Fotorealistisches Rendering in Compositing-Applikationen

Implementierung von Blenders Cycles in The Foundrys Nuke

Diplomarbeit

Ausgeführt zum Zweck der Erlangung des akademischen Grades
Dipl.-Ing. für technisch-wissenschaftliche Berufe

am Masterstudiengang Digitale Medientechnologien an der
Fachhochschule St. Pölten, **Masterklasse Postproduktion**

von:
Oliver Rautner, BSc
dm151561

Betreuer/in und Erstbegutachter/in: Dipl.-Ing.(FH) Mario Zeller
Zweitbegutachter/in: Mag. Franz Schubert

Wien, 10.09.2018

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

- ich dieses Thema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Diese Arbeit stimmt mit der vom Begutachter bzw. der Begutachterin beurteilten Arbeit überein.

....

Ort, Datum Unterschrift

Kurzfassung

Die Arbeit befasst sich mit der Thematik des fotorealistischen Renderings innerhalb von Compositing-Applikationen.

Rendering stellt die Schnittstelle zwischen 3D und 2D dar. Anfänglich wird versucht die grundlegendsten Unterscheidungen zwischen den gebräuchlichen Rendering-Algorithmen im Zusammenhang mit dem Prozess der Generierung des zweidimensionalen Bildes aufzuzeigen; es wird ein vertiefender Einblick in Raytracing, als auch ein Überblick über die Grafikpipeline gegeben.

Es folgt die Darstellung der Technologien, welche gemeinsam die Grundlage für eine fortgeschrittene Rendering-Pipeline bilden. Dabei finden fundamentalste Konzepte, wie etwa Renderpasses und Color Management, aber auch die neuesten Entwicklungen, unter anderem mit der Open Shading Language und der Universal Scene Description, Erwähnung.

Weiters befasst sich die Arbeit mit der Implementierung einer auf fotorealistisches Rendering ausgelegten Renderengine innerhalb einer Compositing-Applikation. Hierzu werden mögliche Konzepte zur Implementierung aufgezeigt und eine dieser umgesetzt; es folgt die Implementierung der Grundfunktionalitäten der Open-Source-Renderengine Cycles in die Compositing-Applikation Nuke von The Foundry durch den Verfasser.

Den Abschluss bildet ein Experteninterview, welches Aufschluss über den notwendigen Funktionsumfang einer derartigen Implementierung, als auch über mögliche Einsatzgebiete gibt.

Abstract

This thesis deals with the topic of photorealistic rendering inside of a compositing application.

Rendering describes the step of converting three dimensional content into a two dimensional image. In the beginning this thesis shows the most fundamental differences of the most common rendering algorithms; it offers detailed information on ray tracing as well as a general overview of the graphics pipeline.

Furthermore, the thesis takes a closer look on the fundamentals of an advanced rendering pipeline including basic concepts, like render passes and color management, and more recent developments, like the Open Shading Language and the Universal Scene Description.

The next chapter deals with the implementation of a render engine, which is dedicated to photorealistic rendering, inside of The Foundry's Nuke - different methods of implementing a render engine are mentioned at this point. Furthermore, this chapter describes the implementation of the open source render engine Cycles.

The last chapter includes an interview with an expert. This section discusses the required features and the possible use cases of photorealistic rendering inside of a compositing application.

Inhaltsverzeichnis

1 Einleitung

Das Bedürfnis nach Effizienz sowie die Unternehmensstruktur von kleinen Unternehmen haben die Hersteller von Software im Bereich der Animation und visuellen Effekte dazu veranlasst, mehr und mehr auf eine Diversifizierung der Anwendungsgebiete ihrer Software zu setzen. Während 3D-Applikationen wie SideFXs Houdini oder Blender nun auch die Möglichkeiten besitzen node-basiert Compositing-Aufgaben zu bewältigen, haben Compositing-Programme wie The Foundrys Nuke oder Blackmagic Designs Fusion einen vollwertigen 3D-Raum zur Lösung der immer komplexer werdenden Tasks inkludiert. Keine der Software für Compositing hat allerdings die Fähigkeit des fotorealistischen Renderings implementiert; im Regelfall sind die Rendering-Algorithmen lediglich für die Projektion von Bildinhalten beziehungsweise für die Verdeckungsberechnung ausgelegt. Im Sinne der Effizienz würde es sich allerdings anbieten, dass man kleinere 3D-Tasks auch in einer Compositing-Applikation umsetzen kann. Dies geht jedoch in den meisten Fällen mit der Notwendigkeit einher, dass der Content auch fotorealistisch gerendert werden kann.

1.1 Thematik und Problematik

Rendering stellt die Schnittstelle zwischen 3D und 2D dar, die Umwandlung von vektorbasierten, dreidimensionalen Szenenbeschreibung in ein zweidimensionales Bild. Dabei handelt es sich um einen Task, welcher stets dem 3D-Department zugeordnet wurde. Die laufend komplexer werdenden Aufgaben im Compositing führten zu einem immer umfassenderen Funktionsumfang von jenen Applikationen und schließlich auch zu der Implementierung von Grundfunktionalitäten von 3D-Räumen. Dies wirft die Frage auf, ob nun auch als eine Folge davon, der Prozess des Renderings in die Compositing-Applikation übertragen und so eine Steigerung in Effizienz und Qualität erreicht werden kann.

Es stellt eine Notwendigkeit dar, dass bei einer derartigen Verlagerung von Aufgaben aus einer 3D- in eine Compositing-Applikation, gleichwertige Tools zur Verfügung gestellt werden müssen, um gleichwertige Qualität erreichen zu

können. Diese Tools umfassen die Möglichkeit des fotorealistischen Renderings. Hierzu werden mittlerweile mehrere Lösungen von Drittanbietern angeboten. Diese Arbeit versucht die Generierung einer eigenen Lösung, um dieses Problem zu bewältigen.

1.2 Motivation

Ich selbst konnte mich immer sowohl für die Umsetzung von Aufgaben innerhalb eines zweidimensionalen als auch dreidimensionalen Kontextes begeistern. Eine Vermischung von diesen zwei Aufgabenbereichen wirkte aus meiner Sicht stets als ein natürlicher Schritt. Auch wenn sich meine Spezialisierung durch die Arbeit als Compositing-Artist mehr und mehr in Richtung des zweidimensionalen Arbeitens verschoben hatte, konnte ich mein großes Interesse an der Arbeit in einem dreidimensionalen Kontext beibehalten.

Aufgrund meiner Spezialisierung entwickelte ich allerdings auch immer mehr Interesse an grundlegendsten technischen Hintergründen und Funktionsweisen von Compositing-Applikationen. Auch stieg mehr und mehr das Interesse in Scripting und der Entwicklung von Tools und Plug-ins, um Adaptionen an der Software, angepasst an meine Nutzungsgewohnheiten, durchführen zu können.

Eine Kombination aus den oben angeführten Gründen stellt die Motivation zur Verfassung dieser Arbeit dar. Ich sehe darin eine Möglichkeit, mich in viele Richtungen weiterentwickeln zu können; neues Wissen und neue Fähigkeiten zu erlangen.

Auch besitzt die Thematik eine starke Aktualität, da, wie einleitend erwähnt, ein starker Trend in Richtung einer universellen, alles umfassenden Software voranschreitet. Erst die Implementierung der Compositing-Applikation Fusion in Blackmagic Designs DaVinci Resolve hat mir diesen Trend wieder deutlich vor Augen geführt. Die Zukunft wird zeigen, ob eine Software, welche alle Stationen einer Pipeline abdeckt, in einer professionellen, stark spezialisierten Umgebung zur Anwendung kommen kann. Die Verwendung einer Renderengine, welche fotorealistisches Rendering in einer Compositing-Applikation erlaubt, soll den ersten Schritt in dieser Entwicklung darstellen.

1.3 Ziele, Forschungsfragen und Hypothesen

Die Arbeit beschäftigt sich mit fotorealistischem Rendering innerhalb von Compositing Applikationen. In diesem Zusammenhang befasst sie sich mit verschiedenen Teilaspekten dieser Thematik, welche sich mit den folgenden Forschungsfragen zusammenfassen lassen:

- Welche technischen Unterscheidungen weisen die verschiedenen Rendering-Algorithmen auf?
- Welche Funktionalitäten würde die Renderengine benötigen, um eine Verbesserung in Workflow und Qualität zu erzielen?
- Welche sinnvollen Anwendungsgebiete lassen sich für die Verwendung einer raytrace-Renderengine innerhalb einer Compositing-Applikation ableiten?
- Wie kann der existierende Raytrace-Renderingalgorithmus von Blender (Cylces) in Nuke implementiert werden?

1.4 Methodik

Die Arbeit umfasst drei Teile, welche auf drei verschiedenen Methoden basieren. Die erste Methodik beschäftigt sich mit der Literaturrecherche, um Informationen zu den technischen Unterscheidungen der Renderengines sowie zu dem möglichen Funktionsumfang zu sammeln. Es soll anhand von Informationen aus der Literatur gezeigt werden, worin sich die vorhandenen Renderer unterscheiden, worin ihre Stärken und Schwächen liegen und wo ihre Verwendung dadurch sinnvoll ist.

Die zweite Methodik umfasst die Entwicklung eines Prototyps und zeigt auf, wie ein auf Fotorealismus orientierter Rendering-Algorithmus in Nuke implementiert werden kann. Hierbei wird beschrieben, welche Möglichkeiten es zur Implementierung gibt; eine dieser Möglichkeiten wird via eines Proof of Concept Prototypen umgesetzt.

Die dritte Methodik gibt ein Interview wieder, welches Aufschlüsse über die möglichen Anwendungsgebiete geben soll sowie noch einmal den notwendigen Funktionsumfang diskutieren. Die Methodik soll dabei dem eines systematisierenden Experteninterviews entsprechen. Die Auswertung erfolgt über eine Transkription, welche zur Auswertung kategorisiert und schließlich im Fließtext zusammengefasst wird. Die Transkription folgt dem möglichst genauen Wortlaut des Interviews.

1.5 Gliederung der Arbeit

Die Gliederung der Arbeit orientiert sich größtenteils an den Forschungsfragen und den Methodiken. Dabei werden die ersten beiden Kapitel die Literaturrecherche umfassen und die Forschungsfragen der technischen Unterschiede sowie des nötigen Funktionsumfanges behandeln. Das folgende Kapitel beschäftigt sich mit der Implementierung des Rendering-Algorithmuses und gibt einen Einblick in den Quellcode des Plug-ins. Das letzte Kapitel umfasst die aus dem Experteninterview gewonnenen Informationen zum nötigen Funktionsumfang einer derartigen Implementierung und den Anwendungsgebieten von fotorealistischen Renderings innerhalb einer Compositing-Applikation.

Im Anhang der Arbeit ist der ungekürzte Quellcode des Plug-ins sowie die Transkription des Experteninterviews enthalten.

Es gilt festzuhalten, dass in dieser Arbeit aus Gründen der leichteren Lesbarkeit lediglich die männliche Schreibweise Verwendung findet; angesprochen sind allerdings zu jedem Zeitpunkt beide Geschlechter sowie Menschen, die sich nicht in dieses heteronormale Geschlechtersystem einordnen lassen (wollen).

2 Rendering-Technologien

Rendering, in der deutschen Sprache auch als Bildsynthese bezeichnet, beschreibt den Prozess, bei welchem die geometrischen Daten einer zwei- beziehungsweise dreidimensionalen Szene in gewöhnliche gerasterte Bilder umgewandelt werden. Hierfür gibt es eine Vielzahl an Algorithmen, welche grundsätzlich in zwei verschiedene Arten unterschieden werden: object-order und image-order Rendering. Beim Rendering besteht die Notwendigkeit, innerhalb einer dreidimensionalen Szene den Zusammenhang zwischen der Geometrie sowie den einzelnen Pixeln darzustellen. Hierfür kann man entweder jedes Objekt in der Szene betrachten und überprüfen, welchen Anteil es an welchem Pixel des zu rendernden Bildes hat (object-order Rendering); oder man iteriert anstelle jedes Objektes über die einzelnen Pixel und untersucht, welches Objekt einen Anteil daran nimmt (Marschner, Shirley, 2016, p. 69).

Object-order Rendering gilt als effizienter - besonders bei weniger komplexen Szenen - und wird daher gerne bei Echtzeitanwendungen verwendet. Object-order Rendering wird auch als Grafikpipeline bezeichnet. OpenGL und auch Direct3D sind Programmierschnittstellen, welche eine derartige Grafikpipeline ansteuern (Marschner, Shirley, 2016, p. 159).

Image-order Rendering ermöglicht im Gegenzug eine einfachere Berechnung von Schattenwurf und korrekten Reflektionen, dies macht es bei fotorealistischen Anwendungsgebieten beliebt. Es gilt bei sehr komplexen Szenen als effizienter und benötigt weniger mathematisches Wissen als object-order Rendering (Marschner, Shirley, 2016, p. 69).

Das folgende Kapitel soll einen Einblick in die Grundlagen der Renderingalgorithmen bieten, welche einerseits fotorealistisches Rendering, wie mit der Renderingengine Cycles, ermöglichen, andererseits schnellere Renderzeiten, welche jener in The Foundrys Nuke implementierte Scanlinealgorithmus versucht umzusetzen. Anhand der Grundlagen der Funktionsprinzipien der einzelnen Algorithmen können anschließend in den folgenden Kapiteln verschiedene Anwendungsgebiete abgeleitet werden.

Das folgende Unterkapitel konzentriert sich auf das für fotorealistisches Rendering übliche Raytracing-Verfahren. Raytracing ist dem image-order Rendering zuzuordnen.

2.1 Raytracing

Bei Raytracing handelt es sich um ein Verfahren, bei welchem Sehstrahlen von einem gewissen Punkt (einem Beobachter beziehungsweise einer Kamera) ausgesendet und deren Gang durch eine dreidimensionale Szene verfolgt wird. Wie oben beschrieben handelt es sich hierbei um image-order Rendering; es werden durch jeden Pixel des zu berechnenden Bildes Sehstrahlen in die Szene geworfen. Grundsätzlich besteht der Raytracing-Algorithmus aus drei integralen Bestandteilen:

- der Strahlenerzeugung (ray generation) - zuständig für die Berechnung der Ursprünge sowie der Richtungen der Sehstrahlen für jeden einzelnen Pixel eines zu rendernden Bildes;
- der Berechnung der Kreuzungspunkte der Strahlen mit der in der Szene befindlichen Geometrie (ray intersection);
- und dem Shading - dem Berechnen der Farbe jedes Pixels des Bildes (Marschner, Shirley, 2016, p. 70).

Der Vorteil bei der Verwendung des Raytrace-Verfahrens ist die Tatsache, dass es relativ einfach ist, Reflektionen und Schattenwurf zu berechnen (Marschner, Shirley, 2016, p. 69).

2.1.1 Strahlenerzeugung/Ray Generation

Wie einleitend beschrieben ist die Strahlenerzeugung - im Englischen ray generation - der erste Schritt zur Berechnung eines gerasterten Bildes mit Hilfe des Raytracing-Verfahrens.

Die einfachste Abstraktion, um sich diesen Schritt vor Augen zu führen, ist die Betrachtung des Funktionsprinzips einer Lochkamera. Eine Lochkamera besteht aus einer lichtdichten Box, an deren einem Ende ein photosensitiver Film beziehungsweise ein Bildsensor ist und sich am gegenüberliegenden Ende ein kleines Loch befindet, durch welches Licht auf die Bildebene fallen kann. Stellt man sich nun vor, dass der Bildsensor anstelle hinter dem Loch der Lochkamera sich vor der Kamera befindet, können Strahlen vom Loch ausgehend durch den Bildsensor gezogen werden - die aus Loch und Bildsensor bestehende Pyramide kann als Betrachtungsvolumen (Viewing volume) bezeichnet werden (Pharr, et.al., 2017, p. 5).

Beim Raytracing-Verfahren besteht jeder Strahl aus einem Ursprung und einer Richtung. Im eben geschilderten Beispiel der abstrahierten Lochkamera handelt es sich beim Ursprung des Sehstrahles um das Loch der Kamera, bei der Richtung

um den Vektor, welcher vom Loch zum jeweiligen Pixel des Bildsensors gedacht werden kann (Pharr, et.al., 2017, p. 7).

Bei der Lochkamera handelt es sich um ein sehr einfaches Kameramodell für eine dreidimensionale Szene, welche allerdings häufigen Einsatz in 3D-Applikationen findet; es ist allerdings zu beachten, dass es wichtige Effekte einer Kamera und deren Optik vernachlässigt. So weist die Lochkamera etwa keine Tiefenunschärfe auf: alles ist scharf abgebildet, im Fokus. Weiters können bei komplexeren Kameramodellen zur Strahlenerzeugung Effekte der Optik simulierte werden, beispielsweise chromatische Aberrationen, Vignettierungen und auch anamorphe Optiken. Selbiges gilt für Bewegungsunschärfe. Auch muss die Sensorfläche, durch welche die Strahlen geworfen werden, nicht eine flache Ebene sein, sie kann beispielsweise auch kugel- oder zylinderförmig sein - derartiges findet etwa bei panoramaartigen Renderings Einsatz (Pharr, et.al. 2017, p. 355).

Liegt der Ursprung der einzelnen Strahlen in einem gemeinsamen Punkt spricht man von einer perspektivischen Projektion. Eine zweite weit verbreitete Projektion ist die orthographische Projektion (Pharr, et.al. 2017, p. 358).

Weisen die Strahlen alle den selben Richtungsvektor auf, sprich sie sind parallel, dann handelt es sich um eine derartige orthographische Projektion. In diesem Fall gibt es keinen gemeinsamen Betrachtungspunkt wie das beschriebene Loch der Lochkamera bei der perspektivischen Projektion; die Strahlen teilen sich keinen gemeinsamen Ursprung. Den Ursprung der Strahlen bildet in diesem Fall hingegen jeder Pixel selbst. Eine derartige Projektion hat zum Unterschied, dass keine perspektivische Verzerrung der Geometrie auftritt; sprich Objekte, die weiter weg von der Kamera sind, wirken nicht kleiner und zueinander parallele Linien in der Szene sind auch am Bild parallel. Folglich ist eine orthographische Projektion eher für technische Darstellung, etwa in CAD-Programmen, geeignet (Marschner, Shirley, 2016, p. 74).

2.1.2 Kreuzungspunkte/Ray Intersections

Ist nun der Strahl, bestehend aus Ursprung und Richtung, bekannt, muss der Kreuzungspunkt, welcher am nächsten zur Kamera ist, zwischen dem Strahl und der Geometrie in der Szene gefunden werden. Um den Schnittpunkt zu finden ist es grundsätzlich notwendig, den Strahl gegen alle in der Szene befindlichen Objekte zu testen und anschließend den Punkt mit der kürzesten Distanz zur Kamera auszuwählen. Da eine derartige brute-force Methode über die Maßen ineffizient und dadurch langsam wäre, werden im Regelfall Strukturen zur

Beschleunigung der Berechnung des Kreuzungspunktes implementiert (Pharr, et.al. 2017, p. 7).

Die bekanntesten Strukturen zur Beschleunigung der Schnittpunktermittlung sind BVH (Bounding Volume Hierachies) und KdTree. BVH basiert auf der Generierung von Bounding Boxen (Hüllkörpern), welche die einzelnen Objekte in der Szene umschließen (Pharr, et.al. 2017, p. 248).

KdTree-Beschleunigung basiert ebenfalls auf Bounding Boxen. Im Gegensatz zu BVH wird allerdings nicht jedem einzelnen Objekt eine Bounding Box zugeordnet, sondern zuerst der ganzen Szene. Übersteigt die Anzahl der Objekte innerhalb dieser Box einen gewissen Grenzwert, wird die Bounding Box weiter in zwei Hälften unterteilt. Objekte werden dabei immer der Hälfte zugeordnet, welche sie umschließt. Haben mehrere Bounding Boxen Anteil am Objekt, wird das Objekt beiden zugeordnet (Pharr, et.al. 2017, p. 284).

Somit können - sowohl bei BVH als auch KdTree - ganze Gruppen an Objekten, welche der Strahl mit einem großen Abstand verfehlen würde, ausgeschlossen werden.

Es gibt eine Vielzahl anderer Methoden zur Beschleunigung des Prozesses, allerdings verwenden fast alle Renderengines heutzutage eine dieser zwei Methoden (Pharr, et.al. 2017, p. 248).

Die Renderengine Cycles verwendet beispielsweise eine Open-Source Implementierung von einem weiter verbessertem BVH-Verfahren, welches ursprünglich von NVidia stammt und durch die Cycles-Developer um Funktionen unter anderem in den Bereichen des Instancing und der Bewegungsunschärfe, erweitert wurde. Laut van Lommel soll es dabei konkurrenzfähiger als KdTree sein (van Lommel, Blender Foundation, 2018).

Ist nun bekannt, gegen welche Objekte der Szene man den Strahl auf einen Schnittpunkt überprüft, gibt es verschiedene Algorithmen um dies zu tun. Welchen Algorithmus man hierzu verwendet, steht grundsätzlich in einer Abhängigkeit von der Art der zu testenden Geometrie der Szene (Marschner, Shirley, 2016, p. 76).

Die einfachste Art einer gekrümmten Oberfläche, welche für eine Renderengine eine gegebene Berechtigung hat, ist die Definition dieser über eine Funktion- es handelt sich um sogenannte Quadriks (Pharr, et.al. 2017, p. 131).

Als Beispiel sei die Kugel (aufgrund der Tatsache der Einfachheit zur Schnittpunktberechnung im Folgenden genauer erklärt) genannt, welche sich über die implizite Gleichung $(x-xc)^2+(y-yc)^2+(z-zc)^2-R^2=0$ darstellen lässt. xc, yc und zc

beschreiben hierbei den Mittelpunkt der Kugel, R den Radius der Kugel und x, y und z einen beliebigen Punkt. Erfüllt der beliebige Punkt die oben genannte Gleichung, so befindet sich jener auf der Oberfläche der Kugel. Ist nun ein Strahl p(t)=e+td gegeben, so kann dieser in die implizite Gleichung der Kugel eingesetzt und diese Gleichung in Richtung t gelöst werden (Marschner, Shirley, 2016, p. 76).

Das Ergebnis der Gleichung kann aus entweder zwei Lösungen bestehen (wenn der Strahl die Kugel durchstößt), wobei das niedrigste, positive Ergebnis den gesuchten Schnittpunkt ergibt; aus einer Lösung (wenn der Strahl die Kugel berührt); beziehungsweise aus keinem Ergebnis (wenn der Strahl die Kugel verfehlt) (Pharr, et.al. 2017, p. 7).

Wohl am weitesten verbreitet beim Rendering von Geometrie ist die Berechnung von Schnittpunkten mit Dreiecken. Hierfür gibt es eine Vielzahl an Algorithmen (Marschner, Shirley, 2016, p. 77).

Ein bekanntes Modell hierzu wurde 1987 von Snyder und Barr vorgestellt, welches eine besonders speichereffiziente Methode zur Parkettierung von komplexen dreidimensionalen Objekten in eine Vielzahl kleinerer Dreiecke und einer anschließenden Schnittpunktberechnung beschreibt. Dieses Modell ermöglichte es erstmals komplexe dreidimensionale Objekte via Raytracing zu rendern, welche aus mehreren Millionen Dreiecken bestanden (Snyder, Barr, 1987, p. 119).

Eine geläufige Praxis bei der Bildsynthese ist oft die Umwandlung von Geometrie in Dreiecke, da deren Berechnung oft einfacher ist als von Polygonen (Marschner, Shirley, 2016, p. 81).

Die Raytrace-Engine von Mantra der Firma Side Effects, um ein Beispiel zu nennen, besitzt verschiedene Algorithmen zur effizienten Schnittpunktberechnung von Punkten, Kreisen, Kugeln, Zylinder, Polygonen und Polygonenetzten. Ist Geometrie zu rendern, welche nicht nativ unterstützt wird, wird sie weiterverarbeitet (Seymour, 2013).

Hat man den Schnittpunkt ermittelt ist das allerdings noch nicht genug für den restlichen Raytrace-Algorithmus. Es werden weitere Parameter benötigt, welche später an den letzten Schritt des Renderings - das Shading - übergeben werden müssen. Dabei handelt es sich einerseits um Angaben bezüglich des Materials des Objektes, andererseits um weitere Informationen zum Schnittpunkt. So ist es fürs Shading stets notwendig, dass man die Surface Normal dem Raytrace-Algorithmus übergibt. Anspruchsvollere Renderengines benötigen oft noch weitere Informationen zum Schnittpunkt (Pharr, et.al. 2017, p. 7).

2.1.3 Shading

Da nun bekannt ist, welche Oberfläche für einen bestimmten Pixel sichtbar ist, kann nun die Darstellung dieser Oberfläche berechnet werden. Hierfür gibt es eine Vielzahl an Shadingmodellen, die in ihrer Komplexität stark variieren können.

Die Grundaufgabe der meisten Shadingmodelle gestaltet sich so, dass sie den Farbwert eines Pixels berechnen, indem sie die Menge des reflektierten Lichtes einer Oberfläche einer beziehungsweise mehrerer Lichtquellen in Richtung der Kamera in Betracht ziehen (Marschner, Shirley, 2016, p. 81).

Diese Funktionen, welche die lichtreflektierenden Eigenschaften der Oberfläche und daraus folgend ihre Erscheinung bestimmen, werden bidirectional reflectance distribution function - kurz BRDF - genannt (Pharr, et.al. 2017, p. 10).

Das folgende Kapitel soll einen Überblick über die Grundprinzipien des Shadings geben, kann aber aufgrund des immensen Umfangs dieses Themas keinen Anspruch auf Vollständigkeit erheben.

2.1.3.1 Lambertian Shading

Das wohl einfachste Shadingmodell, welches auf einer BRDF basiert, ist das Lambertian Shadingmodell. Es basiert auf einer Beobachtung Lamberts im 18ten Jahrhundert. Er beobachtete, dass die Menge der Energie eines Lichtes, welche von einer Lichtquelle auf die Oberfläche eines Objektes geworfen wird, direkt in einem Zusammenhang mit dem Winkel des Objektes zur Lichtquelle steht. Die Oberfläche, welche direkt zur Lichtquelle zeigt, weist hierbei die maximale Menge an reflektiertem Licht auf. Jene Oberfläche wiederum, die normal zur Lichtquelle beziehungsweise parallel zur Lichtrichtung steht, erhält keinerlei Beleuchtung. Die Werte der Beleuchtung zwischen diesen beiden Extremen verhalten sich proportional zum Cosinus des Winkels zwischen der Normalen der Oberfläche und dem Vektor von der Oberfläche zur Lichtquelle. Der Cosinus lässt sich einfach durch das Skalarprodukt der normalisierten Vektoren ableiten. Jene Berechnung wird für jede Farbe des additiven Farbsystems, Rot, Grün und Blau, durchgeführt (Marschner, Shirley, 2016, p. 82).

Es ist anzumerken, dass diese Annahmen auf der Verwendung eines Punktlichtes basieren, was grundsätzlich in der realen Welt so nicht vorkommen kann. Physikalisch korrekte Beleuchtung basiert im Regelfall auf Flächenlichtern. Aufgrund einer einfachen Darstellung der Prinzipien kommt allerdings meist ein Punktlicht zum Einsatz (Pharr, et.al. 2017, p. 8).

Weiters ist festzuhalten, dass es besonders einfach ist, mit mehreren Lichtquellen zu arbeiten, da Beleuchtung linear ist. Folglich kann die Menge an Licht, welche auf die Oberfläche des Objekts fällt, für jede Lichtquelle in der Szene separat berechnet und anschließend miteinander addiert werden (Pharr, et.al. 2017, p. 9).

Das Lambertian Shadingmodell eignet sich lediglich dazu, den Diffusanteil des Shadings einer Oberfläche zu berechnen. Die Farbe eines Objektes ist nicht von der Blickrichtung abhängig (Marschner, Shirley, 2016, p. 82).

2.1.3.2 Blinn-Phong Shading

Aufgrund der Tatsache, dass sich der Lambertsche Shader lediglich für Berechnungen des Diffusanteils eines Objeketes eignet, die meisten Oberflächeneigenschaften von Objekten allerdings auch einen spiegelnden Anteil haben, wurde das Modell von Lambert 1975 von Phong beziehungsweise 1976 von Blinn um einen Glanzpunkt erweitert. Jenes Modell besagt, dass die Reflektion des Lichtes am stärksten ist, wenn der Sehstrahl von Kamera und Schnittpunkt den gleichen Einfallswinkel zur Oberflächennormalen hat wie der Vektor von Lichtquelle zu Schnittpunkt. Dies würde einer Spiegelreflektion entsprechen. Die Stärke des Glanzlichtes nimmt dabei sanft mit der Veränderung der Vektoren ab, sollten diese ihre symmetrischen Positionen verlassen. Dies kann berechnet werden, indem man den Vektor ermittelt, welcher exakt zwischen den beiden Vektoren von Sehstrahl und Lichtstrahl liegt und dessen Richtung man anschließend mit der Oberflächennormalen vergleicht. Ist das Skalarprodukt dieser beiden normalisierten Vektoren 1 - das Maximum - handelt es sich um volle Symmetrie; die beiden Einfallswinkel sind gleich und das Glanzlicht ist am hellsten. Dieses Ergebnis kann als Basis einer Potenz genommen werden, wobei der Exponent die Geschwindigkeit des Abfalls des Glanzlichtes angibt. Somit kann eine Rauheit der Oberfläche simuliert werden. Es gilt, umso größer der Exponent, desto größer der Abfall des Glanzlichtes, desto weniger rau ist die Oberflächenbeschaffenheit des Objektes. Als Richtwert sind Werte zwischen 10 (Eierschale) bis 10000 (annähernd spiegelnd) anzunehmend (Marschner, Shirley, 2016, p. 82).

Heutzutage wird meist rekursives Raytracing zur Berechnung von spiegelnden Reflektionen auf Oberflächen verwendet. Hierzu wird der Strahl, sowie er auf eine Oberfläche trifft, in einen sekundären Strahl aufgebrochen und im gleichen Winkel zum Lot, im beschriebenen Fall der Normalvektor, weggebrochen (Marschner, Shirley, 2016, p. 324).

2.1.3.3 Ambient Shading

Da Oberflächen, welche sich nicht im Blickfeld einer Lichtquelle befinden, als komplett schwarz dargestellt werden, gibt es die Möglichkeit, einen konstanten Wert hinzu zu addieren, um den Effekt von Umgebungslicht beziehungsweise eine Aufhellung durch andere Objekte zu imitieren (Marschner, Shirley, 2016, p. 82).

2.1.3.4 Schatten

Das Verhalten von Licht, wie oben beschrieben, missachtet einen wichtigen Punkt: So hat eine Lichtquelle am Shading einer Oberfläche nur dann einen Anteil - indirektes Licht ausgenommen - wenn der Vektor vom Schnittpunkt des Sehstrahles mit der Oberfläche zur Lichtquelle nicht von einem weiteren Objekt blockiert wird. Bei einem Raytracer ist diese Überprüfung ohne größeren Aufwand möglich, indem man einfach den neu generierten Strahl - shadow ray genannt - auf Schnittpunkte mit anderen Objekten testet. Dabei muss die Entfernung eines möglichen Schnittpunktes mit der Entfernung des Ursprungs des Strahles verglichen werden, um sicher zu gehen, ob sich das Objekt vor oder hinter dem Licht befindet (Pharr, et.al. 2017, p. 10).

2.1.3.5 Distribution Ray Tracing

Das im Kapitel “Schatten” beschriebene klassische Modell zur Berechnung eben dieser Schatten bietet den Nachteil, dass es lediglich die Berechnung von scharfkantigem Schattenwurf ermöglicht. Die Berechnung von weichem Schatten ist eine Notwendigkeit, da sogut wie alle Lichter in der realen Welt eine Fläche als Form aufweisen und eben diese weiche Schattenkanten produzieren. Hierzu wurde 1984 das Distribution Ray Tracing durch Cook vorgelegt. Es bietet neben weichen Schattenkanten auch die Möglichkeiten, Kanten mit Kantenglättung (Antialiasing) darzustellen, Reflektionen von nicht hundertprozentig spiegelnden Oberflächen zu berechnen sowie Tiefen- und Bewegungsunschärfe in das zu rendernde Bild zu implementieren (Marschner, Shirley, 2016, p. 329).

Vor der Verwendung von Distribution Ray Tracing neigte man dazu, ein Flächenlicht durch ein Array mehrerer Punktlichter zu simulieren. Dies resultierte allerdings in Abstufungen im Wurf des weichen Schattens. Cook beschrieb durch Distribution Ray Tracing ein Verfahren, welches zufällige Punkte auf einer Fläche generiert und die Sichtbarkeit jener überprüft. Anschließend wird der Durchschnitt jener gebildet. Jenes Verfahren erwies sich als effizienter, mit einem besseren Ergebnis (Marschner, Shirley, 2016, p. 331).

Zur Berechnung eines anitaliasten Bildes wird im Distribution Ray Tracing anstelle der Berechnung des Pixelwertes durch einen einzelnen Sehstrahl, welcher aus

dem Zentrum des Pixels ausgesendet wird, mehrere ausgesendet, aus welchen im Anschluss der Durchschnitt berechnet wird. Die einzelnen Sehstrahlen des Pixels werden hierzu auf der Fläche des Pixels bis zu einem gewissen Grad zufällig verteilt, um das Auftreten von Moiré-Effekten zu vermeiden (Marschner, Shirley, 2016, p. 330).

Für die Darstellung von Reflektionen auf rauen Oberflächen durch das Distibution Ray Tracing-Verfahren werden lediglich mehrere sekundäre Strahlen ausgesendet, welche durch einen zufälligen Störfaktor, dessen Amplitude dem Index der Rauheit der Oberfläche entspricht, abgelenkt werden. Es wird wieder der Durchschnitt jener sekundären Strahlen gebildet (Marschner, Shirley, 2016, p. 333).

Die Berechnung von Tiefen- sowie Bewegungsunschärfe basiert auf einfachen Annahmen, wobei einerseits bei der Tiefenschärfe in Betracht gezogen wird, dass Licht nicht in einem Punkt, sondern in einer nicht-null großen Optik gesammelt wird, andererseits bei Bewegungsunschärfe, wobei Strahlen über eine Zeitspanne ausgesendet werden (Marschner, Shirley, 2016, p. 332).

2.1.3.6 Lichtabnahme/Lichtverteilung

Ein wichtiger Punkt in Bezug auf Shading ist die Berücksichtigung der Annahme, dass Licht mit der Funktion $1/r^2$ über die Entfernung r abnimmt. Aufgrund der Tatsache, dass es sich bei der oben gegebenen Lichtquelle um ein Punktlicht handelt, welches unendlich klein ist und in alle Richtungen - kugelförmig - gleichermaßen abstrahlt, ist die Darstellung dieser Annahme leicht. Vorzustellen ist eine kugelförmige Geometrie, welche die Lichtquelle umschließt. Vergrößert man den Radius dieser Kugel so vergrößert sich die Fläche, auf welche sich die Energie der Lichtquelle verteilt. Die Helligkeit der Beleuchtung nimmt für die einzelnen Punkte der Oberfläche ab (Pharr, et.al. 2017, p. 8).

2.1.3.7 Fortgeschritterne Reflektionsmodelle/BRDFs

Die oben genannten Shadingmodelle sind kaum für photorealistisches Rendering zu gebrauchen, da sie wichtige physikalische Prinzipien außer Acht lassen. Um ein weiteres Shadingmodell mit bidirectional reflectance distribution function zu nennen, welches um eine Vielzahl von Funktionen erweitert wurde: das Modell von Ashikhmin und Shirley von 2000, welches unter anderem auch einer der BRDFs ist, die in der Blenders Cycles zu finden sind. Ashikhmin und Shirley (2000, p. 1) beschreiben eine Funktion, die ihren Diffuseanteil nicht durch eine von Lampert abgeleitete Funktion berechnet. Dies ermöglicht es, Energieerhaltung des Lichtes auch bei Fresnelverhalten (flach einfallendes Licht verhält sich anders als steiler

einfallendes) zu gewährleisten. Weiters ergänzt es das Phong-Modell um eben dieses Fresnelverhalten sowie um den Funktionsumfang der Anisotropy. Es handelt sich hierbei um einen Algorithmus, der physikalisch plausibles Shading ermöglicht. Man spricht von physically-based rendering (Ashikhmin, Shirley, 2000, p. 1).

Es gibt auch andere Modelle, welche unter anderem auch Volumenstreuung, subsurface scattering, in Objekten mit einbeziehen. Man spricht von BSSRDFs, bidrirectional subsurface scattering reflactance distribution function (Pharr, et.al. 2017, p. 11).

2.1.3.8 Transparenzen und Refraktion

Bei der Berechnung von transparenten Oberflächen kommt es ebenfalls, wie schon bei der Berechnung von Spiegelungen, zum Einsatz von rekursiven Raytracing. Der primäre Sehstrahl, welcher vom Betrachter, beziehungsweise einer Kamera, ausgesendet wird, wird beim Auftreffen auf eine Oberfläche in Sekundärstrahlen aufgebrochen. Einer von diesen dringt hierbei in das Objekt, sollte es transparent sein, ein. Handelt es sich beim Eindringen um einen Übergang von zwei Medien mit unterschiedlicher Ausbreitungsgeschwindigkeit der Lichtwelle, zum Beispiel von Luft in Wasser oder Glas, so kommt es zur Brechung. Hierzu wird der sekundäre Strahl in einem bestimmten Winkel abgelenkt; jener Winkel lässt sich aus den jeweiligen Brechungsindizes der jeweiligen Medien berechnen - das Snelliussche Brechungsgesetz kommt zum Einsatz (Marschner, Shirley, 2016, p. 324).

2.1.3.9 Global Illumination

Um fotorealistische Ergebnisse im Rendering zu ermöglichen ist es notwendig, die globale Beleuchtung (Global Illumination) in Betracht zu ziehen. Die globale Beleuchtung, oft auch als indirekte Beleuchtung oder indirektes Licht beschrieben, beschreibt das Licht, welches von einer Lichtquelle emittiert und von der Oberfläche eines Objektes auf die Oberfläche eines anderen Objektes reflektiert wird. Besondere Notwendigkeit erhält diese Gegebenheit beispielsweise bei Renderings von Räumen; oft erhält etwa die Decke keinerlei direktes Licht - wenn zum Beispiel eine Deckenlampe lediglich nach unten strahlt - die Beleuchtung findet nur durch Reflektion des Lichtes durch andere Objekte statt (Marschner, Shirley, 2016, p. 613).

Eines der ersten Modelle, welches eine globale Beleuchtung ermöglichte, wurde 1979 von Turner Whitted vorgestellt. Er beschrieb einen rekursiven Raytracing-Algorithmus: der von der Kamera ausgesendete Strahl erzeugt beim Auftreffen auf

einer Oberfläche mindestens drei weitere, sekundäre Strahlen; für Reflektion, Refraktion sowie für die Lichtquelle. Dieser Prozess wird rekursiv für die sekundären Strahlen fortgesetzt bis ein ausreichendes Level an Akkuratheit erreicht ist. Das Ergebnis wird am Ende im Pixelwert des primären Strahles berücksichtigt (Whitted, 1980, p. 343).

Es gibt eine Vielzahl an weiteren Verfahren (unter anderem: Cook (1984), welcher als erster eine Monte-Carlo-ähnliche Technik zur Berechnung anwendete; Goral, Torrance, Greenberg und Battaile (1984); Immel (1986)), welche zu einem Großteil Weiterentwicklungen der bestehenden Modelle darstellen (Marschner, Shirley, 2016, p. 613).

Sie alle stellen die Grundlage für Pathtracing dar, welches 1986 durch Kajiya beschrieben wurde. Pathtracing ist besonders geeignet für Szenen mit BRDFs und einer Vielzahl an Objekten (Kajiya, 1986, p. 143).

Pathtracing ist das heute übliche Verfahren zur Berechnung globaler Beleuchtung. Es verwendet den Monte-Carlo-Algorithmus, welcher eine Annäherung des Ergebnisses der Lichtransportfunktion liefert; er verwendet hierzu Zufälligkeit und Wahrscheinlichkeit. Es ist ein hohes Maß an Samples notwendig, um den Pixelwert mit ausreichend hoher Wahrscheinlichkeit festlegen zu können und ein annähernd rauschfreies Bild zu ermöglichen (Marschner, Shirley, 2016, p. 617).

Moderne Renderer mit Pathtracing verwenden in den meisten Fällen bidirektionale Methoden. Hierzu wird sowohl ein Strahl von der Kamera ausgesendet als auch von der Lichtquelle selbst. Dieses Verfahren kann weit effizienter sein, einerseits wegen seiner Natur der Bidirektionalität, andererseits auch durch die Möglichkeit der Anwendung fortgeschrittenerer Methoden zur Gewichtung von Pfaden (Pharr, et.al. 2017, p. 947).

Pathtracing besitzt nicht nur die Möglichkeit indirektes Licht zu berechnen, sondern kommt automatisch auch mit der Funktion die Einstrahlung von direktem Licht zu erheben. Dennoch verwenden fast alle gängigen Renderer traditionelles Raytracing zur Brechnung des direkten Lichtanteils; aus Gründen der Effizienz. Weiters bietet es die Möglichkeit weiche Schattenkanten zu berechnen (Marschner, Shirley, 2016, p. 624).

2.2 Die Grafikpipeline/Rasterization

Wie einleitend erwähnt stellt neben image-order Rendering, wozu Raytracing zu zählen ist, object-order Rendering die zweite große Kategorie an Renderingalgorithmen dar.

Die Idee von object-order Rendering ist es, jedes Objekt nach und nach auf das zu rendernde Bild zu übertragen. Der Prozess jeden Pixel zu finden, an welchem das geometrische Objekt einen Anteil hat, nennt man Rasterization. Die Abfolge an Operatoren, welche notwendig sind um das Bild zu berechnen, nennt man Graphics Pipeline. Derartige Pipelines können sowohl Hardware basierend sein, via APIs wie OpenGL oder Direct3D, oder auch auf Software basieren, wie frühere Versionen von RenderMan zeigen. Während Hardwarepipelines auf Effizienz und Echtzeit getrimmt sind, müssen Sotwarepipelines, auch Productionpipelines genannt, für höchste Qualität und Skalierbarkeit, um auch enorm große Szenen rendern zu können, ausgelegt sein (Marschner, Shirley, 2016, p. 159).

Ein bekanntes Beispiel für eine Renderalgorithmus, welcher auf einer Softwarepipeline basiert, stellt Reyes von Pixars RenderMan dar. Reyes wurde 1987 von Cook, Carpenter und Catmull in ihrem Paper “The Reyes Image Rendering Architecture” beschrieben und ermöglichte es erstmals extrem komplexe Szenen zu rendern. (Cook, Carpenter, Catmull, 1987, p. 95).

2002 implementierte Pixar Rendering via Raytracing in RenderMan, welches erstmals bei dem Film “Cars” ausgiebig zum Einsatz kam (Christensen, Fong, Laur, Batali, 2006 p. 1).

Dana Batali beschreibt in einem Interview, dass RenderMan zum Teil einen hybriden Ansatz für den Film “Die Monster Uni” (2013) verwendete, bei welchem sowohl Raytracing als auch Reyes - überwiegend für das Rendering von Haaren und Fell - verwendet wurden. Der Einsatz verschiebt sich allerdings immer mehr in Richtung Raytracing. RenderMan ist nicht der einzige Renderer, welcher Reyes verwendet. So ist beispielsweise auch der Reyes-Algorithmus in SideFXs Renderer Mantra, unter dem Name „Micropolygon Rendering“, implementiert (Seymour, 2013).

The Foundrys Nuke besitzt als Standardrenderer einen Scanline-Algorithmus, welcher eine derartige Grafikpipeline implementiert. Scanline bezieht sich dabei lediglich auf die Reihenfolge der Verarbeitung der einzelnen Pixel; sie werden Reihe um Reihe verarbeitet. RenderMan ist auch Scanline basierend.

Im Folgenden wird eine kurze Einführung in die Grundlagen des Rendering via Rasterization in einer Grafikpipeline gegeben.

2.2.1 Aufbau der Grafikpipeline

Grundsätzlich lässt sich eine Grafikpipeline in vier Hauptphasen unterteilen: Vertex Processing, Rasterization, Fragment Processing und Blending. Die geometrischen Objekte innerhalb einer Szene werden im Regelfall als eine Menge an Vertices von einer Applikation oder einer Szenenbeschreibung an die Pipeline übergeben. Dabei werden sie in der Vertexphase verarbeitet und an die Rasterizationphase gesendet. In der Rasterizationphase wird die Geometrie in eine Vielzahl kleiner Fragmente - für jeden Pixel des zu rendernden Bildes - aufgebrochen. Die Fragmente werden in der Phase des Fragment Processings verarbeitet und in der Blending-Phase zusammengefügt (Marschner, Shirley, 2016, p. 160).

Es folgt eine ausführlichere Darstellung der einzelnen Phasen, welche bei der Grafikpipeline zum Einsatz kommen.

2.2.1.1 Vertex Processing

Vertex Processing stellt die erste Phase der Grafikpipeline dar. Es nimmt die Information der einzelnen Vertices aus einer Applikation beziehungsweise einer Szenenbeschreibung entgegen. Die Hauptaufgabe dieser Phase ist es, diese Information weiter zu verarbeiten - für die Rasterization-Phase aufzubereiten. Ziel ist es, die Vertices durch verschiedenste Transformationen (modeling, viewing and projection transformations) von ihren originalen Koordinaten in Bildkoordinaten umzuwandeln. Weiters werden zusätzlich Attribute und Informationen, wie die Farbe der Vertices, die Oberflächennormale und Texture Koordinaten, ebenfalls transformiert. Die Ergebnisse werden an die nächste Phase, Rasterization, übergeben (Marschner, Shirley, 2016, p. 171).

2.2.1.2 Rasterization

Die Phase der Rasterization stellt den zentralen Part der Grafikpipeline da. Es nimmt die Informationen der Vertex Processing-Phase entgegen und bricht mit Hilfe von Interpolation der Werte die Geometrie in kleinere Fragmente auf, welche jeweils einem Pixel zugeordnet werden können. Bei diesen Fragmenten handelt es sich im Regelfall um Dreiecke. Dieser Schritt ist gleich zu dem Prozess der Rasterization von zweidimensionalen Bildern; es wird auch mehr und mehr zum Standard, dass 2D-Applikationen eine 3D-Graphikpipeline im Hintergrund verwenden (Marschner, Shirley, 2016, p. 160).

Bei der Rasterization wird Clipping verwendet um Vertices, welche hinter der Kamera liegen, abzuschneiden, da sonst Fehler in jener Phase auftreten würden. Es müssen zumindest die Vertices hinter der Kamera entfernt werden, im Regelfall

werden allerdings alle Vertices gelöscht, welche außerhalb des Viewingvolumen liegen (Marschner, Shirley, 2016, p. 167).

Hierbei spricht man von Culling. Es gibt verschiedenste Culling-Verfahren zur Steigerung der Effizienz. Wie eben erwähnt View Volume Culling, Occlusion Culling - es werden jene Teile der Geometrie entfernt, welche durch andere Geometrie verdeckt werden - und Backface Culling - der Entfernung von Geometrie, welche nicht in Richtung der Kamera zeigt (Marschner, Shirley, 2016, p. 179).

2.2.1.3 Fragment Processing und Blending Phase

Fragment Processing ist jene Phase, welche die Ergebnisse des Rasterizatione-Abschnittes verarbeitet und in Bezug auf Farbe und Tiefe evaluiert. Die Blending-Phase fügt zum Ende der Pipeline die einzelnen Fragmente zusammen, um den finalen Farbwert des Pixels des zu rendernden Bildes zu berechnen. Die Evaluierung der Tiefeninformation der einzelnen Fragmente ist notwendig, sollten mehrere Fragmente einem einzelnen Pixel zugeordnet worden sein. Hierzu gibt es einerseits den Painter's Algorithm. Bei jenem werden Objekte nach der Reihe abgebildet, angefangen beim am weitesten entfernten Objekt bis zum nächsten (back-to-front order). Dieser Algorithmus kann allerdings keine sich schneidenden Objekte berücksichtigen, noch Objekte, die um andere Objekte gebogen sind (Marschner, Shirley, 2016, p. 172).

Um jene Probleme, welcher der Painter's Algorithm mit sich bringt zu umgehen, hat sich die z-Buffer-Methode etabliert. Jene Methode besteht daraus, dass für jeden Pixel neben der Farbinformation (Rot, Grün und Blau), auch die Tiefeninformation des letzten abgebildeten Fragments abgespeichert wird. Ein ab zu bildendes Fragment wird lediglich nur dann für den jeweiligen Pixel dargestellt, wenn der Tiefenwert näher zu der Kamera ist als das zuletzt Abgebildete. Das zusätzlich zur RGB-Information gespeicherte Array an Tiefeninformation wird z-Buffer genannt. Der z-Buffer-Algorithmus ist in der Blending-Phase implementiert. Die Tiefeninformation der einzelnen Fragmente wird durch einfache Interpolation des Tiefenattributes (z-Koordinate) der Vertices berechnet (Marschner, Shirley, 2016, p. 172).

Das Shading der Geometrie passiert bei object-order Rendering entweder in der Fragment Processing-Phase für jedes Fragment - man spricht von Per-Fragment Shading - oder in der Vertex-Phase - Per-Vertex Shading. Beim Per-Vertex Shading wird die Farbinformation lediglich für jeden Vertex berechnet und per Interpolation auf die einzelnen Fragmente übertragen, bei Per-Fragment Shading werden die Shading-Gleichungen für jedes Fragment berechnet. Bei Per-Vertex

Shading kann es aufgrund Interpolation zu Artefakten kommen (Marschner, Shirley, 2016, p. 175).

Aufgrund der Tatsache, dass viele der Objekte in Echtzeitanwendung aus wenigen Vertices bestehen, kommt bei der Hardwarepipeline häufig Per-Fragment Shading zum Einsatz (Marschner, Shirley, 2016, p. 177).

RenderMan hingegen, ein Renderer, welcher für fotorealistische Abbildungen ausgelegt ist, verwendet Per-Vertex Shading. Hierzu zerlegt er die abzubildende Geometrie in Micropolygons, kleine Vierecke. Diese Zerlegung nennt man Dicing. Die Größe dieser Micropolygones entspricht hierbei einem halben Pixel. Ein halber Pixel erfüllt die Nyquist Grenze und das Shading kann folglich auf eine einzelne Farbe pro Micropolygon reduziert werden (Cook, Carpenter, Catmull, 1987, p. 97).

2.2.2 Antialiasing

Genauso wie bei Raytracing produziert der Standard Algorithmus bei der Grafikpipeline ein aliastes Bild, da lediglich ein zentrales Sample pro Pixel genommen wird. Der häufigste verwendete Algorithmus um die Kanten eines gerasterten Bildes zu glätten, ist die Methode des box filterings. Hierzu wird das Bild in der vierfachen Auflösung gerendert (oversampeling) und anschließend über den Durchschnitt einer 4x4 Matrix auf die Zielauflösung reduziert (Marschner, Shirley, 2016, p. 178).

3 Advanced Rendering-Pipeline

Im vorangegangen Kapitel wurden die grundlegendsten technischen Unterschiede der verschiedenen gebräuchlichen Renderalgorithmen aufgezeigt. Renderengines unterscheiden sich allerdings nicht nur in ihren zugrundeliegenden Prinzipien, sondern auch maßgeblich in ihren darüber hinaus laufenden Funktionsumfang, welche dem Anwender weiterführende Möglichkeiten bietet mit computergenerierten Content zu arbeiten.

Im folgenden Kapitel sollen weitere Aspekte dargebracht werden, worin sich Renderengines unterscheiden; sowohl in ihrer Bedienung, als auch in den Möglichkeiten der Weiterverarbeitung ihrer Ergebnisse. Speziell im Hinblick auf die Verwendung von Renderingalgorithmen innerhalb von Compositing-Applikationen zeigt der Umfang der Möglichkeiten in der Weiterverwertung von Ergebnissen der Renderer große Wichtigkeit.

3.1 Shading Languages

Eine Vielzahl an Renderengines stellen eine Programmierschnittstelle für die Beschreibung von Shadingfunktionen zur Verfügung. Diese sollen die Möglichkeit bieten, das Verhalten von Oberflächen über eine Programmiersprache zu beschreiben; ein derartiges Vorgehen ermöglicht es, eine größtmögliche Individualisierbarkeit beim Aussehen von Materialien zu erreichen. Weitere Vorteile bietet es außerdem bei der Erstellung von prozeduralen Texturen und Mustern; eine code-basierte Beschreibung von prozeduralen Materialien bietet ebenfalls im Regelfall größere Flexibilität als die Erstellung via node-basierter visueller Programmierung.

Weite Verbreitung weisen Shading Languages bei Echtzeitrendering auf, da sie bei derartigen zeitkritischen Anwendungen eine Effizienzsteigerung erlauben. Aber auch Productionrenderer, bei welchen der Qualität der einzelnen Frames größere Bedeutung beigemessen wird als der Quantität, besitzen eine Programmierschnittstelle.

Das wohl bekannteste Beispiel hierfür ist RSL - die RenderMan Shading Language - von Pixars RenderMan. RSL weist eine C ähnlichen Syntax auf (Pixar RenderMan, 2018).

Aber auch Houdini unterstützt mit VEX (Vector Expressions) eine Möglichkeit Shader code-basiert zu beschreiben. Auch VEX verwendet hierzu eine Syntax, welche ähnlich zu jenem der Programmiersprache C ist. Der Funktionsumfang von VEX beschränkt sich allerdings nicht nur auf Shading; es können beispielsweise auch unter anderem Primitiveattribute via VEX manipuliert werden. VEX im shading-spezifischen Kontext ist sehr stark an die RenderMan Shading Language angelehnt (SideFX, 2018).

Die mittlerweile am weitesten verbreitete Shading Language stellt die Open Shading Language, kurz OSL, dar.

3.1.1 Open Shading Language/OSL

Die Open Shading Language, kurz OSL, ist eine von Sony Pictures Imageworks entwickelte Shading Language. Anfänglich wurde OSL als eine proprietäre Programmiersprache von Sony für ihre in-house Renderlösung entwickelt, wurde aber wie viele Projekte von Sony Pictures Imageworks später unter einer Open Source Lizenz öffentlich zugänglich gemacht (Gritz, 2017, p. 1).

Neben OSL betreibt Sony Pictures Imageworks eine Vielzahl an anderen Projekten unter einer Open Source Lizenz, welche eine starke Adaptierung und Implementierung in zahllose andere Applikationen finden. Zu diesen Projekten zählen unter anderem das 3D-Fileformat Alambic, das Colormanagementsystem OpenColorIO(OCIO) und das Linux Commandline-Toolset für Python PYP (Sony Pictures Imageworks, 2018).

Open Shading Language wurde ursprünglich von Larry Gritz entwickelt und soll eine moderne Möglichkeit bieten, programmierbare Shader speziell in fotorealistischen Renderer mit neuesten Raytracing- und Global Illumination-Verfahren zu erstellen. Es bietet eine ideale Grundlage zur Beschreibung von Materialien, Lichtern, Displacement und zur Generierung prozeduraler Muster. Auch wurde bei der Erstellung von OSL die Anmerkung von anderen Animations- und VFX-Studios berücksichtigt (Gritz, 2017, p. 1).

Durch die Einbeziehung anderer Studios und des zur Verfügung stellens unter einer Open Source Lizenz wurde die Open Shader Language in eine große Menge anderer Renderer implementiert. Darunter Blenders Cycles, Chaos Groups V-Ray, Pixars RenderMan, Isotropixs Clarisse, Animal Logics Glimpse Renderer und Autodesks beziehungsweise SolidAngles Arnold (Gritz, 2018).

Die Open Shading Language besitzt schon wie RenderMans Shading Language und Houdinis VEX Shading Language zuvor über eine zur Programmiersprache C

ähnliche Syntax. OSL weist allerdings in seiner Funktionsweise signifikante Unterschiede zu anderen Shading Programmiersprachen auf (Gritz, 2017, p. 1).

Gritz schreibt hierzu:

> *OSL's surface and volume shaders compute an explicit symbolic description, called a "closure", of the way a surface or volume scatters light, in units of radiance. These radiance closures may be evaluated in particular directions, sampled to find important directions, or saved for later evaluation and re-evaluation. This new approach is ideal for a physically-based renderer that supports ray tracing and global illumination. In contrast, other shading languages usually compute just a surface color as visible from a particular direction. These old shaders are "black boxes" that a renderer can do little with but execute to for this once piece of information (for example, there is no effective way to discover from them which directions are important to sample). Furthermore, the physical units of lights and surfaces are often underspecified, making it very difficult to ensure that shaders are behaving in a physically correct manner. (Gritz, 2017, p. 1)*

Weiters unterscheidet sich OSL darin, dass es über keinerlei eigene Shader für Lichter verfügt. Lichter verwenden einen normalen Surfaceshader mit lichtemittierenden Qualitäten; folglich muss jedes Licht ein Flächenlicht sein. Außerdem unterscheidet sich OSL in der Handhabung von Transparenzen und der Opazität von Oberflächen sowie in der Auszeichnung von Renderpasses (AOVs) mit Hilfe von Light Pass Expresions. Letzteres ermöglicht es, dass man die Zusammensetzung der Komponenten der Beleuchtung (Specular, Diffuse, Reflektion, Refraktion, usw.) für die einzelnen Renderpasses über eine regular-expression-basierte Beschreibung auszeichnet (Gritz, 2017, p. 2).

Eine besondere Wichtigkeit werden OSL-basierte Shadern bei der Implementierung von Blenders Cycles in The Foundrys Nuke zeigen, da in diesem Zusammenhang OSL den einfachen Austausch von Shadern zwischen verschiedenen Applikationen ermöglicht.

3.2 Universal Scene Description/USD

Universal Scene Description, kurz USD, ist ein Projekt, welches von Pixar verwaltet wird. Es versucht das Problem zu lösen, dass jede Applikation innerhalb einer Animations- oder VFX-Pipeline Unmengen an internen Daten generiert, speichert und überträgt, welche zwischen den einzelnen Applikationen der Pipeline nicht kompatibel sind. Diese Daten werden Scene Description genannt. USD stellt die Bestrebungen Pixars dar, die internen Datenstrukturen der einzelnen Applikationen zu standardisieren und miteinander kompatibel zu gestalten. Die Universal Scene Description ist der Kern der gesamten Pipeline Pixars und bietet ein gemeinsames Fundament der einzelnen 3D-Anwendungen. So werden bereits jetzt Plugins für verschiedenste Applikationen bereitgestellt; darunter Autodesks Maya, SideFXs Houdini und The Foundrys Katana (Pixar, 2018).

USD könnte zukünftig im Zentrum einer noch größeren Anzahl an Applikationen, vor allem auch Renderern stehen, wodurch eine Implementierung jener in andere Applikationen weit vereinfacht werden würde. So wird die Universal Scene Description auch als eines der möglichen zukünftige native Cycles-Fileformat gehandelt. USD stellt im Gegenzug allerdings ein enormes Dependency dar, was das Handling einer Open-Source-Applikation wie Cycles deutlich erschwert (Blender Developers, 2017).

3.3 Alembic

> *Alembic is an open computer graphics interchange framework. Alembic distills complex, animated scenes into a non-procedural, application-independent set of baked geometric results. This 'distillation' of scenes into baked geometry is exactly analogous to the distillation of lighting and rendering scenes into rendered image data. (Alembic, 2018)*

Alembic ist ein Format zur Speicherung von dreidimensionalen Szenen. Es wurde, wie auch schon die Open Shading Language, von Sony Pictures Imgeworks, in diesem Fall allerdings in Zusammenarbeit mit Industrial Light and Magic beziehungsweise Lucasfilm Ltd. entwickelt. Es ist quelloffen und unter einer Open Source Lizenz verfügbar. An seiner nicht-prozeduralen Art zeigt sich eine Fokussierung auf Effizienz und Unabhängigkeit im Hinblick auf die verwendeten Applikationen. Ziel von Alembic ist es folglich das Endresultat von komplexen prozeduralen geometrischen Operationen zu speichern, nicht allerdings den

Abhängigkeitsgraphen an Operatoren, der zur Erstellung des Endresultates notwendig gewesen ist. Alembic-Files ermöglichen beispielsweise eine effiziente Speicherung von animierten Vertex-Positionen und Transformationen. Es handelt sich folglich um ein Cache-Format. Durch seine universelle und offene Natur ist das Alembic-Format ein Teil nahezu aller Programme aus dem VFX- und Animationsbereich und eignet sich dadurch bestens für den Austausch zwischen Applikationen und Departments. Alembic ermöglicht es beispielsweise bequem und effizient animierte Szenen an Lighting- und Rendering-Applikationen oder animierte Charakter aus Autodesk Maya für weitere Charakter-Effekte wie Kleidungsimulation in Houdini zu übergeben. Alembic wird von fast allen VFX- und Animationsprodukten der Firmen Autodesk, The Foundry, SideFX, Blender Foundation, Chaos Group und vielen mehr unterstützt (Alembic, 2018).

Alembic eignet sich durch seine Effizienz und seine breite Unterstützung durch eine Vielzahl an Applikationen gut zur Gestaltung einer umfangreichen Pipeline und auch als Cache-Format von geometrischen Objekten für den Austausch zwischen Compositing-Applikation und Renderengine.

3.4 OpenVDB

OpenVDB ist eine Datenstruktur zur Speicherung und Manipulation von volumetrischen Objekten. Es wurde von DreamWorks Animation entwickelt und 2012 unter einer Open Source Lizenz veröffentlicht. Es wird weiterhin von DreamWorks Animation weiterentwickelt und unterhalten. OpenVDB konnte sich in den vergangenen Jahren zu einem Standard in der Industrie etablieren; zahlreiche 3D-Applikationen und Renderengines unterstützen OpenVDB mittlerweile oder haben es sogar als ihre standardmäßige Datenstruktur adaptiert (OpenVDB, 2018).

OpenVDB beschreibt eine Datenstruktur, welche es ermöglicht, dass volumetrische Objekte nicht mithilfe eines räumlich gleichmäßigen, dreidimensionalen Rasters dargestellt werden, sondern durch eine hierarchische Struktur, welche Sparse Volumes implementiert. Sparse Volumes sind Volumen, welche eine ungleichmäßige Verteilung der Voxeldichte (volumetrische Pixel) aufweisen. OpenVDB ist im weitesten Sinne eine C++-basierte Library, welche eine effiziente Speicherung, effizient sowohl was Arbeitsspeicher als auch Platz auf der Festplatte betrifft, von Volumen, als auch die schnelle und ressourcenschonende Manipulation dieser ermöglicht (Museth, 2013, p. 1).

OpenVDB bietet hierbei eine Vielzahl an Vorteilen gegenüber herkömmlichen Datenstrukturen zur Speicherung von Volumen. So verfügt OpenVDB etwa nicht über eine Bounding Box, sondern eine Domäne innerhalb welcher das Volumen existieren kann, beziehungsweise ist diese Domäne unendlich groß; man spricht von “unbounded”. Weiters weist OpenVDB wie bereits erwähnt einen vergleichsweise kleinen Memory- und Disk-Fußabdruck auf; speziell bei großen volumetrischen Objekten steigt der Speicherbedarf nicht wie bei herkömmlichen Methoden proportional (Museth, 2013, p. 3).

Museth beschreibt in seinem Paper von 2013, dass ein großes volumetrisches Objekt mit herkömmlichen Datenstrukturen eine Größe von bis zu einem Viertel Terabyte erreichen kann, während der Disk-Fußabdruck des gleichen Objektes unter der Verwendung von OpenVDB weniger als ein Gigabyte beträgt (Museth, 2013, p. 6).

Weiters verfügt OpenVDB über ein Toolset, welches unter Zuhilfenahme von effizienten hierarchischen Algorithmen, eine äußerst schnelle Manipulation der Volumen zulässt; dies sowohl in einer zufälligen, als auch sequentiellen Abfolge. Weiters besitzt OpenVDB eine eigene, native BVH-basierte Beschleunigungsstruktur (Museth, 2013, p. 3).

Eine Vielzahl an Applikationen haben mittlerweile OpenVDB als Bestandteil ihres Funktionsumfanges adaptiert, an vorderster Front SideFXs Houdini. SideFX hat eine der umfassendsten Implementierungen von OpenVDB umgesetzt und hat damit sogar seine proprietäre Datenstruktur für Volumen ersetzt. SideFX gilt neben DreamWorks Animation als ein Beitragender für die Entwicklung des OpenVDB-Toolsets. Gemeinsam konnten sie etwa Sculpting Tools für Sparse Volumes im Funktionsumfang von OpenVDB ergänzen. Neben Houdinis Mantra unterstützen unter anderem Pixars RenderMan, Chaos Groups V-Ray, Isotropix Clarisse, Autodesks/Solid Angles Arnold und OTOYs Octane OpenVDB-Rendering. Blender verwendet OpenVDB als Cache-Format für seine Rauch- und Feuersimulationen (OpenVDB, 2018).

OpenVDB stellt einen wichtigen Standard für die Verwendung eines Raytracers innerhalb einer Compositing-Applikation dar, da es die Portierung von volumetrischen Objekten von 3D-Applikationen in die Compositing-Anwendung erlaubt.

3.5 Image-Passes

Ein weiterer wichtiger Punkt anhand dessen man die Qualität des Funktionsumfangs einer Renderengine messen kann, stellt nicht nur die Handhabung während der Bedienung, sondern auch die Möglichkeiten, wie man die Ergebnisse des Renderers weiterverarbeiten kann, dar. Als zentraler Punkt sind hierbei die gegeben Optionen der AOVs (arbitrary output variables) - auch Renderpassen oder Render Elements genannt - zu berücksichtigen. AOVs stellen in den meisten Fällen eine Notwendigkeit dar, da man die Möglichkeit haben möchte, das finale Rendering, welches die Renderengine produziert, noch verfeinern, beziehungsweise verändern zu können, ohne es erneut rendern zu müssen (Okun, Zwerman, 2015, p. 976).

Die am häufigsten verwendeten AOVs betreffen die einzelnen Komponenten der Beleuchtung; unter diese fallen etwa der direkte und indirekte Diffusanteil, die Reflektionen, die Refraktionen, die Glanzpunkte, die Volumenstreuung, die Lichtemission, atmosphärische Effekte und weitere. Dadurch, dass Renderengines diese Komponenten im Regelfall alle einzeln berechnen und das finale Bild lediglich die Summe jener ist, ist es ein Einfaches, die einzelnen Komponenten des Renderers abzuspeichern (Brinkmann, 2008, p. 448).

Da die einzelnen Komponenten wiederum lediglich die Summe der Ergebnisse der einzelnen Lichter sind, erlauben es manche Renderengines auch, die direkten Lichtanteile der einzelnen Lichter auf eigene Passes aufzuteilen. Eine dieser Renderer ist beispielsweise Chaos Groups V-Ray, bei welcher diese AOVs als Light Select Render Elements ausgezeichnet werden (Chaos Group, 2018a).

Die Verwendung von AOVs erlaubt einen weit flexibleren Umgang mit dem Rendering als es ein normales Rendering mit kombinierten Beleuchtungskomponenten zulässt. So besteht etwa die Möglichkeit, Materialeigenschaften im Nachhinein zu verändern, die andernfalls nur mit einem neuen Rendering möglich wären. Man kann beispielsweise die Stärke der Spiegelung auf einer Oberfläche reduzieren, den Diffusanteil farbkorrigieren ohne die Farbe der Reflektionen zu verändern oder auch im Fall von Light Select Render Elements die Farbe und Intensität einzelner Lichter verändern (Brinkmann, 2008, p. 450).

Bei Licht Select Render Elements gilt es allerdings zu beachten, dass Änderungen lediglich den direkten Lichtanteil verändern und andere Beleuchtungskomponenten wie beispielsweise Reflektionen nicht angepasst werden (Chaos Group, 2018).

Die einzelnen AOVs können als einzelne Files abgespeichert werden, beziehungsweise erlaubt das OpenEXR-Fileformat das Speichern von einer beliebigen Anzahl an Kanälen in einer einzigen Bilddatei (Brinkmann, 2008, p. 455).

Besonders im Hinblick auf die Tatsache der Implementierung einer Renderengine in eine Compositing-Applikation sind AOVs, deren Manipulation in den meisten Fällen eine Aufgabe im Compositing ist, besonders wichtig.

3.6 Data-Passes

Renderpasses sind nicht nur zur Zusammensetzung der einzelnen Komponenten der Beleuchtung des Renderings gut, sondern auch um zusätzlich Informationen der 3D-Szene an die Compositing-Applikation zu übergeben. Einer jener Passes, welcher mit am häufigsten Verwendung findet, ist der z-Depth-Pass. Der z-Depth-Pass gibt die Entfernung der im Bild dargestellten Objekte zur Kamera mit Hilfe einer Schwarz-Weiß-Abbildung wieder; ein einzelner Kanal wird benötigt. Die Entfernung der Objekte in der Szene zur Kamera wird hierbei über die Helligkeit dargestellt. Objekte, welche näher bei der Kamera sind, werden hellere, jene, welche sich weiter weg befinden, dunklere Werte zugewiesen. Diese zusätzliche Information ermöglicht es, verschiedenste Effekte im Anschluss an das Rendering im Compositing zu simulieren; dazu zählen unter anderem der Effekt der Tiefenunschärfe und diverse atmosphärische Effekte. Weiters erlaubt der z-Depth-Pass das Kombinieren mehrerer Renderings anhand deren Tiefeninformation. Man spricht von Z-Depth-Compositing. Das Kombinieren mehrerer Renderings basierend auf ihrer Tiefeninformation neigt allerdings dazu Bildartifakte zu produzieren, da wichtige Punkte wie halbtransparente Materialien und anti-aliasedte Kanten nicht berücksichtigt werden - Deep-Compositing ermöglicht auch derartiges, mehr dazu im folgenden Kapitel (Brinkmann, 2008, p. 437).

Weitere Data-Passes wären der Normal-Pass - gibt die Normale der gerenderten Oberfläche an -, der Position-Pass - gibt die XYZ-Koordinate der gerenderten Oberfläche an -, der Motion-Vector- oder Velocity-Pass - enthält Informationen zur Bewegung der gerenderten Oberfläche -, der Objekt- oder Material-ID-Pass - enthält Informationen zur Zusammengehörigkeit der gerenderten Oberfläche nach Objekt oder Shader - und viele weitere.

Algorithmen erlauben es weiters, dass man unter der Zuhilfenahme des Normal- und Position-Pass die Lichtsetzung in einem Rendering bis zu einem gewissen Grad verändert; man spricht von Relighting (The Foundry, 2018a).

Zusätzlich unterstützen mittlerweile eine Vielzahl an Renderengines den von Psyop entwickelten Standard zur Berücksichtigung von Bewegungsunschärfe, Transparenzen, Tiefenunschärfe in Object-ID- beziehungsweise Material-ID-Passes. Dieser Standard verwendet Passes, welche Cryptomatte genannt werden. Es inkludiert weiters Objekt- und Materialnamen, welche direkt in der Compositing-Applikation ausgelesen werden können. Encoders für Cryptomatte stehen mittlerweile unter anderem in Isotropix Clarisse, Chaos Groups V-Ray, SideFX Mantra, Blenders Cycles, Pixars RenderMan, Redshift und Autodesks/SolidAngles Arnold zur Verfügung - Decoders in The Foundrys Nuke, Blackmagic Designs Fusion, Adobes After Effects, Autodesks Flame, Blenders Compositor und Houdinis Compositors (Psyop, 2018).

Weiters besteht natürlich oft die Möglichkeit sich benutzergenerierte Passes zu erstellen, welche zusätzliche Informationen wie Maskierung und ähnliches enthalten können (Okun, Zwerman, 2015, p. 977).

3.7 Deep-Compositing

Deep-Compositing stellt eine Erweiterung des z-Depth-Compositing dar. Es erlaubt verschiedene Renderings anhand ihrer Tiefeninformation zu kombinieren, ohne dass die Notwendigkeit besteht Holdouts zu verwenden (Okun, Zwerman, 2015, p. 687).

Die Verwendung von Deep-Daten beinhaltet ein verändertes Konzept, auf welche Art man Tiefeninformation in einem Bild speichern kann. Anstelle der üblichen Methode im Compositing viele einzelne Ebenen mit Hilfe von Holdout-Mattes zu kombinieren, speichert man beim Deep-Rendering statt einem einzelnen RGBA-Wert und einem dazugehörigem Tiefenwert pro Pixel mehrere ab und kombiniert Ebenen unter der Zuhilfenahme jener. Dadurch wird die Rücksichtnahme auf halbtransparente Oberflächen, Bewegungsunschärfe, Tiefenunschärfen, geglättete Kanten sowie Volumen ermöglicht. Das ursprüngliche Konzept geht auf Pixar zurück; es wurde von Weta Digital, Peregrin Labs und The Foundry weiterentwickelt und implementiert (Seymour, 2014).

Es gilt zu beachten, dass Deep-Compositing eine große Menge an Rechen- und Speicherressourcen benötigt. Beim Compositing steigt sowohl die Prozessor- als auch Speicherauslastung, ebenso bei Rendering, was sich durch längere Renderzeiten bemerkbar macht (Okun, Zwerman, 2015, p. 988).

Die Speicherung von Deep-Information ist innerhalb von OpenEXR-Files möglich. OpenEXR unterstützt seit seiner Version 2.0 diesen Standard; das Fileformat

wurde ursprünglich von Industrial Light and Magic entwickelt, die Deep-Implementierung basiert allerdings auf ODZ, Weta Digitals Deep-Data-Fortmat. Mittlerweile wird Deep-Rendering von den meisten geläufigen Renderern unterstützt. Es ist von vielen Studios, an vorderster Front Weta Digital, ein integraler Bestandteil derer Pipeline (Seymour, 2014).

3.8 Color Pipeline

Viele Applikationen im VFX- und Animationsbereich setzen auf die Verwendung von OpenColorIO (OCIO) für das Farbmanagement. OpenColorIO ist einmal mehr ein Open Source Projekt, welches von Sony Pictures Imageworks ins Leben gerufen wurde. Es ist eine komplette Farbmanagement-Lösung, welche applikationsübergreifend eine einheitliche Methode zur Anwendung von Farbräumen bietet. Dabei bezieht es selbst seine notwendigen Informationen für die Farbraumtransformationen aus dem Konfigurationsfile, welches OCIO zur Verfügung gestellt wird. Die einzelnen Farbraumtransformationen, welche im config.ocio-File vermerkt werden, entsprechen dabei den unterschiedlichen Farbräumen (OpenColorIO, 2018).

Standardmäßig werden Renderings von dreidimensionalen Szenen in einem linearen Farbraum ausgegeben. Dies entspricht der physikalisch korrekten Wiedergabe von Lichtinformation, nicht aber zwingend der, wie das menschliche Auge, beziehungsweise die meisten Bildsensoren von Kameras, die Helligkeitsinformation wahrnehmen. Betrachtet man beispielsweise das gefilmte Material, welches professionelle digitale Filmkameras produzieren, so zeigt sich eine nicht-lineare Farbencodierung, um den aufgezeichneten Dynamikumfang zu erhöhen und eine effizientere Speicherung dessen zu erlauben. Man spricht von logarithmischen Farbräumen. Die Idee dahinter ist, dass man dunkle Bereiche in ihrer Helligkeit mit Hilfe einer logarithmischen Kurve anhebt und helle Bereiche im aufgezeichneten Bild mit einer umgekehrten Kurve absenkt. Dadurch ergibt sich ein kontrastarmes - man spricht von einem flachen - Bild. Die Wiedergabe derartiger Bilder erfolgt anschließend wieder in einer linearen Art, unter Berücksichtigung der Gammakorrektur von Displays; der logarithmische Farbraum bezieht sich im Regelfall nur auf die Speicherung (Brinkmann, 2008, p. 419).

Bei Renderings dreidimensionaler Szenen ist eine derartige Speicherung grundsätzlich nicht notwendig, da sie den unglaublich großen Dynamikumfang von digitalen Renderings innerhalb von HDR(High Dynamic Range)-Fileformaten speichern. Als Standard hierfür hat sich das OpenEXR-Format etabliert. Nichtsdestotrotz stellt die rein lineare Darstellung von Bildern eine sehr

unnatürliche Darstellung dar, da das menschliche Auge und auch die meisten Kameras besonders Highlights verzerrt wahrnehmen. So wie bei vielen anderen Sinneseindrücken verhält sich die empfundene Stärke der Helligkeit proportional zum Logarithmus der tatsächlichen Stärke des physikalischen Reizes mit abnehmender Sättigung in den Highlights. Sogut wie alle Applikationen aus dem VFX- und Animationsbereich betrachten die Helligkeit auf eine normalisierte Art, sprich die Helligkeitswerte der einzelnen Pixel werden mit Gleitkommazahlen zwischen 0 und 1 ausgezeichnet. Der Wert 0 entspricht hierbei Schwarz, der Wert 1 reinem Weiß. 0 und 1 können hierbei auch unter- beziehungsweise überschritten werden. Verfügt ein Format über die Möglichkeit auch Werte zu speichern, welche nicht zwischen 0 und 1 liegen, so spricht man von HDR-Fileformaten. Eine andere Art der Darstellung wäre die Werte als ganze Zahlen zu betrachten, wobei beispielsweise für ein 8-Bit-Bild gilt, dass 0 Schwarz entspricht und 255 Weiß (Brinkmann, 2008, p. 419).

In dieser Arbeit kommt lediglich die normalisierte Darstellung der Pixelwerte durch Gleitkommazahlen zur Anwendung.

Da eine rein lineare Darstellung von Renderings zwischen 0 und 1 einen sehr eingeschränkten Dynamikumfang von nur wenigen Blendenstufen bei der Betrachtung durch ein normales Display besitzt, haben viele Renderengines die Möglichkeit, das von ihnen gerenderte Bild auch nicht-linear auszugeben. So besitzt beispielsweise Chaos Groups V-Ray die Funktion, das Rendering mit einem als “Exponential” bezeichnetem Color Mapping auszugeben. Hierbei werden sehr helle Stellen des Bildes entsättigt und die Helligkeitswerte zwischen 0 und unendlich auf Werte zwischen 0 und 1 logarithmisch neu verteilt (Chaos Group, 2018b).

Einen anderen Ansatz verfolgt Blenders Renderengine Cycles, welche eine OpenColorIO Farbraumtransformation durchführt um ein ähnliches Resultat wie V-Rays “Exponential” zu erzielen. Blenders Ansatz, welcher als “Filmic Blender” bezeichnet wird, verfügt über verschiedenste Vorteile. So wird ein besonders weicher Abfall der hellen Stellen innerhalb des Renderings durch den fortgeschritteneren Algorithmus ermöglicht. Auch erlaubt die Tatsache, dass “Filmic Blender” auf OCIO-basiert die Möglichkeit, dass man die Farbraumtransformation auf andere Applikationen überträgt. “Filmic Blender” wurde ursprünglich von Troy Sobotka als Ergänzung zu Blenders Farbmanagement entwickelt und ist mittlerweile standardmäßig in Blender implementiert (Sobotka, 2018).

Ein Problem, welches die Anwendung von Color Mapping auf Renderings durch Renderengines betrifft, ist die Tatsache, dass es lediglich auf den Beauty-Pass

angewendet werden kann. Dies führt dazu, dass es keinerlei Anwendungsmöglichkeiten bei Multipass-Renderings aufweist. Dies kann umgangen werden, indem man das Color Mapping erst im Compositing nach der Rekonstruktion des Beauty-Passes durch die einzelnen Image-Passes anwendet. Tests meinerseits mit Renderings haben gezeigt, dass die “Softclip”-Node von The Foundrys Nuke mit einer Konvertierung durch eine logarithmische Komprimierung, wobei der “softclip min” Wert 0 entsprechen muss, optische idente Resultate wie V-Rays “Exponential” liefert. Eine Implementierung von “Filmic Blender” über OpenColorIO wird in einem späteren Kapitel beschrieben.

Die Abbildung 1 zeigt den Unterschied einer rein linearen Darstellung (links) und einer Darstellung nach der Anwendung einer logarithmischen Kurve (rechts). In diesem Fall handelt es sich um ein Rendering mit Pixars RenderMan innerhalb von Blender. Das Color Mapping des rechten Bildes wurde erst nachträglich in Nuke mit Hilfe der “Softclip”-Node realisiert. Durch die logarithmische Komprimierung der RGB-Kanäle wird der stark überbelichtet Grün-Kanal von Pixelwerten von teils über 8 auf unter 1 reduziert. Dadurch ergibt sich der weit realistischer wirkende Eindruck beim Abfall des Lichtes, sowie der Sättigung bei sehr hellen Stellen des Renderings.

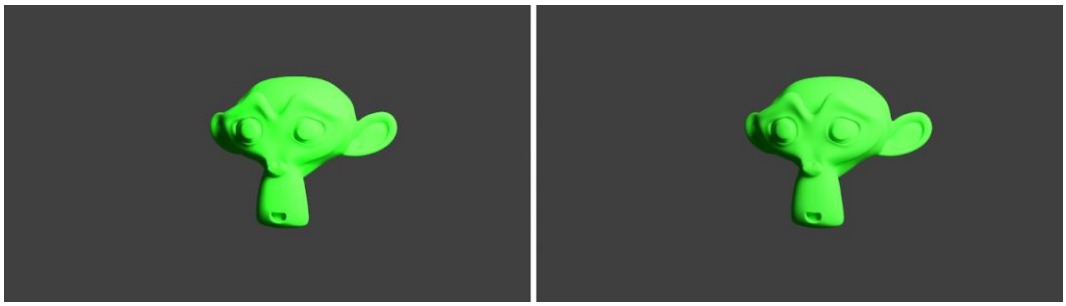

Abbildung 1. Zeigt die logarithmische Komprimierung eines Renderings durch die “Softclip”-Node in Nuke.

Eine weitere Alternative zum standardmäßigen linearen Farbraum stellt ACES dar. ACES (Academy Color Encoding System) ist allerdings mehr als ein schlichter Farbraum für Renderings; es stellt eine standardisierte Farbumgebung für alle Arten von Bildmaterial verschiedensten Ursprungs dar. ACES soll so eine Color Pipeline bieten, die sowohl passend für Renderings von dreidimensionalen Szenen, gefilmtes Videomaterial als auch Fotografien ist. Den Ursprung bildet bei ACES der jeweils native Farbraum der jeweiligen Quelle, welcher durch einen Input Device Transform (IDT) in den ACES-Farbraum transformiert wird. Im ACES-Farbraum finden alle gewünschten weiteren Schritte, wie Compositing und Color

Grading statt. Für die Ausgabe durch ein Display wird auf das File ein Reference Rendering Transform (RRT) sowie ein Output Device Transform (ODT) angewendet. Der ACES-Farbraum stellt dabei einen Ultra Wide Gamut, High Dynamic Range, RGB Farbraum dar, welcher selbst linear ist. ACES soll durch seine Output Transformation in den gewünschten Farbraum, beispielsweise Rec 709 oder P3, auf allen Wiedergabegeräten gleich aussehen. ACES besitzt trotz seiner linearen Natur über die Maßen gute Eigenschaften in Hinblick auf überbelichtete Bereiche innerhalb von Bildern, speziell auch bei Renderings von dreidimensionalen Szenen, welche beim Fotorealismus unterstützen (Fry, 2015).

Ein Problem bei ACES ist allerdings, dass nicht alle Filter-Kernels auf Bilder im ACES-Farbraum angewendet werden können; Filter-Kernels wie “lanczos3” beispielsweise führen bei ACES zu einer schwarzen Umrandung von Highlights (VES Technology Committee, 2012, p. 35).

Das Academy Color Encoding System ist weitestgehend ein Ersatz für die OpenColorIO Pipeline; OpenColorIO unterstützt allerdings die Verwendung von ACES in den jeweiligen Applikationen (OpenColorIO, 2018).

4 Implementierung

Im folgenden Kapitel soll die Implementierung einer Renderengine, welche auf dem Raytracing-Renderingverfahren basiert, in eine Compositing-Applikation beschrieben werden. Compositing beschreibt den Task innerhalb des VFX- und Animationsbereiches, bei welchem einzelne Bildelemente zusammengefügt werden, um das finale Bild zu erstellen. Bei den einzelnen Bildelementen kann es sich um Material verschiedensten Ursprungs handeln; so kann es sich etwa um gefilmtes Material, zweidimensionale Matte Paintings oder Renderings von dreidimensionalen Szenen handeln. In Arbeitsumgebungen großer Firmen sind die Erstellungen der einzelnen Elemente, wie beispielsweise Matte Paintings und Renderings, in verschiedene Departements aufgeteilt, welche auf die jeweiligen Task spezialisiert sind und dafür auch spezialisierte Software verwenden. In kleineren Firmen oder auch vor allem Einzelunternehmen ist dies allerdings oft nicht möglich, da derartige Spezialisierungen mit einem großen Aufwand von Know-How sowie Lizenzkosten verbunden sind. Für derartige Umgebungen ist es oft ratsam, weniger spezialisierte Software zu verwenden beziehungsweise die Software um gewisse Funktionsumfänge zu erweitern, um einen effizienten Workflow zu gestalten. Eine derartige Verschiebung in Richtung Effizienz ist stets mit Einbußen in der Flexibilität verbunden, da spezialisierte Software für einzelne Aufgaben einen weit größeren Funktionsumfang bieten kann. Im Folgenden wird nun also versucht, den Funktionsumfang einer Compositing-Applikation zur Steigerung der Effizienz der VFX-Pipeline zu erweitern. Die Compositing-Applikation soll hierbei Funktionen einer 3D-Applikation übernehmen und fotorealistisches Rendering ermöglichen.

Es gibt verschiedene Compositing-Applikationen, welche auf den Einsatz innerhalb von professionellen Umgebungen ausgelegt sind. The Foundrys Nuke zählt hierbei zu jenen mit den vielfältigsten Aufgabenbereichen, da Resultate höchster Qualität, selbst für die aufwendigsten Aufgaben, erzielt werden können. Dies hat Nuke zu einer der am weitesten verbreiteten Compositing-Applikation im High-End Bereich aufsteigen lassen. Nahezu alle großen und viele der kleinen Anbieter für visuelle Effekte und Animationen setzen auf Nuke. Die Implementierung soll sich deshalb auch auf The Foundrys Nuke beziehen.

Obwohl Compositing in einer traditionellen Sichtweise eine Aufgabe ist, welche sich auf die Kombination von zweidimensionalen Bildelementen bezieht, ist für viele Aufgaben ein 3D-Raum in der Applikation notwendig, um fortgeschrittenere

Compositing-Techniken anzuwenden. So verfügen fast alle Compositing-Applikationen, einschließlich Nuke, einen 3D-Raum.

The Foundrys Nuke verfügt insgesamt über drei verschiedene native Arten seine dreidimensionalen Szenen zu rendern. Diese Arten beinhalten einerseits den “3D Viewer”, welcher die Viewport-Anzeige von Nuke darstellt, der “ScanlineRender”-Renderer, bei welchem es sich um den Standard Produktionrenderer von Nuke handelt, und den “RayRender”-Renderer. Der “RayRender”-Renderer stellt dabei Nukes neueste Renderengine dar und ist die einzige der nativ Implementierten, welche Raytracing-basierend ist. Der Funktionsumfang des “RayRender”-Renderers ist allerdings stark limitiert und lässt kein fotorealistisches Rendering zu. Er verfügt nicht über die Funktionen weiche Schattenkanten zu berechnen, weitere Reflektionen innerhalb von Reflektionen darzustellen, Refraktionen zu berechnen, indirekte Beleuchtung via Global Illumination einzusetzen, Displacement zu generieren und viele mehr. Der “RayRender”-Renderer findet kaum Einsatz; der “ScanlineRender”-Renderer ist jener Renderer, dessen Einsatz in Produktionen zu bevorzugen ist. Der “RayRender”-Renderer ist lediglich dazu gedacht, Reflektionen zu berechnen, um gegebenenfalls einen eigenen Reflection-Pass nachträglich für ein Rendering einer dreidimensionalen Szene in einer Compositing-Applikation zu generieren (The Foundry, 2018b).

Da keine der nativ implementierten Renderer von Nuke fotorealistisches Rendering unterstützt, haben Drittanbieter Plug-ins entwickelt, welche den Funktionsumfang von Nuke um diesen Punkt erweitern sollen. Aufgrund hoher Lizenzkosten sowohl für die Plug-in-Lizenz als auch die Renderlizenzen und den jährlichen Erhaltungskosten ist dies allerdings in vielen Fällen für kleine Unternehmen oder gar Einzelpersonen nicht von Interesse.

Der Raytracing-Renderer, dessen Implementierung im folgenden Kapitel beschrieben ist, soll grundlegende Funktionalitäten bieten und einen ersten Prototyp darstellen.

4.1 Gründe für die Implementierung

Die Gründe für eine Implementierung einer Raytracing-basierten Renderengine mit vollem Funktionsumfang lassen sich aus den in der Literaturrecherche zu den gängigen Technologien beim Rendering angeführten Unterscheidungen ableiten. Auch wenn The Foundrys Nuke über einen Raytracing-basierten Renderer verfügt, beziehen sich die Vergleiche auf den Production-Renderer von Nuke, den

“ScanlineRender”-Renderer, da der “RayRender”-Renderer überwiegend lediglich die Ergänzung des Funktionsumfanges zur Berechnung von Reflektionen darstellt.

Aufgrund der allgemeinen Natur von Raytracing, die dreidimensionale Szene mit Hilfe von Strahlen zu betrachten, ergibt sich eine physikalisch weit korrektere Methode zur Berechnung der Darstellung der Szene. Ereignisse wie Schattenwurf, Reflektionen, Transparenzen und Brechungen, indirekte Beleuchtung und weiteres sind ohne großen Mehraufwand auf physikalisch korrekte Weise zu berechnen. Alle diese Punkte, welche für fotorealistisches Rendering unabdingbar sind, stellen bei Rendering via Rasterization große Probleme dar und sind in Nukes “ScanlineRender”-Renderer nicht möglich. Die “ScanlineRender”-Node lässt einen lediglich Verdeckungen, perspektivische Verzerrungen und ähnliches berechnen. Für das Shading sind nur ein Diffuse-Shader (ähnlich zu Lambert-Shader), ein Phong-Shader, ein Emission-Shader und ein paar wenige weitere vorhanden. Große Wichtigkeit besitzt der “Project3D”-Shader; jener ermöglicht die Projektion einer Textur auf eine dreidimensionale Geometrie aus dem Blickfeld einer gewählten Kamera. Dadurch wird es ermöglicht, zweidimensionale Bildinhalte auf dreidimensionale Objekte zu übertragen, wodurch sich etwa bei Retuschen perspektivische Verzerrungen, parallaktische Verschiebungen und ähnliche Effekte generieren lassen. Nuke kommt mit verschiedene Arten an Lichtquellen, welche dem oben genannten Shadern entsprechend Auswirkung auf die dreidimensionalen Objekte aufweisen (The Foundry, 2018b).

Versucht man nun fotorealistische Bilder anhand von 3D-Szenen zu berechnen ist eine Raytracing-basierte Renderengine notwendig.

4.2 Ansprüche an die Renderengine

Um die oben genannte Möglichkeit des Fotorealismuses zu erreichen und einen Mehrwert für die Arbeit in Compositing-Applikationen zu bieten, benötigt die zu implementierende Renderengine einen gewissen Funktionsumfang. Diesen gilt es in diesem Kapitel zu beschreiben.

Der Funktionsumfang der Renderengine, die es zu implementieren gilt, sollte sich mit vielen Punkten decken, welche in den Kapiteln über Rendering-Technologien und über eine fortgeschrittene Render-Pipeline beschrieben sind. Folglich lassen sich aus der Literaturrecherche Notwendigkeiten für den Funktionsumfang der Renderengine ableiten.

So sollte an erster Stelle die Renderengine über eine API, eine Programmierschnittstelle, verfügen, welche eine offene Kommunikation mit ihr

zulässt und eine einfache Anbindung an die Compositing-Applikation erlaubt. Sollte eine derartige Schnittstelle nicht vorhanden sein, so schränkt dies die Möglichkeit, wie die Renderengine implementiert werden kann, deutlich ein.

Um eine Implementierung via API durchführen zu können, ist es vonnöten, dass der Renderer über eine Art Importer für eine Szenenbeschreibung verfügt. Hierfür gibt es eine Vielzahl an Varianten; etwa über ein RIB-, XML- oder Jasonfileformat. Die modernste und eleganteste Methode wäre allerdings die Verwendung der im Kapitel der fortgeschrittenen Render-Pipeline beschriebenen Verwendung der Universial Scene Description, USD.

Der Austausch von Informationen die Geometrie betreffend, kann entweder in der Szenenbeschreibung selbst passieren oder über ein externes Format abgehandelt werden. Als externes Format würde sich das speziell auf Caching spezialisierte, im vorigen Kapitel beschriebene Alembic-Format empfehlen. Es ist schonend im Hinblick auf Ressourcen und erfährt durch seinen offenen Quellcode eine große Kompatibilität.

Da speziell Volumendaten äußerst viel Speicherbedarf aufweisen können, empfiehlt es sich bei jenen besonders, diese unter der Zuhilfenahme von einem externen Datenformat an die Renderengine zu übergeben. Hier macht besonders OpenVDB auf sich aufmerksam, da es wie schon Alembic große Kompatibilität zwischen den einzelnen Softwarepaketen aufweist und über einen niedrigen Memory- und Disk-Fußabdruck besitzt.

Weiters ist die aufgezeigte Beschreibung von Shadern via einer Shading Language besonders brauchbar. Es lässt zu, dass Shader abseits der Szenenbeschreibung definiert und übergeben werden. So kann die Open Shading Language verwendet werden, um einen Shader innerhalb einer 3D-Applikation zu definieren, in der Compositing-Applikation zu verwenden und zu adaptieren und anschließend an die Renderengine zu übergeben.

Da es sich um die Implementierung in eine Compositing-Applikation handelt, sollte der Renderer über alle wichtigen Funktionen für fortgeschrittene Compositing-Techniken verfügen. Dies inkludiert an vorderster Stelle die Möglichkeit die einzelnen Renderelemente auszugeben; sowohl Image-Passes als auch Data-Passes. Mit Hilfe der Image- und Data-Passes kann man wie im vorigen Kapitel beschrieben, die Renderings nach dem Prozess ihrer Berechnung weiter verfeinern und adjustieren.

Die Verwendung von Deep-Renderings ist lediglich innerhalb von Compositing-Applikationen von Interesse und sollte deshalb auch bei der Implementierung einer Renderengine in eine solche durch den Renderer zur Verfügung gestellt werden.

Was den Prozess des Rendering selbst betrifft, sollte der Renderer über alle Funktionen umfassen, welche im Rendering-Technologien-Kapitel unter Raytracing beschrieben wurden; von fortgeschrittenen Reflektionsmodellen - welche komplexe BRDFs verwenden - über Distibution Ray Tracing - wodurch weiche Schattenkanten, antialiasde Bilder und Tiefen- und Bewegungsunschärfe ermöglicht wird - bis hin zu Transparenzen und Refraktionen sowie Globale Illumination durch Pathtracing.

Eine logarithmische Komprimierung der Helligkeitswerte wie zuvor beschrieben sollte grundsätzlich immer in der Compositing-Applikation passieren. Aufgrund der Bestrebungen reproduzierbare Ergebnisse berechnen zu können, ist es allerdings von Interesse, den gleichen Algorithmus zur Komprimierung sowohl in der Compositing-Applikation als auch im Rendering selbst anwenden zu können.

4.3 Open Source Renderengines

Es existiert eine große Anzahl an Projekten, welche zum Ziel haben, eine Renderengine zu erstellen und unter einer Open-Source-Lizenz zu vertreiben. Viele der Renderer erfüllen die grundlegendsten Kriterien und sind Raytracing-basierend; sie erlauben fotorealistisches Rendering.

An erster Stelle ist Cycles zu nennen. Cycles ist eine Open-Source-Renderengine; es handelt sich um den mit der 3D-Applikation Blender der Blender Foundation ausgelieferten Renderer. Ursprünglich wurde die Entwicklung durch die Blender Foundation und Brecht van Lommel gestartet, umschließt aber mittlerweile eine Vielzahl an freiwilligen Entwicklern (Blender Foundation, 2018a).

Cycles verfügt über eine äußerst solide Implementierung in Blender, besitzt allerdings auch eine Standalone-Applikation, welche über eine XML-API eingebunden werden kann. Leider erlaubt die XML-API nur die Verwendung eines begrenzten Funktionsumfanges von Cycles und auch ist die Standalone-Applikation noch Änderungen unterworfen. Nichtsdestotrotz erlaubt es dennoch die Nutzung vieler Funktionen, welche nicht selbstverständlich sind. So umfasst Blenders Cycles in der Version der Standalone-Applikation die Verwendung von in der Open Shading Language beschriebenen Shadern (Blender Foundation, 2018b).

Ein weiterer Punkt, welcher für die Verwendung von Blenders Cycles spricht, ist die Tatsache, dass es als eines der einzigen Open-Source-Renderer bei großen Produktionen getestet und verwendet wurde, auch um sehr umfangreiche Szenen zu rendern. Als Beispiel ist die Verwendung durch die Firma Barnstorm VFX bei der von Amazon produzierten Serie “The Man in the High Castle” zu nennen. Hierbei wurden zahlreiche fotorealistische Renderings zur Kombination mit real-gedrehten Inhalten gefertigt (Siddi, 2017).

Ein weiteres Beispiel zeigt auch die Verwendung von Cycles für animierte Spielfilme. Eine für September 2018 angekündigte Produktion von Netflix durch das Animationsstudio Tangent Animations setzt ausschließlich auf die Verwendung von Blender und Cycles (Veldhuizen, 2018a).

Ein großer Teil an wünschenswerten Funktionen befindet sich bei Blender Cycles noch in der Entstehung. Das schließt ein: Microdisplacement, Light Groups (um einzelne Lichter nur speziellen Objekten zuzuordnen), OpenVDB Rendering und viele mehr (Blender Foundation, 2018b).

Ein weiterer vielversprechender Raytrace-Renderer mit außerordentlichem Funktionsumfang ist mit appleseed gegeben. Der Renderer ist vom Funktionsumfang ähnlich zu Blender Cycles und unterstützt ebenfalls die von Sony Pictures Imageworks entwickelte Open Shading Language. Weiters weist er sowohl eine umfassende Integration in Autodesks Maya und 3ds Max, in Blender sowie Image Engines Gaffer auf. Weiters verfügt er über eine C++, als auch Python API. Der appleseed-Renderer befindet sich noch in seiner Beta-Phase und hat kaum Verwendung in Produktionen (appleseed, 2018).

Weitere Open-Source-Renderengines, welche Raytracing zur Anwendung bringen, gibt es unter den Namen Yafaray, Mitsuba und RenderPixie. Vielversprechend erscheint außerdem noch LuxCoreRender. Seine Vorgängerversion war zuvor unter dem Namen LuxRender verfügbar, wurde aber in einer überarbeiteten Fassung als LuxCoreRender neu gestaltet. Dabei umfasst die Neuveröffentlichung eine auf C++ und Python API fokussierte Version. Bei LuxCoreRender handelt es sich um einen auf physikalische Korrektheit getrimmten Renderer, welcher den Versuch darstellt, eine Artist-freundliche Version des von Matt Pharr und Greg Humphreyes in ihrem Buch “Physically Based Rendering - From Theory to Implementation” beschriebenen Renderer zu erstellen (LuxCoreRender, 2018).

Aufgrund einer Veröffentlichung der Software im Frühjahr 2018 konnte LuxRenderCore nicht für die Implementierung in eine Compositing-Applikation berücksichtigt werden.

Obwohl die Standalone-Applikation von Blenders Cycles die wohl rudimentärste API zur Verfügung stellt, ist ihre Verwendung in professionellen Produktionen ein wichtiger Punkt, da es die Qualität, Flexibilität und Stabilität der Renderengine selbst aufzeigt. Weiters sind die in Arbeit befindlichen Funktionen sehr vielversprechend und auch soll der Funktionsumfang der Standalone-Applikation, einschließlich der XML-API, verbessert werden. Außerdem ist die Implementierung via Programmierschnittstelle nicht die einzige Möglichkeit eine Renderengine in eine weitere Applikation zu integrieren. Eine native Implementierung des Quellcodes in die API der Compositing-Applikation ist ebenfalls möglich. Die Unterschiede sowie Vor- und Nachteile werden im folgenden Kapitel erörtert.

Da Blenders Cycles der vielversprechenste Renderer in Hinblick auf Qualität und Stabilität darstellt, wird im Folgenden eine Implementierung jenes Renderers versucht und beschrieben.

4.4 Arten der Implementierung

Wie bereits erwähnt, kann eine Implementierungen nicht nur über eine API, eine Programmierschnittstelle, auf Seiten der Renderengine betrieben werden, sondern auch eine direkte Implementierung des Quellcodes des Renderers in eine API, welche durch die Compositing-Applikation zur Verfügung gestellt wird. Die Vor- und Nachteile werden im folgenden Abschnitt diskutiert.

The Foundrys Nuke besitzt sowohl über eine C++-API als auch eine Python-API. Die C++-API wird als NDK, Nuke Developer Kit, bezeichnet, die Python-API auch als Python scripting engine. Beim NDK handelt es sich hierbei um die low level Schnittstelle; jene kann verwendet werden um jegliche Operatoren, zum Beispiel zur Bearbeitung von Bildern, Deep-Daten oder Geometrie, in den Funktionsumfang von Nuke zu implementieren. Code, welcher für das NDK geschrieben wurde, verfügt über eine hohe Schnelligkeit bei der Ausführung und verfügt weiters über die Möglichkeit auf die gesamte Szenenbeschreibung innerhalb von Nuke zuzugreifen. Die Python-API stellt jene Schnittstelle dar, welche sich vorrangig auf das Interface und die high level Node Manipulation bezieht. Die Python-API ist besonders für das schnelle Erstellen von Prototypen sowie für die Automatisierung und Manipulation der Benutzeroberfläche geeignet. Sie bietet kaum Zugriff auf vertiefende Informationen der Szenenbeschreibung sowie gar keine Möglichkeiten zur Manipulation von Bildern (The Foundry, 2018c).

Die beiden Schnittstellen von The Foundrys Nuke lassen verschiedene Möglichkeiten offen, wie Blenders Cycles in die Applikation implementiert werden kann. Dies umfasst unter anderem die Möglichkeit, den Quellcode der Renderengine direkt in Operatoren innerhalb von Nuke zu implementieren.

Operatoren beschreiben hierbei jene Engines, welche für die Bearbeitung von Bild- und 3D-Daten benötigt werden. Operatoren werden innerhalb von Nuke als Nodes repräsentiert, sind allerdings nicht gleich zu setzen. Nodes und Operatoren sind voneinander entkoppelt; Nodes können einen oder mehrere Operatoren ansteuern, welche die Bild- und 3D-Manipulation im Anschluss erledigen. Bei Operatoren handelt es sich um shared libraries, welche dynamisch von Nuke bei Bedarf geladen werden (The Foundry, 2017).

Eine Implementierung von Cycles auf diese Art und Weise würde viele Vorteile bieten. So würde das Rendering der 3D-Szenen direkt in Nuke stattfinden, wodurch die Möglichkeit geschaffen werden würde, eine interaktive Rendervorschau (IPR - Interactive Photorealistic Rendering) im Nuke Viewer anzuzeigen. Auch wäre das Rendering besonders schnell, da keine weitere API der Renderengine angesprochen werden muss. Weiters könnte der Quellcode der Renderengine direkt auf alle im NDK zur Verfügung gestellten Informationen der Szenenbeschreibung zugreifen und verwenden. Dies beinhaltet Informationen zu den einzelnen Punkten der Geometrie, den Surfacenormals, den UV-Koordinaten, den Bewegungsvektoren der einzelnen Punkte zur Berechnung der Bewegungsunschärfe und den zugewiesenen Shadern.

Während diese Methode zur Implementierung aufgrund ihrer oben genannten Vorteilteile auf jeden Fall zu präferieren wäre, hat sie dennoch einige Nachteilen, die sich in einem enormen Aufwand zur Implementierung und Erhaltung wiederspiegeln. Einerseits ist es notwendig, die einzelnen Codeteile von Cycles als Operatoren in Nuke zu implementieren. Dies beinhaltet auch viele Teile, welche die Standalone-Applikation von Cycles automatisiert vor dem Schritt des Renderings durchführt; etwa das Kompilieren von OSL-Shadern oder die Weiterverarbeitung von Geometriedaten. Auch ist dieses Plug-in für jedes Betriebssystem und jede neue Veröffentlichung einer Hauptversion von Nuke durch The Foundry neu zu kompilieren. Änderungen im Cycles-Quellcode, wie sie etwa bei neuen Versionen der Fall sind, müssen direkt in das Plug-in eingebunden werden. Hierbei kann es sich jeweils um sehr arbeitsintensive Schritte handeln. Bei einer derartigen Implementierung ist Python nur auf der Ebene der Benutzeroberfläche notwendig.

Die zweite Methode zur Implementierung des Renderers Cycles basiert ebenfalls auf der Verwendung des NDK, zeitgleich aber auch ausführlicher auf der Python-

API von Nuke. Hierfür wird weiters Cycles Standalone-Applikation benötigt, welche über die XML-API angesteuert wird. Das NDK wird bei dieser Art der Implementierung lediglich dazu verwendet, die Szenenbeschreibung von Nuke in einem für Cycles passenden XML-Format zu exportieren. Es handelt sich um einen mehr oder weniger simplen Exporter, in welchem die gleichen Informationen wie sie bei einer direkten Implementierung des Quellcodes in Nuke von Interesse sind ausgegeben werden; dies betrifft die einzelnen Punkte der Geometrie, die Surfacenormals, die UV-Koordinaten und die Bewegungsvektoren der einzelnen Punkte zur Berechnung der Bewegungsunschärfe. Auch müssen weiters die zugewiesenen Shader sowie Kamera und Licht Information mit ausgegeben werden. Der Aufruf des Renderingbefehles wird im Anschluss unter der Verwendung von Python passieren.

Diese Implementierung ist weit einfacher als jene der direkten Implementierung des Source Codes des Renderers, da die bestehenden Programmstrukturen der Standalone-Applikation verwendet werden können. Auch muss bei einer Änderung des Cycles-Quellcodes nicht zwingend das gesamte Plug-in adaptiert werden, um die neuen Funktionen nutzen zu können; dies ist abhängig von der Art der Änderung.

Die Nachteile einer derartigen Implementierung lassen sich allerdings auch deutlich aufzeigen; der Funktionsumfang, welcher in die Compositing-Applikation implementiert werden kann, ist in einer direkten Abhängigkeit von jenem Funktionsumfang, welcher die XML-API von Cycles bietet. Bei Cycles sind die Einschränkungen der XML-API leider deutlich merkbar.

Die dritte Implemetierungsart, welche hier Erwähnung findet, stellt eine Implementierung dar, welche sich rein zur Entwicklung eines Prototyps eignet. Bei jener Methode findet ausschließlich die Python-API Anwendung sowohl für das Exportieren der Szenenbeschreibung als auch für den Aufruf des Renderingsbefehls. Diese Art der Implementierung ist ohne Zweifel jene, welche den meisten Einschränkungen unterliegt, gleichzeitig aber auch jene, welche in kürzester Zeit eine große Anzahl von Iterationen an Änderungen über das Plug-in zulässt. Dadurch eignet sich diese Methode besonders für das schnelle Prototyping; ein Plug-in, welches in einer Produktion zum Einsatz kommen kann, lässt sich auf diese Art und Weise nicht erstellen.

Auch hier sind die Einschränkungen klar zu definieren. So unterliegt diese Methodik klar den Einschränkungen der XML-API von Cycles, als auch den Einschränkungen der Pyhton-API von Nuke selbst. Nuke lässt es nur zu, Teile der Szenenbeschreibung auszulesen; man kann lediglich auf einen Teil der Informationen, welche die Geometrie betreffen, zugreifen. So lässt sich die

Geometrie selbst und die Normalen der Oberfläche exportieren. Informationen über die UV-Koordinaten der Objekte in der Szene sind genauso wie Informationen zu deren Bewegungsvektoren nicht einsehbar. Zugewiesene Shader lassen sich ausschließlich über einer auf Node-Ebene laufenden und zu programmierenden Abfragestruktur feststellen.

Durch die fehlenden UV-Koordinaten lassen sich keinerlei Texturen auf das Rendering anwenden. Die XML-API von Cycles lässt keine Verwendung externer Cacheformate wie Alembic zu beziehungsweise unterstützt diese Cycles Standalone-Applikation nicht; nur die Verwendung von Geometrie-Information via XML-Fileformat.

Die reine Verwendung von Python macht das Plug-in andererseits auch relativ einfach zu portieren für andere Betriebssysteme. Neue Hauptversionen von Nuke und auch Cycles haben vergleichsweise geringen Einfluss auf die Lauffähigkeit des Plug-ins.

Durch die Möglichkeit schneller Anpassung der Eigenschaften des Plug-ins durchzuführen, lassen sich Probleme wie etwa Unterschiede in den Transformationsmatrizen bei Nuke und Cycles einfach ausfindig machen und anpassen. Es ist erstrebenswert, im Anschluss an die Entwicklung des Prototyps, die gesammelte Information von dessen Entwicklung in den Kontext des NDKs zu übertragen, um die weiteren Funktionen freizuschalten.

Betrachtet man professionelle Implementierungen von Renderern von Drittanbietern in Applikationen, so zeigt sich, dass die wenigsten eine native Integration direkt über den Quellcode wählen. Meist wird der Prozess über eine durch den Renderer zur Verfügung gestellte API abgewickelt. Besonders deutlich ist die Integration bei der Betrachtung des von Pixar entwickelten Plug-ins von RenderMan für Blender, da es sich bei diesem Plug-in um ein Plug-in unter der Open-Source-Lizenz handelt. Der Quellcode ist über das dazugehörige Git Repository abrufbar. Betrachtet man dieses, zeigt sich, dass das Plug-in für Blender ausschließlich in Python verfasst wurde und als ein Exporter fungiert. Die exportierten Daten werden an den lokal-installierten RenderMan-Server übergeben, welcher das Rendering durchführt. Jener stellt hierbei die Standalone-Applikation dar. Das gerenderte Bild wird anschließend via dem RenderMan eigenen Framebuffer IT wiedergegeben, kann aber durchaus auch wieder an Blender zurück übergeben werden beziehungsweise als Bilddatei abgespeichert werden. RenderMan verwendet für den Austausch der Szenenbeschreibung RIB-Files. Es steht außer Frage, dass die Python-API Blenders besser für derartige Aufgaben geeignet ist, als jene von Nuke (prman-pixar, 2018).

Da diese Arbeit die Erstellung eines Prototyps zum Ziel hat, welcher die Grundfunktionalitäten der Renderengine Cycles unterstützt, wird sich im Folgenden auf jene Methode spezialisiert, welche reinen Python-Code verwendet. Um die jeweiligen Einschränkungen zu umgehen, habe ich auch mit den anderen Methoden experimentiert; auch dies wird Einfluss nehmen, wenn auch ungleich weniger.

4.5 Blenders Cycles

Wie bereits erwähnt ist die Wahl des zu implementierenden Renderers auf Blenders Cycles-Renderer gefallen. Jener ist besonders durch seine hohe Rendering-Qualität als auch durch die Verwendung in größeren Produktionen aufgefallen. Weiters ist das große Team an Entwicklern stets darum bemüht den Funktionsumfang des Renderers zu erweitern.

Cycles stellt dabei den raytracing-basierten Production-Renderer von Blender dar, welcher unidirektionales Pathtracing, die Verwendung von CPU und GPU (auch Multi-GPU), das Rendern von Meshes, Hair-Curves, Volumen und Instanzen sowie die Verwendung von Subdivision Surfaces erlaubt. Weiters lässt es die Verwendung von OSL, physically-based Rendering via den Principled BRDF, Global Illumination sowie das Rendering von Displacement zu (Blender Foundation, 2018c).

4.5.1 Building Cycles Standalone-Applikation

Bei Cycles handelt es sich wie bereits erwähnt um einen Renderer, dessen Quellcode offen ist und beliebig verwendet werden kann. Der Quellcode der Standalone-Applikation verfügt über ein eigenes Repository, von welchem es heruntergeladen werden kann. Blenders Cycles ist nicht als binary-Version verfügbar und muss selbst für die jeweilige Plattform kompiliert werden. Es besitzt über ein Makefile, benötigt aber hierfür einige Dependencies, Abhängigkeiten von weiteren zu kompilierenden Bibliotheken.

Die Liste an Dependencies, welche zum Builden von Cycles notwendig sind, umfasst:

- OpenGL
- GLEW
- Boost

- OpenImageIO
- PugiXML
- OSL (optional)

(Blender Foundation, 2017)

Viele dieser Dependencies stehen wiederum in einer Abhängigkeit zu anderen Bibliotheken. Beschrieben wird im folgenden Kapitel das Kompilieren eines Clones des Masterbranches (Version 1.9.0) von Ende 2017, unter der Linux-Distribution CentOS - CentOS 7, Kernel-Version: Linux 3.10.0-693.11.6.el7.x86_64. Bei der Installation dieses Betriebssystems wurden alle vorhandenen Development Tools, Development Libraries und Compatibiliy Libraries mitinstalliert.

Welche Dependency-Versionen zum Builden für Cycles notwendig sind, war für mich nicht immer klar erkennbar und hat sehr viel Zeit in Anspruch genommen. Eine klare Ausweisung der jeweiligen benötigten Versionen war vonseiten der Blender Foundation nicht immer gegeben; in den meisten Fällen hat die aktuellste Version allerdings problemlos funktioniert.

Beim ersten Dependency handelt es sich um OpenGL. OpenGL stellt eine Programmierschnittstelle dar, welche zur Darstellung von zwei- und dreidimensionaler Grafikanwendung ausgelegt ist. Sie wurde 1992 in den Markt eingeführt und wird seitdem von einer Vielzahl an Programmen verwendet. Durch die große Verbreitung dieses Dependencies war in meinem Fall eine Installation nicht notwendig. Auch hat sich die vorinstallierte Version als kompatibel herausgestellt. Eine Installation von OpenGL und dessen Version lässt sich über den Linux-Terminal verifizieren. Hierfür ist die Verwendung des Command-Line Tools glxinfo von großer Hilfe. Es bietet Informationen über Hardwarebeschleunigung, verwendete Grafiktreiber und viele weitere. Unter der Verwendung von glxinfo | grep OpenGL kann man alle zur Verfügung gestellten OpenGL Einträge, einschließlich der Versionsnummer auslesen. Die Version “4.5.0” - “4.6” wäre zu diesem Zeitpunkt (Anfang 2018) die aktuellste Version gewesen - unter der Verwendung des NVIDIA-Treibers “384.111” hat sich für mich als kompatibel herausgestellt. Keine weiteren Schritte waren vonnöten.

Das nächste Dependency stellt GLEW dar. Es handelt sich um eine plattformübergreifende C/C++-Erweiterung, welche effiziente Möglichkeiten bietet, zur Laufzeit festzustellen, welche OpenGL-Erweiterungen auf der Zielplattform unterstützt werden. Die GLEW-Version “2.1.0” stellt die aktuellste Version dar, sie wurde am 31. Juli 2017 veröffentlicht (GLEW, 2017).

Die Version “2.1.0” von GLEW zeigte sich mit Cycles kompatibel. Ein Makefile zum Kompilieren der Library wird zur Verfügung gestellt. Die inkludierten Befehle “make” und “make install” (als Administrator) waren ausreichend; keine weiteren Schritte waren notwendig, der C-Compiler der GNU Compiler Collection wurde verwendet. Jener ist in den vorinstallierten Development Tools von CentOS inkludiert.

Bei Boost handelt es sich um eine Library, welche eine Vielzahl an Unterbibliotheken zur Verfügung stellt. Jene können für verschiedenste Anwendungen eingesetzt werden. Dazu zählen mathematische Anwendungen, Speicherverwaltung und auch Bildverarbeitung. Das Dependency war bereits in der Version “1.53” auf dem CentOS-System vorinstalliert. Zum Zeitpunkt des Kompilierens war die Version “1.66” mit einer Veröffentlichung im Dezember 2017 die aktuellste (Boost, 2018).

Auch bei Boost hat sich die vorinstallierte Version als mit Cycles kompatibel gezeigt; keine weiteren Schritte sind notwendig.

Mit OpenImageIO wird ein weit größeres Dependency benötigt, als jene, welche bisher eine Abhängigkeit darstellten. OpenImageIO ist eine Bibliothek zum Lesen und Schreiben von Bilddateien und eine Ansammlung von Klassen und Programmen. OpenImageIO unterstützt dabei eine große Vielzahl an Dateiformaten; TIFF, JPEG, OpenEXR, PNG, DPX - um nur ein paar zu nennen. Durch seine breite Unterstützung von Dateiformaten steht OpenImageIO selbst in einer Abhängigkeit von anderen Dependencies. Um jene zu definieren muss zuerst eine Version von OpenImageIO festgelegt werden; die aktuellste Version hat sich als nicht kompatibel mit Cycles gezeigt. Cycles und Blender selbst basieren noch auf C++-Code im C++03-Standard, die gegenwärtigen Versionen von OpenImageIO auf dem C++11- und C++14-Standard. Eine passende Version für Cycles zu finden, hat sich als nicht einfach herausgestellt und viel Zeit in Anspruch genommen. Nach langer Recherche und vielen getesteten Versionen hat sich der “Release-1.7.15” als für Cycles passend erwiesen. Bei jener Version handelt es sich um eine der letzten Versionen von OpenImageIO, welche sich unter der Verwendung von einer Flag beim Kompilieren mit CMake als für den C++03-Standard kompatibel erwiesen.

OpenImageIO selbst benötigt zum Kompilieren GCC, Boost, CMake und OpenEXR. Der C-Compiler der GNU-Compiler-Collection (GCC) ist als Development Tool in der Version “4.8.5” vorinstalliert; die Version ist den Angaben von OpenColorIO entsprechend.

Für Boost wird mindestens die Version “1.53” verlangt, welche exakt jener von CentOS vorinstallierten Version entspricht. Von CMake ist eine Version aktueller als “3.2.2” als Abhängigkeit angegeben. CMake stellt dabei eine plattformübergreifende Anwendung dar, welche den Prozess des Kompilierens von Software verwaltet. Für Linux existiert von CMake eine binäre Version und muss nicht kompiliert werden. Ich habe die damals aktuellste CMake-Version “3.10.1” verwendet.

Weiters wird die OpenEXR-Bibliothek als Dependency benötigt; Version “2.2” wird empfohlen, “2.2.1” kam meinerseits zum Einsatz. Um OpenEXR builden zu können muss man wiederum die IlmBase-Bibliothek sowie die zlib-Bibliothek kompilieren. Es werden für die beiden Dependencies von OpenEXR keinerlei Angaben zu der Versionsnummer gemacht; die aktuellsten Versionen zeigten allerdings eine Kompatibilität. Bei zlib handelt es sich um eine Library zur Komprimierung von Daten.

IlmBase wird dabei direkt auch von OpenEXR zur Verfügung gestellt und die Versionsnummer deckt sich mit jener von OpenEXR. Folglich kam IlmBase in der Version “2.2.1” zum Einsatz. Die aktuellste Version von zlib mit der Nummer “1.2.11” fand außerdem Verwendung. Beide Dependencies stellen ein Makefile mit entsprechenden Befehlen, welche dem Makefile selbst zu entnehmen sind, für das Kompilieren (make) und die Installation (make install) zur Verfügung.

Ist nun OpenEXR bereit, kann OpenImageIO in einen binären Maschinencode umgewandelt werden. Wie bereits erwähnt ist hierfür eine Flag notwendig, um die Kompatibilität der Bibliothek mit dem C++-03-Standard zu gewährleisten. Der Befehl “make” ist definiert, zusätzlich ist die Flag “USE_CPP11=0” notwendig.

OpenImageIO verfügt über keinen Installations-Befehl und daher müssen die kompilierten Library- und Header-Files zur weiteren Verwendung mit Cycles in die dazugehörigen globalen Verzeichnisse verschoben werden. Unter Linux gibt es vier Stück davon: /usr/lib, /usr/local/lib, /usr/include und /usr/local/include.

PugiXML ist das letzte obligate Dependecy von Cycles. Es handelt sich um eine Bibliothek, welche Tools zur Verarbeitung von XML-Datein zur Verfügung stellt. Ein Clone des Masterbranches vom Anfang des Jahres 2018 wurde verwendet. Auch PugiXML verfügt lediglich über einen “make”-Befehl. Auch hier muss die Installation händisch erfolgen.

Das Dependency der Open Shading Language ist optional. Der Funktionsumfang ist bei einem nicht Vorhandenseins der Bibliothek in Cycles nicht verfügbar. Blender selbst hat zum Zeitpunkt des Kompilierens die Version “1.7.15” verwendet

und ist dadurch auch wohl für Cycles passend. OSL selbst benötigt Boost, Imath, und OpenImageIO als Dependecies. Weiters benötigt es LLVW und Clang.

Die Umwandlung der OSL-Bibliothek in Maschinencode hat sich als über die Maßen schwierig erwiesen und war schlussendlich für mich nicht möglich. Gescheitert ist das Kompilieren von OSL an LLVM. Da das Dependency nicht obligat für Cycles ist, habe ich dessen Existenz bis auf weiteres nicht berücksichtigt.

Stehen alle Dependencies zur Verfügung, kann Cycles über einen “make”-Befehl kompiliert werden.

Um einen besseren Überblick zu erhalten folgt eine weitere Aufschlüsselung:

- OpenGL - 4.5.0
- GLEW - 2.1.0
- Boost - 1.53
- OIIO - 1.7.15
    a. Boost - 1.53
    b. CMake - 3.10.1
    c. IlmBase 2.2.1
    d. zlib - 1.2.11
    e. OpenEXR - 2.2.1
- PugiXML
- OSL - 1.7.5

Im Anschluss kann die Lauffähigkeit des Programmes via “./cycles” im Verzeichnis der ausführbaren Datei überprüft werden. Um dieses Verzeichnis global verfügbar zu machen, empfiehlt es sich, diesen Ordner den Systemvariablen zu übergeben. Dies kann mit dem Befehl “export PATH=$PATH:” und dem angehängten absoluten Pfad zum gewünschten Verzeichnis erzielt werden.

Das Kompilieren von Cycles war doch einigermaßen zeitintensiv, da es sich als schwierig gestaltete, für jedes Dependency die passende Version zu finden. Besonders das Erstellen einer Kompatibilität zu OpenImageIO hat sich als äußerst komplex erwiesen und einige Zeit in Anspruch genommen. Die Verwendung von Linux hat manche Schritte zweifelsfrei erleichtert, da viele Bibliotheken, welche stark gebräuchlich sind, bereits in der erweiterten Installation inkludiert sind. Andererseits gilt es auch festzuhalten, dass die Dokumentation von manchen Dependencies für andere Betriebssysteme ausführlicher ausfallen.

4.5.2 XML-API

Ist Cycles nun kompiliert, kann es unter der Zuhilfenahme der XML-API angesteuert werden. Dabei kann die XML-Datei direkt an Cycles via den Terminal übergeben werden. Die dafür möglichen Befehle werden in einem späteren Kapitel aufgelistet. Dieses Kapitel befasst sich zur Gänze mit der XML-API, welche durch Cycles zur Verfügung gestellt wird. Leider wird eine Dokumentation zur API von Cycles, beziehungsweise deren Entwicklern, nicht bereitgestellt. Alle vorgebrachten Darstellungen beziehen sich auf Ableitungen von Cycles beigefügten Beispielen. Jene Beispiele sind über das Repository von Blenders Cycles verfügbar; sie sind dem Source-Code angehängt. Weitere Informationen sind direkt dem C++-Quellcode der Standalone-Applikation entnommen.

XML, die Extensible Markup Language, stellt eine standardisierte Auszeichnungssprache dar. Sie erlaubt es, hierarchische, teils sehr komplexe Datenstrukturen in einer für Menschen lesbaren Form abzuspeichern. Eine XML-Datei besteht im Regelfall aus einer XML-Deklaration, welche am Beginn des Dokuments angeführt wird. Diese Deklaration enthält einerseits die XML-Version, meist “1.0”, und das Encoding. Wird das File mit “UTF-8” encoded, können auch Umlaute und andere Sonderzeichen zur Anwendung kommen. Neben der Deklaration kommen weiters Tags zum Einsatz. Tags stellen dabei ein Element innerhalb der XML-Datei dar. Ein Tag wird durch die Verwendung von Spitzenklammern (< und >) definiert. Die einzelnen Elemente besitzen dabei über einen öffnenden Tag, einen Namen, einen Körper und einen schließenden Tag: quasi <Tagname>Körper</Tagname>. Der schließende Tag wird mit einem Slash (/) beschrieben. Nebst diesem Tag existiert auch die Möglichkeit, körperlose Tags zu verwenden. Jene verfügen über keinen Körper und benötigen deshalb kein schließendes Tag, beziehungsweise ist das schließende Tag bereits im öffnenden Tag inkludiert; <Tagname /> (Ernesti, Kaiser, 2017, p. 577).

Weiters können Tags auch mit Attributen versehen werden. Jene Attribute bestehen aus Schlüsseln und Werten; getrennt werden sie durch ein Gleichheitszeichen, Werte müssen stets in Anführungsstriche gesetzt werden (Ernesti, Kaiser, 2017, p. 578).

Jedem Tag können weitere Tags untergeordnet werden (Ernesti, Kaiser, 2017, p. 577).

Eine einfache Szene kann deshalb wie folgt an Cycles übergeben werden:

```
1    <?xml version=”1.0” encoding=”UTF-8”?>
2    <cycles>
3    <camera width="1280" height="720" />
4    <transform translate="1 4 -6" scale="1 1 1" rotate="10 1 4 0">
5        <camera type="perspective" />
6    </transform>
7    <background>
8        <background name="bg" strength="1.0" color="0.2, 0.2, 0.2" />
9        <connect from="bg background" to="output surface" />
10    </background>
11    <include src="cube.xml" />
12    </cycles>
```

Anfänglich wird die XML-Deklaration durchgeführt (Zeile 1), gefolgt von einem alles weitere umschließenden “cycles”-Tag (Zeile 2 und 12). In Zeile 3 wird mittels “camera”-Tag eine Kamera ausgezeichnet sowie deren Renderauflösung definiert. Die Definition der Auflösung erfolgt hierbei mit Attributen, welche dem “camera”-Tag beigefügt werden. Es folgt ein “transform”-Tag (Zeile 4 und 6) in welchem die Transformationsdaten - Translation, Rotation und Skalierung - wieder mit Hilfe von Attributen festgelegt werden. Diesem Tag untersteht ein körperloser Tag, welcher die Kamera als perspektivische Kamera ausweist (Zeile 5). Mit Hilfe des “transform”-Tags ist es auch möglich die Transformation von Objekten zu bestimmen. Dem C++-Source-Code der Standalone-Applikation von Cycles sind die möglichen Attribute und Werte der Kamera zu entnehmen und umfassen unter anderem, aber nicht ausschließlich:

- shuttertime
- rolling_shutter_type
- rolling_shutter_duration
- shutter_curve
- aperturesize
- focaldistance
- blades
- bladesrotation
- matrix
- aperture_rotation
- type
- perspective
- orthograph
- panorama
- panorama_type
- equirectangular

- mirrorball
- fisheye_equidistant
- fisheye_equisolid
- fov
- stereo_eye
- none
- left
- right
- interocular_distance
- convergence_distance
- use_pole_merge
- pole_merge_angle_from
- pole_merge_angle_to
- sensorwidth
- sensorheight
- nearclip
- farclip

Der “background”-Tag in Zeile 7 und 10 umschließt einen weiteren “background”-Tag, welcher den Namen des Umgebungslichtes auf “bg”, die Helligkeit auf “1.0” und die Farbe auf “0.2, 0.2, 0.2” festlegt (Zeile 8). Weiters wird ein “connect”-Tag umschlossen, welcher, vergleichbar mit Node-basiertem Arbeiten, den “background”-Ausgang der “bg”-Node mit dem “surface”-Eingang der “output”-Node des Hintergrundes verknüpft. Ähnlich können auch die Ausgaben von BRDFs an weitere Nodes übergeben werden; dazu später mehr.

In Zeile 11 wird noch auf externe Geometrie zurückgegriffen; in Form eines “include”-Tags, dessen “src”-Attribute den relativen Pfad zum XML-Files der Geometrie angibt. Hierbei kann lediglich auf weitere XML-Daten verlinkt werden, nicht auf andere Dateitypen. Geometrie kann weiters auch direkt im File der Szenenbeschreibung ausgezeichnet werden; es muss nicht auf eine externe Datei referenziert werden.

Geometrie wird dabei folgendermaßen definiert:

```
<mesh P="1.000000 1.000000 -1.000000  1.000000 -1.000000 -1.000000  -
1.000000 -1.000000 -1.000000  -1.000000 1.000000 -1.000000  1.000000
0.999999 1.000000  0.999999 -1.000001 1.000000  -1.000000 -1.000000
1.000000  -1.000000 1.000000 1.000000  " nverts="4 4 4 4 4 4 "
verts="0 1 2 3  4 7 6 5  0 4 5 1  1 5 6 2  2 6 7 3  4 0 3 7  "/>
```

Bei dem oben dargestellten Objekt handelt es sich um einen Würfel. Die Tatsache, dass es sich um Geometrie handelt, wird mit den “mesh”-Tag ausgewiesen. Dabei werden die acht Punkte des Würfels innerhalb des Attributes “P” festgelegt. Jeweils drei Werte bilden einen Vektor, welche die X-, Y- und Z-Position der einzelnen Punkte definieren. Das Attribut “verts” beschreibt weiters, welche Punkte einen Anteil an den jeweiligen Polygonen haben. Das Attribut “nverts” erweitert diese Tatsache noch mit der Information, aus wie vielen Vertices die einzelnen Polygone bestehen. Die Normalen der Geometrie ergeben sich dabei aus der Anordnung des “verts”-Attributes.

Folglich lässt sich aus der dargebrachten Information ableiten, dass es sich um ein Objekt bestehend aus acht Vertices handelt, welche sechs Quads - viereckige Polygone - bilden.

Um Shader zu erstellen wird folgende Syntax vorgegeben:

```
1    <shader name="cube">
2        <diffuse_bsdf name="cube_diffuse" color="0.8, 0.8, 0.8"
roughness=”1.0” />
3        <connect from="cube_diffuse bsdf" to="output surface" />
4    </shader>
```

Shader lassen sich durch den “shader”-Tag beschreiben (Zeile 1 und 4). Dem Shader werden zwei weitere Elemente untergeordnet. Ersterer beschreibt, dass der Shader den Diffuse-BSDF enthält. Er verfügt über drei Attribute, welche den Namen des BRDF-Nodes, die Oberflächenfarbe des Objektes und die Oberflächenbeschaffenheit des Objektes definieren (Zeile 2). In Zeile 3 wird weiters definiert, dass der “bsdf”-Ausgang der BSDF-Node mit dem Namen “cube-diffuse” an den “surface”-Input der “output”-Node des Shader übergeben werden soll.

Es kann nicht nur der “diffuse_bsdf” Verwendung finden, sondern weiters auch ein “glossy_bsdf”, ein “emission”-Shader sowie ein “glass_bsdf”. Der “glass_bsdf” kann weiters noch das Attribut “distribution” mit beispielsweise einem Wert wie “beckmann”, “GGX” oder “Multiscatter GGX” sowie das Attribut des Brechungsindexes “ior” mit einem Wert wie beispielsweise “1.4” entgegennehmen. Auch der “glossy_bsdf” kann über ein “distribution”-Attribut verfügen. Zusätzlich zu den Werten des “glass_bsdf” steht hier noch das Modell von Ashikhmin und Shirley zur Verfügung; “ashikhmin-shirley”-Wert.

Dem C++-Quellcode der Standalone-Applikation ist weiter zu entnehmen, dass der “anisotropic_bsdf”-, der “toon-bsdf”-, der “velvet-bsdf”-, der “principled-bsdf”-, der “translucent-bsdf”-, der “transparent-bsdf”-, der “hair-bsdf” und der “subsurface_scattering”-Shader zur Verfügung stehen.

Zusammenfassend noch einmal die zur Verfügung stehenden Shader laut C++-Source-Code:

- diffuse_bsdf
- glossy_bsdf
- glass_bsdf
- refraction_bsdf
- principled_bsdf
- anisotropic_bsdf
- toon_bsdf
- velvet_bsdf
- translucent_bsdf
- transparent_bsdf
- hair_bsdf
- subsurface_scattering
- emission
- holdout
- ambient_occlusion
- absorption_volume
- scatter_volume
- background_shader
- osl_shader

Nun kann der Shader dem gewünschten Objekt zugewiesen werden:

```
1    <state interpolation="smooth" shader="cube">
2        <include src="cube.xml" />
3    </state>
```

Die Zuweisung erfolgt über den “state”-Tag in Zeile 1 und 3. Als untergeordnetes Element ist hierbei das Objekt definiert. Dem “state”-Tag können zwei Attribute

zugewiesen werden; einerseits das “shader”-Attribut, welches als Wert den Namen des Shaders übergibt sowie das “interpolation”-Attribut, welches festlegt, ob das Objekt ein “Smooth” oder ein “Flat” Shading erhalten soll.

Shader können auch umfangreicher ausfallen als oben dargestellt, wie im folgenden Beispiel zu sehen ist:

```
1    <shader name="sphere">
2        <wave_texture name="tex" scale="2.0" distortion="2.0" />
3
4        <diffuse_bsdf name="diffuse1" roughness="0.0" color="0.8,
0.0, 0.0" />
5        <diffuse_bsdf name="diffuse2" roughness="0.0" color="0.8,
0.8, 0.0" />
6        <mix_closure name="mix" />
8
9        <connect from="tex fac" to="mix fac" />
10        <connect from="diffuse1 bsdf" to="mix closure1" />
11        <connect from="diffuse2 bsdf" to="mix closure2" />
12
13        <connect from="mix closure" to="output surface" />

14        <connect from="tex fac" to="output displacement" />
15    </shader>
```

In diesem Beispiel ist zu sehen, wie in Zeile 2 eine prozedurale Textur erstellt wird. Zwei Diffuse-BSDF werden definiert (Zeile 4 und 5), die anhand der Textur verblendet werden sollen. Hierfür wird in der Zeile 6 eine “mix”-Node erstellt, welche die “bsdf”-Outputs der beiden “diffuse_bsdf”-Nodes anhand des “fac”-Attributes der Textur mischt (Zeile 9, 10, 11). Der Output der “mix”-Node wird in Zeile 13 ausgegeben. Weiters findet die Texture noch Verwendung im Displacement (Zeile 14).

Neben der “wave_texutre” gibt es weiters noch eine “checker-texture” und auch beispielsweise eine “sky_texture”. Laut C++-Source-Code umfassen die Texturing-Nodes weiters noch:

- image_texture
- environment_texture
- gradient_texture
- noise_texture
- voronoi_texture
- musgrave_texture

- magic_texture
- brick_texture
- point_density_texture

Im nächsten Schritt sind Lichter zu definieren. Neben einem punktförmigen Licht und dem Backgroundlight sind weiters noch ein Spotlight, ein Distancelight und ein Arealight möglich. Die Definition eines Punktlichtes ist wie folgt:

```
1    <shader name="point_shader">
2        <emission name="emission" color="0.8 0.1 0.1" strength="100"
/>
3        <connect from="emission emission" to="output surface" />
4    </shader>
5
6    <state shader="point_shader">
7        <light type="point" co="-2.0 1 0.0" size="0.01" />
8    </state>
```

Dabei zeigt sich, dass das Punktlicht aus einem “emission”-Shader besteht, der in Zeile 1 bis 4 definiert wird. In Zeile 6 bis 8 wird der Shader dem Punktlicht zugeordnet. Beim Punktlicht handelt es sich dabei nicht um ein Punktlicht nach klassischer Annahme, sondern um ein kugelförmiges Licht, welches durch den Radius “size” definiert ist. Das “co”-Attribut gibt die Position des Lichtes an.

Das Spotlight verfügt weiters noch über ein “dir”-Attribut für die Richtung, ein “spot_angle”-Attribut für die Größe des Lichtkegels und ein “spot_smooth”-Attribut, welche den Lichtabfall des Lichtkegels definiert.

Zusätzlich zu den Attributen des Pointlights verfügt das Arealight noch über ein “dir”-, ein “axisu”- und ein “axisv”-Attribut.

Weiters ist es über die XML-API auch möglich, das Verhalten des Integrators zu bestimmen. Der Integrator stellt dabei jenen Algorithmus dar, welcher verwendet wird um das Licht zu berechnen. Dabei handelt es sich im Falle von Cycles um einen Pathtracing-Integrator. Dieser Integrator verfügt über zwei Methoden zur Berechnung des Lichts, einerseits die “path tracing”-Methode, andererseits die “branched path tracing”-Methode. Die “path tracing”-Methode wirft jenen Strahl, welcher durch die Kamera ausgesendet wird, in exakt eine Richtung, zu exakt einem Licht weiter. Dies führt dazu, dass jeder individuelle Strahl schneller berechnet werden kann, jedoch oft mehr Strahlen benötigt werden, um ein

rauschfreies Bild zu erreichen. Die “branched path tracing”-Methode wiederum teilt den Strahl bei seinem ersten Auftreffen auf die Oberfläche in alle weiteren Komponenten des Oberflächenshaders auf. Dadurch ist es möglich, feinere Kontrolle über das Sampling zu erlangen. Weiters können bei dieser Methode die einzelnen Strahlen der Kamera alle Lichter berücksichtigen (Blender Foundation, 2018d).

Weiters gibt es noch zwei weitere Sampling-Methoden, welche gemeinsam oder auch getrennt voneinander eingesetzt werden können. Die “indirect light sampling” Methode erlaubt es, dass der Strahl dem jeweiligen Oberflächen BRDF folgt, während die “direct light sampling”-Methode eine Lichtquelle auswählt und den Strahl direkt in Richtung des Lichtes nach verfolgt (Blender Foundation, 2018d).

Der Integrator kann über einen “integrator”-Tag ausgezeichnet werden; die dazugehörigen Attribute lassen sich aus folgendem Beispiel ableiten:

```
<integrator sample_all_lights_direct="true"
sample_all_lights_indirect="true max_volume_bounce="0"
method="branched_path" "/>
```

Das “max_volume_bounce”-Attribut ist dabei ein weiteres Attribut, welches der Standalone-Applikation übergeben werden kann. Laut C++-Source-Code des Renderers ist es weiters möglich folgende Attribute im “integrator”-Tag zu übergeben:

- max_bounce
- max_glossy_bounce
- max_transmission_bounce
- transparent_max_bounce
- ao_bounces
- volume_max_steps
- volume_step_size
- caustics_reflective
- caustics_refractive
- filter_glossy
- seed
- sample_clamp_direct
- sample_clamp_indirect
- motion_blur
- aa_samples

- diffuse_samples
- glossy_samples
- transmission_samples
- ao_samples
- mesh_light_samples
- subsurface_samples
- volume_samples
- start_sample
- sample_all_lights_direct
- sample_all_lights_indirect
- light_sampling_threshold

Weiters beschreibt der C++-Source-Code noch weitere Utility-Nodes wie etwa “invert”.

OSL shader können wie folgt in der XML-API integriert werden:

```
1    <shader name="cube">
2        <osl_shader name="stripes_diffuse" src="./osl/stripes.osl" >
3            <output name="BSDF" type="closure color" />
4        </osl_shader>
5        <connect from="stripes_diffuse BSDF" to="output surface" />
6    </shader>
```

Eine normale Shader-Definition (Zeile 1 und 6) umschließt einen “osl-shader”-Tag, Zeile 2 und 4. Diesem untergeordnet ist ein “output”-Tag, welcher die Funktion erfüllt, den Ausgang der OSL-Node zu definieren. Der in Zeile 5 wird der BSDF der OSL-Node der “Surface” zugewiesen.

Ein Kompilieren des .osl-Files in eine .oso-bytcode-Datein ist nicht notwendig. Dies wird direkt von Cycles erledigt.

Aufgrund der fehlenden Dokumentation der XML-API war es mir wichtig, in diesem Kapitel einen detaillierten Einblick in jene zu bieten.

4.5.3 Renderbefehle der Standalone-Applikation von Cycles

Um eine XML-Datei zu rendern muss sie via den Terminal an Cycles übergeben werden. Dies kann einfach über den Aufruf des Programmes mit der angehängten XML-Datei geschehen.

```
$./cycles cube.xml
```

Dieser Befehl funktioniert nur, wenn man sich mit dem Terminal im Verzeichnis des Programmes selbst befindet; selbiges gilt für das "cube.xml"-File. Möchte man es zu den Systemvariablen des Linux-Systems hinzufügen, kann man das wie bereits beschrieben mit dem "export"-Befehl erledigen; dadurch ist es auch nicht länger eine Voraussetzung, das Programm mit dem Befehl "./" aufzurufen. Andernfalls ist es auch möglich, den absoluten Pfad zum Executable anzugeben. Die XML-Datei kann sowohl mit relativem Pfad, als auch absolutem Pfad definiert werden. Wird dieser Befehl zum Rendern der XML-Datei ausgeführt, öffnet Cycles ein Fenster, welches als Framebuffer fungiert. Angezeigt wird das Bild mit einer laufend steigenden Anzahl an Samples.

Diesen grundlegenden Renderbefehl kann man durch zusätzliche Informationen ergänzen. So ist es möglich, direkt über dieses Terminal-Interface die Grundgröße des Framebuffers festzulegen; "--width %d" und "--height %d" müssen dem Renderbefehl ergänzt werden. "--sampes %d" gibt die zu verwendende Anzahl an Pixelsamples an.

Um ein Beispiel zu nennen:

```
$cycles --samples 100 --width 1280 --height 720 cube.xml
```

Weiters ist es durch die Angabe von weiteren Befehlen möglich, ein für das Rendering zu verwendendes Gerät festzulegen. Dabei kann es sich um ein CPU- oder ein OPENCL-Gerät handeln. Eine Auflistung der möglichen Geräte kann man mit "--list-devices" aufrufen. Auswählen kann man das Gerät mit dem Befehl "--device CPU" oder "--device OPENCL"

Cycles verwendet standardmäßig alle Threads, welche zur Verfügung stehen. Mit dem Befehl "--threads %d" kann allerdings auch eine genaue Anzahl an Threads festgelegt werden, welche verwendet werden sollen.

Das Programm Cycles kann seine Berechnungen nicht nur in den Framebuffer schreiben, sondern auch in eine Bilddatei. Dazu ist der “--background”-Befehl notwendig. Dieser schreibt die Bilddatei an den mit dem Befehl “--output %s” festgelegten Pfad. Wird ein Programm unter der Verwendung des “--background”-Befehls ausgeführt, gibt Cycles den Fortschritt des Renderings innerhalb des Terminals aus; gezeigt werden sowohl die Anzahl an gesendeten Strahlen/Samples als auch der Prozentsatz des Fortschrittes. Diese Anzeige des Fortschrittes kann mit dem Befehl “--quiet” deaktiviert werden.

4.6 Das Plug-in

In diesem Kapitel soll nun ausführlich die Erstellung des Plug-ins für The Foundrys Nuke beschrieben werden. Wie bereits erwähnt soll die Implementierung Python basiert sein. Der ursprüngliche Plan, die Implementierung über ein C++-Plug-in für Nuke zu lösen, hat sich für die Verwendung in diesem ersten Prototypen als nicht rentabel gezeigt. Eine Verwendung des NDKs - der C++-API von Nuke - würde zusätzliche Funktionen bereitstellen, gleichzeitig allerdings auch Einschränkungen im Bereich der Plattformkompatibilität bringen.

Nukes Python-API bietet eine sehr umfangreiche Möglichkeit zum Umgang mit Nuke auf einem sehr hohen Level. Es bietet, wie bereits erwähnt, keine Möglichkeit zur Manipulation von Bildinhalten oder 3D-Daten. Es ist zum Skripten von Automationen und Pipeline Tools geeignet.

Die Python-API ist von The Foundry über den Maßen gut dokumentiert und benötigt an dieser Stelle keine weiteren Erklärungen zu ihrem Funktionsprinzip. Die Beschreibung des Plug-ins, welche in diesem Kapitel folgt, bietet außerdem ausreichend Information dazu.

Das Plug-in besteht innerhalb von Nuke grundsätzlich aus einer Haupt-Node. Eine zusätzliche Node, um eine für den Renderer zugeschnittene Lichtquelle zu bieten, wurde außerdem implementiert.

Es gilt festzuhalten, dass es sich beim folgenden Plug-in lediglich um einen Prototyp handelt, der nicht mehr als die Grundfunktionalitäten von Cycles innerhalb von Nuke zur Verfügung zu stellen versucht. Das Plug-in bietet nur eingeschränkte Parameter, zeigt aber seine grundsätzliche Funktion.

Um eine Version dieses Plug-ins “production-ready” zu gestalten, wären zahlreiche weitere Schritte nötig; die Portierung eines Teils der Funktionen in die C++-API wäre sicherlich einer dieser Schritte. Auch müsste die Shader Zuweisung

adaptiert werden sowie einfach eine Vielzahl an anderen Parametern im grafischen Interface zur Verfügung gestellt werden.

Die Einschränkungen der Implementierung via Python, verursacht durch die Python-API von Nuke, werden im Laufe des nächsten Kapitels hervorgehoben.

4.6.1 cy_render

Die “cy_render”-Node - “cy” soll dabei eine Abkürzung für Cycles bilden - innerhalb von Nuke stellt den Kern der Implementierung dar. Sie erledigt alle elementaren Aufgaben, welche für das Rendering mit Cycles in Nuke notwendig sind. Dazu zählt sowohl das Exportieren der Scene Description, der 3D-Szenenbeschreibung, als auch der Aufruf des Renderingbefehls. In dieser Version des Prototyps, welcher hier in dieser Arbeit dargestellt wird, enthält die “cy_render”-Node außerdem noch zusätzlich Funktionen, die bei einer Weiterentwicklung in separate Nodes ausgelagert werden müssten. Jene sind in diesem Fall nur für Testzwecke in der “cy_render”-Node implementiert. So wird in dieser Version noch das Shading auf globaler Ebene angewendet und innerhalb der Renderer-Node definiert. Auch erfolgt die Implementierung des Umgebungslichtes in der “cy_render”-Node direkt. Da dies allerdings mehr oder weniger einer globalen Einstellung entspricht und nicht mehr als ein Environmentlight definiert wird, kann dies beibehalten werden. Globale Parameter, welche den Integrator betreffen, finden in dieser Version keinerlei Implementierung.

Wie bereits erwähnt findet in dieser Version die Shaderzuweisung auf einer globalen Ebene statt. Eine Differenzierung war zu diesem Zeitpunkt nicht notwendig, da generell noch keine Funktionalität zum Rendering von mehreren Objekten gegeben war.

Zur Programmierung verwendet habe ich ein Addon für Nuke von Adrian Pueyo, den KnobScripter. Dieses Addon ergänzt den Funktionsumfang von Nuke um einen auf Python-Scripting ausgelegten Texteditor. Der KnobScripter erlaubt speziell die Manipulation von Node-Knobs und stellt dabei Funktionen wie Syntax-Highlighting und automatische Einrückungen zur Verfügung. Durch die Verwendung dieser Erweiterung konnte ich den Großteil des Plug-ins direkt in Nuke verfassen.

Grundstein des Plug-ins ist eine “Group”-Node, welche mit einem “Python Script Button” ausgestattet wurde. Dieser Button, mit dem Label “render” versehen, beinhaltet alle Funktionen, welche für das grundlegende Rendering notwendig sind. Im folgenden Text ist das Python-Script des “render”-Buttons ausführlich aufgezeigt.

Die Zeilennummerierung ist durchgängig und entspricht jener des Anhangs.

```
17
18          import os, sys
19          import subprocess
20          from xml.dom import minidom
21
```

In Zeile 18 bis 20 werden einige Standardbibliotheken von Python importiert. Der Import der Module “os” und “sys” der Standardbibliotheken hat sich als eine Konvention meinerseits ergeben. Im folgenden Programm wird nicht auf den Funktionsumfang jener zurückgegriffen. “os” bietet eine Vielzahl an Funktionen, welche das Betriebssystem betreffen. Eine Verwendung dieser erlaubt es unter anderem, dass gewisse Programmteile plattformunabhängig gestaltet werden können. Außerdem kann “os” einige Aufgaben des Moduls “subprocess” übernehmen. Die Funktionen “os.system” und “os.popen” erlauben beispielsweise den Aufruf von Kommandos im Terminal (Ernesti, Kaiser, 2017, p. 509).

Das Modul “sys” der Standardbibliothek stellt verschiedenste Variablen und Funktion zur Verfügung. Jene stehen im Regelfall in einem direkten Zusammenhang mit dem Python-Interpreter selbst (Ernesti, Kaiser, 2017, p. 512).

“subprocess” stellt ein Modul dar, welches ähnlich wie auch schon “os” neue Prozesse erstellen, sich mit deren Input, Output und Fehlercodes verbinden und die generierten Rückgabewerte auslesen kann. “subprocess” soll dabei die von “os” zur Verfügung gestellten Funktionen zur Prozesssteuerung ersetzen (Python Software Foundation, 2018).

Deshalb fand dafür im folgenden Programm auch “subprocess” Anwendung und nicht “os”.

Die Standardbibliothek von Python umfasst zwei Parser, welche die Verarbeitung von XML-Dateien erleichtern sollen. Beide entstammen dem Modul “xml”. Der “ElementTree”-Parser ist speziell für Python zugeschnitten und besitzt besonders beim Lesen von XML-Datein große Vorteile. “dom” hingegen implementiert das standardisierte Document Object Model (Ernesti, Kaiser, 2017, p. 579).

Da im folgenden Programm nur ein sehr begrenzter Funktionsumfang der XML-Parser benötigt wird, habe ich mich für den leichtgewichtigen, standardisierten “dom”-Parser entschieden, auch wenn bei ersten Versionen dieses Programmes

der “ElementTree”-Parser Verwendung fand. Der Umgang mit dem “dom”-Parser gestaltete sich für mich als einfacher und ließ mich umschwenken.

Zur Vollständigkeit kann weiters das Modul “nuke” und “nukescripts” importiert werden. Da der Quellcode allerdings ausschließlich in Nuke verwendet wird, besteht dafür keine Notwendigkeit, da dies hierbei automatisch passiert.

```
24
25        thisNode = nuke.thisNode()
26
```

In Zeile 25 wird eine Variable definiert, welche jene Node speichert, innerhalb welcher der Python-Code ausgeführt wird. Es zeigt sich dabei als Vorteil die Variable zu speichern, da der Kontext, auf welchen sich der Code bezieht, während des Fortlaufens des Programmes verändert werden kann und dieser Aufruf der Funktion nicht mehr möglich ist.

```
28
29        #path generation
30        path = nuke.root().knob('name').value()
31
32        #remove .nk-extension
33        x=3
34        while x>0:
35            path = path[:-1]
36            x = x-1
37
38        path = path.split("/")
39
40        #get name of nuke script
41        scriptName = path[-1]
42
43        del path[-1]
44
45        #path of nukescript
46        path = "/".join(path)
47
```

Die Zeilen 29 bis 46 beziehen sich auf die Generierung verschiedenster Variablen, welche in einem Zusammenhang mit dem Dateipfad und Namen des Nukescripts stehen. Zeile 30 liest den vollständigen Pfad des Files aus. Das beinhaltet auch

den Namen des Scripts, inklusive Dateinamenserweiterung; die Dateinamenserweiterung wird in Zeile 33 bis 36 entfernt. Der überbleibende String wird in Zeile 38 in seine einzelnen Komponenten zerlegt und als Liste abgespeichert. Das letzte Element der Liste entspricht dem Namen des Nukescripts; jener wird in Zeile 41 einer Variablen zugewiesen und in Zeile 43 aus der Liste entfernt. Anschließend wird der Dateipfad wieder zusammengesetzt und gespeichert (Zeile 46).

Es gilt zu beachten, dass in dieser Passage zum Teilen und wieder Zusammenfügen des Pfades keine Rücksicht auf eine Kompatibilität mit anderen Betriebssystemen genommen wurde. Da das Betriebssystem Windows von Microsoft "\" anstelle von "/" verwendet um seine Pfade zu unterteilen, ist dieser Bereich nicht mit Windows kompatibel. Dies lässt sich mit Funktionen aus dem Modul "os" umgehen, welche abhängig vom Betriebssystem den richtigen Schrägstrich wählen.

Außerdem wurde an dieser Stelle noch keine Ausnahmebehandlung für den Fall implementiert, sollte das Nukescript noch nicht gesichert sein. In diesem Fall würde Nuke selbst eine Fehlermeldung auswerfen und das Python-Script beenden. Dies kann mit "try - except" festgestellt und gegebenenfalls darauf reagiert werden. Entweder mit einer eigenen Fehlernachricht (nuke.message()), welche den Grund für das Scheitern angibt oder mit einem Pfad zu einem temporären Verzeichnis.

```
53
54        def invertY(l):
55            for indx, val in enumerate(l):
56                if(indx % 3 == 1):
57                    val = val*-1
58                    l.pop(indx)
59                    l.insert(indx, val)
60                    return l
61
62
63        #adopt rotation... does not work 100%
64
65        def rotateX(l):
66            for indx, val in enumerate(l):
67                if(indx % 3 == 0):
68                    val = val+180
69                    l.pop(indx)
70                    l.insert(indx, val)
71            return l
```

Nuke und Cycles unterscheiden sich in ihren Transformationsmatrizen. Es gestaltete sich als schwierig für mich herauszufinden, wodurch sie sich unterscheiden. Reines Experimentieren zeigte eine Reihe von Unterscheidungen, deren Ausgleich ich in zwei Funktionen definierte (Reihe 54 bis 71). Die erste Funktion behebt die Tatsache, dass der y-Wert der Translation mit “-1” multipliziert werden muss, um den gleichen Output von Nuke und Cycles zu bekommen. Zweite Funktion fügt zum x-Wert der Rotation “180” hinzu. Dies ermöglicht identische Bilder innerhalb von Nuke und Cycles; allerdings nur solange, wie der Kamera keine weiteren Werte für die Rotation übergeben werden. In diesem Fall verhält sich die Kamera auf mir bisher ungeklärte Weise. Das Problem, die Kamera nicht weiter rotieren zu können, ist eines, welches ich trotz höchster Priorität bis zum Schluss nicht lösen konnte.

```
75
76        def groupVectors(lst, n):
77            return zip(*[lst[i::n] for i in range(n)])
78
```

Die dritte Funktion, welche an dieser Stelle definiert wird, befasst sich mit der Formatierung der x-, y- und z-Position der einzelnen Punkte der Geometrie. Sie gruppiert die ihr übergebenen Elemente einer Liste in neuen Listen.

Die folgenden Zeilen beschäftigen sich ausschließlich damit, Informationen aus Nuke aus zu lesen und weiter zu verarbeiten:

```
84
85        #get render resolution
86
87        resolution = nuke.toNode("getResolution")
88        resolutionX = str(resolution.width())
89        resolutionY = str(resolution.height())
90
```

Das umfasst von Zeile 87 bis 89 Informationen, welche die zu rendernde Auflösung betreffen. Die Konvention in Nuke diesbezüglich ist jene, dass die Auflösung des “bg”-Inputs hierfür Verwendung findet. Ist nichts verbunden mit dem Input, wird die Auflösung des Nukescriptes übernommen; wird eine “Reformat”-Node, eine “Constant”-Node oder ein beliebiger Hintergrund mit dem Input verbunden, so wird die Information der Auflösung von jenen übernommen. Diese Implementierung

erfolgt über die Verwendung eines Platzhalters, einer “NoOp-Node” innerhalb der “cy_render”-Node, welche für das Auslesen der Auflösung herangezogen wird.

```
93
94        samples = thisNode.knob("samples").getValue()
95
96        #get Camera Transform
97
98        camInput = thisNode.input(1)
99
100        camTranslate = camInput.knob("translate").getValue()
101        camTranslate = ' '.join(map(str,invertY(camTranslate)))
102
103        camScale = camInput.knob("scaling").getValue()
104        camScale = ' '.join(map(str,camScale))
105
106        camRotate = camInput.knob("rotate").getValue()
107        camRotate = ' '.join(map(str,rotateX(camRotate)))
108
```

Zeile 94 befasst sich mit Informationen, welche die für das Rendering zu verwendenden Samples betreffen. Jene können innerhalb eines Integer-Knobs der “cy_render”-Node übergeben werden. Die Werte der Transformation der Kamera werden in Zeile 98 bis 107 eingelesen und verarbeitet. Andere Werte werden derzeit nicht ausgelesen; die default-Werte von Nuke und Cycles sind ident. Um dies in einer sinnvollen Art und Weise mit allen für Cycles wichtigen Werte zu ermöglichen, müsste eine eigene “camera”-Node für Cycles in Nuke implementiert werden. Dies ist im Moment noch nicht geschehen. Auch ist hier noch keine Ausnahmebehandlung integriert, sollten eine andere oder keine Node mit dem Input verbunden sein.

```
109
110        #get background Info
111
112        envColor = thisNode.knob("envColor").getValue()
113        envColor = ' '.join(map(str,envColor))
114
115        envIntensity =
str(thisNode.knob("envIntensity").getValue())
116
```

Zeile 112 bis 115 verarbeitet Informationen zum globalen Umgebungslicht. Die Abfrage des Knobs, ob die Szene über ein derartiges Licht verfügen soll, erfolgt später.

```
118
119         #get Light Input
120
121         lightInput = thisNode.input(3)
122
123         #get lightInfo
124
125         if lightInput.Class() == "cy_Light":
126             lightConnected = True
127
128             lightType = lightInput.knob("lightType").getValue()
129             lightColor = lightInput.knob("color").getValue()
130             lightColor = ' '.join(map(str,lightColor))
131             lightIntensity =
str(lightInput.knob("intensity").getValue())
132             lightSize = str(lightInput.knob("size").getValue())
133             lightTranslate =
lightInput.knob("translate").getValue()
134             lightTranslate = '
'.join(map(str,invertY(lightTranslate)))
135
136             lightScale = lightInput.knob("scaling").getValue()
137             lightScale = ' '.join(map(str,lightScale))
138
139             lightRotate = lightInput.knob("rotate").getValue()
140             lightRotate = ' '.join(map(str,rotateX(lightRotate)))
141
142         else:
143             pass
144
```

An dieser Stelle wird das mit der Node verbundene Licht und Informationen dazu betreffend abgefragt. Momentan muss das Licht noch getrennt von der Geometrie abgefragt werden, dafür sonst notwendige Abfragestrukturen wurden noch nicht implementiert. Dadurch kann gegenwärtig auch erst ein einzelnes Licht verwendet werden.

Da die möglichen Lichtsettings von Nuke nicht zur Gänze mit jenen von Cycles übereinstimmen, wurde eine eigene “cy_light”-Node implementiert. Dazu später mehr.

Diese Funktion umfasst bereits eine Ausnahmebehandlung sollte kein Licht verbunden sein oder sollte man nur mit Umgebungslicht rendern wollen.

```
146
147         #get shader info // mix shader
148
149         diffuseColor = thisNode.knob("diffuseColor").getValue()
150         diffuseColor = ' '.join(map(str,diffuseColor))
151
152         diffuseRoughness =
str(thisNode.knob("diffuseRoughness").getValue())
153
154
155         glossyColor = thisNode.knob("glossyColor").getValue()
156         glossyColor = ' '.join(map(str,glossyColor))
157
158         glossyRoughness =
str(thisNode.knob("glossyRoughness").getValue())
159
160
161         shaderMix = str(thisNode.knob("shaderMix").getValue())
162
163
```

Zeile 149 bis 161 umfassen alle Abfragen bezüglich Informationen zum Shader. Seine Einstellungen lassen sich via “cy_render”-Node festlegen. Es handelt sich um die Implementierung eines Mix-Shaders, welcher zwischen einem Diffuse- und einem Glossy-Shader mischen kann. Die jeweilige Menge wird mit Hilfe eines Schiebereglers festgelegt.

```
164
165         #build Geo from nuke scene description for cycles xml api
via PythonGeo-Node
166
167         #P - vertex position
168         pyGeo = nuke.toNode("PythonGeo1")
169         gObj = pyGeo['geo'].getGeometry()[0]
170         gObj = list(gObj.points())
171
172         gObj = groupVectors(gObj,3)
173
174         gObjList = []
175
176         for c in gObj:
```

```
177             c = invertY(list(c))
178             c = ' '.join(map(str, c))
179             gObjList.append(c)
180 
181 
182         #gObj = ' '.join(l + ' ' * (n % 3 == 2) for n, l in
enumerate(map(str, gObj)))
183         gObj = '  '.join(gObjList)
184 
185 
186 
187         #verts
188         verts = pyGeo['geo'].getGeometry()[0]
189         verts = verts.primitives()
190         listVerts = []
191 
192 
193 
194         for c in verts:
195             listVerts.extend(c)
196             listVerts.extend("x")
197 
198         listVerts = ' '.join(map(str,listVerts))
199 
200         listVerts = listVerts.replace("x", "")
201 
202 
203 
204         #nverts
205 
206         listNVerts = []
207 
208         for c in verts:
209             nVerts = str(len(c))
210             listNVerts.extend(nVerts)
211 
212         listNVerts = ' '.join(listNVerts)
213 
```

Die Übergabe der Geometrie an Cycles erfolgt in Zeile 168 bis 212. Jene Information wird unter der Verwendung einer “PythonGeo”-Node von Nuke ausgelesen. Bei den von Nuke zurückgegebenen Werten handelt es sich um Tuples, welche allerdings ähnlich zu dem Format sind, welches Cycles über seine XML-API erwartet. Die “PythonGeo”-Node ist dabei nicht über die Benutzeroberfläche verfügbar, sondern lediglich über das Python-Interface von Nuke selbst. Die “PythonGeo”-Node ist nicht dokumentiert, allerdings ist ihr C++-

Source-Code in der Dokumentation des NDKs als Beispiel inkludiert. Aus diesem konnte ich ableiten, dass durch die Verwendung von “.primitives()” und “.points()” die notwendigen Daten generiert werden können. Die “.points()”-Funktion gibt die x-, y- und z-Position der einzelnen Punkte aus; “.primitives()” gibt die einzelnen Punktnummer zurück, welche durch Primitives miteinander verbunden sind. Beides Mal handelt es sich um ein zweidimensionales Array beziehungsweise um Tuples. Die Anzahl der Punkte, welche die einzelnen Primitives zusammensetzten, lassen sich einfach aus der Länge der Liste an Punktnummern ableiten.

Der folgende Teil des Programms befasst sich mit der Erstellung der XML-Datei und ist dabei weitestgehend selbsterklärend.

```
216
217         #---gen XML---
218
219         root = minidom.Document()
220
221         xml = root.createElement("cycles")
222         root.appendChild(xml)
223
224
225
226         #---camera XML begin---
227
228         cameraChild = root.createElement('camera')
229         cameraChild.setAttribute("width",resolutionX)
230         cameraChild.setAttribute("height",resolutionY)
231         xml.appendChild(cameraChild)
232
233
234         transformChild = root.createElement("transform")
235         transformChild.setAttribute("translate",camTranslate)
236         transformChild.setAttribute("rotate", camRotate + " 0")
237         transformChild.setAttribute("scale", camScale)
238         xml.appendChild(transformChild)
239
240
241         cameraChild = root.createElement("camera")
242         cameraChild.setAttribute("type","perspective")
243         transformChild.appendChild(cameraChild)
244
245         #---camera XML end---
246
247
248
```

```
249         #---point light XML begin---
250 
251         shader = root.createElement('shader')
252         shader.setAttribute("name","point_shader")
253         xml.appendChild(shader)
254 
255 
256         emission = root.createElement("emission")
257         emission.setAttribute("name","emission")
258         emission.setAttribute("color",lightColor)
259         emission.setAttribute("strength",lightIntensity)
260         shader.appendChild(emission)
261 
262 
263         connect = root.createElement("connect")
264         connect.setAttribute("from","emission emission")
265         connect.setAttribute("to","output surface")
266         shader.appendChild(connect)
267 
268 
269         state = root.createElement("state")
270         state.setAttribute("shader","point_shader")
271         xml.appendChild(state)
272 
273 
274         light = root.createElement("light")
275         light.setAttribute("type","point")
276         light.setAttribute("co",lightTranslate)
277         light.setAttribute("size",lightSize)
278         state.appendChild(light)
279 
280         #---light XML end---
281 
282 
283 
284         #---background XML begin----
285 
286         if thisNode.knob("environmentLight").getValue() == 1.0:
287 
288             background = root.createElement("background")
289             xml.appendChild(background)
290 
291 
292             background1 = root.createElement("background")
293             background1.setAttribute("name", "bg")
294             background1.setAttribute("strength", envIntensity)
295             background1.setAttribute("color", envColor)
296             background.appendChild(background1)
297 
```

```
298
299             connect = root.createElement("connect")
300             connect.setAttribute("from","bg background")
301             connect.setAttribute("to","output surface")
302             background.appendChild(connect)
303
304         else:
305             pass
306
307         #---background XML end----
308
309
310
311         #---mesh XML begin----
312
313         shader = root.createElement('shader')
314         shader.setAttribute("name","mesh")
315         xml.appendChild(shader)
316
317
318         diffuse1 = root.createElement("diffuse_bsdf")
319         diffuse1.setAttribute("name","diffuse1")
320         diffuse1.setAttribute("color",diffuseColor)
321         diffuse1.setAttribute("roughness",diffuseRoughness)
322         shader.appendChild(diffuse1)
323
324
325         glossy1 = root.createElement("glossy_bsdf")
326         glossy1.setAttribute("name","glossy1")
327         glossy1.setAttribute("roughness",glossyRoughness)
328         glossy1.setAttribute("color",glossyColor)
329         shader.appendChild(glossy1)
330
331         mix1 = root.createElement("mix_closure")
332         mix1.setAttribute("name","mix")
333         mix1.setAttribute("fac",shaderMix)
334         shader.appendChild(mix1)
335
336
337         #connect0 = root.createElement("connect")
338         #connect0.setAttribute("from","checker fac")
339         #connect0.setAttribute("to","mix fac")
340         #shader.appendChild(connect0)
341
342
343         connect1 = root.createElement("connect")
344         connect1.setAttribute("from","diffuse1 bsdf")
345         connect1.setAttribute("to","mix closure1")
346         shader.appendChild(connect1)
```

```
347
348
349         connect2 = root.createElement("connect")
350         connect2.setAttribute("from","glossy1 bsdf")
351         connect2.setAttribute("to","mix closure2")
352         shader.appendChild(connect2)
353
354
355         connect3 = root.createElement("connect")
356         connect3.setAttribute("from","mix closure")
357         connect3.setAttribute("to","output surface")
358         shader.appendChild(connect3)
359
360
361
362         state = root.createElement("state")
363         state.setAttribute("shader","mesh")
364         if thisNode.knob("shadingType").getValue() == 1.0:
365             state.setAttribute("interpolation","smooth")
366         else:
367             pass
368         xml.appendChild(state)
369
370
371
372         mesh = root.createElement("mesh")
373         mesh.setAttribute("P",gObj)
374         mesh.setAttribute("verts",listVerts)
375         mesh.setAttribute("nverts",listNVerts)
376         state.appendChild(mesh)
377
378         #---mesh XML end----
379
380
381         #format xml for writing
382
383         xml_str = root.toprettyxml(indent="\t")
384
385
386
387         #gen folders for xml and image file
388
389         cyPath = path + "/" + "cycles"
390
391         try:
392             os.mkdir(cyPath)
393
394         except:
395             pass
```

```
396
397         cyProPath = cyPath + "/" + scriptName
398
399         try:
400             os.mkdir(cyProPath)
401         except:
402             pass
403
404
405         #gen pathes for rendering-command
406
407         imagePath = str(cyProPath + "/" + scriptName + ".exr")
408
409         xmlPath = str(cyProPath + "/" + scriptName + ".xml")
410
411
412
413         #write xml file to disc
414
415         with open(xmlPath, "w") as f:
416                 f.write(xml_str)
417
418
419
420         #get path of cycles executable
421
422         execPath = thisNode.knob("execPath").getValue()
423
```

Um den Pfad zu Cycles Executable anpassen zu können umfasst die “cy_render”-Node über eine Pfad-Eingabe (Zeile 422).

```
425
426         #---Rendering---
427
428
429         #rendering command for rendering to framebuffer
430
431         if thisNode.knob("renderTo").getValue() == 0.0 :
432             subprocess.call("%s --height %s --width %s --samples %s
%s" % (execPath+"cycles", str(resolutionY), str(resolutionX),
str(samples), xmlPath), shell=True)
433
434
435         #rendering command for rendering to disk/nuke
436
```

```
437         elif thisNode.knob("renderTo").getValue() == 1.0 :
438             subprocess.call("%s --background --output %s --height
%s --width %s --samples %s %s" % (execPath+"cycles", imagePath,
str(resolutionY), str(resolutionX), str(samples), xmlPath),
shell=True)
439             nuke.toNode("Read1").knob("file").setValue(imagePath)
440             nuke.toNode("Read1").knob("reload").execute()
441
442         # only writing the xml file to disc - no rendering comman
is executed
443
444         else:
445             pass
446
```

Den Abschluss bildet der Renderbefehl. Welcher Renderbefehl ausgeführt wird, lässt sich durch eine Dropdown-Liste festlegen. Die Option zum Rendern in den Cycles-Framebuffer (Zeile 431) und in eine Bilddatei, welche umgehend nach dem Schreiben wieder in Nuke eingelesen wird (Zeile 437), stehen zur Verfügung. Weiters besteht auch die Möglichkeit, dass lediglich die XML-Datei geschrieben wird, kein Renderingaufruf durch den ".subprocess"-Befehl wird ausgeführt.

Wie bereits einleitend erwähnt handelt es sich bei dem Plug-in um einen Prototyp. Er verfügt über begrenzte Fähigkeiten sowie keine Ausnahmebehandlung, ist allerdings lauffähig. Eine Portierung in C++-Code wären neben der Fehlerfindung in Bezug auf die unterschiedlichen Transformationsmatrizen die nächsten Schritte. Die Beispiele des NDKs enthalten Vorlagen, welche dahingehend abgeleitet werden können; etwa die "PythonGeo"-Node und ein Beispiel für eine personalisierte "GeoWrite"-Node.

4.6.2 cy_light

Die zweite Node des Plug-ins implementiert eine adaptierte "light"-Node, welche über die speziell für Cycles notwendigen Parameter besitzt. Bei der "cy_light"-Node handelt es sich um eine "Group"-Node, welche im Inneren aus einer "light"-Node besteht. Dies hat zum Vorteil, dass trotz der Personalisierung die Lichtquelle auch im "3D Viewer" über Auswirkungen verfügt. Durch die Platzierung innerhalb einer "Group"-Node besteht allerdings die Möglichkeit, dass lediglich jene Parameter angezeigt werden, die für die Verwendung von Cycles von Interesse sind. Zusätzlich können weiter Parameter, welche Cycles-spezifisch sind, ergänzt werden.

Bei dem implementierten Licht handelt es sich lediglich um ein Punktlicht, um die Funktionalität zu beweisen. Es implementiert alle Parameter, welche für das

Punktlicht relevant sind: Intensität, Farbe, Größe der Lichtquelle und Position. Dabei ist Farbe, Intensität und Position direkt mit der “light”-Node innerhalb der “Group”-Node verbunden. Folglich haben Änderungen sowohl Auswirkung auf den “3D Viewer” als auch auf das Rendering mit Cycles. Die Größe der Lichtquelle ist ein Parameter, welcher nicht in der “light”-Node implementiert ist, da es sich bei der jener nativ in Nuke implementierten um ein unendlich kleines Licht handelt. Da Cycles Ansätze des physikalisch korrekten Renderings verfolgt, verfügt das Punktlicht, das durch Cycles verwendet wird, jedoch über eine tatsächliche Größe. Jene ist durch einen Schieberegler einzustellen.

Die “cy_light”-Node muss als Gizmo exportiert werden und kann nicht als “Group”-Node verwendet werden, da die “cy_render”-Node eine Abfrage der Nodeklasse zur Ausnahmebehandlung durchführt. Ein Eintrag in der “menu.py”-Datei im “.nuke”-Verzeichnis und ein Neustart von Nuke sind erforderlich.

Auf gleiche Art und Weise kann auch eine personalisierte Kamera-Node für die Verwendung mit Cycles erstellt werden, welche über die für Cycles ausschlaggebenden Parameter verfügt; beispielsweise Parameter für Panoramaaufnahmen oder die Stereoskopie.

4.6.3 Implementierung Cycles Color Mapping

Im Kapitel über die fortgeschrittene Renderpipeline wurde die Verwendung von logarithmischer Komprimierung der Helligkeitswerte beschrieben. Die Verwendung der “SoftClip”-Node wurde hierzu aufgezeigt. Blender selbst, wie auch V-Ray, verfügt über sein eigenes Farbmanagement, welche eine Alternative diesbezüglich bereitstellt. Es ist selbstverständlich möglich Standard-Farbräume wie “linear” oder “sRGB” zu verwenden. Viele Nutzer von Blender greifen allerdings auf eine Implementierung eines logarithmischen Farbraumes von Troy Sobotka zurück - Filmic Blender. Jener sorgt für eine logarithmische Komprimierung von 0 bis unendlich beziehungsweise dem Maximalwert des Gleitkommazahlen-Standards auf 0-1. Weiters entsättigt er die Highlights des Bildes. Ein weit realistischer Bildeindruck ist das Ergebnis. Eine Integration in Realaufnahmen wird dadurch stark vereinfacht.

Es ist wünschenswert, die gleichen Ergebnisse von Renderings innerhalb von Blender und Nuke zu erlangen; folglich ist eine Implementierung von Filmic Blender in Nuke anzustreben.

Die Implementierung ist aus einem weiteren Grund anzustreben, da wie bereits erwähnt, die Anwendung lediglich auf den Beauty-Pass sinnvoll ist. Um Multipass-

Compositing zu erlauben sowie diesen Farbraum nicht auf Cycles zu limitieren, ist dessen Anwendung auf das lineare Rendering innerhalb von Nuke zu bevorzugen.

Im Vergleich zur “SoftClip”-Node in Nuke zeichnet die Komprimierung durch Filmic Blender die extremen Highlights im Bild noch weicher und kann zusätzliche Informationen im Schatten abbilden. Auch ist eine etwas stärkere Entsättigung der hellen Stellen im Bild gegeben. Die untenstehende Abbildung (Abbildung 2) zeigt das originale Rendering im Vergleich zu den Ergebnissen der “SoftClip”-Node und zur Filmic Blender-Implementierung in Nuke.

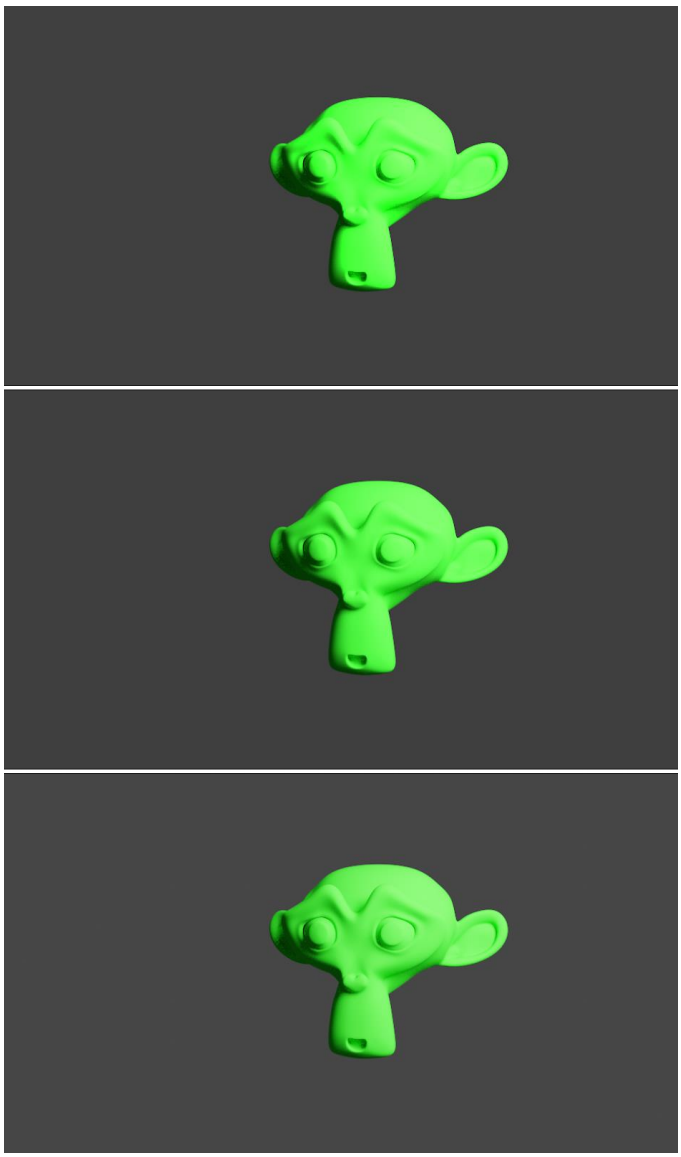

Abbildung 2. Im obersten Bild ist das stark überbelichtete originale Rendering zusehen; in der Mitte jenes mit logarithmischer Komprimierung durch die “SoftClip”-Node; unten das selbe Rendering mit der Komprimierung durch Filmic Blender.

Dieser Farbraum kann aufgrund der Tatsache implementiert werden, da sowohl Nuke, als auch Blender OpenColorIO für ihr Farbmanagement verwenden. Dieses Farbmanagement wird dabei im “config.ocio”-File und den dazugehörigen Luts festgelegt. Jene Konfigurationsdatei, welche Nuke standardmäßig verwendet, befindet sich dabei im Installationsverzeichnis von Nuke unter “/plugins/OCIOConfigs/configs/nuke-default”. Es empfiehlt sich, diese “config.ocio”-Datei nicht zu manipulieren, da es sich hierbei um jenes File handelt, auf welches Nuke bei Problemen mit einem benutzerdefiniertem Konfigurationsfiles zurück greift. Eine Kopie sowohl der Konfigurationsdatei als auch des “luts”-Ordners, innerhalb des “.nuke”-Verzeichnisses zu erstellen liegt nahe.

Die “config.ocio”-Datei von Filmic Blender kann über das GitHub-Repository “filmic-blender” bezogen werden (Sobotka, 2018).

Die Implementierung ist dabei ähnlich zu jener, die Maxime Roz auf seiner Homepage beschreibt (http://www.maximeroz.com/filmic-nuke/), umfasst allerdings einige weitere Schritte, welche für eine uneingeschränkte Verwendung in Nuke notwendig sind; etwa werden auch die von Nuke gebotenen Farbräume beachtet, sowie der “Viewer Color Space”.

Nun gilt es, die Einträge des von Nuke zur Verfügung gestellten “config.ocio”-Files mit jenen von Filmic Blender zu ergänzen. Besonderer Wert ist auf die Beibehaltung der richtigen Syntax zu achten, da eine Verwendung sonst ausgeschlossen ist. Die Luts von Filmic Blender sind in das “luts”-Verzeichnis von Nuke zu verschieben. Wird Nuke nun gestartet, muss in den “Project Settings” unter “Color” das “color management” von “Nuke” zu “OCIO” geändert werden. Anschließend kann das “OCIO config” von “nuke-default” auf “custom” gestellt werden. Unter “custom OCIO config” kann der Pfad zur neuen “config.ocio”-Datei angegeben werden. Nun sollten innerhalb von Nuke, sowohl die standardmäßigen Farbräume von Nuke selbst als auch jene die Filmic Blender umschließt, vorhanden sein.

Wird nun das Rendering als lineares Bild importiert, kann via “OCIO ColorSpace”-Node der Farbraum von “Linear” zu “Filmic Log Encoding” geändert werden. Filmic Blender beinhaltet weiters noch Lookup Tables, welche dafür ausgelegt sind, dass sie einen gewissen Grundkontrast der mit Filmic Blender encodierten Bilddatei hinzufügen. Dies kann unter der Verwendung der “OCIO FileTransform”-Node erzielt werden. Als “working space” ist dabei “Linear” beizubehalten.

Um den gewohnten “Viewer Color Space”, im Regelfall “sRGB”, verwenden zu können ist eine weitere “OCIO ColorSpace”-Node notwendig, welche die “sRGB”-

Transformation wieder umkehrt; sprich “in” ist mit “sRGB”, “out” mit “linear” zu wählen.

Ich habe jenes Konvolut an Nodes in einem Gizmo verarbeitet. Eine Dropdown-Liste mit den zur Verfügung stehenden Kontrast-Luts ist eingerichtet.

Derartig viele Farbraumtransformationen sind der Qualität nicht zuträglich, aber ein visuell identes Bild zur Ausgabe von Blender lässt sich erzielen. Auch finde ich das Ergebnis von Filmic Blender subjektiv ansprechender als jenes, welches durch den “SoftClip”-Algorithmus erzielt wird; auch für andere Renderengines.

4.7 Cycles zukünftige Entwicklungen

Cycles ist eine Renderengine mit einigen aktiven Entwicklern, welche größtenteils in ihrer Freizeit an der Verbesserung der Renderengine arbeiten. Jene versuchen den Funktionsumfang ständig weiter zu vergrößern. Dies umschließt auch Entwicklungen, welche im Hinblick auf die Standalone-Applikation getroffen werden.

Die Entwickler von Blender und auch Cycles kommunizieren über ein öffentliches Forum, welches Auskunft darüber gibt, welche neuen Features und Veränderungen geplant sind; zusätzlich gibt es eine Zusammenfassung der Roadmap. Jene Entwicklungen, an welchen gegenwärtig gearbeitet werden, umschließen unter anderem den OpenVDB-Support sowie die Implementierung von Cryptomatte und auch Light Linking (Blender Foundation, 2018e).

Besonders von Interesse ist jenes Forum, welches sich mit der Verbesserung der Cycles Standalone-Applikation befasst. Die Diskussion reicht dabei schon einige Jahre zurück und umfasst eine komplette Neugestaltung der API. Anfänglich wurde in Betracht gezogen, das XML-Format mit Jason oder RIB zu ersetzen. Auch wurde der Plan geäußert, dass keine Notwendigkeit bestehe die Beschreibung der Geometrie im selben Format der Szenenbeschreibung durchzuführen. Eine Referenzierung auf ein externes Fileformat - im speziellen Alembic - wurde angedacht. Da besonders die Umstellung des Dateiformats für die Szenenbeschreibung mit großen Veränderungen einhergeht, wurde dies noch nicht umgesetzt. Zuletzt wurde von Brecht van Lommel, einer der Hauptentwickler von Cycles, eingebracht, die Verwendung von Universial Scene Description, USD, als natives Cycles-File-Format zu implementieren (Blender Developers, 2017).

Es scheint, als hätte sich mit dem bevorstehenden Release von Blender 2.8 der Fokus momentan allerdings mehr in Richtung Blender verschoben. Die Blender

Foundation konnte jedoch im Juli bekannt geben, dass mit Brecht van Lommel, der ursprüngliche Schöpfer von Cycles als Vollzeit-Entwickler für Blender und Cycles angestellt werden konnte (Veldhuizen, 2018b).

Weiters beschäftigt Tangent Animation, jenes Animationsstudio, welches Blender und Cycles für seine beiden Animationsfilme in Spielfilmlänge verwendete, über vier Vollzeit-Entwickler (stand 2016), welche die für sie notwendigen Funktionen implementieren. Tangent Animation, ansässig in Kanada, ist stets darum bemüht, seine Weiterentwicklungen in den offiziellen Blender Source-Code einfließen zu lassen und sich dadurch bei der Community, welche den Großteil der Entwicklung von Blender und Cycles betrieben hat, zu revanchieren. Weiters hat Tangent Animation mit Stand 2016 zwei Vollzeit-Entwickler der Blender Foundation gesponsert (Blender Foundation, 2016).

5 Anwendungsgebiete und Funktionsumfang

Dieses Kapitel, das letzte dieser Arbeit, umfasst ein Interview zur Feststellung der plausiblen Anwendungsgebiete und dem daraus resultierenden notwendigen Funktionsumfang von fotorealistischem Rendering innerhalb von Compositing-Applikationen. Die gewählte Methodik für das Interview soll dabei dem eines systematisierenden Experteninterviews entsprechen.

> *Das Erkenntnisziel systematisierender Experteninterviews liegt in der möglichst weitgehenden und umfassenden Erhebung des Sachwissens der Experten bezüglich des Forschungsthemas. Das Interview dient der systematischen Informationsgewinnung, und die Funktion des Experten liegt darin, „Ratgeber" zu sein: Wir lernen direkt von den Experten, und zwar in umfassender, analytischer Weise. (Bogner, Littig, Menz, 2014, p. 24)*

Bei dem systematisierenden Experteninterview ist das Ziel, die Generierung von Wissen, welches dem Befragten reflexiv zur Verfügung steht; das Wissen kann direkt abgefragt werden. Ein differenzierter Leitfaden, um die gewünschten Daten zu erheben ist erforderlich (Bogner, Littig, Menz, 2014, p. 24).

Ein Experte definiert sich durch die Eigenschaft, dass jener mit einem sicheren und eindeutigen Wissen hantiert, welches durch ihn zu jedem Zeitpunkt sowohl kommunikativ als auch reflexiv zur Anwendung kommen kann (Bogner, Littig, Menz, 2014, p. 12).

Die Auswertung des Interviews soll anhand einer qualitativen Inhaltsanalyse passieren. Jene eignet sich besonders gut für die Verarbeitung von systematisierenden Experteninterviews. Die Erhebung der Daten aus dem Interview durch diese Inhaltsanalyse lässt sich sehr stark verkürzt folgendermaßen zusammenfassen: Am Anfang steht das Transkribieren des Interviews, welches im Anschluss durch eine Kategorisierung unterteilt wird. Aus diesen Kategorien kann anschließend eine Extraktion der Rohdaten durchgeführt werden, die zur Auswertung herangezogen werden (Bogner, Littig, Menz, 2014, p. 72).

Die Prämisse des Experteninterviews ist jene, dass das durch den Experten vorgebrachte Wissen als Faktum zu betrachten ist, auch wenn viele Aspekte die

persönliche Meinung des Interviewten beinhalten (Bogner, Littig, Menz, 2014, p. 12).

Die Transkription des Interviews versucht dem genauen Wortlaut der Aufnahme zu folgen.

5.1 Interviewpartner

Als Interviewpartner konnte ich Ass.-Prof. Dipl.-Ing. (FH) Valentin Struklec gewinnen. Valentin Struklec studierte an der Fachhochschule Joanneum. Es folgte die erste Anstellung bei einer Filmproduktion. Der ausführlichen Arbeit mit After Effects folgte der Einsatz als Visual Effects Supervisor. Daraufhin wechselte er zu Friendly Fire und nach weiteren zwei Jahren zu MPC nach London. Dort arbeitete er als Compositing-Artist an Filmen wie “X-Men: Erste Entscheidung” und “Prometheus - Dunkle Zeichen”. Es folgte der Wechsel zu MPC Vancouver, wo er ebenfalls an internationalen Spielfilm-Produktionen als Compositing-Artist arbeitete. Anschließend kehrte er wieder nach Österreich zurück, wo er die Stelle als Assistenz-Professor an der Filmakademie Wien für den Masterstudiengang “Digital Art - Compositing” antrat. Weiters gründete er 2013 die Firma vast, welche mittlerweile mehrere Mitarbeiter umfasst und durchaus große Produktionen umsetzt. 2016 arbeitete er außerdem für mehrere Monate als Compositing-Artist für Industrial, Light und Magic, ILM.

5.2 Kategorien der erhobenen Daten

Das Transkript ist ungekürzt und in weitestgehend originalem Wortlaut im Anhang dieser Arbeit inkludiert. Grob lassen sich drei Hauptkategorien aus jenem Gespräch ableiten:

- die aktuelle Situation,
    a. einerseits von Nukes internen Renderer,
    b. andererseits von den angebotenen Renderern von Drittanbietern,
- der gewünschte Funktionsumfang,
- und mögliche Anwendungsgebiete,
    a. sowohl in großen,
    b. als auch kleinen VFX-Studios.

Jedes dieser Kapitel konnte mit ausführlichem Umfang im Interview abgehandelt werden. Im folgenden Text werden die erhobenen Daten präsentiert.

5.3 Die aktuelle Situation

Nuke umfasst gegenwärtig zwei Renderer, den Scanline Renderer und den Rayrender Renderer. Trotz der Tatsache, dass der Rayrender Renderer die aktuellere und technisch fortgeschrittenere Lösung darstellt, handelt es sich beim Scanline Renderer immer noch um den in Produktionen geläufigen. Der Rayrender Renderer kommt lediglich in speziellen Use Cases zum Einsatz, etwa bei der Notwendigkeit von Reflektionen. Spiegelungen sind nicht im Funktionsumfang des Scanline Renderers eingeschlossen. Über die Eigenschaft, fotorealistische Rendering anzufertigen, verfügt allerdings auch der Rayrender Renderer nicht; zu ausgeprägt sind seine Einschränkungen.

Mittlerweile bieten zahlreiche Drittanbieter Plug-ins ihrer Renderer für Nuke an. Bei jener von Chaos Group, dem V-Ray Plug-in, handelt es sich hierbei um das vielversprechendste. Große Anwendung findet allerdings auch dieses nicht, da es sich um ein schwer zu erfüllendes Anforderungsprofil handelt, wenn ein Compositing-Artist zusätzlich über vertiefendes Wissen in einer Renderengine verfügen muss.

Eine weite Verbreitung findet das Plug-in Element 3D für die Software After Effects. Diese gehäufte Anwendung ist der Tatsache geschuldet, dass es sich bei dem Produkt von Adobe um eines handelt, welches auf eine einfachere Bedienung als node-basierte Compositing-Applikationen setzt. Durch diese Demokratisierung des Produkts werden auch viele Gelegenheitsnutzer angesprochen, deren oberstes Ziel nicht zwingend höchste Qualität und Flexibilität ist. Fortgeschrittenere 3D-Programme und Compositing-Pakete bedienen lediglich hochspezialisierte Randzielgruppen.

Immer mehr zeigt sich im Moment der Trend, dass Anbieter versuchen, eine Department-übergreifende Softwarelösung zu entwickeln. Dies macht in Hinblick auf Lizenzkosten und einer Standardisierung der Pipeline Sinn, nicht allerdings bei der Spezialisierung von Artists. Auch ist eine derartige übergreifende Software, zumindest teilweise, in ihrer Möglichkeit beschränkt, stets auf die Wünsche der einzelnen spezialisierten Anwendungsgebiete einzugehen.

5.4 Funktionsumfang

Der nötige Funktionsumfang, welche eine Renderengine, die versucht fotorealistisches Rendering in einer Compositing-Applikation zu implementieren, haben sollte, definiert sich zu einem guten Teil durch ihre Anwendungsgebiete. Soll allerdings ein derartiger Renderer jenen einer 3D-Applikation ersetzen, dann benötigt er auch über die gleichen Funktionen. Sind Limitationen zu erwarten, führt dies innerhalb eines professionellen Anwendungsgebietes zu einer Ablehnung und es kommt folglich nicht zu einer Anwendung.

Weiters ist es wichtig, dass die Funktionen jener der Muttersoftware entsprechen. So soll es jemanden möglich sein, der die ursprüngliche Software bedienen kann, auch die auf eine andere Anwendung übertragene Version handhaben zu können. Folglich leitet sich der Funktionsumfang von den Anforderungen an den Renderer in der Muttersoftware ab.

Außerdem ist dabei wichtig, dass die Parameter jenen der ursprünglichen Software entsprechen. Die größte Lernplattform für Softwarepakete stellt das Internet dar. Werden Informationen in Form von Tutorials und so weiter zur Verfügung gestellt, ist eine Übertragbarkeit auf Implementierungen jener Software in anderen Applikationen wünschenswert.

Weitere Wichtigkeit stellt der Funktionsumfang der Austauschbarkeit dar. Große Unternehmen arbeiten intensiv an der Ermöglichung des fehlerfreien Austausches von Szenenbeschreibungen und auch Shadern. OSL und USD sind definitiv ein Schritt in die richtige Richtung, auch wenn noch zu wenige Applikationen den vollen Funktionsumfang jener Erweiterungen unterstützen. Kontinuität durch alle Softwarepakete in allen Departments ist von enormer Wichtigkeit, wenn es um die kollektive Erstellung eines Shots geht.

Deep-Rendering stellt zusätzlich eine notwendige Funktion dar. Deep hat sich zum Standard bei der Kombination mehrerer Renderings entwickelt, da kein Rendering mit Holdouts mehr von Nöten ist. Eine Unterstützung von Deep wäre deshalb gerade bei der Implementierung in eine Compositing-Applikation wichtig.

Der Trend im Rendering geht definitiv in Richtung Echtzeitrendering. Solange allerdings Einschränkungen beziehungsweise Unterschiede in der Qualität vorhanden sind, ist eine Verwendung in einer Compositing-Applikation nicht relevant, da im Compositing stets die maximale Qualität von Interesse ist. Solange es nicht garantiert ist, dass mit einem anderen Renderer die gleiche Qualität geliefert werden kann, ist ein Umstieg weder kurz- noch langfristig zu rechtfertigen.

5.5 Anwendungsgebiete

Die Anwendungsgebiete für eine auf Fotorealismus ausgerichtete Renderengine innerhalb einer Compositing-Applikation lassen sich schwierig in einem allgemeinen Kontext definieren. Folglich ist ein weiteres Aufbrechen auf die Verwendung in einem großen und einem kleinen Studio notwendig.

Anwendung kann ein Renderer in einem großen VFX-Studio vor allem im Bereich des Matte Painting aufweisen. Für jene gestaltet sich der Weg im Regelfall immer über den 3D-Raum in Nuke. Häufig kommen lediglich Projektionen von Paintings zum Einsatz, doch durch eine Erweiterung der Funktionen in eine derartige Richtung würden sich zusätzliche Möglichkeiten für Artists im DMP-Bereich ergeben.

Verschiedenste Studios, darunter ILM, beschäftigen ein Generalists-Department. Ziel dieses Departments ist es vornehmlich komplexe Environments mit allen zur Verfügung stehenden Mitteln in der gegebenen Zeit umzusetzen. Diese verwenden oft eine Vielzahl an Softwarepaketen, welche in der herkömmlichen, starren Pipeline der Studios keinen Platz hätten. Da hier Problemlösungen abseits der gängigen Konvention gesucht werden und die Artists oft sowohl im Umgang mit Renderengines als auch Compositing-Applikationen geschult sind, kann die Verwendung hier ebenfalls stattfinden.

In kleinen Studios kann eine derartige Implementierung in einem ähnlichen Anwendungsbereich zum Einsatz kommen, da sich hier oft die Problemlösung vergleichbar mit jener der Generalists-Departments gestaltet. Ein Artist muss oft mehrere Tasks übernehmen und trotz fehlender Spezialisierung zu einem Endergebnis kommen.

Weiters zeichnet sich als wahrscheinlich, dass ein derartiger Renderer dieselbe Anwendung in einem kleinen Studio finden würde, wie etwa Element 3D. Der Compositing-Artist verfügt damit über die Möglichkeit zusätzliche 3D-Elemente in Nuke zu generieren.

6 Fazit

Im folgenden Kapitel wird hier noch einmal versucht, die in der Einleitung aufgeworfenen Forschungsfragen zusammenfassend zu beantworten.

Das Thema des fotorealistischen Renderings innerhalb von Compositing-Applikationen ist ein vernachlässigtes, welches stark mit der Akzeptanz unter vielen Artists zu kämpfen hat. Es ist nicht immer klar ersichtlich, bei welchen Anwendungsgebieten es eine sinnvolle Alternative darstellen würde.

Alle Compositing-Applikationen, welche über einen 3D-Raum verfügen, verfügen auch über einen Renderer. Dieser basiert im Regelfall auf der Verwendung der Grafikpipeline und Rasterization und bietet eine äußerst effiziente aber auch beschränkte Methode zur Berechnung des zweidimensionalen Bildes. Der Raytrace-Algorithmus im Gegensatz bietet aufgrund seines Funktionsprinzips eine weit universellere Methode zur Berechnung von Bildinhalten. Jenes Funktionsprinzip erlaubt es, dass das Endergebnis über Gegebenheiten wie etwa korrekte Reflektionen via verschiedener Reflektionsmodellen (BRDFs), weiche Schattenkanten, indirektes Licht, korrekte Brechungen und einen generell weit mehr korrekten Lichttransport verfügt. Durch diese zusätzlichen Funktionen des Raytracers wird es möglich, dass Renderings in einem fotorealistischen Kontext Anwendung finden.

Der nötige Funktionsumfang, über welchen eine Renderengine in einer Compositing-Applikation verfügen soll, steht in einem direkten Zusammenhang mit den geplanten Anwendungsgebieten. Grundsätzlich gilt es festzuhalten, dass wenn der Renderer eine 3D-Applikation ersetzen soll, so muss er über einen vergleichbaren Funktionsumfang verfügen. Eine Wichtigkeit stellt dar, dass die Bedienung der Implementierung stark angelehnt von der Implementierung in die ursprüngliche Software ist. So ist es garantiert, dass Vorwissen, die Dokumentation und Tutorials übertragbar sind.

Die Anwendungsgebiete, welche sich durch die Fähigkeit des fotorealistischen Renderings in einer Compositing-Applikation ableiten lassen, stehen in einer Abhängigkeit vom Einsatzgebiet. Umfasst dieses Einsatzgebiet ein großes VFX-Studio, bei welchem die einzelnen Aufgaben in stark spezialisierten Departments aufgeteilt sind, dann ist eines der Anwendungsgebiete im Bereich der Matte Painting Artists und eines im Bereich der Generalists. Kommt eine derartige Erweiterung des Funktionsumfanges bei kleineren Produktionen zur Verwendung,

dann ist dessen Anwendung eher darin gegeben, als dass der Compositing Artist mit der Möglichkeit ausgestattet wird, dass er selbst weitere, kleinere 3D-Elemente selbst generieren kann.

Die Implementierung einer auf Fotorealismus spezialisierten Renderengine erfragt die letzte Forschungsfrage. Hierzu gibt es grundsätzlich zwei verschiedene Methoden, welche Anwendung finden können. Die erste befasst sich mit einer direkten, nativen Implementierung des Source-Codes der Renderengine in die Compositing-Applikation. Die andere mit einer Ansteuerung des Renderers via dessen API. Beide Methoden verfügen über Vor- und Nachteile. Jene Methode, welche eine Ansteuerung der Renderengine mit Hilfe dessen API bewerkstelligt, ist allerdings unter kommerziellen Produkten die am weitesten Verbreitete, die geläufigste. Aus diesem Grund habe auch ich mich für eine Implementierung anhand dieser Prämisse entschieden. Ein Prototyp eines lauffähigen Plug-ins stellt eines der Endergebnisse der Arbeit dar.

Die Frage, ob interdisziplinäres Arbeiten mit Vorteilen in der Qualität und Effizienz eines Workflows einhergehen kann, kann auch diese Arbeit nur teilweise klären, da eine Generalisierung nicht möglich ist. Es gibt definitiv Punkte, wo ein Einsatz durchaus positiv zu bewerten ist, ein allgemeiner Fall lässt sich jedoch nur schwierig definieren.

Literaturverzeichnis

Alembic (2018). Introduction. Zugriff am 25.08.2018 unter http://www.alembic.io/

appleseed (2018). About. Zugriff am 28.08.2018 unter https://appleseedhq.net/about.html

Ashikhmin, M., Shirley, P. (2000). An Anisotropic Phong BRDF Model. University of Utah

Blender Developers (2017). Improve Cycles Standalone. Zugriff am 19.08.2018 unter https://developer.blender.org/T38279

Blender Foundation (2016). Tangent Animation: Making Ozzy with Blender. Zugriff am 04.09.2018 unter https://www.youtube.com/watch?v=CqSEtTF8hPE

Blender Foundation (2017). README, dem Cycles Source-Code beigelegt. Zugriff am 05.01.2018 unter https://developer.blender.org/diffusion/C/repository/master/

Blender Foundation (2018a). Cycles Development. Zugriff am 28.08.2018 unter https://www.cycles-renderer.org/development/

Blender Foundation (2018b). Developer Docs. Zugriff am 28.08.2018 unter https://wiki.blender.org/wiki/Source/Render/Cycles

Blender Foundation (2018c). Rendering and Beyond. Zugriff am 28.08.2018 unter https://www.blender.org/features/rendering/

Blender Foundation (2018d). Integrator. Zugriff am 31.08.2018 unter https://docs.blender.org/manual/en/dev/render/cycles/settings/scene/render/integrator.html

Blender Foundation (2018e). Roadmap. Zugriff am 04.09.2018 unter https://wiki.blender.org/wiki/Source/Render/Cycles/Roadmap

Blinn, J. F., Newell, M. E. (1976). Texture and Reflection in Computer Generated Images. University of Utah

Bogner, A., Littig, B., Menz, W. (2014). Interviews mit Experten. Springer VS

Boost (2018). Boost-Homepage. Zugriff am 30.08.2018 unter https://www.boost.org/

Brinkmann, R. (2008). The Art and Science of Digital Compositing. Morgan Kaufmann

Chaos Group (2018a). Light Select | vrayRE_Light_Select. Zugriff am 19.08.2018 unter https://docs.chaosgroup.com/display/VRAY3MAYA/Light+Select+%7C+vrayRE_Light_Select

Chaos Group (2018b). Color Mapping. Zugriff am 27.08.2018 unter https://docs.chaosgroup.com/display/VRAY3MAX/Color+Mapping

Christensen, P. H., Fong, J., Laur, D. M., Batali, D. (2006). Ray Tracing for the Movie ‘Cars’. Pixar Animation Studios

Cook, R. L., Carpenter, L., Catmull, E. (1987). The Reyes Image Rendering Architecture. SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques

Ernesti, J., Kaiser, P. (2017). Python 3. Reihnwerk Verlag

Fry, A. (2015). ACES in VFX - Siggraph 2015 Talk. Zugriff am 27.08.2018 unter https://www.youtube.com/watch?v=vKtF2S7WEv0

GLEW (2017). GLEW-Homepage. Zugriff am 30.08.2018 unter http://glew.sourceforge.net/

Gritz, L. (2017). Open Shading Language 1.9 - Language Specification. Sony Pictures Imageworks Inc.

Gritz, L. (2018). README.md aus dem OSL GitHub-Repository. Zugriff am 19.08.2018 unter https://github.com/imageworks/OpenShadingLanguage/blob/master/README.md

Kajiya, J. T. (1986). The Rendering Equation. SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques

LuxCoreRender (2018). About Us. Zugriff am 28.08.2018 unter https://luxcorerender.org/aboutus/

Marschner, S., Shirley, P. (2016). Fundamentals of Computer Graphics. CRC Press

Museth, K. (2013). VDB: High-Resolution Sparse Volumes with Dynamic Topology. ACM Transactions on Graphics, Vol. 32, No. 3, Article 27

Okun, J. A., Zwerman, S. (2015). The VES Handbook of Visual Effects. Focal Press

OpenColorIO (2018). Introduction. Zugriff am 25.08.2018 unter http://opencolorio.org/introduction.html

OpenGL (2018). OpenGL-Homepage About. Zugriff am 30.08.2018 unter https://www.opengl.org/about/

OpenVDB (2018). About OpenVDB. Zugriff am 25.08.2018 unter http://www.openvdb.org/about/

Pharr, M., Jakon, W., Humphreys, G. (2017). Physically Based Rendering. Morgan Kaufmann

Phong, P. T. (1975). Illumination for Computer Generated Pictures. University of Utah

Pixar RenderMan (2018). About RenderMan Pro Server. Zugriff am 19.08.2018 unter https://renderman.pixar.com/resources/RenderMan_20/intro.html

Pixar. (2018). Introduction to USD. Zugriff am 19.08.2018 unter https://graphics.pixar.com/usd/docs/index.html

prman-pixar (2018). RenderMan for Blende - GitHub Repository. Zugriff am 28.08.2018 unter https://github.com/prman-pixar/RenderManForBlender

Psyop (2018). README.md aus dem Cryptomatte GitHub-Repository. Zugriff am 19.08.2018 unter https://github.com/Psyop/Cryptomatte/blob/master/README.md

Python Software Foundation (2018). Python 2.7.15 - suprocess - Subprocess Management. Zugriff am 02.09.2018 unter https://docs.python.org/2/library/subprocess.html

Seymour, M. (2013). The state of rendering - part 2. Zugriff am 08.08.2018 unter https://www.fxguide.com/featured/the-state-of-rendering-part-2/

Seymour, M. (2014). The Art of Deep Compositing. Zugriff am 19.08.2018 unter https://www.fxguide.com/featured/the-art-of-deep-compositing/

Siddi, F. (2017). Visual Effects for The Man in the High Castle. Zugriff am 28.08.2018 under https://www.blender.org/user-stories/visual-effects-for-the-man-in-the-high-castle/

SideFX (2018). VEX language reference. Zugriff am 19.08.2018 unter http://www.sidefx.com/docs/houdini/vex/lang.html

Snyder, J. M., Barr, A. H. (1987). Ray Tracing Complex Models Containing Surface Tessellations. California Institute of Techology

Sobotka, T. (2018). README.md aus dem Filmic Blender GitHub-Repository. Zugriff am 27.08.2018 unter https://github.com/sobotka/filmic-blender

Sony Pictures Imageworks (2018). Opensource. Zugriff am 19.08.2018 unter http://opensource.imageworks.com/

The Foundry (2017). NDK Developers Guide. The Foundry

The Foundry (2018a). Relighting a 2D Image Using 3D Lights. Zugriff am 19.08.2018 unter https://learn.foundry.com/nuke/content/comp_environment/3d_compositing/relighting_2d_images.html

The Foundry (2018b). 3D Nodes. Zugriff am 27.08.2018 unter https://learn.foundry.com/nuke/11.1/content/reference_guide/3d_nodes/3d_nodes.html

The Foundry (2018c). Nuke Developers. Zugriff am 28.08.2018 unter https://www.foundry.com/products/nuke/developers

van Lommel, B., Blender Foundation (2018). Source/Render/Cycles/BVH. Zugriff am 05.08.2018 unter https://wiki.blender.org/wiki/Source/Render/Cycles/BVH

Veldhuizen, B. (2018a). "Next Gen" - Blender Production by Tangent Animation soon on Netflix!. Zugriff am 28.08.2018 unter https://www.blendernation.com/2018/08/20/next-gen-blender-production-by-tangent-animation-soon-on-netflix/

Veldhuizen, B. (2018b). Brecht Van Lommel joins the Blender Institute as full time Blender and Cycles developer. Zugriff am 04.09.2018 unter https://www.blendernation.com/2018/07/06/brecht-van-lommel-joins-the-blender-institute-as-full-time-blender-and-cycles-developer/

VES Technology Committee (2012). Cinematic Color. A VES Technology Committee White Paper

Whitted, T. (1980). An Improved Illumination Model for Shaded Display. Communications of the ACM

Abbildungsverzeichnis

Anhang

A. Quellcode Plug-in

```
1         #---Description---
2         #
3         #Project: Cycles for Nuke - cy_Render Node
4         #Author: Oliver Rautner
5         #Date: August 2018
6         #Script Version: 0.1
7         #Nuke Version: 11.0v4
8         #Python Version: 2.7
9         #Description: script for exporting nuke scene description to
cycles xml-api and executing the rendering command of cycles
10         #
11         #---Description End----
12
13
14
15
16         #Importing python libraries
17
18         import os, sys
19         import subprocess
20         from xml.dom import minidom
21
22
23         #some Variables
24
25         thisNode = nuke.thisNode()
26
27
28
29         #path generation
30         path = nuke.root().knob('name').value()
31
32         #remove .nk-extension
33         x=3
34         while x>0:
35             path = path[:-1]
36             x = x-1
37
38         path = path.split("/")
39
40         #get name of nuke script
41         scriptName = path[-1]
42
43         del path[-1]
```

```
44
45          #path of nukescript
46          path = "/".join(path)
47
48
49          #some functions
50
51          #adopt transformationmatrix nuke to cycles
52          #function for inverting y axis in a vectos
53
54          def invertY(l):
55              for indx, val in enumerate(l):
56                  if(indx % 3 == 1):
57                      val = val*-1
58                      l.pop(indx)
59                      l.insert(indx, val)
60                      return l
61
62
63          #adopt rotation... does not work
64
65          def rotateX(l):
66              for indx, val in enumerate(l):
67                  if(indx % 3 == 0):
68                      val = val+180
69                      l.pop(indx)
70                      l.insert(indx, val)
71              return l
72
73
74          #layouting function for xml syntax
75
76          def groupVectors(lst, n):
77              return zip(*[lst[i::n] for i in range(n)])
78
79
80
81
82          #get Infos from Nuke
83
84
85          #get render resolution
86
87          resolution = nuke.toNode("getResolution")
88          resolutionX = str(resolution.width())
89          resolutionY = str(resolution.height())
90
91
92          #get render settings
93
94          samples = thisNode.knob("samples").getValue()
95
```

```
96        #get Camera Transform
97
98        camInput = thisNode.input(1)
99
100        camTranslate = camInput.knob("translate").getValue()
101        camTranslate = ' '.join(map(str,invertY(camTranslate)))
102
103        camScale = camInput.knob("scaling").getValue()
104        camScale = ' '.join(map(str,camScale))
105
106        camRotate = camInput.knob("rotate").getValue()
107        camRotate = ' '.join(map(str,rotateX(camRotate)))
108
109
110        #get background Info
111
112        envColor = thisNode.knob("envColor").getValue()
113        envColor = ' '.join(map(str,envColor))
114
115        envIntensity =
str(thisNode.knob("envIntensity").getValue())
116
117
118
119        #get Light Input
120
121        lightInput = thisNode.input(3)
122
123        #get lightInfo
124
125        if lightInput.Class() == "cy_Light":
126            lightConnected = True
127
128            lightType = lightInput.knob("lightType").getValue()
129            lightColor = lightInput.knob("color").getValue()
130            lightColor = ' '.join(map(str,lightColor))
131            lightIntensity =
str(lightInput.knob("intensity").getValue())
132            lightSize = str(lightInput.knob("size").getValue())
133            lightTranslate =
lightInput.knob("translate").getValue()
134            lightTranslate = '
'.join(map(str,invertY(lightTranslate)))
135
136            lightScale = lightInput.knob("scaling").getValue()
137            lightScale = ' '.join(map(str,lightScale))
138
139            lightRotate = lightInput.knob("rotate").getValue()
140            lightRotate = ' '.join(map(str,rotateX(lightRotate)))
141
142        else:
143            pass
144
```

```
145
146
147         #get shader info // mix shader
148
149         diffuseColor = thisNode.knob("diffuseColor").getValue()
150         diffuseColor = ' '.join(map(str,diffuseColor))
151
152         diffuseRoughness =
str(thisNode.knob("diffuseRoughness").getValue())
153
154
155         glossyColor = thisNode.knob("glossyColor").getValue()
156         glossyColor = ' '.join(map(str,glossyColor))
157
158         glossyRoughness =
str(thisNode.knob("glossyRoughness").getValue())
159
160
161         shaderMix = str(thisNode.knob("shaderMix").getValue())
162
163
164
165         #build Geo from nuke scene description for cycles xml api
via PythonGeo-Node
166
167         #P - vertex position
168         pyGeo = nuke.toNode("PythonGeo1")
169         gObj = pyGeo['geo'].getGeometry()[0]
170         gObj = list(gObj.points())
171
172         gObj = groupVectors(gObj,3)
173
174         gObjList = []
175
176         for c in gObj:
177             c = invertY(list(c))
178             c = ' '.join(map(str, c))
179             gObjList.append(c)
180
181
182         #gObj = ' '.join(l + ' ' * (n % 3 == 2) for n, l in
enumerate(map(str, gObj)))
183         gObj = '  '.join(gObjList)
184
185
186
187         #verts
188         verts = pyGeo['geo'].getGeometry()[0]
189         verts = verts.primitives()
190         listVerts = []
191
192
193
```

```
194         for c in verts:
195             listVerts.extend(c)
196             listVerts.extend("x")
197 
198         listVerts = ' '.join(map(str,listVerts))
199 
200         listVerts = listVerts.replace("x", "")
201 
202 
203 
204         #nverts
205 
206         listNVerts = []
207 
208         for c in verts:
209             nVerts = str(len(c))
210             listNVerts.extend(nVerts)
211 
212         listNVerts = ' '.join(listNVerts)
213 
214 
215 
216 
217         #---gen XML---
218 
219         root = minidom.Document()
220 
221         xml = root.createElement("cycles")
222         root.appendChild(xml)
223 
224 
225 
226         #---camera XML begin---
227 
228         cameraChild = root.createElement('camera')
229         cameraChild.setAttribute("width",resolutionX)
230         cameraChild.setAttribute("height",resolutionY)
231         xml.appendChild(cameraChild)
232 
233 
234         transformChild = root.createElement("transform")
235         transformChild.setAttribute("translate",camTranslate)
236         transformChild.setAttribute("rotate", camRotate + " 0")
237         transformChild.setAttribute("scale", camScale)
238         xml.appendChild(transformChild)
239 
240 
241         cameraChild = root.createElement("camera")
242         cameraChild.setAttribute("type","perspective")
243         transformChild.appendChild(cameraChild)
244 
245         #---camera XML end---
```

```
246
247
248
249         #---point light XML begin---
250
251         shader = root.createElement('shader')
252         shader.setAttribute("name","point_shader")
253         xml.appendChild(shader)
254
255
256         emission = root.createElement("emission")
257         emission.setAttribute("name","emission")
258         emission.setAttribute("color",lightColor)
259         emission.setAttribute("strength",lightIntensity)
260         shader.appendChild(emission)
261
262
263         connect = root.createElement("connect")
264         connect.setAttribute("from","emission emission")
265         connect.setAttribute("to","output surface")
266         shader.appendChild(connect)
267
268
269         state = root.createElement("state")
270         state.setAttribute("shader","point_shader")
271         xml.appendChild(state)
272
273
274         light = root.createElement("light")
275         light.setAttribute("type","point")
276         light.setAttribute("co",lightTranslate)
277         light.setAttribute("size",lightSize)
278         state.appendChild(light)
279
280         #---light XML end---
281
282
283
284         #---background XML begin----
285
286         if thisNode.knob("environmentLight").getValue() == 1.0:
287
288             background = root.createElement("background")
289             xml.appendChild(background)
290
291
292             background1 = root.createElement("background")
293             background1.setAttribute("name", "bg")
294             background1.setAttribute("strength", envIntensity)
295             background1.setAttribute("color", envColor)
296             background.appendChild(background1)
297
```

```
298
299             connect = root.createElement("connect")
300             connect.setAttribute("from","bg background")
301             connect.setAttribute("to","output surface")
302             background.appendChild(connect)
303
304         else:
305             pass
306
307         #---background XML end----
308
309
310
311         #---mesh XML begin----
312
313         shader = root.createElement('shader')
314         shader.setAttribute("name","mesh")
315         xml.appendChild(shader)
316
317
318         diffuse1 = root.createElement("diffuse_bsdf")
319         diffuse1.setAttribute("name","diffuse1")
320         diffuse1.setAttribute("color",diffuseColor)
321         diffuse1.setAttribute("roughness",diffuseRoughness)
322         shader.appendChild(diffuse1)
323
324
325         glossy1 = root.createElement("glossy_bsdf")
326         glossy1.setAttribute("name","glossy1")
327         glossy1.setAttribute("roughness",glossyRoughness)
328         glossy1.setAttribute("color",glossyColor)
329         shader.appendChild(glossy1)
330
331         mix1 = root.createElement("mix_closure")
332         mix1.setAttribute("name","mix")
333         mix1.setAttribute("fac",shaderMix)
334         shader.appendChild(mix1)
335
336
337         #connect0 = root.createElement("connect")
338         #connect0.setAttribute("from","checker fac")
339         #connect0.setAttribute("to","mix fac")
340         #shader.appendChild(connect0)
341
342
343         connect1 = root.createElement("connect")
344         connect1.setAttribute("from","diffuse1 bsdf")
345         connect1.setAttribute("to","mix closure1")
346         shader.appendChild(connect1)
347
348
349         connect2 = root.createElement("connect")
```

```
350         connect2.setAttribute("from","glossy1 bsdf")
351         connect2.setAttribute("to","mix closure2")
352         shader.appendChild(connect2)
353 
354 
355         connect3 = root.createElement("connect")
356         connect3.setAttribute("from","mix closure")
357         connect3.setAttribute("to","output surface")
358         shader.appendChild(connect3)
359 
360 
361 
362         state = root.createElement("state")
363         state.setAttribute("shader","mesh")
364         if thisNode.knob("shadingType").getValue() == 1.0:
365             state.setAttribute("interpolation","smooth")
366         else:
367             pass
368         xml.appendChild(state)
369 
370 
371 
372         mesh = root.createElement("mesh")
373         mesh.setAttribute("P",gObj)
374         mesh.setAttribute("verts",listVerts)
375         mesh.setAttribute("nverts",listNVerts)
376         state.appendChild(mesh)
377 
378         #---mesh XML end----
379 
380 
381         #format xml for writing
382 
383         xml_str = root.toprettyxml(indent="\t")
384 
385 
386 
387         #gen folders for xml and image file
388 
389         cyPath = path + "/" + "cycles"
390 
391         try:
392             os.mkdir(cyPath)
393 
394         except:
395             pass
396 
397         cyProPath = cyPath + "/" + scriptName
398 
399         try:
400             os.mkdir(cyProPath)
401         except:
```

```
402             pass
403 
404 
405         #gen pathes for rendering-command
406 
407         imagePath = str(cyProPath + "/" + scriptName + ".exr")
408 
409         xmlPath = str(cyProPath + "/" + scriptName + ".xml")
410 
411 
412 
413         #write xml file to disc
414 
415         with open(xmlPath, "w") as f:
416                 f.write(xml_str)
417 
418 
419 
420         #get path of cycles executable
421 
422         execPath = thisNode.knob("execPath").getValue()
423 
424 
425 
426         #---Rendering---
427 
428 
429         #rendering command when rendering to framebuffer
430 
431         if thisNode.knob("renderTo").getValue() == 0.0 :
432             subprocess.call("%s --height %s --width %s --samples %s
%s" % (execPath+"cycles", str(resolutionY), str(resolutionX),
str(samples), xmlPath), shell=True)
433 
434 
435         #rendering command when rendering to disk/nuke
436 
437         elif thisNode.knob("renderTo").getValue() == 1.0 :
438             subprocess.call("%s --background --output %s --height
%s --width %s --samples %s %s" % (execPath+"cycles", imagePath,
str(resolutionY), str(resolutionX), str(samples), xmlPath),
shell=True)
439             nuke.toNode("Read1").knob("file").setValue(imagePath)
440             nuke.toNode("Read1").knob("reload").execute()
441 
442         #when only writing the xml file to disc no rendering comman
is executed
443 
444         else:
445             pass
446 
447 
448 
```

B. Transkript Interview Valentin Struklec

I: Also grundsätzlich wäre es am Anfang einmal interessant, dass du Informationen über dich und deinen Werdegang gibst, damit man die Expertise in einen Kontekt setzen kann.

B: Also ich habe das Studium an der FH Joanneum abgeschlossen, das war damals ... ähhm ... sag ma mal sehr grafiklastiges Studium mit hohem Motion Graphics Anteilen und da bin ich hald dann so in den Bereich reingerutscht, hab dann mit After Effects viel gearbeitet. Bin dann in Wien bei einer Filmproduktion gelandet, in der Werbung, und hab dort viel After Effects gemacht und war aber eigentlich in der Filmabteilung beschäftigt und bin dadurch dann immer wieder, weil einfach der Bedarf an qualifizierten Leuten da war, immer wieder für die Visual Effects irgendwie verantwortlich gewesen. Dann bei größeren Produktionen und dann hat sich das ergeben, dass ich bei einer Produktion, ähhm, in die Rolle des Visual Effects Supervisors schlüpfen hab müssen. Eh ganz am Anfang eigentlich, von meiner Karriere und das war hald, ja, ähm wichtiger Schritt sag ich amal. Und dann hab ich gemerkt, dass das funktioniert hat, aber dass ich technisch nu in der Praxis viel lernen muss und bin dann zu Friendly Fire gegangen, die damals für mich gearbeitet haben, als ich als Visual Effects Supervisor und sie als ausführendes Studio. Und hab dort dann zwei Jahre gearbeitet, hab dann... die nächste Station war London bei MPC. Ähm. Wo ich an X-Men gearbeitet hab, für zwei Monate war das glaub ich die erste Arbeitserfahrung dort. War hald spannend, weils hald andere Pipeline, größere Strukturen und soweiter, gö, und andere Aufteilung. Ähm. Das hat mich ziemlich fasziniert. Also jetzt einerseits von der kreativen Herangehensweise, weil hald die Leute darauf geschult sind, dass die wirklichen Dinge sehen und Bilder gestalten und diese ganzen Dinger und andererseits hald auf einer, von einer Managementseite her, wie man das schafft, dass man 200 Compositor irgendwie gleichzeitig mit Daten beliefert, mit Feedback beliefert, mit... also dass das alles funktioniert, dass die ganze Maschinerie, gö, ähm. Genau, nach MPC war das dann so, dass ich immer wieder nach Österreich zurückgekommen bin, ähm, aber auch nach London gewechselt bin für so kleine Sachen. Also der Lebensmittelpunkt war Österreich, aber arbeitsmäßig hats mich hald immer wieder dorthin gezogen. und eine längere Show war dann Ridley Scotts Prometheus, wo ich dann länger dort war. Also da hab ich dann gesagt, okay, jetzt bleib ich ein paar Monate dort, das zahlt sich aus, das ist einfach ähm ähm herausragendes Ding, des ist jetzt ned so wie wenn man also den neuen “Nachts im Museum 2” macht, also des ist hald echt, das könnte Filmgeschichte sein. Haben wir uns damals gedacht. Ich mein es is eh, irgendwie ist es geblieben, es hat hald viel Diskussion ausgelöst.

I: Ich finde es einen guten Film, also!

B: Du findest as was?

I: Einen guten!

B: Ja, ich hab ihn auch gut gefunden, und die Erwartungen waren riesig, vom Team, gö, keiner hat gewusst, was auf uns zukommt. und Rezensionen ist er zerrissen worden, was hald mühsam wa, dann irgendwie. Aber es immer wieder, also vor allem in Visual Effects Kreisen is es, kann man den Film nennen und kann sagen ich hab da mitgearbeitet, weil die meisten Leut sagen, ja der hat schön ausgeschaut oder so, von den Bildern. Genau, ähm. London war für mich immer so zum Leben eine mühsame Stadt, weil, weil ich das Britische immer so ähm bissi komisch gefunden hab. Und damals haben immer alle gesagt, ja, Vancouver ist super. ähm. Dann hab ich mir, dann hab ich mich dort beworben. Bin dort auch zu MPC gegangen und hab dort insgesamt glaub ich auch zwei Jahre gearbeitet, eben in Vancouver. Wahrscheinlich ein bissi kürzer (unv.). Ähm. Und das war eine andere Struktur; vor allem wars hald in Vancouver so, dass man das gemerkt hat, dass durch diese Tax Incentives, die hald damals grad gegriffen haben, alle Firmen dorthin gezogen sind. Und MPC Vancouver, das hat es ja davor schon länger gegeben, allerdings sind die von einem small studio auf ein big studio gewachsen, weil hald gerade die finanzielle Lage so gegriffen hat, gö, dass die raufskaliert haben. Und du hast hald einerseits noch den vibe gehabt von dem small, von den kleinen Studios, das heißt alle haben irgendwie zumindest versucht sich gegenseitig zu kennen und zu grüßen. Also es is hald ned so anonym gewesen, wie MPC in London, wo du hald teilweise, da waren es hald drei Häuser nebeneinander. Und du bist hald wirklich durch so einen Hausdurchbruch bist ins Zweite marschiert und dann war da das 3D Department, sieben Stockwerke rauf, runter und dann hald ins dritte Haus. Und plötzlich is hald dann noch ein Haus dazu gekommen und so. Also du hast nie die Chance gehabt, oder hättest dort die Chance gehabt dort alle kennen zu lernen, weils hald einfach ein paar 100 Leute waren. Und in Vancouver waren es glaub ich insgesamt zu dem Zeitpunkt 150 bis 200, das skaliert so schnell, dass man des gar nie so genau sagen kann, gö. Ähm aber überschaubar und drei Stockwerke und ähm ja. Und vor allem hat man ja damals schon ein paar Leute gekannt und das war dann relativ familiär im Vergleich zu London. Genau, gleiche Arbeitsstruktur, aber eigentlich ähm und dann ist mir hald irgendwie so geschossen, dass es bei uns eh so is, dass wir im

Prinzip Kontinente wechseln können und ähm also es ist egal, solange du Englisch kannst und solange du die Software kannst, setzt dich hin und arbeitest meistens schon am zweiten, dritten Tag schon effektiv an Dingen. Genau, ähm und nach MPC Vancouver, da wars dann so, dass ich selber irgendwie beschlossen hab, dass ich gerne wieder irgendwie nach Österreich kommen würde, meine Freundin war immer in Wien, die hat das nie aufgegeben, obwohl öfters die Diskussion war und hab dann den Job an der Filmakademie gesehen, hab mich dort beworben und hab dann eigentlich alles abgebrochen in Vancouver um daher zu kommen. Ich hab aber gleichzeitig gemerkt, dass ich auch schaun muss, dass ich kreativ noch weiter arbeite und deswegen hald dann so langsam Studio aufgebaut und das is immer größer geworden und jetzt haben wir da ein paar Leute sitzen und ein paar Räume und machen eigentlich ganz coole Sachen, ja. Muss man sagen. Und genau jetzt... bin ich da wo wir gerade sitzen.

I: Nuke hat ja grundsätzlich seinen eigenen Raytracer, seit Version 10 glaube ich eben, mit dem RayRender. Der hat sehr einen begrenzten Funktionsumfang. Ich habe mich öfters gefragt, ob der überhaupt für irgendetwas verwendbar ist und ob der in Produktionen verwendet werden kann.

B: Mhhm. Ja. Also grundsätzlich ist das so, wie ich das kennengelernt habe; in größeren Häusern sind die Leute technisch sehr scheu. Also. Ich hab mit Matte Painters geredet, die haben 10 Jahre Erfahrung und machen die schönsten Sachen und dann frag ich sie: „Du wie machsten du das in Photoshop, dass du die Ebene dahinter klonst und was auch immer?“ Und die Antwort darauf war: „Ich weiß ned, ich verwende immer nur die zwei Knöpfe eigentlich, gö.“ Also es ist wirklich so, dass die, dass man hald seine eigenen Methoden findet wie man zu dem Ergebnis kommt. Und wie ich das kennengelernt habe, gibts auch sehr kreative Leute, die sich überhaupt ned für das Technische interessieren und die werden hald dann aufgefangen, dass hald in dem Raum hald irgendwo ein Techniker, der diese Fragen beantwortet und so weiter. Und dann gibts oft die Ausnahme. Im Artist Bereich ist eigentlich, dass jemand kreativ gut ist, technisch gut ist und noch ähm bissi einen Kopf für das Management hat. Also das ist so der, die Eier legende Wollmilchsau unter Artist. Die versucht man zu finden. Jemand der eine Planung hat, der weiß, okay, er muss sich das so einteilen. ähm der kreativ ein gutes Auge hat under hald technisch versteht, wie hald die Dinge funktionieren, so dass er sichs hald zu seinem kreativen Ergebnis kommt. Und der raytracer ist ja grundsätzlich ned so alt. Der ist ja jetzt so zwei Versionen dabei oder so?

I: Eben seit 10 ist der, mit 10 ist der gekommen. Glaub, bis 11 war er in der Beta oder sowas

B: Und wo wir den verwendet haben waren Reflektionen. Wenn du jetzt zum Beispiel einen Matchmove hast und du willst hald eine Reflektion über ein Fenster oder so erzielen. Und das soll hald richtig mitwandern. Dann kannst den, du kannst das Faken mit einer zweiten Kamera, die irgendwie das Gleiche machen soll, aber besser wäre es hald, also die leichtere Variante wäre es wenns mit dem Raytracer direkt dann funktioniert. Dass du hald das 3D-Setup hat und dann schaut die Reflektion hald dann richtig aus, eigentlich, ja.

I: Also ich glaub, so wie der von The Foundry beworben worden ist, ist der eh grundsätzlich gedacht gewesen als Erweiterung von Reflektionen vom ScanlineRenderer. Weil recht viel mehr kann er eigentlich eh ned.

B: Ja, der kann das eben ned, die Reflektionen, genau.

I: Ich habe mir die Releasenotes der letzten Version durchgeschaut und da stehen hald als Known Issues ähm drin, dass eben die Soft Shadows fehlen, Refractions fehlen, Transparenzen fehlen. Also ich glaub schon, dass des geplant ist, dass sie es implementieren, aber...

B: Ja, das ist die Frage. ich habe das Gefühl. Also ich hab mir dann auch gedacht, das könnte jetzt zum ersten Mal ein Renderer sein, der den ScanlineRenderer ablöst. Aber es ist von den Artists ned angenommen worden. Das ist meine Wahrnehmung von den ganzen. Also es is nu immer so, ScanlineRenderer ist der Standardrenderer und der ist hald bei jedem für irgendwie alles in Verwendung. Und wenn du irgendwie dann den Fall hast, du brauchst eine Reflektion und du brauchst irgendwas, dann verwendest hald für den einen Usecase den Raytracer hald, gö. Und sonst, ich meine, es ist hald wenn man des hald jetzt so aufsakliert, ähm, die Shotproduktion und man hat den Fall das man Renderfarm hat und einen Submitter und irgendwie ein Rendermanagmenttool und so, dann wird einem schnell klar, dass es irgendwie so viele Fehlerquellen gibt in dem Prozess, die man teilweise gar ned technisch eruieren kann, sondern teilweise is es dann ein Gefühl. Also ich hab das bei MPC zum Beispiel, die erfahrenen Artist, die das hald 15 Jahre gemacht haben, die haben gesagt, nein ich verwende den OFlow nur mit den und ich verwende einen TimeOffset überhaupt ned in meinem Skript und in

der Read-Node mach ich gar nix, weil dann crasht alles und so, gö. Und des ist jetzt gar ned so, dass man das irgendwie erforscht hätte, warum diese diese Fehler produzieren, produziert werden, sondern eher so, dass man da eher so nach einem Gefühl geht. Und man versucht dann immer die Risikofaktoren zu vermeiden. Und der Raytracer, ein neuer Renderer, ist hald so ein Risikofaktor. Das heißt, man wird wahrscheinlich immer versuchen das mit seinen Wegen wo man weiß, dass das funktioniert zerst umzusetzen. Wo man hald mehr praktische Erfahrung damit hat, genau. so hätte ich das jetzt gesehen. Und es wird sicher Artist geben, na, ich mach jetzt alles mit dem Raytracer und das ist macht überhaupt keinen Sinn mit Scanline zu arbeiten und so. Und wir haben bis jetzt auch in der Firma noch niemanden gehabt, der alle geraytraced hätte. Ja. So hätte ich jetzt den aktuellen Zustand gesehen, von dem Ganzen.

I: Mittlerweile gibt es ja wirklich viele Drittanbieter, die was Plugin für Nuke entwickeln, also V-Ray, Octane, RenderMan und Maxwell gibts glaub ich, so was ich jetzt gefunden habe. Hast du da schon jemals was verwendet?

B: Angeschaut. Hab ichs mir amal. Also V-Ray, das hab ich, das war spannend damals. Da hab ich mir schon gedacht, mhmm das könnte tatsächlich, ja, also es hätte einen Platz in der Pipeline, das könnte ein paar Bereiche abdecken die spannend wären. Also gerade wenn man hald so denkt, environments und matte paintings und diese Bereiche, da haben die Renderer schon was zu suchen, ja. aber es ist, ich mein, ich habe das bei MPC zum Beispiel spannend gefunden, weil ich das hald sonst auch noch nie so erlebt habe, also in Österreich ist die Situation eher so du hast, du versucht dich eher als Generalist zu entwickeln, du machst ein bisserl 3D oder du musst dich zumindest auskennen und du machst deine eigenen Renders, du machst meistens deine eigenen Simulationen und machst dann ein bissi Compositing von deinen eigenen Sachen, gö. Was hald den Nachteil hat, du arbeitest immer mit dir selbst, also die Qualität was dann rauskommt beim Endprodukt ist immer nur die Maximale, was du selber schaffen kannst, gö. Ähm Jetzt ist das klar, so wie ich das beobachtet hab im internationalen Bereich gibts ned viele Leute, die was das alles abdecken können und vor allem ist der Qualitätsanspruch so hoch, dass die Leut hald beginnen sich total auf ihren Bereich spezialisieren. Das heißt ein Matte Painter, der nur seine zwei Knöpfe bedient, der macht sein Leben lang eigentlich nix anderes und entwickelt sich hald kreativ irgendwie weiter, aber technisch ist das meistens so der Bereich, in dem er bleibt, so hab ich das kennengelernt. Also ich hab schon Ausnahmen auch kennengelernt, ich habe jemanden kennengelernt, der hat Lighting gemacht und dann hat er Matte Painting gemacht und wollt dann hald auch immer so aus persönlichen Gründen sich weiter entwickeln. Aber die meisten Leute, die was ich

kennen gelernt habe, die machen eben seit Jahren eben immer das Gleiche, das ist schon die Fließbandarbeit, die wo man hald versuch so gut wie möglich zu werden und dann hald versucht die Karriereleiter hald ein bissi nach oben zu steigen. Also so als Junior zu starten, Midlevel, Senior und dann irgendwann mal Supervision hald. Je nachdem wie gut man hald ist und was man machen will, gö. ähm und...

I: Und bei euch ist auch so spezialisiert, bei vast, oder ist das schon alles noch generalisierter was die Leute arbeiten?

B: Wir sind, wir habens schon eingeteilt in Compositing und 3D, dass eigentlich recht fix, ja. Da gibt es wenige Überschneidungen, aber viel Kommunikation. So habe ich das auch kennengelernt, dass das auch gut funktioniert. Und was man hald, der Vorteil, den man dadurch hat, es kommen zwei kreative Visionen zusammen. Also gö, des is immer, wenn man einen Shot bespricht und gemeinsam an was arbeitet, dann hat der 3Dler eine eigene Vorstellung, der sieht ein Bild vor sich. Also wenn jetzt irgendein Environment ist, dann hat er hald einen hellblauen Himmel, den er sieht und ähm bissi durch den Haze durchkommt und so weiter. Und der Compositor macht dann aus dem noch einmal seine eigene Vision. Das ist hald der Vorteil. Daher hab ich eben das Gefühl, dass wenn man da jetzt als Supervisor jetzt so mitredet in den Prozessen, ist es oft gar ned so gut, wenn man da jetzt zu sehr seine eigenen Visionen aufdrückt, sondern zuerst einmal die Leute an ihrem eigenen, also wenn man gute Artist hat, dann funktioniert das. Das man halt amal abwartet was von denen kommt. Also das ist meiner Meinung nach die gescheiteste Variante und wenn man sieht, also als Supervisor jetzt wann man sieht, dass es ned zusammenkommt, das die beiden Visionen ned mit einander funktionieren oder dass die Qualität dann ned erreicht wird, dann muss man hald nachbessern. Dann muss man hald sagen, na, dass müss ma heller machen, das müss ma dunkler machen, da mehr Bounce Light und solche Dinge, ja. Also das sind hald eigentlich die Fälle wo es nur notwendig ist, dass man eingreift. Ähm. Aber im 3D-Bereich sind wir dann schon eher Generalisten. Also wir haben Leut, die machen Shading, Texturing, Modelling, Rendering, Animation, Simulation, ja, das machen wir schon alles eigentlich, ja. Also da täts auch zu wenige Spezialisierungen geben, glaub ich, in Österreich an Leuten. Im Animationsbereich wahrscheinlich am ehesten, gö, dass ma da jemanden findet, der nur Animation macht und machen will und so. Das ist ähm so wie ich das kennengelernt habe ein eigener, eine eigene, Animatoren haben so eigene Charaktereigenschaften, die man so hald dann kaum findet. Also ich hab immer so das Gefühl, das sind fast Schauspieler, in sich die meisten. Wenn man das hald jetzt so klischeehaft sagen kann. Ja. Genau, also so sind wir ungefähr

eingeteilt. Also ich mein, das ist ja auch bei so einem Renderer jetzt, also wenn man V-Ray jetzt zum Beispiel in Nuke verwenden will, ist einmal die Frage, findet man einen Artist, der einerseits V-Ray kann, weil V-Ray ist ja doch, ähm, ein eigener Renderer, der ned so voll überall vertreten ist, also hat schon seinen Platz, aber ist ned der super Standardrenderer hätte ich gesagt, gö. Das heißt, man muss jemanden kennen, der die ganzen V-Ray-Setting irgendwie intus hat oder lernen will oder sich anschauen will, oder gutes 3D-Allgemeinwissen hat und auf dem aufbauend irgendwie sich das anschauen will. Und von einem Compositor hat man hald eine eigene Erwartung dann eigentlich, gö. Also meine Erwartung von einem Comper wäre jetzt, dass er bildgestalterisch arbeiten kann, dass gute Managementfähigkeiten hat, also dass ist wahrscheinlich wichtig, dass er sagt, okay, ich brauch jetzt drei Stunden bis ich das jetzt einmal vorgerendert habe und dann eine halbe Stunde, bis ich zwei Versionen fertig hab, dann hab ich ein Ergebnis. Und das ist im Compositing unheimlich wichtig, gö. Das du ein Selbstmanagement hast, wo du einfach weißt, wann du Arbeit abgeben kannst, wie lange du brauchst dafür oder wie lange du brauchen kannst, ah, gö. Also Schlimmste was einer Produktion passieren kann ist, wenn die Sachen ned fertig werden. Und als Comper bist du immer der Buffer am Schluss, gö. Also du kriegst immer, ned immer unfertige Sachen, aber meistens unfertige Sachen. Du musst oft die Probleme lösen, die früher entstanden sind in der Pipeline, also wenns im Texturing schon Probleme gegeben hat, dann landen die irgendwann mal bei dir. Und du bist derjenige, der endverantwortlich ist für den Shot ist, also bei dir muss er dann am Ende gut ausschauen. Und wie du das machst, also da gibts hald Tricks. Du kannst einen Lensflare drüber bauen, du kannst was vom Licht her was verändern, dass du es silhouettierst zum Beispiel, dass die Kontraste veränderst und so weiter, eben mit der Farbkorrektur und so. Da muss hald jeder selber damit arbeiten. Aber das ist ja mal der wichtigste Bereich, den was ein Comper erfüllen muss eigentlich. Und wenn er jetzt V-Ray-Kenntnisse haben müsste jetzt zusätzlich, also das ist hald einfach kein essentieller Comp-Bereich, der da jetzt wichtig wäre, wenn mans jetzt so aufgebröselt hätte. ähm. Im Matte Painting gibts so eine Überlagerung. Also so wie ichs kennen gelernt habe, da arbeiten viele Matte Painter hald im Photoshop so lang bis sie mit ihrem Ding zufrieden sind. Machen dann ein 3D-Setup mit der Matchmovecamera die sie eben bekommen. 3D-Setup heißt einfach Cards oder so, grobe Geometrie, auf die drauf projiziert wird, dass die Kamerabewegung hald dann das abdeckt, was im Endeffekt dann eigentlich da sein sollte. Und das passiert in Nuke oft. Also das passiert dann, standardmäßig kenn ich das hald dann so, dass das mit ScanlineRenderer der Motionblur hald dann dort erzeugt wird von irgendwelchen Environments und solchen Sachen. ähm. Und oft ist so, also entweder das 3D-Setup wird dann weitergegeben an den Compositor oder er rendert dann wirklich die Elemente in Nuke raus und kriegst dann vorgerenderte Elemente, wie ein 3D-Render im

Prinzip, wo du hald dann Normals und Vektoren und Positionpass und was auch immer haben kannst. Genau.

I: Und gescheiter, dass du dir V-Ray angeschaut hast und dann ned verwendet hast, ist es daran, weil es unwahrscheinlich ist, dass ein Compositing-Artist Fähigkeiten in V-Ray hat und dass sich das ned auszahlt an zu lernen?

B: Also mich persönlich hätte es interessiert, weil ich hald immer wieder so Matte Paintings hald gerne selber mache und so. Also ich bin ned der beste Matte Painter, aber ich versuche mich immer wieder daran, gö. Aber ich finde die Pipeline eigentlich schon gut so wie sie ist. Oder wie sie hald in den meisten Studios ist, dass der 3Dler für sein Ergebnis endverantwortlich ist, gö? Ähm. Sinn machts eigentlich nur dann, wenn das alles zusammenwächst. Was spannend ist weil, da wird sich, da is sich jeder irgendwie einig, dass diese Softwarepakete irgendwann mal zu einem werden, werden sollten.

I: Ein bissi an was Houdini gerade hinarbeitet. Man kann ja mittlerweile auch Compositing darin machen.

B: Ja, genau. Was ned verwendet wird, wenn Nuke noch immer so der Standardcompositor ist. Und ich hab dann auch das Gefühl, dass ist dann auch produktionstechnisch die Notwendigkeit da, dass ein eigener Compositor, der stressresistent ist und wahrscheinlich auch, weiß ich ned, eine eigene Persönlichkeit mitbringt. Das der da dann nochmal daran sitz und das Ganze dann zu Ende führt. Ja. Also ich kenne wenig 3Dler, die das so wie Compositor jetzt eigentlich fertig machen würden. Dass das Arbeitsergebnis... ja. Also ich glaube, das kommt aus dem Produktionsbereich, dass das erhalten bleiben wird. Vielleicht ist es aber irgendwann mal so, dass die gleiche Software verwendet wird. Also wenn jetzt zum Beispiel sagst, DMP ist jetzt auch schon Nuke, aber die rendern dann schon in Nuke ihre Sachen raus und gebens dann dem nächsten weiter, der wieder in Nuke damit arbeitet. Kann mir schon gut vorstellen, dass das dann irgendwann mal so ist, dass... also das wird ned im Nuke sein, das glaube ich ned, aber... im Houdini wahrscheinlich auch ned. Ähm. Aber es wird ein Softwarepaket geben, in den nächsten 10 Jahre, wo hald alles passiert drinnen eigentlich, das glaub ich schon, ja.

I: Blender hat hald ein wenig den Ansatz, da kann man hald Videoschnitt drinnen machen. Resolve entwickelt sich auch mit Fusion.

B: Genau. Das ist ein gutes Beispiel, dass so zwei ganz unterschiedliche Bereiche zusammengewachsen sind, eigentlich gö. Mit Compositing und Color Grading.

I: Und jetzt auch mit Fairlight und Tonmischung. Aber glaube nicht, dass es der Stabilität des Programms hilft.

B: Na. Das machts eigentlich komplexer. gö. Ja, das ist eigentlich sicher auch ein Grund, dass es je mehr Departments es gibt, desto mehr kann man auf die individuellen Probleme dann auch eingehen. Und Stabilität ist tatsächlich ein Thema.

I: Auf der FMX konnte ich mit dem V-Ray-Repräsentanten reden. Er hat gemeint, dass ihr Plugin weit unter den Erwartungen geblieben ist, was den Verkauf und so angeht. Was weit verbreitet ist, ist Element 3D für After Effects. Liegt das eher daran, dass es Sinn macht für Motion Graphics oder einfach nur daran, dass die Adobe Produkte demokratisierter sind und mehr Gelegenheitsnutzer haben, welche jetzt nicht in ein 3D Programm wechseln?

B: Ich glaube, es liegt jetzt eher daran, dass die komplexeren 3D-Programme oder Compositing-Pakete von der Zielgruppe jetzt nur Randerscheinungen sind. Glaube ich. Gö. Also ich weiß ned, wie viele Leute es jetzt weltweit gibt, die Nuke beherrschen, aber After Effects ist hald einfach so vertreten in dem Bereich und du kommst unkomplizierter und schneller zu Ergebnissen, als wenn du jetzt einen Node-basierten Compositor erlernen müsstest. Und After Effects ist hald einfach der Platzhirsch jetzt. Mein Gefühl ist das Photoshop eigentlich so die Standardsoftware von Adobe war, mit der sie immer ihre Marktdominanz ausgebaut haben. Und der Rest von InDesign und After Effects das ist hald dann mit passiert und die haben hald schon, glaub ich, über Jahre einfach konstant ihre Kunden behalten, also ich kenn jetzt auch Leute im Feature Filmbereich, die machen mit After Effects super Sachen, da siehst gar ned, dass das jetzt keine super teure, super professionelle Software ist, wenn mans jetzt vergleicht mit anderen Paketen. Und da hat das Element 3D jetzt einfach automatisch durch das Paket After Effects viel mehr Leute erreicht, gö. Ja. Der Preis ist jetzt vielleicht auch ein bissi abschreckend. Also After Effects macht jetzt, wenn man als Studio

überlegt, viel mehr Sinn. Weil da hast eine monatliche Miete, die ned hoch ist, gö, kannst schneller skalieren, das ist super. Renderfarm funktioniert auch. Es ist hald einfach kreativ limitierend, so wie ich es kennengelernt habe. Obwohl kannst schon viel machen damit. Ich habe mir das Element 3D nie persönlich angeschaut, da hab ich mir auch immer gedacht, ja, weiß ned, wenn ich die Anforderungen hätte, dass ich was richtig Schönes machen will, dann nehm ich mir ein Maya her und ein V-Ray oder sowas, gö. Also mein Vorurteil fürs Element 3D war dann, du stehst irgendwann mal an wahrscheinlich damit. Also irgendwann mal ist die Situation, dass du hald sagst, okay, das Anti-Aliasing funktioniert ned, oder das Shading oder das Displacement, ich weiß gar ned, obs Displacement gibt oder sowas, das ist die Frage. Aber es gibt immer wieder die Use Cases. Also sag ma, du hast ein Flugzeug, das fliegt irgendwie am Himmel, überm Himmel drüber oder so. Dafür ist es super. Also ein stehendes Objekt, einmal durchbewegt, Motion Blur erzeugen und das wars eigentlich , gö.

I: Vielleicht auch für Werbung, wenn ein Produkt hineingehört.

B: Das kann auch gut funktionieren, ja. Wobei, das ist spannend, weil da hab ich jetzt das Gefühl, da werden jetzt die Game Engines immer mehr, werden zum Einsatz gebracht bei solchen Sachen, gö. Also da wachsen jetzt auch ein paar Sachen zusammen, ja.

I: Betreffend den Funktionsumfang. Was würde Sinn machen, dass jetzt bei einer Software drinnen ist, bei einem Renderer. Also von Renderfunktionen her, dass man sagt, man kann damit arbeiten. Bist du eher der Meinung, dass das quasi, okay, wann ich habe, dann schon, dann will ich alles drinnen machen können oder eher so einfach wie möglich, weil sobald es komplexer wird wechsel ich in ein 3D-Programm.

B: Ja, wenns ein 3D-Programm ersetzen soll, dann muss es alles können, was ein 3D-Programm kann. Also das war auch mein Gefühl vom Elements 3D, dass ich mich gar ned damit anfreunden will, weil ich hald irgendwann mal damit anstehe und dann ärgere ich mich damit, wenn ich merke, das kann das ned was ich will eigentlich, gö. Das heißt, es muss dann alles anbieten eigentlich, ja. Das ist auch mein Gefühl. Also beim V-Ray weiß ich auch ned, ob das im Nuke alles abdecken würde.

I: Es ist schon ziemlich komplex von der Implementation.

B: Ich meine, das wichtigste bei den ganzen Themen ist eigentlich das, dass man eine Lösung findet um Shadingparameter auszutauschen. Das wäre das größte Thema eigentlich. Also bei ILM hab ich das spannend gefunden, die haben eine Open Shading Language, ich glaub ein eigenes Paket entwickelt, und bei denen wars hald echt so, dass wenn du jetzt im Katana oder irgendwo im Clarisse was machst, kannst das im 3D Studio mit V-Ray auch rendern und es sollte eigentlich das Gleiche rauskommen. Und das ist einfach ein riesen Vorteil. Die Renderer haben dann schon, man sagt immer, es ist wurscht, weil es ist nur die Technik dahinter, aber es hat einen riesigen Einfluss auf den Look und Feel des Bildes und auch technische Dinge, also ich bin bei einem Clarisse zum Beispiel in einer früheren Version einen Shot, den ich gemacht habe, da is einfach das Deep Render, es is zwar rausgekommen, aber es hat ned das gleiche Anti-Aliasing gehabt wie der Beauty-Render. Wennst du sie jetzt kombiniert hast mit einander, weil das ist separat gerendert worden, hast dann plötzlich Kanten bekommen die ned da sein sollten. Also genau den einen Pixel, wo die zwei Render aufeinander treffen sollten, die haben halt einfach ned gepasst. Und wennst dann die Möglichkeit hast, okay, du schiebst das dann dorthin oder im Idealfall, okay, gibs mir ins Nuke, ich kümmere mich selber darum, dass das irgendwie fertig wird, weil als Comper bist du ja dann manchmal in der Situation, dass du einfach alles ersetzt, was bis jetzt noch ned so wirklich funktioniert hat. Also ich war immer wieder in der Situation, dass ich mir selber einen 3Dler hab suchen müssen, auch in großen Studios, dem ich jetzt gesagt habe, du, renderer mir das jetzt, aber bitte mach das so und stell das ab und schieß mir die Normals dabei raus. Also man muss dann schon als Comper so ein bisserl einen Überblick bekommen, zumindest über Renderer. Da hab ich das Gefühl, dass das sehr wichtig ist. ja. Ich muss ned wissen, wie, wo, was modelliert ist, und ich muss ned wissen wie ein Displacement funktioniert, oder was auch immer, aber ich muss wissen, wo Probleme im Prozess haben entstehen können, wenn die Sachen bei mir landen. Also UVs muss ma auch irgendwie kapiert haben, weil wenn bei dir irgendwo der Fehler entsteht, musst du meistens beim Troubleshooting helfen, als compositing artist und zurück gehen. Und manchmal gehst hald dann zum 3Dler und fragst, wie schaut das bei dir aus. Und dann gehst zum Modeller und den fragst, ja, hast das 3D Modell und kannst mir das mal zeigen. Und du schaust hald dann einfach Step by step wo der Fehler entstanden sein könnte, zum Beispiel. Und da ist dann so ein bissi so ein systemorientiertes Denken dann schon auch wichtig, gö. Also das würde ich jedem empfehlen.

I: Die Octane-Implementierung in Nuke verfügt über sein eigenes Shadernetzwerk-Fenster. Ähnlich dem Material-Network in Houdini oder dem Hypershade in Maya. Also glaubst du, dass jetzt eher so eine Implementierung sinnvoller ist als das, was jetzt RenderMan für Nuke bietet, wo du hald ein Diffuse oder einen Glossy Shader hast und die kannst mixen.

B: Also es ist schwierig, also weil RenderMan ist für mich eigentlich eine Maya Software. Sprich das ist für mich jetzt dort zuhause und dort sind die Grundfunktionen definiert, die er kennen muss. Ein Mantra zum Beispiel gehört ins Houdini, ein Red Shift... das ist hald schwierig mit Renderpaketen, die für mehrere Pakete angeboten werden. Weil V-Ray ist ja eigentlich durch Max definiert. Dafür ist immer die erste Release draußen. Und auch die Userbase konzentriert sich auch immer auf so eine Softwarekombination. Ich mein Maya und V-Ray gibts jetzt schon auch eine große Userbase inzwischen. Aber die meisten Leute kommen hald dann schon ausm Max Bereich die hald V-Ray verwenden. Oder die Gurus hald auch, gö. Also der Grant Warwick macht hald auch nur Max-V-Ray Sachen und das ist hald dann oft schwierig, dass auf Maya zu übertragen, weil hald einfach die Slider anders heißen und so weiter, gö. Und das Problem hat dann einfach Nuke auch, weils hald ein ganz ein anderes Konzept ist. Also wenn du hald jetzt dort ein V-Ray reinbringst, dann müsstest du eigentlich das Wissen, was Online verfügbar ist, also das ist ja eigentlich das, was unsere Lerndatenbank ist, was wir an Tutorials online finden und das was in den wenigen Büchern, welche es gibt, definiert ist und so weiter. Und das muss hald dann übertragbar sein auf die Workflows, die du in Nuke hast, gö. Also wenn du jetzt zum Beispiel ein Displacement steuern willst, dann müsstest du einen Weg finden, wie du... Sag ma jemand, der in Maya V-Ray bedient, der müsste es intuitiv in Nuke auch finden. Den gleichen Weg. Der müsste wissen, okay, das heißt jetzt Displacement und das stecke ich dort an; weil so mach ichs in Maya auch. Also ich finde das schon wichtig, dass es einen definierten Weg gibt und dass es die verschiedenen Softwarepakete dann an den Weg halten. Weil V-Ray für Max und Maya, das ist schon das Gleiche und die gleichen Slider, aber du kannst kein Tutorial in Max für Maya nachmachen, hab ich das Gefühl. Das ist so meine Wahrnehmung von dem Ganzen. Und genau, wie man es machen müsste, kann ich ned sagen. Als wie gesagt, für mich ist es das Wichtigste, dass es intuitiv für jemanden ist, der die Muttersoftware verwendet, dass der das in der Tochtersoftware, oder in der nicht dominanten Software auch machen könnte. Das wäre das Wichtige. Wie ist das bei Blender? Das Cycles ist standardmäßig Blender?

I: Genau. Aber es gibt auch eine Standalone-Applikation, von welcher der Source-Code verfügbar ist. Und die ist grundsätzlich sehr gut in Blender implementiert und

zusätzlich gibts hald noch Leute, die was es zusätzlich noch in Cinema4D implementiert haben. Aber grundsätzlich wirklich funktionieren tut es grundsätzlich nur in Blender.

B: Das heißt, das wäre grundsätzlich die Muttersoftware. Und wenn du das dann in Cinema4D verwendest, dann weißt du, dass da Probleme entstehen und du hast Abstriche und du weißt es wird ned alles funktionieren... Ich mein, dass is glaub ich auch, weil wir so von der Userbase so eine Randerscheinung sind, deswegen gibt es bei uns so technisch unausgereifte Produkte teilweise. Also das ist grundsätzlich oft das Problem von einer Produktion, dass man hald ein bissi einen Polster braucht, jetzt produktionstechnisch. Weil man weiß, irgendwo wird dann irgendwann mal ein Problem entstehen, wo jetzt vier Leute sitzen und überlegen, wie kann man das jetzt lösen. Wir haben lange überlegt, ob wir mit RedShift jetzt Volumes rendern sollen und so weiter und zu dem Zeitpunkt war es hald dann einfach noch ned so weit, dass es das unterstützt hätte. Das nächste Softwareupdate kommt raus, die nächste Version und dann könntest es trauen, dass du es angehst damit oder so. Aber das ist hald oft so eine Gefühlssache. Und je erfahrener die Leute sind, desto mehr gehen sie in diese ausgetretenen Pfade, wo sie wissen, dass das funktioniert.

I: OSL hast du schon angesprochen, dass man Shader austauschen kann. Ist OpenVDB-Unterstützung jetzt eine Voraussetzung, dass man sagt, man kann einen Renderer in einer Compositing-Applikation verwenden, dass man Simulation von Houdini exportieren und in Nuke importieren kann. Ist Universal Scene Description eine Voraussetzung, dass man sagt, man kann die ganze Szenenbeschreibung einer Software in Nuke aufmachen und man hat beispielsweise alle Lichter schon gesetzt und so weiter? Ist so etwas wichtig?

B: Es wäre super, wenn sowas funktionieren würde, ganz klar. Ich habe gerade überlegt, ich mein, ich nenne jetzt keine Namen, aber bei großen Studios ist das jetzt schon so, dass die jetzt so aufgebröselt ist. Also wennst jetzt zum Beispiel... Sag ma mal, du hast eine Szene, du hast eine große Crowd, die wird vom, ähm, da werden ein paar Characters animiert, dann wird von Simulation der Rest simuliert, dann kommt das zusammen und dann endet das bei einem Lighting-Artist, der in Maya und RenderMan das rausrendert. Das is so der Standard Usecase jetzt eigentlich. Dann hast aber Rauch, der da dazwischen drinnen ist, der kommt dann von einem Houdini-Artist und von einem FX-Artist. Meistens ist es dann so, also bei komplexeren Szenen, dass jeder seine Sachen rendert und du als Compositor dann zum Beispiel dann... also mir is so gegangen. Du

bekommst ein Environment, das ist Clarisse, du bekommst die Crowd, das ist RenderMan, du kriegst die Animation, das ist RenderMan von einem anderen Artist, du kriegst die FX, das ist Mantra und du musst es miteinander kombinieren, irgendwie gö. Und das ist eh schon, also wenn man jetzt wieder Anti-Aliasing ansprechen und solche Sachen, dann ist es eh schon eine Herausforderung, weil jeder von den Renderer ein bissi schärfer ist und ein bissi was anderes liefert und so weiter. Aber jeder von denen muss eigentlich das gleiche HDR verwenden wenn er das rendert und die gleichen Lichter. Also das muss einfach gleich sein, weil sonst hast du als Compositor unheimliche Probleme, dass du das zusammenfügt. Und die größeren Studios haben das unter Kontrolle, dass das hald funktioniert eigentlich. Ob du das in Nuke dann auch brauchst oder ob das dann viel bringt, wenn du einen eigenen Renderer in Nuke hast. Eigentlich muss es das unterstützen, ja. Wenn du komplexere Sachen machst in einem Team, dann muss das austauschbar sein. Ich weiß aber jetzt ned, wie die das machen. Ob das Open Scene Description ist, mit denen die diese Sachen austauschen.

I: Diese Universal Scene Description ist die letzten Jahre erst gekommen. Is ja von Pixar entwickelt worden und da gibts er für Maya ein Plugin was sie zur Verfügung stellen und für Houdini, glaub ich. Und die wollen das recht pushen, auch Ilm.

B: Ilm wird auch ziemlich dahinter sein, na sicher. Das is ja super für die, wenn einfach da FX Artist die Szene reinholt und die Szene schaut gleich aus. Jedes mal wenn ich daran denke, denke ich mir, was da alles für Probleme entstehen können, wennst du das überlegst. ja. Und vor allem auch Dinge, die dann irgendwann dann erst später einmal auftauchen. Wie gesagt, du versucht wirklich einen fehlerfreien Workflow hinzukriegen, wo dich dann hald ned irgendwas überrascht. Ich mein das ist hald so. Ich mein bei ILM, mich hat das sehr interessiert, wie deren Pipeline ist. Weil ich mein, die machen riesen Sachen und was die rendern. Das könnte ich auf meinem Rechner ned einmal aufmachen und so weiter. Und mich hat das sehr ernüchtert, dass die Pipeline gefühlt etwa 40 Jahre alt ist. Das ist einfach gewachsen das Ganze und so organisch ist hald dann dort ein kleiner Bereich dazugekommen und da. Aber ich hab das Gefühl, die Dinge von damals bestehen noch immer. Ja. Also da bist sicher bei einem Studio wie Atomic Fiction die es erst seit kurzem gibt und die da immer versuchen, da ein bissi so revolutionär zu denken, ist es pipeline-technisch sicher viel spannender und du hast jetzt auch mehr die Chance, dass du jetzt so einen Bereich auch umsetzt. Bei MPC... ich mein, dass ist hald so ein bissi auch Studiokultur. Bei MPC hab ich das Gefühl gehabt, die Pipeline ist so starr und so vorgegeben, dass selbst ein Junior kommen kann und der wird von einem Supervisor irgendwie durch eine Betreuung soweit hin gebracht, dass er ein Endergebnis liefert. Bei ILM hab ich

ned das Gefühl gehabt, dass die so arbeiten. Also da ist es eher so, dass die Seniors suchen, die das selbst betreuen eigentlich, gö. Und das ist in jedem Studio so ein bissi eine andere Kultur.

I: Da brauchen wir nicht lange darüber reden, aber das eine Unterschtützung von AOVs, Light Select Passes und Deep wichtig wäre?

B: Das wäre, wenn man jetzt mit anderen Softwarepaketen zusammenarbeitet, dann wäre das wichtig. Gerade Deep is der Standard geworden, wenn man Renders miteinander kombiniert, weil du hald keine Holdouts brauchst und so weiter. Und das passt hald dann auch vom anti-aliasing. Genau, was ich vorher noch sagen wollte: Was mir, was ich spannend gefunden habe, bei ILM gibt es hald so einen Bereich, das sind die Generalists. Das war jetzt eine Position, bei der du die kreative und auch technische Freiheit gehabt hast, dass du selber auch Dinge machst. Und der Gedanke dahinter war hald eben, dass super komplexe Shots mit allen Mitteln die verfügbar sind, umgesetzt werden. Was ich spannend gefunden habe, das was sonst eine große Pipeline, eine alte, eine gewachsene und die Artists waren dann ned mit Maya, die haben mit Max gearbeitet, die haben mit V-Ray gearbeitet und haben DMPs selber gezeichnet für die Sachen, also da diese Pipeline wieder komplett aufgebrochen worden. Wahrscheinlich habens irgendwann mal, produktionstechnisch, mitbekommen, dass sie da zu strikt waren und dass das auch nicht immer funktioniert hat, ihre strikten Produktionswege, und haben dann irgendwann mal gesagt, okay, wir brauchen da hald ein paar Leute, die Freiheit haben, dass sie Herausforderungen mit den Lösungswegen angehen sollen, die sie selber entwickeln. Also wir in Österreich haben irgendwie oft das Problem, dass wir irgendwie eine Lösung finden müssen und dann heißt, okay, und dann brauch ich den Renderer dafür und dann installiere ich RedShift und dann brauch ich eine Grafikkarte und das ist ein bissi das Pendant dazu, also hätte ich gesagt. Und dort hätte ich auch die Möglichkeit gesehen, dass ich im Compositing-Package dann einen Renderer integrier, weil das auch weniger departmentsmäßig aufgebröselt ist.

I: Du hast vorher schon Spiele Engines angesprochen. Glaubst, würde das Sinn machen, gerade auch mit den neuen OpenGL-Standards, wo ziemlich gute Renderings auch in Echtzeit möglich sind. Und jetzt sowieso mit dem NVidia Raytracer, diesen RTX. Glaubst du, wird so etwas mehr Sinn machen oder mehr so dieses physically based, unbiased Rendering?

B: Ich glaub, dass des kommen wird, ja. Davon reden ja alle. Da Vlado hat das letzte Mal im Podcast gesagt, sie versuchen hald natürlich auch ins Realtime-Rendering vorzustoßen. Und eigentlich ist es eh ein bissi peinlich, dass wir noch ned dort sind. Dass ein Render noch immer 10 Stunden braucht, auf einem Computer für einen Frame. Das ist eigentlich brutal. Und das sich das auch nie verändert. Das ist hald einfach so ein Standard. Man sagt, eine Stunde pro Frame, pro Rechner ist so der Standardwert. Und dann hast hald noch immer Frames, wo hald Refractions und was auch immer drinnen ist, die ewig rechnen. Ähm. Und weiß ich jetzt ned. Wennst dir jetzt das neue Far Cry anschaut, dass auf einer normalen Gaming-Grafikkarte einfach mit flüssiger Performance läuft, dann denkt man sich schon, warum dauert das bei uns so lange. Es ist auch schwierig jemanden zu erklären der dem Thema fremd ist, warum das solange dauert. Warum das jetzt, wenn wir fertig sind mit dem Shot, dann eine Woche dauert bis der rausgerendert ist und so weiter. Und momentan ist hald einfach der Qualitätsunterschied einfach noch immer da, gö. Also ich glaub einfach, dass das Raytracing, wennst es aufdrehst, dann braucht das hald einfach bis es die ganzen Lichtbounces berechnet hat. Das ist hald einfach auch noch der große Unterschied, hätte ich gesagt. Also wenn man jetzt zum Beispiel diese Light-Bleed-Effekte anschaut, wennst eine helle und eine dunkle Fläche hast oder einen langen Tunnel und da fließt Licht rein oder sowas, dann musst du das hald schön Raytracen, damit das richtig ausschaut. Aber die Game Engines... also ich glaub, dass das VR Thema jetzt die Game Engines jetzt sehr pushen wird. Dass das, wenn die Leute dann aufspringen und das 360 Grad-Video und das ganze Zeugs dann vergessen hoffentlich bald, dann werden die Game Engines sich sehr entwickeln. Und ich glaube, dass es da dann irgendwann, also das wird in 15 Jahren oder so soweit sein, dass es da dann also eine Kombination geben muss, aus Video Live-Footage, dass irgendwie dann so vielleicht pointcloudmäßig aufgenommen wird und Game Engines, die das dann zu spielen mit echter 3D-Verarbeitung. Wird spannend sein, was da entstehen wird.

I: Im Hinblick auf die Integration in eine Compositing-Software, Blender implementiert die neuesten OpenGL-Standards als Viewport-Renderer, das erlaubt recht coole Ergebnisse, macht das Sinn im Compositing?

B: Nein. Im Compositing brauchst du immer die maximale Qualität. Das ist das wichtigste eigentlich. Also das würde im 3D wahrscheinlich mehr Sinn machen, wennst eine Preview haben willst oder sowas. Aber solange die Echtzeitlösung noch ned die Qualität liefert, kannst damit noch ned arbeiten. Also ich bin immer hin und weg von dem, ich glaube, es ist eh Octane, in Cinema4D. Und da schauen ja diese Realtime-Displacements ja wahnsinnig gut aus. Und das entwickelt sich

schon wirklich. Aber es ist noch ned dort, dass es für produktionstechnische -für finale Qualität irgendwie verwendbar wäre, hätte ich das Gefühl.

I: Und jetzt einfach noch generell um die Anwendungsgebiete zu definieren, was so eine Implementierung haben kann. Was man damit in einer Produktion machen kann. Einerseits in einem großen Studio, andererseits in so einem kleinen wie da.

B: In den großen Studios würde es wahrscheinlich in diese Environmentebereich reinfließen. Da hätte ich das gesehen, dass das in den Departments Platz finden könnte. In einem kleinen Studio... da würde dann der Compositor die Möglichkeit haben, selber 3D-Elemente selber zu machen, also so wie das Elements 3D, so in der Art würde es dann verwendet werden. Was vielleicht die Effizienz steigern könnte, aber ich glaube nicht, dass es die Qualität steigern würde. Als Compositor schaust hald auch immer, das ist so eine unausgesprochene Regel, dass dein Shot nie länger als eine halbe Stunde rendert. Also auf der Farm oder Lokal oder was dir hald zur Verfügung steht. Da musst du hald schauen. Gö. Also deine Elemente so precompen, dass wenn du auf render drückst, dass es dann eine halbe Stunde dauert. Es ist dann oft bei Abgabesituationen eh stressig und du musst hald schauen, dass du deinem Supervisor in der Situation so viele Versionen vorlegst, dass der hald noch reagieren kann drauf. Also die schlimmste Situation, ich hab das hald manchmal gehabt mit Artists, die hald am Anfang diese Prozess noch ned gekannt haben, dass ich nach zwei Tagen zum ersten Mal eine Version gesehen hab oder dass die zu mir gekommen sind und gesagt haben, sie haben jetzt los geschickt und das rendert jetzt sieben Stunden. Und damit kannst hald im Compositing... Also das ist hald 3D-Arbeit eigentlich. Du musst hald einfach immer schauen, dass das performant und responsiv bleibt und so, das ist deine Verantwortung. Und deswegen würd jetzt so ein 3D-Paket jetzt Sinn machen. Aber es würde für vorgerenderte Sachen Sinn machen. Also bei Prometheus wars hald so, da haben wir diese ganzen Environments mit Scanline Renderer vorgerendert. Also das war tatsächlich texturierte Geometrie, der ScanlineRenderer hat das ausgespuckt und das coole dabei war, ich hab da hald dann das Script bekommen und hab mir selber Masken zeichnen können. Für den Berg und hab dann hald so in Rot, Grün, Blau mir selber so Masken rausrechen können. Die haben dann hald zusammengepasst einfach mit dem was ich gebraucht habe, gö. Also was das Environment Rendering zum Beispiel geliefert hat. Der Nachteil vom ScanlineRenderer, der war hald so scharf, dass der RenderMan damit hald ned mithalten hat können. Also der macht hald pixelgenau super scharf seine Sachen. Und du hast hald alles gesehen, also der hat der nichts vergeben, beim Texturieren und so weiter. ja.

I: Glaubst, kann in einen großen Studio auch das Problem sein, ähm, es gibt dort recht große Feedbackschleifen mit vielen Leuten, also Regie, DOP, Art Department und den ganzen VFX-Supervisoren, dass es einfach schwer wird zu tracken, welches Department dafür zuständig ist, wenn ein 3D-Element neu gemacht gehört und dann ist das aber im Compositing erstellt worden?

B: Ja, das ist oft gar ned so leicht. Es gibt dann auch oft Shows, wo dann das Personal wechselt zum Beispiel. Und das ist oft ein Problem, dass jetzt jemand auf die nächste Show wandert, weil der wird dort gebraucht, weil er ist derjenige, der Mari Textures macht und so weiter. Und dann popt der Shot zum Beispiel wieder auf, weil irgendjemand bemerkt hat, dass dort weniger Nebel drinnen ist oder dort irgendwas anderes fehlt oder die Texturen falsch ist. Und dann brauchst hald wieder jemanden der das wieder erfüllen kann. Gö. Also sind diese standardisierten Pipelines und die standardisierten Departments für Produktionen eigentlich schon super, wenn sie funktionieren, wenn sie alles abdecken. Ähm. Es wäre dann die Grundvoraussetzung von Compositoren, dass sie den Bereich auch abdecken könnten. Und das kann ned jeder Comper, glaub ich. Das ist glaub ich so ein großes Problem. Also ich kann mir vorstellen, dass wenn du ein kleineres Studio hast, dass du das den Leuten zeigst, dass man das gemeinsam entwickelt, weil wenn du irgendwie vier Comper hast und alle sehr technisch begabt sind, dass man das gemeinsam irgendwie so aufzieht, dass jeder das Wissen hat um den Bereich auch abzudecken. Aber meistens ist es eher ned so. Hätte ich gesagt.

I: Ist bei kleineren Studios eher Effizienz das Wichtigste, weil Artist-Stunden das teuerste sind, oder spielen da Lizenzkosten schon auch eine Rolle?

B: Also Lizenzkosten sind schon auch brutal hoch, das ist mindestens soviel auch, wie die Miete auch. Aber du ersparst dir ja nix dadurch, wennst du es in Nuke renderst, weil Nuke ist ja jetzt auch kein billiges Paket, muss man auch sagen. Dafür das es eigentlich auch nur Compositor ist. Du brauchst eine Nuke-Renderlizenz, die kostet auch wieder 200 Euro oder so pro Rechner. Oder du sagt, die hast eh. Also wennst eh genug Renderlizenzen hast, dann würdest dir da ein bisserl was sparen, aber du musst es dann pro Rechner auch wieder lizenzieren, des hilft nix eigentlich. Also am besten wäre eigentlich eine Software, die alle Pakete beliefert. Wenn man sagt zum Beispiel, eine V-Ray Lizenz, die kannst in Maya, in Max, in Houdini, in Nuke verwenden, das wäre ein guter Usecase. Aber ich glaub das wäre jetzt auch ned so lizensiert, V-Ray jetzt.

I: Da hast die Kosten für die Plugins und die Renderlizenzen. Da exportierst die vrayscenes.

B: Ja, aber das geht ja auch ned überall. Also manchmal muss dann eine Mayaszene exportieren und das mit Maya rendern und so. Also ja. Also ich glaub ned, dass man sich viel ersparen würde, jetzt. Und es ist tatsächlich so, also wir sind immer wieder vor der Entscheidung gestanden, auch wegen Lizenzkosten, sollen wir uns zum Beispiel eine Fusion verwenden, weils hald einfach gratis ist, gö. Wenn man sich unsicher ist, ob man die gleiche Qualität hinbekommt mit der Software, dann zahlt man lieber viel Geld dafür und nimmt die bisserl bessere Variante. So wies mit Nuke hald ist. In unserer Strategie ist einfach das, was die maximale Qualität liefert das Richtige. Und ich glaube, dass ist langzeitmäßig auch die beste Variante zu kalkulieren.

I: Und die Artist-Stunden, Fusion muss man anlernen. Im Vergleich Nuke, können mehr Leute.

B: Ja, du findest hald mehr Artist dafür. Das ist auch wichtig. ja, das stimmt. Also Nuke ist da hald dann einfach schon so ein Standard, gö. Und ich weiß jetzt auch ned, wie man bestimmte optische Effekte im Fusion jetzt zum Nachbauen beginnen würde. Also wenn man sich wirklich darauf spezialisiert, dann würde man es wahrscheinlich hinkriegen, dass man in ein paar Monaten das Setup, alles, auf Fusion überträgt. Aber ich glaub ned, dass es Sinn machen würde für uns jetzt. Und die Entwicklung in Nuke ist hald anscheinend dann einfach besser, die User Base is besser, also die ganzen Tools die da passieren und so weiter. Problemlösungen findest hald bessere, weils hald einfach mehr Leute verwenden und so. Also ich hab das Gefühl, da is es immer gut, also strategisch gescheit, mit dem Platzhirsch zu gehen, mit der dominanten Software irgendwie zu arbeiten.