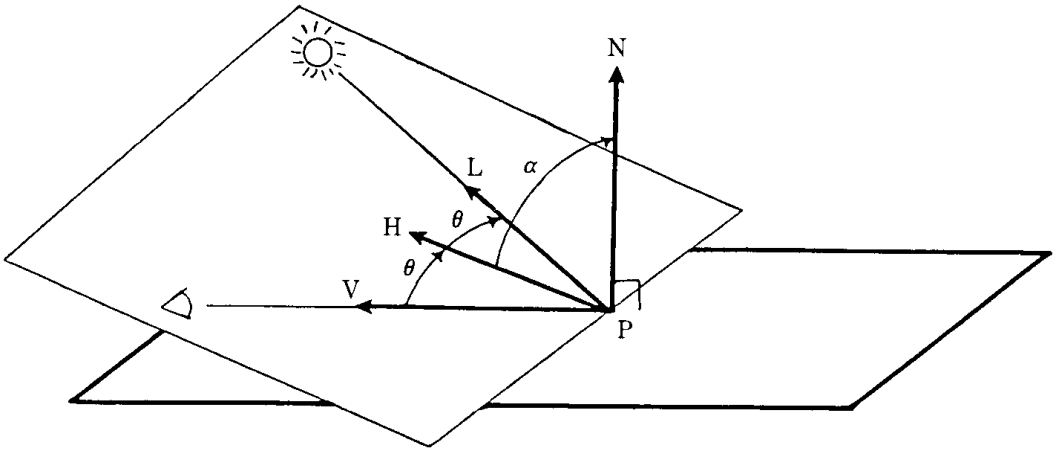


Abbildung 2.9: Zeigt die Anwendung des Cook-Torrance-Modells. H ist die Winkelhalbierende von V und L. (Quelle: [CT82])

2.5 Rasterung

Rasterung beschreibt die Konvertierung von Vektordaten (Beispielsweise Polygonen) in pixelbasierte Daten. Dieser Prozess wird besonders in Echtzeit-Anwendungen wie Videospielen seit vielen Jahren verwendet, da er sehr schnell und performant umgesetzt werden kann. Anders als bei Raytracing beschreibt der Prozess lediglich die Konvertierung der 3D-Szene in Pixel und hat somit keinen Einfluss darauf wie sich beispielsweise Farbwerte ergeben. Der Vorgang teilt sich in zwei Abschnitte auf, die Projektion der 3D-Szene in die Bildebene (2D) und das Herausfinden ob ein Pixel der Bildebene von einem Polygon verdeckt wird.

Die Projektion der 3D Szene wird realisiert mittels Perspektivischer Projektion[24]. Dazu wird mit Hilfe des Projektionsstrahls ermittelt ob ein Dreieck in der Szene von einem Pixel der Bildebene verdeckt wird. Dies ist der Fall sobald der Projektionsstrahl das Dreieck schneidet.

In der 2D Repräsentation werden anschließend die Pixel eingefärbt. Dies geschieht über sogenannte Pixelshader (Programme, die auf dem Grafikprozessor ausgeführt werden). Sobald der Mittelpunkt eines Pixels abgedeckt wird, gehört dieser Pixel zu diesem Dreieck und wird entsprechend eingefärbt (Siehe Abbildung. 2.10). Sollte ein Dreieck genau auf einem Mittelpunkt liegen, wird die "top-left rule" angewendet. Diese besagt, dass ein Pixel zu einem Dreieck gehört, wenn die obere Kante oder die linke Kante des Dreiecks auf der Pixelmitte liegt[25]. Mit der "top-left rule" werden benachbarte Dreiecke nur einmal gezeichnet. Ein einfacher Optimierungsschritt, welcher dafür sorgt, dass nicht die gesamte Bildebene jedes Mal durchlaufen werden muss, um ein Dreieck zu färben, ist die Verwendung von

[24] https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation , Stand: 09. April 2022

[25] https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-rasterizer-stage-rules , Stand: 09. April 2022

Bounding Boxes. Diese sorgen dafür, dass nur die Pixel innerhalb der Bounding Box durch iteriert werden[26].

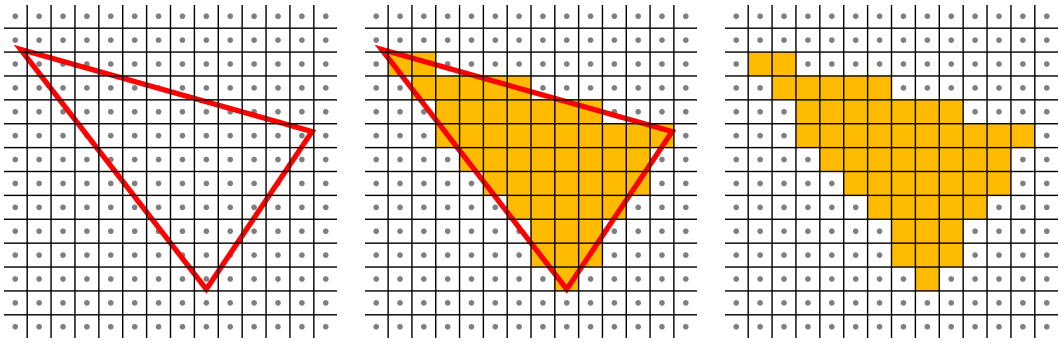


Abbildung 2.10: Zeigt, wie die Pixel der Bildebene mithilfe der Pixelmitte einem Polygon zugewiesen werden und entsprechend eingefärbt werden.

[26] https://mtrebi.github.io/2017/02/01/rasterization-i.html , Stand: 09. April 2022

Kapitel 3

Stand der Forschung und Technik

3.1 Unreal Engine in der Visualisierung

Unreal Engine wird bereits in verschiedenen Bereichen der Visualisierung erfolgreich eingesetzt. In diesem Abschnitt werden ein paar aktuelle Projekte aus unterschiedlichen Bereichen vorgestellt, um einen besseren Einblick zu geben, welche Rollen die Engine bereits in der Praxis erfüllt.

Der Supersportwagen Hersteller Pagani stellte in 2020 seinen neuen Autokonfigurator[1] (Abbildung 3.1a) vor, welcher es dem Kunden ermöglicht, sein Fahrzeug in einer fotorealistischen Echtzeit 3D Anwendung zu individualisieren. Möglich ist dies durch die Nutzung von Pixelstreaming, wodurch ein Videostream der Anwendung live zum Anwender übertragen wird. So benötigt die Anwenderseite keine leistungsstarke Hardware und muss lediglich das Video anzeigen.

Auch für Simulationen wird Unreal Engine bereits eingesetzt, hierbei sind besonders die realistischen Rendermöglichkeiten der Engine interessant. Im November 2021 präsentierte Meta Immersive Synthetics ihren VR Simulator[2] (Abbildung 3.1b), der es Kampfpiloten ermöglicht, durch eine realistische Landschaft zu fliegen und dabei verschiedenste Kampfszenarien und Notfälle zu üben. Der Ansatz, diese Simulationen in VR auszuführen, vereinfacht den Aufbau erheblich und senkt die Kosten verglichen zu bestehenden Simulatoren in diesem Gebiet.

In der Filmproduktion und Werbeindustrie ist man auch auf die Möglichkeiten der Engine aufmerksam geworden. Volkswagen setzte in 2021 ihren Enthüllungswerbespot für den Volkswagen ID.4 mit virtueller Produktion[3] (Abbildung 3.1c) um. Hierbei wurde Unreal En-

[1] https://www.unrealengine.com/en-US/spotlights/digital-showrooms-enrich-pagani-hypercar-configuration-experiences , Stand: 11. April 2022

[2] https://www.unrealengine.com/en-US/spotlights/affordable-aviation-training-in-a-compact-package-at-i-itsec-2021 , Stand: 11. April 2022

[3] https://www.unrealengine.com/en-US/spotlights/volkswagen-turns-to-virtual-production-for-greener-car-commercials , Stand: 11. April 2022

gine verwendet, um auf großen LED-Wänden realistische Umgebungen einzublenden, welche dann ein echtes Auto beleuchten. So ist es möglich in Echtzeit das Auto ohne logistischen Aufwand in die verschiedensten Szenarien zu stellen, mit realistischen Reflexionen und Beleuchtungseffekten auf dem echten Auto. Das Movie Render Queue System wurde hierfür ebenfalls verwendet, um weitere Szeneninformationen für die Postproduktion zu rendern.

(a) (b) (c)

Abbildung 3.1: Bilder aus den oben beschriebenen, mit Unreal Engine umgesetzte Projekten. (Quellen: (Bildausschnitte) 3.1a: https://www.unrealengine.com/en-US/spotlights/digital-showrooms-enrich-pagani-hypercar-configuration-experiences , Stand: 06. April 2022 / 3.1b: https://www.unrealengine.com/en-US/spotlights/affordable-aviation-training-in-a-compact-package-at-i-itsec-2021 , Stand: 06. April 2022 / 3.1c: https://www.unrealengine.com/en-US/spotlights/volkswagen-turns-to-virtual-production-for-greener-car-commercials , Stand: 06. April 2022)

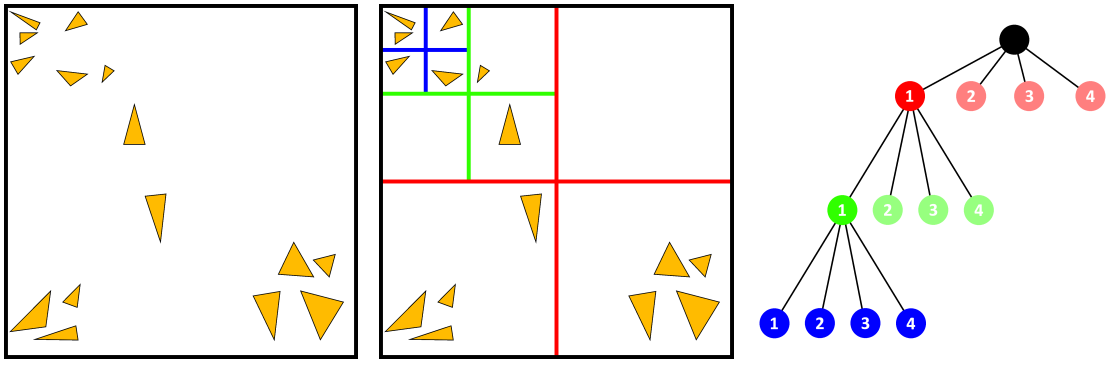
3.2 Beschleunigungsverfahren

Raytracing in seiner Grundform hat als zeitaufwendigsten Faktor die Schnittpunktberechnung der Strahlen mit der Geometrie. Dieses Problem macht 75% bis 95% der Rechenzeit bei der Nutzung von Raytracing aus[4], daher ist man hier auf der suche nach schnelleren Verfahren. Zusätzlich wird dieses Problem schlimmer, da mit steigender Auflösung in der Regel auch die Komplexität der 3D Szenen steigt. In aktuellen Implementierungen werden Beschleunigungssturkturen eingesetzt, welche eine schnellere Schnittpunktberechnung ermöglichen. Dies wird realisiert, indem die Anzahl der zu vergleichenden Dreiecke reduziert wird. Realisiert wird dies, indem die Geometrie räumlich unterteilt wird und in einer hierarchischen Struktur festgehalten wird. Dabei gibt es drei wichtige Faktoren: Die benötigte Zeit, um die Struktur zu initialisieren, der Speicherbedarf der Struktur und wie schnell die Struktur durchiteriert werden kann. In diesem Abschnitt wird auf die wichtigsten Verfahren eingegangen.

[4] http://www-lehre.inf.uos.de/~cg/1997/skript/20_2_4_Effizienz_beim.html , Stand: 11. April 2022

3.2.1 Quadtrees und Octrees

Eine einfache Beschleunigungsstruktur sind Quadtrees (2D) bzw. Octrees (3D). Zur Beschreibung des Prinzips werden hier Quadtrees betrachtet. Dieses Verfahren teilt die Dreiecke der Szene in Quadranten ein und an den notwendigen Stellen in immer kleiner werdende Quadranten, siehe Abbildung 3.2. Wird ein Strahl in die Szene geschossen, wird geprüft, welche Quadranten vom Strahl geschnitten wurden. Somit muss nur der Inhalt dieser Quadranten genauer überprüft werden und der Rest der Szene muss nicht mehr betrachtet werden. Für dieses Verfahren wird angegeben, wie viele Dreiecke sich maximal in einem Quadranten aufhalten dürfen und wie viele Aufteilungen maximal durchgeführt werden sollen. Anschließend wird die Szene rekursiv nach diesen Vorgaben durchgearbeitet und aufgeteilt. Der Algorithmus stoppt, sobald die Maximalanzahl der Aufteilungen erreicht ist oder wenn kein Quadrant mehr eine größere Anzahl an Dreiecken beinhaltet als das vorgegebene Limit[5]. Sollte ein Dreieck in mehreren Quadranten liegen, muss es entweder aufgeteilt werden oder in all diesen Quadranten referenziert werden. Sobald bei diesem Verfahren viele solcher überlappenden Dreiecke in der Szene vorkommen, steigt der Speicherbedarf stark an.



(a) (b) (c)

Abbildung 3.2: Zeigt schematisch den Ablauf eines Quadtrees. Bild 3.2a zeigt den initialen Zustand, Bild 3.2b zeigt, wie die Szene aufgeteilt wird und Bild 3.2c zeigt die dabei entstandene Baumstruktur.

3.2.2 KD Trees

Ein KD Tree (KD = K-dimensional) ist eine balancierte, binäre Baumstruktur[6], die ähnlich wie Octrees und Quadtrees funktioniert. Bei ihnen wird die Szene rekursiv mittels sogenannten Hyperplanes aufgeteilt. Zudem muss auch definiert werden, wie viele Polygone sich

[5] https://www.cg.tuwien.ac.at/sites/default/files/course/4411/attachments/05_spatial_acceleration_0.pdf , Stand: 06. Mai 2022

[6] https://www.uni-weimar.de/de/medien/professuren/medieninformatik/vr/research/real-time-rendering/beschleunigungsstrukturen-fuer-echtzeitfaehiges-ray-tracing/ , Stand: 14. April 2022

maximal auf einer Seite (Half-Space) befinden dürfen[7]. Dabei wird jede Aufteilung der Szene in einer Baumstruktur festgehalten. Blattknoten enthalten hierbei die Referenzen der Dreiecke und innere Knoten stellen die Hyperplanes dar. KD Trees sind Spezialfälle von BSP Trees (BSP = Binary Space Partition), der Unterschied besteht darin, dass KD Trees die Hyperplanes nur parallel zu Basisachsen anordnen, während BSP Trees jeden Winkel zulassen. Diese Einschränkung dient der Limitierung von möglichen Aufteilungen in der Szene. Die dabei entstandene Baumstruktur kann sehr schnell durchlaufen werden, jedoch kann die Initialisierung aufwendiger werden, je nachdem, wie aufwendig geprüft wird, an welcher Stelle die Aufteilung erfolgen soll. Der Speicherverbrauch ist wegen der Möglichkeit von überlappenden Dreiecken den Quad und Octrees ähnlich.

3.2.3 BVH

Die wichtigste Beschleunigungssturktur sind Bounding Volume Hierarchys (BVH). Ein BVH legt eine binäre Baumstruktur an, bei der in den Blättern die Dreiecke referenziert werden und in den Knoten die Bounding Volumes. Der größte Unterschied dieses Verfahrens zu den anderen ist, das sich Bounding Volumes überlagern können, dadurch kann der Speicherbedarf reduziert werden, des Weiteren ist es möglich, mehrere Bounding Volumes in einem Bounding Volume zusammenzufassen. Für den Aufbau muss wie bei den vorherigen Verfahren bekannt sein, wie viele Dreiecke sich maximal in einem Volume aufhalten dürfen. Das ideale Bounding Volume umfasst seinen Inhalt möglichst dicht, theoretisch können auch andere Formen wie beispielsweise Kugeln verwendet werden, jedoch kann hierdurch der Vorgang zu komplex werden, weswegen meistens einfache Quader verwendet werden. Weiterhin ist es sinnvoll, wie bei den KD Trees nicht jede Orientierung eines Bounding Volumes zuzulassen, da dies in den meisten Fällen zu langsam ist wegen der Vielzahl an Möglichkeiten[8]. Bounding Boxen, welche an einer Achse ausgerichtet sind, werden oft wie folgt abgekürzt: Axis aligned Bounding Boxes (AABB) oder Oriented Bounding Box (OBB).

Um zu entscheiden, wo eine Aufteilung stattfinden soll, gibt es mehrere Verfahren. Einfache Beispiele sind den räumlichen oder den Objektmedian zu verwenden, während aufwendigere Techniken wie "Surface Area Heuristic" (SAH) die Traversierungskosten einer potenziellen Aufteilung miteinbeziehen[9]. Die Auswahl eines Verfahrens ist stark davon abhängig, ob die Traversierungsgeschwindigkeit oder der Erstellungsaufwand des BVH für den Einsatzzweck wichtiger ist. Einen umfangreichen Vergleich dieser Thematik haben Tero Karras und Timo Aila in einer Veröffentlichung [KA13] erstellt. Insgesamt betrachtet ist der Speicherverbrauch eines BVH niedriger als bei den beiden vorherigen Verfahren, dadurch dass Bounding Volumes sich überlagern können und zusammengefasst werden können. Erstellungs- und Tra-

[7] https://www.cg.tuwien.ac.at/sites/default/files/course/4411/attachments/05_spatial_acceleration_0.pdf , Stand: 06. Mai 2022

[8] https://www.youtube.com/watch?v=TrqK-atFfWY , Stand: 14. April 2022

[9] https://www.uni-weimar.de/de/medien/professuren/medieninformatik/vr/research/real-time-rendering/beschleunigungsstrukturen-fuer-echtzeitfaehiges-ray-tracing/ , Stand: 14. April 2022

versierungsaufwand sind abhängig von der genauen Implementierung, lassen sich aber mit KD Trees vergleichen. BVHs werden in Unreal Engine und in Autodesk Arnold eingesetzt.

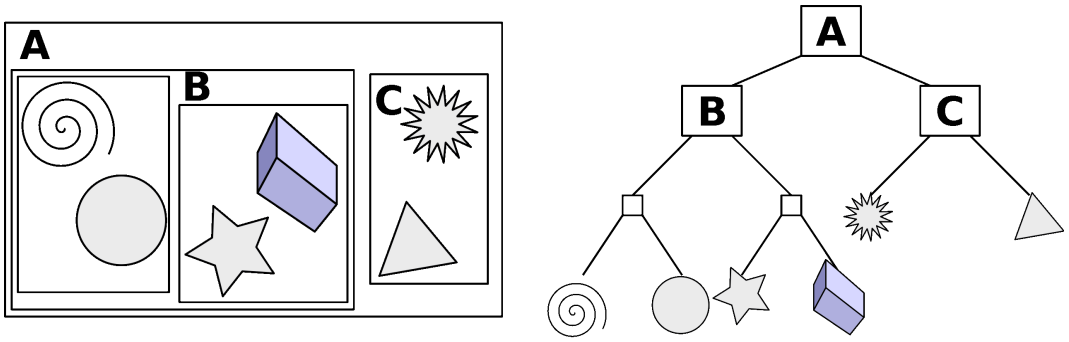


Abbildung 3.3: Zeigt beispielhaft, wie ein BVH aufgebaut ist mit der Verwendung von AABBs und die dadurch entstandene Baumstruktur. (Quelle: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy , Stand: 14. April 2022)

3.2.4 Russisch Roulette

Damit Path tracing ein unverfälschtes Ergebnis produzieren kann, müssten theoretisch alle Strahlen über unendlich viele Bounces verfolgt werden, dies ist jedoch nicht möglich. Russisch Roulette [AK90] ist eine Technik, die im Arnold Renderer [GIF+18] eingesetzt wird, um die Verfolgung von Strahlen einzuschränken, ohne dabei das Ergebnis zu verfälschen. Es funktioniert, indem der Strahl nach jedem Auftreffen mit einer vordefinierten Wahrscheinlichkeit entweder weiterverfolgt wird oder terminiert wird. Sobald ein Strahl weiterverfolgt wird und ein Objekt in der Szene trifft, wird sein ermittelter Wert mit dem Kehrwert der Wahrscheinlichkeit multipliziert, um zu kompensieren, dass andere Strahlen vorzeitig terminiert wurden. Mit diesem Prinzip könnte dieser Algorithmus theoretisch unendlich laufen, weshalb er nicht verfälschend ist[10]. In praktischen Implementierungen dieses Verfahrens wird die Wahrscheinlichkeit, dass ein Strahl weiterverfolgt wird, nach jedem Bounce weiter verringert. Eine zu niedrige Weiterverfolgungswahrscheinlichkeit führt zu Bildartefakten, welche Fireflys genannt werden. Diese treten auf, sobald ein weiterverfolgter Strahl ein Objekt trifft und wegen der geringen Wahrscheinlichkeit diesen nun sehr stark gewichtet, was zu einem hellen Pixel führt. Dieses Verfahren hat auch einen beschleunigenden Effekt, dadurch das Strahlen früher terminiert werden[11].

[10] https://www.cg.tuwien.ac.at/sites/default/files/course/4411/attachments/04_path_tracing_0.pdf , Stand: 06. Mai 2022

[11] https://clarissewiki.com/4.0/russian_roulette.html , Stand: 14. April 2022

3.3 Hardwarebeschleunigung

NVIDIA stellte mit ihrer "Turing" Architektur Grafikkarten vor mit dedizierten Kernen für Raytracing. Die Raytracing Kerne (RT Cores) beschleunigen BVH Traversierung und Ray casting. Raytracing benötigte auf normalen Grafikkarten Tausende Shaderinstruktionen pro Ray cast, um die immer kleiner werdenden Bounding Boxen im BVH zu überprüfen, siehe Abbildung 3.4. Zusätzlich entlasten RT Cores Streamprozessoren, welche nun an anderen Shader Berechnungen arbeiten können. Die Raytracing Kerne werden von APIs wie Microsoft DXR (DirectX Raytracing), NVIDIA OptiX und Vulkan angesprochen[12]. Mit der Nachfolgergeneration "Ampere" gibt NVIDIA eine bis zu Verdopplung der RT-Core-Performance an. Erreicht wird diese über ein optimiertes Cachingsystem und die Möglichkeit Compute Workloads und Grafik Workloads gleichzeitig auszuführen (Async Compute). Dies wird auch dafür genutzt, um Denoising und RT-Core Workloads auszuführen[13]. Mit Turing waren bis zu 72 RT-Cores der ersten Generation auf einer Grafikkarte möglich (TU102 Chip), während mit Ampere bis zu 84 RT-Cores der zweiten Generation realisiert wurden (GA102 Chip).

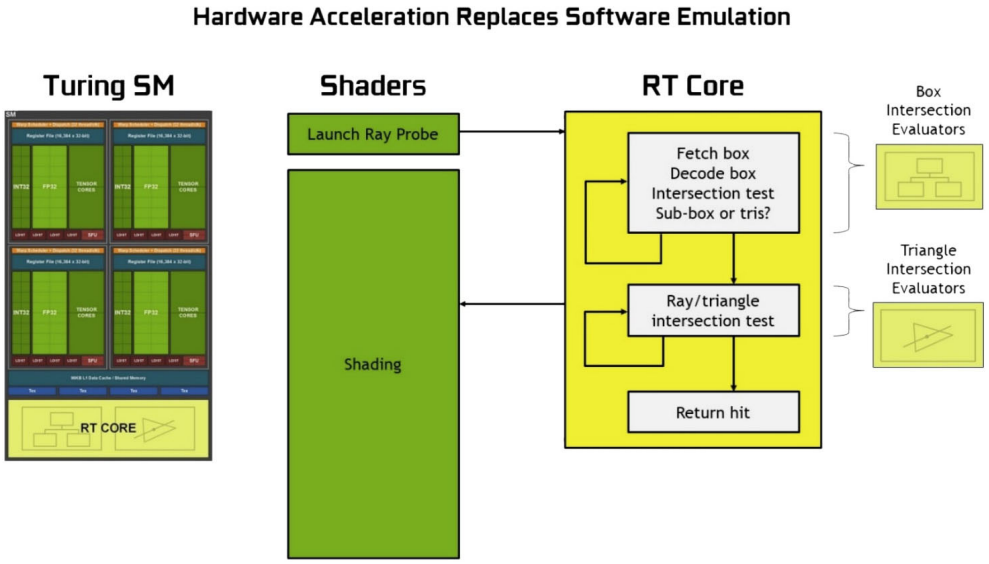


Abbildung 3.4: Zeigt wie Raytracing in der NVIDIA Turing Architektur umgesetzt ist. (Quelle: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf)

Auch AMD hat mit ihren neusten RDNA 2[14] Grafikkarten Hardwarebeschleunigung für Raytracing umgesetzt. Die sogenannten "Ray Accelerators" werden eingesetzt, um Schnitt-

[12] https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf , Stand: 06. Mai 2022

[13] https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf , Stand: 06. Mai 2022

[14] https://www.amd.com/de/technologies/rdna-2 , Stand: 15. April 2022

punkte mit dem BVH zu berechnen. Das BVH wird außerdem in dem "Infinity Cache" verarbeitet, welcher höhere Transferraten und geringere Latenzen ermöglicht. Anders als bei NVIDIA erfolgt die eigentliche Traversierung des BVH über Shader Code welcher auf den Compute Units ausgeführt wird[15]. Am 30. März 2022 kündigte Intel an, dass ihre Xe HPG Architektur, welche in den von Intel angekündigten dedizierten Grafikkarten eingesetzt wird, auch über DirectX 12 und Vulkan kompatible Raytracing Hardware verfügen soll[16].

3.4 Material Modell

Als diffuses BRDF nutzt Unreal das in 2.4 beschriebe Lambert Modell. In 2013 adaptierte die Unreal Engine eine angepasste Form des physikalisch basierten Cook-Torrance Modells, um die Erstellung von realistischen Materialien zu ermöglichen. Die Vereinfachungen dienen der Optimierung für Echtzeit Performance. Von der in 2.6 beschrieben Formel wurde D so übernommen, während G und F angepasst wurden. Für G und F wird das von Christophe Schlick angenäherte Modell [Sch94] mit leichten Anpassungen verwendet [KG13].

3.5 Lightbaking mit Lightmass

Wie bereits in 2.2.1 beschrieben, werden Lightmaps in Unreal Engine mit dem Lightmass Renderer generiert. Zur Berechnung der Lightmaps verwendet Lightmass Photon Mapping[17,18]. Die benötigte Renderzeit einer Lightmap kann je nach Qualitätseinstellungen, Auflösung und verwendeter Hardware sehr unterschiedlich sein, für finale Ergebnisse sind mehrere Stunden nicht unüblich. In den Worldsettings des Levels finden sich die Lightmass Einstellungen, mit denen die Qualität und die Anzahl der Bounces angepasst werden kann. In den Default Einstellungen werden Lightmaps komprimiert, um Grafikspeicher zu sparen. Für Visualisierungen empfiehlt es sich jedoch, diese Kompression auszuschalten, um das bestmögliche visuelle Ergebnis zu erreichen. Lightmass nutzt die CPU um Lightmaps zu berechnen, jedoch ist seit der Engine Version 4.26 eine Beta Implementierung[19] von Lightmass verfügbar, welche die DirectX12 Raytracing (DXR) API nutzt, um Lightmaps schneller auf der GPU zu berechnen. GPU Lightmass supportet seit der Version 4.27 auch die Verwendung von mehreren Grafikkarten[20].

[15] https://www.golem.de/news/radeon-rx-6800-xt-im-test-die-rueckkehr-der-radeon-ritter-2011-152085-3.html , Stand: 15. April 2022

[16] https://www.intel.de/content/www/de/de/architecture-and-technology/visual-technology/arc-discrete-graphics/xe-hpg-microarchitecture.html , Stand: 18. April 2022

[17] https://docs.unrealengine.com/5.0/en-US/lightmass-portals-in-unreal-engine/ , Stand: 15. April 2022

[18] https://docs.unrealengine.com/5.0/en-US/cpu-lightmass-global-illumination-in-unreal-engine/ , Stand: 15. April 2022

[19] https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/GPULightmass/ , Stand: 15. April 2022

[20] https://www.pugetsystems.com/labs/articles/Unreal-4-27-adds-multi-GPU-support-for-lightmass-building-2216/ , Stand: 15. April 2022

3.6 Reflexionen in Unreal Engine ohne Raytracing

Um Reflexionen[21] ohne Raytracing zu ermöglichen, werden in Unreal Engine Screen Space Reflections (SSR), Reflection Captures und planar Reflections eingesetzt. Diese Verfahren bieten ihre eigenen Vor- und Nachteile und werden in der Praxis oftmals zusammen eingesetzt, um bessere Ergebnisse zu erzielen. Obwohl mittlerweile Raytracing für Reflexionen genutzt werden kann, werden diese Verfahren trotzdem gerne verwendet, um beispielsweise eine bessere Performance zu ermöglichen.

Screen Space Reflections sind das einzige Reflexionsverfahren, welches in den Default Einstellungen der Engine eingeschaltet ist[22]. Das Verfahren nutzt die Informationen aus dem Sichtkegel des Betrachters, um Reflexionen zu realisieren. Dies hat den Nachteil, dass es nicht möglich ist, Objekte in Reflexionen zu sehen, welche sich nicht im Sichtkegel des Betrachters befinden. SSR arbeitet in Echtzeit und kann somit auch dynamische Reflexionen abbilden. Mit der Ausnahme von Raytracing ist es das genaueste Reflexionsverfahren der Engine. Weiterhin sind SSR oftmals verrauscht und erzeugen Bildartefakte.

Reflection Captures sind vorberechnete Cubemaps (360° Bild), welche auf die Objekte projiziert werden, die sich in ihrem einstellbaren Bereich befinden[23]. Sie sind nur ein lokaler Effekt und müssen manuell in der Szene platziert werden, weswegen sie ungenau sind. Ihre Vorteile liegen darin, dass sie hochauflösende, scharfe Reflexionen ermöglichen und für Echtzeit Anwendungen keine extra Renderkosten verursachen, da sie vorberechnet sind. Sobald sich Bereiche von Reflection Captures überlagern, werden diese mit einem weichen Übergang zusammengeblendet.

Planar Reflections sind das am wenigsten genutzte Reflexionssystem der Engine. Sie bieten sehr genaue dynamische Reflexionen, sind jedoch aufwendig zu rendern da die Szene für diese Art von Reflexion nochmal gerendert wird[24]. Weiterhin eignen sie sich nur für planare Flächen. Weil die Verfahren in der Praxis zusammen eingesetzt werden, gewichtet die Engine SSR am höchsten, da es das Genaueste der Verfahren ist. Sobald SSR nicht verfügbar ist, fällt die Engine zurück auf Planar Reflections und anschließend auf Reflection Captures.

[21] https://dev.epicgames.com/community/learning/courses/EGR/an-in-depth-look-at-real-time-rendering/rEl/an-in-depth-look-at-real-time-rendering-reflections , Stand: 14. April 2022

[22] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/PostProcessEffects/ScreenSpaceReflection/ , Stand: 14. April 2022

[23] https://docs.unrealengine.com/5.0/en-US/reflections-captures-in-unreal-engine/ , Stand: 14. April 2022

[24] https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/PlanarReflections/ , Stand: 14. April 2022

3.7 Deferred Rendering

Unreal Engine nutzt in den Defaulteinstellungen Deferred Rendering (auch Deferred Shading genannt). Deferred Rendering trennt die Geometrieverarbeitung von der Beleuchtungsberechnung, siehe Abbildung 3.6. Dadurch kann eine große Anzahl von Lichtquellen realisiert werden, da Licht nur für die betroffenen Pixel berechnet wird[25]. Beachtlich ist das Laufzeitverhalten von $O(n)$ bei Deferred Rendering verglichen zu Forward Rendering mit $O(n^2)$[26]. Die Schwächen von deferred Rendering liegen bei transparenten Objekten. Um zu verhindern, dass verdeckte Stellen mehrmals gerendert werden, wird in Unreal zuerst eine Tiefenmap (Z-Map) erstellt. Anschließend rendert die Engine Drawcall für Drawcall das Bild. ein Drawcall ist eine Gruppe von Polygonen welche die selben Eigenschaften besitzen, beispielsweise das selbe Objekt oder das selbe Material. Die Anzahl der Drawcalls hat dabei eine große Auswirkung auf die Performance[27].

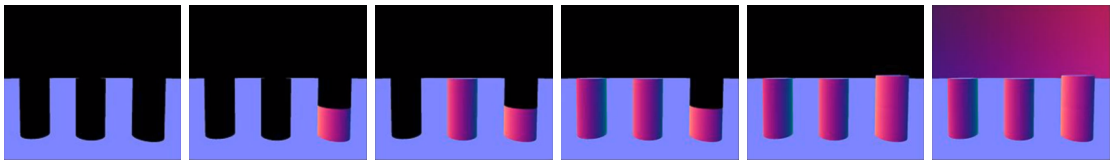


Abbildung 3.5: Zeigt von links nach rechts wie ein Bild (World Normals) Drawcall für Drawcall gerendert wird. Der rechte Zylinder nutzt zwei Materialien weswegen er zwei Drawcalls verursacht. (Quelle: https://dev.epicgames.com/community/learning/courses/EGR/an-in-depth-look-at-real-time-rendering/JEJ/geometry-rendering-part-1 , Stand: 06. April 2022)

Deferred Rendering teilt sich in zwei Abschnitte ein, die Geometriephase und die Beleuchtungsphase. In der Geometriephase werden Daten wie beispielsweise World Normals, Base Color, Specular usw. ermittelt und in Texturen gespeichert. Gesammelt werden diese Texturen G-Buffer (Geometry-Buffer) genannt. Der G-Buffer kann direkt in der Engine betrachtet werden über den "Buffer Overview viewmode", oder im Detail analysiert werden über das opensource Programm Renderdoc[28]. Anschließend werden in einem bildfüllenden Rechteck [Wie12] die Beleuchtungsberechungen anhand der Informationen aus dem G-Buffer Pixel für Pixel ausgeführt[29].

[25] https://learnopengl.com/Advanced-Lighting/Deferred-Shading , Stand: 15. April 2022
[26] https://cs.uni-paderborn.de/cgvb/forschung/deferred-rendering , Stand: 15. April 2022
[27] https://dev.epicgames.com/community/learning/courses/EGR/an-in-depth-look-at-real-time-rendering/JEJ/geometry-rendering-part-1 , Stand: 15. April 2022
[28] https://renderdoc.org , Stand: 15. April 2022
[29] https://learnopengl.com/Advanced-Lighting/Deferred-Shading , Stand: 15. April 2022

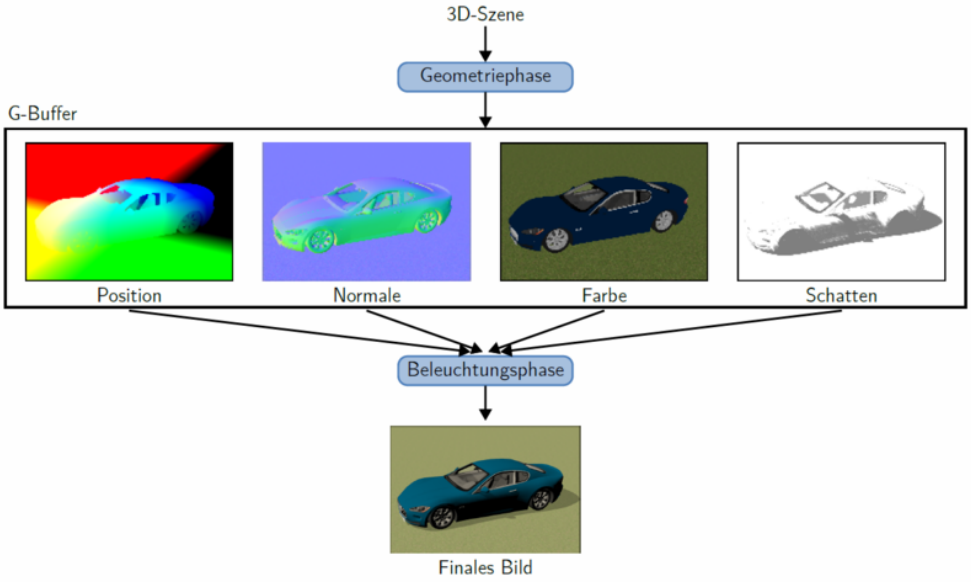


Abbildung 3.6: Zeigt den Ablauf von Deferred rendering. (Quelle: https://cs.uni-paderborn.de/cgvb/forschung/deferred-rendering , Stand: 13. April 2022)

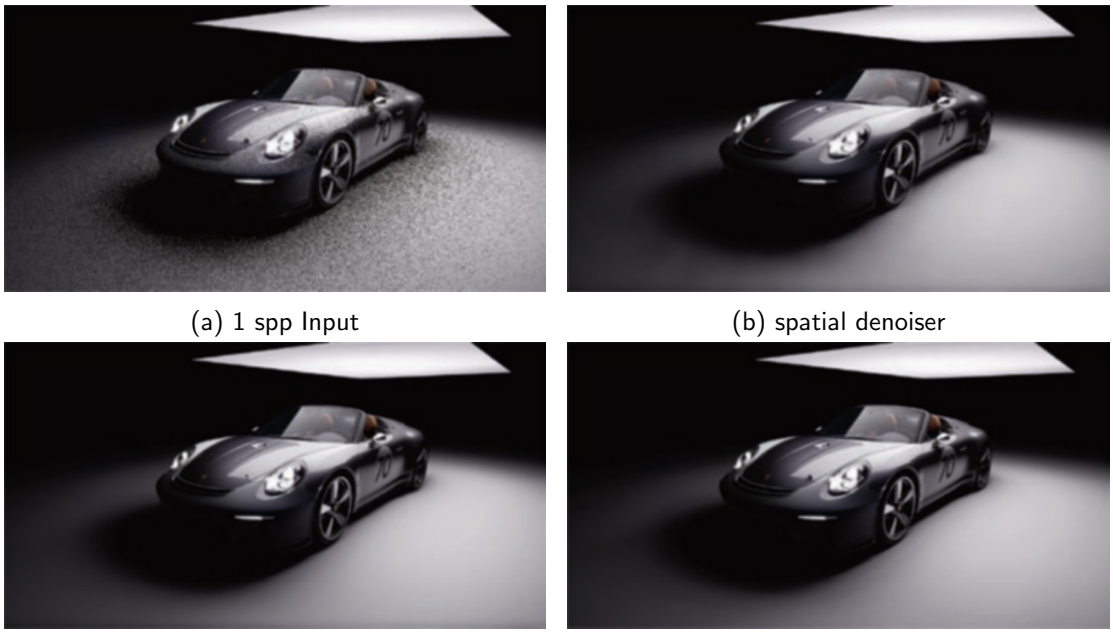
3.8 Real time Raytracing in Unreal Engine

Mit der Version 4.22 wurde Raytracing in der Unreal Engine implementiert. Die Implementierung wurde von Epic Games in Zusammenarbeit mit NVIDIA umgesetzt und baut auf DirectX12 auf. In einer GDC Präsentation[30] wurde erklärt, dass es ein Ziel der Umsetzung war, dass Rasterverfahren und Raytracing zusammen funktionieren und dass dieselben Shadermodelle verwendet werden können. Dies soll verhindern, dass Nutzer Materialien speziell für ein bestimmtes Verfahren erstellen müssen. Damit Raytracing in Echtzeit überhaupt möglich ist, wird Hardwarebeschleunigung in Kombination mit einem sehr geringen Sample Budget eingesetzt. Zusätzlich werden Rekonstruktionsfilter angewendet, die das wegen mangelnden Samples sehr verrauschte Ergebnis zu einem brauchbaren Ergebnis rekonstruieren. Dabei werden getrennte Filter verwendet für die jeweiligen Probleme wie beispielsweise globale Beleuchtung, weiche Schatten oder spiegelnde Reflexionen [LLK+19]. Weiterhin stellt die DirectX Raytracing API eine zwei Ebenen BVH Beschleunigungstuktur bereit, welche sich zusammensetzt aus Top-level acceleration structures (TLAS) und bottom-level acceleration structures (BLAS). BLAS beinhalten die Geometrie in form von AABBs. Für statische Geometrie wird das BLAS nur einmal erstellt, das TLAS wiederum wird in dynamischen Szenen für jeden Frame neu generiert. In diesem Abschnitt wird auf die Implementierung von Raytracing in Unreal Engine eingegangen, dazu werden die folgenden Bereiche betrachtet: Schatten, Reflexionen und globale Beleuchtung.

[30] https://developer.nvidia.com/siggraph/2019/video/sig935 , Stand: 13. April 2022

3.8.1 Umsetzung der Schatten

In der "Speed of Light" Demo von Epic Games, NVIDIA und Porsche wurden Area light Schatten mit nur einem Sample pro Pixel gerendert und anschließend über einen speziell entwickelten denoising Filter entrauscht und rekonstruiert, siehe Abbildung 3.7. Der Filter hat eine räumliche und eine zeitliche Komponente. Die räumliche Komponente baut auf bestehenden Verfahren auf und nutzt zusätzlich Informationen wie Größe und Ausrichtung der Lichtquelle, weswegen er pro Lichtquelle angewendet wird. Die räumliche Komponente versucht anhand dieser Information den für die Situation optimalen Filterkernel zu erstellen. Die zeitliche Komponente kann angewendet werden, um den effektiven Samplecount auf etwa 8 bis 16 zu erhöhen, was jedoch zu einer leichten zeitlichen Verzögerung führt [LLK+19]. Der Samplecount und die Verwendung der Rekonstruktionsfilter (Denoising) kann in der Engine über Konsolenkommandos oder im Post Process Volume eingestellt werden. Diese Optionen sind für alle der Raytracing Features verfügbar.



(a) 1 spp Input

(b) spatial denoiser

(c) spatiotemporal denoiser

(d) Referenz mit 2048 spp

Abbildung 3.7: 3.7a Zeigt das Inputbild, 3.7b zeigt das mit der räumlichen Komponente entrauschte Bild, 3.7c zeigt die zusätzliche Anwendung der zeitlichen Komponente auf das Ergebnis der räumlichen Komponente und 3.7d zeigt das Referenzbild. (Quelle: [LLK+19])

3.8.2 Umsetzung der Reflexionen

Raytracing bietet für Reflexionen große Vorteile gegenüber den bereits in Unreal vorhandenen Verfahren (Abschnitt 3.6). Beispielsweise können dynamisch Objekte abgebildet werden, welche sich nicht im Sichtfeld befinden, oder Streckungen dargestellt werden, welche durch die Normals der Oberfläche verursacht werden. Raytracing ist außerdem ein robusteres Verfahren als SSR für die Anwendung in Dynamischen Szenen [LLK+19]. Echtzeit performance wird erreicht durch mehrere Optimierungen. Eine dieser Optimierungen ist der Raytracing Quality Switch, der über den Material Editor benutzt werden kann. Sobald Raytracing verwendet wird kann über diesen Switch ein vom Nutzer erstellter, vereinfachter Material-Netzwerkzweig (Beispielsweise einsparen der Normalmap) für Reflexionen verwendet werden[31]. Ein weiterer Optimierungsschritt ist die Nutzung eines Verfahrens, welches Strahlen adaptiv früher terminiert. Es bestimmt dies anhand der Roughness des getroffenen Objekts. Im Durchschnitt wird hierdurch die BRDF nur einmal pro Pixel gesamplet, was zu einem sehr verrauschten Ergebnis führt, welches im nächsten Schritt entrauscht wird [LLK+19]. Der dafür entwickelte Denoising Filter hat ähnlich dem Filter für Schatten eine räumliche und eine zeitliche Komponente. Für die Entrauschung der Reflexionen wird ein richtungsunabhäniger BRDF-basierter Filterkernel eingesetzt. Der Filter wird nur auf den reflektierten eingehenden Strahlungsterm angewandt.

3.8.3 Umsetzung von Globaler Beleuchtung

Für die Umsetzung der Globalen Beleuchtung wird Pathtracing mit Next Event Estimation (NEE) eingesetzt [LLK+19]. Zusätzlich wird das in 3.2.4 beschriebene Russisch Roulette verwendet, um Strahlen früher zu terminieren. Dieser Brute Force Ansatz ist jedoch sehr leistungsintensiv, weswegen sich hiermit nur interaktive bzw cinematic Bildwiederholraten erreichen lassen. Desweiteren benötigt man für brauchbare Ergebnise zwischen 16 und 64 Samples, was problematisch ist für die Anwendung in Echtzeit [MSW21]. Auch hier werden wieder Rekonstruktionsfilter eingesetzt um die verrauschten Bilder zu entrauschen. Mit der Engine Version 4.24. wurde die Option hinzugefügt die Final Gather Methode zu verwenden. Diese versucht teure Materialevaluierungs rays durch günstigere Visibilitäts rays zu ersetzen. Erreicht wird dies indem mit nur einem Sample pro Pixel gearbeitet wird und zusätzlich Informationen aus vorherigen Frames verwendet werden. Die Final Gather Methode ist künstlich limitiert auf einen bounce [MSW21].

[31] https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/ , Stand: 13. April 2022

3.9 Hybrid-Raytracing

In Unreal Engine können, wie in 3.8 beschrieben, Raytracing und Rasterverfahren zusammen verwendet werden. Dies ermöglicht es Raytracing nur für die wichtigsten Stellen der Szene zu verwenden, um Performance einzusparen bei nur sehr geringem visuellem Qualitätsunterschied. Realisiert wird dies über das Post Process Volume und die in 3.8.2 beschriebenen Materialswitches. Über das Post Process Volume kann Raytracing gezielt eingestellt werden, um beispielsweise nur für globale Beleuchtung oder nur für Reflexionen genutzt zu werden[32]. Zusätzlich kann dort eingestellt werden, dass ab einem vorgegebenen Roughness Wert die Engine auf Rasterverfahren zurückfallen soll. Ein Post Process Volume kann diese Einstellungen global auf die gesamte Szene anwenden oder lokal innerhalb des Post Process Volumes. Es ist außerdem möglich, mehrere dieser Post Process Volumes in der Szene für lokale Effekte einzusetzen. Für Reflexionen ist es über ein Konsolenkommando[33] möglich zu definieren, dass die Engine für den letzten Bounce auf Reflection Captures zurückfällt.

Ein Beispiel einer solchen hybriden Szene kann in Abbildung 3.8, welche aus der GDC Präsentation[34] von Sjoerd De Jong aus 2019 stammt, gesehen werden. Abbildung 3.8 zeigt einen von ihm erstellen Debugview einer Szene, in dem zu sehen ist, wie aufwendig Reflexionen an den jeweiligen Stellen zu rendern sind. Dabei erscheinen sehr spiegelnde Oberflächen als gelb, da diese Raytracing verwenden und einfach zu berechnen sind und sehr raue Oberflächen als grün, da diese auf Rastermethoden zurückfallen und deswegen sehr schnell zu rendern sind. Rot zeigt Stellen, die sehr aufwendig zu rendern sind, da diese Raytracing verwenden auf Oberflächen, die etwas rau sind und somit teurer zu berechnen sind.

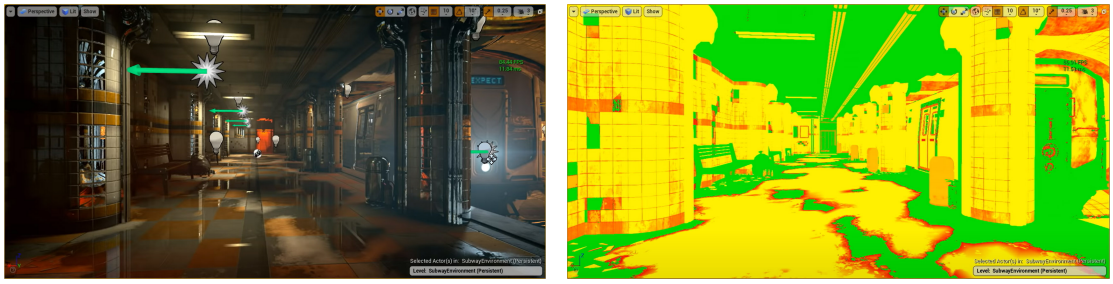


Abbildung 3.8: Zeigt links das Bild der Szene und links einen Debugview welcher zeigt wie "teuer" die Reflektionen an den jeweiligen Stellen zu rendern sind. (Quelle: Screenshots aus dem Youtube Video: https://www.youtube.com/watch?v=EekCn4wed1E , Stand: 14. April 2022)

[32] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/RayTracing/RayTracingSettings/ , Stand: 14. April 2022

[33] https://jaredp94.github.io/Unreal-Development-Guides-and-Tips/Content/RTRT/Reflections.html , Stand: 14. April 2022

[34] https://www.youtube.com/watch?v=EekCn4wed1E , Stand: 14. April 2022

3.10 Beschreibung des Arnold Renderers

Arnold ist ein unverfälschter, physikalisch basierter CPU Renderer dessen Entwicklung bereits im Jahr 1997 durch Marcos Fajardo begann[35]. In 2009 gründete er die Firma Solid Angle, welche weiter an der Entwicklung von Arnold arbeitete. In 2016 wurde die Firma von Autodesk übernommen[36] und seit der Maya Version 2017 ersetzt Arnold den Mental Ray Renderer von NVIDIA als Standardrenderer im Programm.

Arnold setzt auf brute-force Pathtracing [Kaj86], der Grund dafür ist, dass Pathtracing viele Effekte und Probleme sehr elegant lösen kann, jedoch auf Kosten von Renderzeit. Ein weiterer Vorteil von Pathtracing ist, dass es für den Anwender einfach zu bedienen ist. Bei Arnold wird viel Wert darauf gelegt, dass die Bedienung so einfach wie möglich bleibt und dass es ein robustes Gesamtsystem ist. Es wird unidirektionales Pathtracing (Pathtracing aus einer Richtung) verwendet, da sich in der Praxis gezeigt hat, dass bei bidirektionalem Pathtracing sich Renderzeiten verlangsamen und Berechnungen verkomplizieren [GIF+18]. In Arnold ist jede Komponente der Szene ein Knoten (Lichter, Kameras, Geometrie usw.), dies wird sich zu Nutze gemacht bei der Anwendung der Beschleunigungsstruktur. Als Beschleunigungsstruktur wird ein Multilevel BVH verwendet, dafür wird zuerst ein top Level BVH welches alle Knoten beinhaltet, erstellt. Im späteren Renderprozess werden, sobald ein Strahl einen Knoten schneidet, weitere BVHs für den Knoten erstellt. Durch diese verzögerte Verarbeitung wird Rechenzeit und Speicher gespart, da Knoten, welche nicht getroffen werden, gar nicht erst verarbeitet werden. Ein Nachteil dieser Methode ist, dass sich Optimierungen der BVH Hierarchie schwieriger gestalten[GIF+18].Um die BVH Aufteilungen zu realisieren, wird eine binned surface area heuristic (SAH) [PGSS06] benutzt. Arnold verwendet auch das in 3.2.4 beschriebene Russisch Roulette, wodurch Strahlen als günstiger angesehen werden. Noch ein Ansatz zur Beschleunigung ist es, grundsätzlich Rauschen zu minimieren. Dies wird erreicht durch optimierte Sampling Patterns, dazu werden correlated multi-jittered (CMJ) patterns verwendet. Eine weitere verwendete Technik ist multiple importance sampling (MIS) [VG95].

Seit der Version 6 wird die GPU Implementierung des Arnold Renderers als "production ready" gewertet[37], somit kann nun auch die Grafikkarte zum Rendern verwendet werden. Diese Implementierung basiert auf dem NVIDIA-OptiX-Framework weswegen Arnold GPU auch nur mit NVIDIA-Grafikkarten kompatibel ist[38]. Für das Rendern von hochauflösenden Bildern ist jedoch CPU Rendering weiterhin interessant, da für hochauflösende Renderings oftmals komplexere Szenen verwendet werden, welche den der Auflösung entsprechenden Detailgrad ermöglichen. Dadurch kann es schnell dazu kommen, dass der verfügbare Grafikspeicher nicht mehr ausreicht. Zusätzlich ist zu erwähnen, dass bei CPU Rendering sehr viel mehr Arbeitsspeicher möglich ist, zu einem deutlich geringeren Preis.

[35] https://www.arnoldrenderer.com/about/ , Stand: 18. April 2022
[36] https://www.arnoldrenderer.com/news/solid-angle-joins-autodesk/ , Stand: 18. April 2022
[37] https://www.arnoldrenderer.com/news/arnold-6/ , Stand: 14. April 2022
[38] https://docs.arnoldrenderer.com/display/A5ARP/Supported+GPUs , Stand: 14. April 2022

3.11 Überlegungen über mögliche Folgen

Grundsätzlich ist zu erwähnen, dass bei Unreal Features immer mit dem Ziel der Echtzeit Performance entwickelt wird, während bei Arnold wie in 3.10 beschrieben, Genauigkeit und Bedienbarkeit eine sehr wichtige Rolle spielen. Aus den bis hier beschriebenen Techniken zeigt sich bereits, dass in Unreal sehr viele Optimierungen möglich sind, die das Rendern beschleunigen. Außerdem wurden an manchen Stellen Vereinfachungen angewendet, um weiter Zeit einzusparen. Eine der wichtigsten Techniken wie Unreal überhaupt Echtzeit Performance erreichen kann, bei visuell rauschfreier Qualität ist der Einsatz der in 3.8 beschriebenen Denoising Verfahren. Zusätzlich macht Unreal Gebrauch von der in 3.3 erwähnten Raytracing Hardware, um die BVH Traversierung zu beschleunigen. Dies sollte bei weiteren Betrachtungen zu einer erheblichen Beschleunigung gegenüber Arnolds CPU Version führen.

Weiterhin hat Unreal durch die Movie Render Queue die Möglichkeit, mit mehreren spatial oder temporal Samples zu arbeiten. Hierdurch wird derselbe Antialiasingeffekt wie bei Pathtracing erzielt, da mehrere variierende Samples pro Pixel eingesetzt werden. Deswegen ist zu erwarten das, dass Antialiasing in den Unreal Renderings qualitativ den Arnold Renderings ähnlich ist. Unter Berücksichtigung der hier beschriebenen Erkenntnisse ist somit zu erwarten, dass Unreal zu einem visuell ähnlichen Ergebnis bei weniger Renderzeit kommen sollte.

Durch die Verwendung von Lightmaps bieten sich enorme Vorteile, sobald mehr als ein Rendering in derselben Szene erstellt wird, da der aufwendige Baking Prozess nur einmal durchgeführt werden muss. Dabei ist der Berechnungsaufwand für den Baking Prozess stark abhängig von der gewählten Lightmap Auflösung der jeweiligen Assets. Zusätzlich sind die in 3.6 beschriebenen Reflexionsverfahren, welche oft in Kombination mit Lightbaking verwendet werden, wesentlich performanter, aber auch limitierter als Raytracing oder Pathtracing. Somit ist zu erwarten, dass solche Reflexionen wesentlich schneller zu rendern sind, jedoch visuell stärker abweichen können von Unreals Raytracing und Arnold.

In einem Vergleich, der von Julius Hilbig durchgeführt wurde [Hil20] ergab sich bereits, dass es möglich ist, visuell sehr ähnliche Ergebnisse zu Vray mit Unreal zu erreichen. Bei diesem Vergleich wurde jedoch Unreal als Echtzeit-Renderer verwendet. In einem anderen Vergleich [BJLY20] wurde Unreal mit mehreren offline Renderern verglichen, darunter auch Autodesk Arnold. Dort wurde verglichen, wie nah das Unreal Ergebnis an die Ergebnisse der anderen Renderer herankommt. Dabei wurde im Vergleich zwischen Unreal und Arnold ein SSIM Wert (Siehe 4.4) von 84.94% ermittelt. Jedoch wurde auch dort Unreal als Echtzeit-Renderer verwendet, während Arnold mehrere Minuten benötigte für ein Bild.

Kapitel 4

Vergleichskonzept

In diesem Kapitel wird zuerst mit ausgewählten einfachen Szenen die Leistungsfähigkeit der Verfahren Lightbaking, Raytracing und Pathtracing miteinander verglichen. Anschließend wird beschrieben, welche Anforderungen bei einem realistischen Anwendungsfall zu beachten sind. Dabei geht es hauptsächlich um Anforderungen an das 3D-Modell im Bezug auf die Verwendung des Modells in der Unreal Engine. Weiterhin wird auf die Anforderungen und Problematiken eingegangen, welche beim Rendern von Panoramen entstehen. Zuletzt beschreibt dieses Kapitel, welche Metriken angewendet werden, um Bilder objektiv zu bewerten.

4.1 Vergleich von Pathtracing (Arnold), Realtime Raytracing und Lightbaking (beides Unreal)

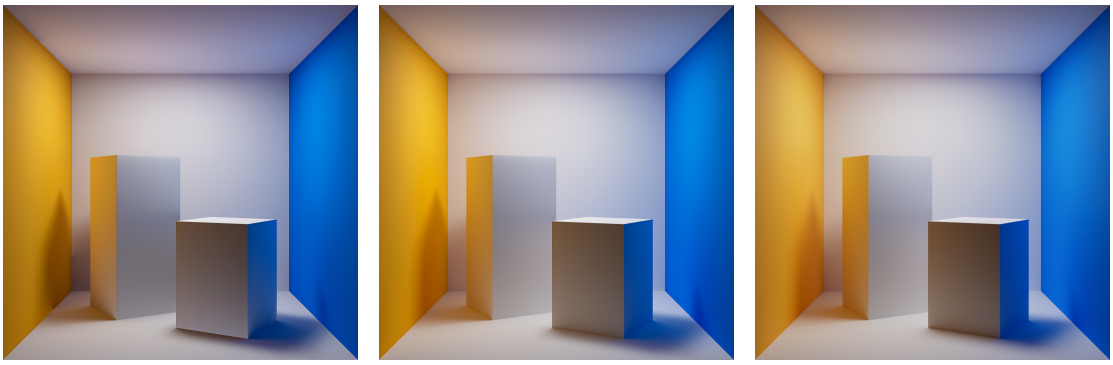
Um die drei Verfahren besser zu veranschaulichen und ihre Unterschiede bzw. Limitierungen aufzudecken, werden in diesem Abschnitt mittels einfacher Tests, die Verfahren direkt miteinander verglichen. Dabei werden gesondert die Kriterien indirekte Beleuchtung, Reflexionen, Schatten und Transluzenz untersucht. Weiterhin werden alle Renderings im EXR-Format exportiert und in Photoshop mittels von Autodesk[1] beschriebenen Lookup Tables farblich angepasst. Damit diese Anpassung auch mit den Bildern aus Unreal durchgeführt werden kann, muss die Tonecurve deaktiviert werden. Dies ist möglich über die Movie Render Queue, welche für das Rendern der Unreal Bilder ebenfalls eingesetzt wird. Es soll auch betrachtet werden, wie sehr sich das Ergebnis visuell zu Arnold unterscheidet. Für die Vergleiche werden weder in Arnold noch die in 3.8 beschriebenen Unreal Denoiser verwendet.

Spezifikationen des Testsystems, welches für die Durchführung verwendet wurde:
CPU: Intel Core i7 12700K, GPU: RTX 2060 Super 8GB, RAM: 48GB 4000MHz.
Unreal Engine Version 4.27.2, Arnold Version MtoA 5.0.0.1 (Core 7.0.0.0) in Maya 2022.3.

[1] https://knowledge.autodesk.com/support/maya/troubleshooting/caas/sfdcarticles/sfdcarticles/Colors-from-Maya-2022-Arnold-render-view-do-not-match-with-EXR-rendering-result-in-Photoshop.html , Stand: 28. März 2022

4.1.1 Indirekte Beleuchtung

Dieser Test zeigt, wie sich die Indirekte Beleuchtung, die durch die diffuse Reflexion an den Objektoberflächen entsteht, bei den drei Verfahren verhält. Hierzu wurde ein Nachbau der bekannten Cornell Box[2] Szene erstellt. Der Aufbau der Szene ermöglicht es, die indirekte Beleuchtung, welche durch das Verfolgen von mehreren Light-Bounces entsteht, zu veranschaulichen. Für den Test wurde nach Ermitteln von geeigneten Einstellungen eine Strahlentiefe von 12 verwendet. Zusätzlich wurden die Rendereinstellungen so getroffen, dass ein visuell rauschfreies Bild entsteht. Als Lichtquelle wird bei Arnold ein Area Light verwendet und in Unreal ein Rect Light, da dies dem Arnold Arealight am nächsten kommt. Die Lichtintensität der Lichtquelle in Unreal wurde visuell angeglichen an die Arnold Lichtquelle, zusätzlich haben beide Lichtquellen dieselbe Größe.



(a) Lightmaps
00:00:01
00:10:53

(b) Raytracing
00:13:38

(c) Pathtracing
01:29:01

Abbildung 4.1: Zeigt die fertigen Renderings des Tests mit den jeweiligen Renderzeiten. Für Bild 4.1a ist zusätzlich noch in einer extra Zeile die Zeit angegeben die das Light Baking benötigt hat.

Die Auswertung zeigt, dass sich die Ergebnisse auf den ersten Blick sehr ähnlich sind, sich jedoch im Detail unterscheiden. Light Baking (Abbildung 4.1a) weicht hierbei am meisten von Arnold (Abbildung 4.1c) ab, dies ist besonders zu sehen in der Färbung des Schattens auf der gelben Wand und im unteren Bereich des höheren Quaders. Unreals Raytracing (Abbildung 4.1b) liefert ein Ergebnis, welches visuell kaum von Arnold zu unterscheiden ist. Bei beiden Verfahren ist zusätzlich die benötigte Renderzeit zu beachten welche deutlich niedriger ausfällt als bei Arnold. Weiterhin ist hier zu berücksichtigen, dass Light Baking nur einmal den zeitaufwendigen Baking Prozess zum Berechnen der Lightmaps durchlaufen muss und anschließend beliebig viele Bilder in wenigen Sekunden generieren kann. Hierbei ist noch zu erwähnen, wenn dieses Verfahren (Light Baking) für hochauflösende Renderings verwendet wird, muss die Auflösung der Lightmap entsprechend an die Renderauflösung angepasst sein.

[2] http://www.graphics.cornell.edu/online/box/ , Stand: 28. März 2022

4.1.2 Reflexionen

Zum Vergleichen der Reflexionen wurde die Szene aus dem vorherigen Test übernommen. Die Oberflächen der Wände wurden glänzender eingestellt und weitere Objekte wurden der Szene hinzugefügt. Die zusätzlichen Objekte bieten unterschiedliche abgerundete Oberflächen und dienen der besseren Evaluierung von Spiegelungen. Reflection Captures und SSR wurden hier getrennt betrachtet um genauer zu zeigen, was das jeweilige Verfahren als Ergebnis liefert. Hierbei ist zu erwähnen, dass diese beiden Verfahren in der Praxis oftmals zusammen verwendet werden. Eine maximale Strahlentiefe von 12 wurde bei allen Verfahren verwendet. Um zu testen, ob auch Objekte sich spiegeln können, welche nicht direkt von dem Betrachter gesehen werden können, wurde eine rote Kugel hinter dem kleineren Quader positioniert.

(a) Ref. Captures
00:00:01
00:12:15

(b) SSR
00:00:01
00:12:15

(c) Raytracing
00:19:03

(d) Pathtracing
01:49:48

Abbildung 4.2: Zeigt die fertigen Renderings des Tests mit den jeweiligen Renderzeiten. Die untere Angabe bei den Bildern 4.2a und 4.2b ist die für das Light Baking benötigte Zeit

Der Vergleich zeigt größere Unterschiede in verschiedenen Bereichen. In Abbildung 4.2a ist die direkte Spieglung der Lichtquelle auf der weißen Wand nicht zu sehen, weiterhin sind die sich spiegelden Objekte nur diffus zu sehen, dies ist aktuell eine der Limitierungen dieser Technik in Unreal. Abbildung 4.2b zeigt sehr auffälige Artefakte in den Reflexionen, weiterhin ist der rote Ball nicht in der Spieglung zu sehen da SSR, wie in 3.6 beschrieben, nur Reflexionen mit Objekten generieren kann, welche für den Betrachter direkt sichtbar sind. Unreals Raytracing erzielt ein Ergebnis das bis auf die Helligkeit der Reflexionen fast identisch zu Arnold ist. In den Reflexionen sind akkurat alle Objekte der Szene zu sehen. Die verdunkelten Reflexionen werden durch das Fehlen der indirekten Beleuchtung verursacht. Diese wird aktuell bei Reflexionen nicht berücksichtigt, was ebenfalls eine Limitierung bedeutet, aber gleichzeitig einen beschleunigenden Effekt hat. Arnold (Abbildung 4.2c) zeigt bei diesem Vergleich das genaue Ergebnis.

4.1.3 Transmission

Um Transmission zu vergleichen, wurde eine neue Szene erstellt. Hierzu wurde eine als Vergrößerungsglas wirkende Linse modelliert, welche in der Szene vor Zylindern aus verschiedenen Materialien steht. Sie verwendet ein Glas Material mit einem Index of Refraction (IOR) von 1,6. Dies führt dazu das die Linse den Bereich den sie verdeckt vergrößert. Der mittlere Zylinder verwendet das Glas Material der Linse, um zu sehen wie sich Glas hinter Glas verhält. Zusätzlich verwenden zwei Zylinder ein Chrom Material zum Beurteilen der Reflexionen auf Objekten hinter Glas. Dem Hintergrund wurde eine Textur zugewiesen, um besser zu identifizieren, wie sich die Refraktion verhält. Zur Ausleuchtung der Szene wurde ein HDR Panorama und ein Directional Light benutzt. Das HDR Panorama hat den zusätzlichen Effekt, das es in Spieglungen sichtbar ist. Für diesen Test wurde eine Strahlentiefe von 12 bei Transmission verwendet.

Da in Unreal Engine die Vertex Normals nur mit einer beschränkten Genauheit darstellt werden um Speicher zu sparen, muss bei dem Modell der Linse die Option High Precision Vertex Normal aktiviert werden. Ansonsten führt Dies zu ungenauen, unschönen Reflexionen, welche besonders sichtbar werden in Kombination mit steigendem IOR, siehe Abbildung 4.4. Durch diese Option werden Vertex Normals mit 16 Bit pro Kanal codiert[3]. Für das Glas Material im Light Baking Test wurde eine angepasste Version des Materials verwendet, welches in der Unreal Dokumentation als Beispiel[4] beschrieben ist. Hierbei ist zu erwähnen, dass bei Glas ohne Raytracing ein eher künstlerischer Ansatz verfolgt wird, bei dem oft mit zusätzlichem Metallic Anteil gearbeitet wird. Die verwendeten Einstellungen sind in Abbildung 4.3 zu sehen. Das Glas Material im Raytracing Test verfolgt einen realistischeren Aufbau und ist ebenfalls in Abbildung 4.3 zu sehen. In Arnold wurde die Glas Voreinstellung verwendet und wie folgt abgeändert: Transmission 0.8, IOR 1.6, Base Weight 1 und Base Color (0,0,0).

[3] https://www.unrealengine.com/en-US/blog/unreal-engine-4-12-released , Stand: 01. April 2022

[4] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Materials/HowTo/Refraction/ , Stand: 01. April 2022

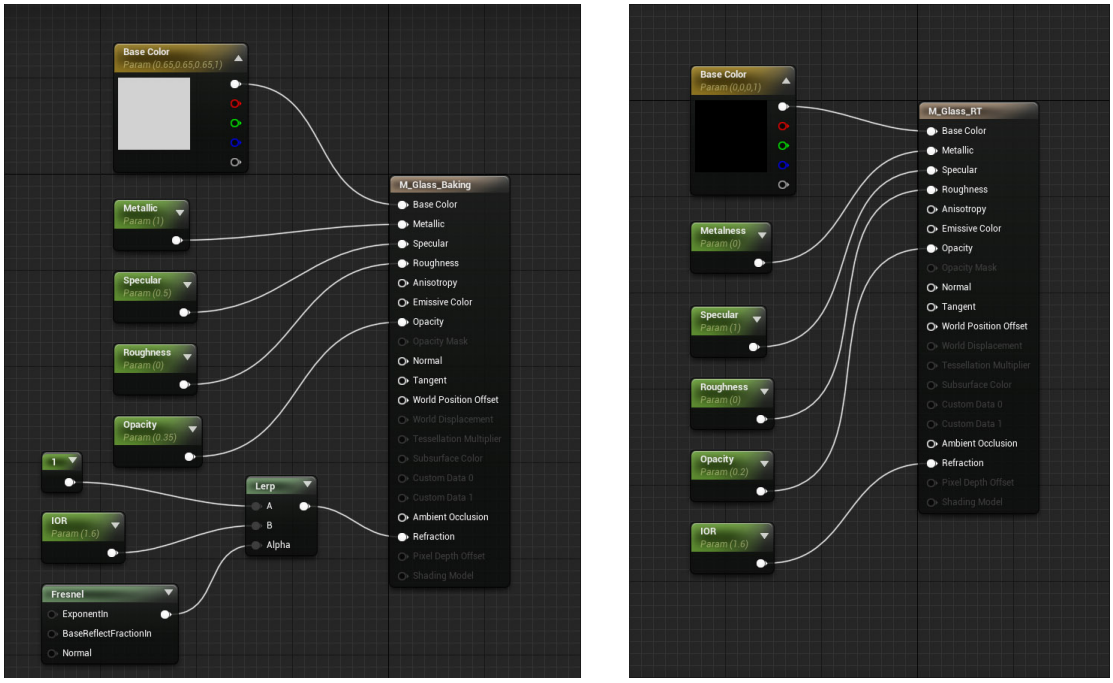
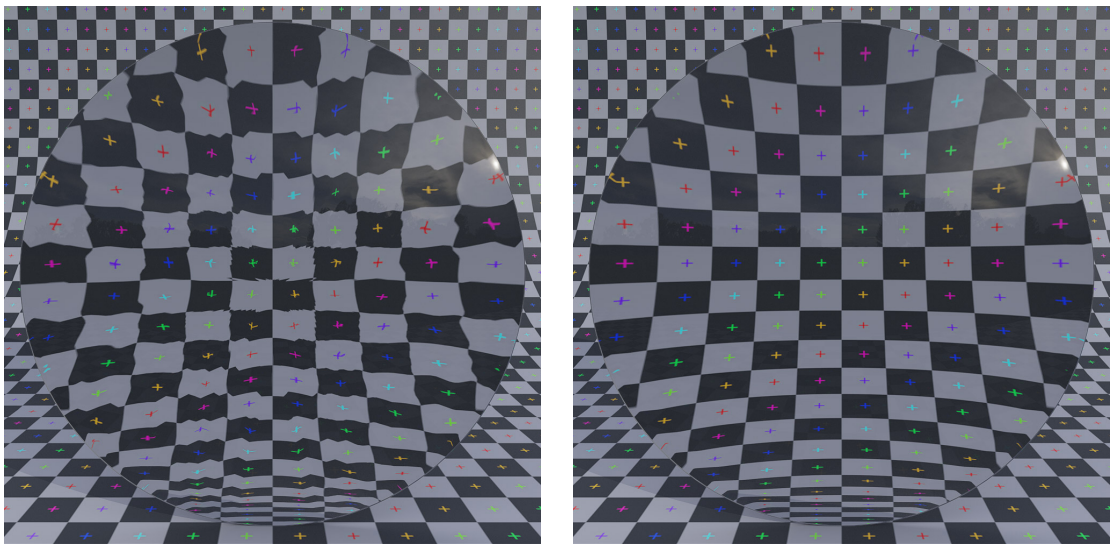
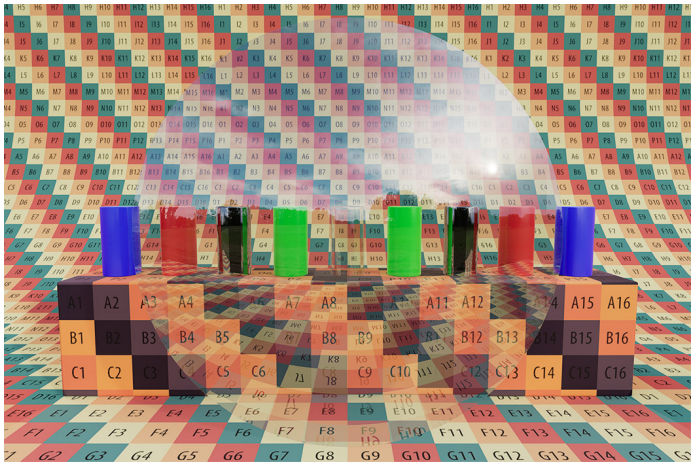


Abbildung 4.3: Zeigt den Aufbau der beiden in Unreal Engine verwendeten Glas Materialien. Links ist das Material für den Lightbaking Test und Rechts das für Raytracing zu sehen.

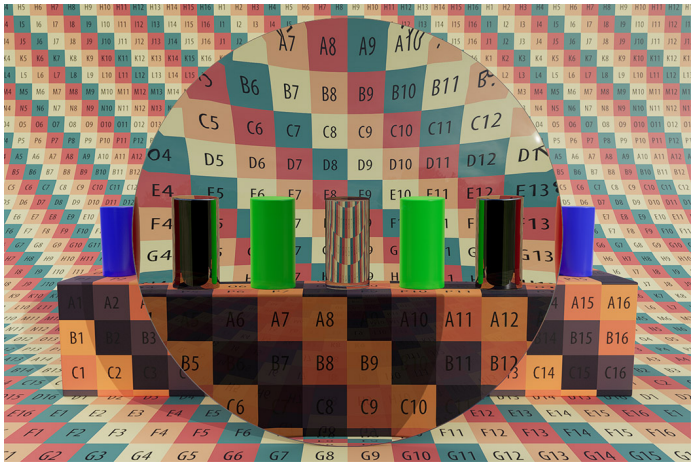


(a) High Precision Vertex Normal Off (b) High Precision Vertex Normal On

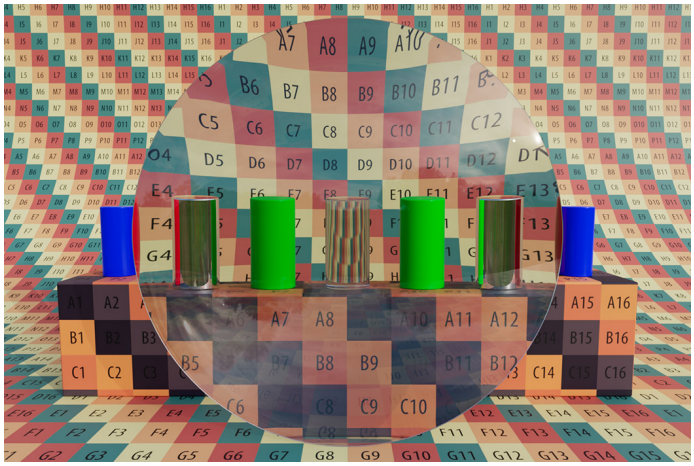
Abbildung 4.4: Zeigt den Effekt welchen die High Precision Vertex Normal Einstellung in Unreal Engine hat.



(a) Light Baking: 00:00:02 + 00:01:08



(b) Raytracing: 00:06:01



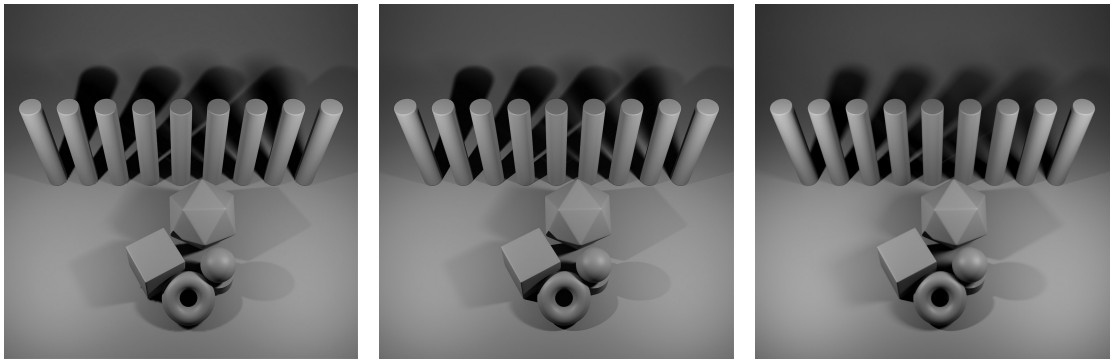
(c) Pathtracing: 00:51:20

Abbildung 4.5: Zeigt die fertigen Renderings der Transmissionstests mit den jeweiligen Renderzeiten. Bei Bild 4.5a wird zusätzlich die benötigte Zeit für das Light Baking addiert.

Der Vergleich zeigt sichtbare Unterschiede zum Arnold Rendering. Das bei Abbildung 4.5a eingesetzte Glas Material war nicht in der Lage, den IOR von 1,6 korrekt nachzubilden, zusätzlich wirkt das Glas tiefenlos und zu spiegelnd, was von der verwendeten Metallic Einstellung verursacht wird. Das Raytracing Ergebnis (Abbildung 4.5b) liefert ein genaueres Bild, die Refraktion wird korrekt abgebildet und alles ist sichtbar. Auffällig ist jedoch, dass die Reflexionen zu schwach erscheinen und in den spiegelnden Zylindern sich schwarze Flächen gebildet haben. Durch Aufhellen der schwarzen Flächen in Photoshop ist es möglich, die Reflexion zu sehen. Dies wird vermutlich verursacht durch die fehlende Berücksichtigung von indirekter Beleuchtung in Reflexionen, welche im vorherigen Test festgestellt wurde. Arnold zeigt gut unterscheidbar das genaue Ergebnis und bietet zusätzlich noch weitere Möglichkeiten, die in Unreal nicht zur Verfügung stehen, wie beispielsweise Dispersion oder Kaustiken. Für Kaustiken ist jedoch zu erwähnen, dass es bereits einen Build[5] der Engine gibt, der Kaustiken unterstützt.

4.1.4 Schatten

Zum Überprüfen der Schatten wurde eine Szene mit einfachen Objekten und zwei unterschiedlich großen Lichtquellen erstellt (Rect Lights/Area Lights). Rechts befindet sich eine größere Lichtquelle, welche weiche Schatten wirft und links eine kleinere Lichtquelle, die harte Schatten wirft. Es wurde eine Reihe von hohen Zylindern platziert, die lange Schatten werfen soll. Dies ermöglicht eine bessere Beurteilung der Schatten in Bezug auf die Randschärfe.



(a) Lightmaps
00:00:01
00:03:19

(b) Raytracing
00:00:37

(c) Pathtracing
00:08:31

Abbildung 4.6: Fertige Renderings der Schatten-Tests mit den zugehörigen Renderzeiten. Für Bild 4.6a ist zusätzlich noch die Zeit für das Light Baking angegeben.

[5] https://developer.nvidia.com/blog/generating-ray-traced-caustic-effects-in-unreal-engine-4-part-1/ , Stand: 2. April 2022

Die Ergebnisse des Tests zeigen, dass Lightbaking (Abbildung 4.6a) und Raytracing (Abbildung 4.6b) ein fast identisches Ergebnis erzielt haben. Lightbaking unterscheidet sich hierbei visuell nur von Raytracing durch auflösungsbedingte Lightmap Artefakte. Raytracing hat jedoch nicht die Limitierung, da es nicht an eine Lightmap Auflösung gebunden ist. Arnold (Abbildung 4.6c) unterscheidet sich in diesem Vergleich von den beiden Unreal Renderings durch weichere, schneller auslaufende Schatten. Dies führt im Detail betrachtet zu einem realistischeren Ergebnis.

4.2 Erstellung von Lightmaps in Arnold

Zusätzlich ist zu erwähnen, dass Arnold auch das Berechnen von Lightmaps unterstützt. Dabei wird die Strahlentiefe aus den Rendereinstellungen übernommen und das Sampling kann im Dialog "Render to Texture" im Feld "Camera Samples (AA)" definiert werden. Diese Funktion wird pro selektiertem Objekt angewendet. Sobald mehrere Objekte selektiert sind, werden mehrere einzelne Texturen mit den spezifizierten Einstellungen gerendert. Dies kann jedoch in komplexeren Szenen schnell aufwendiger werden wegen der hohen Anzahl von Objekten, speziell im Bezug auf pro Asset variierende Lightmap Auflösungen.

Hinzu kommt, dass scheinbar keine Möglichkeit besteht, diese Lightmaps automatisiert mit verschiedenen Einstellungen zu generieren. In der Regel empfiehlt es sich nicht, alle Assets anzuwählen und global eine Lightmap-Auflösung für alle zu verwenden. Ansonsten bekommen Assets ungewollt zu hohe oder zu niedrige Lightmap-Auflösungen. Somit müssten Assets nach Auflösungen gruppiert werden und dann Gruppe für Gruppe gerendert werden oder die UVs der Assets angepasst werden, um die Auflösung anzugleichen.

Weiterhin verwendet Arnold hierzu nur einen UV-Channel. Dies kann zu Problemen führen bei Workflows, bei denen Assets optimiert wurden für die Verwendung mit sich wiederholenden Texturen[6]. Unreal umgeht dieses Problem, indem zwei UV-Channels verwendet werden, darauf wird im folgenden Abschnitt (4.3.1) genauer eingegangen. Aufgrund dieser Erkentnisse wird die Option in dieser Arbeit nicht genauer behandelt.

[6] https://docs.arnoldrenderer.com/display/A5AFMUG/Render+Selection+to+Texture , Stand: 09. Mai 2022

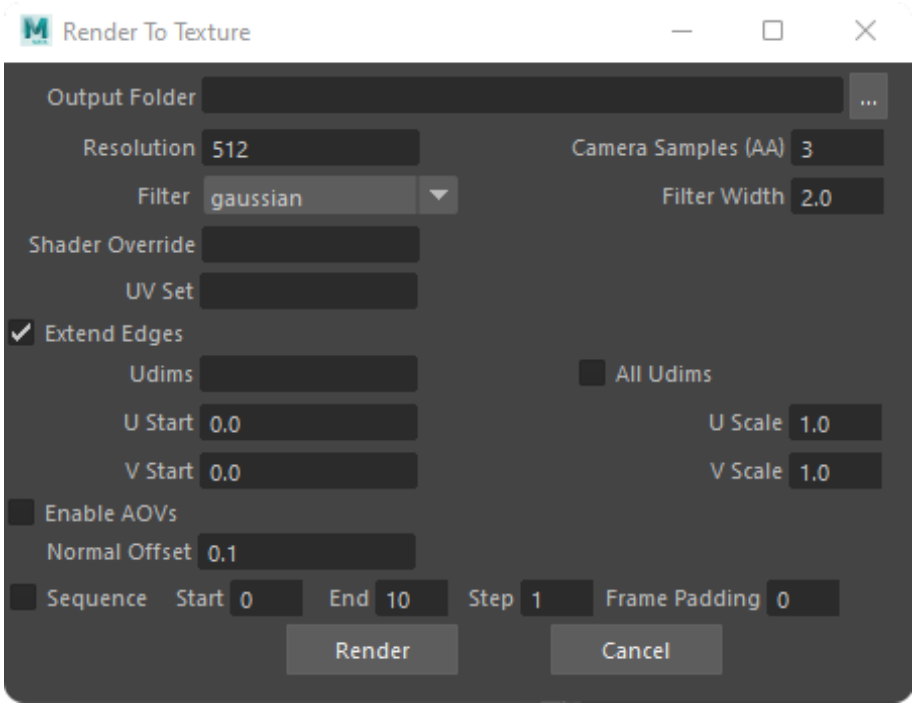


Abbildung 4.7: Zeigt einen Screenshot des Dialogs "Render to Texture" von Arnold in Autodesk Maya. (Quelle: Eigener Screenshot aus Autodesk Maya)

4.3 Konzeption eines realistischen Anwendungsfalls

Um einen realitätsnahen Vergleich zu betrachten, wird ein 3D-Modell einer Küche mit Esszimmerbereich erstellt und anschließend mit den 3 Verfahren Pathtracing (Arnold), Realtime Raytracing und Lightbaking (Unreal Engine) gerendert werden. Der Sinn dieses Vergleichs ist es, den realen Anwendungsfall einer Architektur Visualisierung nachzubilden, dies soll eine bessere Beurteilung ermöglichen als nur einzelne synthetische Tests. Anhand von diesem Versuchsaufbau soll im Kapitel Ergebnisse ausgewertet werden, ob sich eventuelle Einbußen gegenüber eventuellen Geschwindigkeit Vorteilen rechtfertigen. Anhand hiervon soll verdeutlicht werden, welche Ergebnisse bei gleicher Renderzeit zu erreichen sind und welche Ergebnisse bei maximaler Qualität zu erreichen sind.

Das Layout der Küche wird sich an einem bereits existierenden Plan orientieren. Für das Modell der Küche werden Assets wie Wände und Küchenschränke selbst erstellt, um auch auf die besonderen Anforderungen einzugehen, welche Modelle, die in der Unreal Engine verwendet werden, erfüllen sollten. Für zusätzliche Assets wie beispielsweise Einbaugeräte oder Stühle werden importierte Assets verwendet, diese müssen je nach Asset auch aufbereitet werden für die Verwendung in der Unreal Engine. In den folgenden Abschnitten werden Voraussetzungen und Ziele für die Umsetzung des realistischen Anwendungsfalls beschrieben.

4.3.1 Anforderungen an das 3D-Modell

Die Anforderungen an die Modelle teilen sich in technische und visuelle Anforderungen auf. Um einen realistischen Eindruck zu erreichen, sollten alle Modelle nach realen Größen modelliert bzw. skaliert werden. Weiterhin sollten Kanten abgerundet werden, da sich sonst keine Highlights an diesen Stellen bilden können, was wiederum zu einem unschönen Ergebnis führt (siehe Abb. 4.8).

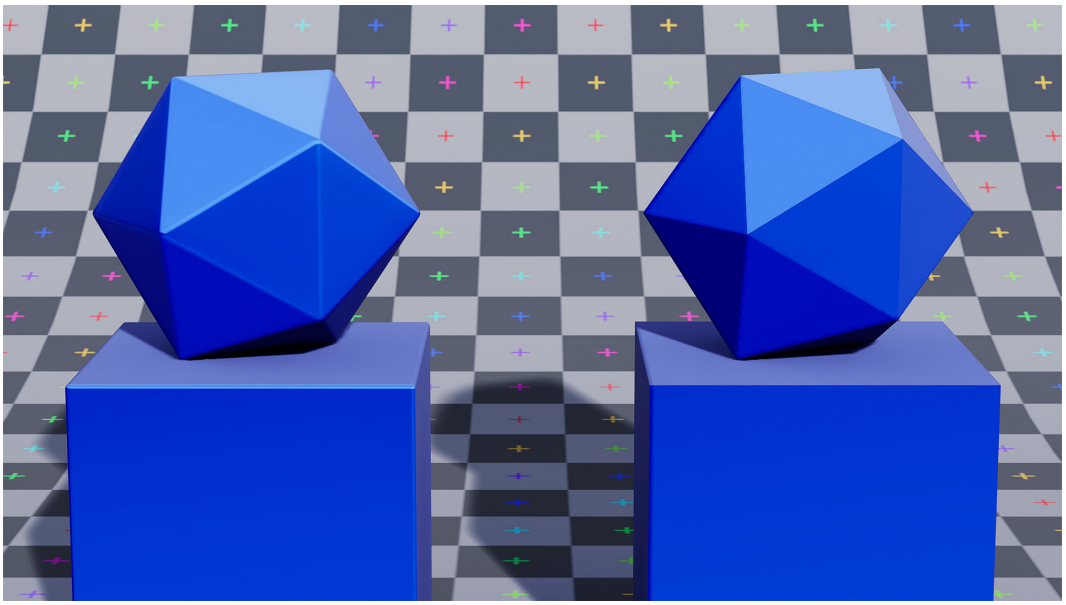


Abbildung 4.8: Beispiel welches den Unterschied zwischen einem Modell mit abgerundet Kanten (Links) und einem Modell ohne abgerunde Kanten (Rechts) zeigt.

Modelle, welche in der Unreal Engine verwendet werden, sollten je nach Einsatzzweck bestimmte technische Anforderungen erfüllen. Der Polycount der Modelle sollte nicht unnötig hoch sein. Als Referenz wird hier der Polycount der ArchViz Interior Szene[7] von Epic Games verwendet. Die Szene beinhaltet das Modell eines Wohnzimmers mit vollständiger Einrichtung und ist bei etwa 6,5 Millionen Polygonen. Für die Visualisierung in diesem Vergleich wird ein kleinerer Wert als Ziel gesetzt da keine zusätzlichen Dekorationen in der Szene verteilt werden. Eine Möglichkeit den Polycount von Modellen zu reduzieren, ist es Rückseiten von Modellen welche der Betrachter sowieso niemals sehen kann zu löschen, da diese nicht notwendig für die Engine sind.

Da der Prototyp auch Lightbaking umfasst, werden UVs für Lightmaps benötigt. In Unreal Engine wird standardmäßig der zweite UV-Channel für die Lightmap verwendet, dementsprechend benötigt jedes Modell zwei UV-Channels. Channel null für Texturen und Channel

[7] https://www.unrealengine.com/en-US/blog/new-archviz-interior-rendering-sample-project-now-available , Stand: 09. März 2022

eins für die Lightmap. Außerdem dürfen sich bei den Lightmap UVs die UV-Shells nicht überlagern, da dies zu Artefakten führen wird, welche nach dem Baking Prozess auffällig sichtbar sind. Des Weiteren dürfen Lightmap UV-Shells nur im Standard 0 bis 1 UV-Space positioniert werden. Um eine hohe Texel Density trotz der eben genannten Einschränkungen zu erreichen, sollten die UV-Shells möglichst platzeffizient gepackt werden. Zusätzlich benötigen die Shells Abstand zueinander, um Lightbaking Artefakte zu vermeiden. Dieser Mindestabstand berechnet sich wie folgt[8]:

$$\frac{1}{Target\ Lightmap\ Texture\ Resolution} = Texel\ Grid\ Spacing \tag{4.1}$$

Wobei Unreal Engine einen Pixel als Padding benötigt, dementsprechend würde man hier anstelle von beispielsweise 512, 510 einfügen.

Die Modelle sollten sinnvoll kombiniert werden und nicht aus vielen einzelnen Bauteilen bestehen, damit der FBX-Import Prozess einfacher ist und die Anzahl der unnötigen Drawcalls minimiert werden kann. Um später in Unreal Engine passende Materialien an den entsprechenden Stellen zuweisen zu können, müssen vorher in Maya diesen Stellen Platzhalter Materialien zugewiesen werden, siehe Abb. 4.9.

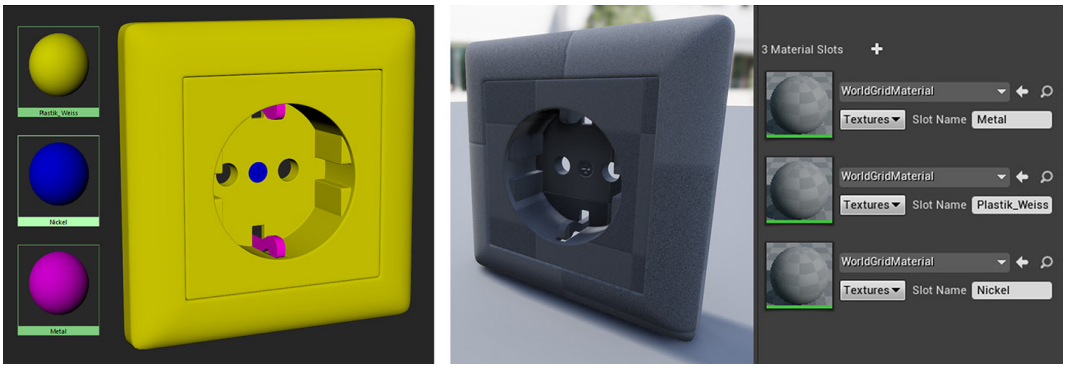


Abbildung 4.9: Links ist ein vorbereitetes Asset in Maya, bei dem jedem Bauteil welches aus einem andern Material besteht, ein Platzhalter (eine andere Farbe) zugewiesen wurde. Rechts ist dasselbe Modell in Unreal Engine mit den automatisch beim Import angelegten Material Slots.

Weil in diesem Vergleich Bilder in hoher Auflösung gerendert werden, braucht es hochauflösende Texturen für Oberflächen, die einen großen Bildbereich abdecken. Texturen die sichtbar sind, aber nicht viel Platz im fertigen Bild einnehmen, sollten entsprechend niedrigere Auflösungen verwenden, um Grafikspeicher einzusparen. Genaue Texturauflösung Werte sind hierbei von der Auflösung des Renderings abhängig.

[8] https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Types/StaticMeshes/LightmapUnwrapping/ , Stand: 08. März 2022

4.3.2 Renderings

Um festzulegen, was als hochauflösend gilt, wurde der Standard ITU-R BT.2020[9] betrachtet. Dort sind die Auflösungen 4K (3840px x 2160px) und 8K (7680px x 4320px) als "Ultra High Definition", Deutsch: ultrahochauflösend definiert[10], weswegen mit dem Prototyp insgesamt zwei 4K Renderings im 16:9 Format umgesetzt werden sollen. Das erste Rendering soll eine Gesamtaufnahme der Küche aus der Mitte des Raumes zeigen. Im zweiten Rendering soll der Esszimmerbereich aus Sicht des Küchenbereichs zu sehen sein. Für einen zweiten Vergleich sollen mehrere 4K Bilder mit einer jeweils grob festgelegten Renderdauer erstellt werden. Dies soll durchgeführt werden mit Unreals Raytracing und Arnold. Dabei wird überprüft, was die beiden Renderer bei derselben vorgegebenen Zeit von jeweils: 1, 2, 5 und 10 Minuten erreichen können.

4.3.3 Panoramen

Für Architektur Visualisierungen werden oftmals mehrere 360° Panoramen gerendert und mithilfe von einem weiteren Tool zu einer sogenannten virtuellen Tour zusammengefügt, dies ermöglicht es dem Betrachter, sich in diesen Bildern umzusehen und interaktiv von Bild zu Bild zu springen, ähnlich wie in Google Streetview[11]. Dieser Ansatz ist beliebt, da es dem Betrachter das Gefühl gibt, selbst durch die Visualisierung zu laufen und er sich somit einen besseren räumlichen Eindruck machen kann. Ein weiterer Vorteil dieser Umsetzung ist, dass keine leistungsstarke Hardware auf der Seite des Betrachters notwendig ist, um diese Bilder anzusehen.

Abbildung 4.10: Das Bild zeigt ein in Unreal Engine gerendertes 360° Panorama. (Quelle der verwendeten Szene: https://www.unrealengine.com/marketplace/en-US/product/archvis-interior-rendering , Stand: 10. April 2022)

[9] https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.2020-2-201510-I!!PDF-E.pdf , Stand: 06. Mai 2022

[10] https://kompendium.infotip.de/bt2020.html , Stand: 27. April 2022

[11] https://www.google.com/intl/de/streetview/ , Stand: 03. April 2022

Besonders für diesen Anwendungszweck sind hohe Auflösungen notwendig, da hier ein 180° x 360° Sichtfeld auf ein rechteckiges 2:1 Bild abgebildet wird. Dies wird mittels Equirectangular Projektion[12] realisiert. Ein Beispiel eines solchen Bildes ist in Abbildung 4.10 zu sehen. Der Betrachter sieht jedoch von diesem Bild immer nur einen bestimmten Bereich, weshalb die Gesamtauflösung höher sein muss. Das Sichtfeld kann je nach Einstellung oder Hardware variieren, jedoch hält es sich in der Regel zwischen 90° und 130° auf[13]. Mit der folgenden Formel kann ausgerechnet werden, wie viele Pixel der Betrachter in einem vorgegebenen Sichtfeld sehen kann. Um die Vertikale mit dieser Formel zu berechnen, müssen die Auflösung und das FOV für die Vertikale eingesetzt werden und die 360 gegen 180 ersetzt werden.

$$\frac{Auflösung\ Horizontal * FOV\ Horizontal}{360} = Sichtbare\ Auflösung\ Horizontal \quad (4.2)$$

Mit dieser Formel kann ausgerechnet werden, dass bei einer Gesamtauflösung von 4K und einem FOV von horizontal 110° und vertikal 90° die für den Betrachter sichtbare Auflösung gerade mal 1173 x 1080 Pixel beträgt, was gerade mal 1,3 Megapixel entspricht. Anhand dieses Beispiels sollte gut erkennbar sein das hier Auflösungen von 8K oder mehr benötigt werden um eine für heutige Verhältnisse angemessene Auflösung zu bieten. Hinzu kommt das für eine solche Visualisierung oftmals mehrere Standpunkte benötigt werden. Die hohe Auflösung, oftmals in Kombination mit der Anzahl der benötigten Bilder, sind der Hauptgrund weshalb für diesen Zweck schnellere Lösungen gesucht werden.

4.4 Objektive Betrachtung anhand einer Metrik

Um eine Personen-unabhängige Bewertung zu realisieren, muss eine objektive Metrik eingesetzt werden. Zwei bekannte Metriken zum Vergleichen von Bildern sind Peak-signal-to-noise-ratio (PSNR) und Structural Similarity Index (SSIM) [WBSS04]. PSNR gibt das Verhältnis zwischen einem maximal Referenzsignal und einem verrauschten Signal an. Als Signal werden zum Beispiel die Helligkeitswerte von einem Bild verwendet, dabei wird Pixel für Pixel verglichen. Die Angabe erfolgt logarithmisch in dB, wobei ein höherer Wert näher an dem Referenzbild (Ground Truth) ist. Für die Auswertung von Farbbildern kann das Bild in ein Graustufenbild konvertiert werden und anschließend betrachtet werden, da das menschliche Auge empfindlicher auf Luminanz-Änderungen reagiert als auf Chrominanz-Änderungen[14]. Diese Metrik eignet sich gut, um zu ermitteln, wie sehr ein verrauschtes

[12] https://wiki.panotools.org/Equirectangular_Projection , Stand: 10. März 2022

[13] https://risa2000.github.io/hmdgdb/ , Stand: 10. März 2022

[14] https://www.ni.com/de-de/innovations/white-papers/11/peak-signal-to-noise-ratio-as-an-image-quality-metric.html , Stand: 27. April 2022

Rendering von einem sauberen Rendering abweicht[15]. PSNR soll hier verwendet werden, um eine numerische Bewertung der Vergleiche umzusetzen, welche die Ähnlichkeit der Bilder bewertet.

SSIM ist eine Metrik, welche versucht, Bilder menschlicher zu betrachten, indem strukturelle Informationen miteinbezogen werden. SSIM betrachtet dabei hauptsächlich drei Faktoren: die Luminanz, den Kontrast und die Struktur. Das Ergebnis von SSIM ist eine Zahl zwischen -1 und $+1$, wobei $+1$ aussagt, dass die Bilder identisch sind[16]. Zusätzlich zu der Auswertung mit PSNR sollen die Renderings aus den Vergleichen mit SSIM bewertet werden. Da Arnold das genaueste Verfahren der hier betrachteten Verfahren ist, werden die Arnold Renderings im ersten Vergleich als Referenz eingesetzt und gegen die Unreal Engine Renderings verglichen. Im zweiten Vergleich sollen SSIM und PSNR genutzt werden, um die verrauschten Bilder mit der Referenz aus dem jeweiligen Renderer vergleichen.

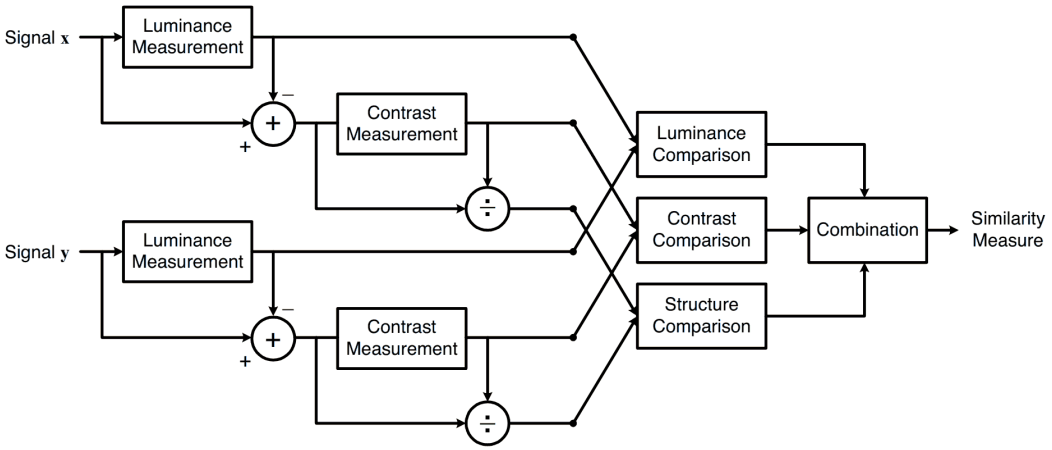


Abbildung 4.11: Zeigt eine Übersicht des SSIM Messungssystems. (Quelle: [WBSS04])

[15] https://videoprocessing.ai/metrics/ways-of-cheating-on-popular-objective-metrics.html , Stand: 27. April 2022

[16] https://medium.com/srm-mic/all-about-structural-similarity-index-ssim-theory-code-in-pytorch-6551b455541e , Stand: 27. April 2022

Kapitel 5

Erstellen des Prototyps

In diesem Kapitel wird auf die Umsetzung des Prototypen eingegangen. In dem folgenden Abschnitt 5.1 wird erklärt, wie die Szene aufgebaut ist und wie Assets erstellt wurden. In den darauf folgenden Abschnitten wird der Umsetzungsprozess in den jeweiligen Programmen (Maya und Unreal Engine) beschrieben, hierbei wird hauptsächlich auf Rendereinstellungen und Licht-Einstellungen eingegangen.

5.1 Aufbau der Szene

Zum Erstellen der Modelle wurde Autodesk Maya verwendet, wobei hierfür auch andere Programme wie Blender oder 3ds Max eingesetzt werden können. Seit der Version 4.26[1] verfügt Unreal Engine auch über eigene Modellier Tools, welche direkt im Editor verfügbar sind, jedoch sind diese noch sehr einfach gehalten, verglichen mit Maya. Zusätzlich wurden kostenfreie 3D-Modelle von Küchengeräten von der Webseite 3DSky[2] importiert. Importierte Modelle verursachen meistens mehr Aufwand als zuerst angenommen, da diese oftmals für andere Zwecke erstellt wurden. Die importierten Modelle wurden in Maya aufbereitet nach den Prinzipien, die in 4.3 beschrieben wurden, um sie in Unreal Engine zu verwenden. Eine vollständige Liste aller Quellenangaben dieser Modelle befindet sich im Anhang A.

[1] https://docs.unrealengine.com/4.27/en-US/WhatsNew/Builds/ReleaseNotes/4_26/ , Stand: 10. März 2022

[2] https://3dsky.org , Stand: 10. März 2022

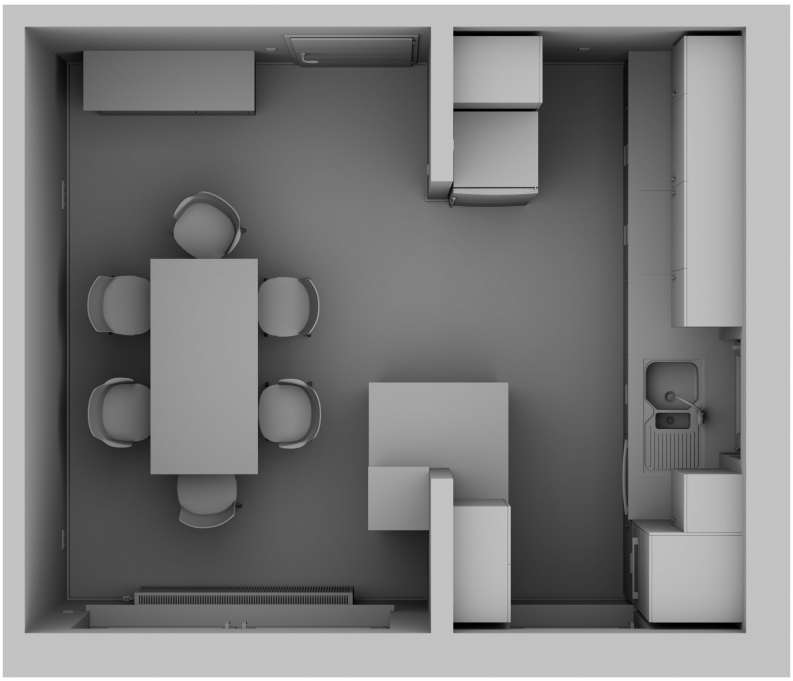


Abbildung 5.1: Ein mit Arnold erstelltes Occlusion Rendering, es zeigt die Draufsicht des fertigen 3D-Modells der Küche. Hierfür wurden die Lampen und die Decke ausgeblendet.

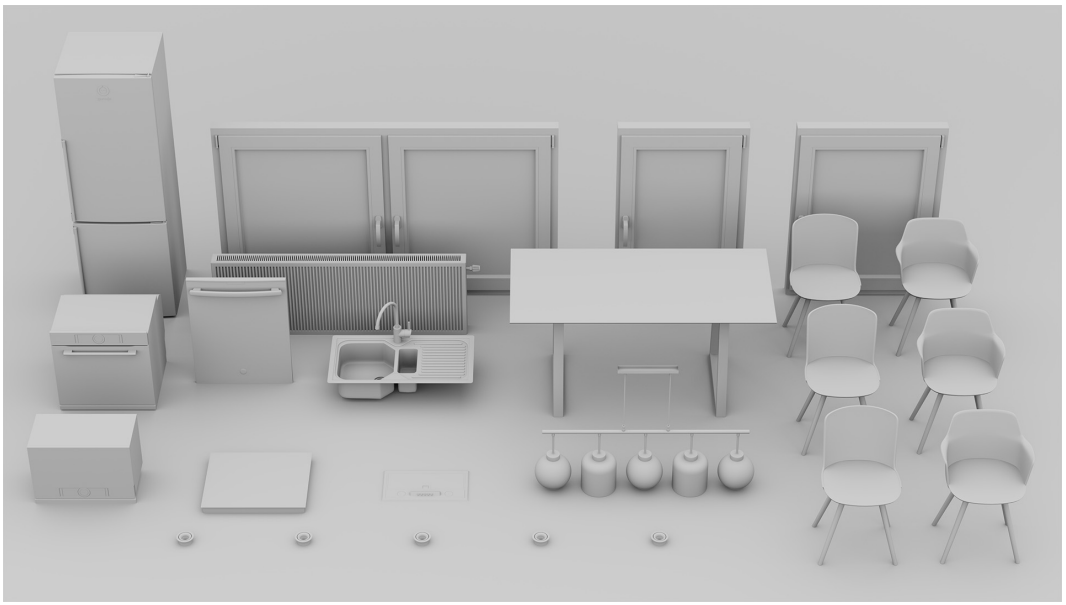


Abbildung 5.2: Übersicht der fertig angepassten 3D-Modelle, die von 3DSky importiert wurden.

5.1.1 Modell Erstellung

In den folgenden Absätzen wird beispielhaft der Aufbau eines selbst erstellten Assets gezeigt. Die Modelle wurden nach realen Maßen modelliert und in der Szene positioniert. Eine Übersicht der Modelle ist in den Abbildungen 5.1 und 5.2 zu sehen. Um den Polycount der Modelle nicht unnötig in die Höhe zu treiben, wurden die Rückseiten gelöscht und Edgeloops nur dann verwendet, wenn diese auch notwendig waren. Die meisten Polygone sind durch das Abrunden der Kanten entstanden. Abbildung 5.3 zeigt den größten Schrank, der in der Küche positioniert wurde, der Polycount des Modells beträgt 6054. Beim Erstellen der Modelle mussten Vertiefungen eingeplant werden für Modelle, welche später hinzugefügt wurden, wie beispielsweise die Spüle. Hierfür wurde lediglich ein Face nach innen extrudiert, um so überlagernde Geometrien zu vermeiden. Dieser Schritt wurde bei der großen Küchenzeile durchgeführt und bei allen Wänden, an denen Steckdosen angebracht wurden.

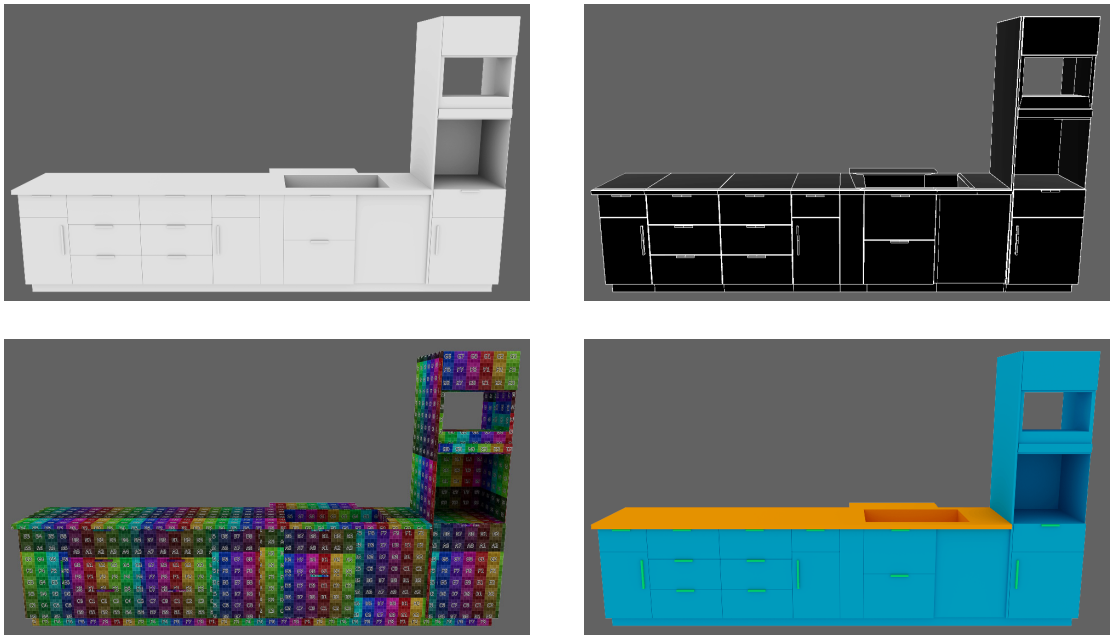


Abbildung 5.3: Verschiedene in Arnold gerenderte Ansichten von dem Küchenzeilenmodell. Zu sehen ist oben links ein Occlusion Rendering, oben rechts ein Kontrast-optimiertes Wireframe Rendering, unten links das Modell mit einer UV-Testmuster-Textur und unten rechts das Modell mit den zugewiesenen Materialplatzhaltern.

5.1.2 UV-Layout

Beim Erstellen des UV-Layouts der Modelle musste darauf geachtet werden, eine einheitliche Texel-Density in der Szene zu haben, um verschieden skalierte Texturen je nach Modell zu vermeiden. Da bei den selbst erstellten Modellen nur mit sich wiederholenden Texturen gearbeitet wurde, konnten UV-Shells auch überlagernd positioniert werden und den 0 bis 1

UV-Space verlassen. Weil für den Test mit baked Lighting in Unreal Engine eine Lightmap erstellt werden muss, wurde wie in 4 beschrieben ein zweiter UV-Channel angelegt. Bei den meisten Assets konnte hierfür einfach das Layout aus Channel 0 dupliziert und übernommen werden und gegebenenfalls noch passend skaliert werden. Bei Modellen, wo dies nicht möglich war, musste ein neues Layout erstellt werden, siehe Abbildung 5.4. Hierfür reichte es meistens aus, die Layoutfunktion des UV-Editors mit den passenden Einstellungen für Padding zum neu-Packen der UVs zu verwenden. Bei den Lightmap UVs wurde weitestgehend auch darauf geachtet, eine einheitliche Texel Density zu haben, bei manchen Assets war das jedoch problematisch, da alle Shells sich im 0 bis 1 UV-Space befinden müssen und sich auch nicht überlagern dürfen. Um die unterschiedlichen Auflösungen auszugleichen, bietet Unreal Engine jedoch eine Lightmap Auflösungs Skalierung pro Actor an. Als letzter Schritt wurden dem Modell die in 4 beschriebenen Platzhalter-Materialien zugewiesen, siehe Abbildung 5.3 unten rechts.

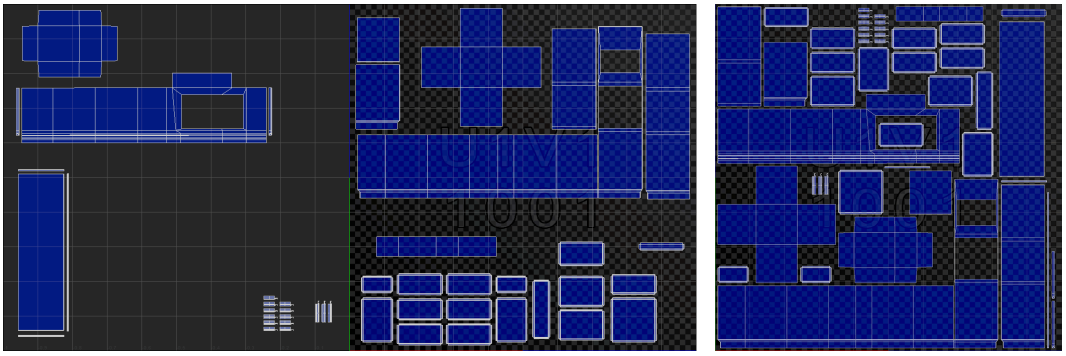


Abbildung 5.4: Zeigt das UV-Layout des Assets welches in 5.3 zu sehen ist. Links ist das UV-Layout in Channel 0, welches für Texturen vorgesehen ist. Rechts zeigt das UV-Layout in Channel 1, das für die Lightmap in Unreal Engine vorgesehen ist.

5.1.3 Texturen

Um die Auswahl für visuell zu der Szene passende Texturen zu vereinfachen, wurde das Modell in die Unreal Engine importiert, damit können diese direkt auf dem eigentlichen Modell mit realistischer Beleuchtung betrachtet werden. Als Quelle für Texturen wurde Quixel Bridge[3] verwendet. Quixel Bridge (Abbildung 5.5) ist ein Programm, welches die Quixel Megascanns Bibliothek[4] bereitstellt. Diese Bibliothek besteht aus einer großen Anzahl von Texturen und 3D-Modellen, die über Fotogrammetrie erstellt wurden. Quixel stellt vollständige Materialien mit Normal, Roughness und weiteren Maps bereit. Eine nützliche Information, die im Programm gelistet wird, ist die reale Größe der gescannten Fläche, mit dieser Information lässt sich ein Material genau auf die richtige Größe skalieren durch Textur Wiederholung oder skalieren der UV-Shells. Die Auflösung der Texturen kann in Quixel Bridge ausgewählt werden, diese reicht meistens von 2K bis hin zu 8K. Mithilfe von Plugins in den jeweiligen Zielprogrammen kann Quixel Bridge mit einem Klick ausgewählte Assets oder Materialien

[3] https://quixel.com/bridge , Stand: 12.März 2022
[4] https://quixel.com/megascans/ , Stand: 12.März 2022

korrekt exportieren und im jeweiligen Programm einrichten.

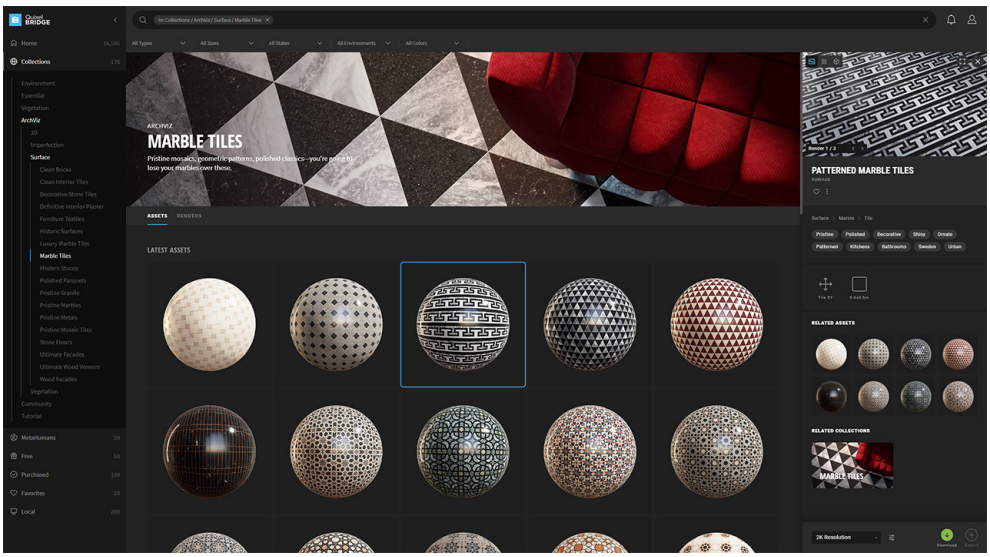


Abbildung 5.5: Zeigt einen Screenshot der Quixel Bridge Benutzeroberfläche. Auf der rechten Seite finden sich Informationen wie die Größe der gescannten Fläche und die Exportauflösung. (Quelle: Eigener Screenshot der Quixel Bridge Benutzeroberfläche)

5.1.4 Lichtquellen

Für die Ausleuchtung der Szene wurden drei längliche Arealights unterhalb der Küchenschränke platziert, welche LED Lampen zur Beleuchtung der Arbeitsfläche darstellen. In der folgenden Umsetzung sollen diese Lampen ein warmweißes Licht ausstrahlen. Für die globale Ausleuchtung der Szene war ursprünglich ein HDR Panorama in Kombination mit einem Directional Light vorgesehen. Dies wurde jedoch problematisch in der Raytracing Umsetzung, da es hier zu Beleuchtungsfehlern kam, welche durch das Skylight in Unreal verursacht wurden. Aus diesem Grund wurde die HDR Panorama Beleuchtung ersetzt gegen Area Lights, die vor den Fenstern positioniert sind. Derselbe Ansatz wird von Epic Games in ihrer ArchViz Demoszene[5] verwendet.

5.1.5 Kameras

Die Kameras wurden wie in 4 beschrieben ausgerichtet. Die Brennweite beider Kameras wurde sehr weitwinklig gewählt, um möglichst viel von dem Raum abzubilden. Beide Kameras nutzen eine Sensorgröße von 23,76 mm x 13,365 mm, diese Größe entstammt einer Voreinstellung, welche in Unreal Engine auswählbar ist. Kamera 1 nutzt eine Brennweite von 16 mm und zeigt das Gesamtbild der Küche, siehe Abbildung 5.6. Kamera 2 nutzt eine Brennweite von 12 mm und zeigt den Esszimmerbereich, siehe Abbildung 5.7.

[5] https://www.unrealengine.com/en-US/blog/new-archviz-interior-rendering-sample-project-now-available , Stand: 24. April 2022

Abbildung 5.6: Wireframe Rendering aus der Sicht von Kamera 1

Abbildung 5.7: Wireframe Rendering aus der Sicht von Kamera 2

5.2 Setup Prozess mit Arnold

Die Einstellungen der Materialien aus Unreal wurde in Arnold übernommen. Für sehr spiegelnde Oberflächen wurde zusätzlich der Albedowert leicht reduziert, da wie in 4.1.2 festgestellt wurde die indirekte Beleuchtung bei den Unreal Engine Raytracing Reflexionen nicht berücksichtigt wird. Dies soll vermeiden, dass diese Oberflächen in Arnold fälschlich zu hell eingestellt werden. Die Lichtquellen wurden visuell an die Lichtquellen in Unreal Engine angepasst. Dieser Vorgang musste visuell erfolgen, da die Verwendung desselben Werts bei allen drei Verfahren anders interpretiert wird. Die Rendereinstellungen für ein annehmbar rauschfreies Bild wurden in Arnold mithilfe der AOVs[6] (Arbitrary output variables) ermittelt, durch sie können einzelne Bestandteile des Renderings wie z.B indirect diffuse oder direct specular gesondert betrachtet werden. Dadurch kann zuverlässig erkannt werden, welche Samples angepasst werden müssen, um Rauschen zu entfernen[7]. Die endgültigen Einstellungen sind in Abbildung 5.8 zu sehen. Für den zweiten Vergleich in 6.2 wurden die Rendereinstellungen für jedes Bild einzeln angepasst. Da es in Arnold nicht die Option gibt, zu definieren, dass eine bestimmte Zeit lang gerendert werden soll, mussten durch Ausprobieren die bestmöglichen Einstellungen für den jeweiligen Zeitintervall ermittelt werden. Die genauen verwendeten Rendereinstellungen zu jedem der Zeitintervalle ist im Anhang B aufgelistet.

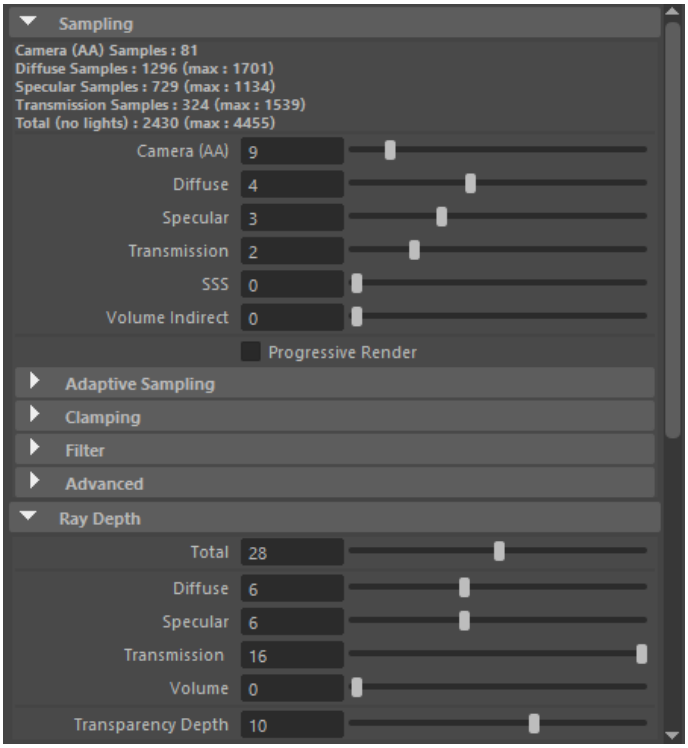


Abbildung 5.8: Zeigt die Arnold Rendereinstellungen für die Renderings in 6.1c und 6.2c

[6] https://docs.arnoldrenderer.com/display/A5AFMUG/AOVs+for+Image+Compositing , Stand: 25. April 2022

[7] https://docs.arnoldrenderer.com/display/A5AFMUG/Removing+Noise , Stand: 25. April 2022

5.3 Setup Prozess mit Unreal

Der Szene wurde eine zusätzliche Geometrie (Shadowbox) hinzugefügt, welche das gesamte Modell des Raumes umgibt. Dadurch soll light leaking Artefakten vorgebeugt werden. Das in 5.1.4 beschriebene HDR Panorama wurde als Szenenhintergrund, welcher durch die Fenster sichtbar ist, verwendet. Hierzu wurde Unreals HDR Backdrop benutzt und das darin beinhaltete Skylight ausgeschaltet.

5.3.1 Konfiguration des Lightmass Renderings

Für das Light Baking in Unreal wurde die CPU Version von Lightmass verwendet. Zuerst wurde die Lightmap Auflösung der Assets in der Szene angepasst, so dass diese auch bei hohen Auflösungen passend aussehen. Bei diesem Vorgang war es hilfreich, dass mit einheitlichen Texel Densitys gearbeitet wurde. Weiterhin ist der View Mode[8] "Lightmap Density" sehr hilfreich, da dieser die Szene mit einer Farbskala-Textur anzeigt, welche die Auflösung der Lightmap repräsentiert, siehe Abbildung 5.9.

Abbildung 5.9: Zeigt den Lightmap Density Viewmode in der Unreal Engine. Die Lightmap Auflösung der Assets steigt mit der Farbe von Blau über Grün bis Rot an. (Quelle: Eigener Screenshot, verwendeten Szene: https://denisgandra.gumroad.com/l/ERhck , Stand: 07. Mai 2022)

[8] https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LevelEditor/Viewports/ViewModes/ , Stand: 25. April 2022

Nach dem Einstellen der Lightmap Auflösungen wurden die Lightmass Einstellungen ermittelt. Hierfür wurden mehrere baking Vorgänge durchgeführt, bis ein geeignetes Ergebnis erreicht wurde. Dieser Prozess kann auf Kosten von Genauigkeit beschleunigt werden, indem die Qualität "Preview" oder "Medium" verwendet wird. Weiterhin wurde die Kompression der Lightmaps ausgeschaltet, da diese wie in 3.5 beschrieben sichtbare Kompressionsartefakte verursacht. Für die endgültigen Einstellungen wurde die Qualität auf Production umgestellt und die in Abbildung 5.10 gezeigten Einstellungen verwendet. Epic Games empfiehlt hierbei, dass "Static Lighting level Scale" und "Indirect Lighting Quality" miteinander multipliziert immer 1 ergeben[9], dies dient der Vermeidung von Artefakten. Damit Lightmass gezielter vorgehen kann wurde ein Lightmass Importance Volume[10] um die Geometrie der Szene platziert. Dies konzentriert die emittierten Photonen auf den Bereich innerhalb des Volumes, Bereiche außerhalb des Volumes erhalten nur einen Bounce mit niedrigerer Qualität.

Im Post Process Volume wurden Screen Space Reflections ausgeschaltet, da diese unschöne Artefakte verursachten. Die Reflexionen in der Szene wurden mit mehreren Reflection Captures realisiert. Diese wurden nach einem hierarchischen Prinzip[11] verteilt. Dabei werden Captures mit kleinerem Radius höher gewichtet als Captures mit größerem Radius. Für das Rendern der Light-Baking-Ergebnisse wurden in der Movie Render Queue 4 Temporal und 4 Spatial Samples eingestellt, diese Anzahl reicht aus, da nur Antialiasing benötigt wird und kein Rauschen entfernt werden muss.

Abbildung 5.10: Zeigt die Lightmass Einstellungen für die Renderings in 6.1a und 6.2a.

[9] https://www.youtube.com/watch?v=ihg4uirMcec , Stand: 25. April 2022

[10] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Lightmass/Basics/ , Stand: 25. April 2022

[11] https://www.youtube.com/watch?v=AQ2jIgqLHPA , Stand: 26. April 2022

5.3.2 Konfiguration des Raytracing Renderings

Für die Raytracing Umsetzung wurde das Lightmass importance volume entfernt, da es nur mit Lightmass kompatibel ist. Als Nächstes musste in den Projekteinstellungen Raytracing aktiviert werden und das Default RHI (Render Hardware Interface) auf DirectX12 umgestellt werden. Im Post Process Volume wurden die benötigten Raytracing Effekte (Globale Beleuchtung, Reflexionen, Transmission) eingestellt, um eine genauere Vorschau im Viewport zu sehen. Diese Effekte sind sehr Performance intensiv und können deswegen in Echtzeit nur mit wenigen Bounces und Samples verwendet werden. Wie in 2.2.3 beschrieben werden die Einstellungen, welche für das eigentliche Rendering verwendet werden, in der Movie Render Queue (MRQ) konfiguriert und vor jedem Render Beginn automatisch ausgeführt. Da in den Vergleichen keine Denoiser verwendet werden, wurden diese über Kommandos in der MRQ ausgeschaltet[12], so ist es möglich, sie im Viewport weiterhin zu nutzen.

Bei den Materialien wurde nur das Glas Material für Raytracing angepasst. Es folgt dem selben Aufbau wie das im Raytracing Vergleich in 4.1.3 gezeigte Glas Material. Es wurden 128 Temporal Samples verwendet, um Rauschen zu reduzieren und um gleichzeitig Antialiasing zu realisieren. Die in der MRQ verwendeten Konsolven Variablen[13] (CVars) sind in Abbildung 5.11 zu sehen. Weiterhin wurde die High Resolution Option der MRQ mit 8-facher Kachelung (64 Kacheln) und einer Überlappung von 0.1 verwendet. Die Kachelung war notwendig, da die 8 GB Grafikspeicher bei der hohen Sampleanzahl nicht ausreichten. Für das Full-HD Rendering wurde eine Kachelung von 4 verwendet (16 Kacheln).

Da auch in der MRQ nicht definiert werden kann, dass eine vorgegebene Zeit lang gerendert werden soll, mussten auch hier Einstellungen angepasst werden, um die Renderings für den zweiten Vergleich zu realisieren. Die genauen verwendeten Einstellungen sind im Anhang B sehen.

[12] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/RayTracing/MovieRenderQueue/ , Stand: 26. April 2022

[13] https://digilander.libero.it/ZioYuri78/ , Stand: 26. April 2022

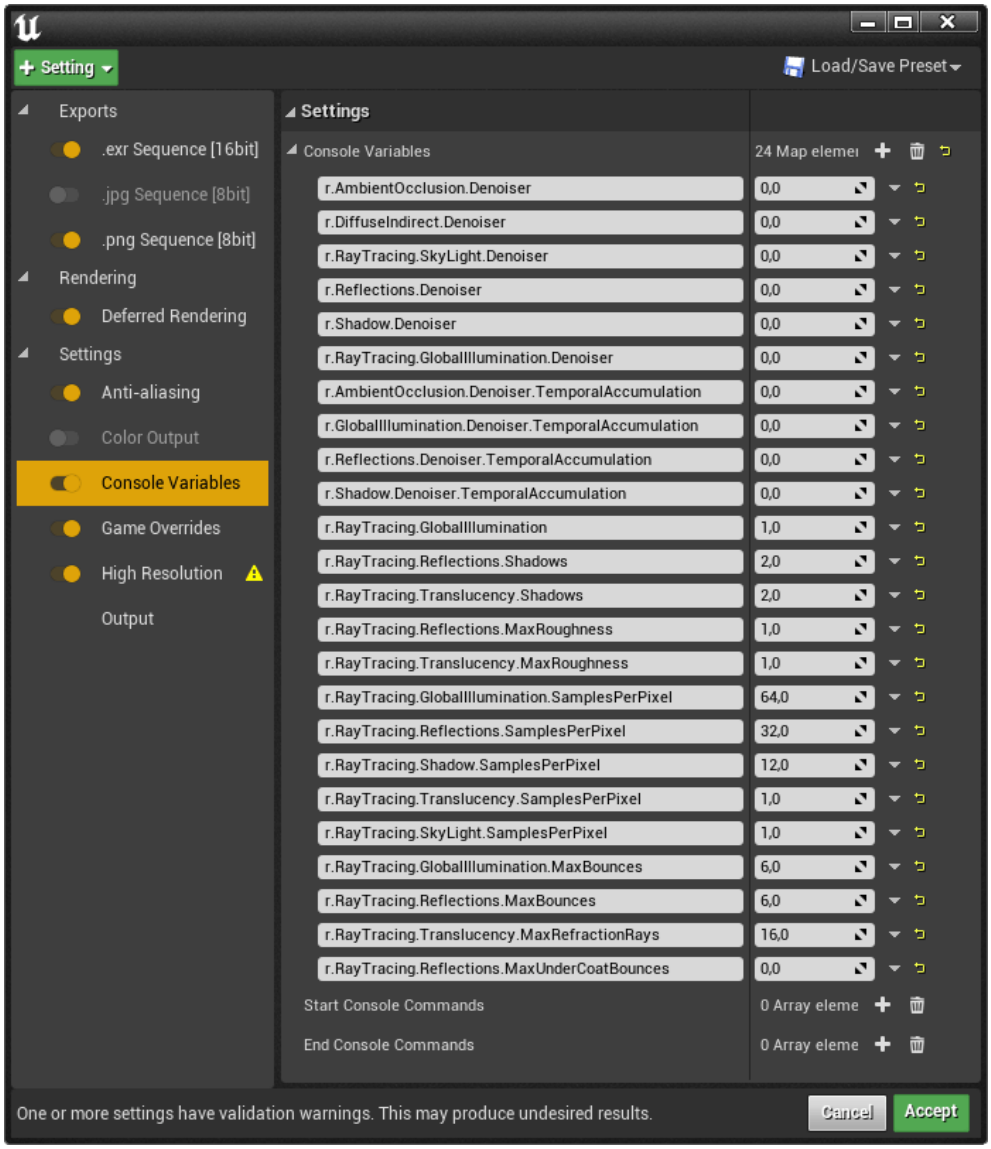


Abbildung 5.11: Zeigt die verwendeten CVars die für die Raytracing Renderings in 6.1b und 6.2b verwendet wurden.

5.3.3 Berechnung der Metriken

Für die Berechnung der Metriken wurde ein in Python geschriebenes Programm[14] verwendet. Das Programm kann zwei Bilder miteinander vergleichen und PSNR, MSE (Mean Squared Error) und SSIM der Bilder berechnen. In dem Programm wurde die SSIM Metrik so implementiert, dass eine Bewertung auf einer Skala von 0 bis 1 erfolgt.

[14] https://github.com/Addi90/SimpleImageComp , Stand: 03. April 2022

Kapitel 6

Ergebnisse

In diesem Kapitel werden die Ergebnisse aus Kapitel 5 präsentiert und ausgewertet. Dabei wird zuerst auf Vergleich 1 eingegangen, bei dem die verschiedenen Verfahren gegeneinander verglichen werden. Anschließend wird in Vergleich 2 gezeigt, wie Arnold und Unreals Raytracing bei gleicher vorgegebener Zeit abschneiden. Dazu wurden zusätzlich mit den in 4.4 beschriebenen Metriken die zeitlimitierten Bilder gegen ein fertig gerendertes Bild aus dem jeweiligen Renderer verglichen. Im Fazit wird anhand von Erkentnissen aus der Umsetzung auf Vor- und Nachteile der einzelnen Verfahren eingegangen, dabei wird auch der Erstellungsaufwand mit einbezogen.

Spezifikationen des Testsystems, welches für die Durchführung verwendet wurde:
CPU: Intel Core i7 12700K, GPU: RTX 3070 Ti 8GB, RAM: 48GB 4000MHz.
Unreal Engine Version 4.27.2, Arnold Version MtoA 5.1.1 (Core 7.1.1.0) in Maya 2022.3.
Im Vergleich zu den Tests in 4.1 wurde eine andere Grafikkarte und eine neuere Arnold Version verwendet. Die Arnold Version wurde aktualisiert wegen einer wichtigen Fehlerbehebung die Intel Alderlake Prozessoren betrifft[1] (Prozessor, der im Testsystem verwendet wird).

6.1 Auswertung von Vergleich 1

Auf den zwei folgenden Seiten werden die Renderings aus Vergleich 1 gezeigt. Anschließend erfolgt die Auswertung des Vergleichs.

[1] https://docs.arnoldrenderer.com/display/A5AFMUG/5.1.1 , Stand: 03. Mai 2022

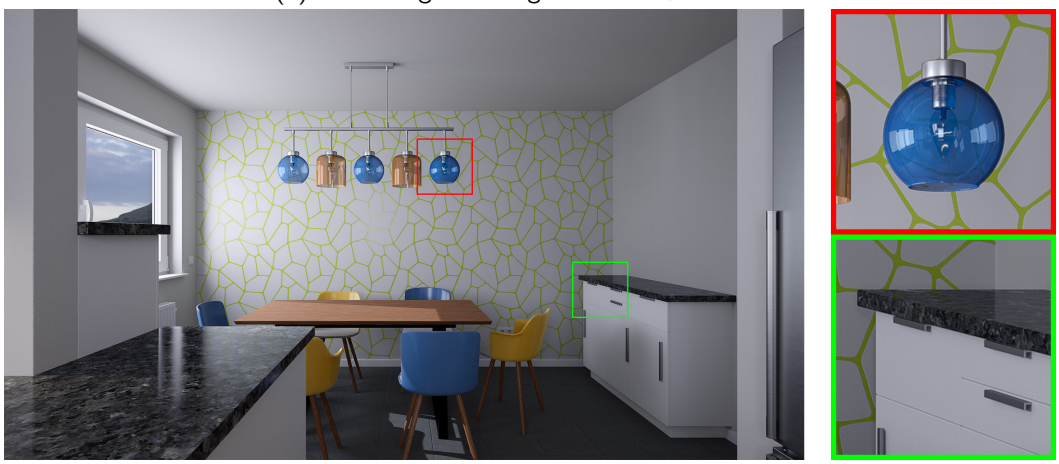
(a) Unreal Light Baking: 02:21:23 + 00:00:05

(b) Unreal Raytracing: 00:44:05

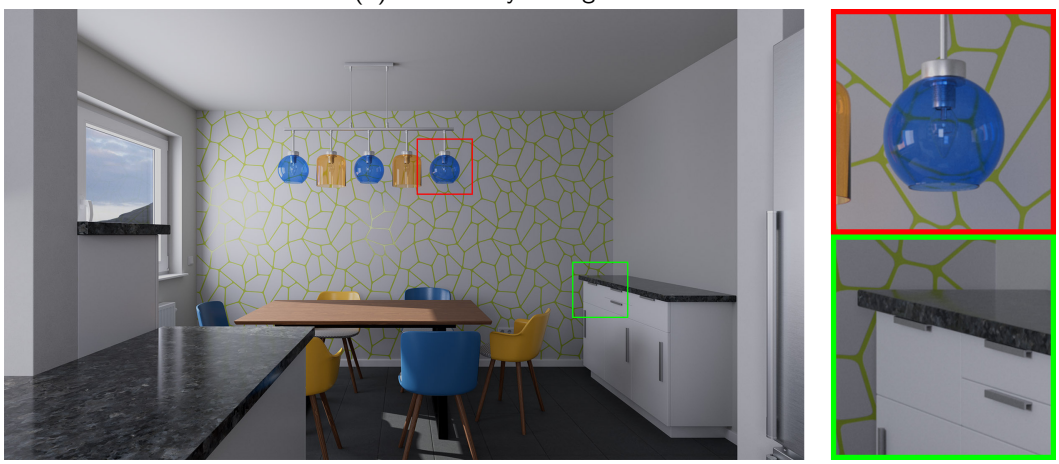
(c) Arnold Pathtracing: 10:13:03

Abbildung 6.1: Zeigt die fertigen Renderings aus der ersten Kameraperspektive. Rechts sind Bildausschnitte zu sehen in 400% Vergrößerung.

(a) Unreal Light Baking: 02:21:23 + 00:00:05



(b) Unreal Raytracing: 00:49:33



(c) Arnold Pathtracing: 11:07:30

Abbildung 6.2: Zeigt die fertigen Renderings aus der zweiten Kameraperspektive. Rechts sind Bildausschnitte zu sehen in 400% Vergrößerung.

Bild 1 (Abb. 6.1)	PSNR	SSIM	Bild 2 (Abb. 6.2)	PSNR	SSIM
Arnold - Raytracing	20.744 dB	0.838	Arnold - Raytracing	23.702 dB	0.820
Arnold - Lightbaking	17.474 dB	0.836	Arnold - Lightbaking	17.991 dB	0.860

Tabelle 6.1: Auswertung der Metriken für Vergleich 1

Bei Betrachtung der gemessenen Metrikwerte in Tabelle 6.1 ist zu sehen, dass bei den PSNR Werten ein größerer Abstand zwischen Raytracing und Lightbaking bei beiden Bildern besteht. Laut PSNR sind also die Raytracing Ergebnisse ähnlicher zu den Arnold Ergebnissen. Die ermittelten SSIM Werte sagen aus, dass die Unreal Renderings beide mit Werten von mehr als 0.8 Arnold sehr ähnlich sind. Bei Bild 2 übersteigt sogar der SSIM-Wert des Lightbaking Renderings den des Raytracing Renderings, obwohl PSNR dort eine größere Differenz ermittelt hat als bei Bild 1. Auch bei eigener Betrachtung sind größere Unterschiede in den Bildern zu sehen. Eine Ursache, warum das Lightbaking fast gleich eingestuft wird wie das Raytracing, könnte restliches Rauschen in den Raytracing Ergebnissen sein, da in den Lightbaking Ergebnissen kein Rauschen vorhanden ist[2]. Ein Beispiel, wie so etwas die SSIM Metrik verfälschen kann, ist in Abbildung 6.3 zu sehen.

Abbildung 6.3: Zeigt wie die SSIM Metrik ausgetrickst werden kann durch das Hinzufügen von Rauschen, wobei GT für Ground Truth steht. (Quelle: https://videoprocessing.ai/metrics/ways-of-cheating-on-popular-objective-metrics.html , Stand: 02. Mai 2022 , [SBR18])

[2] https://videoprocessing.ai/metrics/ways-of-cheating-on-popular-objective-metrics.html , Stand: 02. Mai 2022

Innenraum-Szenen sind für beide Renderer ein sehr anspruchsvolles Testszenario, da deutlich mehr Bounces und Samples für ein korrektes und rauschfreies Ergebnis benötigt werden. Das meiste Rauschen in der Szene wurde durch den indirekten diffusen Anteil verursacht. Die indirekte Beleuchtung wurde von beiden Unreal Verfahren gut umgesetzt, da sie visuell zu Arnold fast identisch aussieht. Dieses Ergebnis stimmt auch überein mit dem gesonderten Vergleich aus 4.1.1. Insgesamt betrachtet ist zu sehen, dass Unreals Raytracing Arnold am ähnlichsten ist. Die größten Unterschiede liegen hier bei den Reflexionen und der Darstellung von Glas.

In Abbildung 6.1b ist an der Kühlschranktür gut zu erkennen, dass Reflexionen dunkler erscheinen, da die globale Beleuchtung, wie bereits in 4.2 festgestellt wurde, in Reflexionen nicht mitberücksichtigt wird. Die Lightbaking Variante zeigt ihre größten Schwächen ebenfalls bei den Reflexionen. Durch die Reflecion Captures sind Biegungen der Reflexionen auf sehr spiegelden Objekten gut sichtbar. In 6.1a ist dieser Effekt gut zu sehen auf der Backofen-Front. Die Biegung ist bedingt durch die Positionierung und den Einflussradius der Reflection Captures. In 6.2a sind auf der linken Granitoberfläche auffällig falsche Reflexionen zu sehen, weiterhin spiegeln Bereiche an der Muster Tapete zu stark verglichen zu Arnold und Unreals Raytracing.

Wie in dem zweiten Rendering gut zu erkennen ist, unterscheidet sich die Darstellung des Glases der Deckenlampe stark je nach Verfahren. Die verwendeten Glas-Materialien für die beiden Unreal Verfahren basieren auf den in 4.5 beschriebenen Materialien. Das getönte Glas im Lightbaking Ergebnis wirkt wegen der Verwendung von Metallic zu hell und zu spiegelnd verglichen mit Arnold. Weiterhin kann bei genauer Betrachtung gesehen werden, dass der IOR nicht korrekt umgesetzt wird, was der Vergleich in 4.5 bestätigt. Im Raytracing Ergebnis (6.2b) kann gesehen werden, dass das Glas etwas undurchsichtiger erscheint als im Arnold Rendering. Verursacht wird das durch die Verwendung von einem Opacity Wert, der größer als 0 ist. Für die Tönung des Glases war dies jedoch notwendig, da ein Wert von 0 zu klarem Glas führt.

Das Lightbaking Ergebnis weist weichere Schatten auf als die anderen beiden Verfahren. Dies kann bei dem einfallenden Licht in der Mitte von Bild 6.1a und unter dem Tisch in Abbildung 6.2a gesehen werden. Raytracing zeigt ein genaueres Ergebnis mit definierteren Schatten, jedoch weichen diese leicht ab von den Arnold Ergebnissen, wo Schatten einen etwas weicheren Rand haben.

Interessante Ergebnisse zeigen sich bei der Betrachtung den benötigten Renderzeiten (Abbildung 6.4 und 6.5). Unreals Raytracing ist hierbei im Schnitt etwa 13-mal schneller als Arnold. Lightbaking benötigte zwar mehr Zeit als das Raytracing Ergebnis wegen der Bakingdauer, jedoch muss das Lightbaking nur einmal in der Szene durchgeführt werden. Somit hat Lightbaking einen großen Vorteil, sobald mehrere Renderings in derselben Szene erstellt werden, wie in Abbildung 6.5 zu sehen ist.

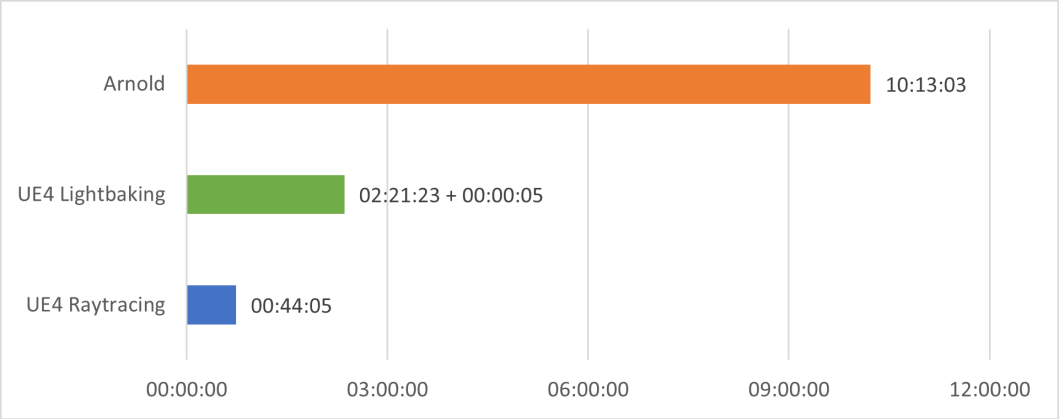


Abbildung 6.4: Balkendiagram der benötigten Renderzeit der jeweiligen Verfahren, für das erste Rendering in 6.1. Bei Lightbaking wird zusätzlich zu den 5 Sekunden Renderzeit die Bakingdauer mit aufgelistet.

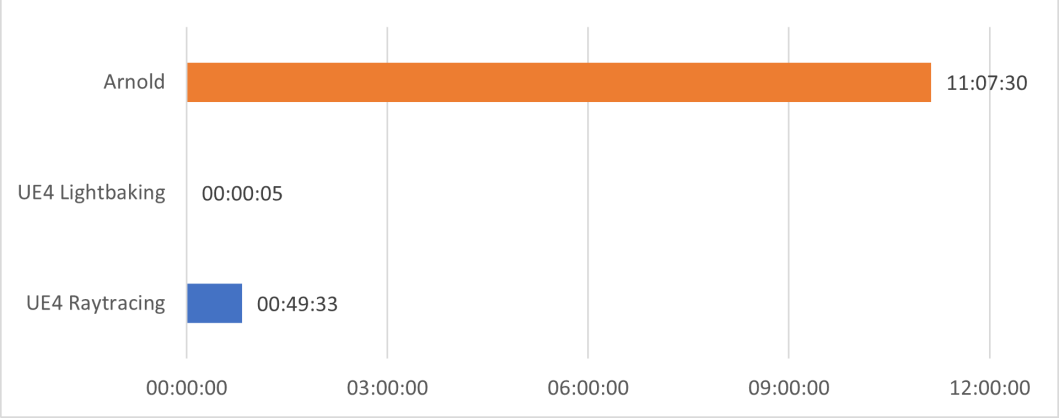


Abbildung 6.5: Balkendiagram der benötigten Renderzeit der jeweiligen Verfahren, für das zweite Rendering in 6.2. Bei diesem Bild fällt der Lightbaking Vorgang weg, weswegen hierbei nur die Renderzeit angegeben wird.

6.2 Auswertung von Vergleich 2

Für Vergleich 2 wurden für jedes Programm vier Renderings erzeugt, für die verschiedene Zeitspannen vorgegeben waren. So konnte beurteilt werden, was bei gleicher Zeitvorgabe erreicht werden kann. Gerendert wurde hierfür die Küchenansicht aus dem ersten Vergleich in 4K. Gezeigt wird hier in den Abbildungen 6.6 und 6.7 jedoch immer nur ein vergrößerter Ausschnitt, damit das Rauschen besser zu sehen ist. Bei diesem Vergleich zeigte sich, dass Unreals Raytracing bereits nach einer Minute höhere PSNR und SSIM Werte erzielt als Arnold nach 10 Minuten. Bei der Betrachtung von zwei zeitgleichen Ergebnissen ist schnell

zu erkennen, dass in den Arnold-Ergebnissen noch deutlich mehr Rauschen vorhanden ist, dies bestätigen auch die PSNR und SSIM Messwerte.

Gerundete Zeit	Genaue Zeit	PSNR	SSIM
1 min	00:00:59	29.956 dB	0.620
2 min	00:02:02	32.370 dB	0.717
5 min	00:04:56	35.114 dB	0.810
10 min	00:09:57	37.510 dB	0.874

Tabelle 6.2: Die Ergebnisse aus Vergleich 2 für Unreals Raytracing.

1 min 2 min 5 min 10 min Referenz

Abbildung 6.6: Zeigt die Unreal Raytracing Ergebnise aus dem zweiten Vergleich. Von einer Minute (Links) bis zur Referenz (rechts)

Gerundete Zeit	Genaue Zeit	PSNR	SSIM
1 min	00:01:00	18.031 dB	0.168
2 min	00:01:58	21.755 dB	0.281
5 min	00:04:36	25.233 dB	0.408
10 min	00:11:42	29.330 dB	0.587

Tabelle 6.3: Die Ergebnisse von Vergleich 2 für Arnold

1 min 2 min 5 min 10 min Referenz

Abbildung 6.7: Zeigt die Arnold Ergebnise aus dem zweiten Vergleich. Von einer Minute (Links) bis zur Referenz (rechts)

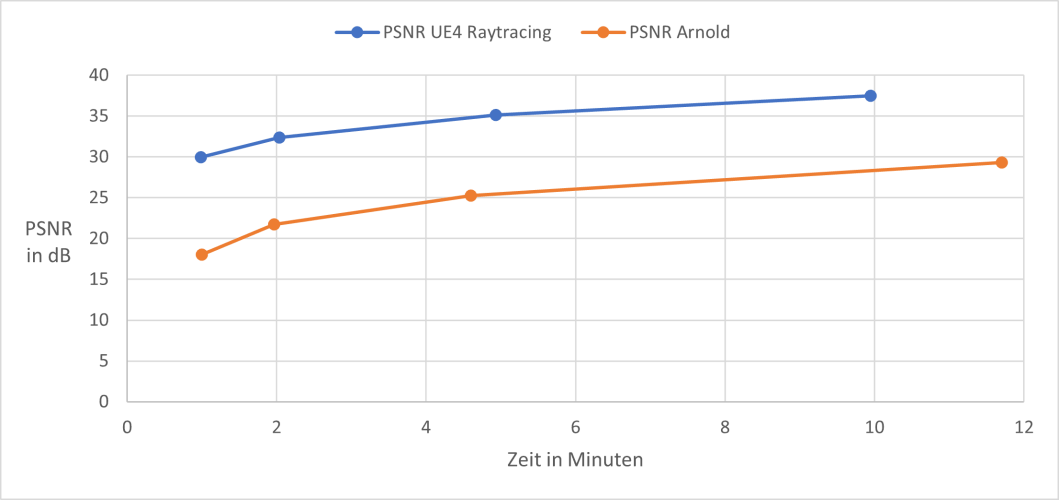


Abbildung 6.8: Zeigt die PSNR Werte von Arnold und Unreal im Vergleich.

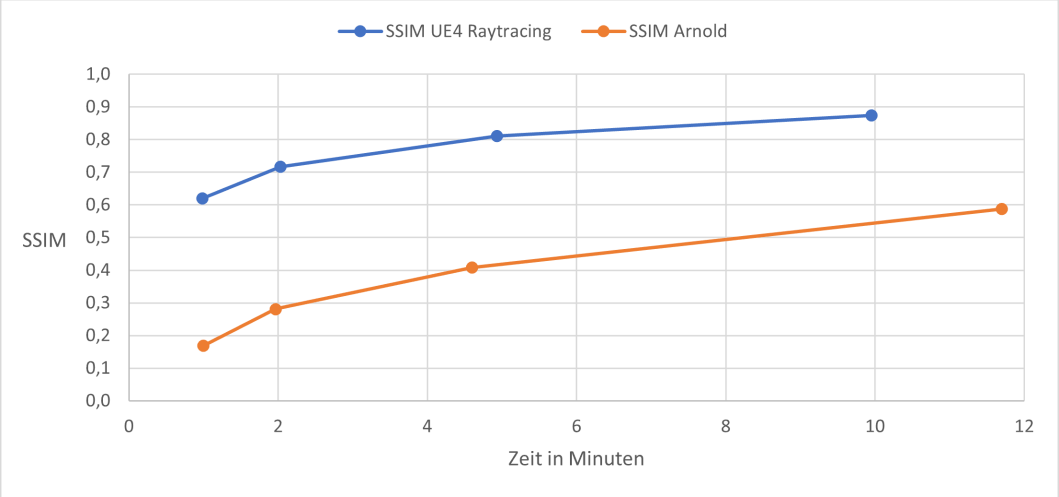


Abbildung 6.9: Zeigt die SSIM Werte von Arnold und Unreal im Vergleich.

Abbildung 6.9 zeigt, dass der SSIM Wert in der Zeitspanne der ungefähr betrachteten 10 Minuten bei Arnold deutlich schneller ansteigt als bei Unreal. Der Wert steigt bei Arnold um 0,419 an, während er bei Unreal um 0,254 ansteigt. Jedoch decken bei Unreal die betrachteten 10 Minuten bereits fast ein Viertel der Renderzeit ab, die für das Referenz Rendering benötigt wurden, weswegen dort im Schnitt deutlich höhere Werte ermittelt wurden. PSNR (Abbildung 6.8) zeigt ähnliches Verhalten. Bei beiden Metriken sind die größten Verbesserungen in den ersten Minuten zu sehen und ein Abflachen der Werte über längere Zeiten.

6.3 Eigene Beobachtungen bei der Umsetzung

Das Auswählen der Materialien erschien wesentlich einfacher in Unreal Engine, da in der Viewport Ansicht in Echtzeit mit realistischer Beleuchtung gearbeitet werden konnte. So können schneller und einfacher mehrere Varianten ausprobiert werden, was am Ende möglicherweise zu besseren Entscheidungen führt. Weiterhin kann durch das Materialinstanzensystem Zeit gespart werden, indem ein paar wenige Master Materialien angelegt und konfiguriert werden, aus denen dann immer wieder Instanzen erzeugt werden, sobald ein Material mit ähnlichen Eigenschaften benötigt wird. Zusätzlich können einfache Variationen eines Materials schnell umsetzt werden durch das Instanziieren von Instanzen. Weiterhin zeigte sich, dass der Polycount der Szene durchaus höher sein konnte als zuerst angenommen. Die zusätzliche Notwendigkeit der Lightmap UVs stellt extra Arbeitsaufwand dar, verglichen mit den anderen beiden Ansätzen, jedoch kann sich dieser lohnen, sobald mehrere Bilder in derselben Szene umgesetzt werden. Das Lightbaking Ergebnis ließ sich zudem interaktiv auf dem Testsystem mit 120 FPS (in 2560x1440) durchlaufen. Außerdem war es möglich, in wenigen Minuten deutlich höhere Auflösungen umzusetzen. Abbildung 6.10 zeigt ein 16K Rendering, welches in 00:01:15 entstanden ist. Hierbei ist zu erwähnen, dass die Beleuchtung nach wie vor mit derselben Lightmap Auflösung realisiert wird.

Abbildung 6.10: Zeigt ein 16K (15360 x 8640) großes Lightbaking Rendering. Rechts sind Bildausschnitte in 1600% Vergrößerung zu sehen.

Die Raytracing Implementierung ist sehr aufwendig für die Nutzung in Echtzeit, für flüssige Bildwiederholraten können nur sehr wenige Samples und Bounces benutzt werden. Dies ist auch der Grund, warum in vielen aktuellen Spielen Raytracing nur für bestimmte Effekte wie bspw. Reflexionen eingesetzt wird und nicht für alles wie in den Vergleichen in dieser Arbeit. Ein genaues Einstellen der Samples ist alleine über den Viewport nicht gut möglich, da die in der MRQ verfügbaren temporal und spatial Samples, welche auch das Rauschen reduzieren, im Viewport nicht zu sehen sind. Deswegen müssen, um genaue Einstellungen zu ermitteln, wie bei Arnold auch, Testbilder gerendert werden. Dabei ist zu erwähnen, dass in Unreal das Rendern eines Bild-Teilbereichs wie in Arnold nicht direkt möglich ist. Ein

weiteres hilfreiches Tool ist der in Unreal integrierte Pathtracer. Dieser ist aktuell noch als Betafeature gelistet, eignet sich aber jedoch gut um "Ground Truth" Renderings der Szene zu erstellen. Da der Pathtracer auf der Raytracing Implementierung aufbaut[3], kann er nahezu ohne extra Aufwand eingesetzt werden. Verglichen mit Unreal erschien jedoch die Bedienung von Arnold einfacher, da weniger Parameter bei den Rendereinstellungen konfiguriert werden müssen. Ein Grund hierfür ist, dass in Unreal mehrere Renderverfahren implementiert sind, die auch miteinander verwendet werden können. Weiterhin gibt es oftmals in der Engine unterschiedlich "teure" Lösungen (bspw. bei den Shadermodellen), die je nach Einsatzzweck verwendet werden können.

6.4 Fazit

Wie in 6.2 zu sehen ist, zeigt sich, dass Unreal früher zu relativ rauschfreien Ergebnissen kommt. Diese bieten eine gute Grundlage für die Verwendung von Denoisern, um so auf noch schnellerem Weg zu rauschfreien Bildern zu kommen. Dasselbe gilt auch für Arnold, jedoch wird dort mehr Zeit benötigt, um eine gut geeignete Grundlage für das Denoising zu erreichen.

Obwohl in den Vergleichen einige Unterschiede zwischen den Verfahren ermittelt wurden, sind die Ergebnisse aus Vergleich 6.1 dennoch gut verwendbar von ihrer visuellen Qualität her. Besonders Raytracing lieferte Ergebnisse, bei denen schwerer zu erkennen war, wo sich Fehler befinden, wenn nicht ein Arnold Bild im direkten Vergleich betrachtet wird. Weiterhin war es sinnvoll, zusätzlich zu den Vergleichen in 4.1, einen realistischeren Testfall zu betrachten, da viele der dort festgestellten negativen Eigenschaften unter extremen Bedingungen betrachtet wurden und in einer komplexeren Szene besser versteckt werden können. Zudem zeigten sich die Stärken und Schwächen der jeweiligen Verfahren. So konnte ermittelt werden, dass die Umsetzung der globalen Beleuchtung mit Lightbaking sehr gut möglich ist und dass Raytracing sich wesentlich besser eignet für Reflexionen und Transluzenz. Hierbei wäre es interessant, einen hybriden Ansatz zu betrachten, wie er in 3.9 beschrieben wurde. Zudem zeigte sich das Lightbaking sich sehr gut eignet für die Erstellung von den in 4.3.3 beschriebenen virtuellen Touren. Da dort oftmals Renderings von vielen Standpunkten in derselben Szene benötigt werden, wird sehr gut davon Gebrauch gemacht, dass die Beleuchtung für die gesamte Szene berechnet wurde. Weiterhin sind die Möglichkeiten der Stapelverarbeitung in der MRQ sehr gut umgesetzt. So können beispielsweise mehrere Renderings mit verschiedenen Einstellungen und Kameras abgearbeitet werden und sogar mit den exakten Rendereinstellungen als Render Queue gespeichert werden. Diese Funktion eignet sich beispielsweise sehr gut, um mehrere Standpunkte einer virtuellen Tour automatisiert zu rendern. Zusätzlich besteht die Option, ein solches Projekt nach Bedarf möglicherweise aufzuwerten zu einer interaktiven Echtzeit Anwendung. Dies erfordert zwar weiteren Arbeitsaufwand, jedoch befindet man sich bereits mit dem Projekt in einem Programm, welches eine solche Umsetzung ermöglichen kann.

[3] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/RayTracing/PathTracer/ , Stand: 03. Mai 2022

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Ziel dieser Arbeit war es zu ermitteln, ob es möglich ist, die Unreal Engine für das Rendern von hochaufgelösten Bildern einzusetzen, um so Renderzeiten zu verkürzen gegenüber offline Renderern wie Autodesk Arnold. Um diese Frage zu beantworten, musste zuerst überprüft werden, wie es Unreal Engine überhaupt möglich ist, schneller Bilder zu generieren als ein offline Renderer, bei scheinbar ausreichender visueller Qualität. Dazu wurden zuerst grundlegende Prinzipien und Systeme wie Lightmaps und Unreals Movie Render Queue betrachtet. Zusätzlich wurde auf Renderverfahren wie Raytracing, Photon mapping und Pathtracing eingegangen.

Mithilfe der Grundlagen konnte nun betrachtet werden, was genau die implementierten Verfahren machen, um Zeit einzusparen. Da bei Raytracing basierten Verfahren die Schnittpunktberechnung der Strahlen mit der Geometrie die meiste Rechenzeit ausmacht, wird möglichst dort optimiert. Dazu wurden unteranderem BVHs betrachtet. BVHs werden von Unreal und Arnold genutzt, jedoch kann Unreal hierbei Gebrauch machen von Grafikkarten, die über spezielle Raytracing Hardware verfügen. Diese spezielle Hardware unterscheidet sich je nach Hersteller, aber grundlegend versuchen sie alle die BVH Initialisierung und Traversierung zu beschleunigen. Als Nächstes wurde betrachtet, wie Unreal ermöglicht außerhalb von Raytracing Reflexionen umzusetzen. Zusätzlich wird darauf eingegangen, wie Echtzeit Raytracing in Unreal möglich gemacht wird. Dabei werden die Punkte: Schatten, Reflexionen und globale Beleuchtung getrennt genauer erklärt, jedoch haben sie alle gemeinsam, dass meistens mit nur sehr wenigen oder nur einem Sample gearbeitet wird. Wegen dieser Limitierung entsteht sehr viel Rauschen, welches entfernt wird mit Denoisern, die speziell angepasst sind für die einzelnen Problembereiche. Zuletzt wurde auf Arnold eingegangen und auf die Möglichkeit, die Raytracing Techniken in Unreal auf hybriden Wegen mit den Raster-Verfahren zu kombinieren.

Nachdem erklärt wurde, wie die genutzten Verfahren in der Theorie funktionieren, wurde betrachtet, welche sichtbaren Unterschiede sich in der Praxis ergeben. Dazu wurden

gesonderte Tests konzipiert, in denen indirekte Beleuchtung, Reflexionen, Transmission und Schatten in allen Verfahren umgesetzt und anschließend miteinander verglichen wurden. Bei der Auswertung dieser Tests zeigte sich, dass die Unterschiede zwischen Unreal und Arnold hauptsächlich bei den Reflexionen und der Transmission liegen. Der Reflexionstest zeige auf dass Reflection Captures und Screen space reflections starke Limitierungen gegenüber Raytracing und Pathtracing haben und dass bei Raytracing die indirekte Beleuchtung der Szene in den Reflexionen nicht berücksichtigt wurde, weswegen die Reflexionen dunkler erscheinen als im Pathtracing Ergebnis.

Da in den Vergleichen nur extreme Bedingungen in gesonderten Tests umgesetzt wurden, erschien es sinnvoll, auch einen realistischeren Anwendungsfall zu betrachten. Hierzu wurden zuerst die Anforderungen an die 3D Modelle betrachtet, damit diese in der Unreal Engine mit beiden Verfahren (Raytracing und Lightbaking) verwendet werden können. Dazu ist auf die Notwendigkeit der Lightmap UVs eingegangen, so wie auf einen sinnvollen Export Prozess der Modelle in die Engine. Weiterhin wurde hier definiert, dass zwei unterschiedliche 4K-Ansichten für den ersten Vergleich und mehrere 4K-Bilder der ersten Ansicht für einen zweiten Vergleich in der Szene gerendert werden. Damit auch eine objektive Bewertung dieser Tests möglich ist, wird auf zwei Metriken (PSNR und SSIM) eingegangen. Diese werden genutzt, um zwei Bilder miteinander zu vergleichen und die Ähnlichkeit der Bilder numerisch auszudrücken.

Nachdem beschrieben wurde, was der Prototyp umsetzen soll, wird darauf eingegangen, wie er in der Praxis realisiert wurde. Hierbei wird zuerst der grundlegende Aufbau der erstellten Szene gezeigt. Anschließend wird auf, das Erstellen und Anpassen der 3D Modelle eingegangen, auf die Einstellungen der Lichtquellen und Kameras und auf die notwendigen Rendereinstellungen in den jeweiligen Systemen.

In der Vergleichsauswertung zeigte sich, dass Raytracing den Arnold Renderings am ähnlichsten ist. Jedoch sind die im vorherigen Vergleich festgestellten Unterschiede bei einem direkten Vergleich der Techniken weiterhin sichtbar. Auch die PSNR Metrik bestätigte die Annahme, dass das Raytracing Ergebnis ähnlicher zu dem Arnold Ergebnis ist. SSIM bewertete jedoch beide Unreal Bilder in etwa gleich. Möglicherweise könnte dies daran liegen, das restliches Rauschen in den Raytracing Ergebnissen die Metrik austricksen, da in den Lightbaking Ergebnissen kein Rauschen vorhanden ist. Insgesamt zeigte sich auch, dass die festgestellten Unterschiede in einer komplexen Szene wesentlich weniger auffallen als in den gesonderten Tests. Weiterhin wurden beachtliche Unterschiede bei den Renderzeiten ermittelt, so war Unreals Raytracing etwa 13-mal schneller als Arnold. Zudem zeigte sich hier der Vorteil von Lightbaking (der einmalige Bakingvorgang), da mehrere Bilder in derselben Szene erstellt wurden. In einem zweiten Vergleich wurde überprüft, welche Ergebnisse mit Unreals Raytracing und mit Arnold in derselben Zeit erreicht werden können. Dabei zeigte Unreal wesentlich bessere Ergebnisse, da Bilder früher rauschfreier wurden, was auch die PSNR und SSIM Messwerten abbilden. Insgesamt zeigen beide Vergleiche, dass die in Unreal gerenderten Bilder gut verwendbar sind, aber auch das im direkten Vergleich mit Arnold weiterhin sichtbare Unterschiede bestehen.

7.2 Ausblick

Wie bereits in dieser Arbeit in 6.4 beschrieben wurde, wäre die Betrachtung eines hybriden Ansatzes, bei dem Raytracing für Reflexionen und Transmission genutzt wird und Lightbaking für globale Beleuchtung interessant. Aufgrund der in dieser Arbeit ermittelten Ergebnisse ist abzunehmen, dass mit so einem Aufbau Ergebnisse erreicht werden können, welche den hier gezeigten Raytracing Ergebnissen visuell sehr ähnlich sind. Dieser Ansatz dürfte auch durch die Eigenschaften von Lightbaking Zeit einsparen, sobald mehrere Renderings in einer Szene erstellt werden. Eine weitere Möglichkeit wäre es, den hier beschriebenen hybriden Ansatz mit der GPU Implementierung von Lightmass durchzuführen, um so möglicherweise noch schneller zu Ergebnissen zu kommen.

Ein weiterer Aspekt, der aufbauend auf dieser Arbeit betrachtet werden kann, ist eine Überprüfung, ab wann ein Rendering ausreichend rauschfrei ist, damit ein Denoiser daraus ein qualitativ hochwertiges, verwendbares Ergebnis erzeugen kann. Interessant dabei wäre auch, ob so etwas anhand einer Metrik ausgedrückt werden kann.

Während diese Arbeit geschrieben wurde, hat Epic Games am 05.04.2022 die Unreal Engine 5 veröffentlicht[1]. Zwei sehr große Neuerungen von Unreal 5 sind ein neues Beleuchtungssystem namens Lumen und ein System, welches dynamisch das Mesh von Modellen in Abhängigkeit von der Betrachtungsdistanz anpasst namens Nanite. Durch Nanite ist es nicht mehr notwendig, den Polycount von Assets wegen Performance zu optimieren. Zusätzlich dazu ermöglicht Lumen interaktive globale Beleuchtung und Reflexionen in Echtzeit ohne Baking Vorgänge[2]. Weiterhin bietet Lumen die Möglichkeit von Software oder Hardware Raytracing an, so können auch Grafikkarten, welche keine spezielle Raytracing Hardware haben, davon Gebrauch machen[3]. Diese Features zeigen sehr viel Potenzial, um den Realismus einer Szene aufzuwerten, bereits in der Preview Phase von Unreal Engine 5 zeigten sich sehr interessante Umsetzungen mit diesen Techniken. Aufbauend auf dieser Arbeit wäre ein Vergleich mit den neuen Techniken Lumen und Nanite angebracht, um zu sehen, wie diese sich zu den hier beschriebenen Verfahren vergleichen.

Zusätzlich zu einem praktischen Vergleich der neuen Techniken sollten diese auch wissenschaftlich betrachtet werden, damit ersichtlich ist, wie die Techniken funktionieren und wie sie Features wie globale Beleuchtung in Echtzeit realisieren können.

[1] https://www.youtube.com/watch?v=7ZLibi6s_ew , Stand: 03. Mai 2022

[2] https://www.unrealengine.com/en-US/unreal-engine-5 , Stand: 03. Mai 2022

[3] https://docs.unrealengine.com/5.0/en-US/lumen-technical-details-in-unreal-engine/ , Stand: 03. Mai 2022

Anhang A

Quellenangaben externer 3D-Assets

Quellenangaben der Assets, die in den Vergleichen in 4.1 verwendet wurden

3D-Modelle

- Katze: https://3dsky.org/3dmodels/show/statuetka_wood_a_cat , Stand: 28. April 2022

Texturen

- Graue UV Textur exportiert aus der Software Blender: https://www.blender.org , Stand: 06. Mai 2022
- UV Karo Muster: https://polycount.com/discussion/186513/free-checker-pattern-texture , Stand: 28. April 2022

Quellenangaben der Assets die in der Küchenszene verwendet wurden
Bei jedem der folgenden Modelle wurden zusätzlich Anpassungen am Mesh und an den UVs vorgenommen.

3D-Modelle

- Induktionskochfeld: https://www.cgtrader.com/free-3d-models/household/kitchenware/miele-hobs-km6090 , Stand: 28. April 2022
- Deckenlampe: https://3dsky.org/3dmodels/show/om_podvesnoi_svetilnik_tk_lighting_3273_cubus , Stand: 28. April 2022
- Mikrowelle: https://3dsky.org/3dmodels/show/mikrovolnovaia_pech_bosch_serie_8_2 , Stand: 28. April 2022
- Backofen: https://3dsky.org/3dmodels/show/dukhovaia_pech_bosch_serie_8_1 , Stand: 28. April 2022
- Geschirrspüler: https://3dsky.org/3dmodels/show/ge_profile_dishwasher , Stand: 28. April 2022

- Küchenspüle: https://3dsky.org/3dmodels/show/moika_kukhonnaia_6 , Stand: 28. April 2022
- Dunstabzugshaube: https://3dsky.org/3dmodels/show/eleyus_into_52_750_is_2 , Stand: 28. April 2022
- Essgruppe: https://3dsky.org/3dmodels/show/4union_dining_set_004 , Stand: 28. April 2022
- Deckenspots: https://3dsky.org/3dmodels/show/om_gira_1 , Stand: 28. April 2022
- Heizkörper: https://3dsky.org/3dmodels/show/radiator_otopleniia , Stand: 28. April 2022
- Fenster: https://3dsky.org/3dmodels/show/okna_pvkh_s_podokonnikom_standartnye_2 , Stand: 28. April 2022

Materialien

- Granit: https://quixel.com/assets/ti0mbcnv , Stand: 28. April 2022
- Aluminium: https://quixel.com/assets/shkaaafc , Stand: 28. April 2022
- Tapete: https://quixel.com/assets/sj0jecrc , Stand: 28. April 2022
- Fliesen: https://quixel.com/assets/uktnccdg , Stand: 28. April 2022
- Gitter: https://ambientcg.com/view?id=MetalWalkway013 , Stand: 28. April 2022
- Holz, Stoff, Metal: https://www.unrealengine.com/marketplace/en-US/product/archvis-interior-rendering , Stand: 28. April 2022
- Muster Tapete: Selbst erstellt basierend auf Design von: https://www.amazon.de/Livingwalls-Vliestapete-Contzen-Designertapete-255273/dp/B008M5YL5A , Stand: 28. April 2022

Texturen

- Plasik Normalmap: https://www.filterforge.com/filters/9926-normal.html , Stand: 28. April 2022
- HDR-Panorama: https://polyhaven.com/a/champagne_castle_1 , Stand: 28. April 2022
- Backofen Front: https://gallerykitchens.b-cdn.net/wp-content/uploads/sites/23/2018/02/HRG6769S6B-bosch-series-8-oven-image-1.png , Stand: 28. April 2022

- Mikrowellen Front: https://www.hai-end.com/images/product_images/original_images/MCSA00776837_423883_BEL634GS1_def.png , Stand: 28. April 2022
- Bunte UV Textur exportiert aus der Software Blender: https://www.blender.org , Stand: 06. Mai 2022

Anhang B

Rendereinstellungen für Vergleich 2

Einstellungen für Arnold

Gerundete Zeit	Genaue Zeiten	Camera (AA)	Diffuse	Specular	Transmission	Lichter
1 min	00:01:00	1	1	1	1	1
2 min	00:01:58	1	2	1	1	2
5 min	00:04:36	1	3	2	1	2
10 min	00:11:42	2	2	2	1	2

Tabelle B.1: Die genauen verwendeten Einstellungen für die Arnold Ergebnisse aus 6.2

Einstellungen für Unreals Raytracing

Gerundete Zeit	Genaue Zeiten	Spatial	Temporal	GI	Reflection	Shadow	Lichter
1 min	00:01:00	1	8	21	11	6	8
2 min	00:01:58	1	10	40	16	6	8
5 min	00:04:36	1	16	58	28	10	8
10 min	00:11:42	2	32	58	28	10	8

Tabelle B.2: Die in Unreal verwendeten Einstellungen für die Ergebnisse aus 6.2. Global Illumination wurde hier mit GI abgekürzt.

Abkürzungsverzeichnis

SSR	Screen Space Reflections
MRQ	Movie Render Queue
GI	Global Illumination
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Memory
AABB	Axis aligned Bounding Boxes
OBB	Oriented Bounding Box
BVH	Bounding Volume Hierarchy
DXR	DirectX Raytracing
API	Application Programming Interface
G-Buffer	Geometry-Buffer
SSIM	Structural Similarity Index
PSNR	Peak-signal-to-noise-ratio
FOV	Field of View
HDR	High Dynamic Range
CVars	Console variables

Literaturverzeichnis

[AK90] Arvo, James ; Kirk, David: Particle transport and image synthesis. In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, 1990, S. 63–66

[App68] Appel, Arthur: Some techniques for shading machine renderings of solids. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, S. 37–45

[BJLY20] Baek, Kwang H. ; Ji, Yun ; Lee, Byung C. ; Yun, Tae S.: 3DCGI workflow proposal for reduce rendering time of drama VFX. In: *Journal of the Korea Institute of Information and Communication Engineering* 24 (2020), Nr. 8, S. 1006–1014

[CPC84] Cook, Robert L. ; Porter, Thomas ; Carpenter, Loren: Distributed ray tracing. In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 1984, S. 137–145

[CT82] Cook, Robert L. ; Torrance, Kenneth E.: A reflectance model for computer graphics. In: *ACM Transactions on Graphics (ToG)* 1 (1982), Nr. 1, S. 7–24

[DBB18] Dutre, Philip ; Bala, Kavita ; Bekaert, Philippe: *Advanced global illumination*. AK Peters/CRC Press, 2018

[FK+04] Fritsch, Dieter ; Kada, Martin u. a.: Visualisation using game engines. In: *Archiwum ISPRS* 35 (2004), S. B5

[GIF+18] Georgiev, Iliyan ; Ize, Thiago ; Farnsworth, Mike ; Montoya-Vozmediano, Ramón ; King, Alan ; Lommel, Brecht V. ; Jimenez, Angel ; Anson, Oscar ; Ogaki, Shinji ; Johnston, Eric u. a.: Arnold: A brute-force production path tracer. In: *ACM Transactions on Graphics (TOG)* 37 (2018), Nr. 3, S. 1–12

[GTGB84] Goral, Cindy M. ; Torrance, Kenneth E. ; Greenberg, Donald P. ; Battaile, Bennett: Modeling the interaction of light between diffuse surfaces. In: *ACM SIGGRAPH computer graphics* 18 (1984), Nr. 3, S. 213–222

[HB97] Hearn, Donald ; Baker, M P.: *Computer graphics, C version*. Pearson Education India, 1997. – 497–500 S.

[Hil20] Hilbig, Julius: *Vergleich und Evaluation von Real-Time-und Offline-Rendering*, Technische Hochschule Mittelhessen, Diplomarbeit, Dezember 2020

[Jen96] Jensen, Henrik W.: Global illumination using photon maps. In: *Eurographics workshop on Rendering techniques* Springer, 1996, S. 21–30

[KA13] Karras, Tero ; Aila, Timo: Fast parallel construction of high-quality bounding volume hierarchies. In: *Proceedings of the 5th High-Performance Graphics Conference*, 2013, S. 89–99

[Kaj86] Kajiya, James T.: The rendering equation. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, S. 143–150

[KG13] Karis, Brian ; Games, Epic: Real shading in unreal engine 4. In: *Proc. Physically Based Shading Theory Practice* 4 (2013), Nr. 3, S. 1. – https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf, Stand: 19. April 2022

[Lam60] Lambert, Johann H.: *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. sumptibus vidvae E. Klett, typis CP Detleffsen, 1760

[LLK+19] Liu, Edward ; Llamas, Ignacio ; Kelly, Patrick u. a.: Cinematic rendering in UE4 with real-time ray tracing and denoising. In: *Ray Tracing Gems*. Springer, 2019, S. 289–319

[Low14] Lowood, Henry: Game engines and game history. In: *History of Games International Conference Proceedings*, 2014, S. 2014

[MSW21] Marrs, Adam ; Shirley, Peter ; Wald, Ingo: *Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Springer Nature, 2021. – 791–821 S.

[PGSS06] Popov, Stefan ; Gunther, Johannes ; Seidel, Hans-Peter ; Slusallek, Philipp: Experiences with streaming construction of SAH KD-trees. In: *2006 IEEE Symposium on Interactive Ray Tracing* IEEE, 2006, S. 89–94

[SBR18] Sharif, Mahmood ; Bauer, Lujo ; Reiter, Michael K.: On the suitability of lp-norms for creating and preventing adversarial examples. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, S. 1605–1613

[Sch94] Schlick, Christophe: An inexpensive BRDF model for physically-based rendering. In: *Computer graphics forum* Bd. 13 Wiley Online Library, 1994, S. 233–246

[SG09] Sweeney, Tim ; Games, Epic: The end of the GPU roadmap. In: *Proceedings of the Conference on High Performance Graphics*, 2009, S. 45–52

[Ste17] Steiner, Peter: *Fotorealistische Visualisierung mittels Game Engines in der Architektur: Untersuchung der Möglichkeiten von Game Engines anhand einer Echtzeitvisualisierung der Schule am Kinkplatz*, Wien, Diplomarbeit, 2017

[Ves14] Vesterinen, Miro: 3D Game Environment in Unreal Engine 4. (2014)

[VG95] Veach, Eric ; Guibas, Leonidas J.: Optimally combining sampling techniques for Monte Carlo rendering. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 1995, S. 419–428

[WBSS04] Wang, Zhou ; Bovik, Alan C. ; Sheikh, Hamid R. ; Simoncelli, Eero P.: Image quality assessment: from error visibility to structural similarity. In: *IEEE transactions on image processing* 13 (2004), Nr. 4, S. 600–612

[Whi05] Whitted, Turner: An improved illumination model for shaded display. In: *ACM Siggraph 2005 Courses*. 2005, S. 4–es

[Wie12] Wiesenhütter, Daniel: *Effiziente Schattenberechnung in Szenen mit vielen Lichtquellen*, Hochschule für Angewandte Wissenschaften München, Diss., 2012

[YLS05] Yu, Tin-Tin ; Lowther, John ; Shene, Ching-Kuang: Photon mapping made easy. In: *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, 2005, S. 201–205