UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Point Rendering

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von
Pascal Dietz

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
Universität Koblenz, Institut für Computervisualistik

Zweitgutachter: Dipl.-Inform. Niklas Henrich
Universität Koblenz, Institut für Computervisualistik

Koblenz, im August 2009

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.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung

Ein Dreieck ist die einfachste geometrische Figur die eine Fläche bildet (es besitzt genau die mindestens dafür nötige Anzahl an Punkten). Mit Dreiecken können alle anderen geometrischen Figuren nachgebildet werden. Deshalb dienen Dreiecke in der Computergrafik als grundlegende Bausteine für das Modellieren und Rendern von dreidimensionalen Szenen. Dementsprechend ist auch die heutige Grafik-Hardware für das Rendern von Dreiecken optimiert, was es aufkommenden Alternativen erschwert mitzuhalten. Entweder muss eine alternative Methode so schnell sein, dass sie allein mit der Rechenleistung der CPU die Geschwindigkeit von hardware-beschleunigten Dreiecken erreicht, oder es muss möglich sein, sie mit den mittlerweile programmierbaren Recheneinheiten(*Shadern*) der GPU effizient umzusetzen.
Eine solche Alternative ist das sogenannte Point Rendering. Levoy und Whitted [1] schlugen ’85 vor, Punkte als grundlegendes Element beim Rendern zu benutzen. Die Idee beim Point Rendering ist es, die zu rendernden Objekte bzw. deren Oberflächen mit einem dichten Netz aus Punkten nachzubilden. Frühere Ansätze von punktbasiertem Rendering richteten ihr Augenmerk dagegen eher auf die Darstellung von z.B. Partikeleffekten wie Rauch oder Feuer.
Das Hauptaugenmerk von Point Rendering Verfahren lag dabei nicht nur darauf, schneller zu sein als Dreiecke, sondern auch den steigenden Speicheraufwand bei großen bzw. hochaufgelösten Modellen zu reduzieren ([6], [10]). Da Point Rendering in der Regel keine umfassenden Nachbarschaftsinformationen zwischen den einzelnen Primitiven (den Punkten) benötigt, sind hier entsprechende Einsparungen möglich.
Jedoch können beim Point Rendering unschöne Bildartefakte auftreten, die vor allem daher rühren, dass Punkte eigentlich keine Dimension haben und deswegen keine geschlossenen Flächen bilden können. Es wurden jedoch Methoden wie z.B. Splatting entwickelt, um diese und andere Probleme zu lösen.

1.1 Motivation und Problemstellung

Die Aussicht auf ein Verfahren, welches das übliche Rendern von Dreiecken beschleunigen oder gar ersetzen könnte, ist reizvoll. Dass Point Rendering in der Tat mit traditionellen Verfahren mithalten kann, zeigen die verschiedenen Arbeiten, die in den letzten Jahren zu diesem Thema veröffentlicht wurden. Beispiele hierfür sind u.a. *QSplat* [6], *Perspective Accurate Splatting* [16] und *Deferred Blending* [18]. Allerdings wurden diese Verfahren speziell für hochaufgelöste Modelle (Modelle mit einer sehr großen Anzahl an Punkten) entwickelt. Daher stellt sich die Frage, ob Point Rendering auch für weniger komplexe Modelle effizient einsetzbar ist und welche Probleme dabei auftreten können, denn um Flächen bzw. ganze Objekte ausschließlich mit Punkten lückenlos darzustellen, sind eigentlich sehr viele Punkte nötig. Können Point Renderer auch bei solchen Modellen qualitativ hochwertige Bilder in angemessener Zeit erzeugen?

1.2 Zielsetzung

Das Ziel dieser Arbeit war es, bestehende Point Rendering Verfahren zu untersuchen und darauf aufbauend einen eigenen Point Renderer zu entwickeln. Mit diesem sollte dann die Anwendbarkeit auf weniger komplexe Modelle geprüft werden. Dabei galt es auftretende Probleme zu analysieren und gegebenenfalls Lösungsansätze zu finden.

2 Grundlagen

Im Folgenden werden bestehende Verfahren bzw. Techniken für die wichtigsten Bausteine eines Point Renderers vorgestellt. Dies soll einen Überblick über die Besonderheiten und Probleme beim Point Rendering und den derzeitigen Stand der Technik liefern.

2.1 Point Rendering Verfahren

Bisherige Point Rendering Verfahren lassen sich im Prinzip in zwei Kategorien einteilen. Auf der einen Seite die Point Renderer, die ausschließlich Punkte benutzen, und auf der anderen Seite die Hybridverfahren, die sowohl Dreiecke als auch Punkte einsetzen um die Vorteile von beiden auszunutzen.

2.1.1 Point Rendering

Point Renderer benutzen ausschließlich Punkte zum Rendern von Modellen. Beispiele hierfür sind u.a. die im Abschnitt 1.1 genannten Point Rendering Verfahren. Dabei kann durch den Verzicht auf die in Dreiecksstrukturen enthaltenen Nachbarschaftsinformationen der benötigte Speicheraufwand erheblich reduziert werden, da pro Punkt weniger Daten anfallen als pro Dreieck. Außerdem ist die Anzahl der Punkte bei regelmäßiger Verteilung geringer als die der Dreiecke. Als kleines Rechenbeispiel hierfür sei der Fall eines Rasters aus $n \cdot n$ Punkten betrachtet, siehe Abbildung 1. Die Anzahl der Dreiecke für solch ein Raster beträgt

$$2 \cdot (n-1) \cdot (n-1). \tag{1}$$

Aus der Formel wird ersichtlich, dass das Verhältnis von Punkten zu Dreiecken bei steigendem n gegen $1:2$ konvergiert. Dies stellt keine allgemein geltende Formel für das Verhältnis zwischen Punkten und Dreiecken dar, da Dreiecksnetze nicht unbedingt genau so aufgebaut sein müssen, sondern soll lediglich die möglichen Einsparungen an zu rendernden Primitiven verdeutlichen. Diese Speichereinsparungen sind insofern wichtig, als dass bei sehr komplexen Modellen mit hoher Samplingdichte der Speicherverbrauch enorm sein kann. Bei zu hohem Speicherverbrauch müssen z.B. ständig Daten aus dem Arbeitsspeicher gelöscht und die neuen Daten eingetragen werden. Dies kostet Zeit und ist deswegen deutlich langsamer als würde der benötigte Datensatz komplett in den Speicher passen, wodurch

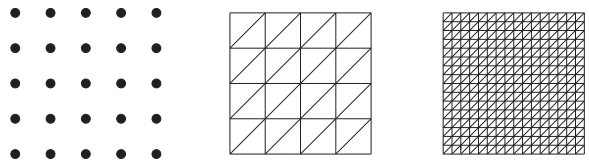

Abbildung 1: Links: Ein Raster aus 25(n=5) Punkten. Mitte: Die daraus resultierenden 32 Dreiecke. Rechts: Ein Dreiecksraster aus 289(n=17) Punkten resultiert in 512 Dreiecken

ein dauerhaft schnellerer Zugriff auf die Daten gewährleistet wäre. Andere Möglichkeiten zur Reduzierung des Speicheraufwandes sowie möglichen Nachteilen die dabei entstehen können, gibt es im Abschnitt 2.3.2.
Point Rendering bringt jedoch auch Probleme mit sich. Es können Bildartefakte auftreten, die einer besonderen Behandlung bedürfen. Z.B. können beim Rendern Lücken entstehen oder an scharfen Kanten Überlappungen. Mehr zu den möglichen Bildartefakten in den Abschnitten 2.4 und 3.5.

2.1.2 Hybridverfahren

Im Gegensatz zu Point Renderern, die ausschließlich mit Punkten arbeiten, versuchen Hybridverfahren sowohl die Punkte als auch die Dreiecke der Modelle zu benutzen, um die Vorteile von beiden Techniken zu vereinen. Beispiele hierfür sind *POP* [8] und *Hardware-Accelerated Point-Based Rendering of Complex Scenes* [11]. Diese Verfahren versuchen Dreiecke weiterhin dort zu benutzen, wo das alleinige Rendern von Punkten zu Bildartefakten führen würde und Punkte dort, wo ohne große Kompromisse in der Bildqualität Einsparungen oder Beschleunigungen erreicht werden können. Damit sind die Einsparungen nicht so groß wie beim Point Rendering, aber dafür gibt es auch weniger Bildartefakte.
In der Regel setzen Hybridverfahren Dreiecke ein, um nahe Objekte detailliert und fehlerfrei darzustellen und greifen bei steigendem Betrachtungsabstand (bzw. sehr kleinen Dreiecken, wo der Qualitätsverlust dem Betrachter kaum oder gar nicht auffällt) auf Punkte zurück.

2.2 Eingangsdaten

Die verschiedenen Point Rendering Verfahren benutzen als Eingangsdaten Punktwolken oder Dreiecksmeshes. Beide Ansätze haben gewisse Vor- und Nachteile, die im Folgenden erläutert werden.

2.2.1 Punktwolken

Punktwolken sind eine Menge von 3D-Koordinaten zu denen keinerlei Nachbarschaftsbeziehungen gegeben sind. Erstellt werden solche Punktwolken z.B. mit 3D-Laserscannern, mit denen reale Objekt von allen Seiten abgetastet werden. Die

Koordinaten der dabei erzeugten sogenannten Samples werden aus der Entfernung und dem Winkel zum Scanner berechnet und digital gespeichert.
Zur weiteren Darstellung mit gängigen Renderverfahren müssen dann die fehlenden Nachbarschaftsbeziehungen berechnet werden, um aus der Punktwolke ein Netz aus Dreiecken(Mesh) zu erzeugen, wie z.B. in *Surface Reconstruction from Unorganized Points* [2]. Aufgrund der Tatsache, dass die von 3D-Scannern erzeugte Menge von Punkten schnell in die Millionen geht (aktuelle Scanner erzeugen über 900.000 Punkte pro Sekunde), ist die Mesh-Erzeugung rechenintensiv und der für die entstehende Struktur benötigte Speicherbedarf enorm. Deswegen werden die erzeugten Meshes oftmals noch reduziert, was wiederum aufwendige Berechnungen erfordert, wobei die Form des ursprünglichen Objekts gewahrt bleiben soll. Als extremes Beispiel für den Speicherbedarf sei hier das *Digital Michelangelo Project* [5] genannt, im Rahmen dessen unter anderem für die Statue des David zwei Milliarden Samples erzeugt wurden und ein Speicherbedarf von 32 Gigabyte (inklusive der Bilder für die Textur) anfiel. Point Renderer die lediglich Punktwolken als Grundlage für ihre Berechnungen brauchen, wie z.B. *Surface Splatting* [9], sparen sich folglich sowohl die Berechnungen des Meshes als auch den erhöhten Speicherbedarf für die Nachbarschaftsinformationen.
Eine gewisse Vorverarbeitung muss jedoch dennoch durchgeführt werden. Für die einzelnen Punkte müssen die für die Beleuchtung benötigten Normalen berechnet werden. Da jedoch keine Nachbarschaftsinformation und somit keine Informationen über die Ausrichtung der Fläche, auf denen die Punkte liegen, vorhanden sind, ist diese Berechnung aufwendiger als bei Dreiecksnetzen. Aufwendiger deshalb, weil zur Berechnung der Normalen eines Punktes Daten über die Struktur der Oberfläche um einen Punkt und damit über seine Nachbarschaft nötig sind. Da bei Punktewolken solche Nachbarschaftsinformationen nicht gegeben sind, müssen diese zumindest lokal berechnet werden. Eine Möglichkeit dazu ist das Anlegen einer Tangentenebene an einen Punkt und die Punkte in seiner Nähe, um somit die lokale Ausrichtung der Fläche zu berechnen und damit die Normale.

2.2.2 Dreiecksnetz

Ein Dreiecksnetz besteht aus Dreiecken, deren strukturelle Zusammengehörigkeit durch Nachbarschaftsinformationen geben sind. Beispielsweise sind einem Dreieck die Punkte bekannt aus denen es besteht und über diese Punkte auch die benachbarten Dreiecke, mit denen es sich diese Punkte teilt. Die für die Beleuchtung benötigten Normalen lassen sich aufgrund der vorhandenen Nachbarschaftsinformationen relativ einfach berechnen. Hierfür werden die Flächennormalen der Dreiecke mit dem Kreuzprodukt aus den Dreieckskanten berechnet und für die Punktnormalen an den Eckpunkten die Flächennormalen der zugehörigen Flächen gemittelt. Als Beispiel siehe Abbildung 2. Die Flächennormalen $n1$, $n2$ und $n3$ berechnen sich aus dem Kreuzprodukt der entsprechenden Kantenvektoren a, b, c und d. Die Punktnormale n berechnet sich dann durch Aufsummierung und Nor-

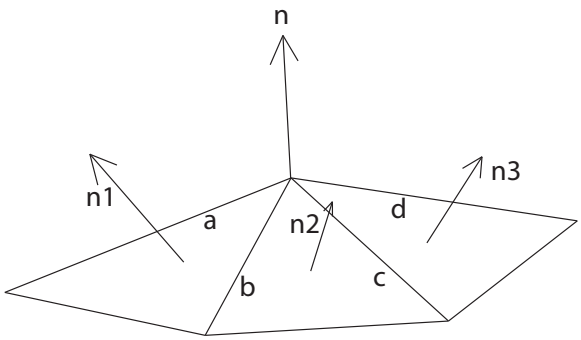

Abbildung 2: Berechnung von Normalen auf Dreiecksnetzen.

malisierung der Flächennormalen:

$$
\begin{aligned}
n1 &= a \times b \\
n2 &= b \times c \\
n3 &= c \times d \\
n &= \frac{n1 + n2 + n3}{\|n1 + n2 + n3\|}
\end{aligned}
$$

Desweiteren haben Dreiecksmeshes den Vorteil, dass sie weitverbreitet und somit leicht zugänglich oder mit entsprechender Software selbst erzeugbar sind. Modelle aus Punktwolken sind nicht so verbreitet und die Erzeugung eigener Scandaten setzt den Zugang zu entsprechenden 3D-Scannern vorraus.
Der Nachteil bei Dreiecksmeshes ist, dass diese für die Darstellung mit Point Rendering manchmal eine zu geringe Auflösung besitzen. Aus diesem Grund erzeugen Point Rendering Verfahren welche Dreiecksnetze als Eingabedaten nutzen üblicherweise ihre eigenen Punktesamples, anstatt direkt auf die Punkte des Netzes zurückzugreifen.

2.3 Datenstruktur

Abgesehen von der Art der Eingabedaten ist es auch relevant, wie man diese speichert. Dies bezieht sich nicht nur auf Point Renderer. Jedem Renderer liegen spezielle Datenstrukturen zugrunde, die den Rendervorgang beschleunigen und/oder den Speicheraufwand reduzieren sollen. Im Folgenden werden zwei in Point Renderern oft verwendete Techniken erläutert. Zum einen Baumstrukturen, die dazu dienen, die eingelesenen Punkte hierarchisch zu ordnen, zum anderen Quantisierung, was im Prinzip einer Art Kompression oder Codierung der Daten gleichkommt.

2.3.1 Baumstruktur

Um nicht alle anfallenden Berechnungen für jeden einzelnen Punkt durchführen zu müssen, setzen Point Render meistens Baumstukturen ein, um räumlich nahe beieinander liegende Punkte zu gruppieren und gemeinsam zu behandeln. Um beispielsweise frühzeitig eine ganze Gruppe von Punkten, deren Normalen vom Betrachter wegzeigen, nicht zeichnen zu müssen (*Backface Culling*). Ohne eine Baumstruktur müsste dieser Normalentest für jeden Punkt einzeln ausgeführt werden, was deutlich langsamer wäre. Dies beschränkt sich nicht nur auf den Test von Normalen, sondern z.B. auch auf Tests der Koordinaten von Punkten um festzustellen, ob einige Punkte vielleicht gar nicht im Sichtfeld liegen und somit auch nicht gezeichnet werden müssen (*View Frustum Culling*). Um *Backface Culling* durchzuführen, müssen die Normalen der Punkte in einem Baumknoten zunächst zusammengefasst werden. In Point Renderern wird dafür üblicherweise auf *The Cone of Normals Technique for Fast Processing of Curved Patches* [3] zurückgegriffen. Dabei wird aus den Normalen der Punkte eine gemittelte Normale, sowie der Winkel zur maximal davon abweichenden Normale berechnet. Desweiteren werden Regionen vor und hinter dem daraus entstehenden Normalenkegel erstellt, für die besondere Eigenschaften gelten. Sollte sich die Position des Betrachters z.B. in einer bestimmten Region vor dem Kegel befinden, so gelten alle Punkte des Kegels auf jeden Fall als dem Betrachter zugewandt und für alle Kindknoten des getesteten Knotens kann dieser Test dann gespart werden (eine genauere Erklärung zu den Normalenkegeln folgt im Abschnitt 3.7).
Es gibt verschiedene Arten von Baumstrukturen: Octrees (wie z.B. in *Surfels* [4]) unterteilen den Datensatz so, dass von der Wurzel an jeder Knoten in acht Untergruppen aufgeteilt wird, *QSplat* [6] benutzt Quadtrees um die Daten in jeweils vier Untergruppen aufzusplitten und *Deferred Blending* [18] benutzt einen BSP-Tree (binary space partitioning) um die Daten in jeweils zwei Hälften zu splitten (siehe Abbildung 3). Egal welche Baumstruktur realisiert wird, letztlich geht es darum eine Hierarchie aufzubauen, die die Traversierung des Datensatzes beschleunigt.
Es lässt sich variieren, wie man die Daten in einem Baum speichert bzw. traversiert. Üblicherweise entweder durch Tiefensuche oder Breitensuche, vergleiche Abbildung 4. Tiefensuche bedeutet, dass bei der Traversierung ein Pfad von der Wurzel an so tief wie möglich verfolgt wird, bis entweder eine Abbruchbedingung erfüllt (z.B. dass der Normalentest für einen Knoten ergeben hat, dass der Sub-Baum unter diesem Knoten nicht gezeichnet werden muss) oder ein Blatt-Knoten erreicht wurde. Dies lässt sich durch Rekursion sehr einfach realisieren. Breitensuche dagegen bedeutet, dass zuerst alle Knoten einer Baumtiefe traversiert werden, bevor die nächst tiefere Stufe traversiert wird. Diese Art der Traversierung ist nicht so einfach durch eine Rekursion durchführbar, jedoch ist es damit möglich, von Beginn des Rendervorgangs an eine komplette, zunehmend höher aufgelöste Version der Szene zu zeichnen, selbst wenn der Baum noch nicht vollständig traversiert wurde. Damit ließen sich z.B. auch erwünschte Frameraten erzwingen ohne zu riskieren, dass das verfrühte Abbrechen des Rendervorgangs zu fehlenden Bildteilen

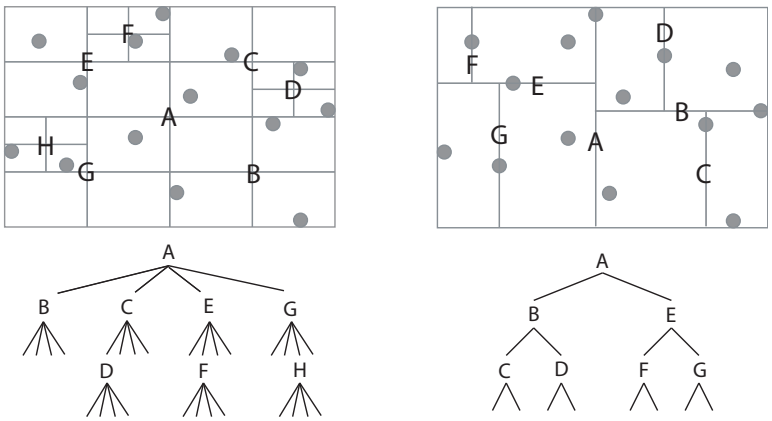

Abbildung 3: Zwei Beispiele zu Baumstrukturen. Links: Ein Quadtree unterteilt die Datenmenge anhand der räumlichen Mittelpunkte jeweils in vier Partitionen. Rechts: Ein BSP-Tree unterteilt die Datenmenge durch ein Element jeweils in zwei Partitionen. Durch welches Element die Unterteilung vollzogen wird lässt sich je genauso variieren wie die Orientierung der Trennlinien (in diesem Beispiel stets durch das mittlerste Punkt auf der Achse der größten räumlichen Ausdehnung).

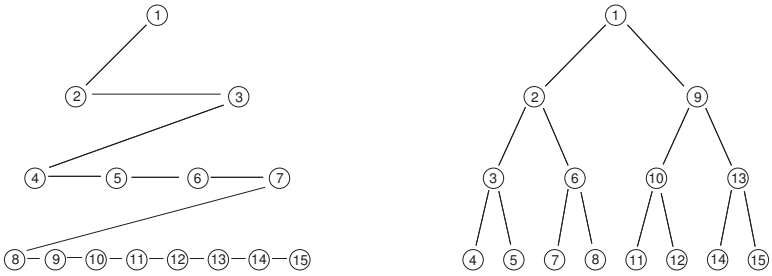

Abbildung 4: Zwei Arten der Baum-Traversierung. Die Kreise stellen Baumknoten dar, die Zahlen geben die Reihenfolge an, in der die Knoten traversiert werden. Links: Traversierungsreihenfolge bei Breitensuche. Rechts: Traversierungsreihenfolge bei Tiefensuche.

führt.
Eine andere Frage ist, welche Positionsdaten in den Knoten gespeichert werden. Die meisten Point Renderer speichern in den Knoten sogenannte *Bounding Spheres*. Dies sind Kugeln, deren Zentren und Radien so gewählt werden, dass sie alle Punkte (bzw. *Bounding Spheres*) ihrer Kinderknoten umschließen. Die eigentlichen Punkte stehen dann erst in den Blättern des Baumes. Eine andere Variante wird z.B. beim Hybridverfahren *POP* [8] angewandt. Hier sind in den Knoten zwar ebenfalls Bounding Spheres gespeichert, aber in den Blättern stehen dann Dreiecke statt der Punkte.

2.3.2 Quantisierung

Unter Quantisierung versteht man die Aufteilung eines stetigen Wertebereichs in Intervalle mit diskreten Werten. Genutzt wird dieses Verfahren unter anderem auch beim Umwandeln von analogen in digitale Signale (z.B. bei Audiodaten wie Musik) oder gängigen Kompressionsverfahren wie der jpeg-Komprimierung von Bildern. Der Zweck dabei ist, einen Wertebereich, der theoretisch unendlich viele Werte annehmen kann, in eine endliche Anzahl von Teilbereichen einzuteilen. Dies reduziert den benötigten Speicherbedarf erheblich, da nun weniger Werte gebraucht werden, um den Wertebereich darzustellen, jedoch auf Kosten der Genauigkeit der Werte. Wie sehr der benötigte Speicherbedarf reduziert werden kann und wieviel Genauigkeit dabei tatsächlich verloren geht, hängt sehr von den Werten und der gewählten Anzahl der Quantisierungsstufen ab.
In vielen Point Rendering Verfahren wird Quantisierung eingesetzt, um den benötigten Speicherbedarf zu verringern. In *QSplat* [6] wird Quantisierung genutzt, um z.B. die Radien von Baumknoten zu vereinfachen. Anstatt jeden Radius eines Knotens durch einen expliziten Wert mit einer Genauigkeit von einigen Nachkommastellen zu berechnen, wird dies lediglich für den Wurzelknoten des Baumes durchgeführt und jeder Kindknoten wird dann relativ zu dessen Position berechnet. Dies geschieht bei *QSplat* in der Form, dass der Radius eines Knotens zwischen einem $1/13$ und $13/13$ des Radius seines Vaterknotens annehmen kann. Das gleiche wird auch für den (x, y, z)-Offset des Zentrums eines Kindknotens vom Zentrum des Vaterknotens gemacht. Kombiniert man beides bekommt man 28561 (13 hoch vier) mögliche Zahlenwerte. Da jedoch viele dieser Werte ungültig sind bzw. nicht auftreten können(z.B. können nicht gleichzeitig sowohl der x als auch der y Wert den maximalen Wert von $13/13$ haben, da dies außerhalb des möglichen Radius läge), verbleiben letztlich 7621 gültige Wertekombinationen (laut den Angaben in *QSplat* [6]). Für die Speicherung dieser Werte reichen somit 13 Bit. Würden wie gewöhnlich Float-Werte für die Koordinaten und den Radius benutzt um die expliziten Werte zu speichern, würde dagegen ein Speicherbedarf von 128 Bit (vier mal 32 Bit) anfallen. An diesem Beispiel sieht man deutlich, in welche Größenordnungen die Einsparungen fallen können. In diesem Fall kostet die Quantisierung jedoch auch Berechnungszeit, da aus den relativen Werten zur Laufzeit wieder explizite Werte berechnet werden müssen.

In *Efficient High Quality Rendering of Point Sampled Geometry* [10] wird Quantisierung benutzt, um auch den Rechenaufwand zu reduzieren. Anstatt die Normalen aller Punkte exakt zu speichern, werden diese in 8192 mögliche Richtungen unterteilt, wodurch nur 13 Bit zur Speicherung gebraucht werden (anstelle von 96 Bit die eine Speicherung mit Float-Werten bräuchte). Diese Werte werden dann auch für das tatsächliche Rendern benutzt, anstatt sie wie bei *QSplat* [6] zur Laufzeit zurückzurechnen. Dadurch muss die Beleuchtung nicht für jeden Punkt und dessen Normale berechnet werden, sondern lediglich für die 8192 möglichen Richtungen. Die Ergebnisse werden in einer Lookup-Tabelle abgelegt und müssen dann für jeden Punkt nur noch ausgelesen werden. Bei einem Modell mit beispielsweise einer Millionen Punkte müssen also in jedem Frame nicht eine Millionen Beleuchtungsberechnungen für die Normalen durchgeführt werden sondern nur 8192. Der optische Verlust der bei dieser Quantisierung auftritt ist für das menschliche Auge bei normaler Betrachtung normalerweise nicht sehr auffällig und daher vernachlässigbar.
In jüngeren Point Rendering Verfahren wird Quantisierung nur noch selten erwähnt. Dies könnte an der rasanten Zunahme der Speichergrößen in den letzten Jahren liegen.

2.4 Splatting

Das Problem bei Punkten als Grafikelement ist, dass ein Punkt im keine Dimension hat. Dies bedeutet er ist infinitesimal klein und es ist prinzipiell nicht möglich damit geschlossene Flächen darzustellen. Dem Punkt einen Radius zu geben und damit als Kugel zu rendern ist möglich, wäre aber relativ langsam, da eine Kugel wiederum ein geometrisches Objekt ist, welches z.B. aus vielen Dreiecken besteht, damit sie rund erscheint. Eine Lösung ist das sogenannte Splatting. Dabei wird anstatt eines Punktes oder einer Kugel eine Scheibe(im Folgenden mit Splat bezeichnet) mit einem gewissen Radius gerendert. Der Name Splatting kommt daher, dass ein Splat in diesem Sinne quasi einer plattgedrückten Kugel gleichkommt. Es gibt viele Möglichkeiten wie man diese Splats erstellen kann, die sich in Berechnungsaufwand und Darstellungsqualität unterscheiden. In den folgenden Abschnitten sollen die einzelnen Aspekte der Splat-Erstellung erläutert werden.

2.4.1 Größe der Splats

Die Größe bzw. der Radius eines Splats ist ein besonders wichtiger Faktor was die Bildqualität angeht. Wird der Radius zu klein gewählt, entstehen Lücken zwischen den Splats, wird der Radius zu groß gewählt, kommt es zu Überlappungen. Diese fallen zwar bis zu einem gewissen Maß nicht auf, können aber gerade an Kanten zu unschönen Überhängen führen. Desweiteren ist der Splatradius wichtig für einen dynamischen Detailgrad (mehr dazu im Abschnitt 2.5). Um den Splatradius zu bestimmen, werden üblicherweise die Distanzen zwischen einem Punkt und seinen Nachbarn verglichen. Der Radius sollte mindestens der Hälfte der berech-

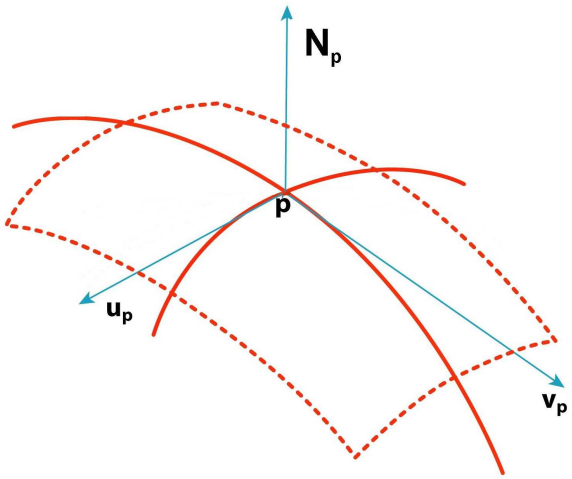

Abbildung 5: Oberflächenkrümmung um einen Punkt. P ist der Punkt und N_p die Normale des Punktes. Die beiden Richtungen u_p und v_p bezeichnen die Richtung der minimalen Oberflächenkrümmung (v_p) und der maximalen Oberflächenkrümmung (u_p). Abbildung entnommen aus [7] und leicht reduziert

neten Distanz zwischen den Punkten entsprechen. Dies kann je nach Anordnung der Punkte allerdings zu Lücken führen, weswegen in *QSplat* [6] empfohlen wird, den Radius für einen Punkt gleich der Hälfte der größten berechneten Distanz in allen Dreiecken, zu denen der Punkt gehört, zu setzen. Dies führt zwar zu relativ großen Splats, garantiert aber, dass keine Lücken auftreten, egal wie die Punkte zueinander stehen.
Ein anderer Ansatz wird in *Differential Point Rendering* [7] verfolgt. Hier wird die Krümmung der Oberfläche in einem Bereich um einen Punkte mit Hilfe der Differentialrechnung bestimmt. Die Größe der Splats wird dann entsprechend der Oberflächenkrümmung gewählt (vergleiche Abbildung 5), sodass Punkte auf stark gekrümmten Flächen einen kleineren Radius bekommen während Punkte auf nur schwach gekrümmten Flächen einen größeren Radius besitzen. Im Anschluss werden redundante Splats entfernt. Eine Redundanz besteht dann, wenn die umliegenden Splats die Oberfläche auch ohne das zu entfernende Splat genau genug reproduzieren können.

2.4.2 Form und Ausrichtung der Splats

Die Form der Splats dient vor allem dazu, entstehende Bildartefakte zu verringern. Theoretisch müssen die Splats nicht rund sein. Auch mit eckigen Splats, in Form von beispielsweise Quadraten, lassen sich bei ausreichender Punktdichte schöne und vor allem schnelle Ergebnisse erzielen. Jedoch entstehen hierbei auch weniger schöne Effekte, vor allem an den Silhouettenkanten: Kanten sind nicht gerade sondern gezackt, da die Ecken der Quadrate über die Kanten hinausragen. Um die-

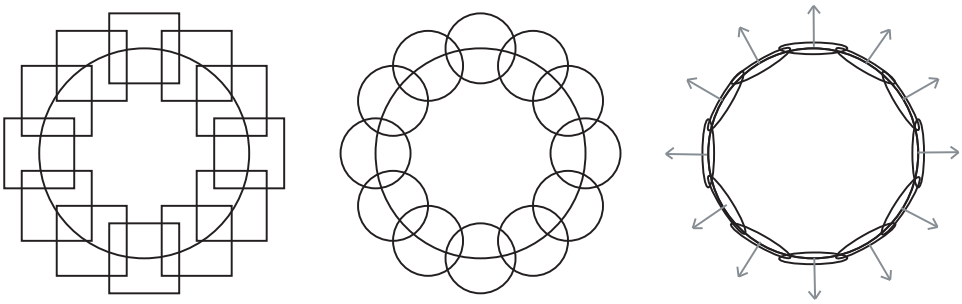

Abbildung 6: Auswirkung der Splatform und -ausrichtung am Beispiel der Silhouette eines Kreises. Links: Quadratische Splats. Mitte: Runde Splats. Rechts: Runde Splats die an den Normalen (Pfeile) ausgerichtet sind (Ellipsen).

sen Effekt zu verringern, werden Splats in der Regel als Scheiben gerendert, da die Ränder so keine Zacken mehr aufweisen. Eine einfache Möglichkeit besteht z.B. darin, die in OpenGL vorhandenen *GL_POINTS* zu verwenden, die je nach Einstellung als Quadrate oder Kreise gerendert werden, welche zum Betrachter hin ausgerichtet sind. Dies verringert zwar die Zacken, sorgt aber dafür, dass die Objekte aufgedunsen wirken, da Punkte die auf Kanten liegen stets entsprechend ihres Radius über die Kanten hinausragen. Eine andere Möglichkeit ist es, mittels einer Textur die Alphawerte eines quadratischen Splats so abzuändern, dass sich eine runde Form ergibt. In *Phong Splatting* [15] dagegen wird der zunächst rechteckige Splat auf die Pixelebene projiziert und dann für jedes Pixel der Abstand zum zentralen Pixel geprüft. Ist der Abstand größer als der gewollte Radius des Splats, wird dieses Pixel nicht gezeichnet.
Um Überlappungen an Kanten zu verringern werden Splats üblicherweise an den Normalen der Punkte ausgerichtet, wodurch die Splats die Form von Ellipsen annehmen. An den Normalen ausrichten bedeutet, dass der Splat so auf der Objektoberfläche aufliegt, dass die Normale des zugehörigen Punkte senkrecht auf ihm stet. Damit bilden die Splats die Objektoberfläche genauer nach und die Objekte wirken nicht mehr so aufgebläht. Vergleiche Abbildung 6.

2.4.3 Clipping

Eine Möglichkeit um unschöne Überlappungen an Kanten zu verringern wäre die Anzahl der erzeugten Samples an solchen Stellen deutlich zu erhöhen. Dann lägen die Punkte so dicht beieinander, dass man die Splatgrößen sehr klein wählen könnte wodurch Überlappungen kaum noch wahrnehmbar wären. Das Problem dabei ist jedoch, dass zunehmend mehr Samples gebraucht würden, je kürzer der Betrachtungsabstand zu solchen Kanten wäre. Eine andere Möglichkeit zum Beseitigen von Überlappungen ist Clipping. Dabei werden benachbarte Splats die sich überschneiden an der Schnittkante abgeschnitten. Vor allem an scharfen Kanten birgt diese eine deutliche Verschönerung des gerenderten Bildes. Allerdings ist die Be-

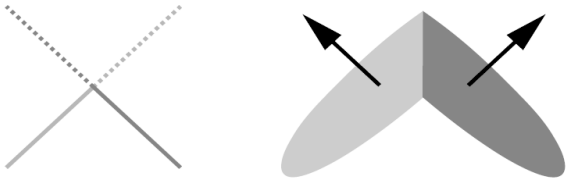

Abbildung 7: Clipping zweier Splats auf einer Kante (Abbildung entnommen aus [14]).

stimmung von Kanten nicht immer ganz einfach, da bei Point Renderern in der Regel ja kaum oder keine Nachbarschaftsbeziehungen vorhanden sind. Ein dafür entwickeltes Verfahren ist *Multi-Scale Feature Extraction on Point-Sampled Surfaces* [13], welches linienartige Merkmale (Kanten) erkennt. *Perspective Accurate Splatting* [16] benutzt damit gefundene Kanten, um die Splats die an diesen liegen miteinander zu clippen.
Eine andere Variante des Clippings bietet *Shape Modeling with Point Sampled Geometry* [14]. Dabei wird zwischen zwei Splats, die auf der jeweils anderen Seite einer Kante liegen, ein neuer Punkt auf dieser Kante berechnet. Dann wird für jede an diesen Punkt grenzende Fläche (an Kanten üblicherweise zwei) ein Splat gezeichnet, wobei diese jeweils an der Normale ihrer zugehörigen Fläche ausgerichtet werden. Die so entstandenen Splats werden dann gegeneinander geclippt. Abbildung 7 zeigt das Clipping zweier solcher Splats.

2.5 Level-of-Detail

Unter Level-of-Detail versteht man, dass Objekte bei zunehmender Entfernung zum Betrachter in geringeren Detailstufen dargestellt werden. Im Bezug auf Point Rendering bedeutet dies, entferntere Objekte mit weniger Punkten darzustellen als näherliegende Objekte. Dabei ist das Ziel die benötigte Renderzeit zu reduzieren, indem weniger Punkte gezeichnet werden müssen und die Detailreduktion so zu wählen, dass dem Betrachter die Veränderung kaum oder gar nicht auffällt. Beim Point Rendering ist die Erstellung unterschiedlicher Detailstufen sehr einfach. Um mehrere Punkte durch einen größeren Punkt darzustellen, muss dessen Zentrum einfach in die Mitte der Punkte gelegt und sein Radius so gewählt werden, dass er die betreffenden Punkte einschließt. Da die meisten Point Renderer mit einer Baumstruktur mit *Bounding Spheres* arbeiten (siehe Abschnitt 2.3.1), sind die Detailstufen damit schon gegeben. Denn die *Bounding Spheres* eines Baumknotens schließen alle *Bounding Spheres* oder Punkte seiner Kindknoten ein. Die bei jeder *Bounding Sphere* vorhandenen Angaben für Position und Radius können benutzt werden, um damit einen Splat zu rendern. Damit stellt jede Stufe des Baumes quasi eine Detailstufe dar, die nicht nur für frühzeitiges *Culling* sondern auch für Level-of-Detail benutzt werden kann.
Es bleibt die Frage, wann welche Detailstufe eingesetzt werden soll, sodass mög-

lichst kein optischer Unterschied wahrnehmbar ist. Die meisten Point Rendering Verfahren berechnen hierfür die Größe, die ein Splat letztlich bei der Darstellung auf dem Bildschirm hätte. Unterschreitet diese Größe einen bestimmten Schwellwert, kann die Detailstufe an dieser Stelle verringert werden. In der Regel wird hierfür der Wert 1 benutzt. D.h. wenn ein Splat auf dem Bildschirm kleiner als 1 Pixel wäre, würde die Detailstufe reduziert werden. Dieser Schwellwert sorgt dafür, dass die Splatdichte nicht deutlich größer als die Auflösung des gerenderten Bildes ist. Dadurch gehen kaum Details verloren (zumindest keine die einem normalen Betrachter auffallen sollten). Der Schwellwert kann auch größer gewählt werden um den Rendervorgang zu beschleunigen, da die Traversion von Teilbäumen dann schon früher abgebrochen und für einen Knoten ein Splat gezeichnet wird. Dabei kann es jedoch je nach Splatgröße zu Lücken im Bild kommen.

2.6 Auslagern von Berechnungen auf die Grafikkarte

Ein Nachteil den Point Renderer gegenüber dreiecksbasierten Renderern haben ist, dass aufgrund der langjährigen Vormachtstellung von Dreiecken die heutigen Grafikkarten für diese optimiert wurden. Point Renderer konnten daher lange Zeit nicht mit dem Grafikkartenprozessor(GPU) arbeiten, sondern mussten alleine mit der CPU auskommen. Mittlerweile lassen sich die Recheneinheiten(*Shader*) in GPUs allerdings auch programmieren. Von daher versuchen viele Point Renderer Teile ihrer Berechnungen auf die GPU auszulagern, um die CPU zu entlasten und die Renderzeit zu verkürzen. Vor allem sind Point Renderer damit nicht mehr an die vorgegebenen Basistypen wie *GL_POINTS* oder *GL_TRIANGLES* gebunden und können eigene Typen erzeugen. Der Nachteil ist, dass nicht einfach nur zusätzliche Funktionen eingefügt werden können, sondern es müssen dann alle Funktionen, die die *Shader* sonst übernehmen, selbst nachprogrammiert werden. Darunter fallen Dinge wie z.B. die Berechnung der Beleuchtung.

3 Umsetzung

In diesem Kapitel werden die für diese Arbeit umgesetzten Verfahren detaillierter erklärt. Desweiteren soll erläutert werden, warum diesen Verfahren der Vorzug gegeben wurde.

3.1 Werkzeuge

Programmiert wurde in der Sprache C++ unter *Microsoft Windows Visual C++ .NET* [20]. Für die Erstellung einer Benutzeroberfläche zum Laden von Objekten und dem Wechsel zwischen den verschiedenen Rendermodi wurde *Qt* [24] verwendet, eine Klassenbibliothek für die plattformübergreifende Programmierung grafischer Benutzeroberflächen. Für die Erzeugung der 3D-Grafik wurde OpenGL [23] verwendet, eine Programmierschnittstelle zur Erzeugung von 2D- und 3D-Computergrafik, wobei *Qt* auch hier die notwendigen Mittel dazu liefert. Mit *Maya*

[19], einer 3D-Modellierungs und Animationssoftware wurden simple Testobjekte erzeugt und in das *OBJ* Format [22] exportiert. Desweiteren wurden in *OBJ* überführte 3D-Modelle des *Stanford 3D Scanning Repository* [25] verwendet sowie Modelle aus *Nate Robins* [21] Tutorials.

3.2 Verfolgte Ansätze

Für diese Arbeit wurde sowohl der Ansatz des Point Renderings als auch der des Hybridrenderings verfolgt, wobei die beiden Verfahren eine unterschiedliche Rolle einnehmen. Mit dem Point Rendering wird getestet, inwiefern sich Punkte als Grundelemente zum Rendern auch bei weniger komplexen Modellen eignen, welche Probleme dabei entstehen und wie diese evtl. überwunden werden können. Der Hybridansatz dagegen nimmt im Prinzip die Rolle eines Beschleunigungsverfahren ein. Dabei soll getestet werden, inwiefern auf Dreiecken basierendes Rendering mittels Point Rendering ergänzt werden kann und ob ein Nutzeffekt dabei herauskommt.
Die Idee war dabei nicht das Rendern von Einzelbildern, sondern die Möglichkeit sich nach dem Vorbild von *QSplat* [6] interaktiv um ein Objekt zu bewegen bzw. dieses zu drehen, um eine Betrachtung von allen Seiten zu ermöglichen.

3.3 Eingangsdaten

Als Eingangsdaten wurden Dreiecksmeshes verwendet. Grund hierfür waren die allgemeine Verfügbarkeit, sowie die Möglichkeit, mit vorhandener Software eigene Testdaten erstellen zu können. Gerade im Hinblick auf die Untersuchung weniger komplexer Modelle wäre es sehr schwierig gewesen, aus Punktwolken bestehende Datensätze hierfür zu bekommen (wer einen 3D-Scanner benutzt ist darauf aus, sehr viele Samples zu erzeugen, damit die Objektoberfläche später so korrekt wie möglich rekonstruiert werden kann).
Um Datensätze einzulesen wurde ein *Model Loader* für das *Object File* Format (*.OBJ*) [22] geschrieben. In diesem Dateiformat werden 3D Geometrie Daten in vergleichsweise simpler Form abgelegt und können relativ einfach ausgelesen werden. Als Orientierung bei der Implementation diente der *Model Loader* von Nate Robins [21]. Für diese Arbeit wurden lediglich die Koordinaten der Punkte, die Dreiecksdaten (welche Punkte zusammen ein Dreieck bilden) und die Normalen (falls vorhanden) ausgelesen. Andere Angaben wurden ignoriert.
Über das Internet sind eine Vielzahl an Modellen im *.OBJ* Format verfügbar und bekannte 3D-Modellierungsprogramme wie *Maya* [19] bieten Exportfunktionen für dieses Format an.

3.4 Datenstruktur

Obwohl im Prinzip drei verschiedene Ansätze vorliegen (nur Dreiecke, nur Punkte und Hybrid) wurde lediglich eine Datenstruktur für alle entwickelt. Die Entwick-

lung von drei voneinander getrennten Strukturen oder gar kompletten Applikationen wäre zu zeitaufwendig gewesen.
Die mit dem *Model Loader* ausgelesenen Daten werden in drei Listen gespeichert. Eine Liste für Punkte, eine Liste für Dreiecke und eine Liste für Normalen. Die Punkte besitzen Zeiger auf die Dreiecke und Normalen zu denen sie gehören, während die Dreiecke wiederum Zeiger auf die Punkte und Normalen besitzen. Damit ist es jederzeit möglich, von einem Punkt auf dessen Dreiecke und von einem Dreieck auf dessen Punkte zuzugreifen, sollte eine Berechnung diese übergreifenden Daten erfordern. Ebenso sind die Normalen stets verfügbar, sowohl die Flächennormalen der Dreiecke, als auch die Normalen der Punkte, egal ob man von einem Punkt oder einem Dreieck ausgeht.
Derartige Nachbarschaftsinformationen zu erhalten scheint dem Grundgedanken von Point Rendering zu widersprechen, allerdings nutzen auch andere Point Renderer solche Dreiecksdaten. Da selbst Point Renderer die lediglich Punkte ohne Nachbarschaftsinformation als Eingangsdaten nutzen diese Informationen zu einem gewissen Grad brauchen (z.B. für die Berechnung der Normalen) und dann aufwendig selbst berechnen müssen, erscheint es wenig sinnvoll diese Informationen zu verwerfen. Zumal ja auch der hybride Ansatz verfolgt wird, bei dem die Dreiecksdaten ebenfalls vorliegen müssen.

3.4.1 KD-Baum

Für die Erstellung der Baumstruktur wurde ein *KD*-Baum (K-Dimensionaler Baum) gewählt. Ein *KD*-Baum ist eine Variation der im Abschnitt 2.3.1 angesprochenen *BSP*-Bäume. In jeder Baumstufe werden die vorhandenen Punkte senkrecht zu einer der K Dimensionen, welche durch die Achsen des Koordinatensystems gegeben sind, aufgesplittet (vergleiche den rechten Teil der Abbildung 3). Nach welcher Achse jeweils gesplittet wird, hängt von dem gewünschten Ergebnis ab. Eine gleichmäßige Rotation der Splitachsen ist ebenso möglich wie das Splitten der jeweils ausgedehntesten Achse. Wo dann auf einer Achse die Splitebene liegt kann ebenfalls variieren. Möglich ist beispielsweise die räumliche Aufteilung durch das Zentrum der Ausdehnung der Achse (also dem arithmetischen Mittel der zwei auf der Achse am weitesten voneinander entfernten Punkte) oder aber auch das Splitten durch den mittlersten Punkt der Punkte. Abbildung 8 zeigt diesen Unterschied. Für diese Arbeit wurde jeweils auf der Achse gesplittet, die die größte Ausdehnung aufwies. Die Splitebene wurde auf den jeweils mittleren Punkt gelegt. Damit ist garantiert, dass sich die Anzahl der Punkte pro Knoten pro Baumstufe halbiert, was unabhängig von der Lage der Punkte zu einer relativ konstanten Traversionsdauer führt (da die Baumtiefe dabei nur noch von der Anzahl der Punkte abhängt, nicht aber von deren Position, wie es bei der rein räumlichen Aufteilung der Fall wäre).
Aufgesplittet wird so lange, bis entweder nur noch ein Punkt in einem Knoten ist oder die räumliche Ausdehnung eines Knotens einen gewissen Schwellwert unterschreitet. Der Schwellwert ist dabei sehr klein gewählt und soll lediglich verhindern, dass bei zu dicht gelegenen Punkten die berechneten Distanzen für die

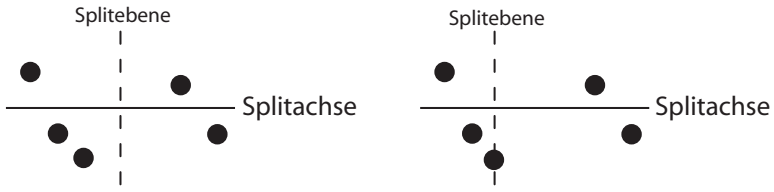

Abbildung 8: Unterschiede der Aufteilung bei der Positionierung der Splitebene. Links: Rein Räumliche Aufteilung durch Positionierung der Splitebene auf dem Zentrum der Splitachse. Rechts: Gleichmäßige Aufteilung der Punkte durch Positionierung der Splitebene auf dem mittleren Punkt.

Positionierung der Splitebene nicht die Genauigkeit von Floatdatentypen übersteigt, was zu Fehlern bei der Baumerstellung führen würde (wenn sich z.B. die x-Koordinate von Punkten die nach der x-Achse sortiert werden sollen erst in einer Nachkommastelle unterscheidet, die der Float-Datentyp nicht mehr speichert, wäre die Sortierung und die Positionierung der Splitachse fehlerhaft). Derart dicht liegende Punkte sind zwar eher selten der Fall, kamen im Verlauf der Arbeit, abhängig vom verwendeten Modell, jedoch durchaus vor.
In jedem Knoten des Baumes wird eine *Bounding Sphere* gespeichert. Deren Zentrum ist das Zentrum der *Bounding Box* um diesen Knoten. Der Radius der *Bounding Sphere* ist dann der Abstand des Zentrums zu den Ecken der *Bounding Box*. Hierin liegt dann auch der Hauptgrund für die Wahl des KD-Tree anstelle anderer Baumverfahren. Dadurch, dass stets die Dimension mit der größten Ausdehnung aufgeteilt werden kann, erreichen die *Bounding Boxen* der Baumknoten eine zunehmend quadratische Form. Dies sorgt für eine geringere Überlappung der *Bounding Spheres* (siehe Abbildung 9) und verringert somit das Auftreten von Bildartefakten (mehr zu Bildartefakten und Überlappungen folgt in späteren Abschnitten). Das zuvor angesprochene Setzen der Splitebene auf den mittlersten Punkt anstatt auf die räumliche Mitte sorgt je nach Lage der Punkte zwar für weniger quadratische *Bounding Boxen*, dennoch erschien die daraus resultierende geringere Baumtiefe und die damit verbundene kürzere Traversionsdauer sinnvoll. Die Baumstufen bilden eine Hierarchie bei der jede Hierarchiestufe die doppelte Anzahl an *Bounding Spheres* und damit quasi die doppelte Auflösung besitzt. Dies ist für die Level-of-Detail Darstellung interessant (Abschnitt 3.6.
Traversiert wird der Baum mittels Tiefensuche, da diese einen geringern Speicheraufwand benötigt und mittels Rekursion simpel umsetzbar ist. Der Vorteil der Breitensuche, dass schon ein vollständiges (wenn auch gering aufgelöstes) Bild gerendert werden kann, selbst wenn der Baum noch nicht fertig traversiert wurde, erschien für die Untersuchung von weniger komplexen Modellen nicht relevant. Dies wäre eher dann interessant gewesen, wenn das Programm über ein *Client-Server* Modell genutzt würde, bei dem ein Benutzer (der *Client*) seine zu rendernde Szene an den Server schickt auf dem das Programm liegt und dieser das Bild rendert und an den Benutzer zurückschickt. Hierbei könnte evtl. die zur Verfügung stehende

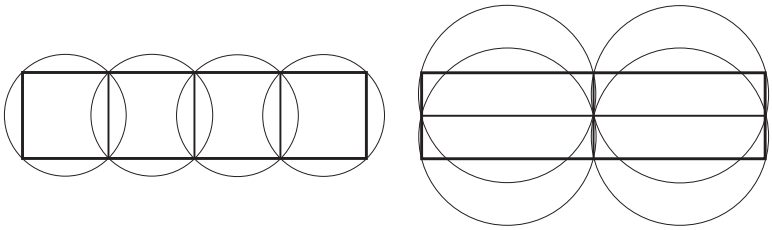

Abbildung 9: Unterschiedliche Größe und Überlappung der Bounding Spheres je nach Baumstruktur am Beispiel einer Szene mit ungleichmäßiger Ausdehnung der Dimensionen. Links: KD-Tree der stets die Dimension mit der größten Ausdehnung teilt. Rechts: Quadtree (bzw. Octree in 3D) der jeweils durch alle Dimensionen gleichzeitig teilt. Die entstehenden Bounding Spheres um die Bounding Boxen der Knoten haben einen deutlichen höheren Überhang als bei dem KD-Tree.

Bandbreite der Verbindung zwischen *Client* und *Server* ein begrenzendes Element sein, dann könnte es Sinn machen nur einen Teil der Daten schicken zu müssen um zumindest ein grobes Bild in adäquater Zeit zu erhalten. Im Rahmen dieser Arbeit ist dies aber nicht der Fall und bei weniger komplexen Modellen ist die Menge an Daten sowieso vergleichsweise gering.

3.4.2 Keine Quantisierung

Auf eine Quantisierung der Daten wurde verzichtet. Der Grund hierfür ist, dass dem Speicheraufwand aufgrund der heutigen Speichergrößen keine Priorität zugewiesen wurde, zumal der Fokus der Arbeit letztlich auch auf weniger komplexen Modellen lag, bei denen der Speicherbedarf theoretisch unbedeutend ist. Außerdem ist die Quantisierung kein Verfahren, welches nur für Point Renderer anwendbar ist. Von daher wäre der Einfluss auf den Vergleich zwischen Punkten und Dreiecken wohl eher uninteressant, da damit beide Verfahren beschleunigt werden können und dies somit eher als allgemeine Methode zur Beschleunigung von Renderern angesehen werden kann.

3.5 Splatting

Das Splatting ist wohl der wichtigste Teil eines Point Renderers. Größe, Form und Ausrichtung der Splats haben einen großen Anteil an der fehlerfreien Darstellung der Szenen. In den folgenden Abschnitten werden die verwendeten Techniken sowie die entstehenden Probleme angesprochen.

3.5.1 Größe und Ausrichtung der Splats

Wie bereits im Abschnitt 2.4.1 besprochen ist die Größe der Splats wichtig für die lückenlose Rekonstruktion der Objektoberflächen. Sind die Splats zu klein, weisen

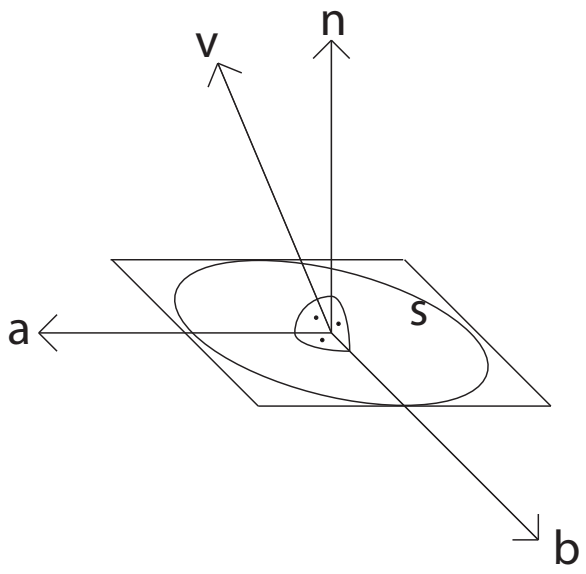

Abbildung 10: Grafik zur Ausrichtung der Splats.

die Oberflächen unschöne Löcher auf. Der auf der Differentialrechnung beruhende Ansatz aus *Differential Point Rendering* [7] erscheint bei der direkten Nutzung von Dreiecksmeshes, wie es bei dieser Arbeit der Fall ist, unnötig aufwendig, da die vorhandenen Nachbarschaftsbeziehungen direkt genutzt werden können, um die Splatgröße zu bestimmen. Wie im Abschnitt 2.4.1 bereits geschrieben, muss die Größe für den Radius eines Splats mindestens so gewählt werden, dass sich benachbarte Splats berühren. Die direkte Nachbarschaft kann aus den Dreiecksdaten ohne großen Rechenaufwand ausgelesen werden. Für die Berechnung des Radius ist dann die Distanz zum entferntesten Nachbarn zu wählen. Um zu erfüllen, dass sich die Splats der beiden Punkte berühren, muss mindestens die Hälfte dieser Distanz als Radius genommen werden. Bei zum Betrachter ausgerichteten Splats entstehen dann normalerweise keine Lücken mehr.
Problematisch sind jedoch die Kanten eines Objekts. Hier ragen die Splats von nahe an den Kanten liegenden Punkten deutlich über die eigentlichen Form des Objekts hinaus und lassen es aufgebläht wirken. Dem wirkt die Ausrichtung der Splats an den Normalen entgegen. Die Ausrichtung sorgt dafür, dass die Splats sich möglichst eng an die tatsächliche Objektoberfläche legen, ähnlich einer Tangente. Um die Ebene in der ein ausgerichteter Splat liegt zu berechnen, wurden zwei Kreuzprodukte benutzt (siehe Abbildung 10). Zuerst wurde ein Kreuzprodukt zwischen der Normalen n des Punktes P und einem beliebig zu wählendem, normalisierten Vektor v, der lediglich nicht die gleiche Richtung wie die Normale n haben darf, berechnet. Daraus ergibt sich ein Vektor a, der senkrecht auf der Normalen steht und somit auf der Ebene, auf der später der Splat S liegen soll. Nun folgt ein Kreuz-

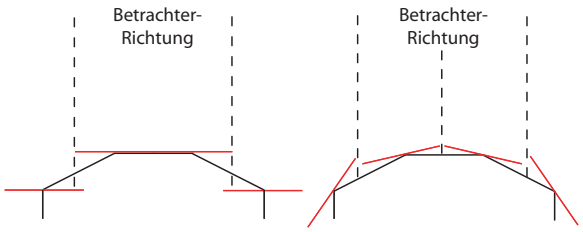

Abbildung 11: Zusammenhang von Größe und Ausrichtung der Splats. Links: Die in Betrachterrichtung ausgerichteten Splats sind zwar aus Sicht des Betrachters lückenlos, weisen aber starke Überhänge an den Silhouettenkanten auf. Rechts: Die Ausrichtung an den Normalen erfordert eine Vergrößerung der Splatradien, da sonst je nach Winkel Lücken auftreten können.

produkt zwischen der Normale n und dem eben errechneten, normalisierten Vektor a. Dies hat einen Vektor b als Ergebnis, welcher ebenfalls auf der gesuchten Ebene liegt und dabei senkrecht auf Vektor a steht. Vektor b wird ebenfalls normalisiert. Um den Splat auf der durch a und b gegebenen Ebene zu zeichnen, werden dann vier Punkte erzeugt. Mit dem Punkt P als Ausgangspunkt und den vier möglichen Kombinationen aus Addition und Subtraktion der Vektoren a und b, multipliziert mit dem vorher errechneten Radius, erhält man vier Eckpunkte eines Quadrates, welches auf der gesuchten Ebene liegt. Mittels *GL_QUADS* lassen sich die vier Punkte dann nutzen um das entsprechende Quadrat auch zu zeichnen. Der Nachteil bei dieser Berechnung ist, dass nicht ersichtlich ist, wie die Vektoren a und b auf der Ebene liegen, weshalb die entstehenden Quadrate unterschiedlich um ihre Normale rotiert sind, abhängig von deren Richtung und dem gewählten Vektor v. Bei der späteren Rundung der Splats (Abschnitt 3.5.3) entfällt dieses Problem jedoch.
Für die Größe der Splats ist dabei jedoch wichtig, dass durch die Ausrichtung an den Normalen wieder Lücken entstehen können (siehe Abbildung 11). Deswegen müssen bei diesem Verfahren die Radien der Splats vergrößert werden. Doch je größer die Splats werden desto mehr Überlappungen gibt es, was sich vor allem an Kanten bemerkbar macht. An Stellen mit vielen Punktsamples, dementsprechend hoher Splatdichte und geringer Splatgröße fällt dies kaum bis gar nicht auf. Sehr problematisch wird es jedoch bei weniger komplexen Modellen und den daraus resultierenden scharfen Kanten, sowie wenn Kantenlängen benachbarter Dreiecke unterschiedlich groß sind (siehe Abbildung 12). Tests ergaben, dass eine Vergrößerung der Splatradien auf etwa 65 Prozent der Distanz zu den entferntesten Nachbarn (anstatt vorher der minimal benötigten 50 Prozent) in den meisten Fällen ausreicht um Lücken zu verhindern. Um Überhänge an Kanten zu vermeiden, reichen eine gute Bestimmung der Splatgrößen und die Ausrichtung der Splats allerdings nicht aus.

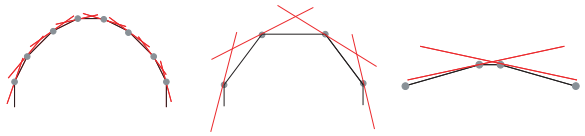

Abbildung 12: Probleme mit der Größe von Splats. Links: Bei ausreichender Samplingdichte treten je nach Betrachtungsabstand kaum sichtbare Überlappungen auf. Mitte: Bei weniger Samples und den daher größeren Splats fällt das Problem deutlicher ins Gewicht. Rechts: Bei unterschiedlich dichten Samples kommt es selbst an seichten Kanten mitunter zu extremen Überlappungen. Bei scharfen Kanten dann erst recht.

3.5.2 Clipping

Bei Objekten mit dichtem Punktenetz ist das Resultat bei den bisher eingesetzten Techniken schon sehr ordentlich. Teilweise ist kaum ein Unterschied zum Rendern mit Dreiecken wahrnehmbar, sofern der Betrachter nicht sehr nah an das Objekt herangeht. Modelle mit scharfen Kanten und geringer Punkteverteilung enttäuschen bei der Darstellung jedoch noch völlig. Die Überlappungen fallen sehr extrem aus und lassen sich weder durch eine minimale Splatgröße, noch durch die Splatausrichtung ausreichend beseitigen. Abbildung 13 zeigt den Unterschied in der optischen Qualität zwischen hochaufgelösten und weniger komplexen Modellen beim Einsatz von Point Rendering. Wie man im unteren, rechten Teil der Abbildung sehen kann, ragen viele Splats unschön über angrenzende Kanten hinaus, ganz zu schweigen von einigen enormen Splats im vorderen Teil der Grafik. Dass dies beim Stanford Bunny kaum auftritt und beim Porsche dagegen so gravierend, liegt vor allem an den unterschiedlichen Distanzen zwischen den Punkten. Beim Stanford Bunny haben fast alle Punkte in etwa den gleichen Abstand zueinander, wodurch die Splats fast gleich groß sind und somit keine Ausreißer auftreten. Beim Porsche allerdings gibt es große Flächen, die im Prinzip nur durch ein paar Punkten an ihren Ecken oder Kanten dargestellt werden. Z.B. besteht die Unterseite des Porsche aus wenigen großen Dreiecken, deren Größe sich über die gesamte Länge des Modells erstreckt. D.h. ein Punkt aus solch einem Dreieck hat auf der einen Seite Nachbarn deren Distanz sehr groß ist und auf der anderen Seite, wo sich der Übergang zwischen Boden und z.B. Front des Porsche befindet, Nachbarn deren Distanz sehr kurz ist. Da stets die Distanz zum entferntesten Nachbarn als Grundlage genommen werden muss, um eine lückenlose Darstellung zu erreichen, tritt hier das bereits im rechten Teil von Abbildung 12 gezeigte Problem in extremen Ausmaßen auf.
Obwohl derartig Überhänge an Kanten bei üblichen Point Rendering Verfahren nicht so stark auftreten, da diese entweder hochaufgelöste Modelle benutzen oder sich ihr eigenes, dichtes Punkteraster erzeugen, kommt es auch dort zu Überhängen. Denn auch bei sehr dichtgelegenen Punkten mit gleichmäßigen Abständen gibt es natürlich kleine Überlappungen an Kanten. Eine weitere Erhöhung der

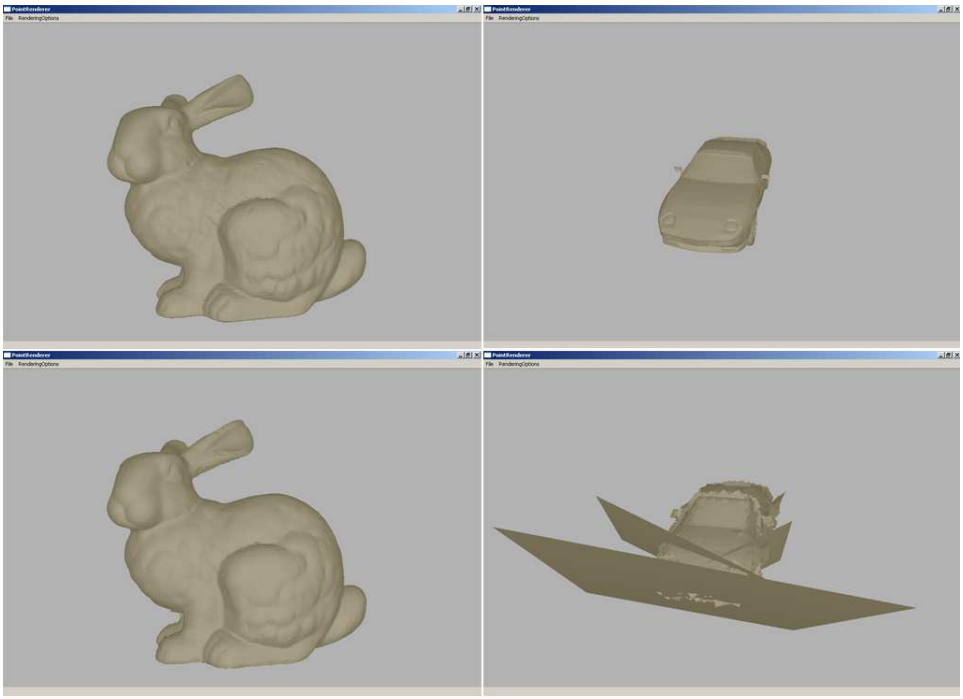

Abbildung 13: Qualitätsunterschied von Point Rendering im Bezug auf das zugrunde liegende Modell. Oben links: Stanford Bunny mit 69666 Dreiecken. Unten links: Stanford Bunny mit Splats (34835 Punkte). Oben rechts: Porsche mit 7322 Dreiecken. Unten rechts: Porsche mit Splats (4099 Punkte).

Punktdichte löst dieses Problem nicht, da ein Betrachter stets noch näher an das Objekt herangehen könnte. Abhilfe kann hier nur das Clippen der überhängenden Splats verschaffen. Da bei Dreiecksnetzen die Punkte stets auf Ecken liegen, erscheint das im Abschnitt 2.4.3 angesprochene Verfahren aus *Shape Modeling with Point Sampled Geometry* [14] am sinnvollsten. Für einen Punkt auf einer Kante wird dabei für jede angrenzende Fläche (meist nur zwei) ein Splat mit Ausrichtung an der entsprechenden Fläche erzeugt. Dann werden die so erzeugten Splats gegeneinander geclippt. Da bei Dreiecksnetzen die Punkte allerdings nicht auf oder in der Nähe von Kanten liegen, sondern auf Ecken, wäre der Aufwand hier deutlich größer, da hier nicht nur zwei, sondern mindestens drei und oft deutlich mehr Flächen an einem Punkt zusammentreffen. Wenn ein Punkt nun z.B. von sechs Dreiecken genutzt wird (was nicht unüblich ist, siehe Abbildung 1, so müssten hier sechs Splats gezeichnet und diese jeweils mit den Splats links und rechts daneben geclippt werden. Die Splats würden damit quasi fast wieder zu Dreiecken umkonstruiert werden. Durch den für solch ein umfangreiches Clipping benötigte Zeitaufwand würde das Rendern von Splats seine Konkurrenzfähigkeit gegenüber Dreiecken mit Sicherheit verlieren.
Daher wurde basierend auf diesem Ansatz eine alternative Methode eingesetzt. Da durch das Clippen an Ecken mit jeweils zwei Nachbarn aus einem Splat sowieso fast ein Dreieck wird, werden stattdessen direkt die zugrunde liegenden Dreiecke gerendert. Damit handelt es sich dann allerdings nicht mehr um einen Point Renderer, sondern um ein hybrides Verfahren. Um zu entscheiden, wann Splats und wann Dreiecke gerendert werden sollen, wird der Winkel zwischen den an einen Punkt grenzenden Flächen untersucht. Dies reduziert sich auf die Berechnung der Winkel zwischen den Normalen der Flächen und der interpolierten Normalen des Eckpunktes, was mit einem Skalarprodukt und der Berechnung dessen Arkuskosinus pro Fläche getan ist. Überschreitet der berechnete Winkel einen gewissen Schwellwert, so wird dies in einer Variable für den Punkt vermerkt. Dies lässt sich bereits im Präprozess durchführen. Zur Laufzeit muss die Variable abgefragt werden und je nach Inhalt der Variable dann in einer Schleife alle Dreiecke, die zu dem Punkt gehören, gezeichnet werden (sofern diese nicht schon vorher bei der Überprüfung eines anderen Punktes gezeichnet wurden). Dies verringert die Rendergeschwindigkeit leider immer noch deutlich, sodass diese Hybridlösung langsamer als das reine Rendern der Splats oder der Dreiecke ist.
Wie gut überhängende Splats damit beseitigt werden, hängt schließlich von dem eingestellten Schwellwert und dem zugrunde liegenden Modell ab. Wählt man den Schwellwert sehr klein gibt es zwar kaum noch Überhänge, aber es werden eben fast nur noch Dreiecke gezeichnet. Abbildung 14 zeigt die Ergebnisse bei unterschiedlich großen Schwellwerten. Dass diese Hybridlösung trotz relativ kleiner Winkel optisch nicht sehr überzeugt, liegt immer noch an dem Problem mit den unterschiedlichen Abständen zwischen den Punkten. Zur Erläuterung soll Abbildung 15 dienen. In der linken Grafik sind die Distanzen zwischen den Punkten in etwa gleich. Der Winkel zwischen den Normalen der Punkte a und b ist kleiner als der Schwellwert, von daher wird ein Splat gezeichnet. Dieser hängt zwar über, dies

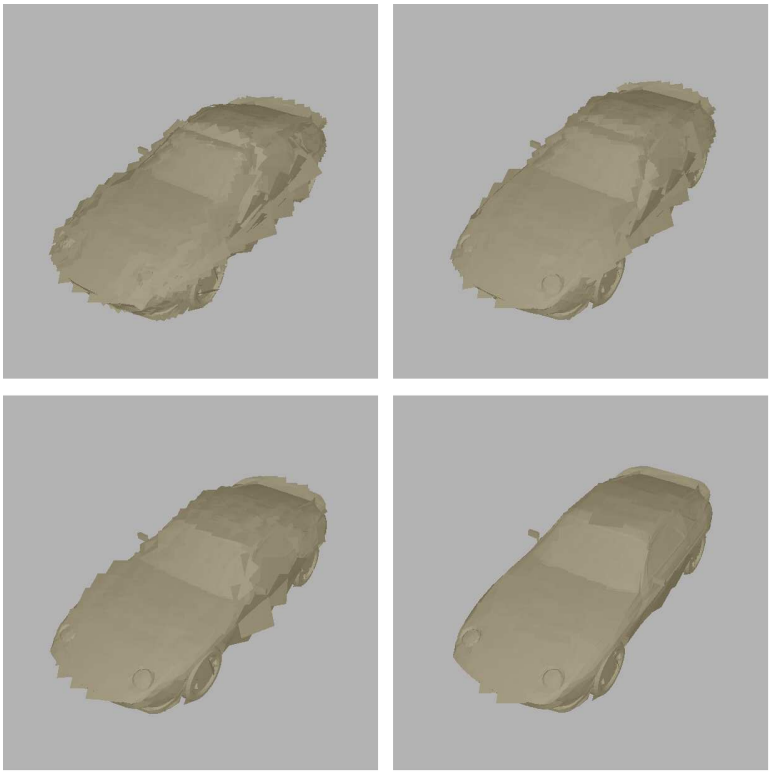

Abbildung 14: Auswirkung unterschiedlicher Schwellwerte beim Winkeltest: Ganz links: 20°, resultiert in 2269 Splats und 5229 Dreiecken. Mitte links: 10°, resultiert in 1804 Splats und 6448 Dreiecken. Mitte rechts: 5°, resultiert in 1142 Splats und 6875 Dreiecken. Ganz rechts: 2°, resultiert in 520 Splats und 7102 Dreiecken. (Das Modell an sich besitzt 4099 Punkte und 7322 Dreiecke)

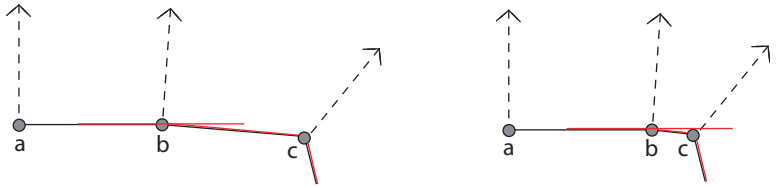

Abbildung 15: Auswirkung unterschiedlicher Distanzen zwischen benachbarten Punkten beim Winkeltest.

fällt wegen dem flachen Winkel allerdings kaum auf. Zwischen den Normalen der Punkte b und c ist der Winkel größer als der Schwellwert, weswegen zwischen den beiden ein Dreieck gezeichnet wird. In der rechten Grafik ist die Distanz zwischen den Punkten b und c deutlich kürzer, dadurch steht der Splat vom Punkt a sehr deutlich über und würde erst bei einem sehr geringen Schwellwert abgefangen.

3.5.3 Form

Die Form der Splats ist ein weiteres Mittel um unsaubere Kanten schöner darzustellen. Um Überhänge und vor allem Sägezahnkanten zu verringern, werden die Quadrate in eine runde Form gebracht. Dazu wird eine quadratische Textur erstellt die lediglich Alpha-Werte enthält, um die entsprechenden Pixel der Quadrate transparent zu machen. In einer Schleife über die 8-Bit-Werte der Textur werden diese entweder auf 0 (für transparent) oder 255 (für nicht-transparent) gesetzt. Um zu entscheiden wann eine Koordinate auf der Textur transparent sein muss, wird deren Distanz zum Zentrum der Textur bestimmt. Ist die Distanz größer als die halbe Seitenlänge der Textur (was quasi dem Radius eines Kreises entspricht, der so breit wie die Textur ist), so wird der Wert für diese Koordinate auf transparent gesetzt, sonst auf nicht-transparent. Ein Codebeispiel für die Erstellung einer Textur mit einer Seitenlänge von 256:

```
// Seitenlängen auf 256 setzen.
int width = 256;
int height = 256;

// Array mit 256*256 Einträgen für die Alphawerte.
BYTE data[65536];

// Doppelte Schleife über Höhe und Breite der Textur.
for(int i = 0; i < height; i++)
{
  for(int j = 0; j < width; j++)
  {
    /*Berechnung der Distanz der aktuellen Koordinaten zum
      Zentrum der Textur. Das Zentrum der Textur wird dabei
      als (0,0) angenommen und die Koordinaten entsprechend
      transliert.*/
    float distance = sqrt((i-height/2.0)*(i-height/2.0)
```

```
                        + (j-width/2.0)*(j-width/2.0));

    if(distance < width/2.0)
    {
      /*Ist die Distanz d kleiner als die halbe Seitenlänge
        der Textur, setze den entsprechenden Wert auf nicht-
        transparent...*/
      data[i*width + j] = 255;
    }else
    {
      //...sonst setze den Wert auf transparent.
      data[i*width + j] = 0;
    }
  }
}
```

Der so mit Werten gefüllte Array wird mit dem entsprechenden OpenGL Befehl als Grundlage für die Textur genommen. Dadurch, dass die zugrunde liegenden Quadrate an den Normalen ausgerichtet werden, erscheinen die Splats dann nicht als Kreise, sondern in Form von zunehmend schmaleren Ellipsen, je mehr deren Normale von der Blickrichtung des Betrachters abweicht.
Diese ellipsischen Splats weisen dann an den Objektkanten weniger Überhänge und Zacken auf, wobei sich diese zusätzliche Verringerung in Grenzen hält, da die Überhänge die auf eckigen Splats beruhen eher gering sind und die Ausrichtung an den Normalen schon den Hauptteil der Überhänge auffängt.

3.6 Level-of-Detail

Die Grundlagen für die Level-of-Detail Berechnungen sind dank der Datenstruktur bereits gegeben. In jedem Knoten der Baumstruktur sind Bounding Spheres durch Position und Radius gegeben, die jeweils die darunter liegenden Bounding Spheres einschließen. Um zu bestimmen ob eine bestimmte Bounding Sphere als Splat gezeichnet werden soll, wird deren Radius auf die Bildebene projiziert. Dazu wird die Berechnung aus *High-quality point-based rendering on modern GPUs* [12] verwendet:

$$size_{win} = r \cdot \frac{n}{z_{eye}} \cdot \frac{h}{t-b} \tag{2}$$

$size_{win}$ bezeichnet die auf die Bildebene prozijierte Größe, r ist der Radius der Bounding Sphere und z_{eye} der Abstand des Zentrums der Bounding Sphere zum Betrachter. n, t und b bezeichnen drei Parameter aus dem *View-Frustum* (*Near*, *Top* und *Bottom*) und h ist die Höhe des Ausgabefensters. $\frac{n}{z_{eye}}$ projiziert dabei den Radius auf die *Near-Plane* und $\frac{h}{t-b}$ skaliert die Werte dann auf Bildkoordinaten (Pixelkoordinaten).
Bei der Traversierung des Baumes wird also von der Wurzel an für jeden Knoten $size_{win}$ berechnet. Dies muss zur Laufzeit geschehen, da sich die Werte für

z_{eye} und h von Frame zu Frame ändern können. Ist $size_{win}$ größer als der eingestellte Schwellwert (üblicherweise 1 Pixel, siehe Abschnitt 2.5), so werden die Kindknoten traversiert. Ist $size_{win}$ allerdings kleiner als der Schwellwert, so wird die Bounding Sphere des aktuellen Knotens als Splat gezeichnet und eine Traversierung der Kindknoten ist nicht mehr nötig. Somit werden Gruppen von Punkten je nach Entfernung durch weniger Splats dargestellt als ihrer Anzahl entspricht, was den Rendervorgang beschleunigt.
Dieses Verfahren ist nicht nur beim Point Rendering einsetzbar, sondern wird ähnlich wie z.B. in *POP* [8] auch zur Beschleunigung beim hybriden Ansatz benutzt. Dabei wird wie oben der Baum durchlaufen und je nach projizierter Splatgröße die Traversion abgebrochen und ein Splat gezeichnet. Wenn bei der Traversierung ein Blatt erreicht wird, so werden hier allerdings keine Splats für die Punkte gezeichnet sondern die tatsächlichen Dreiecke. Damit ist dank der Dreiecke eine Darstellung der genauen Details der Modelle bei naher Betrachtung möglich und dank der Bounding Sphere Splats eine reduzierte, aber dafür beschleunigte Darstellung bei ferner Betrachtung.

3.7 Normalenkegel

Um vom Betrachter abgewandte Flächen frühzeitig aus dem Rendervorgang auszuschließen (*Backface Culling*) wurden die im Abschnitt 2.3.1 angesprochenen Normalenkegel implementiert (*The Cone of Normals Technique for Fast Processing of Curved Patches* [3]). Aber auch andere Beschleunigungsmöglichkeiten sind damit gegeben, z.B. lässt sich mit Normalenkegeln schnell testen, ob Flächen bzw. Gruppen von Flächen nicht von einer Lichtquelle beeinflusst werden und somit die Lichtquelle für diese Flächen abgeschaltet werden kann (was eine schnellere Berechnung der Beleuchtung zur Folge hätte). Die Erstellung der Normalenkegel ist in vier Schritte unterteilt:

- Konstruktion der Normalen
- Konstruktion des schwebenden Kegels
- Konstruktion des verankerten Kegelstumpfes
- Konstruktion von Regionen

Die Normalenkegel wurden ursprünglich für Bézierkurven entwickelt, wobei im ersten Schritt dann zunächst die Normalen auf den Kurven berechnet werden. Das Verfahren ist aber auch auf andere Primitive anwendbar und die Normalen können dann auf die für das jeweilige Primitiv typische Art und Weise berechnet werden. D.h. der erste Schritt zur Erstellung der Normalenkegel entfällt, da die Normalen aus den Modelldaten ausgelesen oder aus den Dreiecken bereits an früherer Stelle berechnet wurden.
Zunächst wird also der schwebende Kegel konstruiert. Schwebend bedeutet, dass der Kegel zunächst keine feste Position besitzt, sondern lediglich eine Orientierung

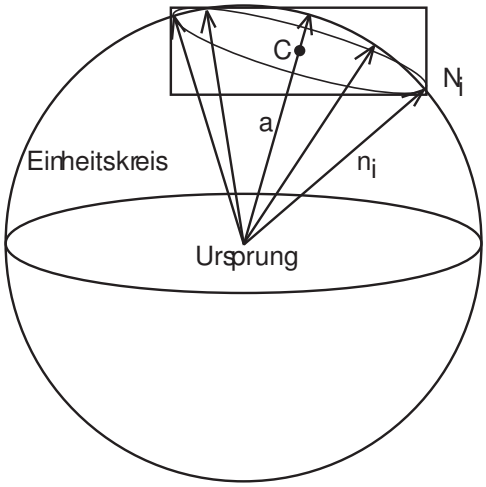

Abbildung 16: Bestimmung des schwebenden Kegels. (Abbildung angelehnt an die entsprechende Grafik in [3])

und einen Öffnungswinkel, welcher alle zum Kegel gehörenden Punkte und deren Normalen erfassen soll. Dabei wird angenommen, dass sämtliche Normalen n_i im Ursprung veranktert sind. Es wird eine Bounding Box um die Spitzen N_i der Normalen berechnet. Das Zentrum C der Bounding Box kann dann folgendermaßen berechnet werden (siehe auch Abbildung 16):

$$\begin{aligned} C_x &= \frac{(\min N_{ix} + \max N_{ix})}{2} \\ C_y &= \frac{(\min N_{iy} + \max N_{iy})}{2} \\ C_z &= \frac{(\min N_{iz} + \max N_{iz})}{2} \end{aligned} \tag{3}$$

Die normalisierte Kegelachse a wird dann durch den Ursprung und das Zentrum C der Bounding Box definiert. Der Halbwinkel des Kegels wird durch den größten Winkel zwischen den Normalen n_i und der Kegelachse a mittels dem Skalarprodukt berechnet:

$$\cos(\alpha) = \min(a \cdot n_i) \tag{4}$$

Falls α größer ist als $\frac{\pi}{2}$ wird der Normalenkegel nicht benutzt, da die Richtungen der Normalen dann mehr als einen Halbkreis abdecken, was kein richtiges *Backface Culling zulässt*.
Der verankterte Kegelstumpf erhält seine Bezeichnung, weil er im Gegensatz zum schwebenden Kegel eine feste Position besitzt und der Kegel quasi zu einem Stumpf geschnitten wird, welcher gerade alle zum Kegel gehörenden Punkte P_i einschließt. Zuerst wird der unterste Punkt B gesucht. Damit ist der Punkt gemeint, der der Kegelspitze im Bezug auf die Richtung der Kegelachse am nächsten liegt. Punkt B

wird ermittelt durch das kleinste Skalarprodukt zwischen der Kegelachse a und den Richtungen vom Ursprung zu den Punkten. Oder anders gesagt: derjenige Punkt, dessen Projektion auf die Kegelachse den kleinsten Wert aufweist. Die untere Schnittebene (quasi die Schnittebene, die die Kegelspitze abtrennt) wird definiert als senkrecht zu Kegelachse auf dem Punkt B. Die Kegelspitze wird so transliert, dass sie auf dem Punkt B liegt. Nun wird für alle Punkte P_i (welche sich über der unteren Schnittebene befinden müssen) geprüft, ob diese von dem translierten Kegel eingeschlossen werden. Dies geschieht durch die Winkelbestimmung zwischen der Kegelachse und den Richtungen der translierten Kegelspitze zu den Punkten. Ist der Winkel für einen Punkt größer als der Halbwinkel α, so wird der Punkt auf die untere Schnittebene projiziert. Berechnet wird der projizierte Punkt P' wie folgt:

$$\begin{aligned}\overline{BP_i'} &= r \cdot (\overline{BP_i} - h \cdot a) \\ h &= \overline{BP_i} \cdot a \\ r &= \frac{\sqrt{\|BP_i\|^2 - h^2} - h \cdot \tan(\alpha)}{\sqrt{\|BP_i\|^2 - h^2}}\end{aligned}$$

h bezeichnet dabei die Höhe von P_i über der unteren Schnittebene und r ist dabei das Verhältnis zwischen den Distanzen zur Kegelachse von P_i' und P_i. Nun wird wieder mittels einer Bounding Box der Mittelpunkt der projizierten Punkte und dem Punkt B auf der unteren Schnittebene berechnet sowie der Radius der Bounding Box. Dann wird der Kegel so transliert, dass die Ränder der Bounding Box auf dem Rand des Kegels liegen. Auf ähnliche Art und Weise wird die obere Schnittebene erstellt, sodass der Normalenkegel in beide Richtungen begrenzt wird (siehe Abbildung 17.

Als letzter Schritt folgt nun die Konstruktion der *frontfacing region* (vor dem Kegel), *backfacing region* (hinter dem Kegel) sowie der neutralen Region (überall sonst). Als Veranschaulichung soll die Abbildung 18 dienen. Der Kegelstumpf $ABCD$ enthält alle Punkte und die Richtungen der Normalen an diesen Punkten. Die *frontfacing region* EFG und die *backfacing region* KLM werden durch die Strecken BG und DK senkrecht auf AB konstruiert. Ebenso CE und AM senkrecht auf CD. F ist die Kegelspitze der *frontfacing region* und ergibt sich aus dem Schnittpunkt von BG und CE. L ist die Kegelspitze der *backfacing region* und ergibt sich aus dem Schnittpunkt von DK und AM. Die beiden Regionen sind also ebenfalls Kegel deren Spitzen auf der Achse des Normalenkegels liegen. Die Halbwinkel der beiden Kegel sind $\frac{\pi}{2} - \alpha$. Die Distanz der Kegelspitze L der *backfacing region* zur unteren Schnittebene AD lässt sich berechnen durch $r_b \cdot \tan(\alpha)$, wobei r_b der Radius der unteren Seite des Kegelstumpfes. Die Position der Kegelspitze der *frontfacing region* lässt sich analog durch $r_t \cdot \tan(\alpha)$ berechnen, wobei $r_t = r_b + l \cdot \tan(\alpha)$.

Mittels dieser Regionen können nun mehrere Aussagen getroffen werden, im folgenden seien allerdings nur die beiden für diese Arbeit relevanten Aussagen aufgeführt:

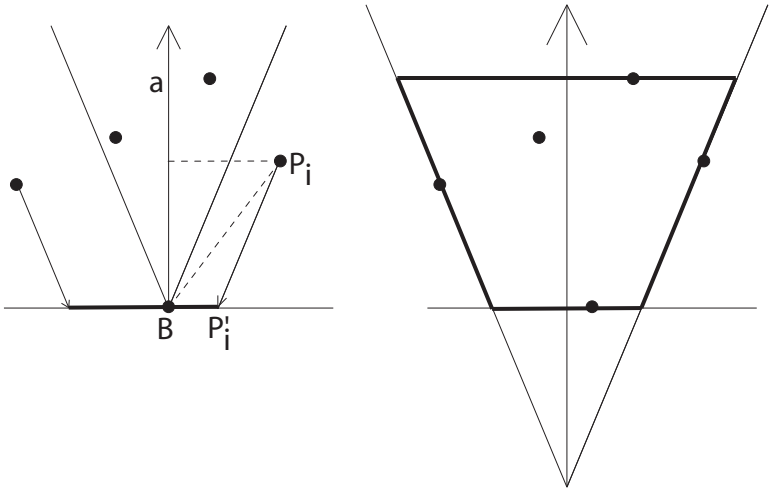

Abbildung 17: Konstruktion des verankerten Kegelstumpfes. (Abbildung angelehnt an die entsprechende Grafik in [3])

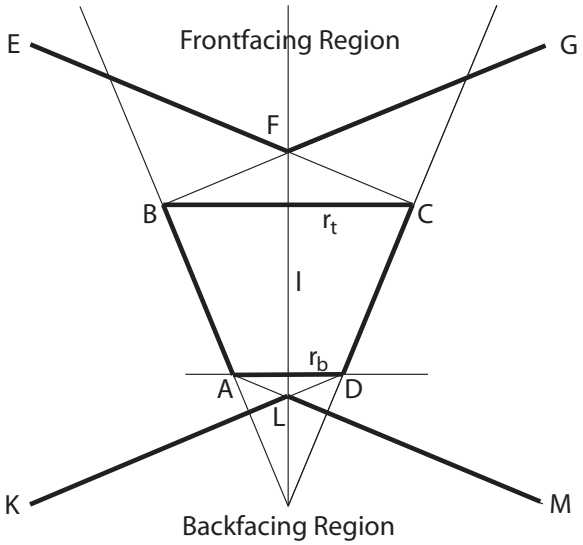

Abbildung 18: Darstellung der Regionen eines Normalenkegels als 2D Querschnitt. (Abbildung angelehnt an die entsprechende Grafik in [3])

- Befindet sich die Kameraposition in der *backfacing region*, so ist der Kegel komplett von der Kamera abgewandt und die dazugehörigen Punkte bzw. Splats müssen nicht gezeichnet werden. Damit lässt sich dieser Teilbaum von der weiteren Traversierung ausschließen.
- Befindet sich die Kameraposition in der *frontfacing region*, so ist der Kegel der Kamera komplett zugewandt und weitere Tests bezüglich der Normalenkegel und damit des *Backface Cullings* sind für diesen Teilbaum nicht mehr nötig.

Der Test ob sich die Kamera in einer der Regionen befindet besteht lediglich aus einer Winkelberechnung mittels dem Skalarprodukt: falls

$$\begin{aligned} a \cdot d_f &\geq \sin(\alpha) \qquad \text{für die } \textit{frontfacing region} \text{ oder} \\ -a \cdot d_b &\geq \sin(\alpha) \qquad \text{für die } \textit{backfacing region}, \end{aligned}$$

so befindet sich die Kameraposition außerhalb der Regionen. a ist dabei wieder die Kegelachse, α der Halbwinkel des Normalenkegels und d_f sowie d_b bezeichnen die normalisierten Richtungen von der Kegelspitze der *frontfacing* bzw. *backfacing region* zur Kameraposition.

4 Ergebnisse

Um die Ergebnisse der unterschiedlichen Verfahren direkt gegenüberstellen zu können, wurde das Programm so gestaltet, dass zwischen den Rendermodi (Point Rendering, Hybrid und zum Vergleichen das Rendern von Dreiecken) zur Laufzeit frei gewechselt werden kann. Zur Messung der Renderzeiten wurde ein Zähler integriert, der sich die Anzahl der gerenderten Frames merkt. Eine in *QT* vorhandene Funktion, *QTimer*, wird genutzt um den Zähler sekündlich auszulesen und damit die erreichte Framerate wiederzugeben. Im folgenden werden sowohl die optischen Ergebnisse als auch die unterschiedlichen Renderzeiten gezeigt. Getestet wurden verschiedene Modelle mit variierender Dichte an Dreiecken bzw. Punkten und weichen bzw. harten Kanten.
Quadratische Splats liefern bei Modellen mit ausreichender Dichte an Punkten und weichen Kanten gute Ergebnisse. In der Tat ist dann optisch teilweise kaum ein Unterschied zu Dreiecken festzustellen (Abbildung 19). Je nach Modell erreichen die Splats aber eine bis zu 50 Prozent höhere Framerate als Dreiecke.
Bei Modellen mit geringerer Punktdichte oder scharfen Kanten sind die Ergebnisse alles andere als schön. Je nach Modell können die Ergebnisse bis zur totalen Unkenntlichkeit degenerieren (Abbildung 20). Der Clipping-Ersatz in Form des eingesetzten Winkeltests kann hier leider nur teilweise Abhilfe verschaffen (Abbildung 20. Bei gering eingestellten Winkeln (in der Abbildung 5° zwischen der Flächennormale und der Normale des Eckpunktes, was einen Winkel von mehr als 10° zwischen Flächen an diesem Punkt verhindert) lassen sich die schlimmsten Überhänge verhindern, die verbliebenen Überhänge fallen dann aber umso mehr

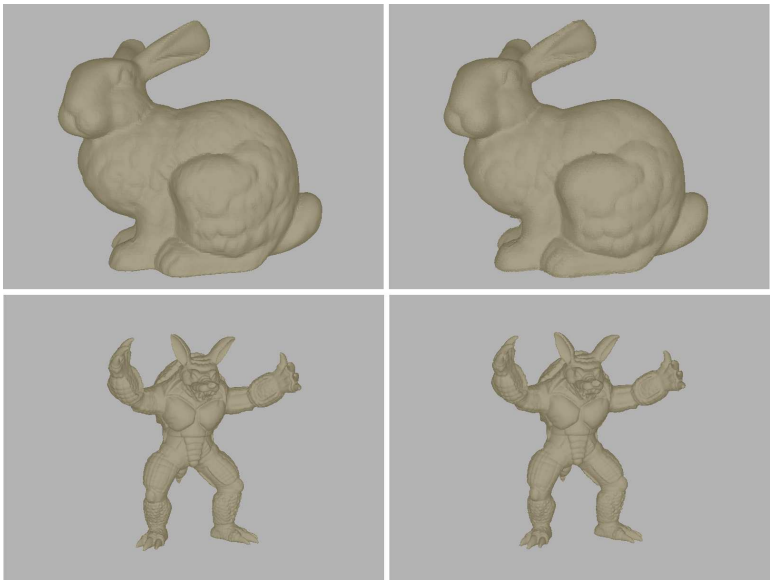

Abbildung 19: Vergleich von Ergebnissen beim Rendern mit Dreiecken und Splats bei relativ hoch aufgelösten Modellen mit weichen Kanten. Links Dreiecke und rechts Splats.

ins Auge. Desweiteren verbleiben je nach Modell nicht mehr viele Splats, was den Geschwindigkeitsvorteil nicht nur aufhebt sondern im Vergleich zu Dreiecken die Framerate sogar um bis zu 70 Prozent senkt, da teilweise für einen Punkt dessen Splat und zusätzlich dessen Dreiecke gerendert werden. Sowohl die optische Verbesserung als auch der Geschwindigkeitsverlust variieren stark mit dem eingestellten Winkel und dem zugrundeliegenden Modell.
Den Splats eine runde Form zu geben bringt keine Abhilfe in solchen Fällen. Vielmehr verhelfen runde Splats den Modellen zu weniger eckigen Kanten vor allem im Bereich der Silhouette eines Objektes. Auch hier hängt der Grad der optischen Verbesserung wieder sehr vom Modell ab (siehe Abbildung 21).
Mit dem Level-of-Detail Verfahren lassen sich die Renderzeiten bei entfernteren Objekten deutlich beschleunigen. Leider kann es hier bei genauerer Betrachtung zu einzelnen fehlerhaften Pixeln kommen (Abbildung 22). Sofern diese Fehlpixel auffallen lassen sie sich aber mit einer geringfügigen Erhöhung der Splatgröße weitgehend beseitigen.

5 Bewertung

Zum Rendern von Modellen mit hoher Punktdichte und einheitlicher Verteilung der Punkte ist Point Rendering gut geeignet. Die Beschleunigung die durch das Rendern der Punkte anstelle der Dreiecke, aufgrund der geringeren Menge an zu

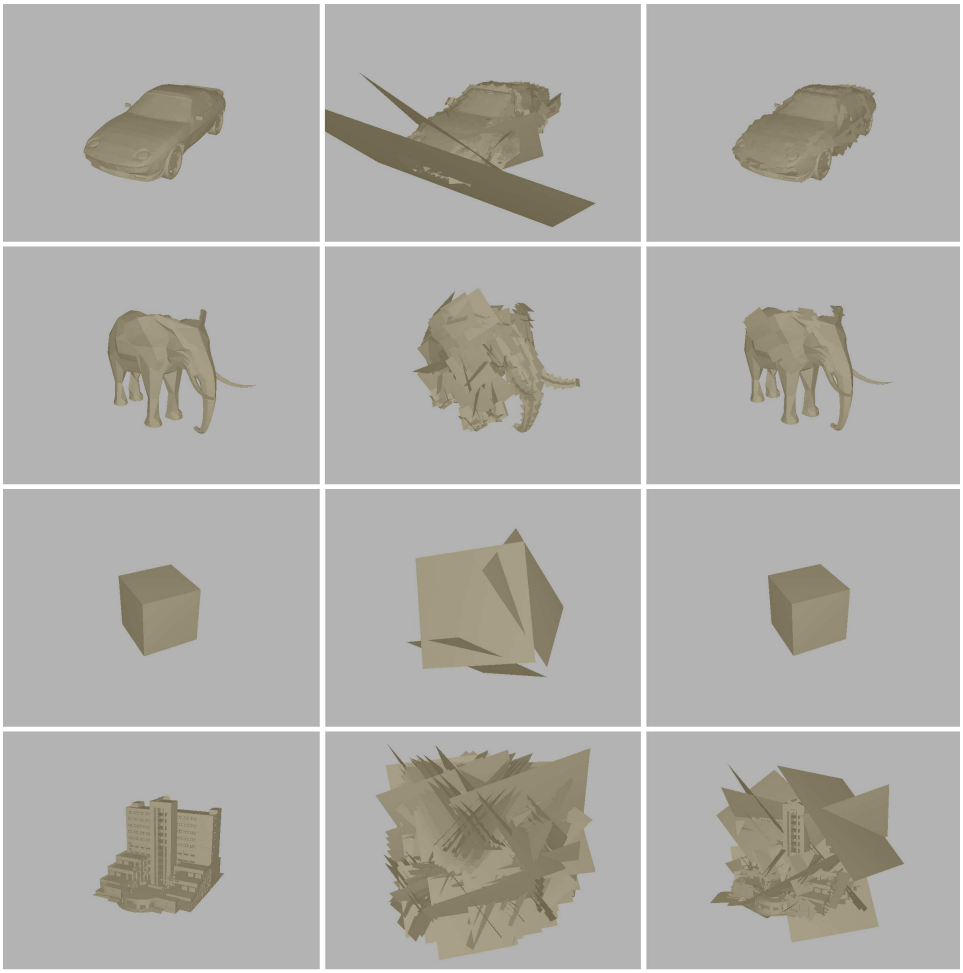

Abbildung 20: Vergleich von Ergebnissen beim Rendern mit Dreiecken und Splats bei relativ gering aufgelösten Modellen mit harten Kanten. Links Dreiecke, in der Mitte Splats und rechts Splats mit eingeschaltetem Winkeltest der Kanten mit einem Winkel von mehr als 10° mit Dreiecken anstatt Splats rendern lässt.

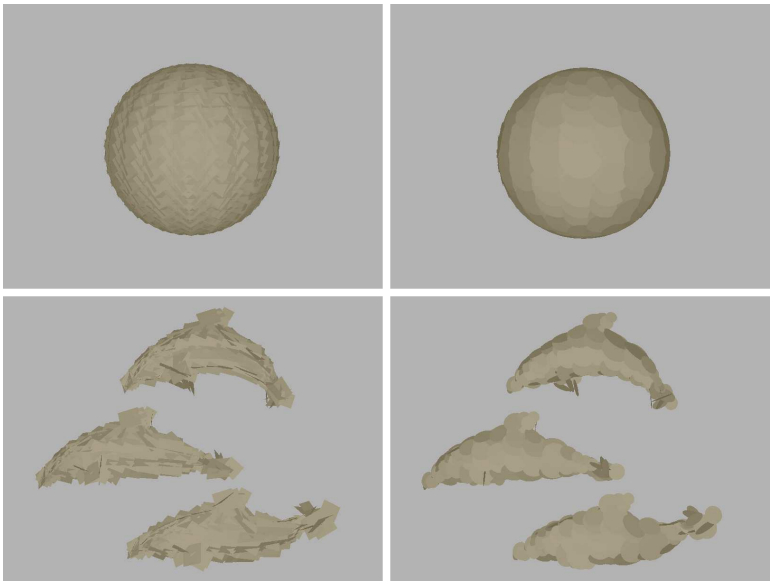

Abbildung 21: Wirkung von runden gegenüber quadratischen Splats. Links quadratische Splats, rechts runde Splats.

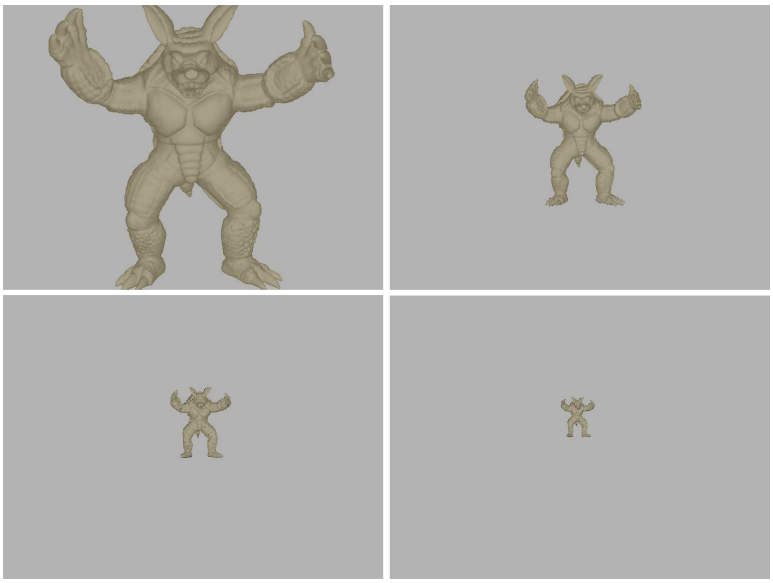

Abbildung 22: Auswirkung des Level-of-Detail Verfahrens. Oben links: Noch ohne LOD, 172.974 Splats. Oben rechts: mit LOD noch 164.131 Splats, etwa 10 Prozent schneller. Unten links: mit LOD noch 110.947 Splats, etwa 50 Prozent schneller. Unten rechts: mit LOD noch 58.052 Splats, etwa 115 Prozent schneller.

rendernden Primitiven, erreicht wird, machen das Verfahren sehr attraktiv. Auch die Optik hat dabei kaum zu leiden.
Problematisch wird es jedoch bei Modellen mit ungünstiger Punkteverteilung. Hier sind die Ergebnisse mangelhaft und die eigentliche Form der Objekte lässt sich teilweise nur noch erahnen. Die größten Probleme treten auf, wenn sich die Abstände zwischen benachbarten Punkten stark unterscheiden. Dann ist die Bestimmung einer Splatgröße die sowohl lücken- als auch überlappungsfrei ist kaum oder gar nicht möglich. Normalerweise umgehen Point Renderer das Problem, indem sie entweder nur von hochaufgelösten Scandaten ausgehen, oder sie erzeugen sich auf einem zu rendernden Modell eigene Punkte, die dann die nötige Dichte und uniforme Abstände aufweisen. Da im Rahmen dieser Arbeit aber explizit weniger komplexe Modelle untersucht werden sollten, konnte auf solch einen Ansatz nicht zurückgegriffen werden (die Erzeugung ausreichender Punkte würde diese Modelle ja wieder komplex machen).
Clipping bzw. der hier verwendete alternative hybride Ansatz mit dem Test der Winkel ist in der Lage einige Probleme zu beheben, je nachdem wie niedrig man den Schwellwert setzt. Da dies die Renderzeit aber erheblich verlangsamt, ist die Konkurrenzfähigkeit zu Dreiecken dann nicht mehr gegeben.
Eine Möglichkeit zum Einsatz von Point Rendering auch bei ungünstigen Modellen bietet das Level-of-Detail Verfahren wie es hier vorgestellt wurde. Dies führte unabhängig vom zugrunde liegenden Modell zu einer merklichen Beschleunigung bei großen Abständen zwischen Betrachter und Objekt, ohne die optische Qualität dabei stark zu beeinträchtigen. Der Aufwand für die Erstellung von Bounding Spheres erscheint deutlich geringer als alternative Verfahren zur Detail-Reduktion bei Dreiecken. Dort müssen Dreiecke zu größeren Dreiecken zusammengeschlossen werden um verschiedene Level-of-Detail Stufen zu erstellen. Das Auswählen und Zusammenführen der richtigen Dreiecke ist jedoch nicht unbedingt trivial.
Die in dieser Arbeit vorgestellten und implementierten Normalenkegel funktionierten zum Ende leider noch nicht fehlerfrei. Es wurden teilweise dem Betrachter zugewandte Flächen ausgefiltert, sodass unschöne Löcher entstanden, während manche abgewandte Flächen nicht entfernt wurden. Deswegen wird die beschleunigende Wirkung dieses Verfahrens aus der Bewertung ausgenommen, da nicht erkennbar ist, welcher Anteil der Beschleunigung aus der falschen Ausfilterung von Flächen entsteht.

6 Ausblick

Um die Effektivität des Winkeltests zu erhöhen, wäre es sinnvoll nicht nur die normalen von angrenzenden Flächen zu testen sondern evtl. auch die an diese Flächen angrenzenden Flächen. Im Prinzip müssten sämtliche Flächen einbezogen werden, die sich innerhalb des Splatradius eines Punktes befinden um den Splat dann an entsprechender Stelle zu clippen bzw. die Dreiecke darzustellen. Damit könnte es

möglich sein, dass in Abbildung 15 erläuterte Problem weitgehend zu lösen. Der hierfür anfallende, zusätzliche Berechnungsaufwand erscheint enorm, sollte sich aber zumindest auf den Präprozess beschränken.
Die im Laufe der Arbeit geplante Beschleunigung durch Nutzung der GPU wurde leider nicht durchgeführt. Die mit *GL_QUADS* erzeugten Splats erfüllten ihren Dienst und der Einsatz der GPU um die Splats auf andere Art und Weise zu erzeugen hätte an deren Aussehen nicht viel verändert. Da die Splats auch ohne die GPU bereits schneller als Dreiecke waren, wurde der weiteren Beschleunigung durch die GPU erstmal keine Priorität zugewiesen. Das Hauptaugenmerk lag eher auf der Lösung der bei Point Rendering und weniger komplexen Modellen auftretenden Probleme, hier schien die GPU nicht helfen zu können. Ein sinnvoller nächster Schritt wäre vielleicht das Nutzen der GPU um effizienteres Clipping (wie z.B. in [16]) anstelle der momentanen hybriden Lösung in Form des Winkeltests durchzuführen, da Clippen bei den weniger komplexen Modellen ein essentieller Schritt zur fehlerfreien Darstellung ist und einiges an Rechenzeit in Anspruch nimmt. Ansonsten wären beim Programmieren eigener Shader wohl eine schönere Beleuchtung in Form von z.B. Phong-Shading möglich (in dieser Arbeit wurde lediglich Gouraud-Shading eingesetzt).
Wie bereits im Abschnitt 5 angesprochen führt Point Rendering bei weniger komplexen Modellen meist zu keinen guten Ergebnissen, da entweder die optische Qualität oder die Renderzeit im Vergleich zu Dreiecken auf der Strecke bleiben. Für schönere Ergebnisse wäre die Erzeugung eigener Punkte auf solchen Modellen zielführender (zumindest was die optische Qualität angeht). Der Aufwand hierfür wäre sicherlich höher als derartige Modelle direkt mit Dreiecken zu rendern, allerdings ließen sich damit zumindest komplexe Szenen rendern die eigentlich für Point Renderer geeignet wären, in denen aber auch weniger komplexe Modelle vorkommen (z.B. eine Szene mit einer hoch aufgelösten Statue die auf einem würfelförmigen Sockel aus nur acht Punkten steht). Interessant wäre in diesem Sinne vielleicht der Einsatz eines Raytracing-Verfahrens, um unabhängig der zugrunde liegenden Modelle eine beliebige Anzahl von Samplen zu erzeugen um die Objektoberflächen ausreichend zu approximieren.

Literatur

[1] M. Levoy, T. Whitted. The Use of Poinst as Display Primitives. Technical Report 85-022. University of North Carolina at Chapel Hill, Computer Science Department, 1985.

[2] H.Hoppe, T.DeRose, T.Duchamp, J.McDonald, W.Stuetzle. Surface Reconstruction from Unorganized Points. In Computer Graphics, SIGGRAPH 92 Proceedings, pages 71-78.

[3] L.A.Shirmun, S.S.Abi-Ezzi. The Cone of Normals Technique for Fast Processing of Curved Patches. Proceeding Eurographics 1993, pages 261-272.

[4] H.Pfister, M.Zwicker, J.van Baar, M.Gross. Surfels: Surface Elements as Rendering Primitives. In Proceedings of SIGGRAPH 2000, pages 335-342.

[5] M.Levoy, K.Pulli, B.Curless, S.Rusinkiewicz, D.Koller, L.Pereira, M.Ginzton, S.Anderson, J.Davis, J.Ginsberg, J.Shade, D.Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. Proc. SIGGRAPH, 2000.

[6] M. Levoy, S. Rusinkiewicz. QSplat: A Multiresolution Point Rendering System for Large Meshes. In SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques (2000), pages 343-352.

[7] A.Kalaiah, A.Varshney. Differential Point Rendering. Proceedings of the 12th Eurographics Workshop on Rendering Techniques, 2001, pages 139-150.

[8] B.Chen, M.X. Nguyen. POP: A Hybrid Point and Polygon Rendering System for Large Data. In Proceedings of IEEE Visualization 2001, pages 45-52.

[9] M.Zwicker, H.Pfister, J.van Baar, M.Gross. Surface Splatting. In Proc. of ACM SIGGRAPH 01, pages 371-378.

[10] M.Botsch, A.Wiratanaya, L.Kobbelt. Efficient High Quality Rendering of Point Sampled Geometry. In Proceedings of the 13th Eurographics workshop on Rendering, 2002, pages 53-64.

[11] L.Coconu, H.C.Hege. Hardware-Accelerated Point-Based Rendering of Complex Scenes. In Proceedings Eurographics workshop on rendering, 2002, pages 43-52.

[12] M.Botsch, L.Kobbelt. High-quality point-based rendering on modern GPUs. In Proc. of Pacific Graphics 2003, pages 335-343.

[13] M.Pauly, R.Keiser, M.Gross. Multi-Scale Feature Extraction on Point-sampled Surfaces. In Eurographics 2003, pages 281-289.

[14] M.Pauly, R.Keiser, L.Kobbelt, M.Gross. Shape Modeling with Point Sampled Geometry. Proceedings of ACM SIGGRAPH 2003, pages 641-650.

[15] M.Botsch, M.Spernat, L.Kobbelt. Phong Splatting. Symposium on Point-Based Graphics, 2004, pages 25-32.

[16] M. Zwicker, J. Räsänen, M. Botsch, C. Dachsbacher, M. Pauly. Perspective Accurate Splatting. In Proceedings of Graphics Interface 04, 2004, pages 247-254.

[17] M.Guthe, P.Borodin, Á.Balázs, R.Klein. Real-time Appearance Preserving Out-of-Core Rendering with Shadows. Rendering Techniques 2004: 15th Eurographics Workshop on Rendering, June 2004, pages 69-80.

[18] Y. Zhang, R. Pajarola. Deferred blending: Image composition for single-pass point rendering. Computers & Graphics vol. 31(2), April 2007, pages 175-189.

[19] Maya
http://www.autodesk.com/maya

[20] Microsoft Visual C++ .NET
http://msdn.microsoft.com/de-de/vstudio/default.aspx

[21] Nate Robins OpenGL Tutorials
http://www.xmission.com/~nate/tutors.html

[22] Object Files Documentation
http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/

[23] OpenGL
http://www.opengl.org/

[24] Qt
http://www.qtsoftware.com/products/

[25] The Stanford 3D Scanning Repository
http://graphics.stanford.edu/data/3Dscanrep/