Vergleich und Evaluation von Real-Time- und Offline-Rendering

Studiengang Medieninformatik

Masterarbeit

vorgelegt von

Julius Hilbig
geb. in Lich

durchgeführt an der
Technischen Hochschule Mittelhessen, Friedberg

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

Friedberg, 2020

Für meinen Opa Martin Holighaus

Danksagung

An dieser Stelle möchte ich allen Personen danken, die beim Erstellen dieser Arbeit eine Hilfe waren.

Zunächst möchte ich meinem Referenten Prof. Dr. Cornelius Malerczyk und Korreferenten M.Sc. Hans Christian Arlt danken, die mich für diese Arbeit betreuten und überhaupt erst ein großes Interesse an der 3D-Grafik in mir weckten.

Meinen Eltern Anne-Charlott und Volker Hilbig möchte ich danken, dass sie mir das Studium überhaupt erst ermöglichten und mich dabei stets unterstützten.

Ebenfalls möchte ich meinen Kommilitoninnen Thamar Vainstain und Olga Zimmermann danken, mit denen ein reger Erfahrungsaustausch möglich war, da sie zeitgleich ihre eigene Masterarbeit schrieben. Aber auch für die stets gute Teamarbeit während des Masterstudiums möchte ich ihnen und auch Ben Wilde danken.

Weiterhin danke ich allen Freunden, Verwandten, Bekannten und auch Unbekannten, die an den beiden Evaluationen teilgenommen haben. Meinen Freunden möchte ich außerdem dafür danken, dass sie wenn stets für Ablenkung vom Schreiben der Arbeit sorgen konnten.

Und ein besonderer Dank geht an meine, um eines ihrer Korrekturkommentare zu zitieren, „tollen, supergenialen, allerbesten Schwester“ Johanna Marek, die sich die Zeit nahm diese Arbeit Korrektur zu lesen, sodass andere Personen hoffentlich weniger Probleme beim Lesen haben werden.

Selbstständigkeitserklärung

Ich erkläre, dass ich die eingereichte Masterarbeit 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, November 2020

Julius Hilbig

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Diese Arbeit beschäftigt sich mit dem aktuellen Stand (November 2020) der Real-Time-Rendering-Technik und vergleicht diese mit dem Offline-Rendering. Der Vergleich geschieht durch eine Beispielanwendung und zwei Evaluationen. Abschließend soll aufgezeigt werden, ob und für welche Anwendungszwecke Real-Time-Rendering das Offline-Rendering ersetzen kann.

1.1 Motivation

Die visuelle Qualität von Videospielen ist in den letzten Jahren stetig gestiegen. Dieser Fortschritt scheint kontinuierlich weiter zu gehen, wie zum Beispiel Bilder des *Unreal Engine 5* Demovideos zeigen (Siehe Abbildung 1.1) [Epi20]. Videospiele mit derartiger grafischer Qualität sind kaum mehr von Computer-Grafik Bild- oder Filmproduktionen zu unterscheiden. Diese verwenden jedoch meist fast ausschließlich das Offline-Rendering, welches mit teils sehr hohen Rechenzeiten für ein einzelnes Bilder verbunden ist. Die Bilder für Videospiele oder andere Echtzeitanwendungen (zum Beispiel interaktive Auto-, Küchen-, etc. Konfiguratoren oder Simulatoren) hingegen müssen in wenigen Millisekunden berechnet werden, damit der Benutzer eine flüssige Bewegung wahrnehmen kann. Das menschliche Auge kann zehn bis zwölf Bilder pro Sekunde als einzelne Bilder wahrnehmen [RM00, S.24]. Bei mehr Bildern pro Sekunde, fängt das Gehirn an eine Bewegung wahrzunehmen. Für Filme wurde eine Bildfrequenz von 24 Hertz mit der Einführung von Tonfilmen standardisiert, mit dieser Wiederholrate ist ein Bild für ca. 41 Millisekunden zu sehen und genauso viel Zeit bleibt bei Real-Time-Renderings, um das nächste Bild zu berechnen. Für Computeranwendungen sind höhere Wiederholraten typisch, die der Wiederholrate des Monitors entsprechen. Diese liegt meistens bei 60 Hertz, was 16,7 Millisekunden Renderzeit für ein Bild entspricht. Für Videospiele werden teilweise auch noch höhere Bildwiederholraten bevorzugt, wie zum Beispiel 120 Hertz (8,3 Millisekunden), 144 Hertz (6,9 Millisekunden) und 240 Hertz (4,2 Millisekunden). Welche deutlich weniger Zeit zur Bildberechnung zulassen und viele aktuelle Videospiele, auch mit sehr guter visueller Qualität, können diese Wiederholraten mit entsprechender Hardware erreichen.

Solch schnelle Berechnungszeiten legen es nahe, nur noch Real-Time-Renderings statt

Abbildung 1.1: Ein Screenshot aus dem Unreal Engine 5 Demovideo von *Epic Games*.
Quelle: https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5 – Epic Games 2020

Offline-Renderings für Bild- und Videoproduktionen zu verwenden. Schließlich können Offline-Renderings mehrere Stunden oder sogar Tage pro Bild benötigen, diese Zeit könnte mit Real-Time-Renderings eingespart werden. Real-Time-Renderings kommen jedoch mit einem Qualitätsverlust einher, denn nicht alle Bildberechnungen können in Echtzeit realistisch durchgeführt werden. Doch dieser Qualitätsunterschied scheint stetig kleiner zu werden. Real-Time-Renderings wurden bereits für einige Film- und Serienproduktionen verwendet, so wurde zum Beispiel die *Unreal Engine 4*[1] für *Westworld*[2], *Solo: A Star Wars Story*[3] und *The Manadalorian*[4] verwendet. Durch die Verwendung einer Echtzeit-Engine können Spezialeffekte und/oder virtuelle Hintergründe direkt am Set sichtbar gemacht werden, wie in Abbildung 1.2 zu sehen ist.

Trotzdem führen die Unterschiede zwischen Real-Time- und Offline-Rendering dazu, dass selbst bei der Entwicklung von Echtzeitanwendungen die Nachfrage besteht, Offline-Renderings von den Szenen der Anwendung zu erstellen, wie die Existenz des *V-Ray for Unreal* Plugins zeigt[5].

[1] https://www.unrealengine.com/en-US/

[2] https://www.unrealengine.com/en-US/spotlights/hbo-s-westworld-turns-to-unreal-engine-for-in-camera-visual-effects – Zuletzt geprüft am 10.09.20

[3] https://www.unrealengine.com/en-US/spotlights/unreal-engine-powers-ilm-s-vr-virtual-production-toolset-on-solo-a-star-wars-story – Zuletzt geprüft am 10.09.20

[4] https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian – Zuletzt geprüft am 10.09.20

[5] https://www.chaosgroup.com/vray/unreal

Abbildung 1.2: Ein Bild vom Set der Fernsehserie *The Mandalorian* auf dem der mit der *Unreal Engine 4* gerenderte virtuelle Hintergrund zu sehen ist.
Quelle: https://www.unrealengine.com/en-US/blog/forging-new-paths-for-filmmakers-on-the-mandalorian – Epic Games 2020

1.2 Problemstellung

Vor dem Start einer Computer-Grafik-Produktion stellt sich aktuell die Frage, welche Art von Rendering verwendet werden sollte. Eine gute Antwort auf diese Frage zu finden ist jedoch schwer. Existierende Gegenüberstellungen sind entweder veraltet oder nicht wissenschaftlich belegt. Außerdem beschäftigen sie sich oft nur mit der resultierenden Optik und selten mit dem Produktionsprozess, zeigen selten die Grenzen der jeweiligen Technik auf, oder eine Kombination dieser Punkte. Zusätzlich wird nicht behandelt, wie die Verwendung von Real-Time-Engines bei der Entwicklung von Offline-Renderings von Vorteil sein kann.

Zusätzlich stellen sich für Produzenten auch diverse Fragen im Bezug auf benötigte Hardware oder Software, sowie zu den Erfahrungen der Mitarbeiter und wie gut diese sich zwischen den Rendertechniken transferieren lassen.

Darüber hinaus ist unklar, ob und wie eine Kombination von Echtzeit-3D-Anwendungen und Offline-Renderings abseits der reinen Bild- und Video-Produktion verwendet werden kann. Zum Beispiel könnten bei einer 3D-Anwendung die etwas in Echtzeit interaktiv darstellt zusätzlich realistischere Renderings benötigt werden.

Letztendlich stellt sich noch die Frage, ob die optischen Unterschiede für Betrachter (Kunden, Investoren, etc.) überhaupt wahrnehmbar oder relevant sind.

1.3 Zielsetzung

In dieser Masterarbeit sollen Real-Time- und Offline-Rendering wissenschaftlich verglichen und evaluiert werden. Das Vorgehen dafür sieht wie folgt aus:

Mehrere 3D-Modelle werden mit verschieden Workflows für Real-Time- und Offline-Renderings geshadet, gerendert und in eine Echtzeit-3D-Anwendung integriert. Dadurch sollen Unterschiede und Gemeinsamkeiten zwischen den beiden Rendertechniken aufgezeigt und analysiert werden. Als Anwendung dient ein zuvor entwickelter Auto-Konfigurator, der im Kapitel 4 näher behandelt wird. Ziel jedes Workflows ist es, ein Auto-Modell für Still-Rendering und für die Echtzeitanwendung zu shaden, dabei werden *Maya*[6] mit *V-Ray*, die *Unreal Engine* und das *V-Ray for Unreal* Plugin verwendet. Letzteres wird zum Import von Materialien von Offline-Renderings in die Unreal Engine und Offline-Renderings vom Editor der Unreal Engine aus verwendet. Die Tabelle 1.1 zeigt die durchgeführten Workflows anhand der Operationen, die in jedem Programm durchgeführt werden.

Tabelle 1.1: Durchzuführende Workflows

Workflow-Nummer	Maya mit V-Ray	Unreal Engine 4 mit V-Ray for Unreal	Unreal Engine 4 ohne V-Ray
1	V-Ray Shading und V-Ray-Offline-Rendering	-	Unreal Shading
2	V-Ray Shading	Import des V-Ray-geshadeten Modells, V-Ray-Offline-Rendering und Verwendung der V-Ray-Shader für Real-Time	-
3	-	V-Ray-Shading, V-Ray-Offline-Rendering und Verwendung der V-Ray-Shader für Real-Time	-
4	-	-	Unreal Shading und Real-Time-Rendering als Still

Zum Vergleich der visuellen Qualität von Real-Time- und Offline-Renderings wird eine Umfrage durchgeführt, in der mit beiden Techniken generierte Bilder und echte Fotos von den Befragten miteinander verglichen werden. Darüber hinaus sollen die Befragten Bilder nach

[6] https://www.autodesk.de/products/maya/overview

ihrem Realismusgrad und ihrem subjektiven Schönheitsgrad bewerten. Außerdem sollen die Umfrage-Teilnehmer angeben, woran sie glauben ein reales Bild von einem computergenerierten Bild unterscheiden zu können. Durch diese Evaluation soll herausgefunden werden, wie deutlich die Unterschiede zwischen den beiden Rendering-Techniken sind und welche Unterschiede oder sonstige Eigenarten von computergenerierten Bildern deutlich wahrgenommen werden.

Der bereits erwähnte Auto-Konfigurator dient außerdem als Anwendungsbeispiel, das sowohl Real-Time- als auch Offline-Rendering in einer Anwendung verwendet. Mit dem Konfigurator soll der Nutzen von Offline-Renderings abseits der Bild- und Filmproduktion durch eine Evaluation geprüft werden. Diese soll feststellen, ob zusätzlich zur Echtzeit-Darstellung eines konfigurierten Autos ein Offline-Rendering notwendig ist und wie lange auf ein solches gewartet werden kann.

1.4 Aufbau der Arbeit

Diese Arbeit teilt sich in zehn Kapitel auf. Das erste Kapitel Einleitung zeigt, warum der aktuelle Stand von Real-Time-Renderings im Vergleich mit Offline-Renderings im Rahmen dieser Arbeit behandelt werden soll und welche Ziele sich daraus ergeben.

Im zweiten Kapitel Stand der Technik werden aktuell für Offline- und Real-Time-Renderings verwendete Rendertechniken aufgezeigt. Dabei werden deren Funktionsweisen erläutert und mit Beispielbildern illustriert. Das dritte Kapitel Grundlagen behandelt danach die im Laufe dieser Arbeit verwendeten Teile der Unreal Engine und V-Ray in Maya oder der Unreal Engine. Zusätzlich werden im vierten Kapitel Auto-Konfigurator die Funktionsweise und Architektur des verwendeten Auto-Konfigurators behandelt, welcher vor Beginn dieser Arbeit entwickelt wurde.

Kapitel fünf Konzept und Entwicklung eines Systems zur Erstellung von Offline-Renderings aus einer Konfiguratoranwendung zeigt, welche Anpassungen an der Konfiguratoranwendung vorgenommen wurden, um mit ihr V-Ray-Renderings von Auto-Konfigurationen erstellen zu können. Im sechsten Kapitel Vergleich der Workflows zur Erstellung von Offline- und Real-Time-Renderings wird das Durchführen der vier Workflows behandelt, mit denen Modelle für die parallele Verwendung in Offline- und Real-Time-Renderings geshadet werden können. Dabei ist der Auto-Konfigurator die Real-Time-Anwendung, in der die Modelle verwendet werden sollen. Nach ihrer Durchführung werden die Workflows basierend auf ihren Einschränkungen, resultierenden Renderings und benötigter Zeit verglichen.

Das siebte Kapitel Analyse von Offline- und Real-Time-Renderings befasst sich zunächst mit der Erstellung von Offline-Renderings und Real-Time-Rendering mit und ohne Real-Time Ray Tracing von einem der zuvor geshadeten Autos. Diese Renderings werden anschließend verglichen und analysiert, um optische Unterschiede oder Gemeinsamkeiten der Rendertechniken aufzuzeigen.

Die Evaluation, in der Befragte Offline-Renderings und Real-Time-Renderings miteinander und mit Fotos vergleichen sollen, wird im achten Kapitel Evaluation zum Vergleich von Offline- und Real-Time-Renderings behandelt. Es wird zunächst der Aufbau der Evaluation dargestellt. Danach werden die Ergebnisse der Evaluation vorgestellt und ausgewertet.

Das neunte Kapitel Evaluation zum Auto-Konfigurator behandelt die Evaluation zur Auto-Konfigurator-Anwendung. Darin wird zunächst dargestellt, wie die Evaluation durchgeführt wurde und wie der Fragebogen aussah. Anschließend werden die Ergebnisse dieses Fragebogens dargestellt.

Das zehnte und letzte Kapitel Zusammenfassung und Ausblick fasst die gesamte Arbeit und ihre Ergebnisse zusammen. Außerdem wird ein Ausblick auf die zukünftigen Entwicklungen und Forschungen im Bereich des Real-Time-Renderings gegeben.

1.5 Zusammenfassung

In dieser Arbeit werden Offline- und Real-Time-Rendering miteinander verglichen. Zu Beginn wird für beide der Stand der Technik aufgezeigt. Dabei wird vor allem auf die Lichtberechnung eingegangen, welche sich zwischen beiden Techniken bisher stark unterschieden hat. Außerdem werden bereits existierende Gegenüberstellungen der beiden Techniken zusammengefasst. Anschließend wird auf die Grundlagen der verwendeten Softwarepakete eingegangen und dargestellt, welche der zuvor im Stand der Technik gezeigten Technologien diese einsetzen. An Software werden die *Unreal Engine 4*, *Maya*, *V-Ray for Maya* und *V-Ray for Unreal* verwendet.

Ein bereits entwickelter Auto-Konfigurator dient für diese Arbeit als Beispielanwendung. Dieser wird so erweitert, dass von der Anwendung aus Offline-Renderings der Auto-Konfiguration erstellt werden können. Dies soll einen direkten Vergleich der beiden Rendertechniken in einem Anwendungsbeispiel ermöglichen. Der Konfigurator erlaubt zusätzlich den Vergleich von vier Workflows zum Shaden eines Auto-Modells, das sowohl in Offline- als auch in Real-Time-Renderings verwendet werden soll. Die Workflows werden im Hinblick auf benötigte Zeit und Einschränkungen bei der Material-Erstellung verglichen.

Für eines der geshadeten Auto-Modelle werden ein Offline-Rendering, sowie Real-Time-Renderings mit und ohne Real-Time Ray Tracing erstellt. Diese Renderings werden verglichen, wobei sich zeigt, dass Schatten und Reflexionen ohne Ray Tracing in Real-Time-Renderings nicht unbedingt realistisch dargestellt werden können. Das Real-Time Ray Tracing kann diese Probleme jedoch beheben. Trotzdem zeigt sich, dass beide Real-Time-Renderings Probleme bei der Darstellung von transparenten Objekten haben.

Mit einer Evaluation werden Offline- und Real-Time-Renderings verglichen. Dabei zeigt sich, dass beiden Techniken für real gehalten werden können. Das Offline-Rendering wird im direkten Vergleich mit Real-Time-Renderings jedoch eher für real gehalten.

Eine Evaluation zum Auto-Konfigurator soll feststellen, ob eine Konfiguratoranwendung mit Real-Time-Renderings beim Autokauf gewünscht ist und ob zusätzliche Offline-Renderings sinnvoll sind. Sie zeigt, dass eine solche Anwendung zwar gern gesehen ist, aber nicht unbedingt notwendig. Das zusätzlich Offline-Rendering gibt zwar einen guten weiteren Eindruck der Konfiguration und auf es zu warten stört auch nicht, jedoch wird es trotzdem nicht für notwendig erachtet.

Kapitel 2

Stand der Technik

2.1 Einleitung

In diesem Kapitel wird gezeigt mit welchen Techniken Offline- und Real-Time-Renderings aktuell entstehen und wie diese grundlegende funktionieren. Dabei wird hauptsächlich das Global Illumination Problem behandelt, dessen Lösung sich zwischen den beiden Techniken aktuell am meisten unterscheidet.

2.2 Offline-Rendering

Das Offline-Rendering, auch Non-Real-Time-Rendering genannt, beschreibt die Berechnung von 3D-Bildern für nicht-interaktive Medien, wie zum Beispiel Bilder, Filme und Videos. Bei Offline-Renderings ist die konkrete Renderzeit eines Bildes nicht essenziell, da die Bilder erst nach Abschluss ihrer Berechnung, für das Zielmedium zusammengesetzt werden. Für einen Film mit 24 Bildern pro Sekunde werden also folglich 24 gerenderte Bilder pro Sekunde angezeigt. Bei Offline-Renderings kann ein Bild für wenige Sekunden oder auch mehrere Tage berechnet werden, abhängig davon wie schnell es benötigt wird und wie genau es berechnet werden soll.

Eingesetzt wird Offline-Rendering in Bild- und Filmproduktion, egal ob für Unterhaltungs-, Werbe- oder Lehr-Produktionen. Gängige Offline Renderer sind zum Beispiel *Arnold*[1], *OctaneRender*[2], *RenderMan*[3] und *V-Ray*[4]. Das Ziel der meisten Renderer ist es, ein möglichst fotorealistisches Ergebnis zu erzielen. Die grundlegenden Algorithmen zur Lichtberechnung, zu denen nachfolgend ein Überblick gegeben wird, sind bereits seit Jahren gegeben. In den letzten Jahren wurde bei der Renderer-Entwicklung vor allen an Optimierungen der Renderzeit und auch der Benutzerfreundlichkeit von bestehenden Techniken gearbeitet. So zum Beispiel Optimierungen am Path Tracing die in „The Path to Path-Traced Movies“ von Per

[1] https://www.arnoldrenderer.com/
[2] https://home.otoy.com/render/octane-render/
[3] https://renderman.pixar.com/
[4] https://www.chaosgroup.com/

H. Christensen und Wojciech Jarosz [CJ16] und dem Vortrag „The Path Tracing Revolution in the Movie Industry“ von Alexander Keller [Kel15] behandelt werden. Oder *Disney's* Versuch nur noch eine einzige Bidirektionale Reflektanzverteilungsfunktion (kurz BRDF) für alle Materialien zu verwenden, mit dem sich Brent Burley in „Physically-Based Shading at Disney“ beschäftigt [Bur12].

Auch ein Blick in die Technical Papers die auf der SIGGRAPH 2019 vorgestellt wurden zeigt dies[5]: Im Bereich „3. Light Science“ beschäftigen sich vier von fünf Papern mit der Optimierung von bestehenden Techniken, um sie effizienter zu machen. Lediglich das Paper „Optimal multiple importance sampling“ von Ivo Kondapaneni und weiteren [KVG+19] beschäftigt sich mit der Optimierung von „Monte Carlo estimators in computer graphics“ um diese zuverlässiger zu machen.

2.2.1 Global Illumination Problem

Zur Berechnung einer 3D-Szene gilt es vom Render-Algorithmus das Global Illumination Problem zu lösen. Globale Illumination beschreibt, wie sich Lichtstrahlen in einem Raum verteilen und setzt sich aus direkter Illumination und indirekter Illumination zusammen. Direkte Illumination beschreibt die direkte Beleuchtung von Objekten durch eine Lichtquelle. Indirekte Illumination hingegen beschreibt die Beleuchtung von Objekten durch Lichtstrahlen, die von einem Objekt reflektiert wurden oder durch ein Objekt transmittiert wurden. Beim Lösen des Global Illumination Problems, soll der verwendete Algorithmus in der Regel versuchen, realistische Beleuchtungsverhältnisse zu simulieren. Lichtstrahlen verfolgen in der Realität teils sehr komplexe Wege: Sie können, wenn sie auf Objekte treffen, reflektiert werden, durch Refraktion ein Objekt durchdringen, aber dabei abgelenkt werden, oder auch absorbiert werden. Im Falle von Reflektion und Refraktion kann sich die Farbe eines Lichtstrahls auch verändern. All das sollte ein Algorithmus zum Lösen des Global Illumination Problems simulieren. Vier gängige Techniken dafür sind **Ray Tracing**, **Path Tracing**, **Photon Mapping** und **Radiosity**. Diese werden nachfolgend näher beschrieben.

Ray Tracing

Beim Ray Tracing werden für jeden Pixel, den das finale Bilde haben soll, Lichtstrahlen in die Szene geschossen. Treffen die Lichtstrahlen auf ein Objekt, wird abhängig von dessen Eigenschaften die anzuzeigende Farbe bestimmt. Falls notwendig werden weitere Strahlen erzeugt, die von diesem Punkt aus in die Szene gehen. Mit diesen Lichtstrahlen wird die Beleuchtung rekursiv berechnet, dies wird als *rekursives Ray Tracing* beschrieben und wurde von Turner Whitted in „An improved illumination model for shaded display“ entwickelt [Whi80]. Die Anzahl an Lichtstrahlen, die pro Pixel in die Szene gesendet und an jedem Schnittpunkt von Strahlen und Objekten erzeugt werden, hat einen starken Einfluss darauf, wie schnell ein Bild mit Ray Tracing berechnet werden kann. Allerdings bedeutet eine höhere Strahlenanzahl, dass ein realistischeres Bild berechnet werden kann.

[5] https://s2019.siggraph.org/conference/programs-events/technical-papers/#program-content – Zuletzt geprüft am 16.09.2020

Abbildung 2.1: Ein Bild aus *Cars* mit Schatten und Reflexionen berechnet durch Ray Tracing.
Pixar 2006 Abbildung aus [CFLB06]

Ein Film der Ray Tracing früh umfassend benutzte war *Pixar's Cars* im Jahr 2005. Per H. Christensen und weitere beschrieben in „Ray Tracing for the Movie 'Cars'“, wie sie den Renderer *Renderman* um Ray Tracing erweiterten und für Cars benutzten [CFLB06]. Ray Tracing war für sie in Cars notwendig, da Reflexionen stark zur Optik eines Autos beitragen und die Berechnung von Reflexionen von Objekten, die selbst reflektierend sind, nur mit Ray Tracing möglich war. Ein weiteres Problem, das Christensen und seine Kollegen mit Ray Tracing lösten, waren Dateimanagement-Probleme die durch Shadow-Maps von großen Szenen ausgelöst wurden. Denn Ray Tracing benötigt keine Shadow-Maps zur Schattenberechnung. Abbildung 2.1 zeigt ein Bild aus Cars mit Schatten und Reflexionen, die durch Ray Tracing berechnet wurden.

Abbildung 2.2: Ein Bild aus *Findet Dory*, das mit Path Tracing berechnet wurde. *Pixar/Disney* 2016 Abbildung aus [CJ16]

Path Tracing

Path Tracing ist eine Art von Ray Tracing, die durch die Verwendung von weniger Strahlen schnellere Ergebnisse mit vergleichbarer Qualität erzielen kann. Per H. Christensen und Weitere beschreiben einfaches Path Tracing in „The Path to Path-Traced Movies“ als eine einfache rekursive Methode. Bei dieser Methode werden zur Farbberechnung eines Pixels eine bestimmte Anzahl von Strahlen vom Standpunkt des Auges durch diesen Pixel verfolgt. An jedem Schnittpunkt eines Strahls mit einem Objekt wird die direkte Beleuchtung des Punktes durch Lichtquellen – unter anderem mit Shadow Rays – bestimmt. Außerdem wird zur Berechnung der indirekten Beleuchtung ein weiterer Strahl vom Schnittpunkt aus in die Szene gesendet. Die Richtung dieses Strahls wird stochastisch, basierend auf den Oberflächeneigenschaften des Objektes gewählt. An Schnittpunkten generierte Strahlen werden genauso verfolgt, wie die ersten Strahlen und können wiederum neue Strahlen erzeugen. Die rekursive Verfolgung endet, wenn ein Strahl keine Oberfläche trifft, der Strahl basierend auf einer Wahrscheinlichkeit endete (zum Beispiel absorbiert wurde) oder die maximale Rekursionstiefe erreicht wurde [CJ16, S.111ff]. Diese Form des Path Tracing geht zurück auf James T. Kajiya, der sie 1986 entwickelte [Kaj86]. Eine Variante des Path Tracing ist das bidrektionale Path Tracing, bei dem nicht nur vom Blickpunkt aus, sondern auch von Lichtquellen aus Strahlen in die Szene gesendet werden[LW93]. Path Tracing fand unter anderen Verwendung in dem Film *Findet Dory* (Abbildung 2.2)

Als Vorteile von Path Tracing nennen Christensen und Weitere folgende: Das Ergebnis ist vorhersehbar. Es ist einfach zu erlernen und gut unter Beachtung der Fristen bei CG und VFX Filmproduktionen zu benutzen ist. Es besitzt nicht zu viele Einstellungsmöglichkeiten, um ein gutes Ergebnis zu erzielen, sodass nicht zu viel experimentiert werden muss. Außerdem skaliert Path Tracing mit Multithreading. Durch progressives Rendering können beim Path Tracing weiterhin schnell erste Eindrücke vom Rendering gewonnen werden. Als

Abbildung 2.3: Nordlichter in *Frozen*, die mit Hilfe von Photon Mapping gerendert wurden. *Disney* 2016 Abbildung aus [CJ16]

Nachteile werden starkes Rauschen und großer Hauptspeicher-Bedarf genannt. Ersteres ist durch moderne Entrauschungstechniken jedoch weniger relevant geworden. Der Bedarf an Arbeitsspeicher steigt durch komplexere Szenen jedoch in vielen Fällen schneller, als die Menge an typischerweise verfügbaren Speicher [CJ16, S.152ff].

Photon Mapping

Das von Henrik W. Jensen entwickelte Photon Mapping erfolgt in zwei Schritten [Jen96]:

1. Die Photon Map wird erstellt, indem Photonen aus den Lichtquellen emittiert und in der Photon Map gespeichert werden, wenn sie auf eine Oberfläche treffen. Beim Treffen auf die Oberfläche, wird basierend auf deren Parametern zufällig bestimmt, ob das Photon reflektiert oder absorbiert wird.

2. Zum Rendern des Bildes wird ein Monte Carlo Ray Tracing benutzt, bei dem die indirekte Beleuchtung von Oberflächen durch die Photon Map bestimmt wird.

Verwendung findet Photon Mapping häufig in Filmen, um bestimmte durch Photonen ausgelöste Effekte zu rendern, wie Kaustiken oder Photonenstrahlen, zu denen Beispielsweise die Nordlichter gehören. Für letzteres wurde Photon Mapping in *Frozen* verwendet, was in Abbildung 2.3 zu sehen ist.

Abbildung 2.4: Global Illumination durch Radiosity in *Monsters University*. *Disney/Pixar* 2013 Abbildung aus [CHS+12]

Radiosity

Zum Rendern mit dem Radiosity-Verfahren wird die Szene in kleine Patches aufgeteilt. Jeder Patch kann Licht emittieren und reflektieren. Der Einfachheit halber wird davon ausgegangen, dass alle Oberflächen perfekt diffus reflektieren. Für jeden Patch wird berechnet, wie viel Energie er zusammen mit der reflektierten Energie aus der Szene insgesamt emittiert. Das Radiosity-Verfahren wird von Thomas Rauber in „Algorithmen der Computergraphik“ [Rau93, S.363-394] ausführlich erklärt. Das Verfahren basiert auf dem Energieerhaltungssatz laut dem, in einem abgeschlossenen System keine Energie verloren geht. Die Energie ist hierbei die Energie des Lichtes.

Das Radiosity-Verfahren eignet sich jedoch nur zur Berechnung von Licht in Umgebungen mit diffusen Objekten und kann dadurch meist nur ergänzend zu anderen Lichtberechnungstechniken verwendet werden. Zum Beispiel zum Cachen der Global Illumination in *Monsters University*[CHS+12] (Siehe Abbildung 2.4).

2.2.2 Lokale Beleuchtungsmodelle

Lokale Beleuchtungsmodelle dienen zur Farbbestimmung von Oberflächen, basierend auf der Beleuchtung der Oberfläche. In 3D-Programmen verwenden Shader heutzutage meist mehrere lokale Beleuchtungsmodelle zusammen, um eine Oberfläche akkurat beschreiben zu können. Diese Oberflächenbeschreibungen werden, wie zu vor bereits erwähnt, von einigen globalen Beleuchtungsmodellen verwendet, um zu entscheiden, wie sich ein Lichtstrahl verhält, wenn er auf eine Oberfläche trifft. Folgend werden kurz einige der typischen lokalen Beleuchtungsmodelle erklärt. Weitere, nachfolgend nicht erklärte lokale Beleuchtungsmodelle sind Oren-Nayar und Minnaert für diffuse Reflexionen, Ward für spiegelnde Reflexionen, Hanrahan-Krueger für Subsurface Scattering, sowie Cel Shading und Gooch Shading für

nicht-photorealistische Anwendungsfälle. Jedes Beleuchtungsmodell hat eine bidirektionale Reflektanzverteilungsfunktion (kurz BRDF), die verwendet wird, um das reflektierte Licht beziehungsweise dessen Intensität an einem Punkt einer Oberfläche zu berechnen.

Lambert

Das Lambert-Beleuchtungsmodell beschreibt komplett matte beziehungsweise diffus reflektierende Oberflächen. Nach dem Lambertschen Gesetz ist die Helligkeit einer diffusen Oberfläche unabhängig vom Betrachtungswinkel, aber abhängig vom Winkel der Lichtquelle zur Oberfläche [Lam60]. Zusammen mit der Oberflächenfarbe wird die reflektierte Intensität wie folgt berechnet:

$$I_D = L \cdot NCI_L \tag{2.1}$$

In obiger Formel ist I_D die Intensität des diffus reflektierten Lichtes, L ein normalisierte Vektor, der von der Oberfläche zur Lichtquelle zeigt, N der Normalenvektor der Oberfläche, C die Farbe der Oberfläche und I_L die Intensität des eintreffenden Lichtes.

Phong

Beim Phong-Beleuchtungsmodell werden Grundreflektivität (engl. ambient reflectivity), diffuse Reflektivität (Lambertsche Reflektivität) und spiegelnde Reflektivität (engl. specular reflectivity) kombiniert, um die reflektierte Lichtintensität an jedem Punkt einer Oberfläche zu bestimmen [Pho75]. Zum Berechnen der Farbe eines Pixels ergibt sich die berechnete Intensität mit folgender Formel [Lyo93, S.2] (abgeändert um Abbildung 2.5 zu entsprechen):

$$I_{total} = C_a I_a + \sum\nolimits_i (C_d I_{ds_i}(N \cdot L_i) + C_s I_{ds_i}(R_i \cdot V)^n) \tag{2.2}$$

Dabei ist I_{ds_i} die Farbe beziehungsweise Intensität des i-ten Lichtes für diffus und spiegelnd und I_a die Farbe des Grundlichtes (engl. ambient lighting). C_a, C_d und C_s sind Farbvektoren, die die Reflektivität der Oberfläche für Grund-, diffuse und spiegelnde Beleuchtung beschreiben. N ist der Normalenvektor der Oberfläche, L_i ist ein Richtungsvektor, der von der Oberfläche zum i-ten Licht zeigt, R_i ist der Richtungsvektor, der durch die Reflexion des Licht-Richtungsvektors entsteht und V ist der Richtungsvektor von der Oberfläche zum Betrachtungspunktes. Die Vektoren sind in Abbildung 2.5 zu sehen. n ist die *Glanz*-Konstante der Oberfläche; je höher diese ist, desto stärker spiegelt die Oberfläche.

Blinn-Phong

Das Blinn-Phong-Beleuchtungsmodell ist eine Modifikation des Phong-Modells, bei der anstatt der Multiplikation $R \cdot V$ von reflektierten Lichtvektor R und Betrachtungsvektor V, ein *Halfway-Vektor* H berechnet wird, der auf halbem Weg zwischen V und dem Lichtvektor L liegt [Bli77]:

$$H = \frac{L + V}{||L + V||} \tag{2.3}$$

In Formel 2.2 kann $R_i \cdot V$ mit $N \cdot H_i$ ersetzt werden. Dadurch ist das Blinn-Phong-Modell weniger rechenintensiv, als das Phong-Modell.

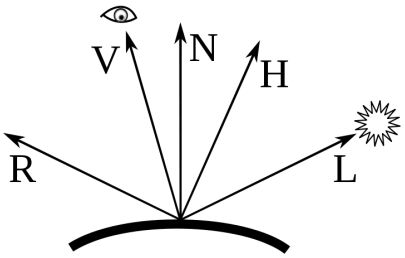


Abbildung 2.5: Die Vektoren zur Berechnung von Phong und Blinn-Phong Shading
Quelle: https://en.wikipedia.org/wiki/File:Blinn_Vectors.svg – Zuletzt geprüft: 17.09.2020

Cook-Torrance

Mit Hilfe des Cook-Torrance-Modells soll die Intensität von spiegelnden Glanzlichtern (engl. specular highlights) berechnet werden, die an Punkten einer Oberfläche entstehen, deren Normale dem *Halfway-Vektor* zwischen Betrachtungsvektor und Lichtvektor entspricht. Dies bedeutet, dass die Reflexion des Lichtes direkt in das Auge des Betrachters fällt. Dem zu Grunde liegt die Annahme, dass jede Oberfläche in *Microfacets* aufgeteilt ist. Dies sind sehr kleine Flächen die selbst perfekt spiegelnde Reflektoren sind. Wenn eine Oberfläche insgesamt sehr spiegelnd ist, sind die meisten Normalen dieser *Microfacets* mit der allgemeinen Normale in ihrer Umgebung identisch. Ist die Oberfläche jedoch rau, weichen die Normalen der meisten *Microfacets* ab und sind kaum einheitlich, wodurch das reflektierte Licht gestreut wird. Im Phong-Modell entspricht die Glanzlicht-Intensität k_{spec} dem folgenden Teil von Formel 2.2:

$$k_{spec} = (R \cdot V)^n \tag{2.4}$$

Und für Blinn-Phong dementsprechend:

$$k_{spec} = (N \cdot H)^n \tag{2.5}$$

Für das Cook-Torrance-Modell hingegen wird der spiegelnde Anteil und somit die Glanzlicht-Intensität wie folgt berechnet:

$$k_{spec} = \frac{FDG}{\pi(N \cdot L)(N \cdot V)} \tag{2.6}$$

Dabei ist F der Fresnel-Term, D ist die Funktion, die die Verteilung der Winkel der *Microfacets* beschreibt, und G ist der geometrische Dämpfungsterm (engl. geometric attenuation term), der den Schattenwurf durch die *Microfacets* wie folgt beschreibt:

$$G = min\left(1, \frac{2(H \cdot V)(V \cdot N)}{V \cdot H}, \frac{2(H \cdot N)(L \cdot N)}{V \cdot H}\right) \tag{2.7}$$

Für die Verteilungsfunktion D stehen verschiedene Funktionen zur Auswahl. Zunächst wurde meist die von Petr Beckmann und Andre Spizzichino entwickelte Verteilungsfunktion

verwendet [BS87]:

$$D = \frac{e^{-tan^2(\theta_h)/\alpha^2}}{\pi\alpha^2 cos^4(\theta_h)} \tag{2.8}$$

Dabei ist $\theta_h = arccos(N \cdot H)$ und α der quadratische Mittelwert der Steigung der Microfacets und somit der Roughness-Parameter der Oberfläche.

1975 entwickelten S. Trowbridge und K. P. Reitz die Trowbridge-Reitz- (D_{TR}) und generalized Trowbridge-Reitz-Verteilungen (D_{GTR}), welche sie einerseits mit bestehenden Verteilungen verglichen, aber auch an die Messdaten von rauem Glas anpassten [TR75]:

$$D_{TR} = c/(\alpha^2 cos^2\theta_h + sin^2\theta_h)^2 \tag{2.9}$$

$$D_{GTR} = c/(\alpha^2 cos^2\theta_h + sin^2\theta_h)^\gamma \tag{2.10}$$

c ist eine Skalierungskonstante und mit dem Exponent γ lässt sich anpassen, wie schnell die Funktion abfällt.

Shading

Das Shading von Oberflächen geschieht basierend auf ihren Normalen, wie bei den zuvor genannten Modellen gezeigt. Für das Shaden eines 3D-Modells kann entweder **Flat Shading** oder **Smooth Shading** verwendet werden.

Beim **Flat Shading** wird die Beleuchtung für jedes Polygon bestimmt. Als Punkt der Lichtbestimmung dient entweder der erste Vertex des Polygons oder das Zentrum beziehungsweise der geometrische Schwerpunkt des Polygons. Die Beleuchtung ist abhängig von der Normale des Polygons und wird für dessen gesamte Fläche verwendet. Dieses Shading führt zu scharfen Kanten zwischen Polygonen und spiegelnde Oberflächen werden nur schlecht dargestellt, da Glanzlichter nicht zwingend auf einem der berechneten Punkte liegen müssen.

Beim **Smooth Shading** wird die Farbe pro Pixel bestimmt, wodurch ein besserer Farbverlauf ohne scharfe Kanten an Polygon-Grenzen entsteht. Zwei typische Smooth Shading Arten sind **Gouraud Shading** und **Phong Shading**.

Für das **Gouraud Shading** wird die Beleuchtung pro Vertex mit dessen Normale berechnet. Um die Farbe eines Pixels, der in einem Polygon liegt, zu bestimmen, wird zwischen den Werten für die Vertices des Polygons interpoliert [Gou71].

Für das **Phong Shading** wird für jedes Pixel eines Polygons, eine Normale aus den Normalen der Vertices des Polygons interpoliert. Diese interpolierte Normale, wird für das Beleuchtungsmodell verwendet, um die Farbe des Pixels zu berechnen [Pho75]. Das beschriebene Vorgehen löst ein Problem des Gouraud Shadings: Glanzlichter die in der Mitte eines Polygons liegen, konnten nicht berechnet werden. Mit dem Phong Shading passiert dies nicht, da jedes Pixel eine eigene Normale verwendet.

2.3 Real-Time-Rendering

Beim Real-Time-Rendering (auch Echtzeit-Rendering) müssen Bilder, im Gegensatz zum Offline-Rendering, „sofort“ berechnet werden. Wenn ein Nutzer eine Eingabe tätigt, muss sich das Bild unmittelbar entsprechend verändern, ohne dass es für den Nutzer „ruckelt“. Das Real-Time-Rendering findet in Videospielen am meisten Verwendung, aber auch weitere Anwendungen benötigen 3D-Bilder in Echtzeit, wie zum Beispiel Simulatoren (Flugsimulator, etc.), Produkt-Konfiguratoren und -Demonstratoren oder Visualisierungen, die sonst für den Menschen nur schwer einsehbar wären, wie zum Beispiel 3D-Wettersimulationen oder im Inneren von menschlichen Körpern für die Medizin.

Eine typische Bildwiederholrate für Videospiele ist 60 Hertz, welche die meisten Monitore standardmäßig verwenden, wodurch für die Bildberechnung $\frac{1}{60}$ Sekunde also 16,67 Millisekunden verfügbar sind. Mittlerweile sind sogar noch höhere Bildwiederholraten, wie 90 Hertz, 120 Hertz, 144 Hertz oder 240 Hertz, verbreitet und bieten entsprechend noch weniger Zeit zur Bildberechnung. VR-Anwendungen haben zusätzlich noch die Herausforderung, dass zwei Bilder (eins pro Auge) gleichzeitig für meist 90 Hertz berechnet werden. Folglich bleibt für eine realistische Bildberechnung wenig Zeit und zusätzlich wird gute und spezialisierte Hardware benötigt. Während einige Offline-Renderer mit CPU rendern, andere mit der GPU und manche beides unterstützen, passiert die Bildberechnung beim Real-Time-Rendering in der Regel ausschließlich auf der GPU. Die CPU wird währenddessen für andere Aufgaben, wie Physik-Simulation oder sonstige Anpassungen der 3D-Szene verwendet. Damit die Bildberechnung in „Echtzeit“ möglich ist, also mit einer Framerate die eine flüssige Anzeige erlaubt, werden andere und effizientere Algorithmen als für Offline-Renderings benötigt und wenn möglich werden Berechnungen schon zuvor getätigt, sodass sie während des Renderprozesses verwendet werden können.

Für Real-Time-3D-Anwendungen werden typischerweise 3D Game Engines verwendet, die mehrere Systeme zusammenfassen, die für eine solche Anwendungen benötigt werden. Dazu gehören neben der Grafik-Engine zur Bildberechnung meist ein System zur Physik-Simulation, eine Sound-Engine, die Datenverwaltung, ein System für Benutzereingaben zur Steuerung, die Netzwerk-Komponenten und das Skripting, mit dem der Spielverlauf programmiert wird. Aktuelle 3D Game Engines sind unter anderen *Frostbite-Engine 2*[6], *CryEngine V*[7], *Unity*[8] und *Unreal Engine 4*[9].

2.3.1 Rasterung

Beim Prozess der Rasterung (engl. Rasterization) werden Vektorgrafiken in eine Rastergrafik (Pixelbild) umgewandelt. Zum Rastern einer 3D-Szene wird zunächst das Bild aus einer Perspektive auf eine Ebene projiziert, um das resultierende 2D-Bild anschließend zu rastern. Beim Rastern wird geprüft, ob ein Pixel in einem projizierten 3D-Objekt liegt. Wenn ja, erhält das Pixel die Farbe des Objektes (Siehe Abbildung 2.6).

[6] https://www.ea.com/frostbite
[7] https://www.cryengine.com/
[8] https://unity.com/
[9] https://www.unrealengine.com/

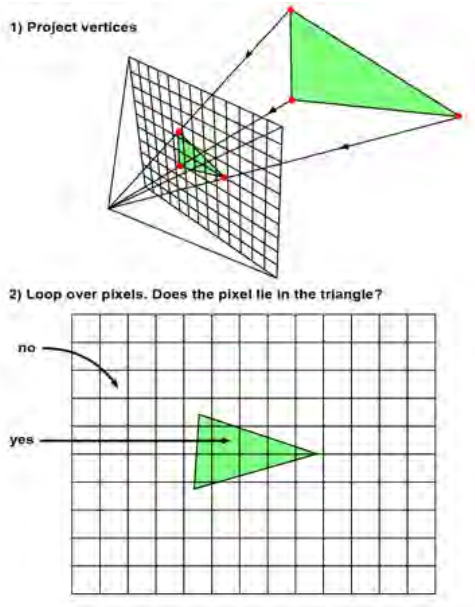


Abbildung 2.6: Die zwei Schritte der 3D-Rasterung: 1. Projektion der 3D-Objekte auf eine Ebene. 2. Überprüfen, ob ein Pixel in der Projektionen eines Objektes liegt. Wenn ja wird es entsprechend eingefärbt.
Quelle: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation – Zuletzt geprüft: 18.09.2020

Der Rasterungsprozess ist deutlich schneller, als ein vollständige Ray-Tracing und wird deswegen für die Echtzeit-Grafik verwendet. Die gängigen Programmierschnittstellen für 3D-Grafik *Direct3D*[10] und *OpenGL*[11], sowie dessen Nachfolger *Vulkan*[12], legen die folgenden „Triangle Rasterization Rules“ für die Rasterung fest:

- Der Mittelpunkt eines Pixels ist entscheidend: Ist dieser in einem Dreieck, ist das Pixel Teil des Dreiecks.

- Wenn ein Dreieck durch die Mitte eines Pixels verläuft, gilt die „top-left filling convention“. Das Pixel erhält die Farbe des Dreiecks, dessen obere (top edge) oder linke Kante (left edge) durch das Pixel geht. Abbildung 2.7 zeigt dies rechts: Zwei Dreiecke teilen sich eine Kante, die mittig durch die Pixel verläuft, die Pixel erhalten die Farbe des Dreiecks, dessen linke Kante in den Pixel liegt. Eine Kante gilt als obere Kante, wenn sie horizontal ist und über den anderen Kanten des Dreiecks liegt. Somit haben die meisten Dreiecke nur linke und rechte Kanten, welche nicht horizontal sein dürfen (siehe links in Abbildung 2.7).

[10] https://docs.microsoft.com/en-us/windows/win32/direct3d
[11] https://www.opengl.org/
[12] https://www.khronos.org/vulkan/

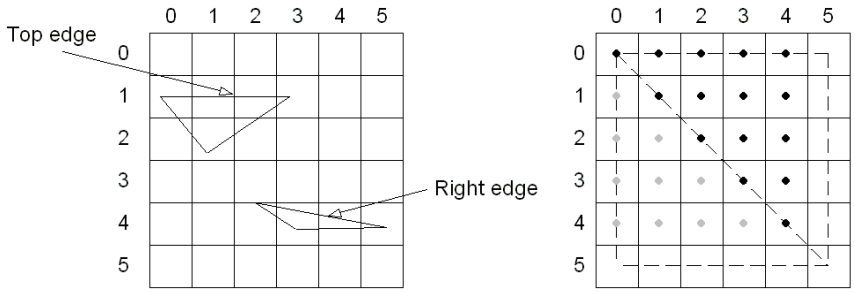


Abbildung 2.7: Links: Obere Kante und rechte Kante an zwei Dreiecken. Rechts: Zwei Dreiecke werden nach der Top-Left Convention gerastert.
Quelle: https://docs.microsoft.com/en-us/windows/win32/direct3d9/rasterization-rules – Zuletzt geprüft: 18.09.2020

Für die 3D-Rasterung werden stets Dreiecke verwendet, weswegen 3D-Modelle für Echtzeit-Renderings stets nur aus dreieckigen Polygonen bestehen. Falls nötig, konvertiert die Game Engine Modelle entsprechend.

Ein Problem der Rasterung ist, dass Kanten, die nicht senkrecht oder waagerecht verlaufen, einen Treppeneffekt im Pixelbild haben. Damit dieser verhindert wird, ist ein Anti-Aliasing von Nöten. Dazu gibt es unter anderen die folgenden Anti-Aliasing-Ansätze, welche in Abbildung 2.8 zu sehen sind:

- **FXAA (Fast approximate anti-aliasing):** Basierend auf der Luminanz des gerenderten Bildes werden an Kontrastreichen Stellen Kanten gesucht. Die Pixel-Positionen der Kante werden in eine Sub-Pixel-Verschiebung in einem 90°-Winkel zur Kante in beide Richtungen transformiert. Diese Sub-Pixel-Verschiebung wird auf das gerenderte Bild angewendet. Dies beschrieb Timothy Lottes in „FXAA“ [Lot09].

- **SSAA (Super sampling anti-aliasing):** Das Bild wird mit einer höheren Auflösung gerendert und dann zu einer kleineren Auflösung runter gerechnet, wodurch Kanten automatisch geglättet werden. Dies ist aufgrund der höheren Auflösung jedoch sehr rechenintensiv.

- **MSAA (Multi sampling anti-aliasing):** Ist eine angepasste Variante von SSAA bei der nur für die Pixel, in denen mehrere Dreiecke liegen, das Shading der Dreiecke für jedes Subpixel berechnet wird. Ist nur ein Dreieck in einem Pixel wird das selbe berechnete Shading für alle Subpixel verwendet.

- **TAA (Temporal anti-aliasing):** Hierbei werden Informationen aus dem aktuellen und zuvor gerenderten Bilder verwendet, um Kanten zu glätten. Außerdem können auch temporale Aliasing-Probleme verhindert werden, also das Springen von sich bewegenden Objekten, wenn die Bildwiederholrate zu langsam ist, um die Geschwindigkeit des Objektes flüssig darzustellen. Beim Temporalen Anti-Aliasing kann allerdings, bei niedrigeren Framerates, ungewollte Bewegungsunschärfe durch Verrechnung der einzelnen Bilder entstehen.

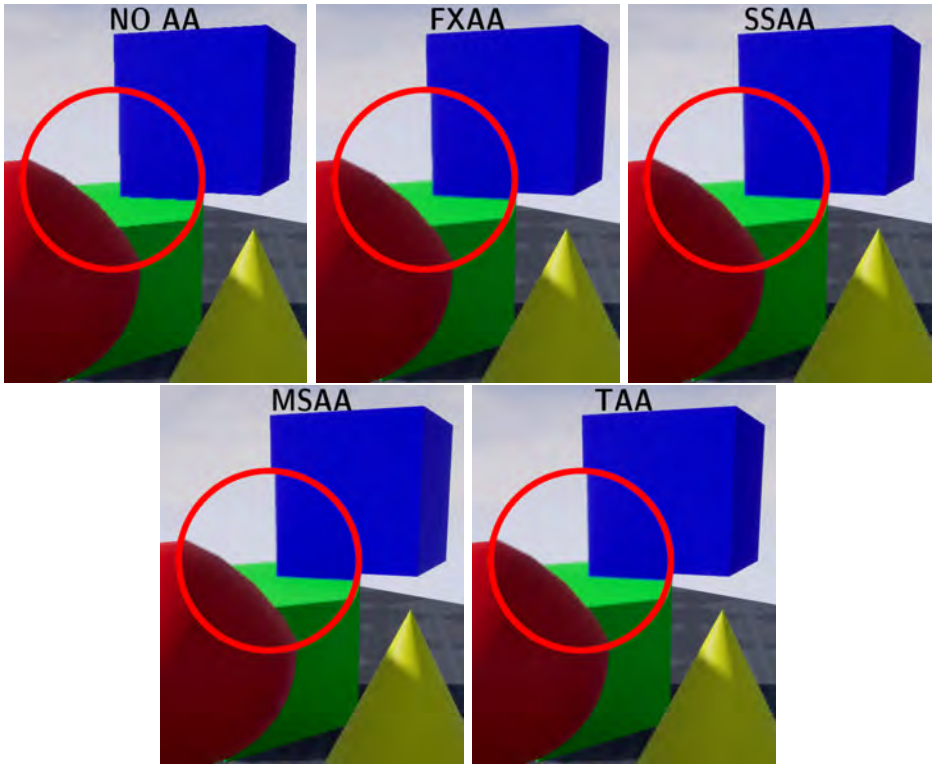


Abbildung 2.8: Das gleiche Bild, mit unterschiedlichem Anti-Aliasing und 2-facher Vergrößerung im roten Kreis. Von oben links nach unten rechts mit keinem Anti-Aliasing, FXAA, SSAA, MSAA und TAA.

2.3.2 Global Illumination in Echtzeit

Da bis vor wenigen Jahren Ray Tracing in Echtzeit noch nicht möglich war, werden zur globalen Beleuchtung in Echtzeit-3D-Anwendungen mehrere Techniken zur Lichtberechnung kombiniert.

Lightmap

Für möglichst realistische und akkurat berechnete Schatten, kann das Licht während des Entwicklungsprozesses vorberechnet, also „gebaket“ werden. Dazu wird die Lichtintensität auf Oberflächen berechnet und in zugehörigen Texturen, den Lightmaps, gespeichert. Diese Technik wurde erstmals für das Computerspiel *Quake* (1996) verwendet [Abr00]. Lightmaps können mit unterschiedlichen Offline-Rendering-Techniken erstellt werden, deren Berechnungen in Textur-Form abspeicherbar sind. Zum Beispiel verwendet *Valve's*[13] *Source Engine* eine Variante von *Radiosity* namens *Radiosity Normal Mapping*, bei der zusätzlich die Richtung des einfallenden Lichtes gespeichert wird [MMG06]. Das für die *Unreal Engine 4* verwendete

[13] https://www.valvesoftware.com/

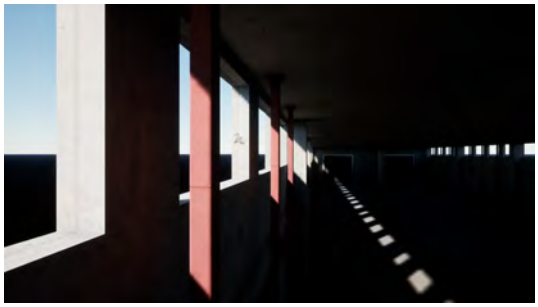


Abbildung 2.9: Eine Szene in der *Unreal Engine 4*, mit komplett statischer Beleuchtung, die in Lightmaps gespeichert ist.

Lightmass[14] verwendet hingegen *Photon Mapping*.

Das Lightmapping lässt sich jedoch im Regelfall nur mit statischen Objekten verwenden, da es unabhängig vom Realtime-Renderprozess berechnet wird. Teilweise können animierte Lightmaps verwendet werden, um aufwendig zu berechnendes Licht mit Bewegungen zu verwenden. Allerdings wird dies von Game Engines, wie *Unity* und *Unreal Engine*, nicht direkt unterstützt. Außerdem ermöglichen einige Engines, wie zum Beispiel die *Unreal Engine*, das Anpassen einiger Eigenschaften von statischen Lichtern in Echtzeit, wie die Farbe oder Intensität, wodurch das „gebakete“ Licht verändert werden kann.

Für das Ligthmapping benötigen Objekte UV-Maps ohne Überlappungen, damit Schatten ohne Problem auf ihnen dargestellt werden können. Die Editoren von Engines können diese meist auch automatisch generieren.

Abbildung 2.9 zeigt einen Screenshot aus der *Unreal Engine 4* in dem die gesamte Beleuchtung vorberechnet wurde und in Lightmaps gespeichert ist.

Shadow Map

Für die Schattenberechnung in Echtzeit, welche für bewegliche Objekte zwingend notwendig ist, wird das Shadow Mapping verwendet. Zum Berechnen von Shadow Maps wird der Z-Buffer benötigt, welcher für jeden sichtbaren Punkt einer Szene die Tiefe (Z-Wert) speichert. Der Z-Buffer ermöglicht somit, dass zu rendernde Objekte nicht vor dem Rendern sortiert werden müssen, sondern danach mit Hilfe des Z-Werts nur in der richtigen Reihenfolge angezeigt werden müssen. Die Berechnung von Shadow Maps findet nach Lance Williams wie folgt statt [Wil78, S. 271]:

1. Eine Perspektive auf die Szene wird von der Lichtquelle aus konstruiert. Dabei müssen nur die Z-Werte gespeichert werden.

2. Die Szene wird von der Kameraperspektive aus konstruiert. Es existiert eine lineare Transformation, die jedem X-Y-Z-Punkt der Kamerperspektive eine X-Y-Z-Koordinate

[14] https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/Lightmass/index.html

Abbildung 2.10: Links: Eine Szene in der *Unreal Engine 4*, mit komplett dynamischer Beleuchtung, bei der Shadow Maps zum Einsatz kommen.
Rechts: Die selbe Szene mit dynamischem Skylight und statischem Directional Light.

in der Perspektive der Lichtquelle zuordnet. Beim Generieren eines Punktes in der Kameraperspektive wird er in der Lichtquellenperspektive auf Sichtbarkeit geprüft, bevor sein Shading-Wert bestimmt wird. Ist der Punkt nicht für die Lichtquelle sichtbar, liegt er im Schatten und wird entsprechend geshadet.

Dieser Prozess wird mit mehreren Lichtquellen und abhängig von der Art der Lichtquelle komplexer und rechenaufwendiger. Links in Abbildung 2.10 ist eine komplett mit Shadow Mapping beleuchtete Szene in der *Unreal Engine 4* zu sehen. Die Kombination von Shadow Mapping und Lightmapping ist rechts in Abbildung 2.10 zu sehen. In dieser wurde, wie für die *Unreal Engine 4* typisch, ein statisches Directional Light und ein dynamisches Skylight verwendet.

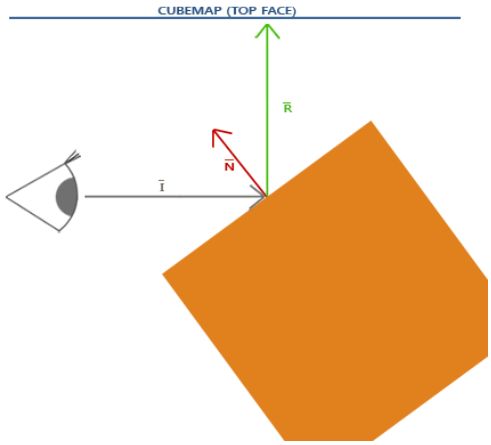


Abbildung 2.11: Die Reflexionsbestimmung durch eine Cubemap
Quelle: https://learnopengl.com/Advanced-OpenGL/Cubemaps – Zuletzt geprüft: 21.09.2020

Reflexionen

Echtzeit-Renderings ohne Ray Tracing benötigen alternative Wege um Reflexionen zu berechnen. Typisch sind drei Möglichkeiten zum Erzeugen von Reflexionen in Echtzeit: *Cubemap Reflections*, *Planar Reflections* und *Screen Space Reflections*. Meist werden diese kombiniert, um gute Ergebnisse zu erzielen.

Für *Cubemap Reflections* rendern *Reflection Captures* (Sphere/Box Reflection Capture in der *Unreal Engine*, Reflection Probe in Unity) eine Cubemap der Szene, damit diese auf reflektierenden Oberflächen angezeigt wird. Zum Bestimmen der anzuzeigenden Reflexion wird mit dem Betrachtungsvektor I und dem Normalenvektor N der reflektierenden Oberfläche der Reflexionsvektor R berechnet. Der Schnittpunkt von R mit der Cubemap bestimmt dann die anzuzeigende Reflexion (Siehe Abbildung 2.11).

Im Normalfall werden diese Cubemaps separat gerendert und nicht in Echtzeit. Dies kann abhängig von der verwendeten Engine aber auch eingestellt werden, allerdings benötigt das stetige Rendern einer Cubemap auch entsprechend viele Ressourcen. Abbildung 2.12 zeigt rechts wie Reflexionen mit Cubemaps ausehen können.

Für flache spiegelnde Oberflächen werden *Planar Reflections* verwendet. Diese rendern permanent ein Bild aus der Perspektive der spiegelnden Oberfläche und zeigen es als Reflexion auf dieser an. Die *Unreal Engine* hat dazu zum Beispiel den *Planar Reflection Actor*[15]. *Planar Reflections* müssen jedoch sparsam eingesetzt werden, um keinen zu großen Performanceeinfluss zu haben. In Abbildung 2.13 ist eine *Planar Reflection* einmal alleine und einmal kombiniert mit *Cubemap Reflections* zu sehen.

Screen Space Reflections (kurz SSR) sind eine ressourcensparende Methode zur Reflexionsberechnung in Echtzeit, denn sie zeigen bereits gerenderte Objekte im Blickfeld der Kamera, gespiegelt auf reflektierenden Oberflächen. Dazu werden Reflexionsstrahlen nur im

[15] https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/PlanarReflections/index.html

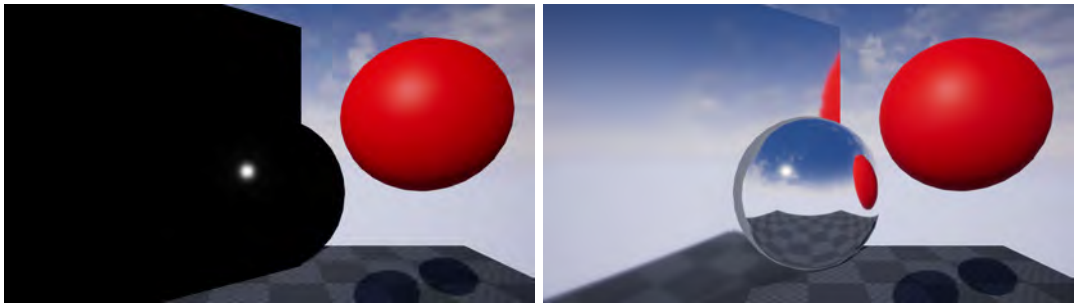


Abbildung 2.12: Links: Beispiel in der *Unreal Engine 4* ohne Reflexion. Rechts: Mit Reflexion durch eine Cubemap mit der aufgenommenen Szene

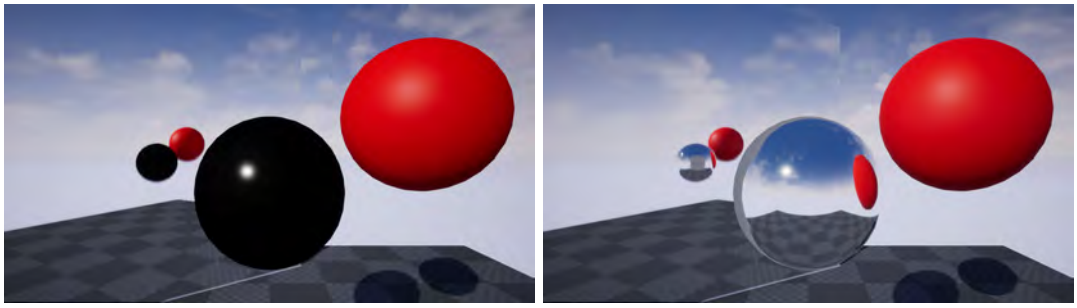


Abbildung 2.13: Links: Nur Planar Reflection auf einer Ebene in der *Unreal Engine 4*. Rechts: Zusammen mit einer Cubemap.

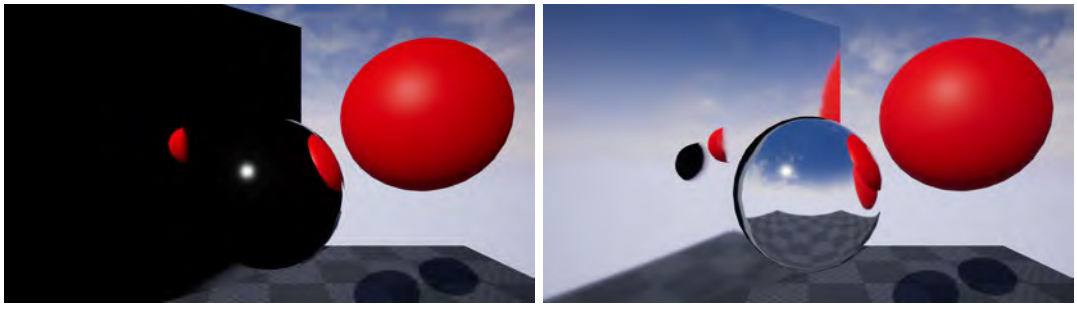


Abbildung 2.14: Links: Nur Screen Space Reflection in der *Unreal Engine 4*. Rechts: Zusammen mit einer Cubemap.

Screen Space verfolgt, was allerdings zur Folge hat, dass nicht sichtbare Objekte auch nicht in Reflexionen erscheinen können. Da Screen Space Reflections alleine meist nicht ausreichen, werden sie auch mit Cubemap Reflections kombiniert. Dabei können allerdings Fehler auftauchen, wie doppelte Reflexionen eines Objektes, was in Abbildung 2.14 zu sehen ist.

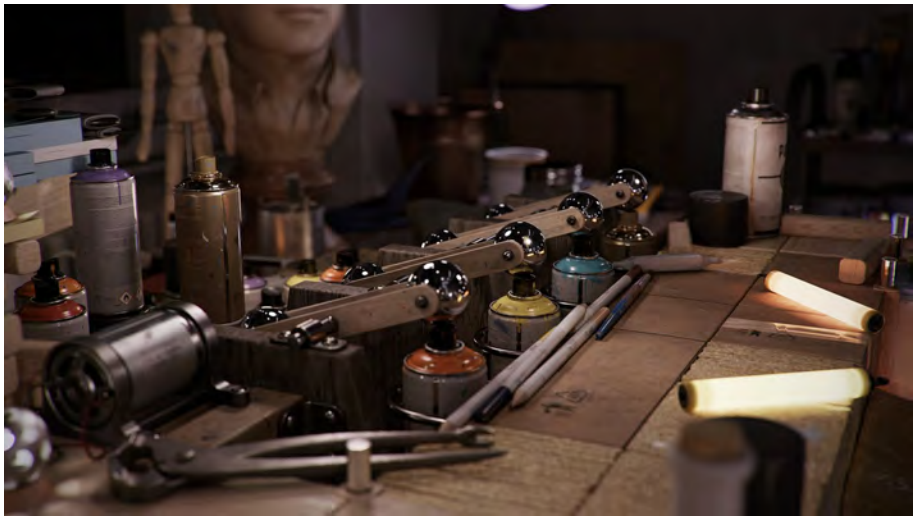


Abbildung 2.15: Screenshot aus *Nvidia's Marbles at Night*, einem Demovideo für Real-Time Ray Tracing auf Grafikkarten der *Ampere*-Architektur.
Quelle: https://www.youtube.com/watch?v=NgcYLIvlp_k – Zuletzt geprüft: 21.09.2020

Real-Time Ray Tracing

Seit 2018 ermöglichen Grafikkarten mit *Nvidia's Turing*-Architektur, sowie der älteren *Pascal* oder der neueren *Ampere*-Architektur, Ray-Tracing in Echtzeit für Computerspiele und ähnliche Anwendungen. Schon länger werden Grafikkarten für Ray Tracing verwendet, allerdings vor allen für Offline-Renderings. Bereits 2007 legten Johannes Gunther und Weitere die Grundlage für das aktuelle Real-Time Ray Tracing auf Grafikkarten [GPSS07]. Dabei erreichten sie jedoch mit einem Punktlicht und Schatten noch nur einstellige Bilder pro Sekunde, also noch deutlich zu wenig für eine flüssige Wiedergabe. Sie verwendeten eine Bounding Volume Hierachy (kurz BVH), bei der Szenen in einzelne Untersegmente aufgeteilt werden, um Strahlen schneller verfolgen zu können. Dieselbe Technik verwendet *Nvidia* für ihre Turing RT Cores, die Rechenkerne der GPU die für das Ray Tracing zuständig sind [Nvi18, S. 30]. Üblicherweise wird das Real-Time Ray Tracing in Real-Time-Renderings nur als zusätzlicher Effekt verwendet. Dabei kann es einzeln beispielsweise für Global Illumination, Ambient Occlusion, Reflexionen und Transmissionen ein- und ausgeschaltet werden, wodurch auch weniger Performance zum Berechnen benötigt wird. Außerdem werden für das Real-Time Ray Tracing meist nur wenige Strahlen verwendet, um eine schnelle Berechnung zu ermöglichen. Dies hat allerdings teils starkes Rauschen zur Folge, weswegen die RT Cores außerdem für das Entrauschen optimiert sind. In *Nvidia's* aktuellster Architektur, *Ampere*, wurden die RT Cores weiter verbessert, um höhere Framerates zu erzielen. Zusätzlich ermöglichen sie auch Motion Blur, der mit Ray Tracing berechnet wird [Nvi20]. Abbildung 2.15 zeigt ein Beispielbild für *Nvidia's* Real-Time Ray Tracing. In Abbildung 2.16 ist das zuvor verwendete Reflexionsbeispiel mit Real-Time Ray Tracing zu sehen.

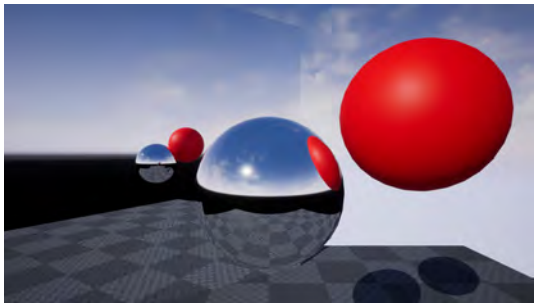


Abbildung 2.16: Reflexionsbeispiel in der *Unreal Engine 4* mit Real-Time Ray Tracing. Berechnet auf einer *Nvidia GTX 1080* ohne RT Cores.

Grafikkarten von *AMD* werden mit der *RDNA 2*-Architektur[16] ebenfalls Real-Time Ray Tracing unterstützen. Die ersten Grafikkarten mit dieser Architektur erscheinen in November und Dezember 2020. Die *RDNA 2*-Architektur wird auch in der im November 2020 erscheinenden neuen Generation von Videospielkonsolen verwendet und Real-Time Ray Tracing auf diesen ermöglichen. Diese Konsolen sind die *Play Station 5*, *Xbox Series S* und *Xbox Series X*. Beim Verfassen diese Arbeit lag allerdings noch kein Whitepaper zur *RDNA 2*-Architektur vor (Stand 02.11.2020).

2.3.3 Lokale Beleuchtung in Echtzeit

Die lokale Beleuchtung bei Real-Time-Renderings kann meist dieselben Funktionen verwenden, die auch für Offline-Renderings verwendet werden. Da in Echtzeit meist noch kein Ray-Tracing durchgeführt wird, sind auch weniger lokale Beleuchtungs-Berechnungen nötig – in der Regel nur eine pro Pixel. Für manche Algorithmen sind jedoch trotzdem Optimierungen für die Nutzung in Echtzeit notwendig. Zum Beispiel wird häufig die von Christophe Schlick [Sch94] formulierte Annäherung des Fresnel-Terms für das Cook-Torrance-Modell verwendet.

[16] https://www.amd.com/de/technologies/rdna-2

2.4 Aktuelle Vergleiche von Real-Time- und Offline-Rendering

Vergleiche von Real-Time- und Offline-Renderings finden sich online meist in Form von kurzen Artikeln oder Blogs, in denen häufig dargestellt wird, wie Real-Time-Rendering anstatt des Offline-Renderings eingesetzt werden kann[17,18,19]. Gleichzeitig werden aber auch die Schwächen des Real-Time-Renderings aufgezeigt. Folgend werden zunächst drei solcher Artikel kurz zusammengefasst. Außerdem wird eine Studie vorgestellt, die die Wahrnehmung von Real-Time-Rendering in verschieden Bereichen prüfte, die bisher Offline-Rendering verwendeten.

In „The Main Advantages of Real Time Engines vs Offline Rendering in Architecture." zeigt Mirko Vescio [Ves20] zum Beispiel die Vorteile von Real-Time-Rendering bei Architekturvisualisierungen. Dabei behandelt er einerseits den für sein Unternehmen nötigen Zeitunterschied: Wenn ein drei Minuten langes Video erstellt werden soll, würde dies beim Offline-Renderings mit einem Computer 37 Tage dauern, während es mit Real-Time-Rendering innerhalb von zehn Minuten gerendert werden kann. Mirko Vescio's Unternehmen kann außerdem durch die Verwendung von *Unity* zur Videoerstellung Kosten sparen. Üblicherweise verwendeten sie Renderfarms, um Videos nicht einen Monat lang selbst zu rendern. Dadurch entstehen allerdings zusätzliche Kosten pro Video, das auf der Renderfarm gerendert werden soll. Wenn ein Kunde Anpassungen am Video wünscht sind diese Ausgaben erneut notwendig. Mit *Unity* oder anderen Real-Time-Engines können die Anpassungen vorgenommen und direkt gezeigt werden. Außerdem entstehen für das Rendern des neuen Videos keine zusätzlichen Kosten, da es direkt in der Engine innerhalb der Videolänge neu berechnet werden kann. Darüber hinaus sieht Mirko Vescio auch den Vorteil der Interaktivität im Real-Time-Rendering, durch die Kunden eine bessere Verbindung zum gezeigten Produkt herstellen können.

Henry Winchester von der *Chaos Group* beschreibt in „Real-Time, Ray-Traced and Rasterized Rendering explained" [Win19] zunächst vereinfacht, wie Real-Time-Rendering mit Rasterung und Offline-Rendering mit Ray Tracing funktioniert. Als Hauptvorteil von gerasterten Renderings nennt er die Geschwindigkeit, mit der sie berechnet werden können. Für Renderings mit Ray Tracing sieht er deren Qualität als Hauptvorteil. Dazu sagt er weiterhin, dass Ray Tracing, im Gegensatz zur Rasterung, in der Lage ist Schatten, rekursive Reflexionen, Refraktion und reflektiertes Licht akkurat darzustellen. Für gerasterte Renderings ist eine gute Qualität für diese Features laut ihm entweder nicht erreichbar oder muss „gefaked" werden. Christopher Nichols, Geschäftsführer von *Chaos Group Labs*, wird im Artikel zitiert und sieht folgendes Hauptproblem bei der Erstellung von Real-Time-Renderings in Vergleich zu Offline-Renderings: Bei Offline-Renderings mit Ray Tracing muss der Künstler sich nur Gedanken über den Aufbau der Szene machen. Ein fotorealistisches Rendering kostet den Künstler nur Renderzeit. Bei Real-Time-Renderings hingegen muss der Künstler wissen, wie

[17] http://www.cgchannel.com/2010/10/cg-science-for-artists-part-1-real-time-and-offline-rendering/

[18] https://80.lv/articles/integrating-real-time-rendering-into-film-production-pipeline/

[19] https://www.vfxvoice.com/illuminating-the-path-ahead-for-real-time-ray-tracing/

er Schatten, Reflexionen und Refraktionen so „faked“, dass sie realistisch aussehen und benötigt mehr Zeit zum Erstellen der Szene.

Für Armen Barsegyan ergibt sich laut seinem Blog ein großer Unterschied zwischen den Workflows von Real-Time- und Offline-Rendering [Bar18]: Das Real-Time-Rendering ermöglicht es, schnell viele Iterationen durchzuführen, bei denen experimentiert werden kann. Auf Grund der Renderzeit muss für Offline-Renderings in der Regel ausführlich geplant werden. Falls Probleme erst im Rendering auffallen, wird erneut viel Zeit für das Rendering benötigt.

Eine durch *Forrester Consulting* durchgeführte und von *Epic Games* in Auftrag gegebene Studie [Ava18] sollte 2018 die Wahrnehmung von Real-Time Engines in den Bereichen Medien und Entertainment, Produktion und Architektur aufzeigen. Die Schlüsselergebnisse waren folgende:

- **Die Verwendung von Real-Time-Rendering soll ansteigen.** 81 Prozent der befragten Unternehmen werden wahrscheinlich innerhalb von einem Jahr beginnen, Real-Time Engines zu verwenden und 59 Prozent planten dies bereits.
- **Real-Time Rendering steigert die Produktivität von Mitarbeitern.** 90 Prozent der befragten Unternehmen denken, dass Real-Time-Rendering die Produktivität steigern würde.
- **Iterations- und Revisionsprozesse werden beschleunigt.** 82 Prozent der befragten Unternehmen können ihre Prozesse mit Real-Time-Rendering beschleunigen.
- **Mitarbeiter können 25 Prozent Zeit einsparen, im Vergleich zu traditionellen Prozessen.** 83 Prozent der befragten Unternehmen sparen mit Real-Time-Rendering 25 Prozent Zeit, im Vergleich zu ihren zuvor verwendeten Prozessen.
- **Entwickeln und Wiederverwenden von Assets spart Zeit und schafft Konsistenz.** 67 Prozent der befragten Unternehmen können durch Real-Time Engines wiederverwendbare Assets erstellen und dadurch ein einheitliches Markenbild schaffen.

Laut der Studie ermöglichen Real-Time Engines mehr Kreativität, weil mehr Dinge schneller ausprobiert werden können, als es mit Offline-Renderings möglich wäre. Weiterhin ermöglichen Real-Time Engines Kunden und Investoren frühe Demonstrationen von Produkten und deren Designs. Die Studie nennt drei Gründe warum viele Unternehmen noch keine Real-Time Engines verwenden:

- **Unbegründeter Glaube von hohen Kosten hindert Unternehmen an der Verwendung von Real-Time Engines.** 42 Prozent der Befragten sahen Software-Kosten als Hauptproblem, welches die Verwendung von Real-Time Engines verhindert. Obwohl diese Kosten durch „lite“ Versionen und Abo-Modelle nicht so hoch sind, wie erwartet wird. Zusätzlich halten 27 Prozent die Kosten der Implementierung für zu hoch und weitere 27 Prozent wissen nicht, ob die Investition in die Verwendung von Real-Time Engines rentabel ist.

- **Mangelnde Fähigkeiten und Erfahrung mit Real-Time Engines verhindert deren Verwendung.** Für die Verwendung von Real-Time Engines müssen Mitarbeiter entweder geschult oder neue Mitarbeiter eingestellt werden. 30 Prozent der Unternehmen sagen, dass unerfahrene Angestellte im Bereich der Real-Time Engines deren Verwendung verhindern.
- **Aufzeigen, für was Real-Time Engines verwendet werden können.** Unternehmen müssen darüber informiert werden, was mit Real-Time Engines möglich ist.

Kapitel 3

Grundlagen

3.1 Einleitung

In diesem Kapitel werden die Grundlagen der verwendeten Software erklärt, mit der Real-Time- und Offline-Renderings für diese Arbeit erstellt werden. Dabei wird vor allem auf die verwendeten Teile der Software eingegangen.
An Software wird die *Unreal Engine 4*, sowie *V-Ray for Maya* und *V-Ray for Unreal* verwendet. Ausschlaggebend für diese Wahl war das *V-Ray for Unreal* Plugin, mit dem ein möglichst nahtloser Übergang zwischen der Erstellung von Real-Time- und Offline-Renderings möglich sein sollte.

3.2 Unreal Engine

Die von *Epic Games*[1] entwickelte *Unreal Engine* ist eine Game Engine, die in ihrer ersten Version 1998 veröffentlicht wurde. Aktuell befindet sie sich in der Version 4.25.3 (Stand 23.09.2020). Neben der Spiele-Entwicklung findet die *Unreal Engine* auch Verwendung in 3D-Echtzeitanwendungen für Unternehmen, wie zum Beispiel für 3D-Produktkonfiguratoren, Produktdemonstratoren oder Trainings- und Simulationsanwendungen, und auch für die Filmproduktion. Zum Entwickeln fasst der *Unreal Editor* mehrere Programmelemente zusammen, wodurch Entwickler in einer zentralisierten Entwicklungsumgebung arbeiten können. Zu diesen Elementen gehören unter anderen folgende:

- Der **Level-Editor** in dem Objekte in einem Level platziert und konfiguriert werden können.

- Der node-basierte **Material-Editor**, zum Erstellen von Materialien. Diese Materialien können durch einige Nodes auch bei Laufzeit verändert werden.

[1] https://www.epicgames.com/site/de/home

- Zum Programmieren wird die node-basierte Skriptsprache **Blueprint Visual Scripting**[2] verwendet. Alternativ kann auch komplett mit C++ programmiert werden oder Funktionen in C++ programmiert werden, die für Blueprints aufrufbar sind. Programmierbar sind sowohl Anwendungslogik als auch Skripts zur Editor-Steuerung, welche außerdem mit Pyhton programmiert werden können.

- Der **Widget Editor**, mit dem Benutzeroberflächen erstellt werden können.

Zwar ist *Unreal Engine* nicht Open Source, aber der gesamte Quellcode auf *GitHub* frei zugänglich[3]. Dadurch kann die Engine falls nötig umfangreich angepasst werden.

3.2.1 Material-Erstellung in der Unreal Engine

Die Materialien der *Unreal Engine 4* verwenden nach Brian Karis in „Real Shading in Unreal Engine 4“ [KE13] für diffuse Reflexionen das lambertsche Modell, für spiegelnde Cook-Torrance mit der Trowbridge-Reitz-Verteilung und Schlick's Annäherung für den Fresnel-Term (Vergleich: Kapitel 2, Unterkapitel 2.2.2).

In der *Unreal Engine* gibt es zwei Objektarten für Materialien: *Materials* und *Material Instances*. Letztere sind Instanzen von Materialien, bei denen nur noch Parameter verändert werden können, die im instanziierten Material als solche gekennzeichnet wurden. *Material Instances* müssen somit bei der Entwicklung nicht neu kompiliert werden, wenn sie angepasst werden. Außerdem können die meisten Parameter von *Material Instances* auch bei Laufzeit angepasst werden. *Materials* hingegen müssen nach Anpassungen neu kompiliert werden und werden in Form eines Node-Netzwerkes erstellt. Zur umfangreichen Auswahl an Nodes gehören unter anderen Skalare und zwei bis vierdimensionale Vektoren als Konstanten oder Parameter, Rechenoperationen, Texturen, Funktionen wie die Fresnel-Funktion oder auch Inputs wie zum Beispiel Time (die aktuelle Laufzeit der Anwendung). Auch eigene Funktionen können in Form von *Material Functions*[4] erstellt und verwendet werden. Abbildung 3.1 zeigt einen einfachen Material-Graph, bei dem ein Vector4Parameter für die BaseColor verwendet wird, zwei skalare Parameter für Metallic und Roughness und eine skalare Konstante für Specular. In einer Instanz dieses Material lassen sich nun Farbe, Roughness und Metallic noch anpassen, während Specular fest vorgegeben ist. Eine Material Instance vom abgebildeten Material „M_Base“, bei der die Farbe geändert wurde, ist in Abbildung 3.2 zu sehen. In Abbildung 3.1 ist außerdem zu sehen, welche anpassbaren Parameter das Standard Shading Model „Default Lit“ hat. Die ausgegrauten Parameter sind erst mit anderen Shading Models oder anderen Blend Modes verfügbar. Zum Beispiel hat Opacity nur einen Einfluss, wenn der Blend Mode auf Tranlucent oder einem vergleichbaren steht, oder Subsurface Color nur, wenn das Shading Model Subsurface ist.

[2] https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html

[3] https://docs.unrealengine.com/en-US/GettingStarted/DownloadingUnrealEngine/index.html

[4] https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/Functions/index.html

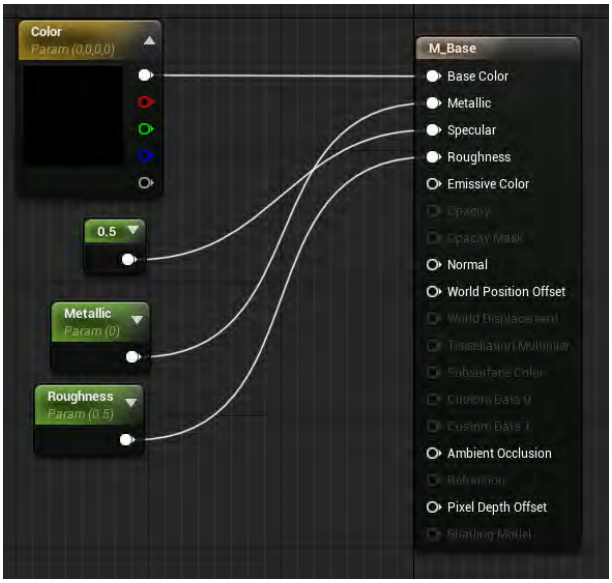


Abbildung 3.1: Ein einfacher Material-Graph in der *Unreal Engine* mit Parametern für Farbe, Metallic und Roughness und einer Konstante für Specular.

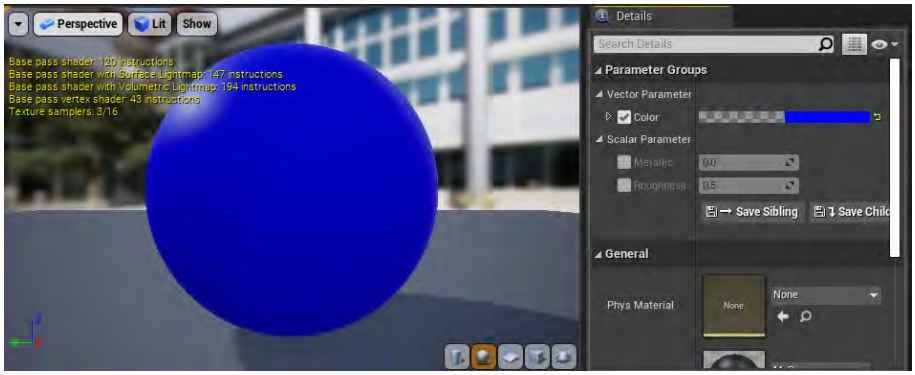


Abbildung 3.2: Eine Instanz des Materials aus Abbildung 3.1 mit angepasster Farbe

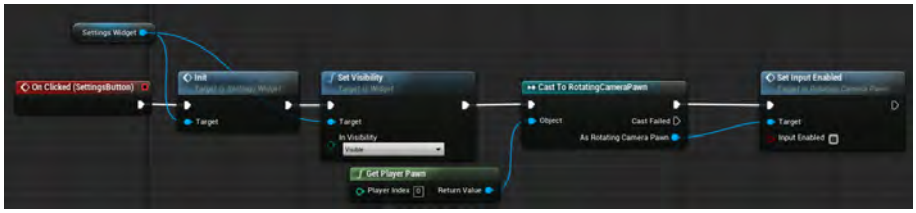


Abbildung 3.3: Ein Blueprint-Skript, das beim Klicken des „SettingsButton“ ein Widget anzeigt und den Input für den Spieler-Pawn ausschaltet.

3.2.2 Blueprint Visual Scripting

Das *Blueprint Visual Scripting* ist eine objektorientierte, node-basierte Skriptsprache, mit der im Regelfall komplette Anwendungen in der Unreal Engine entwickelt werden können. Auf C++ muss meistens nur zurückgefallen werden, wenn die Performance von Blueprints nicht ausreichend für die Anwendung ist, externe Systeme implementiert werden sollen oder sehr komplexe Features zu entwickeln sind, die mit Blueprints nicht realisierbar wären. Ein Beispiel für ein Blueprint-Skript zeigt Abbildung 3.3. Das Beispiel kommt aus der Blueprint einer Benutzeroberfläche und wird ausgeführt, wenn der Button „SettingsButton“ geklickt wird, um dann zunächst die Funktion „Init“ eines „Settings Widget“ auszuführen, dieses anzuzeigen und die Eingabe für den vom Spieler kontrollierten Pawn zu deaktivieren.

Nachfolgend werden Begriffe erklärt, die speziell für die *Unreal Engine* und das *Blueprint Visual Scripting* sind und in dieser Arbeit Anwendung finden.

Actor Jedes Objekt, das in einem Level platziert werden kann, ist ein Actor. Actor unterstützen 3D-Transformationen und können durch Programm-Code erstellt oder entfernt werden[5].

Component Eine Component (zu deutsch Komponente) ist Teil eines Actors, der eine bestimmte Art von Objekt beinhaltet[6]. Das Objekt kann in Form einer Child Actor Component selbst ein Actor sein.

Pawn Pawns sind eine Art von Actor, die der Spieler/Benutzer oder eine KI steuern kann[7].

Character Ist eine Erweiterung der Klasse Pawn und ist unter anderen in der Lage zu gehen, zu laufen und zu springen[8].

PlayerController Ist ein Interface, das die Eingaben des Spielers/Benutzer an den kontrollierten Pawn weitergibt[9].

Game Mode Ist die Definition des gespielten Spiels und dessen Regeln. Dazu zählt zum Beispiel auch welcher Pawn standardmäßig vom Spieler / Benutzer kontrolliert wird[10].

Game Instance Ist eine Manager-Klasse, in der Informationen zum aktuellen Spiel gespeichert werden können. Diese Informationen gehen nicht verloren, wenn ein neues Level geladen wird.

Save Game Mit der Klasse Save Game können Daten dauerhaft gespeichert werden, sodass sie auch nach Beenden des Spiels / der Anwendung noch erhalten sind und bei erneutem Starten geladen werden können.

Structure Entwickler können Structures erstellen, die dann als neuer Datentyp behandelt werden. Eine Structure kann mehrere Werte von gleichen oder verschiedenen Datentypen zusammenfassen.

Widget Blueprint Ist eine spezielle Blueprint-Art die zum Erstellen von Benutzeroberflächen dient[11].

3.2.3 Unreal Engine Editor Scripting und Web Remote Control

Seit der Version 4.20 unterstützt die *Unreal Engine* das Kontrollieren des *Unreal Editors* mit Hilfe von Skripts[12]. Diese Skripts können mit Blueprints oder auch mit Python entwickelt werden. Hauptsächlich ermöglicht das Editor Scripting mit Blueprint-Funktionen oder Funktionen aus der Python-Bibliothek „unreal“ Assets oder das aktuelle Level zu manipulieren. So können zum Beispiel die Kollisionen für Static Mesh-Assets automatisch generiert oder auch mehrere Actor in einem Level gleichzeitig neu positioniert werden. Es können auch Widget Blueprints für Editor-Skripts erstellt werden, welche dann in die UI des Editors integriert werden können, um sie wiederholt zu verwenden.

Mit der *Unreal Engine* Version 4.23 wurde das Editor Scripting noch um die Web Remote Control API erweitert[13]. Diese ermöglicht das Ausführen von Editor-Skripts, das Aufrufen von Funktionen die in Blueprints oder Python nutzbar sind, sowie das Lesen und Ändern von Attributen, die von Blueprints oder Python anpassbar sind, durch HTTP-Requests. Dafür muss im *Unreal Editor* der *Web Remote Control Server* mit dem Konsolenbefehl „WebControl.StartServer“ gestartet werden. Anschließend können PUT-Requests an folgende URLs gesendet werden, die abhängig von der Art der auszuführenden Aktion sind:

[5] https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/index.html
[6] https://docs.unrealengine.com/latest/INT/Engine/Components/
[7] https://docs.unrealengine.com/en-US/Gameplay/Framework/Pawn/index.html
[8] https://docs.unrealengine.com/en-US/Gameplay/Framework/Pawn/Character/index.html
[9] https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/PlayerController/index.html
[10] https://docs.unrealengine.com/en-US/Gameplay/Framework/GameMode/index.html
[11] https://docs.unrealengine.com/en-US/Engine/UMG/UserGuide/WidgetBlueprints/index.html
[12] https://docs.unrealengine.com/en-US/Engine/Editor/ScriptingAndAutomation/index.html
[13] https://docs.unrealengine.com/en-US/Engine/Editor/ScriptingAndAutomation/WebControl/index.html

Property lesen oder anpassen: `http://localhost:8080/remote/object/property`

Funktion aufrufen: `http://localhost:8080/remote/object/call`

Im Body der PUT-Request wird die auszuführende Funktion oder der abzurufende Wert angeben. Der Request-Body, um eine Funktion namens „CallTest“ des Actors „SampleActor“ im Level „Sample Level“ aufzurufen, sieht zum Beispiel wie folgt aus:

```
1 {
2     "objectPath":"/Game/Levels/SampleLevel.SampleLevel:
    PersistentLevel.SampleActor",
3     "functionName":"CallTest"
4 }
```

Wichtig ist hierfür den einzigartigen „ID Name“ des Actors im Level zu verwenden, welcher beim Hovern mit der Maus über dem Actor im Outliner angezeigt wird.

3.3 V-Ray

V-Ray ist ein von der *Chaosgroup*[14] seit 1997 entwickelter Offline-Renderer, der als Plugin für viele 3D-Programme verfügbar ist, darunter unter anderen *3dsMax*, *Maya*, *Blender* und *Unreal Engine 4*. Folgend wird behandelt wie mit *V-Ray* in *Maya* und *Unreal Engine* Materialien und Renderings erstellt werden.

3.3.1 V-Ray for Maya

Für diese Arbeit wird die Version 4.04.03 von *V-Ray for Maya* in *Maya 2018* verwendet.

V-Ray Materialien können in Maya, wie für dieses typisch, im Hypershade in Form eines *Shading Network* genannten Node-Graphen erstellt und angepasst werden (siehe Abbildung 3.4). V-Ray for Maya hat keine Einschränkungen an Nodes, die in einem Material nicht verwendet werden können. Neben dem Standard-Material *VRay Mtl* existieren unter anderen noch Materialien für Autolack (*VRay Car Paint Mtl*), Haare (*VRay Mtl Hair 3* oder *VRay Hair Next Mtl*) oder auch ein Material mit dem mehrere Materialien kombiniert werden können (*VRay Blend Mtl*). Für das reflektive Verhalten eines *VRay Mtl* lässt sich einstellen welche BRDF und somit welches lokale Beleuchtungsmodell verwendet werden soll. Zur Auswahl stehen Phong, Blinn, Ward und GGX. GGX ist eine Verteilungsfunktion für Microfacets die der Trowbridge-Reitz-Verteilung gleicht und im Cook-Torrance Modell verwendet werden kann.

Zum Rendern mit V-Ray in Maya kann eingestellt werden ob auf der CPU oder mit den CUDA-Kernen der GPU gerendert werden soll. Für CPU-Rendering kann für die Global Illumination eingestellt werden, welches Verfahren (*GI Engine*) für Primärstrahlen und Sekundärstrahlen (reflektierte oder refraktierte Strahlen) zu verwenden ist. Für die Primary Engine stehen Irradiance Map, Light Cache und Brute Force zur Auswahl. Bei der Irradiance

[14] `https://www.chaosgroup.com/`

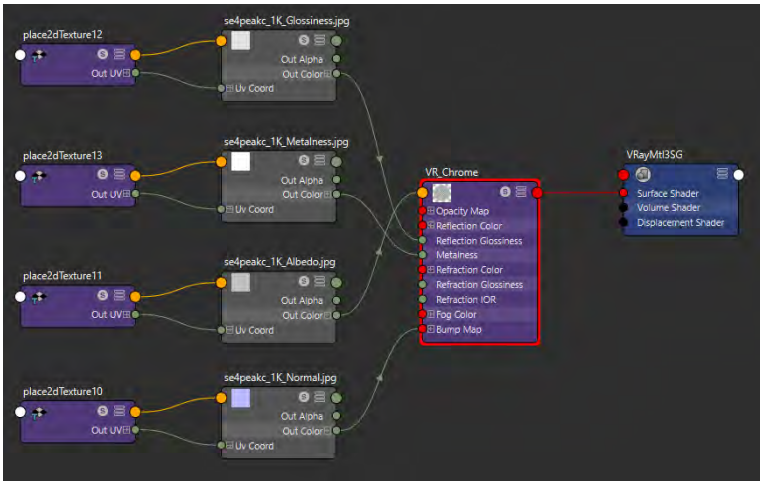


Abbildung 3.4: Das Shading Network eines VRay Mtl in Maya. Das Material hat vier Texturen als Inputs für Diffuse Color, Reflection Glossines, Metalness und Bump Map.

Map wird das Bild nach und nach in immer höherer Auflösung gerendert und dabei in weniger detaillierten Teilen der Szene weniger Strahlen verwendet. Die Strahlen bei Irradiance Map werden nicht reflektiert sondern verwenden die Secondary Engine für die Farbbestimmung eines Punktes. Bei der Secondary Engine stehen somit noch Light Cache und Brute Force zur Auswahl. Beim Light Cache werden mehrere Lichtstrahlen in die Szene geschickt und an jedem Reflexions-Punkt eines Strahls wird die Beleuchtung des restlichen Pfads gespeichert. Trifft ein Strahl auf einen Punkt, der im Light Cache gespeichert ist, muss er also nicht weiter verfolgt werden, sondern kann die gespeicherte Beleuchtung verwenden. Für den Light Cache kann eingestellt werden, wie viele Strahlen in die Szene gehen und wie groß die gespeicherten Punkte sein sollen, dabei ermöglichen kleinerer Punkte detailliertere Beleuchtung. Bei Brute Force wird jeder Punkt einzeln berechnet und entsprechend die Strahlen verfolgt. Über Parameter kann eingestellt werden wie viele Primärstrahlen pro Pixel ausgesendet werden (Subdivs). Wenn Brute Force auch die Secondary Engine ist, wird pro Reflexion/Refraktion ein neuer Strahl erstellt und es kann eingestellt werden wie oft ein Strahl abprallen kann. Bei GPU-Renderings wird nur eine GI Engine für den gesamten Renderprozess verwendet, wodurch Irradiance Map nicht verwendbar ist. Außerdem kann für GPUs eingestellt werden wie groß und wie viel Bit pro Pixel die Texturen verwenden sollen, denn während des Renderprozesses müssen diese vollständig im vorhandenen GPU-Speicher liegen.

3.3.2 V-Ray for Unreal

Verwendet wird die Version 4.30.23 von *V-Ray for Unreal* in der *Unreal Engine* Version 4.24.3.

Auch Materialien für V-Ray-Renderings können im Unreal Editor in Graphform erstellt werden, allerdings werden nicht alle Nodes von *V-Ray for Unreal* unterstützt[15]. Drei Material-Vorlagen enthält das Plugin bereits, die den V-Ray-Materialien in andereren Anwendungen entsprechen sollen und beim Importieren von V-Ray Szenen verwendet werden. Diese sind *VRayMtl*[16], *VRayCarPaintUberMtl*[17] und *VRayPBRMtl*[18]. Letzteres soll einen PBR-Workflow ähnlich dem der *Unreal Engine* mit V-Ray ermöglichen. Beim Erstellen eines dieser Materialien wird letztlich eine Material Instance erstellt, die genauso angepasst werden kann (Siehe Unterkapitel 3.2.1).

Für V-Ray Renderings im Unreal Editor kann nur die GPU verwendet werden. Dabei kann aber zwischen Rendern mit CUDA oder mit der *Nvidia OptiX Ray Tracing Engine*[19] gewählt werden, wovon letztere Option jedoch experimentell ist und nur mit *Nvidia RTX* Grafikkarten funktioniert. Somit stehen für die Global Illumination auch nur Light Cache und Brute Force zur Auswahl. Zum Entrauschen des finalen Bildes kann neben dem V-Ray Denoiser auch ein Nvidia AI Denoiser verwendet werden. Spezifische Einstellungen für V-Ray for Unreal sind außerdem die Auflösung vom Skylight einzustellen und Static Meshes nicht in diesem anzuzeigen oder die Auswahl eines Fallback Materials, das angezeigt wird, wenn ein Material Nodes enthält, die V-Ray nicht darstellen kann. Ansonsten gleichen die V-Ray-Rendereinstellungen denen in anderen 3D-Programmnen. Das Starten eines V-Ray-Renderings geschieht durch das Klicken auf den V-Ray-Button über der Level-Ansicht. Animationen können mit dem Render-Button des *Level Sequencers* der *Unreal Engine* auch mit V-Ray gerendert werden.

V-Ray for Unreal ermöglicht zusätzlich das Light Mapping mit V-Ray durchzuführen. Neben vielen Material-Nodes unterstüzt *V-Ray for Unreal* auch viele weitere Features der *Unreal Engine* noch nicht, wie zum Beispiel instanziierte Meshes, Partikel, den HDRI-Backdrop-Actor oder Decals.

[15] https://docs.chaosgroup.com/display/VRAYUNREAL/Supported+Features
[16] https://docs.chaosgroup.com/display/VRAYUNREAL/VRayMtl
[17] https://docs.chaosgroup.com/display/VRAYUNREAL/VRayCarpaintUberMtl
[18] https://docs.chaosgroup.com/display/VRAYUNREAL/VRayPBRMtl
[19] https://developer.nvidia.com/optix

Kapitel 4

Auto-Konfigurator

In diesem Kapitel wird der Auto-Konfigurator und seine Programmarchitektur behandelt. Dieser dient im Rahmen dieser Arbeit für Workflow-Vergleiche, für Renderings für die Evaluation und als Anwendungsbeispiel, in dem sowohl Offline- als auch Real-Time-Rendering verwendet werden. Für letzteres wird er um ein Feature erweitert. Der Konfigurator wurde mit der *Unreal Engine* unter Verwendung des *V-Ray for Unreal* Plugins entwickelt. Dabei war seine Verwendung für diese Arbeit bereits vorgesehen.

Die Konfigurator-Anwendung hat die grundlegenden Features, die von einem Produkt-Konfigurator zu erwarten sind (Siehe Abbildung 4.1): Mit der Maus lässt sich bei gedrückter linker Taste die Kamera um das Fahrzeug rotieren. Links können abhängig vom aktiven Auto-Modell Materialien oder auch Objekte, wie Felgen, für die Konfiguration ausgewählt werden und zwischen verschieden Kameraperspektiven kann gewechselt werden. Oben rechts kann die Hintergrundumgebung und das aktuelle Modell gewählt werden.

Abbildung 4.1: Screenshot des Auto-Konfigurators auf dem die Farbe des Auto-Lacks angepasst wird.

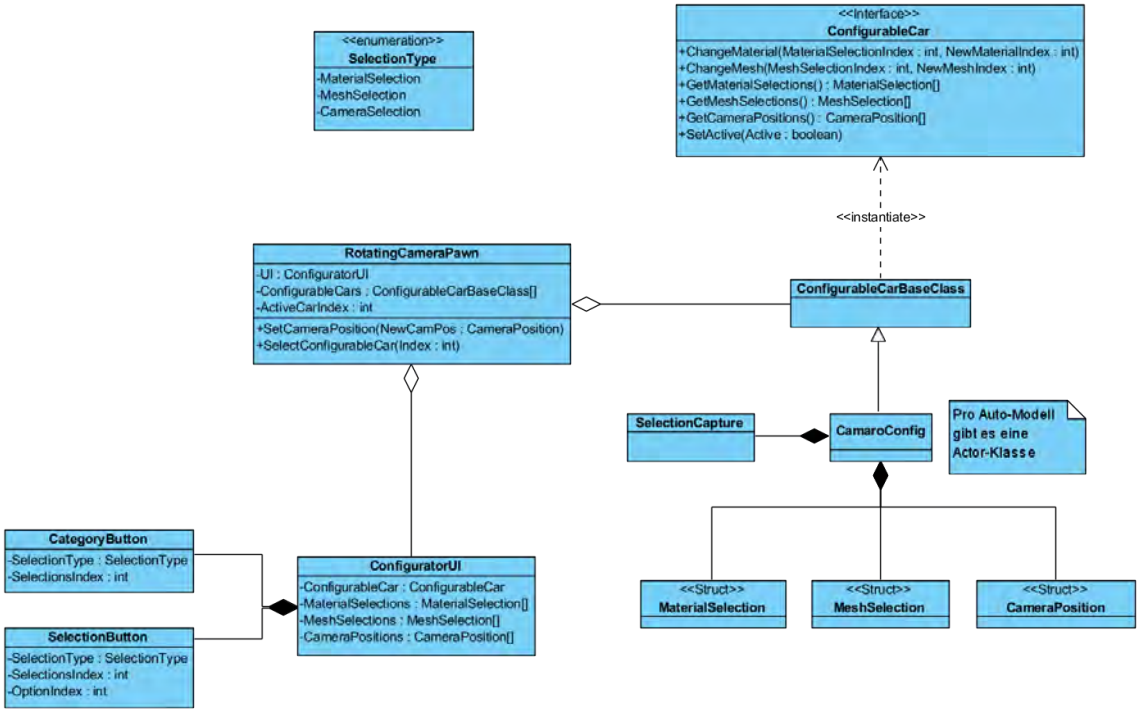


Abbildung 4.2: Ein Klassendiagramm des Auto-Konfigurators reduziert auf die relavanten Actor, Parameter und Funktionen.

4.1 Programmarchitektur des Auto-Konfigurators

In Form eines Klassendiagramms ist die Architektur des Auto-Konfigurators in Abbildung 4.2 zu sehen.

Der Benutzer steuert den **RotatingCameraPawn**. Dieser ermöglicht das Rotieren und Zoomen der Kamera, dabei wird gleichzeitig eine VRay-Kamera mit positioniert, sodass sie stets an derselben Stelle wie die Nutzer-Kamera ist. Diese Kamera soll später für Renderings verwendet werden können. Außerdem dient der RotatingCameraPawn zur Verwaltung der verwendbaren Auto-Modelle, dazu müssen im Level Actor vom Typ **ConfigurableCarBaseClass** in das Array *ConfigurableCars* des Pawns eingegeben werden. Beim Starten der Anwendung erstellt der Pawn die Benutzeroberfläche mit den zu verwendenden Autos und initialisiert sie mit dem Auto, das durch *ActiveCarIndex* angegeben ist. Wird ein anderes Auto-Modell in der Benutzeroberfläche ausgewählt, wird es ebenfalls vom Pawn geändert.

Ein konfigurierbares Auto wird durch eine Actor-Klasse erstellt, die als Elternklasse **ConfigurableCarBaseClass** hat, welche wiederum das Interface **ConfigurableCar** implementiert, so zum Beispiel **CamaroConfig**. Das Interface gibt Funktionen vor, mit denen ein

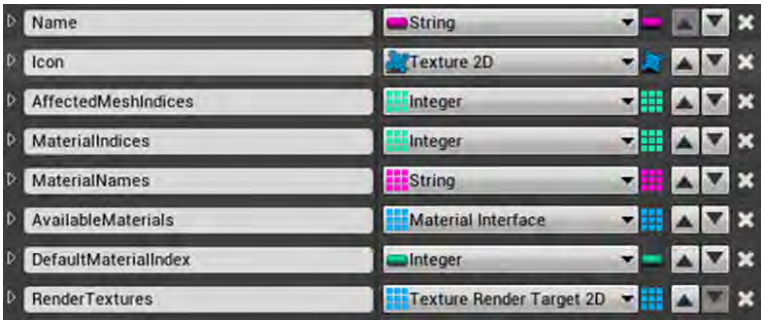


Abbildung 4.3: Das Struct MaterialSelection in dem alle Werte, die für die Materialanpassung nötig sind, gespeichert werden.

Auto-Modell von der Benutzeroberfläche aus konfiguriert werden kann. Für ein Auto-Modell existieren drei Arten von Einstellungsmöglichkeiten:

1. Ein Material verändern (zum Beispiel den Autolack)
2. Ein oder mehrere gleiche Meshes austauschen (zum Beispiel die Felgen)
3. Die Kameraposition ändern, also den Punkt um den sich die Kamera dreht

Diese Einstellungsarten werden in Form der Structs **MaterialSelection**, **MeshSelection** und **CameraPosition** abgebildet. Jeder Konfigurator-Actor eines Auto-Models hat ein Array jedes Structs,in dem die möglichen Materialien, Meshes und Kamerapositionen eingetragen werden. Das Struct MaterialSelection ist in Abbildung 4.3 zu sehen. Ein solcher Eintrag im Array eines Auto-Modells repräsentiert eine Materialanpassung wie zum Beispiel die Lackfarbe. Diese wird in der Benutzeroberfläche durch **Name** und **Icon** angezeigt. Das Array **AffectedMeshIndices** enthält die Indices aller Meshes des Autos, auf denen das Material auszutauschen ist. **MaterialIndices** enthält dazu zugehörig den Index des Materials auf dem Mesh mit selben Index. **MaterialNames** und **AvailableMaterials** sind die auswählbaren Materialien als Name und Material Interface. Mit **DefaultMaterialIndex** wird das Standard-Material angegeben. **RenderTextures** dient zur Anzeige des Materials auf der Benutzeroberfläche und wird im Rahmen dieser noch erklärt. Das Struct **MeshSelection** ist sehr ähnlich aufgebaut, verwendet anstatt Materialien jedoch Meshes. Abbildung 4.4 zeigt das Event **ChangeMaterial**, mit dem unter Verwendung des Structs MaterialSelection ein Material geändert wird. Dazu nötig sind der Index der Materialanpassung im MaterialSelections-Array und der Index des neuen Materials. Für jedes Mesh, das in dieser MaterialSelection angegeben ist, wird das Material mit SetMaterial geändert.

Das Struct **CameraPositon** ist in Abbildung 4.5 zu sehen. **CameraName** und **CameraIcon** dienen zur Anzeige in der Benutzeroberfläche. Mit den übrigen Einträgen wird beim Ändern der Kameraposition die Transformation und Parameter des **RotatingCameraPawn** geändert, sodass er die Position entsprechend einnimmt. Zur Einstellung der Kamerapositionen an einem Auto-Modell wurde der Actor **CameraPosition** erstellt, der als ChildActor verwendet werden kann, um die Kamera visuell zu platzieren und die Rotation einzuschränken.

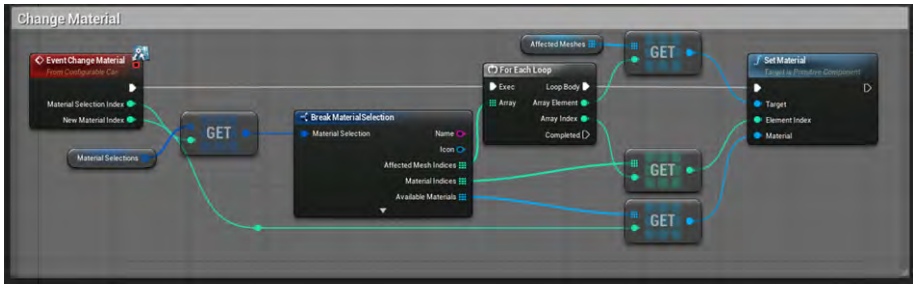


Abbildung 4.4: Das Event ChangeMaterial, mit dem ein Material des Autos geändert wird.

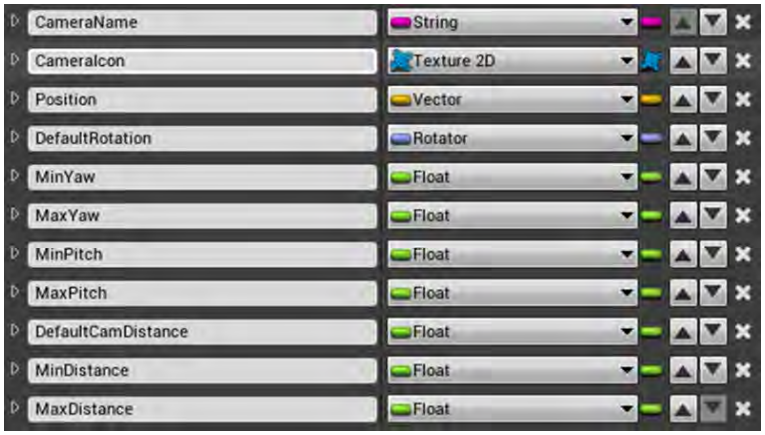


Abbildung 4.5: Die Einträge der Structures CameraPosition.

Im Construction Script eines Auto-Modell-Actors werden dann die Werte der Child Actor in das Array des Structs CameraPosition geschrieben.

Die bereits in Abbildung 4.1 zu sehende Benutzeroberfläche wird durch die Widget Blueprint **ConfiguratorUI** dargestellt. Wenn ein Auto im RotatingCameraPawn eingestellt wird, wird die Funktion **Init** der ConfiguratorUI ausgeführt, wodurch die Variablen **ConfigurableCar**, **MaterialSelections**, **MeshSelections** und **CameraPositons** gesetzt werden. Dann wird für jeden Array-Eintrag in MaterialSelections und MeshSelections ein **CategoryButton** erstellt. Die CategoryButtons befinden sich ganz links, in Abbildung 4.1 sind dies also „Carpaint“, „Rims“ und „Camera“. Der CategoryButton für die Kamerapositionen wird nur erstellt, wenn mehr als eine Position vorhanden ist. Die Art der Anpassung wird in der Variable **SelectionType** im CategoryButton gespeichert und der Index im entsprechenden Array in **SelectionsIndex**. Die **SelectionButtons** werden generiert, wenn ein CategoryButton gedrückt wurde. Die Darstellung von Materialien und Meshes in der UI ist komplett dynamisch. Dafür erstellt ein Auto-Konfigurations-Actor bei seiner Aktivierung für jedes Material und Mesh, das ausgewählt werden kann, einen **SelectionCapture**-Actor. Dieser besteht aus einer MeshComponent und einer SceneCaptureComponent2D, was eine simple Kamera ist, die ihr

Abbildung 4.6: Der SelectionButton für ein Material namens „Blue Metallic“.

Bild in einer RenderTexture zur Weiterverwendung speichert. Für Materialien ist die MeshComponent eine Sphäre, die das Material verwendet und für Meshes das entsprechende Mesh. Nach dem Erstellen des SelectionCapture-Actors wird das Bild der SceneCaptureComponent2D einmal gerendert und im **RenderTextures**-Array der MaterialSelection oder MeshSelection gespeichert. Diese RenderTextures verwendet die ConfiguratorUI dann zur Anzeige in den SelectionButtons, wie in Abbildung 4.6.

Abbildung 4.7: Das Studio-Level im Unreal Editor. Zu sehen sind neben den Auto-Modell mit Kamerapositionen und den Meshes des Levels, oben eine *VRayLightRect* und zwei von fünf SphereReflectionCaptures über dem Boden im Vordergrund.

4.2 Level des Auto-Konfigurators

Der Konfigurator hat zwei Level, die als Hintergrund-Umgebungen auswählbar sind. Folgend wird auf den Aufbau dieser Level eingegangen, um darzustellen, was später die Renderings beeinflusst.

Das Level „Studio“ (Abbildung 4.7) zeigt die Auto-Modelle auf einem runden Podest, das von einem stilisierten Polygon-Boden und einer weißen Wand mit Hohlkehle umgeben ist. Zur Belichtung dient ein komplett weißer *VRayLightDome* und ein kreisförmiges, statisches *VRayLightRect*, das sich über dem Auto befindet. Für Reflexionen werden fünf *SphereReflectionCaptures* verwendet; eine in der Mitte des Studios in einer Höhe von 131 cm (an einer Position wo sich das Auto-Modell befindet) und vier weitere über den Boden rundum das Podest damit dieser auch Reflexionen hat.

Im Level „Warehouse“ (Abbildung 4.8) wird eine realistischere Hintergrundumgebung verwendet. Zur Beleuchtung dient ein *VRayLightDome* mit einem 360°-HDR-Bild und ein statisches Directional Light. Zusätzlich ist in jedem Fenster ein *LightmassPortal*[1] mit denen die Strahlenanzahl erhöht wird, die von ihnen ausgehen. Für die Reflexionen wird nur eine *SphereReflectionCapture* an der Position des Auto-Modells verwendet.

[1] https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/LightmassPortals/index.html

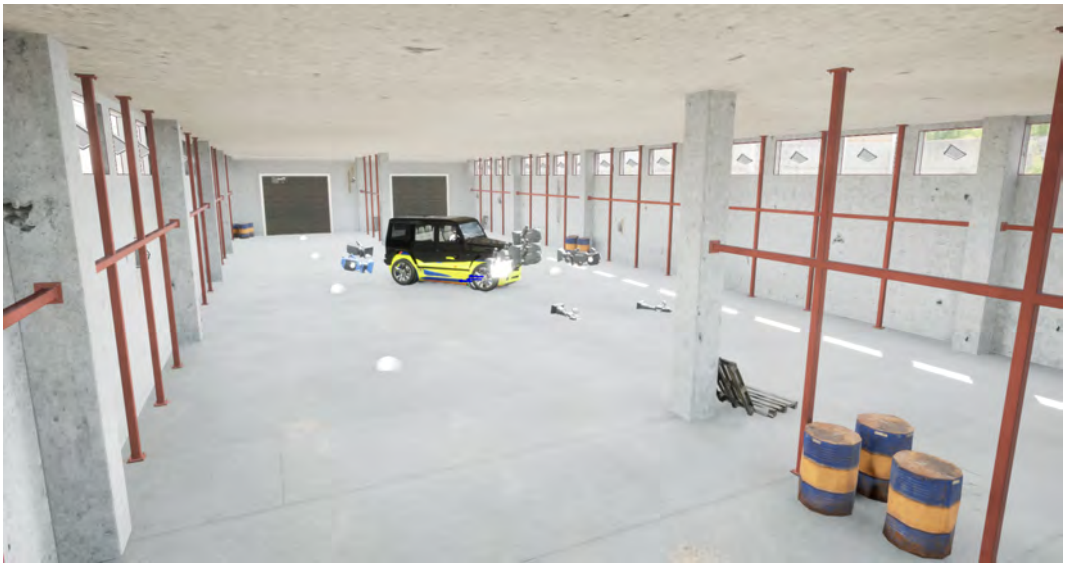


Abbildung 4.8: Das Warehouse-Level im Unreal Editor. In jedem Fenster sind die *LighmassPortals* an ihrem Icon zu erkennen.

Kapitel 5

Konzept und Entwicklung eines Systems zur Erstellung von Offline-Renderings aus einer Konfiguratoranwendung

5.1 Einleitung und Konzept

Mit dem Auto-Konfigurator soll ein Anwendungsbeispiel entwickelt werden, das Offline- und Real-Time-Rendering kombiniert. Der Konfigurationsprozess geschieht dabei komplett in Real-Time und wurde bereits entwickelt (siehe Kapitel 4). Ergänzt werden soll dieser durch die Möglichkeit, ein Offline-Rendering der Konfiguration zu erstellen. Dadurch soll der Benutzer ein möglichst realistisches Bild seiner Konfiguration erhalten können.

Vergleichbar ist dies mit einem Feature des Online-Konfigurators vom *Audi e-tron*[1]. Dieser ermöglicht ein Video der eigenen Konfiguration herunterzuladen. Allerdings wird für dieses die Echtzeitdarstellung verwendet. Außerdem wird das Video nicht wirklich dynamisch für die Konfiguration erstellt: Für jede mögliche Konfiguration existiert bereits eine Datei zum Herunterladen, was an den Dateinamen erkennbar wird.

Der Auto-Konfigurator soll stattdessen dem Benutzer ermöglichen, eine Kameraperspektive frei auszuwählen und daraufhin ein Rendering zu starten. Konfigurator-Anwendungen werden typischerweise auf einem Computer bei einem Auto-Händler, oder im Falle von Online-Konfiguratoren auf Privatgeräten ausgeführt. Ein Offline-Rendering sollte idealerweise nicht auf diesen durchgeführt werden müssen. Denn diese Computer sollten einerseits weiter bedienbar sein und haben außerdem nicht zwingend die nötige Hardware, um ein Rendering schnell durchzuführen. Stattdessen sollte eher auf einem Renderserver gerendert werden.

Zusammenfassend soll der Auto-Konfigurator wie folgt um ein Offline-Rendering-Feature ergänzt werden: Hat der Benutzer seine Wunschkonfiguration fertiggestellt, wählt er eine

[1] http://audietron.zerolight.com/#/ – zuletzt geprüft 29.09.2020

Abbildung 5.1: Variablen der Configurator_GameInstance werden beim Levelwechsel gesetzt.

Kameraposition und drückt einen Renderbutton. Die Konfigurator-Anwendung sendet die notwendigen Informationen zur Konfiguration an einen Renderserver. Dazu gehören das gewählte Auto-Modell, alle aktuellen Materialien beziehungsweise Meshes, die verwendet werden, und die verwendete Umgebung, sowie die aktuelle Kameraposition und -rotation. Anschließend startet das Rendering auf dem Server.

In den folgenden Abschnitten werden zunächst zwei andere Erweiterungen der Konfiguratoranwendung behandelt, die sie von ihrem Stand vor Beginn der Arbeit unterscheidet. Anschließend wird die Offline-Rendering-Erweiterung behandelt, welche in Form eines Prototypen implementiert wird.

5.2 Konfigurationen zwischen Leveln beibehalten

Beim Wechsel der Hintergrundumgebung in der originalen Konfiguratoranwendung, wurde erneut das erste Auto-Modell mit Standard-Konfiguration angezeigt. Der Grund dafür ist, dass ein neues Level geladen und nicht nur der Hintergrund geändert wurde. Für ein besseres Nutzererlebnis soll die Konfiguration zwischen den Leveln erhalten bleiben. Dazu wird eine Game Instance namens **Configurator_GameInstance** verwendet. Denn Game Instances behalten ihre Daten auch beim Levelwechsel. Die **Configurator_GameInstance** enthält eine Integer-Variable **CarIndex**, sowie zwei Float-Arrays namens **ActiveMaterialIndices** und **ActiveMeshIndices**. **CarIndex** speichert das aktuell verwendete Auto-Modell und die beiden Arrays speichern die verwendeten Materialien und Meshes. Hierfür wären Integer-Arrays sinnvoller, aber für das Offline-Rendering-Feature werden noch zwingend Float-Arrays nötig sein. Die Werte der Variablen werden in der ConfiguratorUI gesetzt, wenn das Level geändert wird. Dies ist in Abbildung 5.1 zu sehen. Die Funktion **CreateIndexArrays** füllt dabei die Arrays **ActiveMaterialIndices** und **ActiveMeshIndices** mit den Indices der Materialien und Meshes, die aktuell verwendet werden.

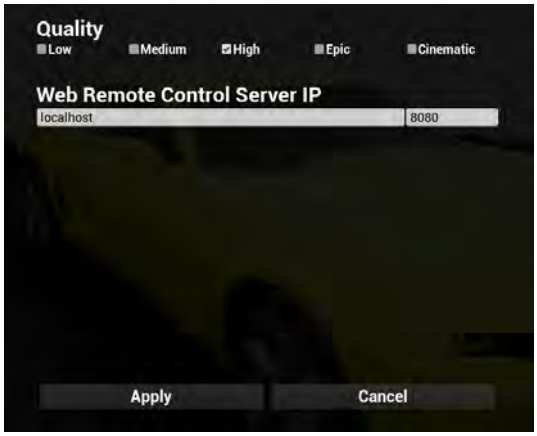


Abbildung 5.2: Das Einstellungsmenü des Auto-Konfigurators mit Qualitätsoptionen und IP- und Port-Eingabe für den Renderserver.

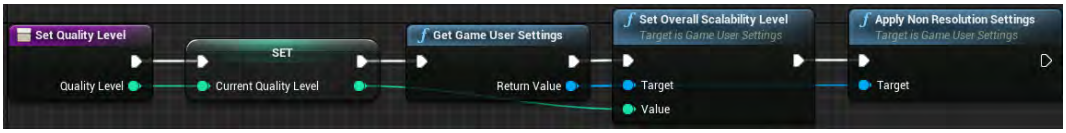


Abbildung 5.3: Die Funktion SetQualityLevel verwendet die GameUserSettings zum Anpassen und Speicher der graphischen Qualität.

5.3 Einstellungsmenü

Das Einstellungsmenü kann in der Benutzeroberfläche durch das Klicken auf das Zahnrad eingeblendet werden. Einerseits ermöglicht es die graphische Qualität anzupassen und andererseits die IP-Adresse und Port des Servers einzustellen, der für das Offline-Rendering verwendet werden soll. Das Menü ist in Abbildung 5.2 zu sehen. Die Qualitätseinstellung geschieht über die *Scalability Levels* der *Game User Settings* der Unreal Engine. Diese fassen mehrere Qualitätseinstellungen zusammen und speichern sie automatisch, sodass sie beim Neustart der Anwendung noch erhalten sind. Nachdem der „Apply"-Button gedrückt wurde, wird die in Abbildung 5.3 zu sehende Funktion **SetQualityLevel** ausgeführt. Der Input **QualityLevel** ist abhängig von der gewählten CheckBox im Einstellungsmenü.

Beim Drücken des „Apply"-Buttons wird außerdem die IP-Adresse und Port des Renderservers gesetzt. Dieser wird als Web Remote Control Server bezeichnet, da letztlich dieser für das Starten des Renderings zuständig ist. IP-Adresse und Port werden in der **Configurator_GameInstance** durch Aufrufen der Funktion **SetServerInfo** gesetzt. Diese ist in Abbildung 5.4 zu sehen. Die Funktion setzt nicht nur die entsprechenden Variablen der Game Instance, sondern speichert sie auch in einem SaveGame, damit sie auch beim Neustart der Anwendung noch erhalten sind.

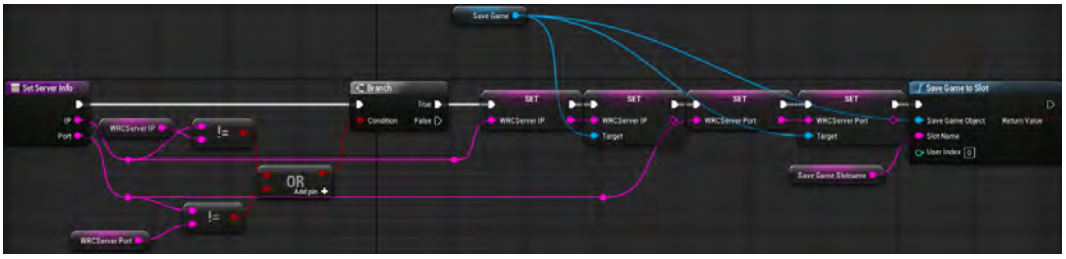


Abbildung 5.4: Mit der Funktion SetServerInfo der Configurator_GameInstance werden IP-Adresse und Port des Web Remote Control Servers gesetzt und in einem SaveGame gespeichert.

Abbildung 5.5: Das EditorUtilityWidget mit dem der Web Remote Control Server im Unreal Editor gestartet und gestoppt werden kann.

5.4 Offline-Rendering vom Konfigurator aus

Für ein Rendering von der vom Benutzer vorgenommenen Konfiguration, müssen alle relevanten Informationen von der Konfiguratoranwendung an den Unreal Editor übertragen und anschließend das V-Ray Rendering im Editor gestartet werden. Die Informationen werden als HTTP-Requests an die Web Remote Control API gesendet. Dafür findet das VaRest Plugin[2] Verwendung, mit dem HTTP-Requests in Blueprints erstellt und versendet werden können. Grundlegend sieht der Prozess zum Starten eines Offline-Renderings folgendermaßen aus und wird nachfolgend noch genauer erläutert:

1. Der Benutzer drückt nach dem Konfigurieren eines Autos und Positionieren der Kamera den Button „Start Rendering“.

2. Level, Konfiguration und Kameraposition werden an den Unreal Editor übertragen und übernommen.

3. Das V-Ray Rendering wird im Unreal Editor gestartet.

Auf dem Renderserver muss folglich der Unreal Editor mit dem V-Ray for Unreal Plugin laufen. Im Unreal Editor muss außerdem der Web Remote Control Server laufen. Dieser wird über Konsolenbefehle gestartet, welche aber zum Beispiel auch durch ein *Editor Utility Widget* aufgerufen werden können. Ein solches Widget wurde für diese Arbeit entwickelt und

[2] https://www.unrealengine.com/marketplace/en-US/product/varest-plugin

ist in Abbildung 5.5 zu sehen. Es ermöglicht neben dem Starten und Stoppen des Servers auch das Level zurückzusetzen, indem das aktuelle Level ohne zu Speichern neu geladen wird.

5.4.1 EditorRenderActor

Auf der Editor-Seite wurde außerdem der **EditorRenderActor** erstellt. Dieser muss in jedem Level vorhanden sein, damit die HTTP-Requests zur Render-Vorbereitung korrekt ausgeführt werden können. Im EditorRenderActor gibt es drei Funktionen, die per HTTP-Request aufgerufen werden sollen: **PrepareRender**, **StartRender** und **AbortRender**.

PrepareRender

Die Funktion **PrepareRender** hat die Parameter **CameraLocation**, **CameraRotation**, **ActiveCarIndex**, **ActiveMaterialIndices[]** und **ActiveMeshIndices[]**. Abbildung 5.6 zeigt die komplette Funktion. Zunächst wird der RotatingCameraPawn im Level gesucht. Anschließend werden die Positon und Rotation des VRayCamera-Actors, welcher im Pawn gespeichert ist, auf die Parameter CameraLocation und CameraRotation gesetzt. Danach wird die VRayCamera gepilotet, dadurch wird ihre Perspektive im Viewport eingenommen und das nächste V-Ray-Rendering kann mit dieser durchgeführt werden. Schließlich werden für das Auto-Modell mit dem Index ActiveCarIndex die Materialien und Meshes, die per Parameter vorgegeben werden, eingestellt und alle nicht aktiven Autos unsichtbar gemacht.

StartRender und AbortRender

Die Funktionen **StartRender** und **AbortRender** rufen beide jeweils ein Python-Skript auf, mit dem das Rendering gestartet oder abgebrochen wird. Es werden Pyhton-Skripte verwendet, da das V-Ray-Rendering nicht direkt über reine Editor-Skripte gestartet werden kann. Dies wurde von der Chaosgroup bestätigt, könnte aber noch in einer zukünftigen Version von V-Ray for Unreal möglich werden[3]. Stattdessen wird die Python-Bibliothek *PyAutoGUI*[4] verwendet, mit der Maus und Tastatur programmatisch gesteuert werden können, wodurch ein Python-Skript mit einer Benutzeroberfläche interagieren kann. Diese Bibliothek wird genutzt, um den V-Ray-Button im Unreal Editor zu drücken, um ein Rendering zu starten, oder den Stop-Button im *V-Ray Frame Buffer* (kurz VFB) zu drücken, um ein Rendering abzubrechen. Allerdings hat PyAutoGUI die Einschränkung, dass das zu bedienende Fenster sich auf den Hauptbildschirm befinden muss. Zur Verwendung einer externen Bibliothek mit dem Unreal Engine Python Plugin sollte diese am besten in einem Ordner liegen, der Teil von **sys.path** der Unreal Engine Python-Umgebung ist[5]. Die PyAutoGUI Bibliothek wurde

[3] https://forums.chaosgroup.com/forum/v-ray-for-unreal-forums/v-ray-for-unreal-general/1075982-using-editor-scripting-to-start-a-v-ray-rendering#post1076697 – zuletzt geprüft 29.09.2020

[4] https://pypi.org/project/PyAutoGUI/

[5] https://docs.unrealengine.com/en-US/Engine/Editor/ScriptingAndAutomation/Python/index.html

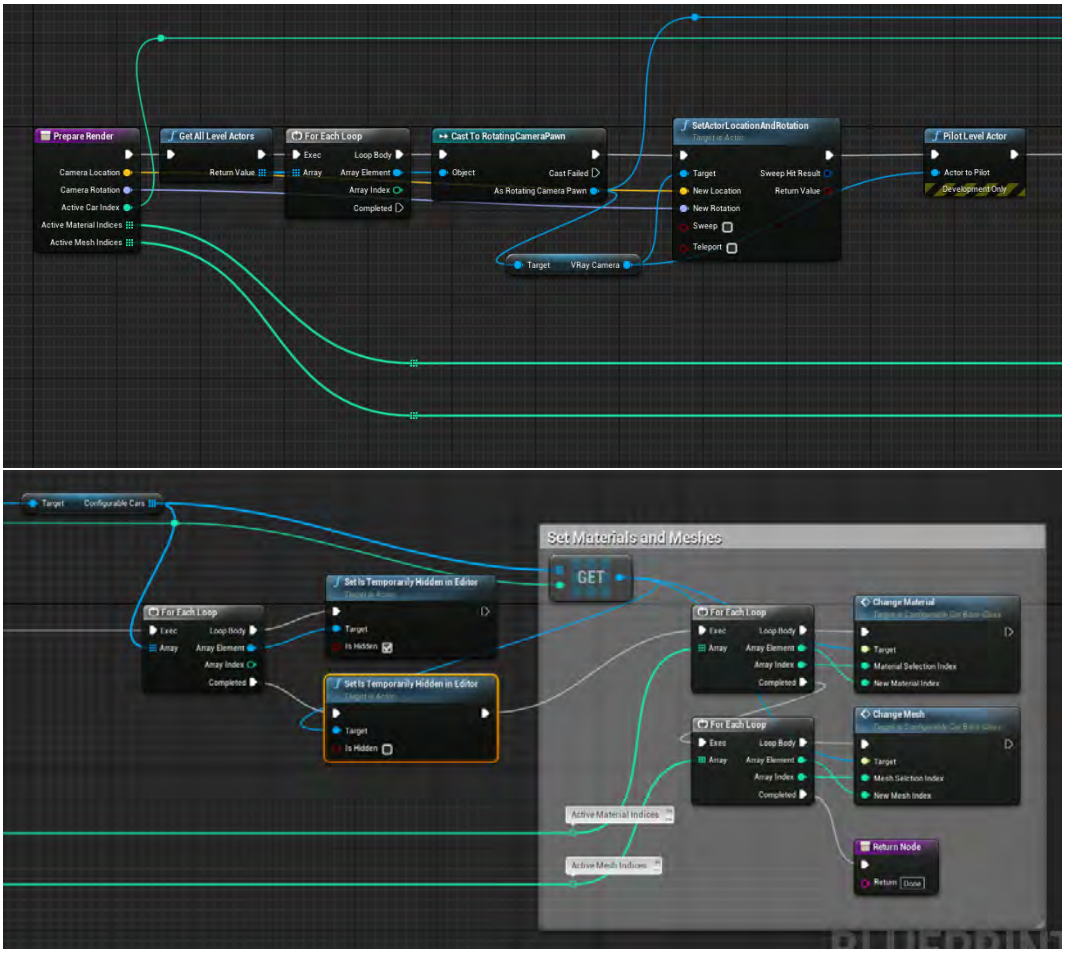


Abbildung 5.6: Die Funktion PrepareRender im EditorRenderActor. Zur besseren Lesbarkeit in zwei Bilder aufgeteilt.

im Projekt in den Ordner „Content/Python“ kopiert, in dem auch die verwendeten Skripte liegen. StartRender führt das folgende Python-Skript aus:

```
1 import pyautogui, time
2 import unreal
3 editor=pyautogui.getWindowsWithTitle("Unreal Editor")[0]
4 editor.minimize()
5 editor.maximize()
6 time.sleep(0.5)
7 nextMouseCoords = pyautogui.center(pyautogui.locateOnScreen(unreal
      .SystemLibrary.get_project_content_directory()+"Python/"+"
      VRayButton.png"))
8 pyautogui.moveTo(nextMouseCoords)
9 pyautogui.click()
```

In diesem wird zunächst das Unreal Editor Fenster minimiert und maximiert, um in den Vordergrund zu treten. Zur Sicherheit wird eine halbe Sekunde gewartet, bevor der V-Ray-Button mit Hilfe eines Screenshots auf dem Bildschirm gesucht, die Maus über ihn bewegt und geklickt wird, wodurch das Rendering startet. Zum Abbrechen eines laufenden Renderings, führt AbortRender folgendes Skript aus, das im V-Ray Frame Buffer den Stop-Button drückt:

```
1 import pyautogui, time
2 import unreal
3 vfb=pyautogui.getWindowsWithTitle("V-Ray frame buffer")[0]
4 vfb.maximize()
5 time.sleep(0.5)
6 nextMouseCoords = pyautogui.center(pyautogui.locateOnScreen(unreal
      .SystemLibrary.get_project_content_directory()+"Python/"+"
      VRayStopButton.png"))
7 pyautogui.moveTo(nextMouseCoords)
8 pyautogui.click()
```

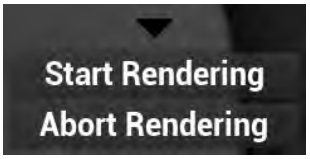


Abbildung 5.7: Buttons zum Starten und Abbrechen eines Renderings in der Benutzeroberfläche des Auto-Konfigurators.

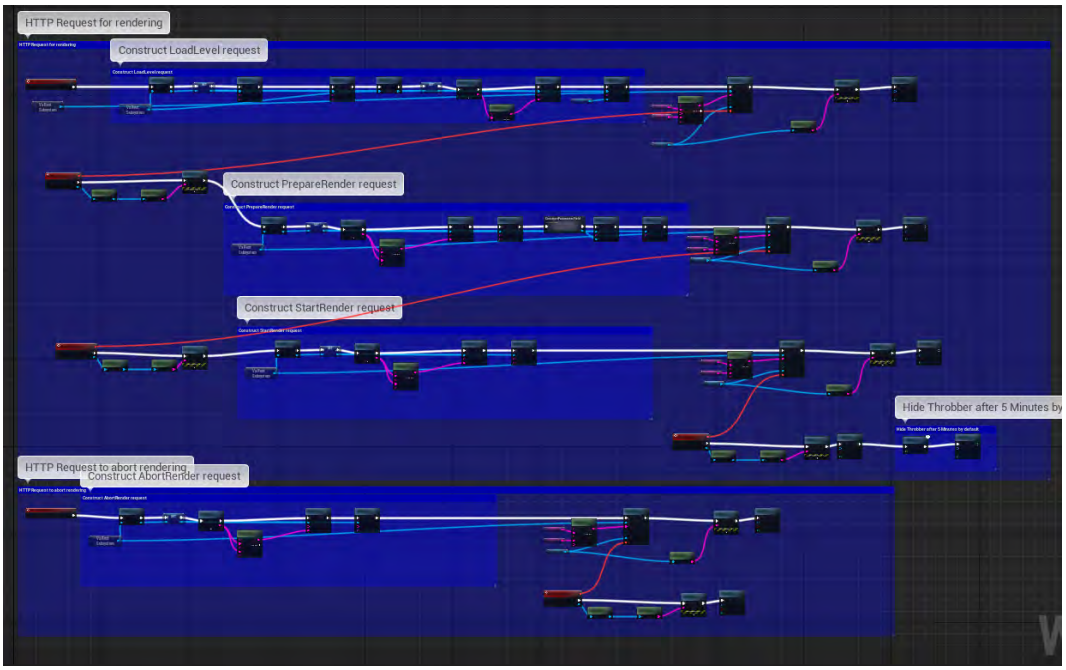


Abbildung 5.8: Übersicht über die Blueprint-Nodes, mit denen die HTTP-Requests zum Starten und Abbrechen von Renderings konstruiert und gesendet werden.

5.4.2 Web Remote Control

In der ConfiguratorUI befinden sich unten rechts ein- und ausblendbare Buttons zum Starten und Abbrechen des Renderings (Siehe Abbildung 5.7). Beim Drücken dieser Buttons werden in der ConfiguratorUI-Blueprint HTTP-PUT-Requests mit Hilfe des VaRest-Plugins erstellt und an die in den Einstellungen festgelegte IP-Adresse mit Port gesendet. An Abbildung 5.8 lässt sich dieser Ablauf anhand der Nodes nachvollziehen. Nachfolgend werden Beispiele für die Bodies gezeigt, die von der Blueprint konstruiert werden. Alle Requests werden als PUT-Request an die URL für Funktionsaufrufe gesendet. Auf dem lokalen System also an `http://localhost:8080/remote/object/call`. Beim Drücken auf „Start Rendering“ wird zunächst eine PUT-Request gesendet, mit der das Level geladen wird. Zum Laden des Warehouse-Levels sieht deren Body zum Beispiel wie folgt aus:

```
{
    {
        "objectPath":"/Script/EditorScriptingUtilities.
    Default__EditorLevelLibrary",
        "functionName":"LoadLevel",
        "parameters": {
            "AssetPath":"/Game/Levels/Warehouse"
        }
    }
}
```

Wurde diese Request erfolgreich gesendet, wird die Request zum Ausführen der PrepareRender-Funktion des EditorRenderActors erstellt und gesendet, deren Body wie folgt aussehen kann:

```
{
    "objectPath":"/Game/Levels/Studio.Studio:PersistentLevel.
    EditorRenderActor",
    "functionName":"PrepareRender",
    "parameters": {
        "CameraLocation": {
            "X":0.0,
            "Y":0.0,
            "Z":0.0
        },
        "CameraRotation": {
            "Pitch":90,
            "Yaw":0,
            "Roll":0
        },
        "ActiveCarIndex":0,
        "ActiveMaterialIndices": [2,1],
        "ActiveMeshIndices": []
    },
    "generateTransaction":true
}
```

Abschließend wird die Request, die die StartRender-Funktion im EditorRenderActor aufruft, gesendet. Dafür sieht der Body zum Beispiel wie folgt aus, wobei sich nur der Levelname im „objectPath“ – abhängig vom aktuellen Level – ändert:

```
{
    "objectPath":"/Game/Levels/Studio.Studio:PersistentLevel.
    EditorRenderActor",
    "functionName":"StartRender"
}
```

Die Requests zum Abbrechen des Renderings, welche beim Drücken des „Abort Renderings“-Buttons gesendet wird, sieht nahezu identisch aus. Sie ruft jedoch statt „StartRender“ die Funktion „AbortRender“ auf.

Kapitel 6

Vergleich der Workflows zur Erstellung von Offline- und Real-Time-Renderings

6.1 Einleitung

In diesem Kapitel werden vier Workflows verglichen, mit denen Offline- und Real-Time-Renderings in Maya und der Unreal Engine mit V-Ray erstellt werden können. Dabei sollen Unterschiede und Gemeinsamkeiten im Shading- und Rendering-Prozess aufgezeigt und herausgefunden werden, wie ein Plugin wie V-Ray for Unreal bei der gemeinsamen Erstellung von Offline- und Real-Time-Renderings hilft. Ziel jedes Workflows soll es sein, ein Bild eines Auto-Modells zu erstellen und das Auto in einer Real-Time-Anwendung darzustellen. Die vier durchzuführenden Workflows wurden bereits in Tabelle 1.1 dargestellt. Der vierte Workflow entspricht jedoch dem Unreal Engine-Teil des ersten Workflows und wird deshalb nicht erneut durchgeführt. Stattdessen findet eine Durchführung der Workflows wie in Tabelle 6.1 statt.

Die Workflows werden für drei unterschiedliche Auto-Modelle durchgeführt, um Ergebnisse zu erhalten, die möglichst unabhängig vom Modell sind. Die Modelle sind in Abbildung 6.1 zu sehen.

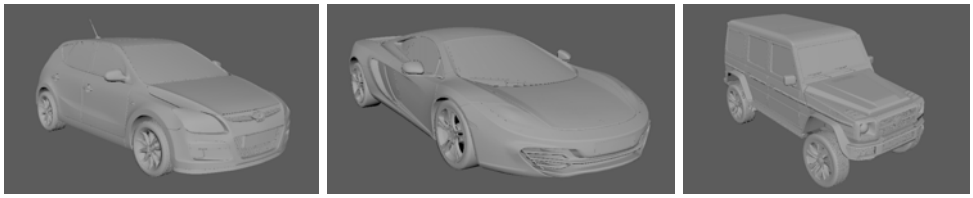


Abbildung 6.1: Die drei Auto-Modelle, die für den Workflow-Vergleich verwendet werden, mit dem Standard-Shader im Maya-Viewport. Von links nach rechts: Hyundai i30, McLaren MP4-12C, Mercedes-Benz G-Klasse Brabus

Tabelle 6.1: Durchzuführende Workflows

Workflow-Nummer	Maya mit V-Ray	Unreal Engine 4 mit V-Ray	Unreal Engine 4 ohne V-Ray
1	V-Ray Shading und V-Ray-Offline-Rendering	-	Unreal Shading und Real-Time-Rendering als Einzelbild (entspricht Workflow 4)
2	V-Ray Shading	Import des V-Ray-geshadeten Modells, V-Ray-Offline-Rendering und Verwendung der V-Ray-Shader für Real-Time	-
3	-	V-Ray-Shading, V-Ray-Offline-Rendering und Verwendung der V-Ray-Shader für Real-Time	-
4	-	-	Unreal Shading und Real-Time-Rendering als Einzelbild (wird in Workflow 1 durchgeführt)

6.2 Vorbereitung

Vor Beginn der einzelnen Workflows wurde zunächst die Studio-Szene als V-Ray-Scene (.vrscene) exportiert, um auch in Maya für Vergleich-Renderings verwendet zu werden. Allerdings funktionierte das Importieren nicht richtig; die meisten Polygone waren nicht mehr vorhanden und Materialien fehlten. Stattdessen wurde das Level als FBX-Datei exportiert, in Maya importiert und Materialien und Lichter für V-Ray nachgebaut. In der importierten Szene wurden alle nicht weiter benötigten Unreal-Objekte entfernt (Collision-Meshes, Reflection Captures, Level-Settings).

Für die V-Ray-Lichter in der Unreal Engine konnte die zu verwendende Einheit nicht eingestellt werden. In Maya wurde für das VRayRectLight Lumen verwendet. Das in der Unreal Engine verwendete VRayPBRMaterial ist allerdings nicht in Maya vorhanden. Stattdessen wurden die Materialien in der Unreal Engine durch neue, ähnlich aussehende VRayMtl ersetzt, die in Maya nachgebaut werden konnten. Dabei wurde das *GGX Modell* für Reflektionen eingestellt und ein *GGX Tail Falloff* von fünf verwendet, um den Materialien in der

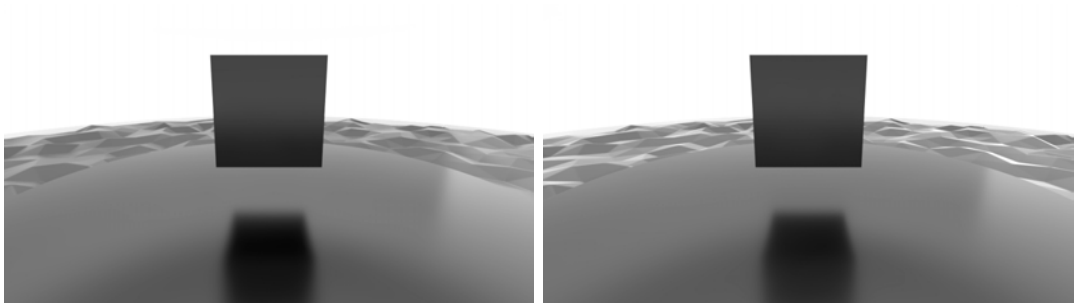


Abbildung 6.2: V-Ray-Renderings vom Studio-Level links aus Maya, rechts aus der Unreal Engine.

Unreal Engine zu entsprechen. Außerdem müssen die Normal Maps im Tangent Space verwendet werden. Abbildung 6.2 zeigt V-Ray-Renderings vom Studio-Level, die in Maya und der Unreal Engine erstellt wurden. Es fällt auf, dass die Reflexionen auf dem Boden in der Unreal Engine heller sind als in Maya.

Zusätzlich wurden die Workflows mit Primitiven 3D-Objekten getestet. Dabei wurden folgende Erfahrungen gemacht:

- VRayMtl hat in der Unreal Engine im Gegensatz zu Maya keinen Metalness Parameter. In Maya erstellte Materilien müssen zur Darstellung von metallischen Oberflächen den Fresnel IOR Parameter verwenden, damit sie von V-Ray for Unreal importiert werden können.
- Maya Nodes, wie zum Beispiel „reverse“ werden nicht von V-Ray for Unreal mit importiert. Selbes gilt für Anpassungen an der Textur-Node wie zum Beispiel „invert“. Anpassungen wie das Invertieren von Roughness-Texturen müssen folglich direkt an der Textur-Datei vorgenommen werden.

Für das Shading in der Unreal Engine wurden Materialien für allgemeine Oberflächen und Glas erstellt. Für Autolack wurde das Material-Packet „Automotive Materials“ importiert[1] und daraus das Material „M_CarPaint_Master“ verwendet.

Für allgemeine Oberflächen ermöglicht das erstellte **MasterMaterial** diffuse Farbe, Licht-Emission, Metallic, Normal-Map, Opazität, Roughness und Specular mit Farben, Texturen oder Skalaren anzupassen. Außerdem kann Tiling und Rotation der Texturen eingestellt werden. Abbildung 6.3 zeigt links die mögliche Parameter in einer Material Instance von **MasterMaterial**. **M_MasterGlass** ermöglicht zusätzlich noch die Verwendung von Refraktion mit Parametern für Index of Refraction, Fresnel Falloff und Frosted Glass Blur, aber ohne Specular-Textur (siehe Abbildung 6.3 rechts).

[1] https://docs.unrealengine.com/en-US/Resources/EpicGamesContent/AutomotiveMaterialPack/index.html

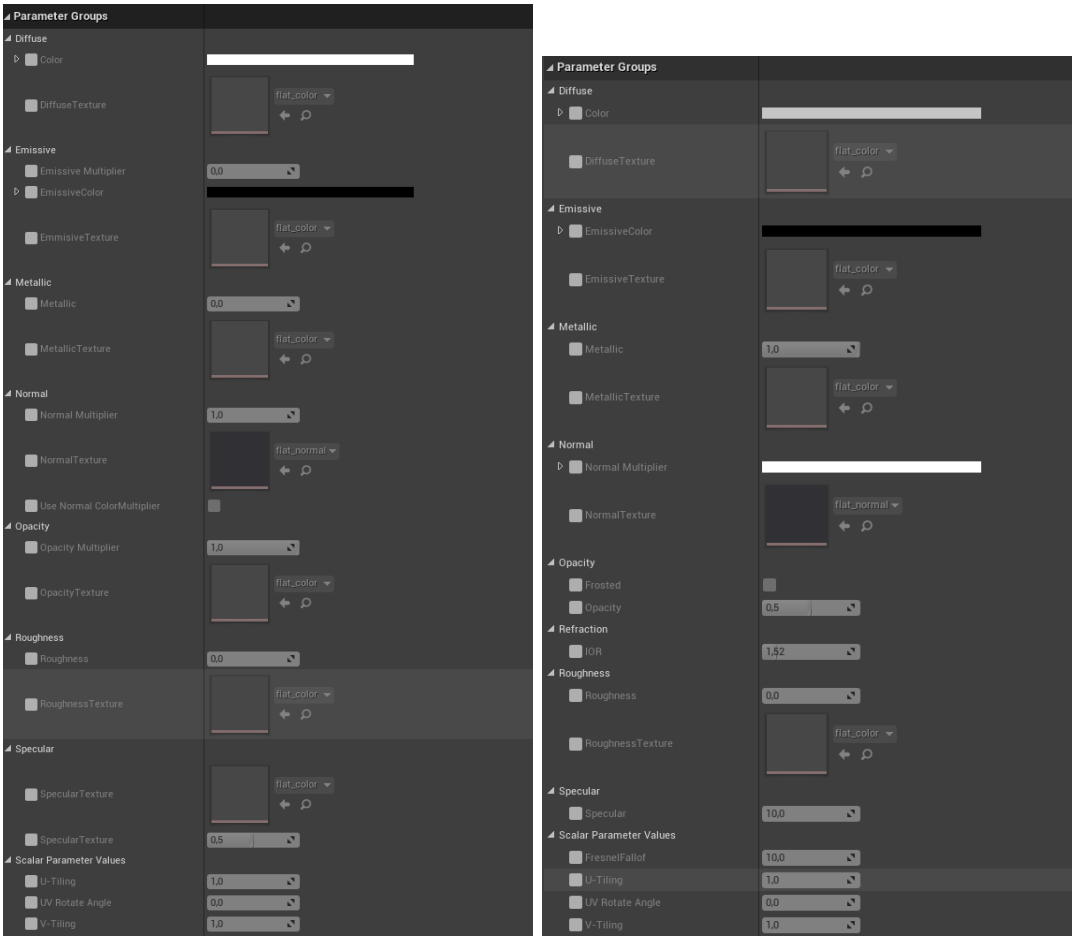


Abbildung 6.3: Die Parameter der Material Instances von MasterMaterial (links) und M_MasterGlass (rechts).

6.3 Durchführung der Workflows

Unabhängig vom Workflow waren grundlegende Vorarbeiten am Modell nötig, damit es sowohl für V-Ray-Renderings als auch für die Verwendung im Konfigurator geeignet ist. Dazu gehört das Sortieren von Meshes in Gruppen, abhängig vom Material, sodass die Materialzuweisung schnell möglich ist, das Erstellen oder Verbessern von UV-Maps, Cleaning des Meshes, sowie Testen, ob das Mesh korrekt in der Unreal Engine dargestellt wird.

Für den **ersten Workflow** wird das Auto-Modell zunächst in Maya für V-Ray geshadet und gerendert. Dabei können jegliche Nodes für Materialien verwendet werden. Beim Shaden für die Darstellung im Konfigurator sind in der Unreal Engine ebenfalls keine Einschränkungen vorhanden. Allerdings sollten die Materialien in V-Ray und in der Unreal Engine gleich aussehen, wodurch beim Shading in der Unreal Engine ein klares Ziel gegeben ist. Es ist aber zu beachten, dass in der Praxis meist eine Vorgabe zum Beispiel für den Lack existiert,

Abbildung 6.4: Links das V-Ray-Rendering des ersten Workflows aus Maya und rechts das Rendering aus der Unreal Engine.

die in beiden Darstellungsformen erreicht werden sollte. Abbildung 6.4 zeigt die Renderings des ersten Workflows für das McLaren-Modell.

Beim **zweiten Workflow** ist der Export/Import von V-Ray for Maya zu V-Ray for Unreal zu beachten. Materialien sollen in Maya so erstellt werden, dass sie problemlos in die Unreal Engine importiert werden können. Wie bereits im vorigen Unterkapitel 6.2 erwähnt, muss für metallische Materialien der Fresnel IOR verwendet, Texturen müssen ohne Nodes, also direkt in ihrer Datei, angepasst und auf die Verwendung von weiteren Nodes muss verzichtet werden.

Export und vor allen der Import von vrscenes können teilweise sehr lange dauern. Zusätzlich unterscheiden sich die Materialien noch nach dem Import und müssen angepasst werden. Die Materialien VRayCarPaintMtl in Maya und VRayCarPaintUberMtl in der Unreal Engine unterscheiden sich zu stark. In Maya können Grund-Reflexion und -Glossiness, einige Parameter für Flakes, sowie Stärke und Glossiness des Coats eingestellt werden. In der Unreal Engine hingegen gibt es nur die Einstellungen dafür, wie metallisch und reflektierend die Base ist und wie stark und reflektierend der Coat sein soll (siehe Abbildung 6.5). Außerdem unterscheiden sich auch refraktive Materialien nach dem Import deutlich vom Original. Weiterhin rendert ein importiertes VRayMtl mit einer Opacity-Textur standardmäßig trotzdem komplett undurchsichtig, solange der Parameter „OpacityTex_AlphaFromIntensity“ nicht eingeschaltet wurde.

Abbildung 6.6 zeigt ein V-Ray-Rendering aus Maya neben in der Unreal Engine erstellten V-Ray-Renderings, davon eines direkt nach dem Import und eines mit Anpassungen, um den Original ähnlicher zu sein. Das importierte Auto-Lack-Material ist ohne Anpassungen vor allen zu Matt. Außerdem wird das Glas mit weißem Fog dargestellt, da das VRayMtl in der Unreal Engine keinen Parameter wie „Diffuse Amount“ in Maya hat, wird die „Diffuse Color“ für den diffusen Anteil verwendet. Damit transparente Materialeen in der Unreal Engine den in Maya erstellten Materialien entsprechen, müsste eigentlich „Diffuse Amount“ mit „Diffuse Color“ multipliziert werden oder am besten bereits in Maya der diffuse Anteil nur über „Diffuse Color“ eingestellt werden. In Abbildung 6.7 sind die Renderings der Unreal Engine für diesen Workflow zu sehen.

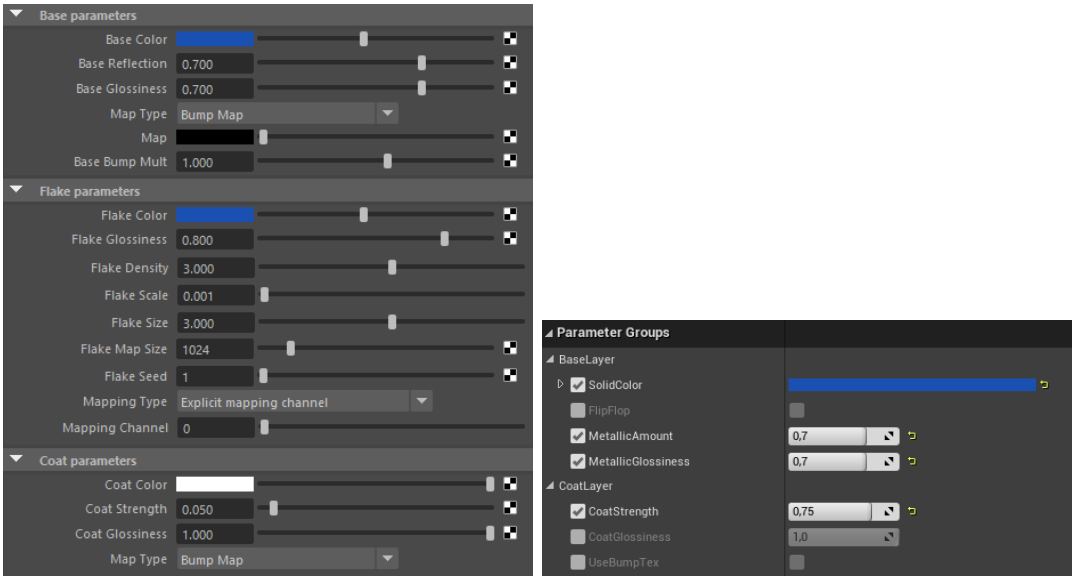


Abbildung 6.5: Links die Parameter eines VRayCarPaintMtl in Maya und rechts eines VRayCarPaintUberMtl in der Unreal Engine.

Abbildung 6.6: Links ein V-Ray-Rendering aus Maya. Daneben das V-Ray-Rendering direkt nach dem Import in der Unreal Engine, gefolgt von einem V-Ray-Rendering aus der Unreal Engine mit Anpassungen.

Abbildung 6.7: Links das Unreal Engine Rendering des zweiten Workflows ohne und rechts mit Anpassungen.

Abbildung 6.8: Links das Unreal Engine Rendering und rechts das V-Ray-Rendering des dritten Workflows.

Der **dritte Workflow** erlaubt das Verwenden von Nodes für Unreal Materialien, solange V-Ray for Unreal diese auch unterstützt (siehe Kapitel 3, Unterkapitel 3.3). Parameter von Instanzen der in V-Ray for Unreal enthaltenen Materialien können aber ohne weitere Einschränkungen angepasst werden.

Da das Material gleichzeitig sowohl für Real-Time- als auch für Offline-Renderings erstellt wird, ist die Optik der Materialien bei beiden Rendertechniken auch vergleichbar, wie in Abbildung 6.8 zu sehen ist.

Für den **vierten Workflow**, der als Teil des ersten Workflows durchgeführt wurde, können alle Möglichkeiten zur Material-Erstellung der Unreal Engine ausgenutzt werden. Das Real-Time-Rendering für diesen Workflow war bereits in Abbildung 6.4 rechts zu sehen.

Die Workflows wurden für die Auto-Modelle „Hyundai i30“ und „McLaren MP4-12C“ in der Reihenfolge erster, zweiter, dritter Workflow und für den „Mercedes-Benz G-Klasse Brabus“ in umgekehrter Reihenfolge durchgeführt. Denn beim Durchführen der Workflows fiel auf, dass durch vorherige Workflows bereits grob bekannt war, welche Parameter einzustellen waren, wodurch weniger ausprobiert wurde. Die folgende Tabelle 6.2 zeigt die für jeden Teil eines Workflows und zugehörigen Renderings benötigte Zeit pro Auto-Modell und die Durchschnittszeit für alle drei Modelle. Die Renderings wurden alle in der Auflösung 1280x720 Pixel erstellt, für V-Ray in der Unreal Engine das Preset 3 „Balanced“ mit dem V-Ray Denoiser verwendet. Die Rendereinstellungen in Maya wurden auf dieselben Werte gesetzt. Der zum Rendern verwendete Computer hatte einen AMD Ryzen 7 1700 Prozessor (acht Kerne mit 3,7 GHz), 32 GB Arbeitsspeicher (in Dual Channel mit 2933 MHz) und eine Nvidia GTX 1080 Grafikkarte.

Tabelle 6.2: Workflow-Zeiten

Aufgabe	**Zeit Hyundai**	**Zeit McLaren**	**Zeit Brabus**	**Durchschnittszeit**
Vorarbeit	02:30:00	01:30:00	01:45:00	01:55:00
Workflow 1	02:55:52	01:47:19	00:53:22	**01:52:11**
V-Ray Shading in Maya	01:40:00	01:13:00	00:26:00	01:06:20
V-Ray Rendering in Maya	00:05:52	00:03:19	00:04:22	00:04:31
Shading in UE4 (**Workflow 4**)	01:10:00	00:31:00	00:23:00	**00:41:20**
Workflow 2	02:18:53	01:09:25	01:07:25	**01:31:54**
V-Ray Shading in Maya	01:10:00	00:37:00	00:41:00	00:49:20
V-Ray Rendering in Maya	00:05:59	00:03:01	00:04:59	00:04:40
V-Ray Import und Shader-Anpassungen in UE4	00:58:00	00:27:00	00:17:00	00:34:00
V-Ray Rendering in UE4	00:04:54	00:02:24	00:04:26	00:03:55
Workflow 3	00:51:03	00:32:51	00:33:29	**00:39:08**
V-Ray Shading in UE4	00:44:00	00:29:00	00:28:00	00:33:40
V-Ray Rendering in UE4	00:07:03	00:03:51	00:05:29	00:05:28

6.4 Vergleich der Workflows

Durch die Durchführung zeigte sich, dass Workflows mit nur einem Shading-Prozess, also Workflow drei und vier deutlich schneller durchführbar sind. Dadurch lässt sich dann auch die Vermutung aufstellen, dass der zweite Workflow ähnlich schnell wäre. Denn für diesen müsste nur in Maya geshadet und die Materialien in die Unreal Engine importiert werden. Allerdings sind noch einige weitere Anpassungen nötig, damit die Materialien vergleichbar aussehen, sodass ein zweiter, weniger umfangreicher Shading-Prozess durchgeführt werden muss.

Zusätzlich schränkt der zweite Workflow die Möglichkeiten beim Shading deutlich ein, denn Materialien müssen so erstellt werden, dass sie problemlos importiert werden können.

Auch der dritte Workflow ist insofern bei der Material-Erstellung einschränkend, wie V-Ray for Unreal Nodes unterstüzt. Allerdings sind diese Einschränkungen weniger umfangreich.

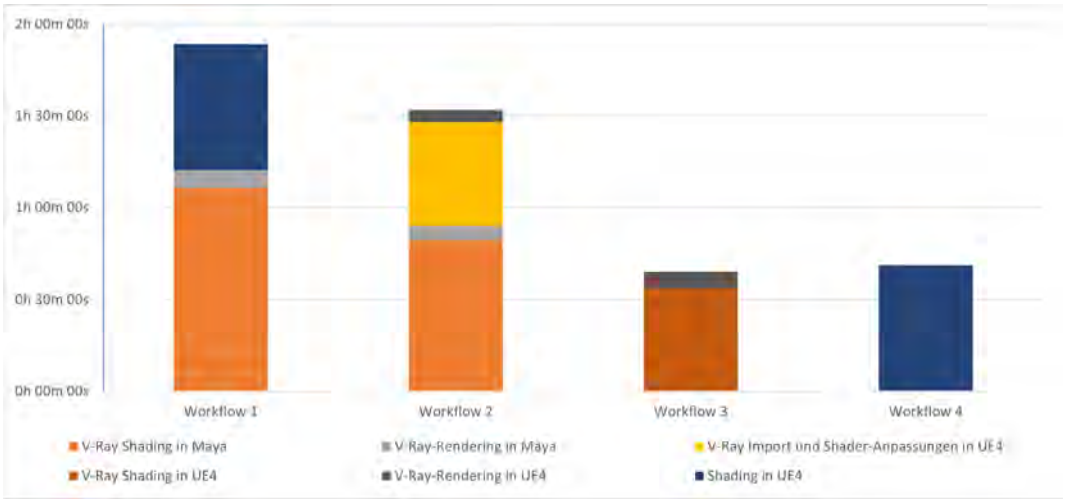


Abbildung 6.9: Die Durchschnittszeiten in Form eines Säulendiagramms

Die Durschnittszeiten der Workflows sind in Abbildung 6.9 als gestapeltes Säulendiagramm dargestellt. Dabei wurde die Zeit für Vorarbeiten nicht mit einbezogen, da sie für alle Workflows gleich sein sollte. Anhand des Diagramms wird deutlich, wie viel Zeit Workflow drei und vier im Vergleich zu den anderen beiden einsparen können.

Kapitel 7

Analyse von Offline- und Real-Time-Renderings

In diesem Kapitel wird zunächst dargestellt, wie V-Ray- und Real-Time-Renderings im Unreal Editor mit den zuvor importierten Auto-Modellen erstellt wurden. Dabei wurden ein V-Ray-Rendering und zwei Real-Time-Renderings von einem Auto-Modell berechnet. Für die Real-Time Renderings wurde einmal Rasterization und einmal Real-Time Ray Tracing verwendet. Anschließend werden diese Renderings analysiert und verglichen.

7.1 Erstellung der Renderings

Die Renderings, welche nachfolgend verglichen werden, wurden in einem simplen Level erstellt, das in Abbildung 7.1 zu sehen ist. Die Renderings entstehen mit der linken V-Ray-Kamera und zeigen das Modell des *McLaren MP4-12C*. Zur Beleuchtung dient ein *VRay-*

Abbildung 7.1: Level, in dem die Renderings zum Vergleich erstellt werden.

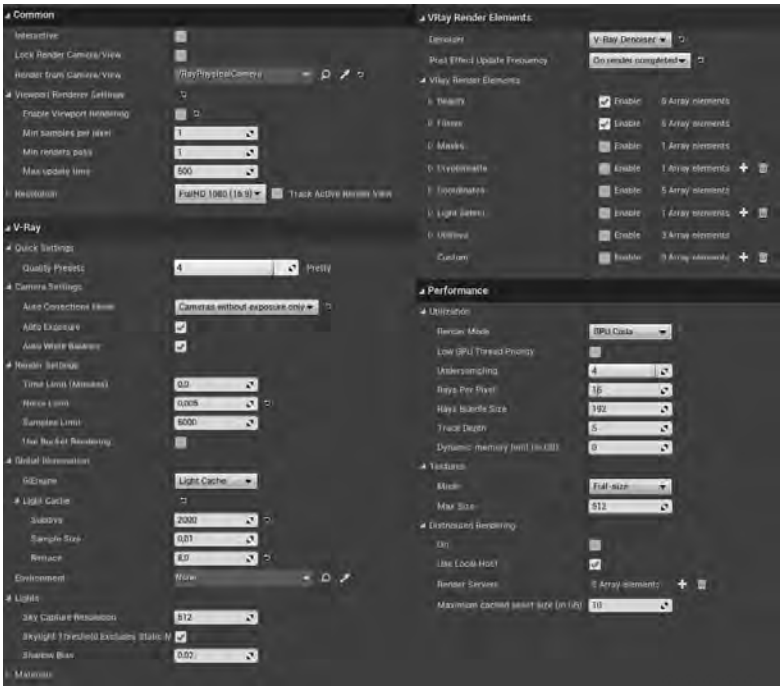


Abbildung 7.2: Die verwendeten V-Ray Rendereinstellungen.

LightDome mit HDRi und ein Directional Light. Ein *Post Process Volume* ist beim Modell des McLarens zu sehen. Dieses wirkt auf das gesamte Level und dient zum Aktivieren der meisten Real-Time Ray Tracing Optionen.

Für das V-Ray-Rendering werden die in Abbildung 7.2 zu sehenden Einstellungen verwendet. Diese entsprechen dem Preset 4 „Pretty“ mit dem V-Ray Denoiser aktiviert. Beim Rendering werden zusätzlich die Render Elements aus den Gruppen „Beauty“ (Background, GI, Lighting, Reflection, Refraction, Specular, Self Illumination und Sub Surface Scatter) und „Filters“ (Diffuse Filter, Reflection Filter, Refraction Filter, Shadows und Atmosphere) mit berechnet, um sie für die Analyse verwenden zu können.

Für das mit Rasterung berechnete Real-Time-Rendering der Unreal Engine 4 wurden grundlegend die Standard-Einstellungen mit dem Qualitäts-Level auf „Cinematic“ verwendet. Für das Anti-Aliasing dient dabei Temporales Anti-Aliasing (TAA). Einzige für das Rendering relevante Abweichung von den Standard-Einstellungen ist das Standard Render Hardware Interface (Default RHI), welches auf *DirectX 12* geändert wurde. Dies ist zur Verwendung von Real-Time Ray Tracing zwingend notwendig. Beim Real-Time-Rendering mit Ray Tracing sind die entsprechenden Einstellungen im *Post Process Volume* aktiviert. Abbildung 7.3 zeigt diese und welche Anpassungen daran vorgenommen wurden. Der Parameter „Refraction“ wurde aufgrund der unerwarteten Ergebnisse, die in Abbildung 7.4 zu sehen sind, deaktiviert.

Standardmäßig ist in Skylights die Option „Ray Traced Shadows“ deaktiviert, wodurch Schatten vom Skylight weiterhin über Shadow Maps berechnet werden. Der *VRayLightDome* erstellt eine Skylight-Komponente zur Beleuchtung in Echtzeit, die nicht direkt vom Level-

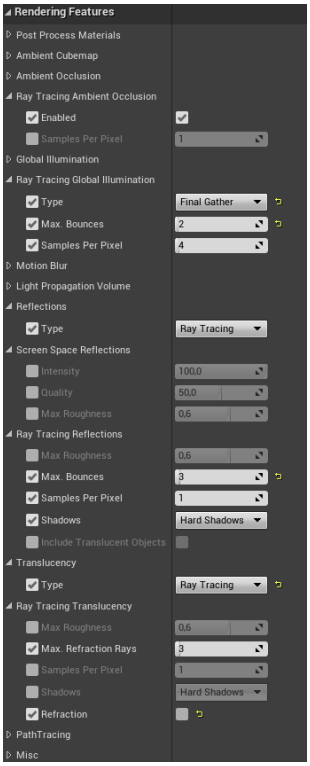


Abbildung 7.3: Die Einstellungen des Post Process Volumes für das Real-Time Ray Tracing. Gelbe Kennzeichnen Abweichungen von den Standardeinstellungen.

Abbildung 7.4: Links ein Real-Time-Rendering mit Ray Tracing mit dem Parameter „Refraction“ aktiviert und rechts deaktiviert.

Editor aus angepasst werden kann. Damit „Ray Traced Shadows“ für diesen im Level-Editor aktiviert werden können, wurde die VRayLightDome-Blueprint um einen Parameter „Ray Traced Shadows“ erweitert. Diese ändert den gleichnamigen Parameter in der Skylight-Komponente, sodass alle Schatten per Ray Tracing berechnet werden können.

Abbildung 7.5: Das V-Ray-Renderings des McLaren-Modells.

Abbildung 7.6: Ein Real-Time-Rendering des McLaren-Modells, das mit Rasterization berechnet wurde.

7.2 Vergleich und Analyse der Renderings

Die Abbildungen 7.5, 7.6 und 7.7 zeigen die erstellten Renderings.

Alle nicht-transparenten Materialien werden zwischen den verschiedenen Techniken sehr ähnlich dargestellt. Im direkten Vergleich sind vor allen Unterschiede bei der Darstellung von Schatten und transparenten beziehungsweise refraktierenden Materialien zu erkennen.

Im V-Ray-Rendering (Abbildung 7.5) und Real-Time-Rendering mit Ray Tracing (Abbildung 7.7) ist im Schatten des Autos auf dem Boden ein Verlauf zwischen dem Kernschatten

Abbildung 7.7: Ein Real-Time-Rendering des McLaren-Modells, das mit Ray Tracing berechnet wurde.

Abbildung 7.8: Von Links nach Rechts: „Lighting“ Render Element des V-Ray Renderings, „Lighting Only“ vom gerasterten Unreal Engine-Rendering und „Lighting Only“ vom Unreal Engine-Rendering mit Ray Tracing.

und dem umliegenden Halbschatten zu sehen. Beim gerasterten Real-Time-Rendering (Abbildung 7.6), welches Schatten mit Shadow Maps berechnete, fehlt dieser Verlauf hingegen. Deutlicher werden die Schatten in Abbildung 7.8 dargestellt, in denen der „Lighting Only“ View Mode der Unreal Engine und das Render Element „Lighting“ von V-Ray zu sehen sind. Der Kernschatten unter dem Auto wird vom *VRayLightDome* und seinem *Skylight* erzeugt, während der umliegende Schatten durch die direktionale Lichtquelle erzeugt wird. Wenn das *Skylight* keine Schatten mit Ray Tracing erzeugen würde, die Option „Ray Traced Shadows“ also deaktiviert wäre, würde der Schatten dem mit Shadow Maps erzeugten Schatten ähnlicher sein. Dies zeigt Abbildung 7.9. Außerdem erzeugen die beiden Renderings mit Ray Tracing detailliertere und genauere Schatten, was vor allen am Kühlergrill vorne oder am Schatten des Seitenspiegels zu erkennen ist. Die Schatten des Real-Time-Renderings mit Ray Tracing fallen jedoch im Vergleich zum V-Ray-Rendering deutlich heller und weniger Kontrastreich aus. Im Falle von statischen Objekten können vergleichbar dunkle, kontrast-

Abbildung 7.9: Das Real-Time-Rendering des McLaren-Modells mit Ray Tracing und „Ray Traced Shadows“ für das Skylight deaktiviert.

Abbildung 7.10: Gerastertes Real-Time-Rendering mit Lightmaps für Schatten.

reiche Schatten für Real-Time-Renderings mit Lightmaps erzeugt werden, was in Abbildung 7.10 zu sehen ist. In dieser Abbildung sind die Schatten auf dem Auto-Modell nicht besonders gut, da das Modell nicht für die Verwendung als statisches Objekt vorgesehen wurde. Deswegen wird es als ein einziges Mesh verwendet und mit der maximalen Auflösung für Lightmaps von 4096x4096 Pixeln in der Unreal Engine können Schatten nicht sehr genau auf dem Modell dargestellt werden. Der mit Lightmaps erzeugte Schatten auf dem Boden ähnelt dem V-Ray-Rendering mehr, als die anderen Real-Time-Darstellungen.

Bei der Darstellung von transparenten Objekten und dahinter liegenden Objekten fallen in den Real-Time-Renderings einige Schwächen im Vergleich zum Offline-Rendering von V-Ray auf. Im Rendering mit Real-Time Ray Tracing scheint Licht nicht durch das Glas der Windschutzscheibe oder des Scheinwerfers. Glanzlichter, wie zum Beispiel vor dem Lenkrad, fehlen komplett. Wird aber Ray Tracing für Reflections deaktiviert, erscheinen diese Glanzlichter wieder (siehe Abbildung 7.11). Allerdings sind sie weniger intensiv, als im gerasterten Real-Time-Rendering oder im V-Ray-Rendering. Im gerasterten Real-Time-Rendering (Abbildung 7.6) ist ebenfalls im Umfeld des Lenkrad eine Schwäche dieser Darstellung zu erkennen: Reflexionen der transparenten Oberfläche müssen über dahinter liegenden Objekten angezeigt werden. Wie intensiv die Reflexionen dargestellt werden, hängt von der Helligkeit der dahinter liegenden Objekten ab. Je dunkler diese sind, desto deutlicher ist die Reflektion zu erkennen.

Abbildung 7.11: Real-Time-Rendering mit Ray Tracing von der Windschutzscheib Links: Reflexionsberechnung mit Ray Tracing. Rechts: Reflexionsberechnung stattdessen mit Screen Space Reflections.

Zunächst weniger deutliche Unterschiede sind die Reflexionen und Glanzlichter. Die Glanzlichter sind in den Renderings mit Ray-Tracing intensiver, als im gerasterten Real-Time-Rendering. Dies kann zum Beispiel am vorderen Reifen ersichtlich werden.

Reflexionen sind beim V-Ray-Rendering am schärfsten und detailliertesten. Im gerasterten Real-Time-Rendering werden neben den Screen Space Reflections die *Skysphere* in Reflexionen dargestellt. Diese Reflexionen sind nicht überall sehr scharf und können Objekte nicht darstellen, die außerhalb des Screen Space liegen und nicht in Reflection Captures aufgenommen wurden. Die Renderings mit Ray Tracing hingegen können solche Objekte darstellen, wie zum Beispiel das blaue Auto, das auf der Stoßstange in Abbildung 7.12 zu sehen ist.

Ein Problem der Renderings mit Real-Time Ray Tracing kann Rauschen sein, wie auf der Motorhaube in Abbildung 7.12 unten links zu sehen. Rechts davon wurden die Samples pro Pixel für Reflexionen auf 8 anstatt 1 gestellt, wodurch das Rauschen verschwindet. Allerdings verringert sich die Performance dadurch massiv.
Für die Renderings wurde ein Computer mit einem AMD Ryzen 7 1700 Prozessor (8 Kerne mit 3,7 GHz), 32 GB Arbeitsspeicher (Dual Channel bei 2933 MHz) und einer Nvidia GTX 1080 verwendet. Die Grafikkarte erlaubt zwar Real-Time Ray Tracing, ist jedoch nicht dafür ausgelegt und besitzt keine dafür optimierten *RT Cores*. Für die drei zuvor gezeigten Renderings (Abbildungen 7.5, 7.6 und 7.7) waren die in Tabelle 7.1 zu sehenden Renderzeiten erforderlich.

Tabelle 7.1: Renderzeiten und Framerates der verglichenen Renderings

	Zeit pro Bild	**Framerate**
V-Ray	13 min 31 s	-
Unreal Engine Rasterization	10,5 ms	95 fps
Unreal Engine Ray Tracing	137 ms	7,3 fps

Abbildung 7.12: Motorhaube und Stoßstange in den Renderings. Von oben links nach unten rechts: V-Ray-Rendering, gerastertes Real-Time-Rendering, Real-Time-Rendering mit Ray Tracing und Real-Time-Rendering mit Ray Tracing und „Samples per Pixel“ für Reflexionen auf acht.

Kapitel 8

Evaluation zum Vergleich von Offline- und Real-Time-Renderings

8.1 Evaluationsaufbau

Mit der Evaluation zum Vergleich von Offline- und Real-Time-Renderings soll festgestellt werden, wie gut Offline- und Real-Time-Renderings von realen Fotos unterschieden werden können. Außerdem soll herausgefunden werden, welche der beiden Rendertechniken eher für real gehalten wird. Ebenso wurden die Teilnehmer befragt, worauf sie geachtet haben, um ein reales beziehungsweise computergeneriertes Bild zu erkennen.

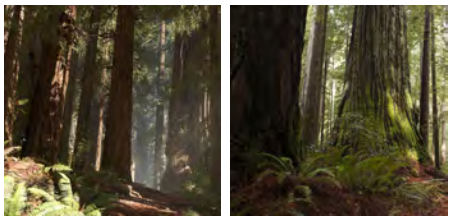
Zu Beginn der Evaluation wird die Befragungsgruppe durch einige Fragen eingeordnet. Dazu wurde das Alter der Personen, wie oft sie sich Filme oder Serien mit Computergraphik-Elementen anschauen und wie oft sie 3D-Videospiele spielen erfragt. Außerdem wurde gefragt, ob Erfahrungen bei der Erstellung von 3D-Grafiken oder 3D-Echtzeit-Anwendungen besteht.

In der Evaluation sollten die Befragten zunächst bei fünf Bilderpaaren auswählen, welches sie für real halten, und bei drei weiteren Bilderpaaren, welches sie für computergeneriert halten. Nachfolgend werden die Bilderpaare in der Form „X.Y“ nummeriert. X kennzeichnet, ob der Befragte das Foto oder das computergenerierte Bild wählen sollte; 1 für Foto und 2 für computergeneriertes Bild. Y ist die Position des Bilderpaares in der jeweiligen Gruppe. Das Bilderpaar 1.3 ist also das dritte Bild, bei dem die Befragten das Foto auswählen sollten. Mit Option 1 wird nachfolgend stets das linke Bild eines Bilderpaares beschrieben und mit Option 2 entsprechend das Rechte. Die Bilderpaare stellen jeweils das gleiche oder zumindest etwas Ähnliches dar. Zwei Bilderpaare (1.3 und 1.4) vergleichen direkt Offline- und Real-Time-Renderings miteinander, während der Befragte davon ausgeht, dass eines der Bilder ein Foto ist. Dadurch soll herausgefunden werden, welche Rendertechnik eher für ein Foto gehalten wird. Auf den Bildern sind folgende Kombinationen von Offline-Renderings, Real-Time-Renderings und Fotografien zu sehen:

- **Bilderpaar 1.1** (Abbildung 8.1a):
  Option 1: V-Ray-Rendering vom *McLaren MP4-12C* aus dem in Kapitel 7 behandelten Level.
  Option 2: Fotografie eines realen *McLaren MP4-12C*[1].
- **Bilderpaar 1.2** (Abbildung 8.1b):
  Option 1: Real-Time-Rendering aus der Frostbite Engine von *Star Wars Battlefront (2015)*[2].
  Option 2: Foto vom Photogrammetrie-Shooting, das für *Star Wars Battlefront (2015)* durchgeführt wurde[3].
- **Bilderpaar 1.3** (Abbildung 8.1c):
  Option 1: Mit V-Ray gerenderte Innenraum-Visualisierung von Pasquale Scionti[4].
  Option 2: Real-Time-Rendering aus der Unreal Engine mit Ray Tracing vom gleichen Raum (mit anderen Materialien und Lichtstimmung)[5].
- **Bilderpaar 1.4** (Abbildung 8.1d):
  Option 1: Gerastertes Real-Time-Rendering des *Hyundai i30* aus der Unreal Engine im Level aus Kapitel 7 erstellt.
  Option 2: V-Ray-Rendering von der selben Kameraperspektive aus.
- **Bilderpaar 1.5** (Abbildung 8.1e):
  Option 1: Foto eines Waldes von Scott Aspinall[6]
  Option 2: In *Blender* erzeugtes Offline-Rendering vom Benutzer „Major4z“ das den Wettbewerb für fotorealistische Renderings von *BlenderGuru* 2012 gewann[7].
- **Bilderpaar 2.1** (Abbildung 8.1f):
  Option 1: Unreal Engine Real-Time-Rendering einer Innenraum-Visualisierung von Pasquale Scionti[8].
  Option 2: Innenraum-Fotografie von Jiri Lizler[9].
- **Bilderpaar 2.2** (Abbildung 8.1g):
  Option 1: V-Ray-Rendering des *Hyundai i30* im Level aus Kapitel 7.
  Option 2: Foto eines realen *Hyundai i30*[10].
- **Bilderpaar 2.3** (Abbildung 8.1h):
  Option 1: Foto von Trent Crawford[11].

[1] **Quelle:** http://www.rikonlondon.com/product/mp4-12c/ – Zuletzt geprüft: 09.10.2020
[2] **Quelle:** https://www.artstation.com/artwork/GElrB – Zuletzt geprüft: 09.10.2020
[3] **Quelle:** https://media.contentapi.ea.com/content/starwars-ea-com/de_DE/starwars/battlefront/news-articles/how-we-used-photogrammetry/jcr:content/body/image_0.img.jpg – Zuletzt geprüft: 09.10.2020
[4] **Quelle:** https://www.artstation.com/artwork/aR9zg9 – Zuletzt geprüft: 09.10.2020
[5] **Quelle:** https://www.artstation.com/artwork/bamR6v– Zuletzt geprüft: 09.10.2020
[6] **Quelle:** https://www.scottaspinall.com/enchanted/ – Zuletzt geprüft: 09.10.2020
[7] **Quelle:** https://www.blenderguru.com/competitions/photorealism-competition-results – Zuletzt geprüft: 09.10.2020
[8] **Quelle:** https://www.artstation.com/artwork/JyWNA – Zuletzt geprüft: 09.10.2020
[9] **Quelle:** https://www.jirilizler.com/hotely-a-resorty/l65y9tr60gsd0rcnnd2mhe0pqvdtrr – Zuletzt geprüft: 09.10.2020
[10] **Quelle:** https://de.auto-data.org/hyundai/hyundai-i30-2007-1-6-crdi-116-hp-dpf/ – Zuletzt geprüft 09.10.2020
[11] **Quelle:** https://www.flickr.com/photos/146919244@N07/33091229730 – Zuletzt geprüft:

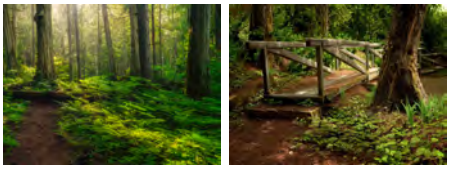
(a) Bilderpaar 1.1



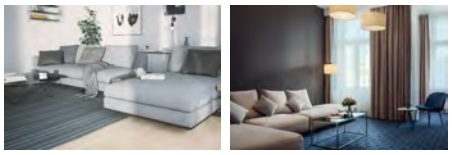
(b) Bilderpaar 1.2

(c) Bilderpaar 1.3

(d) Bilderpaar 1.4

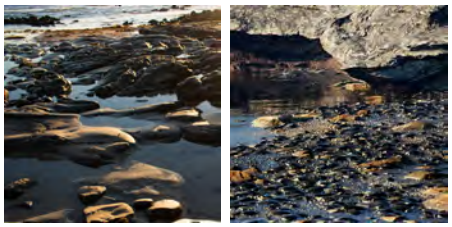


(e) Bilderpaar 1.5



(f) Bilderpaar 2.1

(g) Bilderpaar 2.2



(h) Bilderpaar 2.3

Abbildung 8.1: Bilderpaare bei denen die Befragten das reale (Bilderpaare 1.1 bis 1.5) beziehungsweise computergenerierte Bild (Bilderpaare 2.1 bis 2.3) wählen sollten.

Option 2: Vom *Flickr*-Benutzer „kooooola“ erstelltes Real-Time-Rendering aus der Unreal Engine[12]

09.10.2020

[12] **Quelle:** https://www.flickr.com/photos/99646627@N03/18542824813/ – Zuletzt geprüft: 09.10.2020

(a) Bild 3.1

(b) Bild 3.2

(c) Bild 3.3

Abbildung 8.2: Bilder, die die Befragten nach Realismus- und Schönheitsgrad bewerten sollten.

Anschließend sollten die Befragten drei Bilder jeweils nach Realismusgrad und Schönheitsgrad auf Skalen von eins bis fünf bewerten. Folgende Bilder wurden dabei gezeigt:

- **Bild 3.1** (Abbildung 8.2a): Ein V-Ray-Rendering, wie es mit dem Auto-Konfigurator erstellt werden kann.
- **Bild 3.2** (Abbildung 8.2b): Ein mit der Unreal Engine 5 erstelltes Real-Time-Rendering aus der Unreal Engine 5 Demo [Epi20][13].
- **Bild 3.3** (Abbildung 8.2c): Eine eigene Fotografie.

Mit den Bewertungen von Bild 3.1 soll geprüft werden, für wie realistisch ein vom Auto-Konfigurator erstelltes Offline-Rendering eingeschätzt werden würde. Bei Bild 3.2 hingegen, wie realistisch Real-Time-Renderings der nächsten Game-Engine-Generationen wirken.

[13] **Quelle**: https://www.gamespot.com/images/1300-3670717/ – Zuletzt geprüft: 12.10.2020

Abschließend wurde gefragt, welche Kriterien die Befragten verwendeten, um ein Foto von einem computergenerierten Bild zu unterscheiden. Zur Auswahl standen die folgenden Optionen und es war möglich weitere anzugeben:

- Reflexionen
- Schattenwurf
- Beleuchtung
- Farben
- Formen, Modellqualität (z.B zu scharfe Kanten)
- Refraktion (Verhalten von Licht wenn es durch etwas hindurch scheint, wie z.B. durch Glas)
- Mängel (Dreck, Kratzer, Staub, etc.)

In Anhang A ist der vollständige Fragebogen einsehbar.

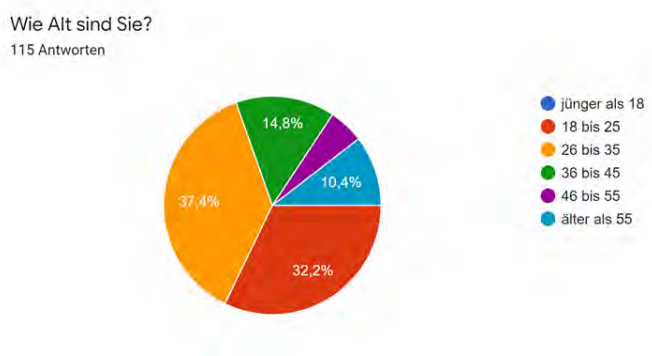


Abbildung 8.3: Diagramm Altersgruppen

8.2 Auswertung

Mit *Google Sheets* wurden die Ergebnisse ausgewertet. Diagramme wurden mit *Google Forms*, *Google Sheets* oder *Microsoft Excel* erstellt. Statistische Tests wurden mit *Past 3*[14] durchgeführt.

8.2.1 Befragungsgruppe

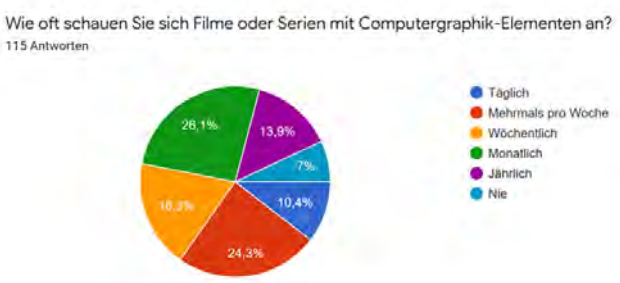
An der Evaluation zum Vergleich von Offline- und Real-Time-Renderings nahmen 115 Personen teil. Abbildung 8.3 zeigt die Aufteilung der Altersgruppen. Das durchschnittliche Alter lag dabei zwischen 26 und 35. Der Medianwert ist die selbe Altersgruppe.

Filme und Serien mit Computergraphik-Elementen schauen die Befragten im Durchschnitt wöchentlich. Dies ist auch der Medianwert (Aufteilung in Abbildung 8.4a).

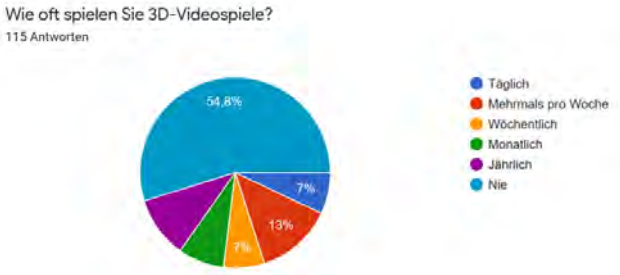
Der durchschnittliche Befragte spielt 3D-Videospiele jährlich. Der Medianwert liegt jedoch bei nie, denn 54,8 Prozent der Befragten spielen nie 3D-Videospiele (Abbildung 8.4b).

Durchschnittlich und auch im Medianwert haben die Befragten keine Erfahrungen mit der Erstellung von 3D-Grafiken oder 3D-Echtzeitanwendungen. Insgesamt 24,35 Prozent der Befragten gaben an Erfahrungen mit der Erstellung von 3D-Grafiken zu haben (Abbildung 8.4c). Nur 12,17 Prozent haben Erfahrung mit der Entwicklung von 3D-Echtzeit-Anwendungen (Siehe Abbildung 8.4d).

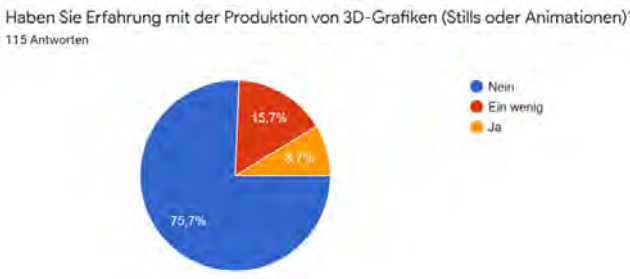
[14] https://palaeo-electronica.org/2001_1/past/issue1_01.htm



(a) Diagramm zu Filme und Serien mit Computergraphik-Elementen



(b) Diagramm zu 3D-Videospielen



(c) Diagramm zu Vorerfahrungen mit der Erstellung von 3D-Grafiken

(d) Diagramm zu Vorerfahrungen mit 3D-Real-Time-Anwendungen

Abbildung 8.4: Diagramme zum Konsum und Erfahrungen mit der Erstellung von Medien mit 3D-Grafik

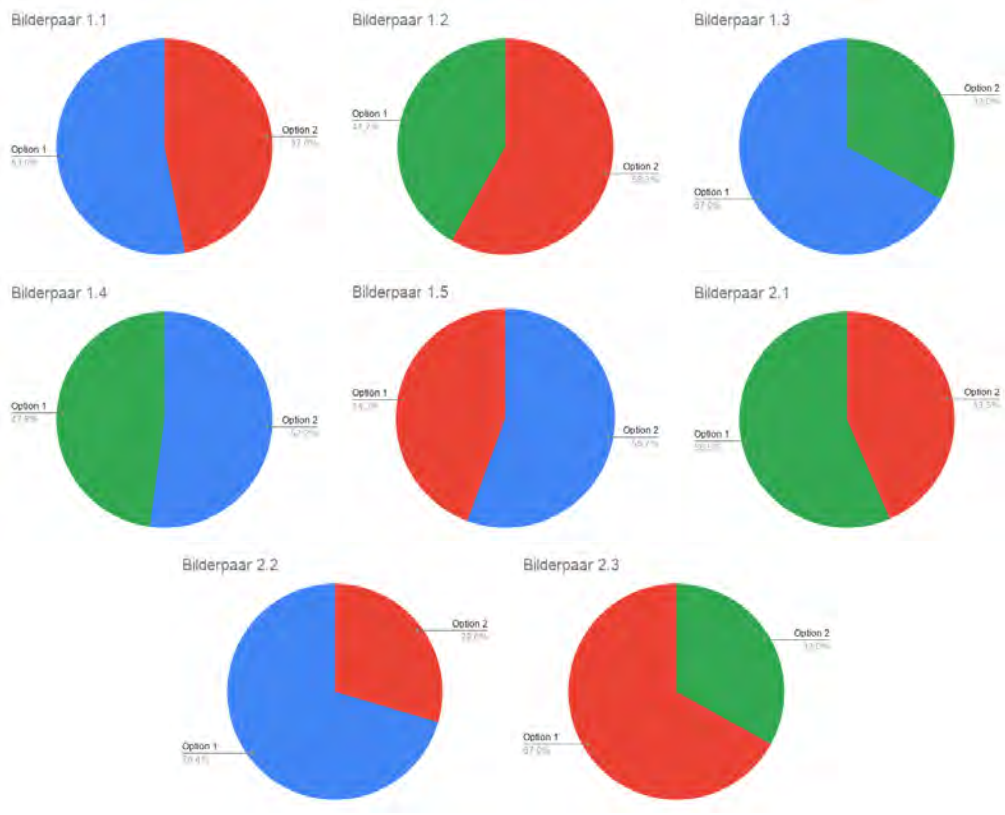


Abbildung 8.5: Kreisdiagramme zu den Vergleichen von Fotos, Real-Time- und Offline-Renderings. Die Farben kennzeichnen dabei die Art des Bildes: Rot für Fotografien, Blau für Offline-Renderings und Grün für Real-Time-Renderings.

8.2.2 Vergleich der Bilderpaare

In Abbildung 8.5 ist in Form von Kreisdiagrammen zu sehen, welche Bilder die Befragten für real (Bilderpaare 1.1 bis 1.5) beziehungsweise für computergenerierte Bilder (Bilderpaare 2.1 bis 2.3) hielten.

Die Bilderpaare lassen sich in drei Gruppen einteilen: Vergleich von Real-Time-Rendering und Foto, Vergleich von Offline-Rendering und Foto, und Vergleich von Real-Time- und Offline-Renderings. Die Bilderpaare 1.2, 2.1 und 2.3 gehören der ersten Gruppen an, 1.1, 1.5 und 2.2 der zweiten Gruppe, und 1.3 und 1.4 der dritten Gruppe. Durch die Einteilung in Gruppen, können durchschnittliche Erkennungsraten der einzelnen Gruppen berechnet werden. Dies ist in Tabelle 8.1 zusammen mit den einzelnen Erkennungsraten der Bilder ablesbar. Eine Erkennungsrate von 50 Prozent ist dabei so zu interpretieren, dass der Realismusgrad der beiden Bilder für die Befragten nicht unterscheidbar war. Das computergenerierte Bild also die gleiche Qualität hat, wie ein Foto. Die durchschnittliche Erkennungsraten weichen nicht mehr als zehn Prozentpunkte von 50 Prozent ab. Somit sind beide Rendertechniken im Durchschnitt nicht deutlich von Fotos zu unterscheiden.

Die Bilderpaare 2.2 und 2.3 weichen deutlicher von einer Erkennungsrate von 50 Prozent ab. Das für Bilderpaar 2.2 erstellte Offline-Rendering ist also nicht realistisch genug, da das reale Bild von 70,43 Prozent der Befragten erkannt wurde. Bei Bilderpaar 2.3 zeigt sich überraschenderweise etwas gegenteiliges: Das Real-Time-Rendering ohne Ray-Tracing wurde nur von 33,04 Prozent der Befragten als computergeneriert erkannt.

Außerdem wird bei Bilderpaar 1.3 das Offline-Rendering eher für das Foto gehalten. Allerdings könnte dies an den sehr unterschiedlichen Beleuchtungsverhältnisse liegen.

Tabelle 8.1: Durchschnittliche Erkennungsraten beim Vergleich von Real-Time-Rendering vs. Foto, Offline-Rendering vs. Foto und Real-Time- vs. Offline-Rendering

		Durchschnittliche Erkennungsrate	**Standardabweichung**
Real-Time vs. Foto	**Erkennungsrate**	49,28%	14,08%
Bilderpaar 1.2	58,26%		
Bilderpaar 2.1	56,52%		
Bilderpaar 2.3	33,04%		
Offline vs. Foto	**Erkennungsrate**	53,91%	14,37%
Bilderpaar 1.1	49,96%		
Bilderpaar 1.5	44,35%		
Bilderpaar 2.2	70,43%		
Real-Time vs. Offline	**Offline für Foto gehalten**	59,57%	10,45%
Bilderpaar 1.3	66,96%		
Bilderpaar 1.4	52,17%		

Tabelle 8.2: Aufteilung von „Erkannt“ und „Nicht Erkannt“ pro Altersgruppe für Real-Time-Renderings im Vergleich mit Fotos

Real-Time vs. Foto	**18 bis 25**	**26 bis 35**	**36 bis 45**	**46 bis 55**	**älter als 55**
Erkannt	64	65	17	9	15
Nicht Erkannt	47	64	34	9	21
Erkennungsrate	57,66%	50,39%	33,33%	50,00%	41,67%

Erkennungsrate in Abhängigkeit vom Alter

Mit den Bildervergleichen wurde des Weiteren überprüft, ob ein Zusammenhang zwischen Alter oder Erfahrungen mit der Erstellung von 3D-Grafiken beziehungsweise 3D-Echtzeit-Anwendungen besteht.

Zur Prüfung auf eine Abhängigkeit zwischen Alter und Erkennungsrate wurden die Antworten für die Bilderpaar-Gruppen an Hand der Altersgruppen aufgeteilt, die durchschnittliche Erkennungsrate je Altersgruppe berechnet und ein χ^2-Test für jede Bilderpaar-Gruppe durchgeführt. Die **Hypothese** dabei ist: Die Erkennungsrate ist abhängig vom Alter am 10%-Niveau (mit höchstens 10% Irrtumswahrscheinlichkeit). Dementsprechend ist die zu widerlegende Nullhypothese H_0: Alter und Erkennungsrate sind unabhängig.

Tabelle 8.2 zeigt für die Bilderpaar-Gruppe „Real-Time-Renderings und Fotos“ die Anzahl der Antworten von „Erkannt“ und „Nicht Erkannt“ pro Altersgruppe, mit denen der χ^2-Test durchgeführt wird. Dazu wurden die Antworten der drei Bilderpaare dieser Gruppe jeweils pro Altersgruppe aufsummiert.

Tabelle 8.3 zeigt die Ergebnisse des χ^2-Tests. Mit einer Wahrscheinlichkeit von p = 0,056 (5,6 Prozent) sind Alter und Erkennungsrate unabhängig. Somit wird H_0 abgelehnt. Beim Vergleich zwischen Real-Time-Renderings und Fotografien besteht also ein Zusammenhang zwischen dem Alter und der Erkennungsrate mit einer Irrtumswahrscheinlichkeit von 5,6 Prozent.

Tabelle 8.3: χ^2-Test zu Abhängigkeit von Alter und Erkennungsrate für Real-Time-Renderings im Vergleich mit Fotos

Chi squared	*Real-Time vs. Foto*		
Rows, columns:	2, 5	**Degrees freedom:**	4
Chi2:	9,2075	**p (no assoc.):**	0,056117
Monte Carlo p:	0,056		

In Tabelle 8.4 ist die Aufteilung von „Erkannt“ und „Nicht Erkannt“ für den Vergleich von Offline-Renderings und Fotos zu sehen. Die Ergebnisse des zugehörigen χ^2-Tests sind in Tabelle 8.5. H_0, also die Unabhängigkeit von Alter und Erkennungsrate, muss hierbei angenommen werden, denn die Wahrscheinlichkeit für Unabhängigkeit liegt mit p = 0,15 (15 Prozent) über dem 10%-Niveau.

Tabelle 8.4: Aufteilung von „Erkannt“ und „Nicht Erkannt“ pro Altersgruppe für Offline-Renderings im Vergleich mit Fotos

Offline vs. Foto	**18 bis 25**	**26 bis 35**	**36 bis 45**	**46 bis 55**	**älter als 55**
Erkannt	59	75	27	12	13
Nicht Erkannt	52	54	24	6	23
Erkennungsrate	53,15%	58,14%	52,94%	66,67%	36,11%

Tabelle 8.5: χ^2-Test zu Abhängigkeit von Alter und Erkennungsrate für Offline-Renderings im Vergleich mit Fotos

Chi squared	*Offline vs. Foto*		
Rows, columns:	2, 5	**Degrees freedom:**	4
Chi2:	6,7425	**p (no assoc.):**	0,15014
Monte Carlo p:	0,1432		

Tabelle 8.6: Aufteilung von „Erkannt“ und „Nicht Erkannt“ pro Altersgruppe für Offline-Renderings im Vergleich mit Real-Time-Renderings

Real-Time vs. Offline	**18 bis 25**	**26 bis 35**	**36 bis 45**	**46 bis 55**	**älter als 55**
Erkannt	44	58	17	5	13
Nicht Erkannt	30	28	17	7	11
Erkennungsrate	59,46%	67,44%	50,00%	41,67%	54,17%

Beim Vergleich von Offline-Renderings mit Real-Time-Renderings (Verteilung in Tabelle 8.6) wird „Offline-Rendering für das Foto halten“ als „Erkannt“ interpretiert. Mit dem Ergebnis des χ^2-Tests (Tabelle 8.7) muss auch hier die Nullhypothese H_0 angenommen werden, denn mit einer Wahrscheinlichkeit von 24,92 Prozent sind Alter und Erkennungsrate hier unabhängig.

Tabelle 8.7: χ^2-Test zu Abhängigkeit von Alter und Erkennungsrate für Offline-Renderings im Vergleich mit Real-Time-Renderings

Chi squared	*Real-Time vs. Offline*		
Rows, columns:	2, 5	**Degrees freedom:**	4
Chi2:	5,3938	**p (no assoc.):**	0,24923
Monte Carlo p:	0,2485		

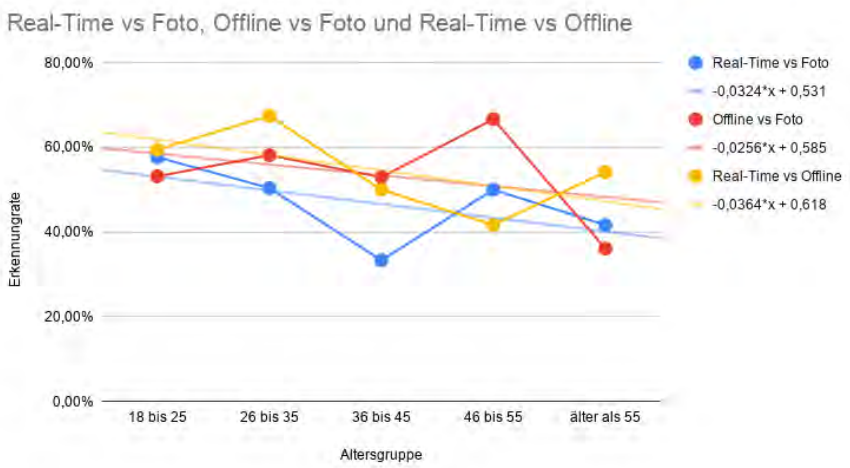


Abbildung 8.6: Verlauf der durchschnittlichen Erkennungsrate in Abhängigkeit vom Alter mit den Trendlinien für jede der Bilderpaar-Gruppen.

In Abbildung 8.6 ist der Verlauf der Erkennungsrate mit dem Alter für die drei Bilderpaar-Gruppen (Real-Time vs. Foto, Offline vs. Foto und Real-Time vs. Offline) dargestellt. Außerdem sind die Trendlinien eingezeichnet, deren Funktionen rechts vom Diagramm stehen. Alle Trendlinien haben ein negative Steigung, somit sinkt die Erkennungsrate mit steigendem Alter. Allerdings kann den vorigen χ^2-Tests zu Folge, nur für den Vergleich von Real-Time-Renderings und Fotos mit einer Irrtumswahrscheinlichkeit von weniger als 10 Prozent eine Abhängigkeit zwischen Alter und Erkennungsrate nachgewiesen werden.

Erkennungsrate in Abhängigkeit von Erfahrungen mit 3D-Grafik oder 3D-Echtzeit-Anwendungen

Tabelle 8.8: Verteilung von „Erkannt“ und „Nicht Erkannt“, abhängig von der Erfahrung bei der Erstellung von Offline-Renderings (CG Erfahrung)

Real-Time vs. Foto	**keine CG Erfahrung**	**Mit CG Erfahrung**
Erkannt	122	48
Nicht Erkannt	139	36
Erkennungsrate	46,74%	57,14%
Offline vs. Foto	**keine CG Erfahrung**	**Mit CG Erfahrung**
Erkannt	132	54
Nicht Erkannt	129	30
Erkennungsrate	50,57%	64,29%
Real-Time vs. Offline	**keine CG Erfahrung**	**Mit CG Erfahrung**
Erkannt	105	32
Nicht Erkannt	69	24
Erkennungsrate	60,34%	57,14%

Das Vorgehen zur Feststellung von Abhängigkeiten zwischen Erfahrung mit der Erstellung von 3D-Grafiken oder 3D-Echtzeit-Anwendungen sieht identisch aus: Die „Erkannt“- und „Nicht Erkannt“-Antworten wurden abhängig von den Erfahrungen für jede Bilderpaar-Gruppe aufsummiert und mit diesen ein χ^2-Test durchgeführt. Tabelle 8.8 zeigt die Verteilungen von „Erkannt“ und „Nicht Erkannt“ für Befragte mit und ohne Erfahrungen zur Erstellung von 3D-Grafiken.

Die **Hypothese** lautet: Die Erkennungsrate ist abhängig von Erfahrungen mit der Erstellung von 3D-Grafiken (Offline-Renderings). Die zu überprüfende Nullhypothese H_0 ist die Gegenhypothese, dass keine Abhängigkeit besteht.

Die Ergebnisse des χ^2-Tests zur Prüfung dieser Hypothese zeigt Tabelle 8.9. Am 10%-Niveau wird H_0 für die Vergleiche von Renderings mit Fotos abgelehnt. Bei Real-Time-Renderings im Vergleich mit Fotos ist die Erkennungsrate mit einer Wahrscheinlichkeit von p=0,0973 (9,73 Prozent) unabhängig von den Erfahrungen. Beim Vergleich von Offline-Renderings mit Fotos sogar nur mit einer Wahrscheinlichkeit von p=0,0283 (2,83 Prozent). Beim Vergleich von Real-Time- mit Offline-Renderings besteht jedoch keine Abhängigkeit zu den Erfahrungen, um ein Offline-Rendering für das Foto zu halten mit p=0,671 (67,1 Prozent) ist die Erkennungsrate unabhängig von den Erfahrungen bei der Erstellung von 3D-Grafiken und somit wird H_0 angenommen.

Das Erkennen von Fotos ist dementsprechend abhängig von Erfahrungen mit dem Erstellen von 3D-Grafiken. Die Erkennungsrate lag bei erfahrenen Befragten höher. Beim Vergleich von Offline- und Real-Time-Rendering machen die Erfahrungen jedoch keinen Unterschied; die Erkennungsraten sind für Befragte mit und ohne Erfahrung mit 3D-Grafik ähnlich.

Tabelle 8.9: Ergebnisse der χ^2-Tests zur Prüfung auf Abhängigkeit zwischen CG Erfahrungen und Erkennungsrate.

Chi squared	*Real-Time vs. Foto*		
Rows, columns:	2, 2	**Degrees freedom:**	1
Chi2:	2,7497	**p (no assoc.):**	0,097274
Monte Carlo p :	0,1035		
Chi squared	*Offline vs. Foto*		
Rows, columns:	2, 2	**Degrees freedom:**	1
Chi2:	4,808	**p (no assoc.):**	0,028327
Monte Carlo p :	0,0306		
Chi squared	*Real-Time vs. Offline*		
Rows, columns:	2, 2	**Degrees freedom:**	1
Chi2:	0,18034	**p (no assoc.):**	0,67108
Monte Carlo p :	0,7562		

Tabelle 8.10: Verteilung von „Erkannt“ und „Nicht Erkannt“, abhängig von der Erfahrung bei der Entwicklung von 3D-Echtzeit-Anwendungen (Real-Time Erfahrung)

Real-Time vs. Foto	**keine Real-Time Erfahrung**	**mit Real-Time Erfahrung**
Erkannt	145	25
Nicht Erkannt	158	17
Erkennungsrate	47,85%	59,52%
Offline vs. Foto	**keine Real-Time Erfahrung**	**mit Real-Time Erfahrung**
Erkannt	154	32
Nicht Erkannt	149	10
Erkennungsrate	50,83%	76,19%
Real-Time vs. Offline	**keine Real-Time Erfahrung**	**mit Real-Time Erfahrung**
Erkannt	114	23
Nicht Erkannt	82	11
Erkennungsrate	58,16%	67,65%

In Tabelle 8.10 sind die Verteilungen in Abhängigkeit von vorliegenden Erfahrungen bei der Entwicklung von 3D-Echtzeit-Anwendungen zu sehen. Von den 14 Befragten mit Real-Time-Erfahrungen gaben im übrigen zwölf an ebenfalls Erfahrungen mit 3D-Grafik zu haben. Die zu prüfende **Hypothese**, mit der Nullhypthese H_0 als Gegenhypothese, lautet hier: Die Erkennungsrate ist abhängig von Erfahrungen bei der Entwicklung von 3D-Echtzeit-Anwendungen. Die Ergebnisse der durchgeführten χ^2-Tests zeigt Tabelle 8.11.

Tabelle 8.11: Ergebnisse der χ^2-Tests zur Prüfung auf Abhängigkeit zwischen Real-Time Erfahrungen und Erkennungsrate.

Chi squared	*Real-Time vs. Foto*		
Rows, columns:	2, 2	**Degrees freedom:**	1
Chi2:	2,0095	**p (no assoc.):**	0,15631
Monte Carlo p :	0,1798		
Chi squared	*Offline vs. Foto*		
Rows, columns:	2, 2	**Degrees freedom:**	1
Chi2:	9,5518	**p (no assoc.):**	0,0019976
Monte Carlo p :	0,0024		
Chi squared	*Real-Time vs. Offline*		
Rows, columns:	2, 2	**Degrees freedom:**	1
Chi2:	1,082	**p (no assoc.):**	0,29825
Monte Carlo p :	0,3559		

Für die Vergleiche Real-Time-Rendering mit Foto und Real-Time- mit Offline-Rendering muss H_0 am 10%-Niveau angenommen werden denn die Wahrscheinlichkeiten für Unabhängigkeit sind mit 15,63 und 29,83 Prozent zu hoch. Nur bei den Vergleichen von Offline-Renderings mit Fotos ist eine Abhängigkeit zwischen Erkennungsrate und Real-Time Erfahrungen mit einer Irrtumswahrscheinlichkeit von p=0,002 (0,2 Prozent) nachweisbar.

Erfahrungen bei der Entwicklung von 3D-Echtzeit-Anwendungen bieten beim Erkennen von Real-Time-Renderings weder beim Vergleich mit Fotos noch mit Offline-Renderings einen Vorteil, im Vergleich zu den unerfahrenen Befragten. Allerdings ist beim Erkennen von Offline-Renderings im Vergleich mit Fotografien eine deutlichere Abhängigkeit vorhanden, als bei den Befragten mit CG Erfahrungen.

8.2.3 Bewertung der Bilder

Nachfolgend werden die Bewertungen der drei Bilder dargestellt, welche nach Realismus- und Schönheitsgrad bewertet werden sollten.

Das arithmetische Mittel für den Realismusgrad von Bild 3.1 liegt bei 3,21 und Median bei 3 mit einer Standardabweichung von 1,14. Der Schönheitsgrad hat für das arithmetische Mittel den Wert 2,37 und einen Median von 2 mit einer Standardabweichung von 1,19. Die Histogramme dazu sind in Abbildung 8.7 zu sehen. Dieses durch den Auto-Konfigurator

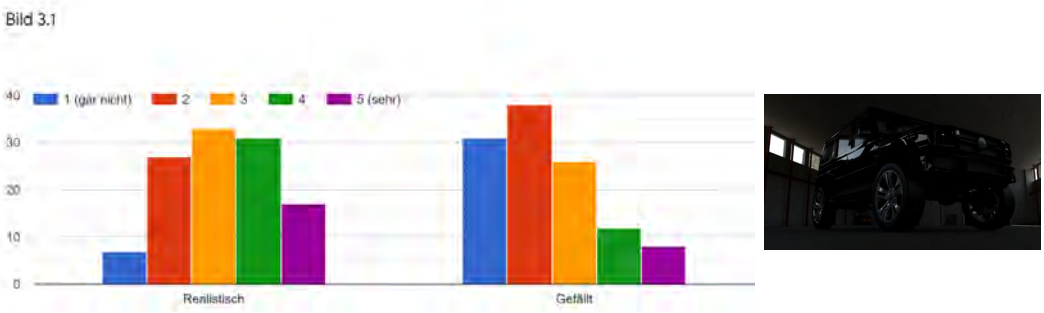


Abbildung 8.7: Die Histogramme für die Bewertungen zum Realismus- und Schönheitsgrad von Bild 3.1 (rechts).

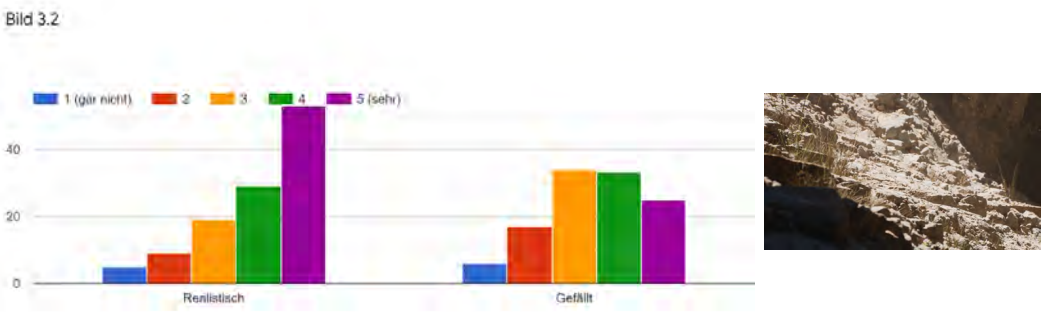


Abbildung 8.8: Die Histogramme für die Bewertungen zum Realismus- und Schönheitsgrad von Bild 3.2 (rechts).

erstellte Rendering hat somit einen mittelmäßigen Realismusgrad und gefiel den Befragten im Durchschnitt nicht.

Bild 3.2 fanden die Befragten im Durchschnitt realitischer, denn das arithmetische Mittel liegt bei 4,01 und der Median bei 4 mit einer Standardabweichung von 1,16. Das arithmetische Mittel für den Schönheitsgrad liegt bei 3,47, Median bei 4 und die Standardabweichung bei 1,14. Somit gefiel dieses Real-Time-Rendering mehr Befragten, als Bild 3.1. Die Histogramme für Bild 3.2 zeigt Abbildung 8.8.

Ähnliche Werte liegen für den Realismusgrad des Fotos (Bild 3.3) vor mit einem arithmetischen Mittel von 3,84 und Median von 4 bei einer Standardabweichung von 1,14. Der Schönheitsgrad liegt geringfügig höher mit dem arithmetischen Mittel von 3,94, Median bei 4 und Standardabweichung 1,05. Abbildung 8.9 zeigt die zugehörigen Histogramme.

Obwohl Bild 3.3 ein Foto ist, wird es im Durchschnitt für etwas unrealistischer als das Real-Time-Rendering Bild 3.2 gehalten.

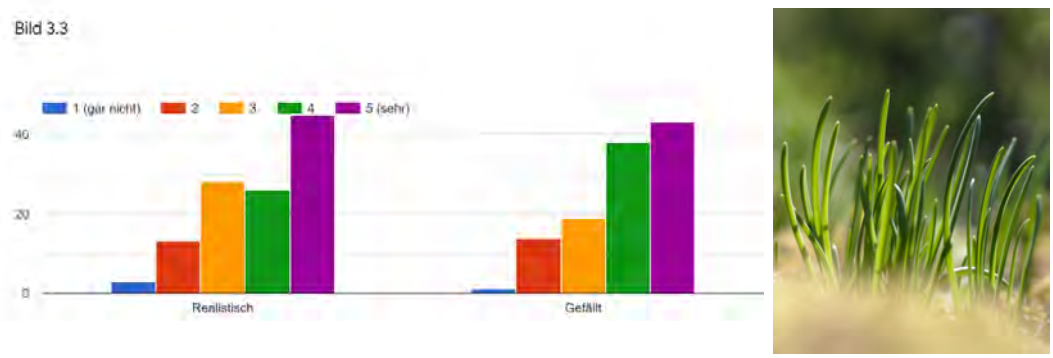


Abbildung 8.9: Die Histogramme für die Bewertungen zum Realismus- und Schönheitsgrad von Bild 3.3 (rechts).

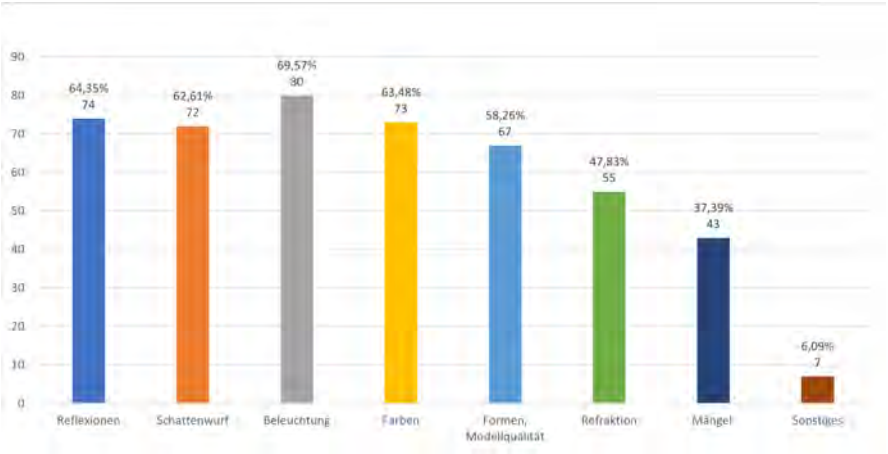


Abbildung 8.10: Histogramm der Auswahlkriterien

8.2.4 Auswahlkriterien

Das Histogramm in Abbildung 8.10 zeigt welche Auswahlkriterien von wie vielen Befragten bei der Unterscheidung von realen und computergenerierten Bilder verwendet wurden. Über den einzelnen Balken steht dabei die Anzahl der Personen, die angaben dieses Kriterium verwendet zu haben und welchem Prozentsatz der Befragten dies entspricht. Sieben Personen gaben weitere Erkennungskriterien an. Dazu gehörten unter anderen „Gefühl“, „zu Perfekt“, „natürlicher Wuchs von Bäumen“ oder auch, dass eines der Bilder von einer Person wiedererkannt wurde. Interessant ist hierbei, dass Mängel am wenigsten bewusst von den Befragten als Auswahlkriterium verwendet wurden, obwohl Mängel und Imperfektionen häufig als ausschlaggebend für photorealistische Renderings genannt werden[15,16,17].

Es wurde weiterhin geprüft, ob es eine Abhängigkeit zwischen den angegebenen Auswahlkriterien und den Erkennungsraten für die drei Bilderpaar-Gruppen gibt. Dazu wurden die Angaben für „Erkannt“ und „Nicht Erkannt“ nach den Auswahlkriterien gefiltert und für jede der drei Bilderpaar-Gruppen aufsummiert, ähnlich wie im Unterkapitel 8.2.2 für die Altersgruppen und Erfahrungen mit 3D-Grafik oder 3D-Echtzeit-Anwendungen. Damit ergibt sich Tabelle 8.12, in der zusätzlich die Erkennungsraten angegeben sind. Weitere angegebene Kriterien wurden dabei ignoriert.

Die **Hypothese** lautet: Die Erkennungsrate ist abhängig von den verwendeten Auswahlkriterien. Die Nullhypothese H_0 ist die Gegenhypothese dazu. Mit den aufsummierten Werten wurde pro Bilderpaar-Gruppe ein χ^2-Test durchgeführt, deren Ergebnisse in Tabelle 8.13 zu sehen sind. Bei keinem der Tests liegt p unter dem Signifikanzniveau von 10 Prozent, somit wird H_0 angenommen. Die Erkennungsrate ist für die verwendeten Bilderpaare also unabhängig von den von Befragten angewendeten Auswahlkriterien.

[15] https://pixelperfect-studios.com/necessary-imperfections/ – Zuletzt geprüft 13.10.2020

[16] https://blog.usejournal.com/5-reasons-why-your-3d-renders-look-fake-5d20d8118023 – Zuletzt geprüft 13.10.2020

[17] https://visualise.setvisions.co.uk/Posts/2204/photorealism-the-art-of-imperfection-pix-cgi – Zuletzt geprüft 13.10.2020

Tabelle 8.12: Aufsummierte Mengen von „Erkannt“ und „Nicht Erkannt“ pro Bilderpaar-Gruppe, gefiltert nach Auswahlkriterien

Auswahlkriterium	**Reflexionen**	**Schattenwurf**	**Beleuchtung**	**Farben**	**Formen, Modellqualität**	**Refraktion**	**Mängel**
Real-Time vs. Foto							
Erkannt	114	110	114	104	107	83	74
Nicht Erkannt	108	106	126	115	94	82	55
Erkennungsrate	51,35%	50,93%	47,50%	47,49%	53,23%	50,30%	57,36%
Offline vs. Foto							
Erkannt	118	114	129	123	104	90	68
Nicht Erkannt	104	102	111	96	97	75	61
Erkennungsrate	53,15%	52,78%	53,75%	56,16%	51,74%	54,55%	52,71%
Real-Time vs. Offline							
Erkannt	82	78	95	85	85	66	46
Nicht Erkannt	66	66	65	61	49	44	40
Erkennungsrate	55,41%	54,17%	59,38%	58,22%	63,43%	60,00%	53,49%

Tabelle 8.13: χ^2-Tests zur Prüfung auf Abhängigkeit zwischen Auswahlkriterien und Erkennungsrate für die drei Bilderpaar-Gruppen.

Chi squared	*Real-Time vs. Foto*		
Rows, columns:	2, 7	**Degrees freedom:**	6
Chi2:	4,7477	**p (no assoc.):**	0,57656
Monte Carlo p :	0,5811		
Chi squared	*Offline vs. Foto*		
Rows, columns:	2, 7	**Degrees freedom:**	6
Chi2:	1,037	**p (no assoc.):**	0,98417
Monte Carlo p :	0,9829		
Chi squared	*Real-Time vs. Offline*		
Rows, columns:	2, 7	**Degrees freedom:**	6
Chi2:	3,9173	**p (no assoc.):**	0,68786
Monte Carlo p :	0,6946		

8.3 Zusammenfassung der Evaluationsergebnisse

An den Erkennungsraten der Bilderpaare lässt sich ablesen, dass Real-Time-Rendering definitiv in der Lage ist eine Optik zu erreichen, die mit der Realität verwechselt werden kann. Im Durchschnitt liegen die Erkennungsraten der Bilderpaar-Gruppen Real-Time vs. Foto und Offline vs. Foto mit jeweils 49,28 und 53,91 Prozent sehr nah an 50 Prozent, dem Wert bei dem von einem ununterscheidbaren Realismusgrad ausgegangen werden kann.

Beim direkten Vergleich von Real-Time- und Offline-Renderings wird im Durchschnitt mit 59,57 Prozent eher das Offline-Rendering für real gehalten. Allerdings wurden hierfür nur zwei Bilderpaare verwendet. Ausschlaggebend für den hohen Durchschnitt ist Bilderpaar 1.3, bei dem zu 66,96 Prozent das Offline-Rendering für real gehalten wurde. Dieses Bilderpaar zeigt zwei Bilder einer identischen Szene, welche mit V-Ray und dem Real-Time Ray Tracing der Unreal Engine gerendert wurden. Allerdings haben die beiden Bilder sehr unterschiedliche Lichtverhältnisse, welche womöglich die Entscheidung beeinflusst haben.

Unerwartet war die Erkennungsrate für Bilderpaar 2.3, bei dem die Fotografie von 66,96 Prozent der Befragten für ein Rendering gehalten wurde und das Real-Time Rendering ohne Ray Tracing für real.

Für die Erkennungsrate von Real-Time-Renderings im Vergleich mit Fotos wurde eine Abhängigkeit vom Alter nachgewiesen: Jüngere Evaluationsteilnehmer erkannten Real-Time-Renderings im Durchschnitt eher als computergeneriert, als ältere Teilnehmer. Dieser Trend zeigt sich auch beim Vergleich von Offline-Renderings mit Fotos und dem direkten Vergleich von Real-Time- mit Offline-Renderings, bei dem Offline-Renderings von jüngeren Teilnehmern eher für Fotografien gehalten werden, mit steigendem Alter dies aber ausgeglichener wird. Allerdings konnte für diese beiden Bilderpaar-Gruppen keine Abhängigkeit zwischen Alter und Erkennungsrate nachgewiesen werden.

Eine Abhängigkeit der Erkennungsrate von vorhandenen CG-Erfahrungen konnte nachgewiesen werden: Erfahrene Befragte erkannten die Renderings eher als unerfahrene Teilnehmer. Beim Vergleich von Real-Time-Renderings machten die Erfahrungen jedoch keinen Unterschied. Erfahrungen mit Real-Time-Rendering brachten allerdings keinen zusätzlichen Vorteil bei den Erkennungsraten von Real-Time-Renderings. Eine Abhängigkeit für den Vergleich von Real-Time-Renderings mit Fotos konnte für diese Befragungsgruppe nicht nachgewiesen werden. Allerdings konnten sie Offline-Renderings noch zuverlässiger erkennen, als Teilnehmer, die angaben Erfahrung mit der Erstellung von Offline-Renderings zu haben. Wobei zwölf der 14 Befragten mit Real-Time-Erfahrung auch angaben CG-Erfahrungen zu haben.

Die Bewertungen des Realismusgrad von Bild 3.2 zeigen, dass die Optik der kommenden Unreal Engine 5 für ähnlich realistisch gehalten wird, wie die eines Fotos (Bild 3.3).

Die Auswahlkriterien Reflexionen, Schattenwurf, Beleuchtung, Farben und Formen, Modellqualität wurden alle von mehr als 58 Prozent der Befragten verwendet. Refraktion und Mängel jedoch von weniger als 48 Prozent. Interessant ist vor allen, dass Mängel von 62,61 Prozent der Befragten nicht bewusst als Auswahlkriterium verwendet wurde, um Renderings zu identifizieren. Denn oft werden Mängel und Imperfektionen als ausschlaggebend für photorealistische Renderings genannt.

Eine Abhängigkeit zwischen den bewusst verwendeten Auswahlkriterien und der Erkennungsraten der drei Bilderpaar-Gruppen konnte jedoch nicht nachgewiesen werden.

Kapitel 9

Evaluation zum Auto-Konfigurator

Der Auto-Konfigurator soll als Beispiel für eine Anwendung dienen, die sowohl Real-Time- als auch Offline-Renderings erstellt, sodass deren Nutzen verglichen werden kann. Mit der Evaluation soll festgestellt werden, ob eine Konfigurationsanwendung mit der aktuellen Real-Time-Optik ohne Ray Tracing beim Auto-Kauf gewünscht ist. Außerdem soll geprüft werden, ob ein zusätzliches Offline-Rendering den Konfigurator sinnvoll ergänzt.

9.1 Durchführung der Evaluation

Im Rahmen der Evaluation hatten Teilnehmer zunächst die Möglichkeit, mit der Auto-Konfigurator-Anwendung ein Auto nach Wunsch zu konfigurieren. Da die Konfigurator-Anwendung in ihrer Prototypen-Form nicht zwingend auf allen Computern läuft, wurde sie ausschließlich auf dem bereits für die Renderings verwendeten Rechner ausgeführt. Dadurch hatten nicht alle Teilnehmer der Evaluation die Möglichkeit den Konfigurator selbst vor Ort zu bedienen. Stattdessen konnten sie über eine Bildschirmübertragung das Auto sehen, welches nach ihrem Wunsch angepasst wurde. Die Bildschirmübertragung fand über *Discord*[1] statt, welches Übertragungen für die Verwendung mit Computerspielen optimiert hat.

Nachdem ein Teilnehmer mit seiner Konfiguration zufrieden war, sollte er eine Kameraposition und Umgebung auswählen, von der aus das V-Ray-Rendering seiner Auto-Konfiguration erstellt werden sollte. Von dieser Position aus wurde außerdem ein Screenshot aus dem Konfigurator erstellt.

Für das V-Ray-Rendering wurde eine Auflösung von 1280x720 Pixel und das Preset 3 „Balanced“ mit dem V-Ray Denoiser verwendet, um die Renderzeit möglichst niedrig zu halten. Im Level „Studio“ lag die Renderzeit in der Regel unter zehn Minuten. Im Level „Warehouse“ wurde die Renderzeit, abhängig von der Zeit des Teilnehmers, auf zehn oder 30 Minuten beschränkt, damit nicht zu lange auf ein Rendering gewartet werden muss. Durch den V-Ray Denoiser haben Renderings mit beiden Zeit-Beschränkungen eine gute Qualität. Kleine Artefakte konnten jedoch mit nur zehn Minuten Renderzeit auftauchen, wurden aber von den Teilnehmer nicht bemängelt.

[1] https://discord.com/

Abbildung 9.1: Screenshots aus dem Konfigurator (links) mit zugehörigen V-Ray-Renderings (rechts), die im Rahmen der Evaluation erstellt wurden.

Abbildung 9.1 zeigt zwei Konfigurationen von Teilnehmern, jeweils als Screenshot vom Konfigurator und V-Ray-Rendering. Nach Abschluss des Renderings wurde dem jeweiligen Teilnehmer sowohl das V-Ray-Rendering als auch der erstellte Screenshot zum Vergleich geschickt. Dadurch dürften auch mögliche Kompressionsartefakte, die bei der Bildschirmübertragung auftreten könnten, weniger Einfluss auf die Antworten der Teilnehmer haben.

Im Anschluss daran sollten die Teilnehmer einen Fragebogen ausfüllen. In diesem wurde zunächst gefragt, ob sie die Konfigurator-Anwendung selbst bedient haben oder per Bildschirmübertragung teilnahmen. Anschließend wurden zur Einordnung der Befragungsgruppe, die selben Fragen gestellt, wie in der Evaluation zum Vergleich von Offline- und Real-Time-Renderings (siehe Kapitel 8).

Zum Konfigurator sollten die Befragten zunächst angeben, wie sehr sie den folgenden Aussagen zustimmen, welche nachfolgend identisch nummeriert werden. Zur Auswahl standen dabei „Stimme nicht zu“, „Stimme eher nicht zu“, „Stimme eher zu“ und „Stimme zu“:

1. Die Konfigurator-Anwendung hat eine gute optische Qualität.
2. Mit dem Konfigurator bekomme ich einen deutlich besseren Eindruck vom Auto, als wenn ich nur Farbmuster und Bilder des Autos sehen könnte.
3. Ohne Konfigurator würde ich keine Kaufentscheidung treffen können.

4. Vor zukünftigen Auto-Käufen möchte ich, dass eine vergleichbare Konfigurator-Anwendung zur Verfügung steht.

Zusätzlich sollten sie angeben, ob und wenn ja welches Level sie zum Konfigurieren präferierten.

Bei den folgenden Aussagen, die sich auf das V-Ray-Rendering beziehen, sollten die Teilnehmer angeben, wie sehr sie diesen zustimmen. Nachfolgend wird weiterhin diese Nummerierung für die Aussagen verwendet:

1. Das Bild hat eine bessere Qualität als die Konfigurator-Anwendung selbst.
2. Das Bild würde eine Kaufentscheidung des konfigurierten Autos erleichtern.
3. Die Optik der Konfigurator-Anwendung ist im Vergleich zum Bild für eine Kaufentscheidung nicht ausreichend.
4. Auf das Bild warten zu müssen, ist für mich kein Problem.
5. Bei zukünftigen Auto-Käufen sollte eine Konfigurator-Anwendung in der Lage sein qualitativ ähnliche Bilder zu erstellen.

Abschließend gaben die Teilnehmer noch an, wie lange sie bereit wären, auf ein V-Ray-Rendering zu warten, wenn sie vor Ort warten müssten. Ebenso gaben sie an, wie lange sie warten würden, wenn es ihnen später zugesendet werden würde, zum Beispiel per E-Mail.

Der vollständige Fragebogen ist in Anhang B zu finden.

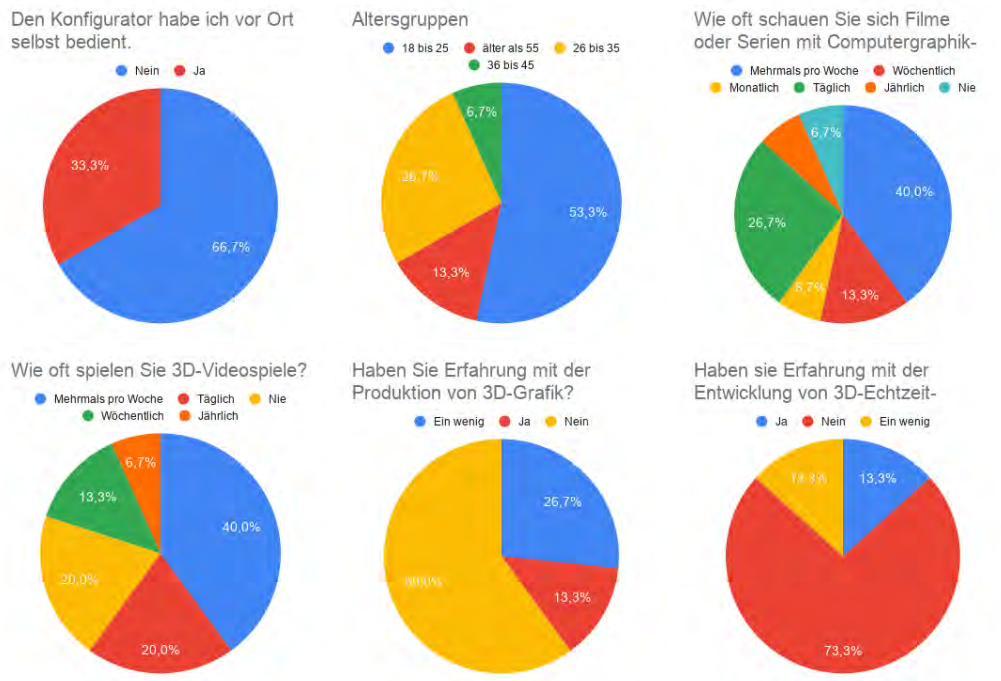


Abbildung 9.2: Kreisdiagramme zu den Fragen, mit denen die Befragungsgruppe eingeordnet werden kann.

9.2 Evaluationsergebisse

9.2.1 Befragungsgruppe

An der Evaluation zum Auto-Konfigurator nahmen 15 Personen teil. Davon konnten fünf den Konfigurator vor Ort bedienen, während die übrigen zehn über eine Bildschirmübertragung ihr Auto konfigurierten.

Die durchschnittliche Altersgruppe war 26 bis 35, wobei der Median bei 18 bis 25 lag.

In Abbildung 9.2 sind die Antworten zu den ersten Fragen in Form von Kreisdiagrammen dargestellt. An diesen kann, neben dem Alter und wie der Konfigurator bedient wurde, abgelesen werden, welche Erfahrungen die Befragungsgruppe bereits mit dem Konsum und der Produktion von 3D-Grafiken hat.

9.2.2 Fragen zum Auto-Konfigurator

In Abbildung 9.3 ist in Form eines Histogramms zu sehen, wie viele Befragten den Aussagen inwiefern zustimmten. Abbildung 9.4 zeigt die aus den Antworten resultierenden Box-Plots. In dem Boxplot wurde die Zustimmung numerisch dargestellt, dabei entspricht eins „Stimme nicht zu“ und vier „Stimme zu“. Die Aussagen lauteten wie folgt:

Abbildung 9.3: Histogramme für die Antworten zu den Aussagen zum Auto-Konfigurator.

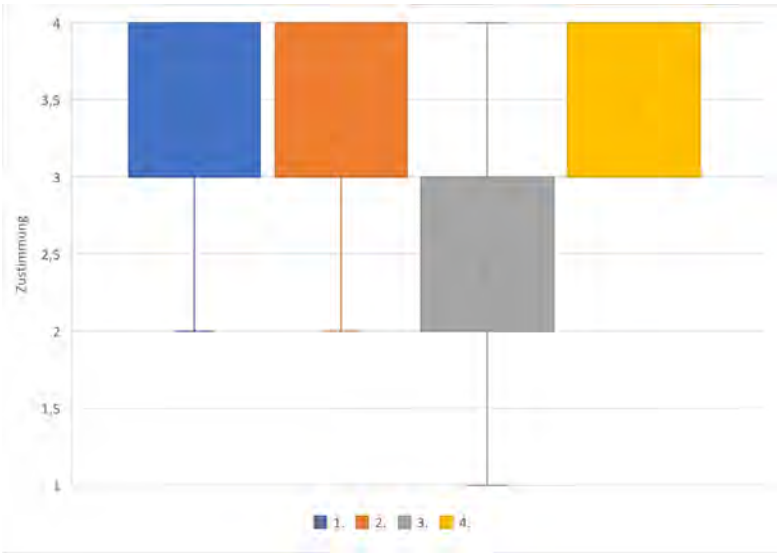


Abbildung 9.4: Box-Plots zu den Aussagen zum Auto-Konfigurator. Das Kreuz markiert das arithmetische Mittel.

1. Die Konfigurator-Anwendung hat eine gute optische Qualität.
2. Mit dem Konfigurator bekomme ich einen deutlich besseren Eindruck vom Auto, als wenn ich nur Farbmuster und Bilder des Autos sehen könnte.
3. Ohne Konfigurator würde ich keine Kaufentscheidung treffen können.
4. Vor zukünftigen Auto-Käufen möchte ich, dass eine vergleichbare Konfigurator-Anwendung zur Verfügung steht.

Wie sich erkennen lässt fanden die meisten Befragten die optische Qualität des Konfigurators gut. Im Durchschnitt und Median stimmten die Befragten der ersten Aussage eher zu.

Abbildung 9.5: Kreisdiagramm zum bevorzugten Level für das Konfigurieren.

Eine sehr ähnliche Zustimmung hat die zweite Aussage. Die Befragten bevorzugen den Konfigurator also gegenüber Farbmustern und Bildern.

Weniger Zustimmung fand die dritte Aussage. Im Durchschnitt und Median benötigen die Befragten eher keinen Auto-Konfigurator, um eine Kaufentscheidung für ein Auto zu treffen. Allerdings liegt für diese Aussage eine größere Streuung der Antworten vor: Die Standardabweichung liegt bei 0,92.

Vor zukünftigen Auto-Käufen würden jedoch alle Evaluations-Teilnehmer gerne eine vergleichbare Konfigurator-Anwendung benutzen können, denn alle Teilnehmer stimmten der vierten Aussage mindestens eher zu.

Bei der Wahl der Hintergrundumgebung zum Konfigurieren des Autos fand das Level „Warehouse“ mit 46,7 Prozent am meisten Zuspruch. Das „Studio“-Level bevorzugten nur 26,7 Prozent und weitere 26,7 Prozent fanden beiden Umgebungen gleich gut (Siehe Abbildung 9.5). Dies deutet daraufhin, dass zum Konfigurieren eine realitätsnahe Umgebung bevorzugt wird.

9.2.3 Fragen zum Offline-Rendering

Abbildung 9.6 zeigt die Histogramme für die Aussagen zum Auto-Konfigurator und Abbildung 9.7 die resultierenden Box-Plots. Die Aussagen waren folgende:

1. Das Bild hat eine bessere Qualität als die Konfigurator-Anwendung selbst.
2. Das Bild würde eine Kaufentscheidung des konfigurierten Autos erleichtern.
3. Die Optik der Konfigurator-Anwendung ist im Vergleich zum Bild für eine Kaufentscheidung nicht ausreichend.
4. Auf das Bild warten zu müssen, ist für mich kein Problem.
5. Bei zukünftigen Auto-Käufen sollte eine Konfigurator-Anwendung in der Lage sein qualitativ ähnliche Bilder zu erstellen.

Abbildung 9.6: Histogramme für die Antworten zu den Aussagen zum Offline-Rendering.

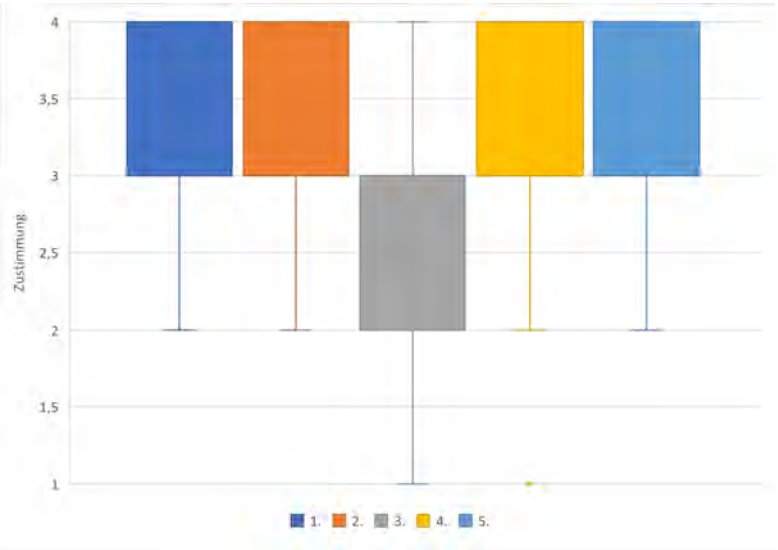


Abbildung 9.7: Box-Plots zu den Aussagen zum Offline-Rendering. Das Kreuz markiert das arithmetische Mittel.

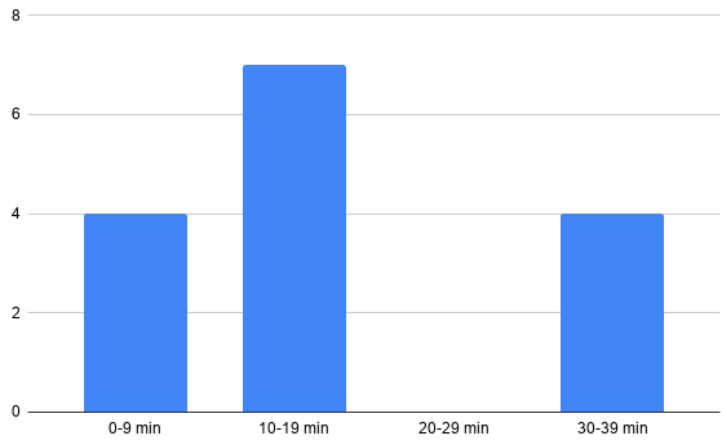


Abbildung 9.8: Histogramm für Wartezeit vor Ort.

Fast alle Befragten fanden die optische Qualität des Offline-Renderings besser als die der Anwendung. Durchschnitt und Median liegen beide bei „Stimme zu“. Auch eine Kaufentscheidung würde das Rendering den meisten erleichtern.

Weniger Zustimmung fand die dritte Aussage mit Durchschnitt und Median bei „Stimme eher nicht zu“. Die Optik der Konfigurator-Anwendung wäre also meistens ausreichend für eine Kaufentscheidung und das Offline-Rendering nicht notwendig.

Auf das Bild warten zu müssen, war für die meiste Befragten kein Problem. Trotzdem sollten zukünftige Konfigurator-Anwendungen den meisten Teilnehmer zu Folge in Zukunft eine ähnliche Qualität, wie das Offline-Rendering erreichen können.

Abbildung 9.8 zeigt in Form eines Histogramms, wie lange wie viele der Befragten bereit wären, bei einem Auto-Händler vor Ort auf ein Offline-Rendering zu warten. Im Durchschnitt würden die Befragten 14 Minuten 45 Sekunden auf das Bild warten. Der Median lag bei 10 Minuten. Die maximale von drei Personen angegebene Wartezeit liegt bei 30 Minuten. Die minimale hingegen bei 1 Minute 33 Sekunden. Für die meisten Personen wäre also eine Renderzeit, wie die V-Ray-Renderings für diese Evaluation sie benötigten, kein Problem. Allerdings sollten die Level dafür optimiert sein. Das „Warehouse“-Level zeigte, dass 10 Minuten Renderzeit bei Räumen mit Fenstern nicht unbedingt ausreichend sind.

Wenn das Offline-Rendering später zum Beispiel per E-Mail zugesendet wird, wären die Befragten bereit deutlich länger zu warten. Das Histogramm dazu zeigt Abbildung 9.9. Im Durchschnitt könnten die Befragten 16 Stunden 26 Minuten 11 Sekunden auf das Bild warten. Der Medianwert liegt bei 23 Stunden. Die kürzeste angegebene Wartezeit liegt bei einer Stunde, die längste hingegen bei 48 Stunden. Sollte das Rendering also nicht in weniger als 15 Minuten berechnet werden können, um es den Kunden direkt mitzugeben, wären diese auch bereit deutlich länger zu warten, wenn es ihnen zu gesendet wird.

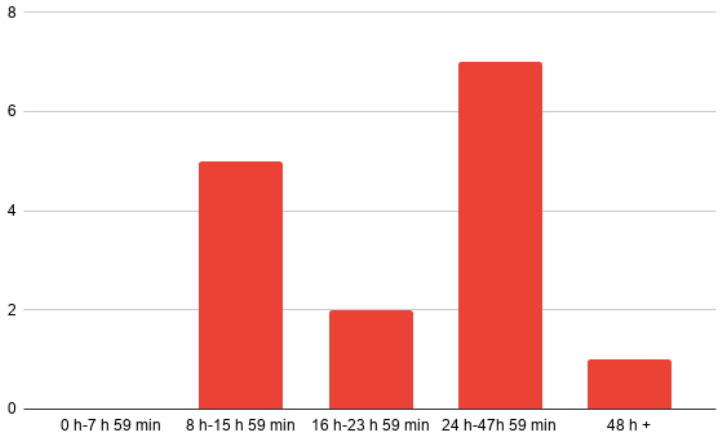


Abbildung 9.9: Histogramm für Wartezeiten bei späterer Zusendung.

Kapitel 10

Zusammenfassung und Ausblick

10.1 Zusammenfassung

Im Rahmen dieser Arbeit wurden das Real-Time- und das Offline-Rendering miteinander verglichen. Offline-Rendering fand bisher vor allen in verschiedenen Bild- und Videoproduktionen Verwendung. Real-Time-Rendering wird hauptsächlich in Videospielen verwendet. Beim Real-Time-Rendering ist es bisher meist nicht möglich eine realistische Lichtberechnung durchzuführen. Trotzdem sind Real-Time-Renderings mittlerweile teilweise so qualitativ hochwertig, dass sie anstatt von Offline-Renderings bereits für einige Film- und Serienproduktionen verwendet wurden.

Für einen möglichst direkten Vergleich der beiden Rendertechniken wurde in dieser Arbeit eine Auto-Konfigurator-Anwendung verwendet, bei der beim Konfigurieren Real-Time-Renderings erzeugt werden und nach Abschluss der Konfiguration ein Offline-Rendering des Autos erstellt werden kann. Ein bereits entwickelter Auto-Konfigurator wurde dazu um das Feature zur Erstellung von Offline-Renderings erweitert. Dieser war mit der *Unreal Engine 4* entwickelt worden und die Offline-Renderings wurden mit *V-Ray for Unreal* ermöglicht.

Mit Hilfe der Konfigurator-Anwendung wurden vier Workflows für das Shading von Auto-Modellen getestet, die sowohl für die Verwendung in einer 3D-Echtzeitanwendung als auch für die Erstellung von Einzelbildern geshadet werden müssen. Die Workflows waren folgende:

1. Das Modell wurde in einer 3D-Anwendung (verwendet wurde Maya) für das Offline-Rendering (erstellt mit V-Ray) geshadet und das Rendering erstellt. Anschließend wurde es erneut für die verwendete Game Engine (Unreal Engine 4) geshadet.

2. Shading und Offline-Rendering findet erneut in Maya statt. Mit Hilfe von V-Ray for Unreal wird das geshadete Modell in die Unreal Engine importiert und falls nötig angepasst.

3. V-Ray for Unreal wird verwendet, um im Unreal Editor gleichzeitig für das V-Ray-Rendering und die Real-Time-Darstellung zu shaden. Das V-Ray-Rendering wird mit V-Ray for Unreal durchgeführt.

4. Es wird nur in der Unreal Engine geshadet. Ein einzelnes Real-Time-Rendering dient als Einzelbild.

Die Durchführung der Workflows zeigte, dass der dritte und der vierte Workflow durch den einmaligen Shading-Prozess am schnellsten durchgeführt werden können. Darüber hinaus bringen sie keine unerwarteten Einschränkungen mit sich. Dies gilt auch für den ersten Workflow, welcher durch das zweimalige Shading jedoch deutlich langsamer ist. Die meisten Einschränkungen bringt der zweite Workflow mit sich, da nicht alle Material-Einstellungen problemlos ex- und importiert werden können, wodurch umfangreiche Materialanpassungen für die Real-Time-Renderings notwendig sind. Folglich war der zweite Workflow auch nur geringfügig schneller als der Erste. Einen Vorteil, den das Shading mit Hilfe einer Game Engine bringt, unabhängig ob für Real-Time- oder Offline-Rendering, ist die unmittelbare Vorschau der gesamten Szene. Diese wird dabei in deutlich besserer Qualität und mit besserem Materialverhalten dargestellt, als es zum Beispiel im Viewport von Maya der Fall wäre. Dadurch kann zum Beispiel schneller mit Materialien experimentiert werden.

Nach der Durchführung der Workflows wurden Offline- und Real-Time-Renderings von einem der Auto-Modelle erstellt, um sie zu vergleichen und zu analysieren. Neben einem gerasterten Real-Time-Rendering wurde auch eines mit Real-Time Ray Tracing erstellt. Es zeigte sich, dass die Schattenberechnung in Echtzeit mit Hilfe von Shadow Maps weiche Schatten nicht gut darstellen kann, was bei vielen Lichtquellen oder der Verwendung von Image Based Lighting notwendig ist. Für die Schatten von statischen Objekten kann jedoch auch das Lightmapping verwendet werden, was ähnliche Schatten wir V-Ray erzeugte. Außerdem können Reflexionen in gerasterten Real-Time-Renderings durch die Verwendung von Cubemaps oder Screen Space Reflections nicht akkurat berechnet werden. Diese Probleme des Real-Time-Renderings kann das Real-Time Ray Tracing lösen und kommt dabei an eine ähnliche Qualität, wie die mit V-Ray erstellten Offline-Renderings. Bei der Darstellung von transparenten Oberflächen haben jedoch beide Real-Time-Rendertechniken Probleme: Bei der Rasterung beeinflussen Reflexionen auf zum Beispiel Glas, wie dahinter liegende Objekte dargestell werden. Dadurch werden diese Objekte zum Beispiel heller dargestellt, als in der Realität. Das Real-Time Ray Tracing hat das Problem, dass es bei der gleichzeitigen Verwendung von Ray Tracing für Reflexionen und Transparenz nicht mehr darstellt, wie das Licht durch spiegelnde, transparente Objekte fällt.

Es wurde darüber hinaus eine Evaluation zum Vergleich von Offline- und Real-Time-Renderings durchgeführt, um zu prüfen, ob Offline-Renderings noch für deutlich realistischer gehalten werden als Real-Time-Renderings. Dazu wurden mehrere Bilderpaare gezeigt in denen mit Offline- oder Real-Time-Rendering erstellte Bilder mit Fotografien oder miteinander verglichen wurden. Die Befragten sollten dann angeben, welches Bild sie für real halten. Außerdem sollten weitere Bilder nach ihrem Realismusgrad bewertet werden und die Befragten sollten angeben, worauf ihre Entscheidungen basierten.

Mit der Evaluation konnte festgestellt werden, dass Bilder beider Rendertechniken im Vergleich mit Fotografien im Durchschnitt nur zu ungefähr 50 Prozent erkannt werden (49,28 Prozent für Real-Time-Renderings und 53,91 Prozent für Offline-Renderings). Somit

können beide Techniken eine Optik erreichen, die mit der Realität verwechselt werden kann. Beim direkten Vergleich von Bildern beider Rendertechniken, wurden Offline-Renderings allerdings zu 59,57 Prozent für das Foto gehalten.

Zusätzlich wurde durch die Evaluation gezeigt, dass das Erkennen von Real-Time-Renderings abhängig vom Alter ist. Mit steigendem Alter wurden die Renderings seltener erkannt. Auch die durchschnittliche Erkennungsrate von Offline-Renderings fiel mit steigendem Alter, allerdings konnte für diese keine Abhängigkeit vom Alter nachgewiesen werden.

Eine Abhängikeit zwischen vorliegenden Erfahrungen mit 3D-Grafik und dem Erkennen von Renderings konnte allerdings nachgewiesen werden: Erfahrene Befragte erkannten Renderings im Durchschnitt eher als solche. Zusätzliche Erfahrungen mit der Entwicklung von 3D-Echtzeitanwendungen sorgten allerdings nicht für eine bessere Erkennungsrate von Real-Time-Renderings.

Außerdem wurde auf eine Abhängigkeit zwischen den bewusst verwendeten Auswahlkriterien, die die Befragten angegeben hatten, und der Erkennungsrate geprüft. Eine solche Abhängigkeit konnte jedoch nicht nachgewiesen werden.

Zum Auto-Konfigurator wurde eine weitere Evaluation durchgeführt. Diese sollte Meinungen zum Nutzen einer solchen Anwendung beim Autokauf sammeln und feststellen, ob das zusätzliche Offline-Rendering notwendig ist. Durch die Ergebnisse der Evaluation zeigte sich, dass eine Konfigurator-Anwendung zwar gerne bei einem Autokauf gesehen wird, aber nicht für eine Kaufentscheidung notwendig ist. Das zusätzliche Offline-Rendering wird ebenfalls als hilfreich angesehen. Die für dieses notwendige Wartezeit ist in der Regel auch kein Problem. Allerdings würde die Optik der Konfigurator-Anwendung den meisten Befragten bereits für eine Kaufentscheidung ausreichen. Die Befragten wünschten sich allerdings, dass eine Konfigurator-Anwendung in Zukunft eine ähnliche optische Qualität, wie das Offline-Rendering erreichen soll.

Zusammenfassend lässt sich sagen, dass Real-Time-Renderings mittlerweile durchaus mit der Realität verwechselt werden können und somit ähnlich wie Offline-Renderings eingesetzt werden können. Durch das Real-Time Ray Tracing werden einige Schwächen beseitigt, die das Real-Time-Rendering im Vergleich zum Offline-Rendering noch hatte. Trotzdem existieren noch Unterschiede in den Bildern der beiden Techniken, die unter Umständen dazu führen, dass das Offline-Rendering dem Real-Time-Rendering vorgezogen werden kann. Für die Beispielanwendung des Auto-Konfigurators zeigte sich, dass ein Offline-Rendering zusätzlich zur Real-Time-Anwendung zwar eine bessere Darstellung des Autos erzielt, allerdings nicht unbedingt benötigt wird. Durch die Verwendung von Real-Time Ray Tracing für den Konfigurator könnte es auch komplett obsolet sein, da die Darstellung im Konfigurator dann noch stärker der von Offline-Renderings ähnelt.

10.2 Ausblick

Das Real-Time Ray Tracing, welches in dieser Arbeit neben dem Verfahren der reinen Rasterung für Real-Time-Renderings behandelt wurde, ist aufgrund der benötigten Hardware noch nicht sehr stark verbreitet. Bei einer höheren Verbreitung der notwendigen Hardware, ist davon auszugehen, dass das Real-Time Ray Tracing immer mehr Verwendung in Videospielen und anderen 3D-Echtzeitanwendungen findet. Dadurch ist zu erwarten, dass die Qualität von Videospielen und ähnlichen Anwendungen in Zukunft noch weniger von Offline-Renderings unterschieden werden kann.

Es ist auch zu erwarten, dass das Real-Time-Rendering immer mehr für die Produktion von Filmen und Serien Verwendung finden wird. Denn es ermöglicht virtuelle Effekte und Umgebungen direkt am Set sichtbar zu machen, was die Arbeit von Schauspielern, Kameramännern und Regisseuren vereinfachen kann. Außerdem können Hintergrund und Effekte wenn nötig noch beim Dreh angepasst werden.

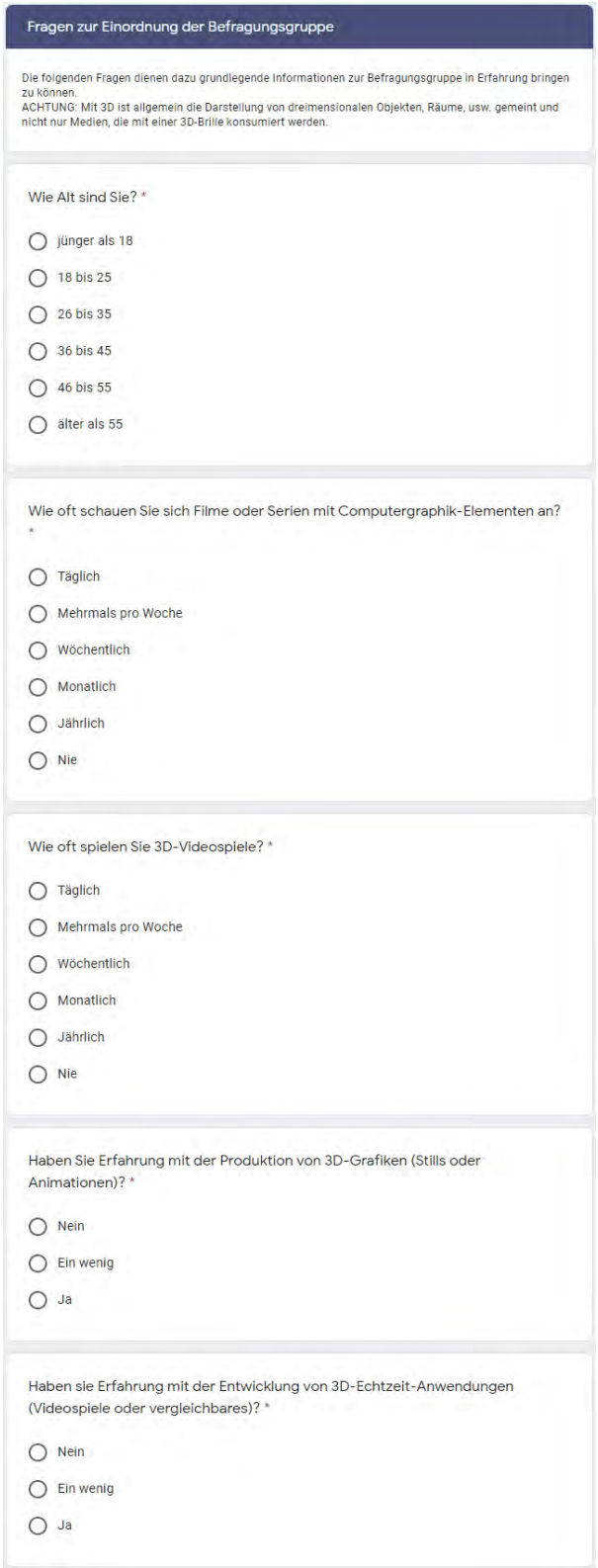
Durch die steigende Qualität von Real-Time-Renderings stellt sich die Frage, ob ein wie in dieser Arbeit entwickeltes Feature zum Erstellen von Offline-Renderings von einer Konfiguratoranwendung aus überhaupt noch notwendig sein wird. Trotzdem ist eine weniger prototypenhafte Umsetzung durch zukünftige Updates an V-Ray for Unreal denkbar.

Diese Arbeit befasste sich vor allen mit den Unterschieden und Gemeinsamkeiten in der Lichtberechnung bei Offline- und Real-Time-Renderings. Weiterhin könnten andere Bereiche der 3D-Grafik zwischen den beiden Techniken in Zukunft wissenschaftlich verglichen werden, wie zum Beispiel Partikel, Flüssigkeits- oder Physiksimulationen.

Außerdem kann eine Wiederholung der Evaluation zum Vergleich von Offline- und Real-Time-Renderings mit anderen Bildern sinnvoll sein. Dadurch könnten aus den Ergebnissen der Evaluationen Aussagen abgeleitet werden, die weniger von den verwendeten Bilder abhängen, sondern noch stärker von den verwendeten Rendertechniken.

Anhang A

Evaluationsfragen zum Vergleich von Offline- und Real-Time-Renderings



Fragen zur Einordnung der Befragungsgruppe

Die folgenden Fragen dienen dazu grundlegende Informationen zur Befragungsgruppe in Erfahrung bringen zu können.
ACHTUNG: Mit 3D ist allgemein die Darstellung von dreimensionalen Objekten, Räume, usw. gemeint und nicht nur Medien, die mit einer 3D-Brille konsumiert werden.

Wie Alt sind Sie? *

- jünger als 18
- 18 bis 25
- 26 bis 35
- 36 bis 45
- 46 bis 55
- älter als 55

Wie oft schauen Sie sich Filme oder Serien mit Computergraphik-Elementen an? *

- Täglich
- Mehrmals pro Woche
- Wöchentlich
- Monatlich
- Jährlich
- Nie

Wie oft spielen Sie 3D-Videospiele? *

- Täglich
- Mehrmals pro Woche
- Wöchentlich
- Monatlich
- Jährlich
- Nie

Haben Sie Erfahrung mit der Produktion von 3D-Grafiken (Stills oder Animationen)? *

- Nein
- Ein wenig
- Ja

Haben sie Erfahrung mit der Entwicklung von 3D-Echtzeit-Anwendungen (Videospiele oder vergleichbares)? *

- Nein
- Ein wenig
- Ja

Abbildung A.1: Die erste Seite des Evaluations-Formulars zum Vergleich von Offline- und Real-Time-Renderings

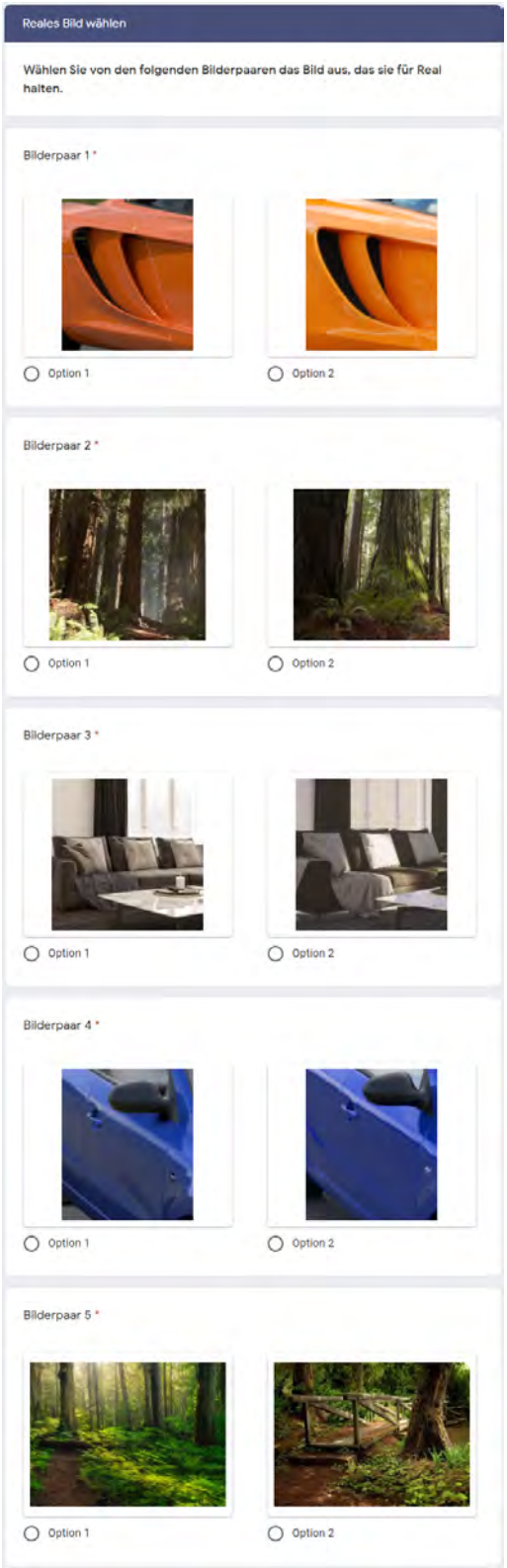


Abbildung A.2: Die zweite Seite des Evaluations-Formulars zum Vergleich von Offline- und Real-Time-Renderings

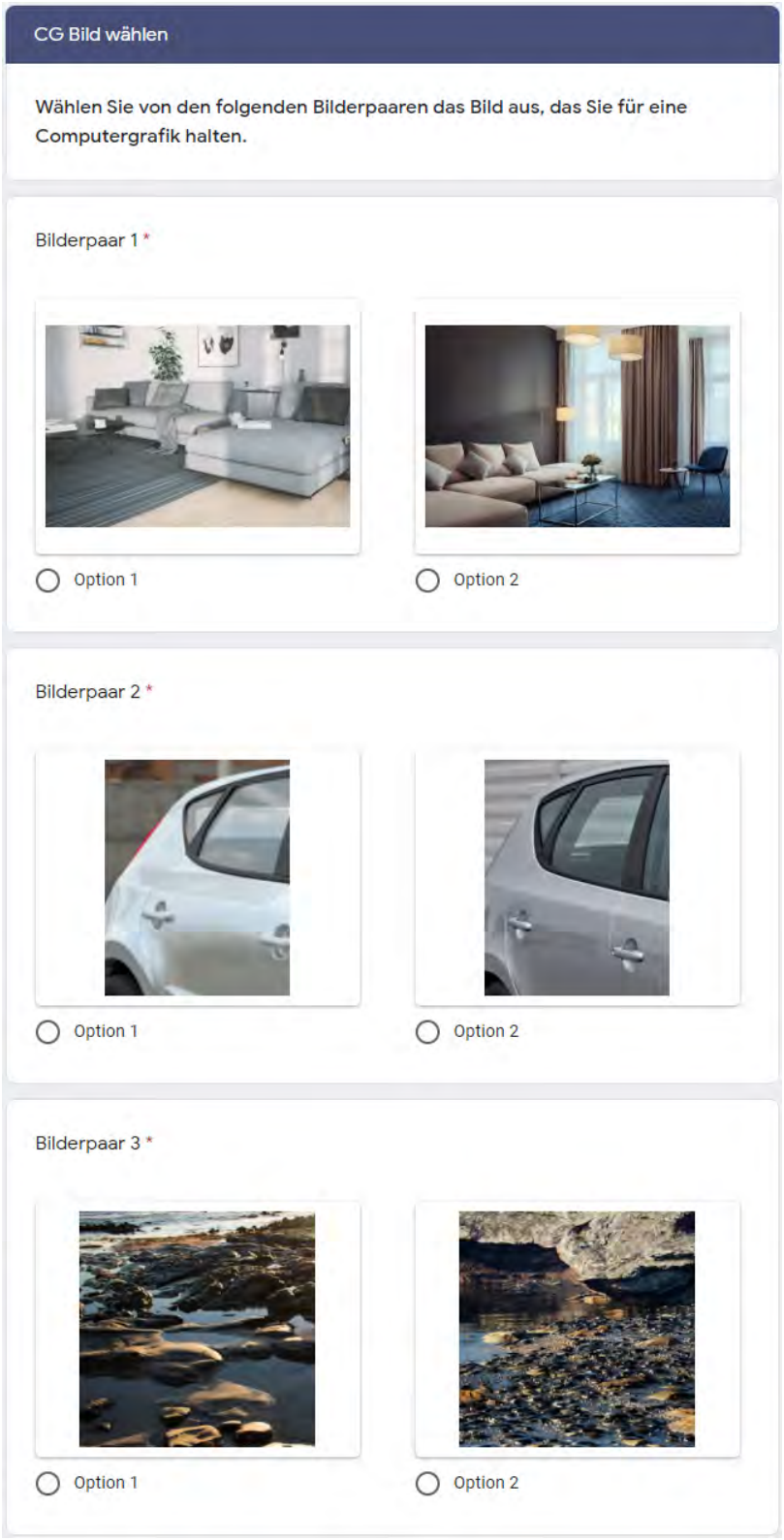


Abbildung A.3: Die dritte Seite des Evaluations-Formulars zum Vergleich von Offline- und Real-Time-Renderings

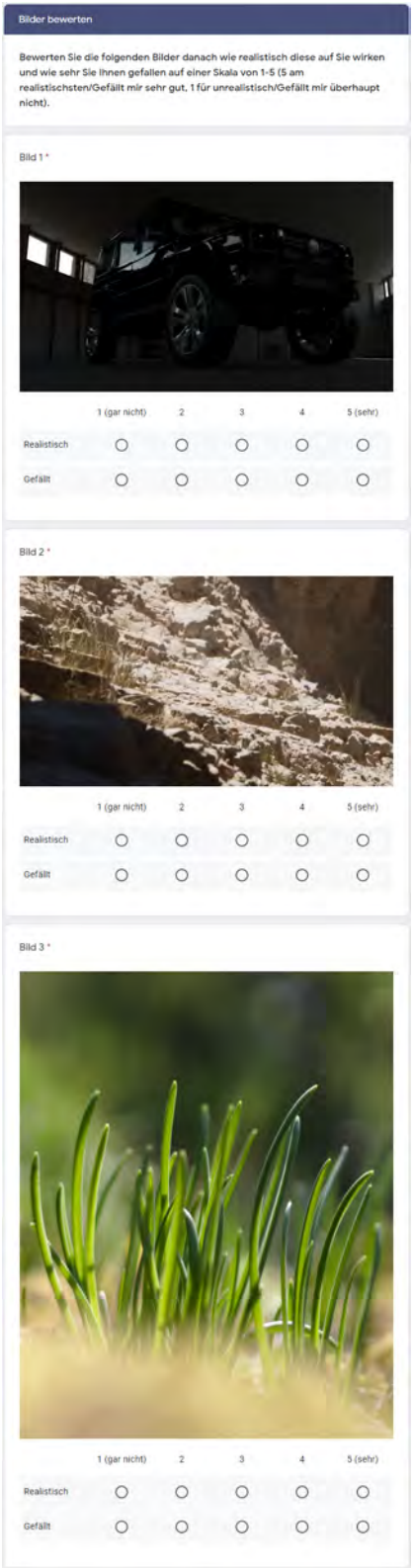


Abbildung A.4: Die vierte Seite des Evaluations-Formulars zum Vergleich von Offline- und Real-Time-Renderings

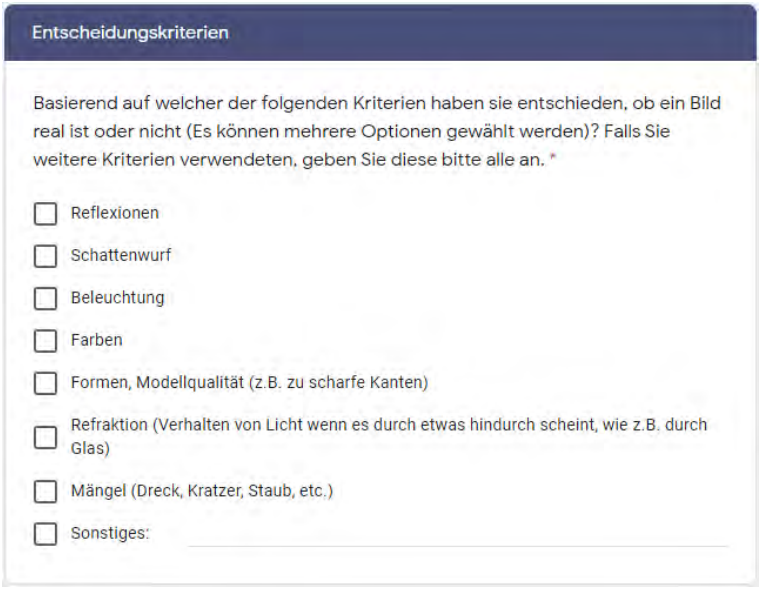
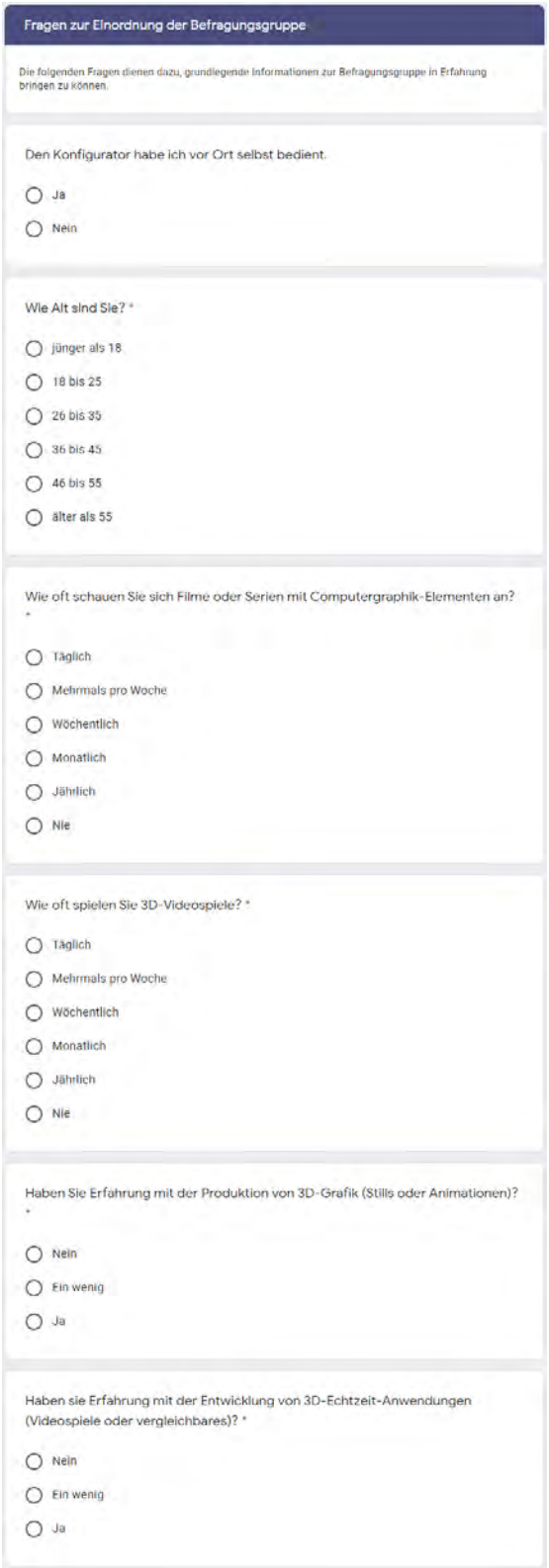


Abbildung A.5: Die fünfte Seite des Evaluations-Formulars zum Vergleich von Offline- und Real-Time-Renderings

Anhang B

Evaluationsfragen zum Auto-Konfigurator



Fragen zur Einordnung der Befragungsgruppe

Die folgenden Fragen dienen dazu, grundlegende Informationen zur Befragungsgruppe in Erfahrung bringen zu können.

Den Konfigurator habe ich vor Ort selbst bedient.

- Ja
- Nein

Wie Alt sind Sie? *

- jünger als 18
- 18 bis 25
- 26 bis 35
- 36 bis 45
- 46 bis 55
- älter als 55

Wie oft schauen Sie sich Filme oder Serien mit Computergraphik-Elementen an? *

- Täglich
- Mehrmals pro Woche
- Wöchentlich
- Monatlich
- Jährlich
- Nie

Wie oft spielen Sie 3D-Videospiele? *

- Täglich
- Mehrmals pro Woche
- Wöchentlich
- Monatlich
- Jährlich
- Nie

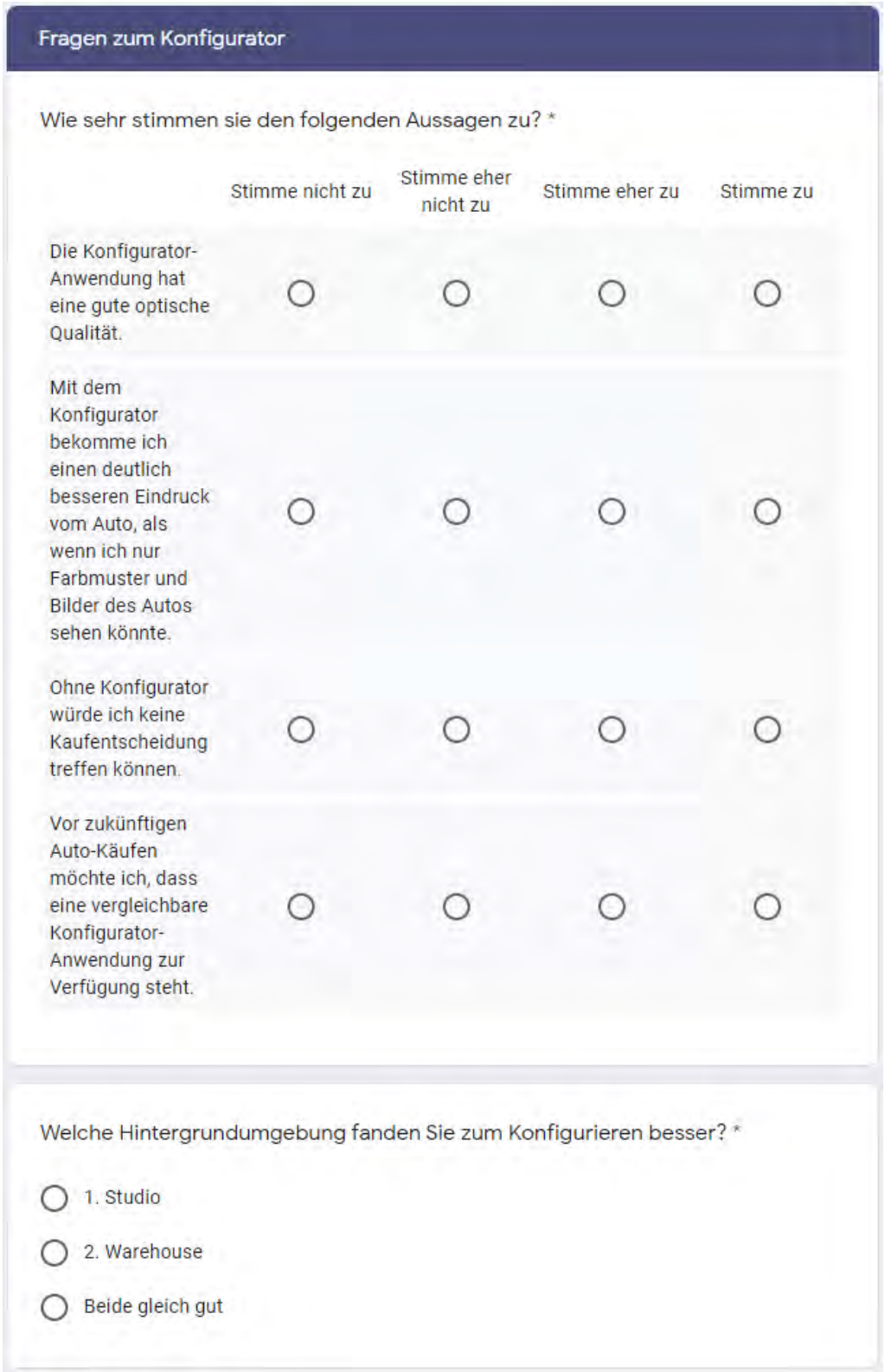
Haben Sie Erfahrung mit der Produktion von 3D-Grafik (Stills oder Animationen)? *

- Nein
- Ein wenig
- Ja

Haben sie Erfahrung mit der Entwicklung von 3D-Echtzeit-Anwendungen (Videospiele oder vergleichbares)? *

- Nein
- Ein wenig
- Ja

Abbildung B.1: Die erste Seite des Evaluations-Formulars zum Auto-Konfigurator



Fragen zum Konfigurator

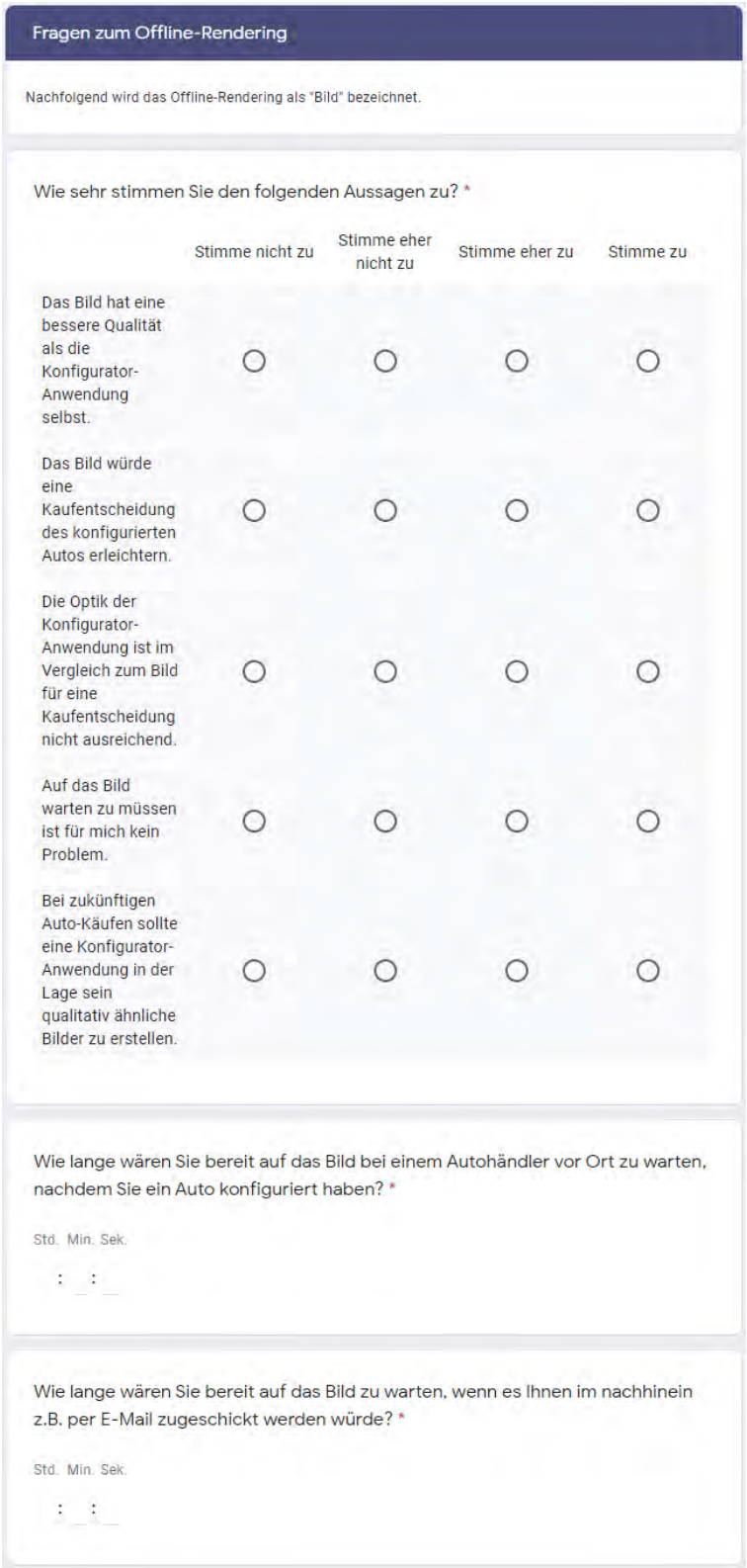
Wie sehr stimmen sie den folgenden Aussagen zu? *

	Stimme nicht zu	Stimme eher nicht zu	Stimme eher zu	Stimme zu
Die Konfigurator-Anwendung hat eine gute optische Qualität.	○	○	○	○
Mit dem Konfigurator bekomme ich einen deutlich besseren Eindruck vom Auto, als wenn ich nur Farbmuster und Bilder des Autos sehen könnte.	○	○	○	○
Ohne Konfigurator würde ich keine Kaufentscheidung treffen können.	○	○	○	○
Vor zukünftigen Auto-Käufen möchte ich, dass eine vergleichbare Konfigurator-Anwendung zur Verfügung steht.	○	○	○	○

Welche Hintergrundumgebung fanden Sie zum Konfigurieren besser? *

- ○ 1. Studio
- ○ 2. Warehouse
- ○ Beide gleich gut

Abbildung B.2: Die zweite Seite des Evaluations-Formulars zum Auto-Konfigurator



Fragen zum Offline-Rendering

Nachfolgend wird das Offline-Rendering als "Bild" bezeichnet.

Wie sehr stimmen Sie den folgenden Aussagen zu? *

	Stimme nicht zu	Stimme eher nicht zu	Stimme eher zu	Stimme zu
Das Bild hat eine bessere Qualität als die Konfigurator-Anwendung selbst.	○	○	○	○
Das Bild würde eine Kaufentscheidung des konfigurierten Autos erleichtern.	○	○	○	○
Die Optik der Konfigurator-Anwendung ist im Vergleich zum Bild für eine Kaufentscheidung nicht ausreichend.	○	○	○	○
Auf das Bild warten zu müssen ist für mich kein Problem.	○	○	○	○
Bei zukünftigen Auto-Käufen sollte eine Konfigurator-Anwendung in der Lage sein qualitativ ähnliche Bilder zu erstellen.	○	○	○	○

Wie lange wären Sie bereit auf das Bild bei einem Autohändler vor Ort zu warten, nachdem Sie ein Auto konfiguriert haben? *

Std. Min. Sek.

: :

Wie lange wären Sie bereit auf das Bild zu warten, wenn es Ihnen im nachhinein z.B. per E-Mail zugeschickt werden würde? *

Std. Min. Sek.

: :

Abbildung B.3: Die dritte Seite des Evaluations-Formulars zum Auto-Konfigurator

Glossar

Actor Ein Actor ist in der *Unreal Engine* jedes Objekt, das in einem Level platziert werden kann. Actors unterstützen 3D Transformationen

Blueprint Blueprints ermöglichen es Entwicklern in der *Unreal Engine* Programmlogik existierenden Gameplay Klassen mit Hilfe des *Blueprint Visual Scripting* hinzuzufügen. Gameplay Klassen sind alle Klassen die bei Laufzeit der Anwendung für den Anwender relevant sind und manipuliert werden können. Beim erstellen einer Blueprint muss eine Elternklasse ausgewählt werden, auf der die Blueprint basiert. Die Elternklassen sind Actor, Pawn, Character, PlayerController und Game Mode.

Character Ist ein Pawn, der in der Lage ist zu gehen, zu laufen, zu springen und weiteres.

Component Eine Component (zu deutsch Komponente) ist ein Teil eines Actors in der *Unreal Engine*, der eine bestimmte Art von Objekt beinhaltet. Das Objekt kann selbst ein Actor sein.

Game Instance Speichert in der *Unreal Engine* Spieldaten, sodass sie beim Laden eines neuen Levels erhalten bleiben.

Game Mode Definiert was in der *Unreal Engine* für ein Spiel gespielt wird und welche Regeln gelten. Dazu zählt zum Beispiel auch welcher Pawn standardmäßig vom Anwender/Spieler kontrolliert wird.

Level Level bezeichnet in Videospielen Spielabschnitte in denen sich die Objekte befinden, die der Spieler sieht und mit denen er interagieren kann. In der *Unreal Engine* stellt eine Level einen nahezu unendlich großen Raum da in dem Objekte platziert werden können, um die Spielwelt zu bauen. Vergleichbar mit Szenen in anderen 3D-Anwendungen.

Offline-Rendering Offline-Rendering fasst alle Rendering-Techniken zusammen, bei denen es typisch ist auf einzelne Bilder länger warten zu müssen, als das Bewegungen flüssig dargestellt werden können.

Pawn Ein Pawn ist ein Actor, über den der Anwender die Kontrolle übernehmen und ihn steuern kann.

PlayerController Ist ein Actor der *Unreal Engine* der Eingaben des Anwenders an Pawns weiterleitet, damit der Anwender diese steuern kann.

Real-Time-Rendering (auch Echtzeit-Rendering genannt) Real-Time-Renderings sind die typische Rendertechnik für Videospiele oder anderen Anwendungen, die Bilder unmittelbar nach Benutzerinteraktionen gemäß dieser anzeigen müssen. Beim Echtzeit-Rendering müssen die Bilder in wenigen Millisekunden generiert werden, damit der Benutzer das gerenderte Geschehen flüssig Wahrnehmen kann.

Rendern ist in der Computergrafik der Prozess der Bilderzeugung anhand von Rohdaten, die das Bild geometrisch beschreiben.

Save Game Ein Save Game ermöglicht der *Unreal Engine* Daten im permanenten Speicher des Computers zu speichern. Dadurch bleiben Daten auch beim Neustart der Anwendung noch erhalten.

Literaturverzeichnis

[Abr00] Abrash, Michael: *Quake's Lighting Model: Surface Caching*. https://www.bluesnews.com/abrash/chap68.shtml. Version: 2000

[Ava18] Avasthy, Tarun ; Epic Games (Hrsg.): Real-Time Rendering Solutions: Unlocking the Power of Now / Forrester Consulting. Version: 2018. https://cdn2.unrealengine.com/Unreal+Engine%2Fresources%2FEpic-Games-Real-Time-Rendering-TLP_post-production_R3-2f4769b9b2adfca45af876c2f701f65ec6ef1228.pdf. 2018. – Forschungsbericht

[Bar18] Barsegyan, Armen: *Unity and Unreal Engine: Real-Time Rendering VS Traditional 3DCG Rendering Approach*. https://cgicoffee.com/blog/2018/01/unity-real-time-rendering-vs-offline-cgi. Version: 2018

[Bli77] Blinn, James F.: Models of light reflection for computer synthesized pictures. In: *Proc. 4th annual conference on computer graphics and interactive techniques* (1977), 192–198. https://doi.org/10.1145/563858.563893

[BS87] Beckmann, Petr ; Spizzichino, André: *The scattering of electromagnetic waves from rough surfaces*. Norwood, MA : Artech House, 1987 (The Artech House radar library). – ISBN 0890062382

[Bur12] Burley, Brent: Physically-Based Shading at Disney. (2012). https://media.disneyanimation.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf

[CFLB06] Christensen, Per H. ; Fong, Julian ; Laur, David M. ; Batali, Dana: Ray Tracing for the Movie 'Cars'. Version: 2006. http://dx.doi.org/10.1109/RT.2006.280208. In: *IEEE Symposium on Interactive Ray Tracing, 2006*. Piscataway, NJ : IEEE Service Center, 2006. – DOI 10.1109/RT.2006.280208. – ISBN 1424406935, 1–6

[CHS+12] Christensen, Per H. ; Harker, George ; Shade, Jonathan ; Schubert, Brenden ; Batali, Dana: Multiresolution Radiosity Caching for Global Illumination in Movies. In: *ACM SIGGRAPH 2012 Talks*. New York, NY, USA : Association for Computing Machinery, 2012 (SIGGRAPH '12). – ISBN 9781450316835, 1

[CJ16] Christensen, Per H. ; Jarosz, Wojciech: The Path to Path-Traced Movies. In: *Foundations and Trends® in Computer Graphics and Vision* 10 (2016), Nr. 2, S. 103–175. http://dx.doi.org/10.1561/0600000073. – DOI 10.1561/0600000073. – ISSN 1572–2740

[Epi20] Epic Games: *Unreal Engine 5 Revealed! Next-Gen Real-Time Demo Running on PlayStation 5*. https://vimeo.com/417882964. Version: 2020

[Gou71] Gouraud, H.: Continuous Shading of Curved Surfaces. In: *IEEE Transactions on Computers* C-20 (1971), Nr. 6, S. 623–629

[GPSS07] Gunther, Johannes ; Popov, Stefan ; Seidel, Hans-Peter ; Slusallek, Philipp: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In: *2007 IEEE Symposium on Interactive Ray Tracing*, 2007, S. 113–118

[Jen96] Jensen, Henrik W.: Global Illumination Using Photon Maps. In: *Proceedings of the Eurographics Workshop on Rendering Techniques '96*. Berlin, Heidelberg : Springer-Verlag, 1996. – ISBN 3211828834, S. 21–30

[Kaj86] Kajiya, James T.: The rendering equation. In: *ACM SIGGRAPH Computer Graphics* 20 (1986), Nr. 4, S. 143–150. http://dx.doi.org/10.1145/15886.15902. – DOI 10.1145/15886.15902. – ISSN 00978930

[KE13] Karis, Brian ; Epic Games: *Real Shading in Unreal Engine 4*. https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf. Version: 2013

[Kel15] Keller, Alexander: *The Path Tracing Revolution in the Movie Industry*. http://dx.doi.org/10.1145/2776880.2792699. Version: 2015

[KVG+19] Kondapaneni, Ivo ; Vevoda, Petr ; Grittmann, Pascal ; Skřivan, Tomáš ; Slusallek, Philipp ; Křivánek, Jaroslav: Optimal multiple importance sampling: ACM Transactions on Graphics, 38(4), 1-14. (2019). http://dx.doi.org/10.1145/3306346.3323009. – DOI 10.1145/3306346.3323009

[Lam60] Lambert, Johann H.: *Photometria: sive De mensura et gradibus luminis, colorum et umbrae*. Augsburg : Christoph Peter Detleffsen für die Witwe von Eberhard Klett, 1760

[Lot09] Lottes, Timothy: *FXAA*. http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf. Version: 2009

[LW93] Lafortune, Eric P. ; Willems, Yves D.: Bi-directional path tracing. In: *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, 1993, 145–153

[Lyo93] Lyon, Richard F.: Phong Shading Reformulation for Hardware Renderer Simplification. (1993). http://dicklyon.com/tech/Graphics/Phong_TR-Lyon.pdf

[MMG06] Mitchell, Jason ; McTaggart, Gary ; Green, Chris: Shading in Valve's Source Engine. In: *ACM SIGGRAPH 2006 Courses*. New York, NY, USA : Association for Computing Machinery, 2006 (SIGGRAPH '06). – ISBN 1595933646, S. 129–142

[Nvi18] Nvidia: Turing-Architecture Whitepaper. (2018). https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[Nvi20] Nvidia: NVIDIA Ampere GA102 GPU Architecture. (2020). https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

[Pho75] Phong, Bui T.: Illumination for computer generated pictures. In: *Communications of ACM 18* (1975), S. 311–317. http://dx.doi.org/10.1145/360825.360839. – DOI 10.1145/360825.360839

[Rau93] Rauber, Thomas (Hrsg.): *Algorithmen in der Computergraphik*. Wiesbaden : Vieweg+Teubner Verlag, 1993 (Leitfäden und Monographien der Informatik). http://dx.doi.org/10.1007/978-3-322-89537-0. http://dx.doi.org/10.1007/978-3-322-89537-0. – ISBN 978–3–519–02127–8

[RM00] Read, Paul ; Meyer, Mark-Paul: *Restoration of Motion Picture Film*. 1. Aufl. s.l. : Elsevier professional, 2000 (Butterworth-Heinemann series in conservation and museology). – ISBN 075062793X

[Sch94] Schlick, Christophe: An Inexpensive BRDF Model for Physically-based Rendering. In: *Computer Graphics Forum* 13 (1994), Nr. 3, S. 233–246. http://dx.doi.org/10.1111/1467-8659.1330233. – DOI 10.1111/1467–8659.1330233

[TR75] Trowbridge, T. S. ; Reitz, K. P.: Average irregularity representation of a rough surface for ray reflection: Journal of the Optical Society of America, 65(5), 531. (1975). http://dx.doi.org/10.1364/JOSA.65.000531. – DOI 10.1364/JOSA.65.000531

[Ves20] Vescio, Mirko: *The main advantages of Real Time engines vs Offline rendering in Architecture*. https://oneirosvr.com/real-time-rendering-vs-offline-rendering-in-architecture/. Version: 2020

[Whi80] Whitted, Turner: An improved illumination model for shaded display. 23 (1980), Nr. 6. http://dx.doi.org/10.1145/1198555.1198743. – DOI 10.1145/1198555.1198743

[Wil78] Williams, Lance: Casting curved shadows on curved surfaces. In: *ACM SIGGRAPH Computer Graphics* 12 (1978), Nr. 3, S. 270–274. http://dx.doi.org/10.1145/965139.807402. – DOI 10.1145/965139.807402. – ISSN 00978930

[Win19] Winchester, Henry: *REAL-TIME, RAY-TRACED AND RASTERIZED RENDERING EXPLAINED*. https://www.chaosgroup.com/blog/real-time-ray-traced-and-rasterized-rendering-explained. Version: 2019