UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Rendering von Haaren und Fell

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Emma Jane Kraft

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	M. Sc. Bastian Krayer (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Februar 2018

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

In der Computergrafik stellte das echtzeitfähige Rendern von Haaren und Fell ein Problem dar. Die Berechnung der Beleuchtung, Schattierung und Transparenz erfordert einen hohen Rechenaufwand, welcher sich negativ auf die Performanz auswirkt. Doch durch verbesserte Hardware und neue Verfahren ist es möglich, solch komplexe Effekte in Echtzeit zu simulieren. In folgender Arbeit werden die Grundlagen des Renderings von Haaren erläutert. Außerdem wurde im Rahmen der Arbeit eine echtzeitfähige Demo implementiert, deren zugrunde liegende Verfahren und Funktionalitäten beschrieben werden. Um die Demo zu evaluieren wurde die mögliche Anzahl an Bildern pro Sekunde bei Modellen unterschiedlicher Komplexität gemessen. Schließlich wurden die Ergebnisse mit Bildern von echten Haaren verglichen.

Abstract

In Computer Graphics, rendering of fur and hair in real-time used to be a problem. The simulation of illumination, self-shadows and transparency requires many computations on a graphic processing unit which limit the performance. Due to improved hardware and new algorithms it is possible to render these complex effects in real-time. The basics of rendering hair and fur are presented in this thesis. Furthermore a program was designed and implemented, which is able to render hair and fur in real-time. The used methods and implementation are described. Lastly the number of frames per second are measured when rendering models of different complexity. Additionally, the rendered images are compared to images of real hair and fur.

Inhaltsverzeichnis

1 Einleitung

Die folgende Abschlussarbeit erläutert verschiedene Verfahren zum korrekten Rendering von Haaren und Fell, wobei die korrekte physikalische Simulation außer Acht gelassen wurde.

1.1 Motivation

Ob gepflegt oder fettig, ob blond oder brünett - Haare spielen im Leben eine wichtige Rolle, denn oft vermitteln sie uns einen ersten Eindruck über einen Menschen. Zu diesem tragen viele Feinheiten der Haare, wie beispielsweise Haaranzahl, zum Gesamtbild bei. Die Anzahl steht dabei im Verhältnis mit Alter und Ethnie eines Menschen. So haben Afrikanerinnen durchschnittlich ca. 81.000 Haare, Asiatinnen ca. 89.000 Haare und Europäerinnen ca. 121.000 Haare. Dies kann ein Problem für die Hardware darstellen.

Doch die Verfahren zum Rendern von Haaren wurden schneller und besser, genau wie sich die Hardware stetig verbesserte. Durch neue *Shader* lassen sich Haare als einzelne Linien darstellen und nicht mehr als Polygonzug, wie im Videospiel *Tomb Raider* aus dem Jahr 1996 (Abbildung 1). Durch Frameworks wie *HairWorks* von NVIDIA oder *TressFX* von AMD lassen sich Haare mit korrekter Beleuchtung, Schattierung und Transparenzen rendern und simulieren. Des Weiteren werden die Frameworks zum Rendern von Fell und Gräsern benutzt. Im Spiel *Tomb Raider* aus dem Jahr 2013 wird TressFX erstmals verwendet, vergleiche dazu Abbildung 1.

Abbildung 1: Lara Croft aus dem Spiel *Tomb Raider* im Jahr 2013 (links) und 1996 (rechts), entnommen aus[1]

[1] http://www.xboxoneuk.com/2016/02/14/tomb-raider-some-handy-facts-and-figures/, Stand 09.02.2018

1.2 Aufbau

Die folgende Abschlussarbeit ist in die Kapitel *Grundlagen*, *Rendering* und *Implementation* aufgeteilt. Diese sind wiederum in *Beleuchtung*, *Schatten* und *Transparenzen* eingeteilt, da dies die Schlüsselelemente des Haar-Renderings sind. Im Grundlagen-Kapitel werden alle zum Verständnis des Rendering-Kapitels wichtiger Verfahren erläutert. Das Implementations-Kapitel gibt Code wieder, welcher zur Realisierung, der im Rendering-Kapitel vorgestellten Verfahren, verwendet wurde.

2 Grundlagen

Im Folgenden werden alle für diese Arbeit grundlegenden Verfahren zum Rendering von Haaren erläutert. Dazu gehört Grundwissen über die Grafik-API *OpenGL*, Verfahren zur *Beleuchtung*, *Schattierung* und *Transparenzberechnung* von Objekten.

2.1 OpenGL

Das folgende Kapitel basiert auf Informationen der *OpenGL Super Bible* [1].

Eine in der Computergrafik weitverbreitete Schnittstelle zur Grafikprogrammierung ist OpenGL. Sie wurde 1992 von Silicon Graphics Inc. (SGI) entwickelt, welche für die Herstellung von Computern mit leistungsstarker Grafik bekannt war. Es handelt sich dabei um eine Open-Source-Software, sodass sie von jedem verwendet werden kann.

Da sich die Computergrafik in den darauffolgenden Jahren weiterentwickelte, wurde OpenGL an die neuen Standards angepasst. So wurde im Jahr 2004 die *OpenGL Shading Language* (GLSL) in Version 2.0 eingeführt, welche das Programmieren von *Shadern* ermöglicht. Zunächst gab es *Vertex* und *Fragment Shader*, bis im Jahr 2008 OpenGL um einen *Geometry Shader* erweitert wurde. Doch da die Funktionalitäten des neuen Shaders für neuere Verfahren der Computergrafik nicht ausreichend waren, wurde ein Jahr später der *Tessellation Shader* der API in Version 4.0 hinzugefügt. Um die GPU auch als *GPGPU (General Purpose Computation on Graphics Processing Unit)* nutzbar zu machen wurde der Schnittstelle im Jahr 2012 der *Compute Shader* hinzugefügt. Seitdem kann die GPU nicht nur für Grafikanwendungen verwendet werden, sondern auch für allgemeine Berechnungen, welche auf der GPU parallelisiert werden können. Die aktuellste Version, die 2017 veröffentlicht wurde, ist 4.6.

In den folgenden zwei Unterkapiteln wird auf den Aufbau von OpenGL eingegangen, also die Funktionsweise der Shader und der Pipeline.

2.1.1 Shader

Mit der Einführung der GLSL konnten nun Shaderprogramme genutzt werden. Ein Shaderprogramm besteht aus vom Benutzer erstellten Shadern. Zum Zeitpunkt des Erscheinens der Version 2.0 gab es den *Vertex* und *Fragment Shader*, später wurden *Tessellation Control*, *Tessellation Evaluation*, *Geometry* und *Compute Shader* hinzugefügt. Da für diese Arbeit keine Parallelisierung von allgemeinen Berechnungen auf der GPU nötig ist, werden im Folgenden keine *Compute Shader* beschrieben.

Vertex Shader Der einzige Shader, der in jedem Shaderprogramm vorhanden sein muss, ist der *Vertex Shader*. Dieser ist für die Verwaltung der

Geometrien zuständig. So werden die im Hauptprogramm erstellten *Vertex Buffer Objects* (VBO), bzw. das *Vertex Array Object* (VAO), an den Shader übergeben. Ein VBO beinhaltet Attribute der Geometrie wie beispielsweise Vertices, Normalen oder Tangenten. Um die Verwaltung der Buffer Objekte zu vereinfachen, werden mehrere VBOs zu einem VAO gebündelt. Das VAO dient dem Shader also als Dateninput. Als Datenoutput muss im Shader die OpenGL-Variable `gl_Position` gesetzt werden, wofür meistens die an den Shader übergebenen Vertices verwendet werden. Um beispielsweise eine Bewegung der Geometrie zu erzeugen müssen weitere Attribute an den Shader übergeben werden. Zum Beispiel eine *Model-Matrix*, welche die Transformationen der Geometrie speichert. Außerdem sollten noch eine *View-Matrix* und eine *Projection-Matrix* an den Shader übergeben werden, damit die Geometrie korrekt in die Welt projiziert werden kann.

Tesselation Control Shader In der OpenGL Version 4.0 wurden zwei neue Shader hinzugefügt, um Tessellation umzusetzen. Tessellation beschreibt dabei die Unterteilung eines Primitivs in kleinere Primitive. Damit ist es möglich einen Würfel als Geometrie an den *Vertex Shader* zu übergeben und diesen mittels *Tessellation Control* und *Tessellation Evaluation Shader* weiter zu unterteilen, sodass am Ende eine Kugel entsteht. Abbildung 2 zeigt die immer feiner aufgelöste Geometrie eines menschlichen Kopfes. Die Tessellierung wird in OpenGL in drei Stufen unterteilt: *Tessellation Control Shader*, *Tessellator* und *Tessellation Evaluation Shader*, wobei die Shader vom Benutzer programmierbar sind, der *Tessellator* jedoch nicht.

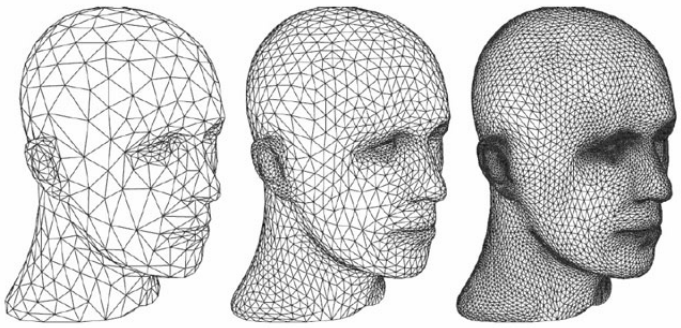


Abbildung 2: Verschiedene Faktoren für Tessellation, entnommen aus [2]

Der Tessellation Control Shader arbeitet auf Kontrollpunkten, welche die Vertices aus dem Vertex Shader sind. Eine festgelegte Menge dieser Punkten wird dabei als *Patch* bezeichnet. Der Shader legt den Modus und die Faktoren der Tessellierung fest. Der Modus gibt dabei an, ob Dreiecke oder Quads unterteilt werden sollen. Die Faktoren der Tessellierung beschreiben, wie das

[2] https://i.stack.imgur.com/uT6do.jpg, Stand 10.01.2018

Patch unterteilt werden soll. Beispiele für Faktoren wird in Abbildung 2 illustriert.

Tessellation Evaluation Shader Nachdem das Patch unterteilt ist, müssen die Ergebnisse des *Tessellators* im *Tessellation Evaluation Shader* ausgewertet werden. Diesem wird der Modus übergeben, welcher im vorangegangenen Shader verwendet wurde. Außerdem werden die baryzentrischen Koordinaten der neu hinzugefügten Punkte übergeben. Somit lässt sich ein neuer Punkt erzeugen, welcher durch die *build-in*-Variable `gl_Position` gesetzt wird.

Geometry Shader Während der Tessellation Shader zum Erzeugen vieler neuer Geometrien verwendet wird, ist es möglich mit dem *Geometry Shader* die Geometrie selbst zu verändern. So lassen sich beispielsweise die Normalen von Dreiecken leicht darstellen. Wie der Vertex und essellation Shader, welche für Geometrien genutzt werden, muss auch bei diesem Shader die *build-in*-Variable `gl_Position` gesetzt werden. Diese kann auch mehrmals im Shader belegt werden, da dieser nicht pro Vertex arbeitet, sondern pro Primitiv. Daher handelt es sich bei den in-Variablen auch immer um Arrays. Im Shader muss auch festgelegt sein, welche Art von Primitiv übergeben wird und welche Art von Primitiv ausgegeben werden soll. Um die Geometrie im Shader zu verändern, müssen zwei *build-in*-Funktionen verwendet werden: `EmitVertex()` und `EndPrimitive()`. Erstere signalisiert, dass dem Shader ein neuer Punkt hinzugefügt wurde. Die zweite Funktion schließt die Erstellung eines Primitivs ab.

Fragment Shader Nachdem die Geometrie in die Szene transformiert, durch Tessellation verfeinert und sie verändert wurde, wird die Farbe berechnet. Dabei muss im *Fragment Shader* die Output-Variable gesetzt werden. Diese kann ein vec4 sein, falls eine RGBA-Textur verwendet wird. Ist beispielsweise ein Tiefenbild gewünscht, kann die Output-Variable auch ein *float* sein. Innerhalb des Shaders können Beleuchtung und Verschattung berechnet werden, siehe Kapitel 4.

2.1.2 Pipeline

Der einzige Shader, welcher für OpenGL benötigt wird, ist der Vertex Shader. Alle Anderen sind optional. Je nach Problemstellung kann man ausgewählte Shader zu einem Shaderprogramm *linken*.

Die erste Stufe der OpenGL-Pipeline ist der *Vertex Fetch*. Hierbei wird das im Hauptprogramm gesetzte VAO an den Vertex Shader übergeben. Dieser setzt die Punkte der Geometrie in die Szene. Daraufhin werden diese, falls vorhanden, an den Tessellation Control Shader übergeben, welcher die Einstellungen für den Tessellator setzt. Daraus werden die baryzentrischen

Koordinaten ausgelesen und im Tessellation Evaluation Shader die neu hinzugefügten Vertices gesetzt. Falls das Primitiv oder die Geometrie verändert werden sollen, wird der Geometry Shader danach ausgeführt. Daraufhin werden die Daten zur nicht programmierbaren Stufe des *Primitive Assembly* weitergegeben. Dort werden die Punkte zu einem Primitiv zusammengefügt, welches danach durch den *Rasterizer* in Fragmente unterteilt wird. Diese spiegeln die Pixel des Bildschirms wieder. Nach zwei nicht-programmierbaren Stufen der Pipeline folgt der Fragment Shader, welcher die Farbe in den *Frame Buffer* schreibt. Dieser Buffer kann entweder der Bildschirm oder ein *Frame Buffer Object* (FBO) sein. In ein FBO zu rendern ermöglicht die weitere Verarbeitung der dem FBO angehefteten Texturen.

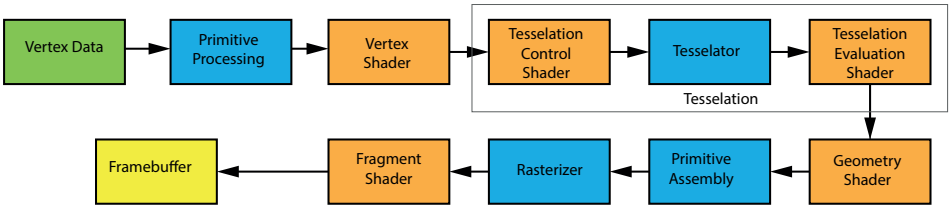


Abbildung 3: OpenGL Pipeline Version 4.0

2.2 Beleuchtung

Die Beleuchtung ist in der Computergrafik von großer Bedeutung, da sie den Effekt der räumlichen Tiefe erzeugt, der Gegenstände in einer virtuellen Welt realistischer aussehen lässt. Dabei gibt es verschiedene Beleuchtungsverfahren, welche meistens eine Gemeinsamkeit besitzen: die Aufteilung in *ambiente*, *diffuse* und *spekulare* Beleuchtung. Wie diese verschiedenen Bestandteile berechnet werden, ist jedoch vom Verfahren abhängig. Das Ergebnis einer Beleuchtung nach Phong [2] wird in Abbildung 4 in die einzelnen Bestandteile aufgeteilt. Diese werden für die finale Farbgebung addiert.

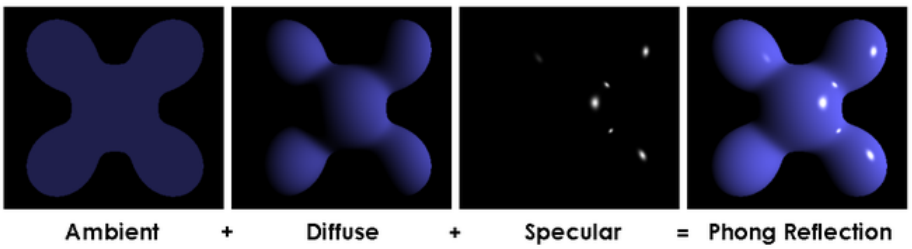


Abbildung 4: Die verschiedenen Bestandteile von der Phong-Beleuchtung, entnommen aus [3]

[3] https://en.wikipedia.org/wiki/Phong_reflection_model, Stand 15.01.2018

Der ambiente Teil der Beleuchtung beschreibt dabei das Licht, welches durch Streuung auf das Objekt scheint. Es wird das indirekte Licht simuliert, welches auf die Objekte der Szene fällt und als Grundbeleuchtung dient. In Abbildung 4 fällt auf, dass auf die Form des Objekts, welches ambient beleuchtet wurde, nicht geschlossen werden kann, da keine räumlichen Effekte entstehen. Diese treten erst bei diffuser Beleuchtung auf.

Der diffuse Teil des Lichts beleuchtet das Objekt der Szene gleichmäßig von der Lichtquelle aus. Es werden Lichtstrahlen simuliert, die an dem Objekt in unterschiedliche Richtungen reflektiert werden. Die Richtungen der reflektierten Lichtstrahlen sind materialabhängig; ist ein Material ideal diffus gleicht dies einer *Lambertschen Reflexion*. Ein Objekt erscheint somit aus allen betrachteten Winkeln gleich hell. Ein Beispiel hierfür ist ein Blatt Papier, denn es wirkt aus allen Blickwinkeln gleich hell. Im Allgemeinem gilt nach dem *Lambertschen Gesetz*, dass die Strahlstärke kreisförmig verteilt ist, wenn die Strahldichte des Objekts konstant ist (vergleiche Abbildung 5). Dies ergibt sich durch den perspektivischen Effekt, welcher die Strahlungsstärke mit flacher werdendem Winkel zur Oberfläche des Objekts verringert.

Der spekulare Teil der Beleuchtung ist der Teil des Lichts, welcher auf dem Objekt regulär reflektiert wird. Für die Farbgebung ist dabei nicht die Objektfarbe von Relevanz, wie es bei ambienter und diffuser Beleuchtung der Fall ist, sondern die Farbe des Lichts bestimmt die Farbe des spekularen Terms. Wie man in Abbildung 4 sehen kann, handelt es sich beim spekularen Term um die *Highlights* des Objekts.

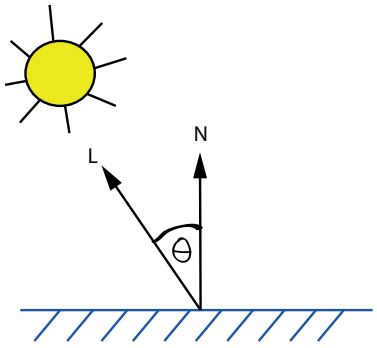


Abbildung 5: Lambert Beleuchtung

2.2.1 Kajiya und Kay

Das von *Kajiya und Kay* [3] im Jahr 1989 veröffentlichte Verfahren zur Beleuchtung, war das Erste, das sich nur auf Haare spezialisierte. Die Beleuchtung wird hierbei in eine diffuse und eine spekulare Komponente aufgeteilt, eine ambiente Komponente gibt es hierbei nicht. Der spekulare Teil der Beleuchtung ist dabei eine auf Zylinder übertragene Variante des spekularen

Terms nach Phong. Die diffuse Komponente ist vom Lambertschen Modell auf kleine Zylinder abgeleitet.

Ein wichtiger Unterschied zwischen Kayija und Kay und Phong ist, dass nicht mittels der Oberflächennormale, sondern durch die Normal-Ebene beleuchtet wird. Kayija und Kay gehen davon aus, dass jedes einzelne Haar aus mehreren Liniensegmenten besteht. Jedes Segment besteht wiederum aus zwei Punkten, welche jeweils eine Tangente besitzen. Dieses Modell wird in Abbildung 7 verdeutlicht. Um ein Fragment zu beleuchten, benötigt man die aktuelle Position des Fragments x_0, die dazugehörige Tangente $\vec{t}$, den Lichtvektor $\vec{l}$ und den Kameravektor $\vec{v}$, welcher von x_0 zur Lichtquelle, beziehungsweise zur Kamera zeigt. Außerdem sind $\vec{t}$, $\vec{l}$ und $\vec{v}$ normalisiert.

Um die diffuse Komponente zu berechnen, muss der Lichtvektor in die Normal-Ebene, welche orthogonal zur Tangente steht, projiziert werden:

$$\vec{l_p} = \frac{\vec{l} - (\vec{t} \cdot \vec{l}) \cdot \vec{t}}{\|\vec{l} - (\vec{t} \cdot \vec{l}) \cdot \vec{t}\|} \tag{1}$$

Nach einigen Umformungen des diffusen Lambertschen Beleuchtungsmodells lässt sich die diffuse Komponente nach Kajiya und Kay aufstellen:

$$\Psi_d = c_d \cdot \sin(\vec{t}, \vec{l}) \tag{2}$$

Hierbei handelt es sich bei Ψ_d um den diffusen Term und c_d ist die diffus reflektierende Komponente. Der Eingabewert für die trigonometrische Funktion ist der Winkel zwischen den Vektoren.

Der Wert der diffusen Beleuchtung wird mit dem der Spekularen addiert. Dieser berechnet sich ähnlich zu der in Phong (1975) [2] vorgestellten Gleichung:

$$\Psi_s = c_s \cos^m(\vec{v}, \vec{v_p}) \tag{3}$$

Dabei ist c_s die spekulare reflektierende Komponente, m die Glanzzahl und Ψ_s der spekulare Term. Der projizierte Vektor $\vec{v_p}$ berechnet sich analog zu Formel 1. Das finale Beleuchtungsmodell nach Kajiya und Kay multipliziert die Helligkeit komponentenweise mit der Summe aus den Gleichungen 2 und 3.

Ein Nachteil des Verfahrens ist, dass es nicht energieerhaltend ist. Die mangelnde Energieerhaltung stellt ein Problem für die physikalische Korrektheit des Renderings dar, doch können Grafiker so besser Eigenschaften der Haare verändern, wie in Kapitel 3.1 weiter beschrieben.

2.2.2 Marschner et al.

Das von Kajiya und Kay [3] vorgestellte Modell verwenden *Marschner et al.* [4] in ihrer Arbeit über die Streuung von Licht durch Haare. Es wird mit dem Koordinatensystem von Kajiya und Kay gerechnet: $\{\vec{t}, \vec{b}, \vec{n}\}$. Um eine

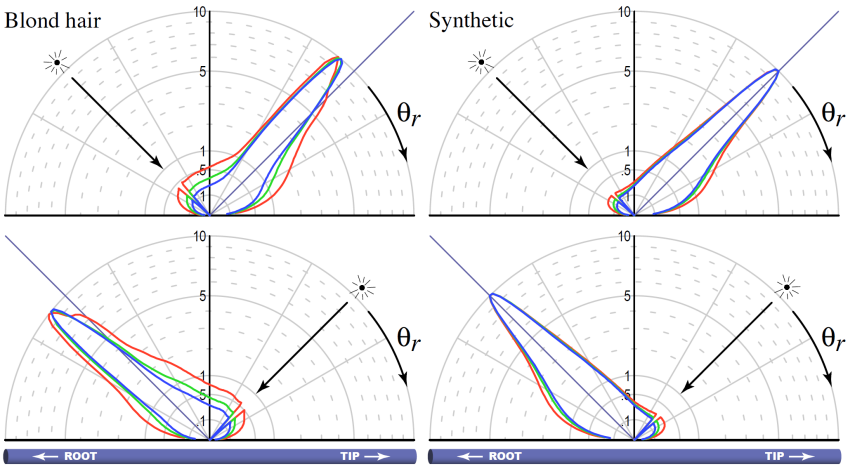


Abbildung 6: Messungen der Reflektion auf der Haaroberfläche von blondem und synthetischen Haaren, [4]

allgemeine *Scattering*-Function für Haare aufzustellen, untersuchten sie deren Streuungseigenschaften. Bei den Messungen fiel auf, dass das Licht nicht nur auf der Haaroberfläche reflektiert, sondern dass mehrere Streuungen auftreten.

Es gibt drei verschiedene Pfade bei der Streuung von Licht durch Haare: **R-**, **TT-** und **TRT-**Pfad, wie in Abbildung 8a dargestellt. Beim R-Pfad handelt es sich um die primäre spekulare Reflektion des Lichts an der Oberfläche eines Haars. Bei der Vergrößerung eines einzelnen Haars erkennt man *Haarkutikel*, welche an Dachziegel erinnern. Diese werden in Abbildung 8a angedeutet. Die Oberfläche eines Haars ist also nicht glatt, sondern unregelmäßig, weswegen die Richtung des primären spekularen Highlights bei Haaren einige Grad von der erwarteten Richtung abweicht. Dieser Effekt unterstreicht den Unterschied von synthetischen zu echten Haaren, da Synthetische keine Kutikel aufweisen. Daher reflektieren diese den Lichtstrahl an der Oberflächennormale, welche bei synthetischen Haaren senkrecht zur Oberfläche ist. Bei echten Haaren ist die Normale durch die Haarkutikula ca. 3° zur Haarspitze geneigt. Dieser Unterschied wird in Abbildung 6 verdeutlicht.

Der TT-Pfad beschreibt das Licht, welches auf die Oberfläche trifft, durch die Haarfaser scheint und auf der Rückseite wieder austritt. Dabei wird der Lichtstrahl bei Ein- und Austritt in beziehungsweise aus der Faser gebrochen. Anstatt an der Haaroberfläche beziehungsweise Haarinnenfläche zu reflektieren wird der Strahl weitergeleitet (*engl.: transmit*).

Beim Reflektieren des Lichts innerhalb der Haarfaser entstehen Kaustiken unterhalb des primären Highlights. Dies geschieht durch Licht, welches durch den TRT-Pfad beschrieben wird: Zuerst wird der Strahl ins Haarin-

nere weitergeleitet (*transmit*), im Inneren des Haars reflektiert (*reflection*) und schließlich wieder aus dem Haar geleitet (*transmit*). Dabei wird das Licht, ähnlich wie beim TT-Pfad, bei Ein- und Austritt in beziehungsweise aus dem Haar gebrochen. Dadurch liegt das zweite spekulare Highlight vom ersten Highlight in Richtung der Haarspitzen. Die Farbe der sekundären spekularen Reflektion hängt dabei von der Farbe der Haarfaser ab, im Gegensatz zur primären Reflektion, deren Farbe vom Licht abhängig ist. Die drei verschiedenen Pfade werden in Abbildung 8a illustriert.

Ziel der Arbeit war es, ein allgemeingültiges Beleuchtungsmodell für Haare aufzustellen, welches die drei Pfade des Lichts berechnet, da diese für die korrekte Wahrnehmung eine bedeutende Rolle spielen. Die Scattering-Funktion S, die nach den Messungen aufgestellt wurde lautet:

$$S(\omega_i, \omega_r) = \frac{d\bar{L}_r(\omega_r)}{d\bar{E}_i(\omega_i)} \tag{4}$$

Bei den Eingangsparametern der Funktion S handelt es sich um die Lichtrichtungen ω_i und ω_r, welche vom betrachteten Fragment zur Licht-quelle beziehungsweise Kamera verlaufen. Bei $\bar{L}_r$ handelt es sich um die Strahldichte, welche die Kamera empfängt. Die Bestrahlungsstärke, welche von der Lichtquelle ausgesendet wird, beschreibt $\bar{E}_i$. Allerdings handelt es sich bei der von Marschner et al. verwendeten Bestrahlungsstärke und Strahldichte um Funktionen, die sich nicht nur über eine Halbkugel, sondern über eine Kugel erstrecken. Es sind also infinitesimale Größen, die angenähert werden müssen. Unter Miteinbeziehung des Durchmessers D eines Haares lässt sich die Funktion der Bestrahlungsstärke (Gleichung 5) und somit das Scattering-Integral (Gleichung 6) berechnen.

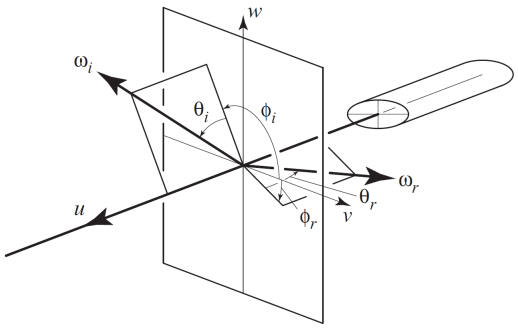


Abbildung 7: Hier tbn als uvw, ω_i entspricht dem Winkel zwischen Normal-Plane und dem Sehstrahl, ω_r entspricht dem Winkel zwischen Kamerastrahl und Normal-Plane, ϕ_i entspricht dem Winkel zwischen der Bitangente und dem Sehstrahl, ϕ_r entspricht dem Winkel zwischen der Bitangente und dem Kamerastrahl; [4]

$$d\bar{E}_i(\omega_i) = DL_i(\omega_i)\cos\theta_i d\omega_i \tag{5}$$

$$\bar{L_r}(\omega_r) = D\int S(\omega_i,\omega_r)L_i(\omega_i)\cos\theta_i d\omega_i \tag{6}$$

Da in Gleichung 5 der Durchmesser mit der Strahldichte multipliziert wird, bedeutet dies für die Haarfaser, dass sie heller erscheint, je größer der Durchmesser ist. Mit den vorangegangenen Arbeiten von Marcuse 1974, Adler et al. 1998 und Mount et al. 1998 lässt sich die Scattering-Function weiter umformen:

$$S(\phi_i,\theta_i;\phi_r,\theta_r) = \frac{1}{\cos^2\theta_d}\cdot M(\theta_i,\theta_r)N(\eta'(\theta_d);\phi_i,\phi_r) \tag{7}$$

Die Scattering-Function wird in Gleichung 7 in zwei 2D-Funktionen M und N unterteilt, wobei M die longitudinale und N die azimuthale Scattering-Funktion ist. Das Produkt der Funktionen wird durch $\cos^2\theta_d$ geteilt, um den Raumwinkel des spekularen Kegels zu projizieren. Durch die Annahme, dass der Durchschnitt einer Haarfaser rund ist, lässt sich die Funktion N auf einen Parameter reduzieren. Des Weiteren wird wegen der Symmetrie des Kreises nur die Differenz ϕ der Winkel ϕ_i und ϕ_r als Eingabeparameter benötigt.

$$M_P(\theta_h) = g(\beta_p;\theta_h - \alpha_p) \tag{8}$$

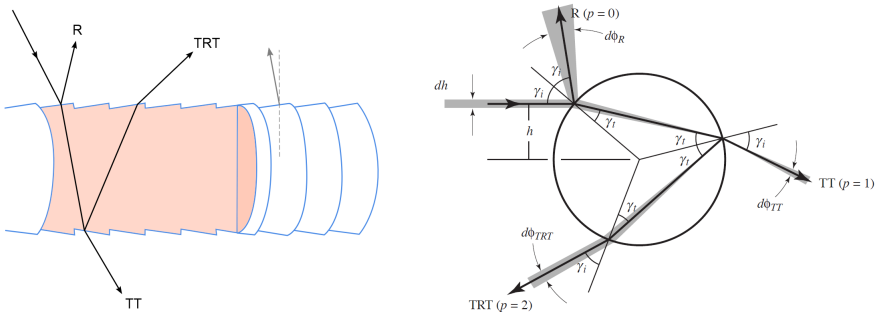
In Formel 8 ist $p \in P$ und $P = \{R = 0, TT = 1, TRT = 2\}$, welche den Pfad des Lichts beschreiben. Marschner et al. näherten mit einer Gauss-Funktion $g(x;\sigma^2)$ die longitudinale Scattering-Funktion an, nachdem sie Messungen analysiert hatten. Bei dem Winkel θ_h handelt es sich um den Mittelwert der Winkel θ_i und θ_r. Des Weiteren sei α_p der Winkel der longitudinalen Verschiebung und β_p der Winkel, der die longitudinale Breite (oder auch *Standard Abweichung*) beschreibt. Eine bildliche Beschreibung dieser Parameter findet sich in Kapitel 3.1 in Abbildung 15.

Nach den Gesetzen von Bravais und Fresnel lässt sich eine Gleichung zur Berechnung der azimuthalen Scattering-Funktion aufstellen:

$$N(\phi) = \sum_P N_p(p,\phi) \tag{9a}$$

$$N_p(p,\phi) = \sum_r A(p,h(p,r,\phi))|2\frac{d\phi}{dh}(p,h(p,r,\phi))|^-1 \tag{9b}$$

Die allgemeine Formel 9a zur Berechnung der azimutalen Lichtstreuungen summiert die für den Pfad spezielle Formel 9b zur Streuung über alle Pfade P auf. Zugrunde liegt der Gleichung die Verfolgung der Lichtstrahlen, wie sie in Abbildung 8b illustriert sind. In Formel 9b gibt die Funktion $h(p,r,\phi)$ den Abstand des Lichtstrahls zur Mitte des Kreises zurück, welche vom Pfad p,



(a) Die drei verschiedenen Lichtpfade: R-, TT- und TRT-Pfad

(b) Die Winkel der Lichtbrechungen innerhalb und außerhalb des Haares

Abbildung 8: Der Verlauf der Pfade laut Marschner et al.; [4]

von der Wurzel r und dem Winkel ϕ abhängig ist. Dieser Funktion zugrunde liegt die zu lösende Gleichung $\phi(p,h) - \phi = 0$, welche sich mittels der Winkel des reflektierten beziehungsweise transmitteten Strahls weiter zu $\phi(p,h) = 2p\gamma_t - 2\gamma_i + p\pi$ umformen lässt. Für $p = 0$ (R) und $p = 1$ (TT) ergibt sich eine Wurzel zur Lösung der Gleichung, bei $p = 2$ (TRT) sind es jedoch eine oder drei Wurzeln, also eine oder drei Pfade. Für Streuungsberechnungen müssen alle Pfade eines Winkels ϕ berechnet werden, da zu dem Winkel mehrere Höhen h existieren. Da das Licht mit jeder Reflexion oder Transmission durch Absorption an Intensität verliert, muss dieser Verlust in Formel 9b berücksichtigt werden. Dazu existiert der *Attenuation-Factor* $A(p,h)$:

$$A(0,h) = F(\eta', \eta'', \gamma_i) \tag{10a}$$

$$A(p,h) = (1 - F(\eta', \eta''\gamma_i))^2 F(\frac{1}{\eta'}, \frac{1}{\eta''}\gamma_t)^{p-1} T(\sigma_a, h)^p, \tag{10b}$$

wobei $T(\sigma_a, h)^p$ der Faktor der Absorption der Segmente innerhalb des Haares ist. Dabei steht $F(\eta', \eta'', \gamma)$ für die allgemeine Fresnel-Gleichung, welche in Marschner angepasst wurde.

2.3 Schatten

Zur Wahrnehmung von Objekten tragen Schatten maßgeblich bei, da sie einen räumlichen Eindruck vermitteln. Um Schatten in einer virtuellen Welt zu rendern, gibt es verschiedene Verfahren. Angefangen hatte es zunächst mit Punktschatten. So wird in *Tomb Raider* (1996) ein schwarzer Kreis unter der Spielfigur gerendert, siehe Abbildung 9.

Anstatt einen Kreis unter das zu beschattende Objekt zu rendern, etablierte sich in den darauf folgenden Jahren die Implementation planarer Schatten. Diese simulieren eine Lichtquelle, die paralleles Licht von oben auf das Ob-

Abbildung 9: Punktschatten unter *Lara Croft*, der Hauptfigur des Spiels *Tomb Raider* (1996), entnommen aus [4]

jekt ausstrahlt. Es wird also kein schwarzer Kreis unter das Szenenobjekt gerendert, sondern das Objekt wird auf die X-Z-Ebene projiziert.

Da sich die Lichtquellen in der realen Welt nicht konstant über Objekten befinden, wurden andere Verfahren entwickelt, mit denen es möglich ist angenäherte, physikalisch korrekte aussehende Schatten zu berechnen. Beim *Shadow Mapping*, das von Williams [5] im Jahr 1978 beschrieben wurde, handelt es sich um ein solches Verfahren.

Um realistische Schatten darzustellen, muss bekannt sein, welche Fragmente der Szene im Schatten liegen und welche nicht. Um diese Information zu erlangen wird in einem ersten Renderpass die Kamera in die Lichtquelle transformiert. Dabei wird nur in den *Depth Buffer* geschrieben. In einem zweiten Renderpass wird danach aus Sicht der ursprünglichen Kameraposition die Szene gerendert. Der zuvor beschriebene Depth Buffer wird benutzt, um die Tiefe des Fragments zu vergleichen: die Tiefe aus Sicht der Lichtquelle und aus Sicht der Kamera. Um diese unterschiedlichen Tiefen vergleichen zu können, muss projektives Texturmapping angewandt werden. Das betrachtete Fragment wird in das Koordinatensystem der Lichtquelle transformiert und daraufhin mit der *Texture-Bias-Matrix* multipliziert. Diese Matrix verschiebt die Werte des Viewports von $[-1..1]$ nach $[0..1]$, sodass auf die Textur, welche einen Bereich von 0..1 hat, zugegriffen werden kann. Da die Projektion die homogene Koordinate verändert, müssen anschließend x-, y- und z-Werte des Fragments durch die 4. Koordinate geteilt werden. Dadurch entstehen neue x-, y- und z- Werte für das betrachtete Pixel, wobei

[4] https://www.giantbomb.com/tomb-raider/3030-27312/user-reviews/2200-25718/, Stand 24.01.2018

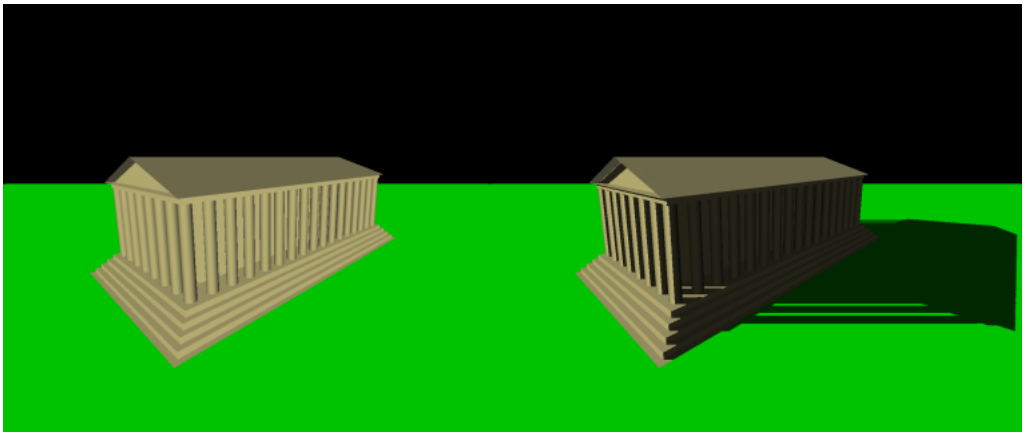


Abbildung 10: Szene ohne und mit Schatten. Entnommen aus [5]

x- und y-Werte als Texturkoordinaten verwendet werden können und der z-Wert nun die projizierte Tiefe speichert. Die aus der Textur ausgelesenen Tiefenwerte können nun mit dem z-Wert verglichen werden. Ist das Fragment im Schatten, so ist die Tiefe aus Sicht der Lichtquelle kleiner als die Tiefe aus Sicht der Kamera. Falls nicht, wird das Fragment beleuchtet. Dies resultiert in binären Schatten.

2.3.1 Deep Shadow Maps (Lokovic und Veach)

Für semi-transparente Oberflächen und volumetrische Objekte, wie beispielsweise Haare, Rauch oder Nebel, bietet sich das Shadow Mapping Verfahren nicht an. Diese Objekte blockieren nicht das gesamte Licht, sondern leiten und streuen es weiter in die Szene. Des Weiteren spielt die Selbstverschattung eine wichtige Rolle beim Rendern von semi-transparenten Objekten. Lokovic und Veach [6] entwickelten für diese Problemstellung im Jahre 2000 *Deep Shadow Maps*. Im Gegensatz zu Shadow Maps, die die Tiefe eines Pixels in einer Textur speichern, wird bei einer Deep Shadow Map für jedes Pixel eine *Visibility-Funktion* gespeichert. Diese gibt die ungefähre Menge an Licht an, das durch das Pixel einer gegebenen Tiefe dringt. Eine Deep Shadow Map ist also eine Textur oder ein 2D-Array aus Pixeln, das eine Visibility-Funktion pro Pixel speichert.

Der Wert der Funktion an einer gegebenen Tiefe z beschreibt einen Teil des Lichts aus der Lichtquelle, welches bis zur Tiefe z strahlt. Diese Funktion beschreibt auf welche Gegenstände der Strahl mit zunehmender Tiefe trifft und wie viel Lichtenergie dabei absorbiert wird. Eine Visibility-Funktion wird beschrieben durch:

[5] Von Praetor alpha aus der englischsprachigen Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9016280, Stand 08.02.2018

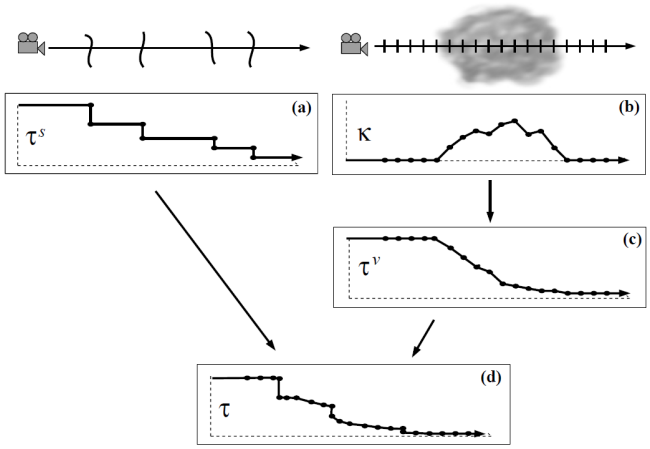


Abbildung 11: Transmittance Funktion nach Lokovic und Veach 2000 [6]

$$V_{i,j}(z) = \int_{-r}^{r}\int_{-r}^{r} f(s,t)\tau(i+\frac{1}{2}-s, j+\frac{1}{2}-t, z)\mathrm{d}s\mathrm{d}t \tag{11}$$

Dabei entspricht r dem Filterradius und $f(s,t)$ dem Filter-Kernel. Außerdem beschreibt $\tau(x,y,z)$ in Gleichung 11 die Transmittance-Funktion, wobei $(i+\frac{1}{2}, j+\frac{1}{2})$ das Pixel-Zentrum ist. Diese Transmittance-Funktion, vergleiche Abbildung 11d, gibt die Lichtdurchlässigkeit eines Punktes $P(x,y)$ zu einer gegebenen Tiefe z an. Da zur Berechnung der Visibility-Funktion eine Tiefe gegeben sein muss, lässt sich die Transmittance-Funktion in Glei-chung 11 als Lichtdurchlässigkeit aller Punkte innerhalb der Ebene zur Tiefe z beschreiben. Außerdem werden r^2 Transmitance-Funktionen gesampelt, welche mittels eines Filters gewichtet werden. Zur Berechnung einer Transmit-tance-Funktion müssen zunächst alle Schnittpunkte eines Punktes $P(x,y)$ mit dem Sehstrahl ermittelt werden. Bei den ermittelten Schnittpunkten unterscheidet man zwischen einem Oberflächen- und einem Volumenschnittpunkt. Somit lässt sich die Transmittance-Funktion aufteilen in:

$$\tau(z) = \tau^s(z) \cdot \tau^v(z) \tag{12}$$

In obiger Gleichung beschreibt $\tau^s(z)$ die *Oberflächen-Transmittance-Funktion*, siehe Abbildung 11a, und $\tau^v(z)$ die *Volumen-Transmittance-Funktion*, vergleiche Abbildung 11c. Die Oberflächen Transmittance-Funktion wird durch die Verwendung aller Oberflächenschnittpunkten eines Primärstrahls abgeschätzt. Anhand der Tiefe z_i^s und der Opazität O_i eines Schnittpunktes wird die Funktion, beginnend bei einer Transparenz von 1, mit der Transparenz des nächsten Schnittpunktes $1 - O_i$ multipliziert. Anschließend ist τ^s eine ungefähre konstante Funktion.

Um die Volumen-Transmittance $\tau^v(z)$ zu berechnen muss die atmosphärische Dichte gesamplet werden. Diese Dichte wird durch die Einteilung des

Sehstrahls in lineare Intervalle bestimmt. Jeder Sample-Punkt hat einen Tiefenwert z_i^v und einen *Extinction-Koeffizienten* K_i. Dieser beschreibt den Verlust des Lichts entlang des Strahls. Zwischen den Samples wird durch $T_i = \exp(-(z_{i+1}^v - z_i^v)(K_{i+1} + K_i)/2)$ die Transmittance linear interpoliert und somit ergibt sich mittels der *Extinction-Funktion* K (Abbildung 11b) die Volumen-Transmittance-Funktion $\tau^v(z)$:

$$\tau^v(z) = \exp(-\int_0^z K(z')dz') \tag{13}$$

Da die Visibility-Funktion $V_{i,j}(z)$ pro Pixel einer Tiefe z eine quadratische Anzahl n an umliegenden Transmittance-Funktionenen filtert, lässt sich Gleichung 11 umschreiben in:

$$V_{i,j}(z) = \sum_{k=1}^{n} w_k \tau_k(z) \tag{14}$$

Anstelle eines Filter-Kernels wie in Gleichung 11 wird in Gleichung 14 die normalisierte Filter-Gewichtung für jeden Sample-Punkt $P_{\text{Sample}}(x_k, y_k)$ mit der Transmittance-Funktion desselben Sample-Punktes multipliziert.

Da die Visibility-Funktion viel Speicherplatz benötigt, wird diese komprimiert: Es werden in einem Array floating-point Paare gespeichert, welche aus einem Tiefenwert und dem Ergebnis der Visbility-Funktion zu dieser gegebenen Tiefe bestehen. Bei der Komprimierung der Funktion muss eine Methode gewählt werden, welche die Tiefenwerte wichtiger Merkmale nicht beeinflusst, da bereits ein kleiner Fehler des Tiefenwerts Artefakte verursacht. Es muss also die Bedingung $z \in [0, \infty)$ gelten. Dazu werden ein *Maximum-Fehler* L_∞ und ein Greedy-Algorithmus verwendet. Es wird also aus einer gegebenen Visibility-Funktion $V(z)$ unter Berücksichtigung eines Fehlertoleranzwertes ϵ eine komprimierte Visbility-Funktion $V'(z)$ für alle Tiefen berechnet werden:

$$|V'(z) - V(z)| \leq \epsilon \tag{15}$$

Eine Eigenschaft von $V'(z)$ ist die geringere Anzahl an Kontrollpunkten im Vergleich zu $V(z)$. Der Algorithmus arbeitet in aufsteigender Tiefe inkrementell. Mittels Formel 15 wird bei jedem Schritt das längstmögliche Liniensegment, welches innerhalb der Fehlergrenze liegt, durch den Algorithmus eingetragen. Der Ursprung des momentan betrachteten Liniensegments wird dabei korrigiert und es müssen zur weiteren Berechnung die Richtung und Länge des Liniensegments bestimmt werden. Um diese Problemstellung zu vereinfachen, werden die Output-Werte der Tiefe auf einen kleinen Teil der Input-Werte begrenzt.

Für jeden Ursprung des neuen Liniensegments wird der Bereich der erlaubten Steigungen $[m_{lo}, m_{hi}]$ des Segments geprüft. Jeder neue Kontrollpunkt (z_j, V_j) bestimmt eine neue Begrenzung des aktuellen Steigungsin-

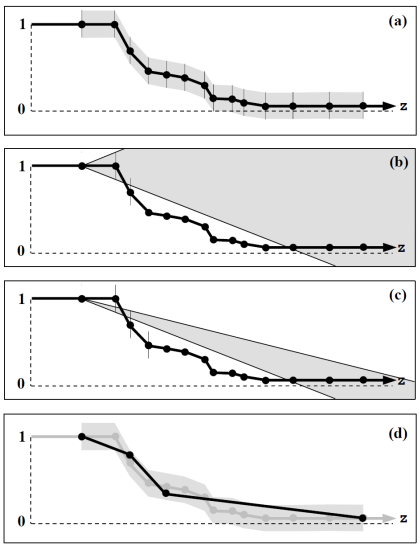


Abbildung 12: Kompression der Visibility Funktion; [6]

tervalls: Jedes Segment wird dabei durch das sogenannte *Target Window* gezwängt, welches sich zu zwei Punkten $(z_j, V_j \pm \epsilon)$ ausdehnt. Nachdem der Algorithmus inkrementell abgearbeitet wurde, ergibt sich eine komprimierte Visibility-Funktion $V'(z)$ mit deutlich weniger Kontrollpunkten, die dennoch die ursprüngliche Funktion in den wichtigsten Stellen widerspiegelt (vergleiche Abbildung 12).

Um eine Deep Shadow Map nach Lokovic und Veach [6] zu benutzen, muss zunächst die Szene rekonstruiert und gesamplet werden. Durch die Visibility-Funktion, die mit der Tiefe z ausgewertet werden kann, ist es möglich, die Deep Shadow Map zu konstruieren. Mit einem gegebenen Punkt (x, y, z) wird der gefilterte Schattenwert berechnet:

$$V(x, y, z) = \frac{\sum_{i,j} w_{i,j} V_{i,j}(z)}{\sum_{i,j} w_{i,j}} \tag{16}$$

Hierbei ist $w_{i,j}(x, y) = f(i + 0.5 - x, j + 0.5 - y)$ das gefilterte Gewicht des Pixels (i, j). Die Summen beschränkten sich auf den Filterradius.

Durch das von Lokovic und Veach entwickelte Verfahren können volumetrische Objekte wie beispielsweise Haare realistisch schattiert werden. Wegen der hohen Anzahl an Berechnungen, die zur Erstellung einer Deep Shadow Map nötig sind, ist dieses Verfahren jedoch nicht echtzeitfähig. Doch war es eines der ersten Schattierungsverfahren, das nicht die Tiefe, sondern die Visibility speicherte.

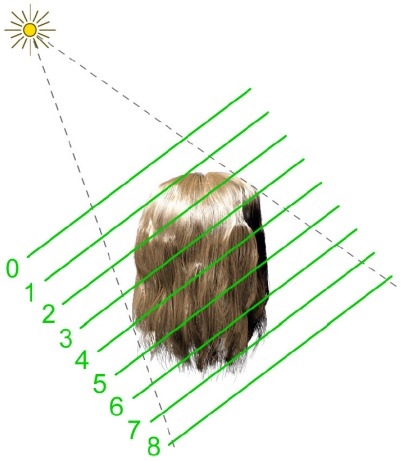


Abbildung 13: Opacity Maps, entnommen aus [8]

2.3.2 Opacity Maps (Kim und Neumann)

Auf dem Verfahren der Deep Shadows Maps von Lokovic und Veach [6] baut die im Jahre 2001 von Kim und Neumann [7] vorgestellte Methode zur Berechnung der Schatten von volumetrischen Objekten. In den von ihnen vorgestellten *Opacity Maps* wird jedoch keine Visibility-Funktion gespeichert, sondern die Opazität Ω. Die in Opacity Maps verwendete Transmissions-Funktion gleicht der volumetrischen Transmissions-Funktion (13) in Kapitel 2.3.1. Sie lässt sich zu einem gegebenen Punkt p berechnen durch:

$$\tau(p) = \exp(-\Omega) \tag{17}$$

Die Opazität beschreibt dabei den Teil des Lichts, welcher durch das volumetrische Objekt transmittiert wird und dabei an Energie verliert. Sie lässt sich mittels einer Dichte-Funktion $\rho(l')$ beschreiben durch:

$$\Omega = \int_0^l \rho(l')dl' \tag{18}$$

Die von Kim und Neumann vorgestellten Opacity Maps bestehen aus mehreren einzelnen Opacity Maps, welche senkrecht zum Lichtstrahl stehen. Diese werden in Abbildung 13 illustriert.

In jede dieser Opacity Maps wird die Opazität eines Pixels zu einer Tiefe, welche durch die Tiefe der Map gegeben ist, gespeichert. So ergibt sich eine über Texturen angenäherte Sichtbarkeits-Funktion. Bei der Berechnung von Schatten werden benachbarte Opacity Maps gesamplet und zwischen ihnen interpoliert. Es kann eine beliebige Anzahl an Maps verwendet werden.

Ein Vorteil dieses Verfahrens ist, dass es mit Punkten, Linien und Polygonen verwendet werden kann. Nachteile der Methode sind Artefakte, die

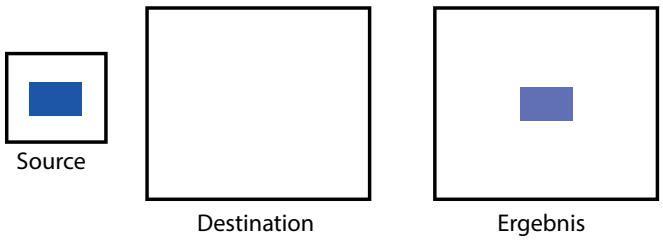


Abbildung 14: Blending

bei einer zu geringen Anzahl an Texturen auftreten können. Bei einer höheren Anzahl von Schichten, beispielsweise 256, ist es möglich, dass Artefakte auftreten können, doch sind bei einer so hohen Anzahl an Schichten nur ca. 2.3 fps möglich.

2.4 Transparenzen und Blending

Da Szenen der realen Welt oftmals semi-transparente Objekte enthalten, gibt es eine Annäherung für diese Materialeigenschaft in der Computergrafik. Dabei hat die Reihenfolge, in der die Objekte gerendert werden, eine wichtige Bedeutung. Zuerst müssen alle opaken Objekte von vorne nach hinten, und danach die transparenten Objekte von hinten nach vorne gerendert werden. Im Verfahren nach Porter and Duff [9] wird die finale Farbe eines Pixels C_f einer Szene (vergleiche Abbildung 14), bestehend aus einer Hintergrundfarbe C_0 und einer Vordergrundfarbe C_1, durch die Bedeckung der vorderen Oberfläche α_1 berechnet:

$$C_f = C_1 + (1 - \alpha_1)C_0 \tag{19}$$

Nach [9] wird Gleichung 19 auch als *OVER*-Operator für *vormultiplizierte Coverage* bezeichnet. Falls der Grad der Verdeckung $alpha_1 = 1$, so bedeckt die vordere Fläche den Hintergrund. Wenn $alpha_1 = 0$, dann wird die Hintergrundfläche nicht von der Vordergrundfläche bedeckt, da das Licht durch die vordere Fläche auf den Hintergrund strahlt. Diese Methode zur Berechnung von Transparenzen wird auch *Back to Front Alpha Blending* oder *Compositing* genannt.

Zum Rendern von Haaren ist dieses Verfahren jedoch nicht geeignet, da die Reihenfolge des Renderings wichtig ist. So würde eine Szene mit 10 000 Haaren einen hohen Zeit- und Speicherverbrauch zum Sortieren der einzelnen Haare benötigen. Um Transparenzen von komplexen Szenen zu berechnen, wurden Verfahren zur *Order Independet Transparency* (OIT) vorgestellt. Diese Methoden benötigen keine Sortierung der transparenten Objekte und reduzieren so den Zeit- und Speicherverbrauch. Eines dieser Verfahren ist beispielsweise *Depth Peeling* von Everitt [10] aus dem Jahr 2001, in welchem die Szene entlang der Tiefe in Schichten unterteilt wird. Ein anderes

Verfahren von Barta et al. [11], welches im Jahr 2011 veröffentlicht wurde, beschreibt die Erstellung einer *Per-Pixel Linked-List*. Diese teilt die Szene in eine gegebene Anzahl von Schichten auf. Jeder Eintrag eines Pixels der Liste, speichert dabei eine Referenz auf das Pixel, welches sich dahinter befindet. Um diese Liste dynamisch zu erstellen, veröffentlichten Yang et al. [12] ihre Methode zur echtzeitfähigen Konstruktion einer Per-Pixel-Linked-List. Ein Verfahren zur Berechnung von Transparenzen durch die Gewichtung der Tiefe wird in Kapitel 3.3 genauer beschrieben.

Durch Blending lassen sich nicht nur Transparenzen berechnen, sondern es lässt sich zum Beispiel auch das Licht simulieren, welches durch ein Objekt scheint. Dafür muss die Blending-Funktion angepasst werden. Einige Verfahren wie *Deep Opacity Maps* [8] oder *Depth Weighted OIT* verwenden additives Blending zur Simulation des Lichts.

3 Rendering

Im Folgenden werden alle für die Implementation genutzten Verfahren erläutert. Diese sind *An Artist Friendly Hair Shading System* [13] von Sadeghi et al. zur Berechnung der Beleuchtung, *Deep Opacity Maps* [8] von Yuksel und Kayser zur Schattenberechnung und *Depth Weighted OIT* [14] von McGuire und Bavoil zur Transparenzberechnung.

3.1 Beleuchtung nach Sadeghi et al.

Da das Beleuchtungsmodell für Haare nach Marschner et al. energie-erhaltend ist, können die Attribute nicht korrekt verändert werden, da es sonst möglich ist, dass mehr Energie im System ist als zuvor. Sadeghi et al. untersuchten das Modell von Marschner et al. [4], um dieses in ein für Grafiker benutzerfreundliches Modell zu überführen. Das System sollte nun also nicht mehr über „Physically Based Controls“ gesteuert werden, sondern durch „Artist Friendly Controls“. Zu diesem Zweck wurden Grafiker befragt, welche Eigenschaften der Haare sie einstellen möchten: es sollten primäres und sekundäres Highlight, Glints und Rim Light einstellbar sein. Um eine Annäherung zu finden, wurde das Licht, wie bei Marschner, in drei Pfade aufgeteilt: R-, TT- und TRT-Pfad.

Für den Pfad des Lichts, welches direkt an der Haaroberfläche reflektiert wird – also den R-Pfad – sollten die Farbe, Intensität, longitudinale Position und Breite einstellbar sein, ebenso wie für den TT-Pfad, der zusätz-lich eine Einstellung der azimuthalen Breite ermöglicht. Der TRT-Pfad sollte aufgeteilt werden in *TRT-Pfad ohne Glints*, bei welchem die Farbe, Intensität, longitudinale Position und Breite einstellbar sind, und *Glints*, bei welchen die Farbe, Intensität und die Frequenz der Vorkommens eingestellt werden können. Je nachdem welcher Grafiker befragt wird, unterscheiden sich die gewünschten Einstellungen.

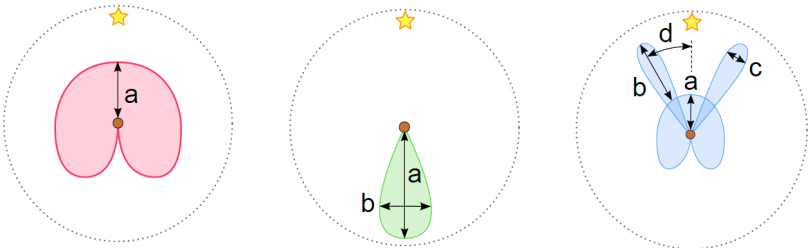
Als Ausgangsfunktion gilt dabei die von Marschner et al. aufgestellte Gleichung:

$$f_s(\theta_h, \phi) = \sum_P \frac{1}{cos^2(\theta_d)} M_p(\theta_h) N_p(\phi) \tag{20}$$

Um ein Beleuchtungsmodell mit benutzerfreundlichen Einstellungen zu modellieren, muss Formel 20 in ihre Bestandteile zerlegt werden, um sie umzuschreiben. So stellen Sadeghi et al. eine abgewandelte Funktion vor, mit der die longitudinale Streuung berechnet werden kann:

$$M'_p(\theta_h) = g'(\beta_p; \theta_h - \alpha_p) \tag{21}$$

Hierbei ist zu beachten, dass es sich nicht um dieselbe Gaussfunktion wie in Kapitel 2.2.2 handelt, denn der Mittelwert dieser Funktion ist 0 und



(a) R-Pfad **(b)** TT-Pfad **(c)** TRT-Pfad

Abbildung 15: Visualisierung der Berechnungen nach Sadeghi et al., wobei a die Intensität I_p, b bei TT die azimuthale Breite ξ_{TT}, bei TRT ist b die relative Intensität I_g, c der longitudinale Shift α_p und d der Halbwinkel zwischen den Glints beschreibt, entnommen aus [13]

die Höhe beträgt 1. Wie in Marschner stehen β_p und α_p für die longitudinale Breite beziehungsweise für den longitudinalen Shift. Während es eine Funktion zur Berechnung der longitudinalen Streuung gibt, existieren vier Funktionen zur Berechnung der azimuthalen Streuung:

$$N'_R(\phi) = \cos(\frac{\phi}{2}) \tag{22a}$$

$$N'_{TT} = g'(\xi_{TT}, \pi - \phi) \tag{22b}$$

$$N'_{TRT-G} = \cos(\frac{\phi}{2}) \tag{22c}$$

$$N'_G = I_g g'(\xi_g; G_{angle} - \phi) \tag{22d}$$

$$N'_{TRT} = N'_{TRT-G} + N'_G \tag{22e}$$

In den Gleichungen 22b und 22d beschreiben ξ_{TT} und ξ_g die azimuthale Breite für den jeweiligen Pfad. Außerdem beschreibt I_g in Formel 22d die relative Intensität der Glints und die Intensität des sekundären Highlights und G_{angle} steht für den Halbwinkel der Glints, welcher für jedes Haar unterschiedlich ist, und zwischen 30° und 45° liegt. Eine visuelle Beschreibung der Parameter findet sich in Abbildung 15 und bei Sadeghi et al. [13].

Anhand der aufgestellten Formeln zur Berechnung der longitudinalen und azimuthalen Streuung lässt sich eine allgemeine Streuungsfunktion nach Sadeghi et al. aufstellen:

$$f'_s(\theta_h, \phi) = \sum_P f'_p / \cos^2(\theta_d) \tag{23}$$

$$f'_p(\theta_h, \phi) = C_p I_p M'_p(\theta_h) N'_p(\phi) \tag{24}$$

Durch die in Gleichung 24 hinzugefügten Variablen wie C_p und I_p lassen sich Farbe und Intensität des jeweiligen Pfades einstellen. Somit ist es

nun möglich, die Farbe direkt zu übergeben, sodass die Einstellungen zum Rendern von Haaren benutzerfreundlicher sind.

3.2 Schatten nach Yuksel und Keyser (Deep Opacity Maps)

Die in Kapitel 2.3 vorgestellten Methoden zur Schattierung von Haaren speichern keine Shadow Map, sondern eine Sichtbarkeits-Funktion beziehungsweise die Opazität. Auf die von Kim und Neumann [7] entwickelten Opacity Maps bauen Yuksel und Keyser [8] im Jahre 2008 auf.

In den vorgestellten *Deep Opacity Maps* (DOM) wird die Opazität eines Pixels gespeichert. Zur Erstellung werden drei Render-Passes benötigt. So wird im Ersten eine Shadow Map erzeugt, im Zweiten wird die Deep Opacity Map generiert und im Dritten wird das Objekt mit Hilfe der Deep Opacity Map schattiert.

Im Gegensatz zu Opacity Maps, benötigen Deep Opacity Maps nur drei Schichten, welche die Opazität speichern. Dazu reicht eine RGBA-Textur bereits aus, da jede Schicht einem Kanal zugeordnet wird. Außerdem ist ein Kanal für die Tiefe bestimmt, weshalb in einer RGBA-Textur Informationen zu drei Schichten gespeichert werden können. Diese Schichten werden nicht orthogonal zum Lichtstrahl gewählt, sondern sind abhängig von der Tiefe aus Sicht der Lichtquelle. Dies hat den Effekt, dass die Grenzen zwischen den Schichten die Form der Haargeometrie annehmen, vergleiche dazu Abbildung 16.

Es ist auch möglich, mehr als drei Schichten als Deep Opacity Map zu nutzen. Dazu müssen mehrere Draw-Buffer verwendet werden, wobei n Buffer $4n - 1$ Schichten erlauben. Allerdings tragen mehr als drei Schichten nicht zu einer Verbesserung der Qualität des Ergebnisses bei.

Im Vergleich zu Shadow Maps, welche wegen der binären Entscheidungen eine hohe Auflösung der Textur benötigen, reichen für Deep Opacity Maps bereits 8-Bit-Texturen aus. Es gibt keinen signifikanten Unterschied der gerenderten Haare zwischen einer 8- und 16-Bit-Textur.

Bei der Generierung wird in einem zweiten Render-Pass die Tiefe aus der zuvor berechneten Shadow Map gelesen und anhand dieser werden die weiteren Grenzen z_1 und z_2 der Schichten berechnet:

$$z_1 = z_0 + d$$
$$z_2 = z_1 + d$$

Für jeden Vertex der Geometrie wird nun in der Deep Opacity Map die Opazität gespeichert. Dazu wird der Vertex in das Licht-Koordinatensystem transformiert und mit einer Bias-Matrix multipliziert, damit korrekte Texturzugriffe möglich sind. Die ausgelesene Tiefe des Vertex z_p wird nun mit

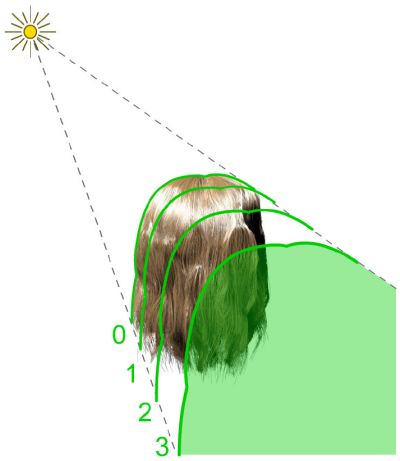


Abbildung 16: Unterteilung der Haargeometrie in drei Schichten nach Yuksel. Der eingefärbte Bereich wird der letzten Schicht zugeordnet. Entnommen aus [8]

den Grenzen verglichen:

$$\mathrm{DOM}(z_p) = \begin{cases} 1.\text{ Schicht, wenn } z_0 \leq z_p < z_1 \\ 2.\text{ Schicht, wenn } z_1 \leq z_p < z_2 \\ 3.\text{ Schicht, wenn } z_2 \leq z_p \end{cases}$$

Dabei zu beachten ist, dass die Opazität eines Pixels eines Layers die Summe aller beitragenden Fragmente ist. Fällt ein Pixel also in die ersten Schicht, so wird die Opazität nicht nur in die erste, sondern in alle dahinter liegenden Schichten eingetragen, vergleiche dazu Listing 3 .

Yuksel und Keyser ist bei der Grenzbehandlung von Vertices, welche außerhalb der letzten Schicht liegen, aufgefallen, dass diese der letzten Schicht hinzugefügt werden können. Die gesamte Opazität einer Schicht ist die Summe aller beitragenden Fragmente. Dies gleicht additivem Blending, welches in OpenGL durch die Blending-Funktion `GL_BlendFunc(GL_ONE, GL_ONE)` aktiviert wird.

Zur Schattierung wird aus Sicht der Kamera, also des Betrachters, gerendert. Dabei wird jedes betrachtete Fragment ins Licht-Koordinatensystem transformiert und erneut mit einer Bias-Matrix multipliziert. Die Schicht des Fragments wird ermittelt, und dann zwischen den Schichten interpoliert. Anschließend wird der Schatten-Faktor durch Formel 17 berechnet und mit der Haarfarbe multipliziert.

3.3 Transparenzen nach McGuire und Bavoil

Um die Transparenz von Haaren zu berechnen, muss ein OIT-Verfahren verwendet werden, da das Sortieren der Haare zeitintensiv ist. In der 2013

vorgestellten Methode von McGuire und Bavoil [14] wird auf dem Verfahren der *blended Order-Independent-Transparency* aufgebaut. Bei diesen wird der Compositing Operator vermengt (engl. blended). Da nur der Oparator angepasst wird, sind blended-OIT-Verfahren simpel gehalten. Der von McGuire und Bavoil [14] vorgestellte Compositing Operator baut auf früheren blended-OIT-Verfahren auf und erweitert diese durch eine Gewichtungsfunktion $w(z_i, \alpha_i)$:

$$C_f = \frac{\sum_{i=1}^{n} C_i \cdot w(z_i, \alpha_i)}{\sum_{i=1}^{n} \alpha_i \cdot w(z_i, \alpha_i)} (1 - \prod_{i=1}^{n} (1 - \alpha_i)) + C_0 \prod_{i=1}^{n} (1 - \alpha_i) \tag{25}$$

Dabei verhält sich die Gewichtungsfunktion ähnlich zu einer Visibility-Funktion und beschreibt den Verlust des Lichts durch Objekte der Szene. Der Beitrag nimmt also ab, wenn ein Lichtstrahl durch ein Objekt fällt. Dies beschreibt die Transparenz eines volumetrischen Objekts wie Haare oder Rauch, ohne dass eine Sortierung der Szenenobjekte notwendig ist. Die Gewichtungsfunktion ist dabei von der Tiefe z_i und dem Coverage-Wert $alpha_i$ abhängig. Dadurch werden Haare, die nah an der Kamera sind, jedoch einen geringen Coverage-Wert besitzen, kaum wahrgenommen. Dieser Effekt lässt sich in der realen Welt beobachten.

4 Implementation

Das folgende Kapitel beschreibt die Implementation der im vorangegangenen Kapitel erläuterten Verfahren zum Rendern von Haaren. Abbildung 17 vermittelt eine Übersicht der programmierten Rendering-Pipeline.

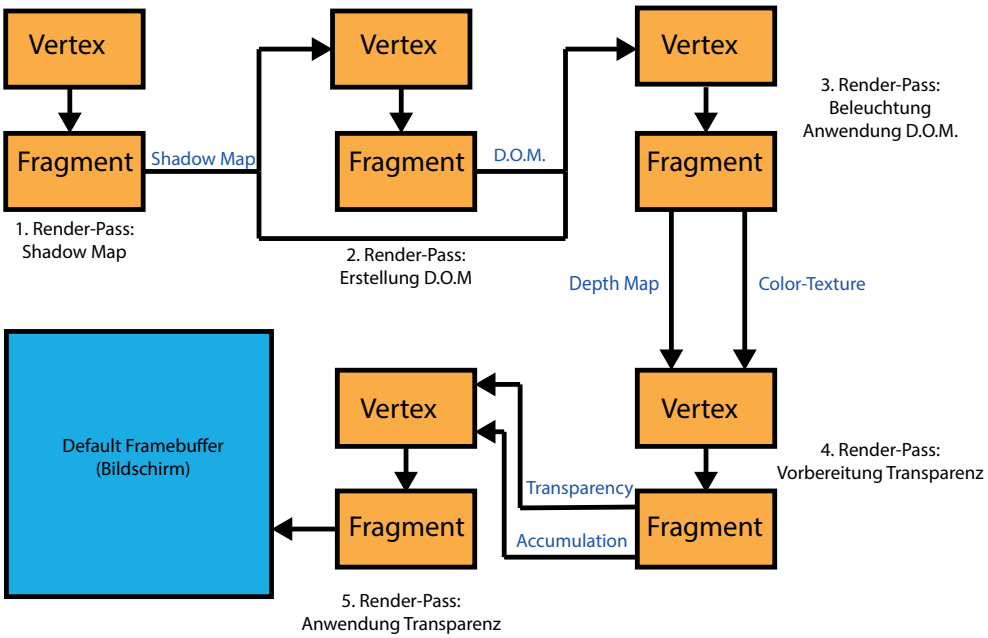


Abbildung 17: Rendering-Pipeline

4.1 Geometrien

4.1.1 Behaarung von Objekten

Um Testszenen für Fell zu erstellen, wird ein Shader verwendet, welcher aus Vertex-, Tesselation- und Geometry-Shader besteht. Dieser generiert aus den Normalen eines gegebenen Objekts Haare. Dazu wird das Objekt im Vertex-Shader durch Model- und View-Matrix transformiert, sodass die Punkte im Kamerakoordinatensystem liegen. Anschließend wird der Tesselation-Shader ausgeführt, welcher die Geometrie höher auflöst. Dadurch werden auch mehr Normalen, also mehr Haare, erstellt. Die neu erstellten Vertices und Normalen werden an den Geometry-Shader übergeben, siehe Listing 1 Zeile 4 und 5.

```
1  layout(triangles) in;
2  layout(line_strip, max_vertices = 10) out;
3
4  in vec3 tesPos[];
5  in vec3 tesNorm[];
6
7  out vec4 g_passPosition;
8  out vec4 g_passTangent;
9
10 void main() {
11
12   vec3 pos = (tesPos[0] + tesPos[1] + tesPos[2])/3.0f;
13   vec3 normal = (tesNorm[0] + tesNorm[1] + tesNorm[2])/3.0f;
14
15   vec4 mid = vec4(pos, 1.0f);
16   vec4 dir = vec4(normal, 0.0f);
17   vec4 down = viewMatrix * vec4(0.0f, -0.5f, 0.0f, 0.0f);
18
19   float len = 0.03f;
20   vec4 prev = mid;
21   g_passPosition = projMatrix * prev;
22   gl_Position = projMatrix * prev;
23   EmitVertex();
24
25   for(int i = 0; i < 10; i++)
26   {
27         vec4 point = prev + dir * len;
28         dir += down * len;
29         g_passTangent = point-prev;
30         g_passPosition = projMatrix * point;
31         gl_Position = projMatrix * point;
32         EmitVertex();
33         prev = point;
34   }
35 EndPrimitive();
36 }
```

Listing 1: Geometry Shader zur Behaarung von Objekten

Der oben dargestellte Geometry-Shader erhält als Primitiv-Input Dreiecke und wandelt diese in Linien um, welche aus zehn Vertices bestehen (Zeile 1 und 2). Da zum Beleuchten der Haare keine Normalen, sondern Tangenten benötigt werden, berechnet der Shader diese. Um die Haare zu generieren werden der Mittelpunkt und die gemittelte Normale des Dreiecks berechnet. Der Punkt wird als Startpunkt des Haares verwendet (Zeile 22) und mit

der Projektions-Matrix multipliziert. Die Normale wird verwendet, um die nächsten Punkte des Haares zu berechnen. Innerhalb der `for`-Schleife werden die übrigen Punkte des Primitivs berechnet, während die Richtung des Haares mit jedem Durchlauf mehr nach unten abgelenkt wird. Außerdem werden die Tangenten berechnet und gespeichert.

4.1.2 HAIR-Files von Yuksel

Auf der Internetseite[6] von Cem Yuksel, der sich in der Computergrafik unter anderem mit dem Rendering von Haaren beschäftigt, finden sich Haargeometrien, welche im HAIR-Format gespeichert werden. Dieses Format wurde von Yuksel entwickelt.

Eine HAIR-Datei kann aus mehreren Arrays bestehen, muss jedoch mindestens ein *Points-Array* enthalten. In diesem Array stehen die 3D-Positionen der Haare, wobei kein Punkt doppelt verwendet wird. Die Punkte eines einzelnen Haares sind im Array hintereinander eingetragen, sodass diese von Haarwurzel nach Haarspitze geordnet sind. Bei den im Array gespeicherten Positionen handelt es sich um 32-Bit-Float Werte, die x-, y- und z-Werte der Position stehen demnach hintereinander. Somit hat das Array eine Länge von $3 \cdot \text{num(points)}$.

Durch das Points-Array lassen sich die Tangenten der Haare berechnen, welche die Verbindung zwischen den Haar-Vertices beschreibt. Somit sind alle zum Rendern wichtigen Informationen vorhanden.

Die optionalen Arrays einer HAIR-Datei sind: *Segments-Array*, *Color-Array*, *Transparency-Array* und *Thickness-Array*. Das Segments-Array beinhaltet die Anzahl der Segmente eines einzelnen Haares, welche als 16-Bit unsigned Integer gespeichert werden. Dadurch kann ein Haar aus bis zu 65.536 Segmenten bestehen. Die Länge dieses Arrays gibt die Anzahl an Haaren wieder.

Die Farbe eines Punktes wird als 32-Bit-Float im Color-Array gespeichert. Dabei stehen die Werte der Farbe, ähnlich wie im Points-Array, hintereinander und sind von Wurzel nach Haarspitze sortiert. Da die Farbe eines Punktes aus RGB-Werten besteht, ist die Länge des Arrays gleich der Länge des Points-Array. Die Dicke eines Haares wird im Thickness-Array gespeichert, wobei die Länge des Arrays der Anzahl der Punkten gleicht. Das letzte optionale Array, das Transparency-Array, beschreibt die Transparenz eines Punktes. Es handelt sich um 32-Bit-Float-Werte und es wird pro Punkt ein Wert gespeichert. Somit hat das Array dieselbe Länge wie beispielsweise das Transparency- oder Thickness-Array.

Um das Format zu benutzen, muss die Klasse `cyHairFile` dem Projekt hinzugefügt werden. Des Weiteren wurde eine Geometrie-Klasse implementiert, welche die VBOs und das VAO befüllt und an die Grafikkarte übergibt.

[6] www.cemyuksel.com/research/hairmodels, Stand 05.02.2018

4.2 Beleuchtung

Für die Beleuchtung wurde das in Kapitel 3 vorgestellte Verfahren nach Sadeghi et al. [13] implementiert, wobei der diffuse Term durch eine Annäherung zu Kajiya und Kay [3] in Zeile 3 berechnet wird.

Um den spekularen Term zu bestimmen, werden Lichtvektor $\vec{l}$ und Viewvektor $\vec{v}$ in die Normal-Plane projiziert, welches durch das Skalarprodukt in Zeile 5 und 6 in Listing 2 berechnet wird. Für eine genaue Beschreibung der Winkel vergleiche Abbildung 7. In Zeile 11 und 12 werden die Winkel zwischen Licht- bzw. Viewvektor und Normalplane berechnet, welche für die Bestimmung des Halbwinkels ω_h und des Differenzwinkels ω_d benötigt werden.

```
1  void main(){
2
3    vec4 diffuse = 0.4 * sqrt( 1 - dot(L,T) * dot(L,T)) * col;
4
5    float sin_theta_i = dot(L,T);
6    float sin_theta_o = dot(V,T);
7
8    vec3 lightVecPerp = normalize(L - sin_theta_i * T);
9    vec3 eyeVecPerp = normalize(V - sin_theta_o * T);
10
11   float theta_i = asin(sin_theta_i);
12   float theta_o = asin(sin_theta_o);
13   float theta_h = (theta_i + theta_o)/ 2;
14   float theta_d = (theta_o - theta_i) / 2;
15
16   float phi = acos(min(1.0,dot(eyeVecPerp,lightVecPerp)));
17
18   M.x = gauss(theta_h - longi_shift.x, longi_width.x);
19   M.y = gauss(theta_h - longi_shift.y, longi_width.y);
20   M.z = gauss(theta_h - longi_shift.z, longi_width.z);
21
22   N.x = cos(phi/2);
23   N.y = gauss(PI - phi,azi_width);
24   N.z = N.x + intensity * gauss(glint_angle - phi,azi_width);
25
26   float proj_angle = 1.0/pow(cos(theta_d),2);
27   vec4 specular = R * proj_angle + TT * proj_angle
28                 + TRT * proj_angle;
29   fragmentColor = diffuse + specular;
30 }
```

Listing 2: Fragment Shader zur Beleuchtung von Objekten

Durch θ_h lässt sich die longitudinale Scattering-Function M bestimmen, welche für die jeweiligen Pfade (R, TT, TRT) bestimmt wird. In Listing 2 steht `.x` für den R-Pfad, `.y` für den TT-Pfad und `.z` für den TRT-Pfad. Durch den Winkel ϕ lässt sich die azimuthale Scattering-Function berechnen. Die Farbe eines Pfades lässt sich bestimmen durch:

$$Color_p = c_p \cdot i_p \cdot M_p \cdot N_p \ ,$$

wobei c_p die Farbe und i_p die Intensität des Highlights ist. Abschließend berechnet sich die finale spekulare Farbe aus der Summe der pfadspezifischen Produkte aus $Color_p$ und `proj_angle` (Zeile 26 und 27).

4.3 Schatten

Wie bereits in vorangegangenen Kapiteln erläutert, spielen Schatten eine wichtige Rolle in der Computergrafik. So wurden *Deep Opacity Maps* nach Yuksel und Keyser [8] implementiert. Dazu werden zwei Render-Passes benötigt: einen, um die Shadow Map zu erstellen, und einen, um die Deep Opacity Map zu berechnen.

Im ersten Render-Pass wird die Kamera in die Lichtquelle gesetzt und aus Sicht dieser gerendert. Die dabei abgespeicherte Shadow Map wird als Textur abgespeichert und dem nächsten Render-Pass als Input-Textur beigefügt.

```
1  in vec4 posLightSpace;
2
3  uniform sampler2D shadowMap;
4
5  out vec3 deepOpacityMap;
6
7  void main(){
8    vec3 projCoords = posLightSpace.xyz / posLightSpace.w;
9    float z0 = texture(shadowMap, projCoords.xy).x;
10   float z1 = z0 + layerSize;
11   float z2 = z1 + layerSize;
12   float currDepth = projCoords.z + 0.001;
13
14   if(z0 <= currDepth)
15     deepOpacityMap.x = alpha;
16   if(z1 <= currDepth)
17     deepOpacityMap.y = alpha;
18   if(z2 <= currDepth)
19     deepOpacityMap.z = alpha;
20 }
```

Listing 3: Fragment Shader zur Erstellung einer Deep Opacity Map

Um eine Deep Opacity Map zu erstellen muss korrekt auf die dem Shader übergebene Shadow Map zugegriffen werden, vergleiche Listing 3 Zeile 8. Aus der ausgelesenen Tiefe und einer gegebenen *Layer Size* lassen sich die Grenzen z_0, z_1 und z_2 berechnen. Dabei kann die Größe des Layers linear, aber auch logarithmisch gewählt werden. Im letzten Schritt der Erstellung wird die aktuelle Tiefe (Zeile 14) gegen die Schichten getestet. Falls das betrachtete Fragment innerhalb einer Schicht liegt, so wird der Alpha-Wert für den korrespondierenden Kanal gesetzt. Liegt ein Fragment innerhalb der ersten Schicht, so wird im R-Kanal der Wert eingetragen. Falls ein Fragment in einer tieferen Ebene liegt, so wird der Alpha-Wert auch in den vorangegangenen Schichten eingetragen.

Damit eine Deep Opacity Map zum Schattieren benutzt werden kann, wird diese im nächsten Render-Pass als Input-Textur verwendet.

```
1  float calcShadowFactor(){
2    vec4 opValue = texture(deepOpacityMap, projCoords.xy);
3
4    //Layer-Berechnung wie in Listing 3
5
6    int maxLayers;
7
8    if(z0 <= currDepth && currDepth < z1)
9      maxLayers = 1;
10   if(z1 <= currDepth && currDepth < z2)
11     maxLayers = 2;
12   if(z2 <= currDepth)
13     maxLayers = 3;
14
15   for(int layer = 1; layer < maxLayers; layer++){
16     t = clamp((currDepth-layerStart)/layerSize, 0.0, 1.0);
17     occlusion += mix(opValue[layer-1], opValue[layer],t);
18     layerStart += layerSize;
19   }
20   return exp(-occlusion);
21 }
```

Listing 4: Funktion zur Schattenberechnung

Durch den Zugriff auf die Deep Opacity Map lässt sich der Opazitätswert für ein gegebenes Fragment bestimmen. Damit korrekt schattiert werden kann, muss für dieses Fragment das korrespondierende Layer bestimmt werden. In Listing 4 ist dies durch die Variable `maxLayers` gegeben, welche innerhalb der Zeilen 8 bis 13 gesetzt wird. Da das korrespondierende Layer nun bekannt ist, wird die *Occlusion* zwischen den Schichten interpoliert. Abschließend wird der Shadow Factor durch die in Kapitel 2.3.2 vorgestellte

Formel 18 zurückgegeben. Zur Anwendung des Shadow Facotrs wird dieser bei der Farbgebung mit der Farbe multipliziert. Ein Beispiel für einer Deep Opacity Map findet sich in Kapitel A in Abbildung 27.

4.4 Transparenzen

Da nach dem Verfahren von McGuire [14] die Farbe sowie die Tiefe benötigt wird, wird im Render-Pass zur Beleuchtung in einem FBO eine Farbtextur und eine Tiefentextur abgespeichert. Im ersten Render-Pass werden die Transparenzen berechnet und im zweiten Pass wird die finale Farbe mit Transparenz gesetzt.

Da im ersten Render-Pass in zwei Texturen gerendert wird, welche unterschiedliche Blending-Funktionen und Clear-Colors verwenden, müssen diese durch OpenGL-Befehle gesetzt werden. Um einem FBO mit zwei Color-Buffern unterschiedliche Blending-Funktionen und Clear-Colors zu übergeben werden diese gesetzt durch:

```
glBlendFunci(0, GL_ONE, GL_ONE);

glBlendFunci(1, GL_ZERO, GL_ONE_MINUS_SRC_ALPHA);

glClearBufferfv(GL_COLOR, 0, clearAccumColor);

glClearBufferfv(GL_COLOR, 1, clearRevealageColor);
```

Dabei bestimmt der erste, beziehungsweise zweite, Funktionsparameter, für welchen Color-Buffer die jeweilige Blending-Funktion oder Clear-Color gesetzt werden soll.

Im Fragment-Shader werden die zwei Color-Texturen befüllt durch:

```
1 layout(location = 0) out vec4 accumTex;
2 layout(location = 1) out vec4 transtex;
3
4 void main(){
5   float z = texture(depthMap, projCoords.xy).x;
6   vec4 objColor = texture(colorMap, projCoords.xy);
7   accumTex = objColor * weight(z,alpha);
8   transTex = vec4(alpha);
9 }
```

Listing 5: Fragment Shader zur Transparenzberechnung

Da dieser Render-Pass in zwei Color-Buffer rendert, muss durch die `location` der Index der Color-Buffer gesetzt werden, vergleiche Zeile 1 und 2 in Listing 5. Nach McGuire [14] wird in `accumTex` die Farbe durch eine Funktion gewichtet. Gängige Gewichtungsfunktionen werden in [14] aufgelistet.

In `transTex` wird die Transparenz gespeichert. Beispiele dieser Texturen finden sich im Anhang in Abbildung 28.

Um die Transparenz auf das finale Bild anzuwenden werden die Texturen des vorangegangenen Passes dem letzten Render-Pass übergeben. Die finale Transparenz erhält man durch den korrekten Texturzugriff auf die Transparenz-Textur, siehe Listing 6 Zeile 3. Die finale Farbe wird aus der akkumulierten Textur gelesen und durch die homogene Koordinate geteilt, vergleiche Zeile 4.

```
1 void main(){
2   vec4 ac = texture(accum, projCoords.xy);
3   float r = texture(trans, projCoords.xy);
4   fragmentColor = vec4(ac.xyz/ clamp(ac.w, 1e-4, 5e4), r);
5 }
```

Listing 6: Fragment Shader zur Transparenzanwendung

5 Evaluation

Um das Programm zu evaluieren wurde in verschiedenen Testszenarien die Anzahl an Frames gemessen, welche das Programm innerhalb einer Sekunde erzeugt. Außerdem soll das Ergebnis mit echten Haaren verglichen werden. Dazu wurde ein Tower-Rechner mit folgenden Spezifikationen verwendet:

- Betriebssystem: Windows 10 (64 bit)
- Prozessor: Intel i7-6700 (3.4 GHZ)
- Grafikkarte: NVIDIA GTX 980 (Maxwell-Architektur), 4GB VRAM
- Arbeitsspeicher: 16GB

Das Programm wurde mit dem 64-bit Compiler von Visual Studio 2015 kompiliert. Des Weiteren wurde mit einer Auflösung von 800x800 Pixeln gerendert.

5.1 Aufbau der Testszenarien

Die Testszenarien lassen sich unterteilen in *Haare* und *Fell*, welche sich in die Schritte der Render-Pipeline aufteilen:

- Beleuchtung (B)
- Beleuchtung und Schatten (B + S)
- Beleuchtung, Schatten und Transparenzen (B + S + T)

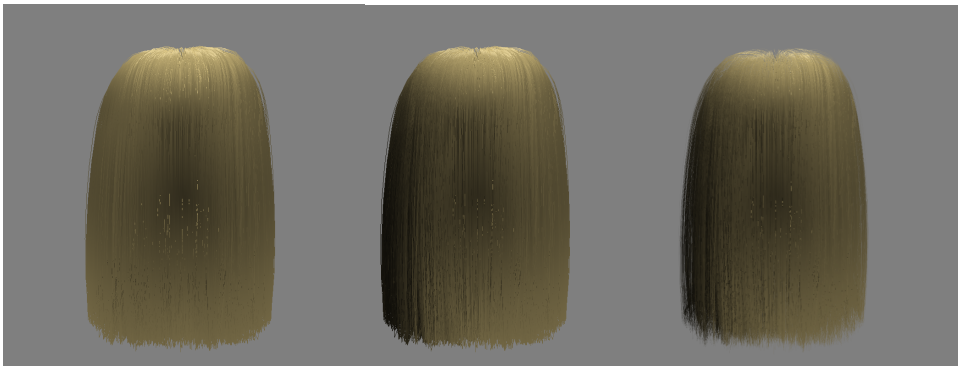
Des Weiteren wurden für Haare, sowie für Fell, verschiedene Haargeometrien verwendet, welche sich in der Anzahl an Vertices unterscheiden.

5.2 Auswertung der Testszenarien

Szenario	160.000 Vertices	687.532 Vertices	1.031.268 Vertices
B	1220.98 fps	1141.42 fps	830.58 fps
B + S	295.32 fps	261.91 fps	182.06 fps
B + S + T	146.24 fps	96.07 fps	66.08 fps

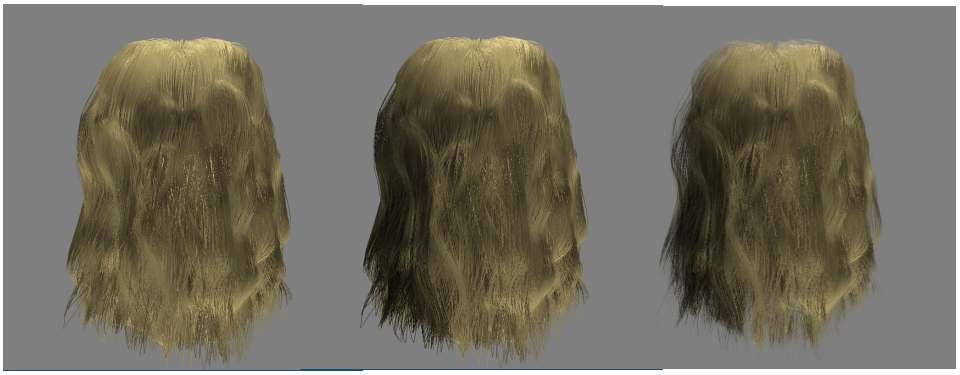
Tabelle 1: Ergebnisse des Haar-Renderings

Aus B im Vergleich zu B + S oder B + S + T lässt sich folgern, dass die Beleuchtung alleine nur einem geringen Anteil der Berechnungen entspricht. Dies liegt daran, dass der Tiefentest aktiviert ist und Blending deaktiviert. Somit werden nur die Vertices betrachtet, die von der Kamera gesehen werden. Außerdem wird für die Beleuchtung nur ein Render-Pass benötigt. Ergebnisse für die Beleuchtung der Modelle werden in den Abbildungen 18a, 19a und 20a dargestellt.



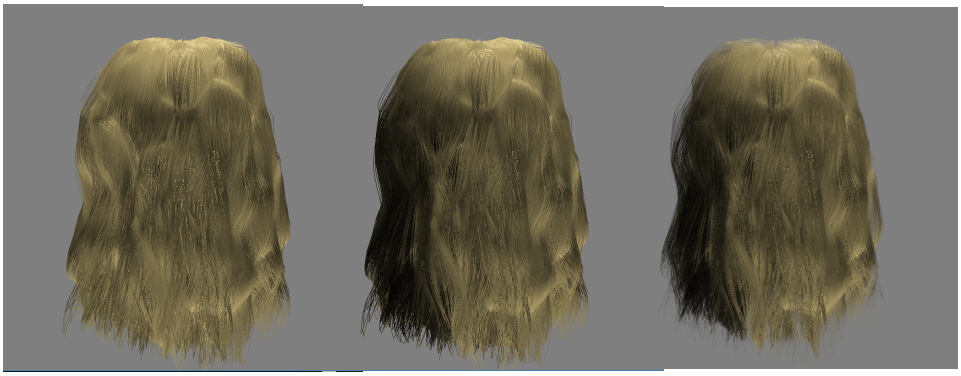
(a) Beleuchtung (Lichtquelle rechts) **(b)** Beleuchtung und Schatten **(c)** Beleuchtung, Schatten und Transparenz

Abbildung 18: 10.000 Haare dargestellt durch 160.000 Vertices



(a) Beleuchtung (Lichtquelle rechts) **(b)** Beleuchtung und Schatten **(c)** Beleuchtung, Schatten und Transparenz

Abbildung 19: 10.000 Haare dargestellt durch 687.532 Vertices



(a) Beleuchtung (Lichtquelle rechts) **(b)** Beleuchtung und Schatten **(c)** Beleuchtung, Schatten und Transparenz

Abbildung 20: 15.000 Haare dargestellt durch 1.031.268 Vertices

Da für die Beleuchtung und Schattierung drei Render-Passes benötigt werden, sinkt die Frame-Rate stark, ist aber noch im für Menschen als in Echtzeit wahrnehmbaren Bereich von mehr als 60 fps. Nicht nur die zwei zusätzlichen Render-Passes sind für die niedrigere Rate verantwortlich, sondern auch die Erstellung der Deep Opacity Map. Für diese wird das Blending aktiviert und der Tiefentest ausgestellt, sodass jeder Vertex des Modells betrachtet wird. Die Ergebnisse für Beleuchtung und Schattierung werden in Abbildung 18b, 19b und 20b illustriert.

Im letzten Szenario wurde die gesamte Render-Pipeline getestet: Beleuchtung, Schattierung und Transparenz. Dies benötigt, wie in Abbildung 17 beschrieben, insgesamt fünf Render-Passes. Für die Transparenzberechnung werden zwei zusätzliche Render-Passes benötigt. Im ersten Pass zur Vorbereitung der Transparenzen wird der Tiefentest erneut deaktiviert und das Blending aktiviert. Außerdem wird in einen Framebuffer mit zwei Color-Attachments gerendert. Im zweiten Pass ist der Tiefentest deaktiviert und Blending aktiviert. Trotz fünf Render-Passes, in welchen aufwändige Berechnungen durchgeführt werden, bleiben alle Modelle echtzeitfähig. Die final gerenderten Bilder mit allen fünf Render-Passes sind in Abbildung 18c, 19c und 20 dargestellt.

Durch Tesselation können einfach viele neue Vertices hinzugefügt werden. Durch die `gl_LevelInner` und `gl_LevelOuter` Variablen im Tesselation Control Shader lassen sich einstellen, wie hochauflösend tesseliert werden soll. Es wurde als Grundobjekt ein Stanford-Bunny gewählt, welcher mit Tesselation-Level zwei beziehungsweise vier gerendert wurde.

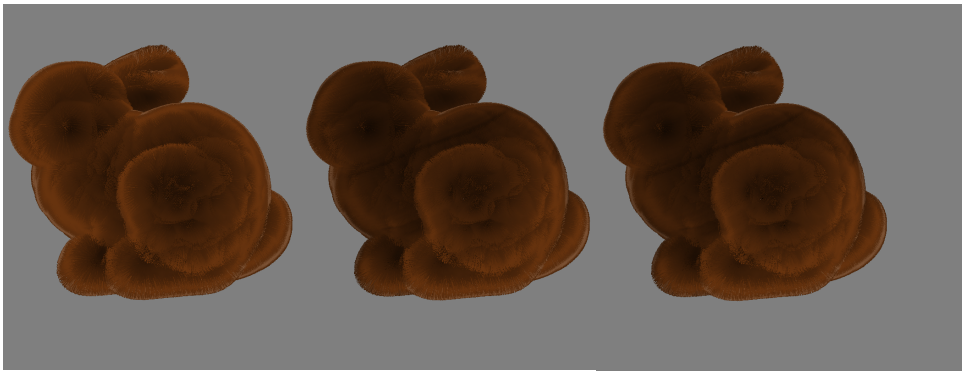
Szenario	2.088.900 Vertices	22.977.900 Vertices
B	86.63 fps	80.24 fps
B + S	85.04 fps	39.03 fps
B + S + T	80.12 fps	22.47 fps

Tabelle 2: Ergebnisse des Fell-Renderings

Da die Beleuchtung nicht sehr aufwändig ist, werden selbst mit 22.977.900 Vertices echtzeitfähige Resultate erzielt. Die Ergebnisse des Fell-Renderings werden in Abbildungen 21a und 22a gezeigt.

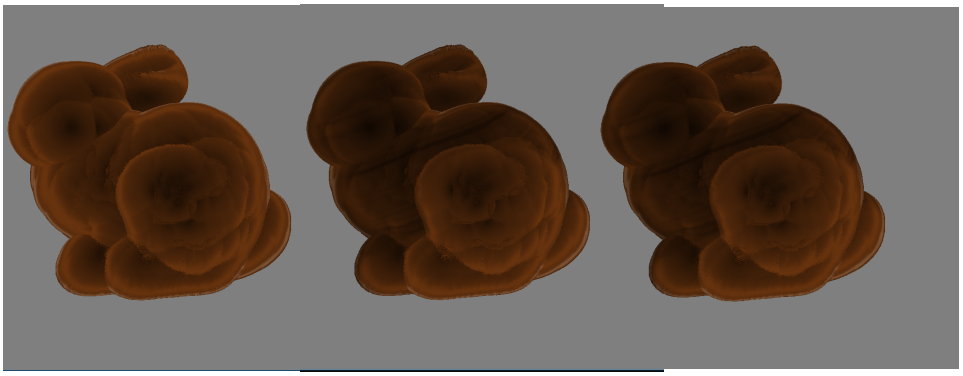
Die Schattenberechnung trägt bei einem Tesselation-Level von zwei, also bei 2.088.900 Vertices, nicht signifikant zur Verschlechterung der Frame-Rate bei. Allerdings wirken sich die zwei zusätzlichen Render-Passes auf eine Geometrien mit vielen Vertices aus, da jeder Vertex zur Berechnung der Deep Opacity Map evaluiert wird. Daher ist ein Abfall der Frame-Rate um knapp 50% zu erkennen. Die Ergebnisbilder werden in Abbildung 21b und 22b dargestellt.

Im letzten Szenario wird ein erneuter Abfall der Frame-Rate um ca. 44% bei komplexer Geometrie erkennbar. Wie auch im vorangegangenen Schat-



(a) Beleuchtung (Lichtquelle rechts) **(b)** Beleuchtung und Schatten **(c)** Beleuchtung, Schatten und Transparenz

Abbildung 21: Tesselation-Level 2, 2.088.900 Vertices



(a) Beleuchtung (Lichtquelle rechts) **(b)** Beleuchtung und Schatten **(c)** Beleuchtung, Schatten und Transparenz

Abbildung 22: Tesselation-Level 4, 22.977.900 Vertices

tenverfahren, wird auch bei der Transparenzberechnung der Tiefentest ausgestellt und somit erneut jeder Vertex getestet. Bei einem Tesselation-Level von zwei sinkt die Frame-Rate nur leicht ab. Die Ergebnisse des Renderns werden in Abbildung 21c und 22c illustriert.

5.3 Qualitäts-Evaluation

Um die Qualität des Ergebnisses einzuordnen wurden mit einer Kamera Fotos aufgenommen. Die fotografierte Szene glich dabei der Testszene des Programms: Lichtquelle rechts oben von den Haaren. Außerdem wurden die Haarfarben aufeinander abgestimmt. Ein direkten Vergleich der Bilder findet sich in Abbildung 23.

Da die Haargeometrie in Abbildung 23a mehr Locken aufweist als Abbildung 23b, treten unterschiedliche Highlights nach Marschner auf. Außerdem

(a) Echte Haare

(b) Gerenderte Haare

Abbildung 23: Haar Vergleich

wurde ein Blitz verwendet, weswegen die echten Haare etwas heller aussehen als die gerenderten Haare.

Der Effekt der Selbstverschattung lässt sich in beiden Bildern gut vergleichen: Es ist klar erkennbar, dass die Haare, welche auf der rechten Seite sind, deutlich heller sind, als auf der linken Seite des Kopfes.

Die Transparenzen lassen sich in Abbildungen 23a und 23b gut vergleichen. Da Transparenzen vor allem dann auftreten, wenn Haare verein-zelt vorkommen. Dies lässt sich in beiden Bildern gut erkennen. In Abbildung 23a lässt sich der Effekt der Transparenzen auf Höhe der linken Schulter erkennen. In der nebenstehenden Abbildung ist dies an den Haarspitzen, sowie an der äußeren Grenze der Geometrie erkennbar.

Die in Abbildung 21 und 22 dargestellten Stanford-Bunnies unterscheiden sich durch ihr Tesselation-Level voneinander. Der höhere Tesselierungsgrad erzeugt jedoch so viele Haare, sodass kaum erkennbar ist, dass keine Oberfläche, sondern einzelne Haare gerendert werden.

Die Beleuchtung des gerenderten Hasen, vergleiche dazu Abbildung 24b, unterscheidet sich von der Beleuchtung des echten Hasen (Abbildung 24a). Dies könnte daran liegen, dass nur eine Farbe für das gesamte Fell gesetzt wird, obwohl Felle meistens gemustert sind. Außerdem wurden alle Haare mit der gleichen Länge gerendert, obwohl das Fell eines Tieres unterschiedlich lang ist. Ein weiterer Grund könnte auch die verwendete Fellgeometrie sein, da Fell meistens in eine gegebene Richtung wächst (auch Strich genannt). Da die einzelnen Haare der Geometrie nach unten abgelenkt werden (vergleiche Kapitel 4.1.1), sieht das Fell nicht realistisch aus und wird anders beleuchtet.

[7] https://www.dkfindout.com/uk/animals-and-nature/rabbits-and-hares/, Stand 12.02.18

(a) Echter Hase, entnommen aus [7]

(b) Gerendertes Fell, Ergebnisbild aus Abbildung 21c

Abbildung 24: Fell Vergleich

Die Schattierung des Hasen sieht beinahe realistisch aus. Durch das Schattierungsverfahren treten horizontale Streifen auf, siehe Abbildung 24b. Dies könnte auf die Generierung der Layer zurückgehen, da sich die Geometrie eines Tieres nicht trivial verallgemeinern lässt.

Die Transparenzen lassen sich nicht intuitiv miteinander vergleichen, da die Haare eines Tieres unterschiedliche Längen und Richtungen haben. Da die Länge bei jedem Haar gleich ist und die Richtung von den Normalen des Modells abhängig sind, treten keine vereinzelt vorkommenden Haare auf, welche für den Effekt der Transparenz wichtig sind.

6 Fazit und Ausblick

Um Haare und Fell zu rendern müssen mehrere Effekte beachtet werden. So zum Beispiel die Scattering-Effekte, welche bei der Beleuchtung auftreten. Außerdem tragen Selbstverschattung und Transparenzen zur realistischen Wahrnehmung bei. Durch die Menge an Haaren, beziehungsweise Vertices, die gerendert werden, ergeben sich viele Berechnungen, die für jedes Fragment durchgeführt werden müssen. Die implementierten Verfahren erreichen dabei Ergebnisse in Echtzeit.

Das Beleuchtungsmodell nach Marschner et al. verwendet den *acos* und den *asin*, welche nicht sehr performant sind. Daher stellen Pharr und Randima [15] einen Verbesserungsvorschlag vor: das Abspeichern der Winkel in Texturen. Dies würde die Performanz der Beleuchtung verbessern. Das Schattierungsverfahren nach Yuksel und Keyser [8] benötigt viele Berechnungen zur Erstellung der Deep Opacity Map, jedoch ist dieses Verfahren noch echtzeitfähig. Da die Berechnung auf einem weighted-OIT-Verfahren basiert, ist die Transparenzberechnung nicht so aufwändig wie andere gängige OIT-Verfahren.

Um realistischere Ergebnisse für das Rendering von Fellen zu erreichen, muss ein anderer Algorithmus zur Erstellung der Geometrie verwendet werden. Dieser sollte die unterschiedliche Länge, Richtung und Farbe der einzelnen Haare beachten. Des Weiteren bietet das Verfahren nach Yuksel et al. [16] eine Methode zur Generierung von Haaren. Diese sieht vor, dass ein *Hair Mesh* erzeugt wird, in welchem Haare durch Tesselation generiert werden. Die so erzeugten Haare lassen sich in ihrer Form und Farbe individuell anpassen. Dieses Verfahren lässt sich auch auf Tierfell übertragen. Des Weiteren könnte ein anderes Schattierungsverfahren für Fell verwendet werden, da bei Deep Opacity Maps Artefakte auftreten. In *TressFX* von AMD wird beispielsweise eine angenäherte Visibility-Funktion zur Berechnung von Schatten verwendet, welches in „GPU Pro 5“ von Wolfgang Engel beschrieben wird [17]. Außerdem könnte ein Verfahren zur Berechnung der *Global Illumination* nach Yuksel et al. [18] implementiert werden.

Zudem ist eine Simulation der Haare sinnvoll, um eine realitätsnahe Darstellung von Haaren und Fell zu erzielen. Dazu sollten die Berechnungen der Simulation auf der Grafikkarte stattfinden. Falls Hair Meshes [16] verwendet werden, bietet sich zum Beispiel eine volumetrische Stoffsimulation nach Wu und Yuksel [19] an.

Zu den heute bestehenden Verfahren werden in Zukunft einige hinzukommen, da durch die stetige Weiterentwicklung der Hardware immer mehr möglich wird. Es werden sich mehr Möglichkeiten bieten, um Haare in der virtuellen Welt realistisch darzustellen.

A Anhang

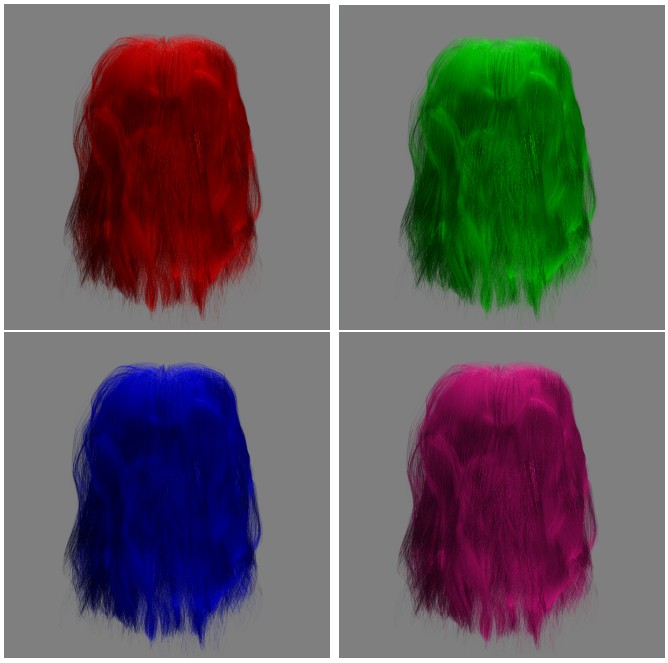
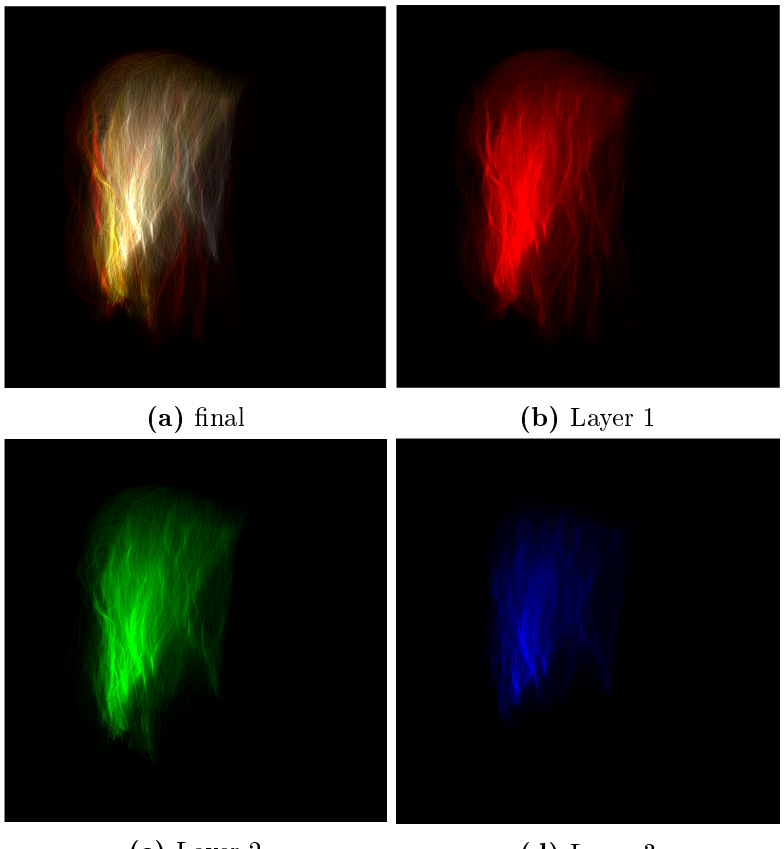


Abbildung 25: Ergebnisbilder mit unterschiedlichen Haarfarben

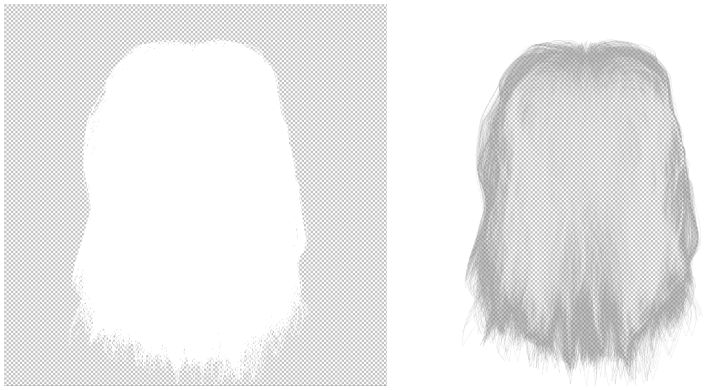
Abbildung 26: Echte Haare von nahem betrachtet: Beleuchtung, Schattierung und Transparenzen lassen sich auf dem Bild gut erkennen



(a) final **(b)** Layer 1

(c) Layer 2 **(d)** Layer 3

Abbildung 27: Deep Opacity Map nach Yuksel und Keyser [8]



(a) Akkumulations-Textur **(b)** Transparenz-Textur

Abbildung 28: Texturen der Transparenzberechnung nach McGuire und Bavoil [14]

Literatur

[1] Graham Sellers, Richard S Wright Jr, Nicholas Haemel, et al. Opengl superbible. *Addison Wisley*, 7, 2016.

[2] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[3] James T Kajiya and Timothy L Kay. Rendering fur with three dimensional textures. In *ACM Siggraph Computer Graphics*, volume 23, pages 271–280. ACM, 1989.

[4] Stephen R Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan. Light scattering from human hair fibers. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 780–791. ACM, 2003.

[5] Lance Williams. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM, 1978.

[6] Tom Lokovic and Eric Veach. Deep shadow maps. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 385–392. ACM Press/Addison-Wesley Publishing Co., 2000.

[7] Tae-Yong Kim and Ulrich Neumann. Opacity shadow maps. In *Rendering Techniques 2001*, pages 177–182. Springer, 2001.

[8] Cem Yuksel and John Keyser. Deep opacity maps. In *Computer Graphics Forum*, volume 27, pages 675–680. Wiley Online Library, 2008.

[9] Thomas Porter and Tom Duff. Compositing digital images. In *ACM Siggraph Computer Graphics*, volume 18, pages 253–259. ACM, 1984.

[10] Cass Everitt. Interactive order-independent transparency. *White paper, nVIDIA*, 2(6):7, 2001.

[11] Pál Barta, Balázs Kovács, Supervised László Szecsi, and László Szirmay-kalos. Order independent transparency with per-pixel linked lists. *Budapest University of Technology and Economics*, 2011.

[12] Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.

[13] Iman Sadeghi, Heather Pritchett, Henrik Wann Jensen, and Rasmus Tamstorf. An artist friendly hair shading system. In *ACM Transactions on Graphics (TOG)*, volume 29, page 56. ACM, 2010.

[14] Morgan McGuire and Louis Bavoil. Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):122–141, December 2013.

[15] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation.* Addison-Wesley Professional, 2005.

[16] Cem Yuksel, Scott Schaefer, and John Keyser. Hair meshes. In *ACM Transactions on Graphics (TOG)*, volume 28, page 166. ACM, 2009.

[17] Wolfgang Engel. *GPU Pro 5: advanced rendering techniques.* CRC Press, 2014.

[18] Cem Yuksel, Ergun Akleman, and John Keyser. Practical global illumination for hair rendering. In *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*, pages 415–418. IEEE, 2007.

[19] Kui Wu and Cem Yuksel. Real-time hair mesh simulation. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2016)*, New York, NY, USA, 2016. ACM.