Technische Universität München
Fakultät für Informatik
Lehrstuhl für Computer Grafik und Visualisierung (I 15)
Prof. Dr. Rüdiger Westermann

Ausarbeitung

zum

Systementwicklungsprojekt im Wintersemester 2004/05

Multiresolution deformable Objects: Model generation and advanced rendering techniques

Betreuer:

Dipl.-Inf. Joachim Georgii

Bearbeiter:

Stefan Plafka
Jochen Strunck
Ludwig Hoegner

Eingereicht am: 20. Februar 2005

Gliederung

1. Einleitung 3

2. 3D-Objekte Tetraedisieren 4

2.1. Einlesen der Oberflächendateien 4

2.1.1. Einlesen in Tetgen 4

2.1.2. Einlesen in Netgen 5

2.2. Tetraedisieren 5

2.2.1. Tetraedisierung in Tetgen 5

2.2.2. Tetraedisieren in Netgen 6

2.2.3. Fehlerkorrektur der Ausgabedateien 6

2.3. Erzeugung der endgültigen Datensätze 6

2.3.1. Festlegung und Auswahl der gewünschten Qualität 7

2.3.2. Speicherung der Ausgabedaten 7

3. Deformation 8

3.1. Finden der Controlmesh Flächen 8

3.2. Beschleunigen des Vorgangs 11

4. Darstellung der hochauflösenden Oberfläche 12

4.1. Darstellen des hochauflösenden Oberfläche 12

4.2. Berechnung der Normalen 13

4.3. GPU Beschleunigung 14

5. Erweiterte Rendering Techniken 16

5.1. Erstellen einer Felloberfläche 16

5.2. Dynamisches Fell 18

5.3. Darstellung 18

5.3.1. Glanzlicht 18

5.3.2. Fell auf bestimmte Teile des Objektes begrenzen 19

6. Ergebnisse 20

6.1. Laufzeiten der verschiedenen Versionen 20

7. Literaturverzeichnis 21

1. Einleitung

Eine Deformationssimulation in Echtzeit von Volumen Objekten hilft Anwendern physikalische Begebenheiten nachzustellen. Die korrekte Berechnung von Volumenobjekten [1] ist sehr rechenintensiv und kann deshalb nur mit niedrig aufgelösten Objekten durchgeführt werden. Um trotzdem eine realistische Darstellung der Oberfläche zu erhalten, wird eine hochauflösende Oberfläche an das niedrigauflösende Objekt (Controlmesh) gebunden (vgl. ***Abb. 1.1***). Diese muss sich bei Bewegungen und Verzerrungen genauso verhalten wie das Controlmesh.

In den folgenden Kapiteln wird die Vorgehensweise mithilfe einer gewichteten Interpolation, sowie verschiedenen Verfahren zur Beschleunigung der Angleichung und der Anzeige erläutert. Weiterhin werden einige Methoden zur naturgetreueren Darstellung der Objekte präsentiert. Abschließend werden die Laufzeitergebnisse verglichen und auf eventuelle Verbesserungen hingewiesen.

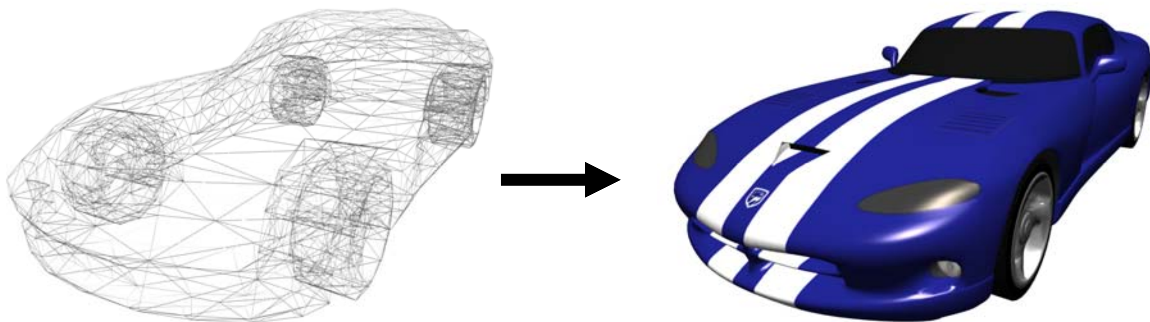


Abb. 1.1: *Hochauflösender Mesh wird an Controlmesh gebunden.*

2. 3D-Objekte Tetraedisieren

Der erste Schritt der Simulation ist die Erzeugung der nötigen Eingangsdaten. Hierfür werden verschiedene Auflösungen des zu simulierenden Objektes benötigt. Diese liegen in unserem Fall als obj-Dateien vor, wie sie von 3D-Kontruktionsprogrammen wie Maya erzeugt werden. Diese müssen von reinen Oberflächenmodellen in Volumenmodelle umgewandelt werden, die dann aus Tetraedern bestehen. Diese Aufgabe übernimmt ein Tetraedisierer. Hier wurden die beiden Programme Tetgen und Netgen verwendet. Die Volumenobjekte, die dabei erzeugt werden, dienen anschließend als Kontrollmesh in der Simulation.

2.1. Einlesen der Oberflächendateien

Wir beginnen die Umwandlung der Objekte mit der Anpassung des Dateiformats, dabei werden die Punkte und Dreiecksoberflächen aus der .obj-Datei ausgelesen und in einem anderen, für den Tetraedisierer lesbaren Format gespeichert.

2.1.1. Einlesen in Tetgen

Für Tetgen werden die Punkte und Oberflächen in einer .poly-Datei gespeichert. Dabei sind die Punkte mit Koordinaten und einem Index versehen, in den Oberflächensegmenten sind die Indizes der zu ihnen gehörenden Punkte vermerkt. Leider sind die Eingangsdaten nicht immer. Befinden sich Löcher in der Oberfläche, so kann der Tetraedisierer keine Volumenbestimmung durchführen und bricht ab.
Für Einzelne abstehende Dreiecke findet eine Fehlerkorrektur statt. Sie sucht im ersten Schritt nach Kanten von Fläche, die nur einmal vorkommen. Denn dann hat dieses Oberflächensegment an dieser Stelle keinen Nachbarn und ist daher fehlerhaft.
Ein weiteres Problem für Tetgen sind sich kreuzende Flächen, weil er dann nicht mehr eindeutig den Oberflächenverlauf festlegen kann. Tritt ein solcher Fall auf, sendet Tetgen eine Fehlermeldung. Diese wird vom Konvertierungsprogramm abgefangen und so verarbeitet, dass es eine der beiden kreuzenden Flächen auf einen Punkt der anderen Fläche zusammenschrumpft. Dabei werden alle Punkte dieser Fläche ebenfalls auf diesen einen Punkt verschoben, wodurch alle Flächen, die einen dieser Punkte benutzen die entstandene Lücke schließen. Anschließend werden Flächen, die dadurch auf Linien zusammengeschrumpft sind – Was dann passiert, wenn z.B. ein Dreieck mit einer Seite mit dem zusammengeschrumpften Dreieck verbunden ist und dann 2 Punkte auf einen zusammenschmelzen – ebenfalls auf denselben Punkt zusammengeschrumpft, auf den schon die kreuzende Fläche reduziert wurde.
Beide Korrekturverfahren führen dazu, dass einzelne Punkte nicht mehr verwendet werden, weil die zu ihm gehörenden Flächen gelöscht wurden. Um Tetgen die Arbeit zu erleichtern werden in einem letzten Korrekturschritt alle nicht verwendeten Punkte gelöscht, anschließend die bestehenden Punkte wieder mit neuen fortlaufenden Indizes ausgestattet und diese entsprechend in den Flächen vermerkt.
Leider können mit diesem Verfahren nicht alle Fehler korrigiert werden, so dass es sein kann, dass fehlerhafte Modelle gar nicht eingelesen werden können oder lediglich bestimmte Auflösungen nicht funktionieren.

2.1.2. Einlesen in Netgen

Für den Netgen gestaltet sich die Anpassung der Eingangsdaten leichter: Sie werden vom obj-Format in das stl-Format konvertiert. Hier stehen in der Definition jedes Oberflächenelements seine Punkte mit Koordinaten. Die Indizierung der Punkte entfällt. Auch verfügt Netgen über eigene Fehlerkorrekturen, doch kann auch er bei vielen Fehlern oder Spezialfällen keine korrekten Ergebnisse liefern.

2.2. *Tetraedisieren*

Nachdem nun die Eingangsdaten im passenden Format für den jeweiligen Tetraedisierer vorliegen, werden alle Eingangsauflösungen des Objekts mit 5 verschiedenen Qualitätsstufen, sprich Elementezahlen in Volumenmodelle umgewandelt. Dafür werden die Tetraedisierer über Kommandozeilenaufrufe gestartet. Da sie als eigene Programme vorliegen, kann jederzeit eine neuere Version eingesetzt werden, ohne unser Umwandlungsprogramm ändern zu müssen.

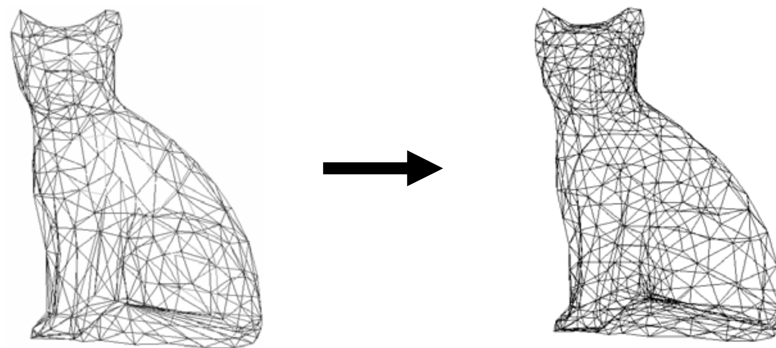


Abb. 2.1: *Oberflächenmodell vor und Volumenmodell nach der Tetraedisierung*

2.2.1. Tetraedisierung in Tetgen

Zur Erzeugung verschiedener Qualitätsstufen einer Eingangsdatei greifen wir in Tetgen auf die Option „Quality-Mesh-Generation“ (Option –q) zurück. Dieser Parameter beeinflusst direkt die Genauigkeit der Berechnung und damit die Tetraederzahl. Tetgen gibt standardmäßig 3 Dateien aus: Eine node-Datei, sie enthält indiziert alle Punkte mit ihren räumlichen Koordinaten, eine ele-Datei, sie enthält alle Tetraeder und für jeden Tetraeder die Indizes der Punkte, die ihn bilden, und eine face-Datei, die die Oberfläche spezifiziert. Diese face-Datei wird für unseren Zweck nicht benötigt und ihre Erzeugung daher abgeschaltet (Option –F). Die node-Datei und die ele-Datei entsprechen bereits dem Format des Simulationsprogramms und werden für jede Qualitätsstufe jeder Eingangsauflösung abgespeichert. Außerdem schalten wir im Tetgen die Kommandozeilenausgaben aus (Option –Q). Im Kommandozeilenaufruf sieht das wie folgt aus:

Tetgen –FQq<Wert für q> <Umzuwandelnde poly-Datei>

Für die Erzeugung der verschiedenen Qualitätsstufen wird nun der Wert hinter „q“ verändert: 2.0 ist die gröbste Stufe, je näher man an 1.0 kommt, um so feiner wird der Mesh. Wir erzeugen 5 Qualitätsstufen pro Eingangsdatei mit den Werten 2.0, 1.8, 1.6, 1.4 und 1.2. Diese sind jedoch zwischen 2.0 und >1.0 frei wählbar.

2.2.2. Tetraedisieren in Netgen

Auch Netgen erzeugt über Kommandozeilenaufruf 5 Qualitätsstufen. Allerdings gibt man hier direkt eine von 5 festen Qualitätsstufen an, anstatt wie bei Tetgen selber an Parametern zu schrauben. Die 5 Stufen heißen nach zunehmender Qualität sortiert: „very coarse“, „coarse“, „moderate“, „fine“, „very fine“. Diese werden in den Kommandozeilenaufruf eingesetzt. Außerdem wird die Ausgabedatei angegeben, die von Netgen im vol-Format gespeichert wird. Aus dieser Datei werden nach der Tetraedisierung für jede Qualitätsstufe jeder Eingabedatei die node- und ele-Datei erzeugt, wie sie auch von Tetgen generiert wird. Mit dem Befehlt „-batchmode“ wird die Kommandozeilenausgabe von Netgen unterdrückt. Der gesamte Aufruf sieht dann wie folgt aus:

```
Ng -geofile=<Umzuwandelnde stl-Datei> -meshfile=<Ausgabedatei im vol-Format>
   -<Qualität> -batchmode
```

2.2.3. Fehlerkorrektur der Ausgabedateien

Durch das Tetraedisieren entstehen manchmal auch neue Fehler in den Daten. So kommt es vor, dass in der node-Datei einzelne Punkte gespeichert werden, die zu keinem Tetraeder gehören. Solche Punkte müssen entfernt werden, da sie das Simulationsprogramm zum Absturz bringen können.

2.3. Erzeugung der endgültigen Datensätze

Das Tetraedisieren ist nun abgeschlossen, jedoch bleibt noch ein weiterer Schritt zu tun: Aus der Vielzahl an in jeweils 5 Qualitätsstufen erzeugten Datensätze müssen die optimalen für die Simulation ausgesucht und zusammengestellt werden.

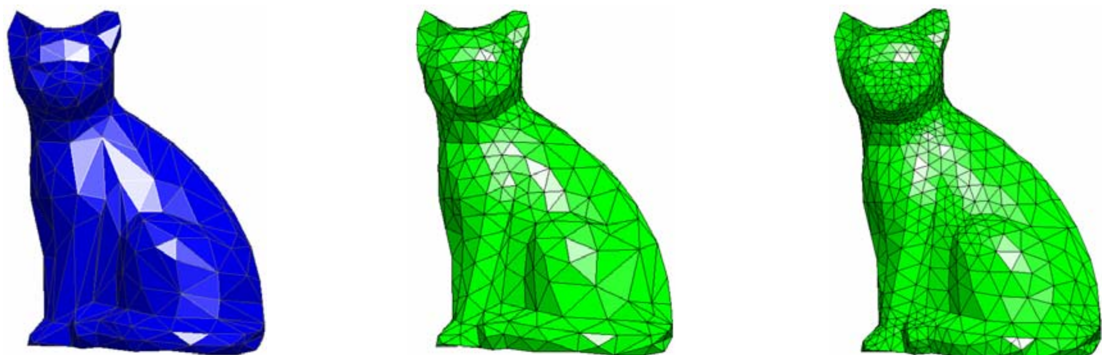


Abb. 2.2: *Links: Originaldatei; Mitte und rechts: Tetraedermodelle aus Netgen, in der Mitte mit Qualitätsstufe „moderate“, rechts mit Stufe „fine“*

2.3.1. Festlegung und Auswahl der gewünschten Qualität

Um die optimale Qualität ermitteln zu können, muss sie zuerst festgelegt werden. Dies geschieht beim Aufruf des Konvertierungstools:

Anzahl der Eingangsdaten:

Dieser Parameter gibt an, wie viele Detailstufen, also obj-Dateien eines Objekts eingelesen werden sollen. Das Programm sucht dann solange nach Dateien mit den Namen objektname_<0...∝>, bis er die gewünschte Zahl gefunden hat.

Anzahl der Ausgabelevel:

Dieser Parameter gibt an, wie viele Qualitätsstufen die endgültige Ausgabe für die Simulation enthalten soll.

Mindestzahl und Höchstzahl:

Diese Parameter geben an, wie viele Elemente das niedrigste, bzw. höchste Ausgabelevel enthalten soll. Gibt es mehr als 2 Ausgabelevel, werden die Elementezahlen der Zwischenlevel errechnet nach der Formel:

Mindestzahl * ((Höchstzahl / Mindestzahl)^(1/(Anzahl Level-1)))^k

Nachdem das Programm nun weiß, wie viele Elemente ein Level haben soll, sucht es die ele-Dateien nach der Datei durch, deren Elementezahl am nächsten am gewünschten Wert liegt. Dabei wird beginnend von der Höchsten Qualitätsstufe der feinsten Eingangsdatei gesucht. Dadurch wird sichergestellt, dass immer die höchst mögliche Qualität verwendet wird, die mit der gewünschten Elementezahl erreicht werden kann.

2.3.2. Speicherung der Ausgabedaten

Im letzten Schritt werden jetzt in aufsteigender Reihenfolge die ausgewählten Qualitätsstufen zusammengefasst. Alle node-Dateien werden in einer einzigen zusammengefasst, ebenso die ele-Dateien, so dass am Schluss je eine node- und ele-Datei mit den gesamten Daten aller Detailstufen für die Simulation vorliegen. Zusätzlich wird bereits eine erste einfache config-Datei für das Objekt angelegt mit der Objektskalierung, Standardmaterial und –textur, so dass man das jetzt fertige Objekt im Simulationsprogramm bereits betrachten kann.

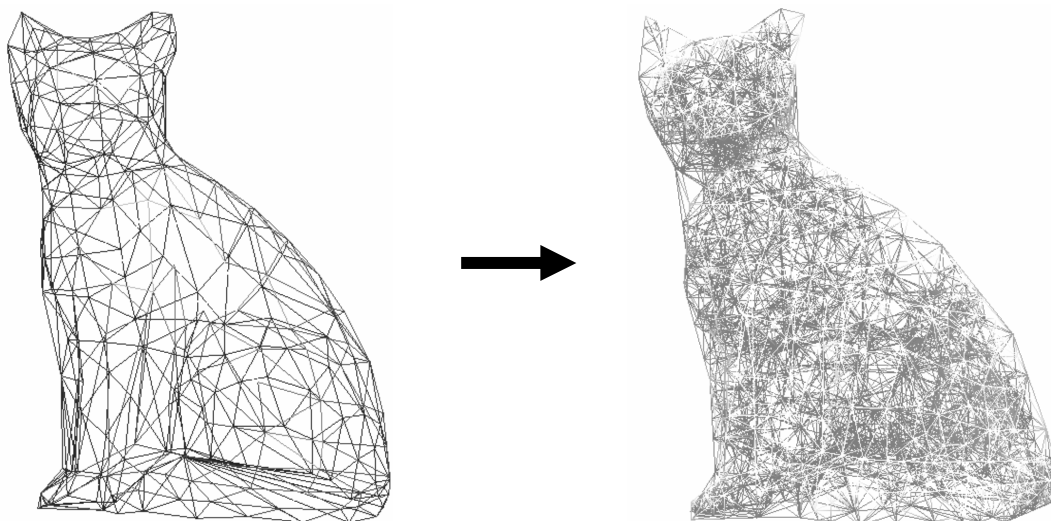


Abb. 2.3: *Links: Originaldatei mit 512 Flächen, rechts: Tetraedermodell mit 10.000 Elementen*

3. Deformation

Aufgabe dieses Abschnittes ist die Deformation der hochauflösenden Oberfläche. Auf der linken Seite der ***Abb. 3.1*** ist ein physikalisch korrekt berechnetes Tetraedermodell zu sehen. Die rechte Seite zeigt ein detailliertes Mesh, dessen Punkte dieselben Bewegungen wie die des Controlmeshes vollziehen sollten.

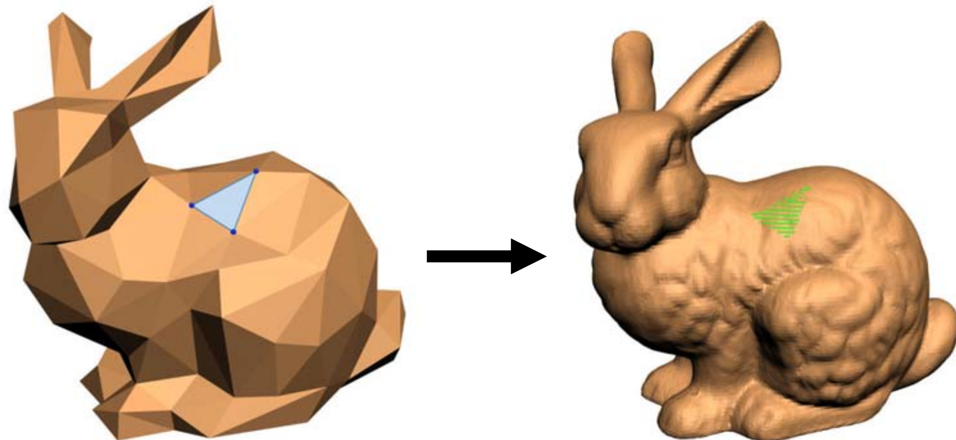


Abb. 3.1: *Physikalisch korrekt berechnetes Volumenobjekt mit dem Ziel der Angleichung*

3.1. Finden der Controlmesh Flächen

Jede Fläche der hochauflösenden Oberfläche soll sich also entsprechend der korrespondierenden Fläche des Control Meshes verhalten. Da zur Definition einer Fläche mindestens 3 Punkte benötigt werden, werden für alle Punkte aller Flächen der detaillierten Oberfläche die jeweils 3 „wichtigsten" Punkte des Control Meshes zur Positionsbestimmung verwendet. Somit wird jeder Hochauflösende Punkt mit diesen drei Kontrollpunkten verbunden, zwischen denen eine gewichtete Interpolation durchgeführt wird, um ihren jeweiligen Einfluss auf die Position des hochauflösenden Punktes festzulegen. Die obige ***Abb. 3.1*** zeigt grün alle Punkte, die auf das links dargestellte blaue Dreieck abgebildet werden sollten. Bewegt oder verformt sich die blaue Dreiecksoberfläche, müssen sich alle grünen Punkte dementsprechend mitbewegen. Durch eine gewichtete Interpolation ist dies möglich Aus dieser Strategie folgt die erste entscheidende Fragestellung: Welches sind die drei besten Kontrollpunkte für jeden hochauflösenden Punkt?

Eine mögliche einfache Lösung besteht darin, den dichtesten Abstand zu den drei Kontrollpunkt zu nehmen. Das Ergebnis sieht auch ziemlich gut aus, doch werden dabei teilweise komplett falsche Zuordnungen getroffen. In der ***Abb. 3.2*** wird ein solches Beispiel gezeigt. Dabei kann der Controllmesh aus sehr große Flächen bestehen. In so einem Fall ist der kleinste Abstand (rot gezeichnet) eine Verbindung zu einem ganz anderen Teilobjekt.

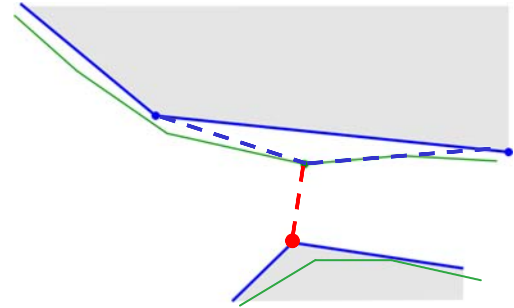


Abb. 3.2: *Finden der Kontrollpunkte mit Hilfe des Punkteabstands*

Eine weitere Möglichkeit die Zuordnung zu treffen, ist den kleinsten Abstand zur Flächenebene zu verwenden und die drei Punkte, die diese Fläche aufspannen, für die Interpolation zu nutzen. Diese Methode liefert komplett verkehrte Zuordnungen. Aus der ***Abb. 3.3*** ist dies gut zu erkennen. Ganz woanders liegende Flächen können sehr kleine Flächenabstände zu dem Punkt haben und werden somit bevorzugt.

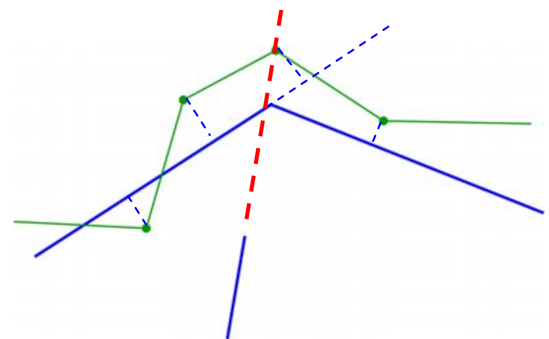


Abb. 3.3: *Finden der Kontrollpunkte mit Hilfe des kleinsten Abstands zur Fläche*

Die nächste Methode zum finden der Kontrollpunkte liefert ähnlich schlechte Ergebnisse. Von jedem Punkt wird ein Lot auf alle Ebenen gefällt. Lässt sich ein Punkt dabei auf einer Ebene abbilden, so werden die Punkte dieser Ebene als Kontrollpunkte verwendet. Lässt sich der Punkt auf keine Ebene Projizieren, so wird die Ebene gewählt, bei der das gefällte Lot des Punktes auf die Ebene am wenigsten außerhalb der Ebene liegt. Auch hier können Flächen gewählt werden, die ganz woanders liegen, wie in ***Abb. 3.4*** gut zu sehen ist.

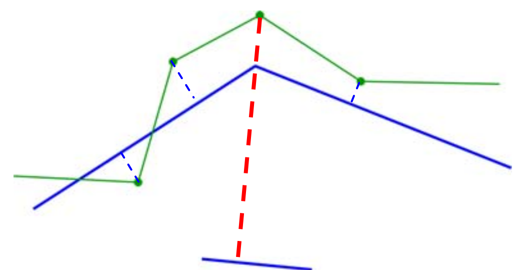


Abb. 3.4: *Finden der Kontrollpunkte mit Hilfe einer Projektion auf die Fläche*

Da keines der vorgestellten Verfahren alleine gute Ergebnisse geliefert hat, wurden für die endgültige Lösung die letzten zwei Verfahren kombiniert. Jeder hochauflösende Punkt wurde erst einmal auf alle Flächen projiziert. Dabei war der Abstand zur Fläche ein Nebenprodukt, das gespeichert wurde. Danach werden die baryzentrischen Koordinaten *(Definition folgt unter Formel 3.1)* dieses Punktes zu den anderen drei Punkten der Fläche berechnet. Mit Berücksichtigung des Abstandes des Punktes zur Fläche wird nun die beste Fläche gewählt, in denen die Werte der baryzentrischen Koordina-

ten möglichst groß sind (am besten positiv). Somit werden Punkte, die weit entfernt zu der Fläche liegen, nicht gewählt, da das Verhältnis mit dem Abstand zu schlecht ist.

Die drei Referenzpunkte der Fläche werden nun mit den baryzentrischen Koordinaten für die Gewichtung zu diesem hochauflösenden Punkt abgespeichert.

Baryzentrische Koordinaten:

Seien $x_1, ..., x_n$ die Eckpunkte eines Simplex[1] im Vektorraum A. Wenn für einen Punkt p aus A folgende Gleichung erfüllt ist,

$$(a_1 + \cdots + a_n)p = a_1 x_1 + \cdots + a_n x_n$$

so nennen wir die Koeffizienten $(a_1, ..., a_n)$ *baryzentrische Koordinaten* von p zu $x_1, ..., x_n$. Die Eckpunkte haben die Koordinaten (1, 0, 0, ..., 0), (0, 1, 0, ..., 0), ..., (0, 0, 0, ..., 1). Baryzentrische Koordinaten sind nicht eindeutig: Für jedes von Null verschiedene b sind $(b\,a_1, ..., b\,a_n)$ ebenfalls baryzentrische Koordinaten von p.

Falls die Koordinaten positiv sind und sich zu 1 aufsummieren, so liegt der Punkt p in der konvexen Hülle von $x_1, ..., x_n$, also dem Simplex mit diesen Eckpunkten.

Stellen wir uns Massen im Verhältnis $a_1, ..., a_n$ an den Eckpunkten des Simplex vor, so liegt der Massenschwerpunkt (das Baryzentrum) in p. Dies ist der Ursprung des Begriffs "baryzentrisch", eingeführt 1827 von August Ferdinand Möbius.

Formel 3.1: *Baryzentrische Koordinaten*

[1] **Simplex**: Bezeichnet ein n-dimensionales Polytop mit der minimalen Anzahl, also n+1 Ecken.

3.2. Beschleunigen des Vorgangs

Bis jetzt müsste man bei der Suche der Kontrollpunkte für jeden hochauflösenden Punkt (n) jede Fläche (m) des Tetraeder Controlmeshes durchlaufen. Diese Methode hat damit eine Laufzeit von $O(n^m)$. Um dies zu beschleunigen, wurden folgende zwei Verbesserungen vorgenommen:

1. Ein neues Array aufbauen, welches nur die Außenflächen des Tetraeder Controlmeshes besitzt.
2. Das Objekt in Raumsegmente aufteilen und nur die Flächen in diesem Segment überprüfen.

Das Objekt wird vom Schwerpunkt aus in seine Segmente geteilt. Dabei sind die X-, Y-, Z-Achsen die Segmentteiler und Grenzflächen werden in jedem benutztem Segment eingetragen, so wie es in ***Abb. 3.5*** zu sehen ist. Alles was zum linken vorderen oberen Segment eingetragen ist, wurde blau gekennzeichnet.

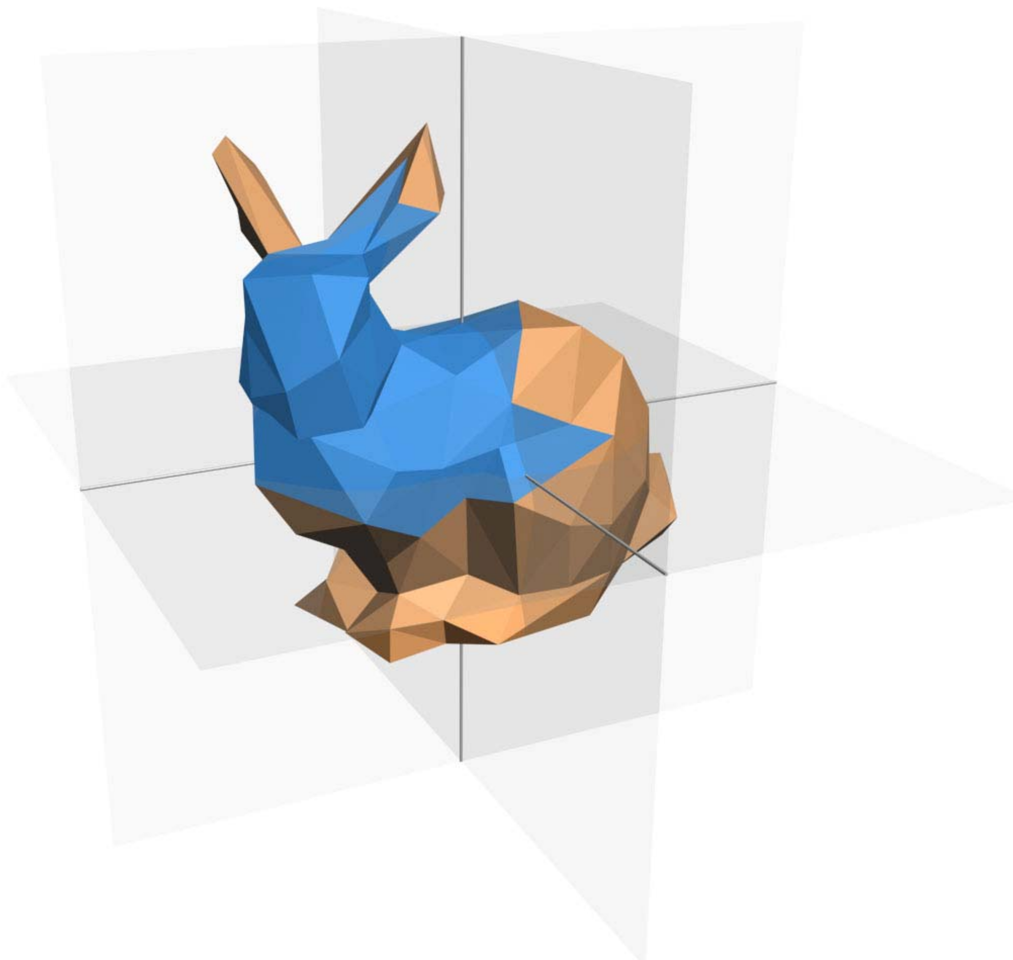


Abb. 3.5: *Einordnen der Kontrollflächen in Raumsegmente*

4. Darstellung der hochauflösenden Oberfläche

Zur Darstellung der Oberfläche wird das Framework von der physikalischen Berechnung von Objekten benutzt. Da die Darstellung plattformunabhängig sein soll, wird openGL[2] mit dem Werkzeug GLUT [3]benutzt. Durch eine Checkbox-Anwahl wird dann die neue hochauflösende Oberfläche angezeigt.

4.1. Darstellen des hochauflösenden Oberfläche

Nachdem alle Vorbereitungen mit dem Binden auf das Controlmesh erledigt sind, braucht man jetzt nur die Punkte des hochauflösenden Meshes mit Hilfe openGL darzustellen. Auf die Originalkoordinaten jedes hochauflösenden Punktes wird eine Verschiebung wie folgt addiert:

$$P_{neu} = P_{Original} + CP_1 \cdot w_1 + CP_2 \cdot w_2 + CP_3 \cdot w_3 \qquad \textit{Formel 4.1}$$

Hierbei ist $P_{Original}$ der unveränderte Punkt, CP_1 der Verschiebungsvektor des 1. Kontrollpunktes und w_1 dessen Gewichtung, die mit den baryzentrischen Koordinaten (Formel 3.1) berechnet wurde.
Liegt keine Bewegung und Verzerrung vor, wird $CP_1 \cdot w_1 + CP_2 \cdot w_2 + CP_3 \cdot w_3 = 0$ und P_{neu} ist gleich $P_{Original}$.

Nun sieht das Ergebnis noch sehr kantig aus (siehe ***Abb. 4.1***), was daran liegt, dass jeder Punkt mehreren Flächen zugeordnet ist und es somit mehrere Normalen für einen Punkt gibt. Dieses wird im nächsten Unterpunkt behoben.

Abb. 4.1: *Objekt nicht geglättet*

---

[2] **openGL** (Open Graphics Language):
Dies ist eine Spezifikation für ein plattform- und programmiersprachen-unabhängiges API (Application Programming Interface) zur Entwicklung von 3D-Computergrafik. Der OpenGL-Standard beschreibt etwa 250 Befehle, die die Darstellung komplexer 3D-Szenen in Echtzeit erlauben. Hersteller können jedoch auch eigene Erweiterungen definieren.

[3] **GLUT** (openGL Utility Toolkit):
GLUT erledigt plattformunabhängig viele ständigen Aufgaben wie z.B. das Erstellung von Fenstern oder Tastaturein- und Ausgaben.

4.2. Berechnung der Normalen

Um eine scheinbar glatte Oberfläche zu erhalten, darf ein Punkt nur eine Normale besitzen. Hierzu wird für alle Flächen die Flächennormale auf ihre anliegenden Punkte addiert. Danach wird jeder Punkt normalisiert.

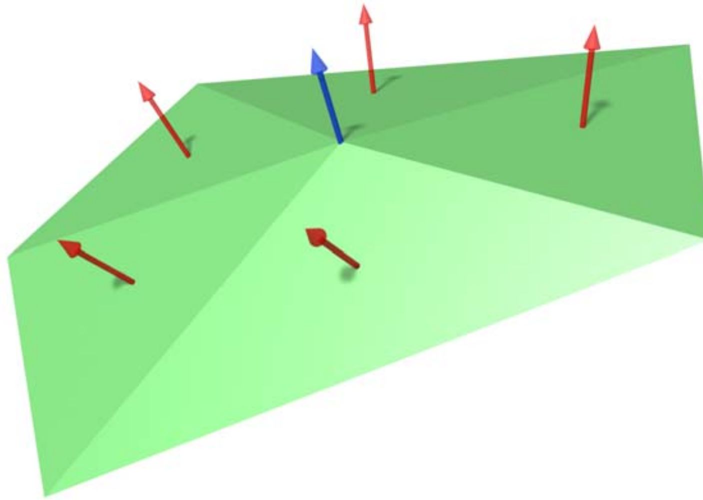


Abb. 4.2: *Normalenberechnung*

Wie in ***Abb. 4.2*** zu erkennen, hat der mittlere Punkt die Durchschnittsnormale seiner anliegenden Flächen. Das Ergebnis in ***Abb. 4.3*** sieht jetzt glatt aus.

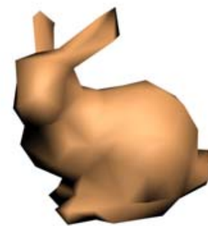


Abb. 4.3: *Objekt geglättet*

4.3. GPU Beschleunigung

Aufgrund der hohen Parallelisierung sind die Grafikkarten bei der Berechnung einer Operation auf viele Daten meist schneller als die CPU. Die GPU[4] ist also somit von ihrer Hardware Architektur schon für Interpolationen, wie sie im letzten Abschnitt beschrieben wurden, optimiert. Ein anderer wichtiger Punkt ist, dass die 3D-Geometrie selbst für animierte Objekte nicht in jedem Frame über den Bus geschoben werden muss. Vielmehr genügt es, wenn der Grafikkarte einige wenige Kontrollwerte übergeben werden und die hochaufgelöste Geometrie dazwischen interpoliert wird.

Der gegenwärtige Aufbau der Grafikkarten besteht aus zwei wichtigen Einheiten: dem Vertex Shader[5] und dem Fragment Shader (siehe ***Abb. 4.4***). Der Vertex Shader ist für die Platzierung der Geometrie im Raum zuständig, der Fragment Shader übernimmt die Berechnung der Farbwerte der einzelnen Bildpunkte. Anzumerken ist, dass die Fixed Function Pipeline nur in genau der Reihenfolgen wie in ***Abb. 4.4*** zu sehen ausgeführt werden kann. Es ist auch nicht möglich, einen der Shader einzeln aufzurufen.

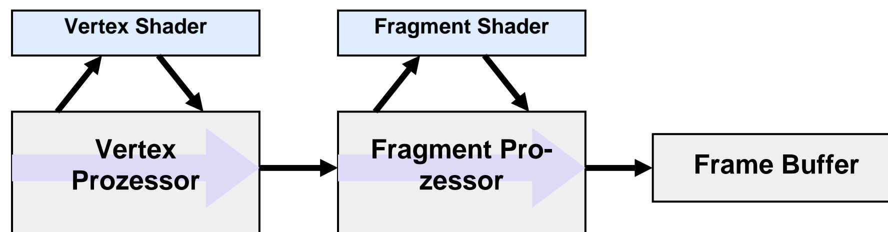


Abb. 4.4 *Fixed Function Pipeline*

Der Fragment Shader ist auf heutigen Grafikkarten deutlich Leistungsfähiger und schneller als der Vertex Shader. Die Positionsdaten für die Deformation werden aber schon im Vertex Shader benötigt. Um jetzt trotzdem in den Vorteil der höheren Geschwindigkeit des Fragmentshaders zu kommen, benutzen wir ein 2-Pass-Rendering.
Im ersten Durchlauf, dem Pre-Pass, wird die Deformations-Interpolation im Fragment Shader berechnet und in einen Offscreenbuffer ausgegeben. Anschließend kann der Vertex Shader im zweiten Durchlauf die Offscreen Inhalte verwenden, um die Vertices zu positionieren. Dies ist auch gleichzeitig das Nadelöhr an diesem Vorgehen. Zum Zeitpunkt der Erstellung dieser Arbeit gab es folgende Möglichkeiten, die Offscreen Daten in den Vertexbuffer zu laden:

- Texture-fetch in Vertex-Shader
- Copy to Bufferobject auf der GPU
- Superbuffer (derzeit nicht von Nvidia unterstützt)

Für die Standartdarstellung der Oberfläche verwendet diese Arbeit jeweils einen Texture-fetch für Vertexpostion und Normale. Für das Fell-Shading (siehe Abschnitt 5 Erweiterte Rendering Techniken) wird ein Copy to Bufferobject auf der GPU verwendet.

---

[4] **GPU** (Graphics Processing Unit): Recheneinheit der Grafikkarte.
[5] **Shader:** ein Programm auf der Grafikkarte zur Berechnung der Vertex-Position oder Fragment-Farbe.

Die Genaue Vorgehensweise sei anhand folgender Grafik (***Abb. 4.5***) erläutert. Zuerst werden die Vertex-Positionen des Hochauflösenden Meshes in eine Positionstextur geschrieben. Hierbei werden anstelle der normal vorgesehenen *RGB*-Werte die XYZ-Werte in jedes Element der Textur gesetzt. Sodann werden für die Berechnung von jedem Vertex 3 Deformations-Vektoren mit den zugehörigen Gewichten benötigt (siehe ***Formel 4.1***). Hierfür werden 2 *RGBA*-Texturen angelegt, in denen die Texturkoordinaten zu den entsprechenden Kontrollpunkten der Deformations Textur eingetragen sind. In die zwei letzten Werte der zweiten Textur (*BA*) werden die Gewichte geschrieben. Da die Summe der drei Gewichte immer 1 ergibt, kann der dritte Wert problemlos auf der GPU berechnet werden.
Analog wird für die Berechnung der Normalen vorgegangen, so dass am Ende je eine Textur für Vertexposition und Normale ausgegeben wird.

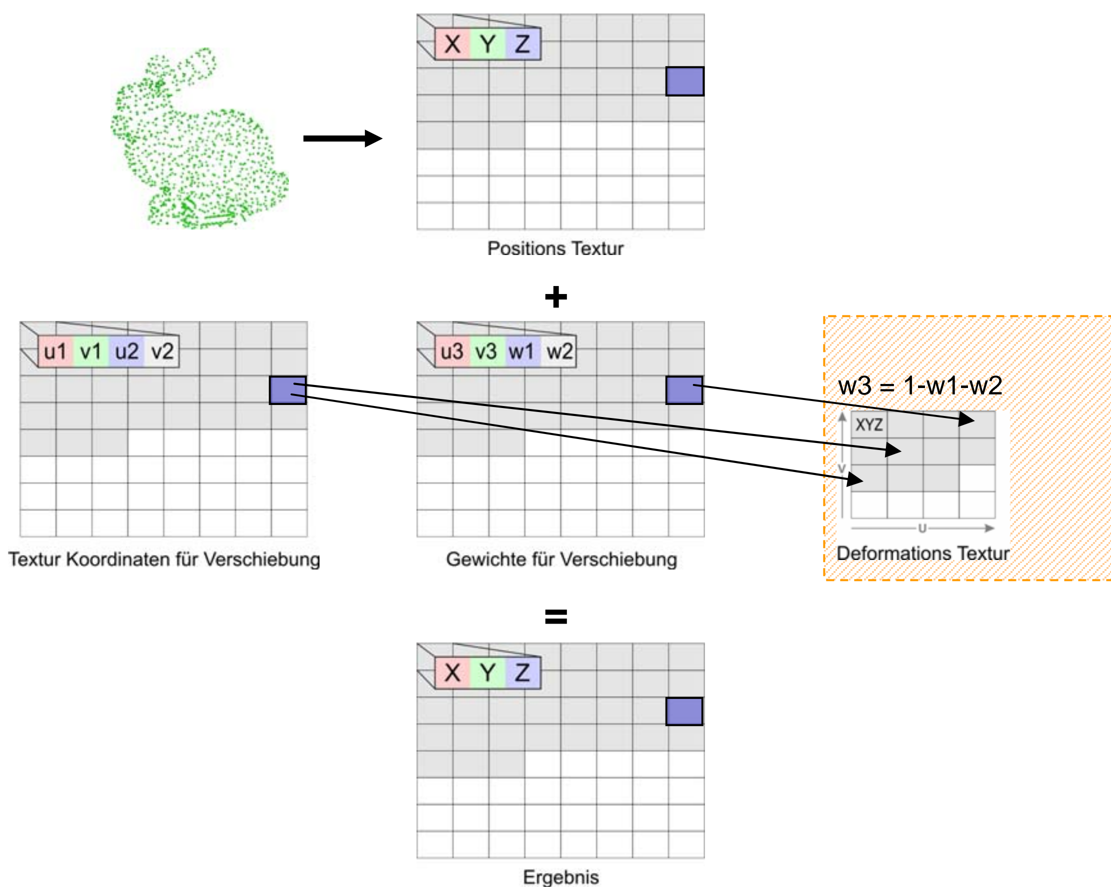


Abb. 4.5 *Pre-Pass der GPU Beschleunigung. Zeigt den Ablauf für die Berechnung eines Elements (Blaues Rechteck)*

Die ersten drei Texturen müssen nur einmalig generiert und auf die GPU geladen werden. Für die laufende Darstellung muss jetzt nur noch die niedrig auflösende Deformationstextur (orange schraffiert in ***Abb. 4.5***) in jedem Frame auf die Karte geladen werden, und somit sinkt der Bustransfer.

5. Erweiterte Rendering Techniken

Bislang haben wir eine Möglichkeit, feine Oberflächen über einen groben Kontrollmesh zu bewegen. Diese Oberfläche verfügt zwar über eine Textur, ist aber ansonsten glatt. Häufig wird der Realitätsgrad gesteigert, indem Glanzlicht und Reflektion des Materials angepasst werden. Bei unserem Testobjekt, einem Hasen, reicht das aber nicht aus, sondern wir benötigen eine andere Oberfläche. Die nächsten Abschnitte beschreiben, wie eine Felloberfläche erstellt wird.

5.1. Erstellen einer Felloberfläche

Würden für jedes Fellhaar Polygone generiert, ist das Ergebnis eine Geometrie mit mehreren Millionen Polygonen. Diese Mengen sind nur schwer in Echtzeit darzustellen. Ein anderes Problem ist, dass mit Geometrie kaum eine realistische Darstellung von Fell möglich ist. Die Haare wären viel kleiner als ein Pixel in der Ausgabe und es würden sich somit selbst mit Antialiasing Sampling-Probleme ergeben, mit dem Ergebnis von Moiré-Bildung und Flimmern bei Bewegung.
Deshalb wird in dieser Arbeit ein Volumen-Textur-Ansatz verwendet. Das Volumen wird durch mehrere teiltransparente Schichten dargestellt. Diese Schichten werden durch Extrusion der Grundfläche in Richtung der Normalen erstellt (siehe ***Abb. 5.1***).

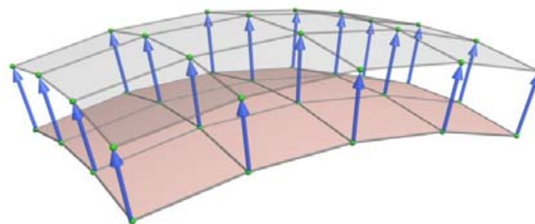


Abb. 5.1 *Schichtenerstellung: Extrusion in Richtung der Normalen*

Für die Generierung der Fell-Textur wird eine Textur (z.B. 128*128) mit vier Elementen (RGBA) pro Texel[6] angelegt. Die Position und Form der Haare wird dadurch bestimmt, dass dem Alphakanal jedes Texels ein Zufallswert zugewiesen wird (siehe ***Abb. 5.2***). Der Wert „0“ steht für transparent und somit für Haar-Zwischenräume, größere Werte ergeben ein mehr oder weniger langes Haar.
Auf diesen Daten basierend wird für jede Schicht eine Textur angelegt. Hierbei werden je nach Abstand der Schicht zum Fellansatz Anpassungen vorgenommen: Um eine Selbstabschattung zu simulieren, werden die Farben Richtung Fellansatz dunkler. Die Werte des Alphakanals werden Richtung Fell-Spitze durchsichtiger, so dass die Spitzen dünner werden. Bei geringem Alphawert ist die Stelle schon vor Erreichen der obersten Schicht durchsichtig und es entstehen somit auch kürzere Haare.

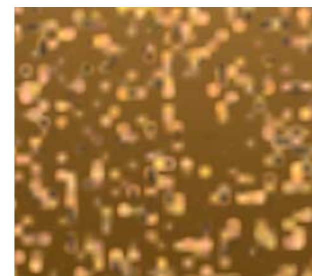


Abb. 5.2 *Fell Textur einer Schicht*

[6] **Texel**= Texture Element. Pixel einer Textur.

Abb. 5.3 *Links: einzelne Fellhaare. Rechts: feines Fell*

Um die Felltextur auf der Oberfläche zu Platzieren werden Texturkoordinaten benötigt. Die Schwierigkeit ist nun, dass das Fell ohne Verzerrungen auf beliebigen Oberflächen liegen muss. Der Abstand der einzelnen Haare sollte schließlich überall gleich sein.

Die Qualitativ beste Lösung hierfür wären „Lapped Patches“ [5]. Diese Methode hat aber den Nachteil, dass sie relativ langsam und sehr aufwändig zum Programmieren ist. Ein anderer einfacher Ansatz sind die „Box Texture-Coordinates“. Hierbei wird für jedes Oberflächensegment bestimmt, welcher Würfelseite es am meisten zugewandt ist. (vgl. ***Abb. 5.4***; z.B. grüne Flächen werden der oberen Seite zugeordnet). Anschließend werden den Flächen die Planaren Texturkoordinaten der entsprechenden Würfelseite zugewiesen, wobei jede Fläche nur genau einer Seite zugeordnet wird.

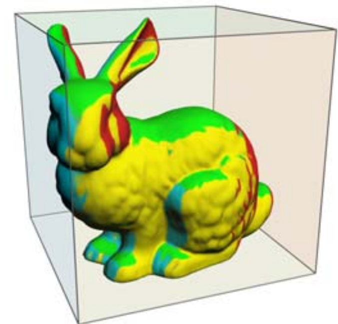


Abb. 5.4 *Box TextureCoords.*

An dieser Stelle sei noch anzumerken, dass aus technischen Gründen nur ein Texturkoordinatenpaar pro Vertex[7] möglich ist (OpenGl Bufferobjects). Um das zu erreichen wird jeder Vertex der an einer Textur-grenze liegt in zwei Vertices mit je einem Texturkoordinatenpaar aufgeteilt. Durch diese Aufteilung sind die Flächen aber nicht mehr zusammenhängend und die Flächennormalen werden nicht mehr über die Texturgrenzen hinweggeglättet, mit der folge, dass ein Spalt im Fell entstehen würde (siehe ***Abb. 5.5***) . Um das zu vermeiden, werden alle IDs der Vertices die an einer Texturgrenze liegen in eine Liste Geschrieben und anschließend deren Normalen zusammen geheftet.

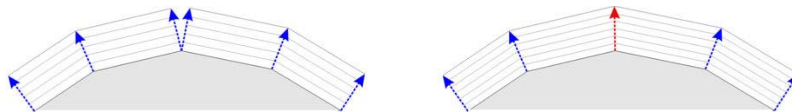


Abb. 5.5 *Links: die Normalen an den Texturgrenzen bilden einen Spalt im Fell. Rechts: die Normalen der beiden Vertices wurden geglättet.*

[7] **Vertex:** Punkt im 3D-Raum. Ein Eckpunkt einer Fläche.

5.2. Dynamisches Fell

Wenn das Objekt verformt wird, sollte das Fell entsprechend „hinterher wehen“ (siehe ***Abb. 5.6***). Hierfür wird die Geometrie des aktuellen Frames mit der des letzten Frames verglichen und daraus an jedem Vertex der Geschwindigkeitsvektor errechnet. Jetzt werden die Fellschichten entgegengesetzt zum Geschwindigkeitsvektor verschoben. Dabei ist der Einfluss auf die Schichten nahe dem Fellansatz sehr gering und steigt mit der Entfernung.
Nun muss die Fellbewegung noch begrenzt werden, da es ansonsten passieren könnte, dass sich das Fell in die Grundfläche verschiebt und somit nicht mehr sichtbar ist.

Abb. 5.6 *Bewegung nach rechts* *Bewegung nach links*

5.3. Darstellung

5.3.1. Glanzlicht

Das Glanzlicht von Fell unterscheidet sich in einigen Punkten von dem einer normalen Fläche. Trivialerweise muss das Glanzlicht mit dem Alphakanal vom Fell skaliert werden, damit nicht die Fellzwischenräume glänzen. Ein anderer Punkt betrifft den Winkel in dem das Glanzlicht auftritt, da das Fell senkrecht zur Grundebene steht. Für eine korrekte Fell-Beleuchtung müssen hier „shaded Lines“ verwendet werden, oder die Reflektionskurve wird verändert. Letzteres sei hier erklärt: Auf glatten Oberflächen ist das Glanzlicht an der Stelle, an der der Einfallswinkel des Lichtes gleich dem Ausfallswinkel in Richtung Betrachter ist (siehe ***Abb. 5.7*** der Senkrechte Strich). Wird nun das Glanzlicht nach außen geschoben so wird ein samtiger Glanz Effekt erreicht.

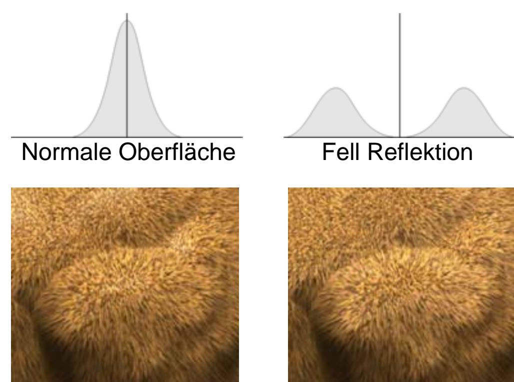
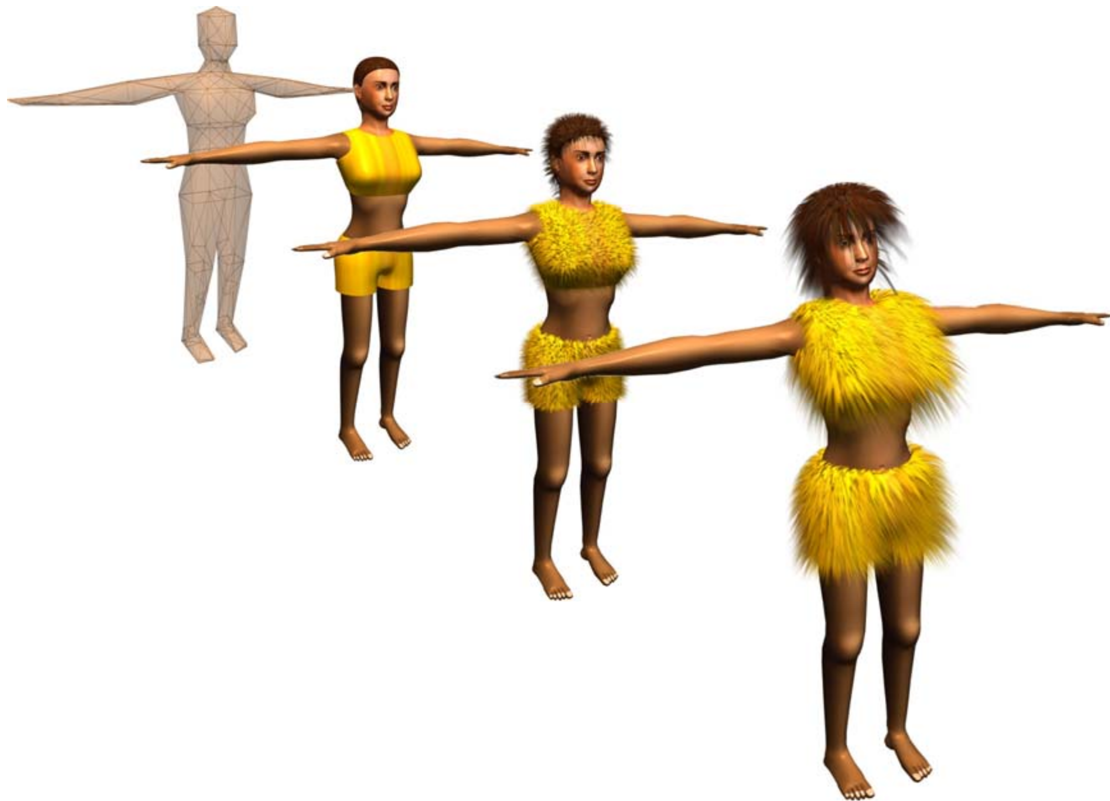


Abb. 5.7 *ReflektionsKurve*

5.3.2. Fell auf bestimmte Teile des Objektes begrenzen

Um noch mehr Einfluss auf das Rendering zu haben sollte das Fell an verschiedenen Stellen unterschiedlich eingefärbt werden können und noch wichtiger: bestimmte Stellen sollten ohne Fell dargestellt werden können.
Für das Färben des Fells wird eine zweite Textur mit unabhängigen Texturkoordinaten verwendet. Wird jetzt ein weißes Fell mit leichten Helligkeitsvariationen generiert kann die Farbe mit einer einfachen Multiplikation der Fell-Textur und der Farb-Textur im Fragment Shader angepasst werden. Um bestimmtet Stellen des Fells freizustellen wird der Alphakanal der Farb-Textur verwendet. An Stellen mit Alphakanal-Wert Null werden die Fellschichten transparent gezeichnet (discard - Befehl im Fragmentshader).
Jetzt wird noch das Material der Grundfläche angepasst. Da Flächen unter dem Fell verschattet sind, muss ein Schatten Effekt simuliert werden. Dazu wird auf Flächen unter dem Fell die Textur abgedunkelt und außerdem das Glanzlicht schwächer und diffuser dargestellt, als auf Fell-freien Flächen.



6. Ergebnisse

6.1. *Laufzeiten der verschiedenen Versionen*

Vergleich der 5 Renderpfade bei verschiedenen Objekten mit unterschiedlichen Polygonzahlen. Die Ergebnisse beziehen sich auf ein System mit P4 2.8 GHz und einer GeForce 6800 GT. Alle Tests wurden mit animierter Deformation und „per-pixel lighting[8]" durchgeführt.

Objekt	Polygone	Tetraeder	CPU	GPU 1 Pass	GPU 2 Pass	Fell CPU	Fell GPU
Bunny	8k	630	155	144	162	20,0	39,0
BunnyTex	8k	630	157	141	157	18,7	33,0
Bunny	8k	11206	57	53	55	15,7	26,0
Bunny	64k	630	33	52	73	2,5	12,5
Bunny	64k	11206	24,5	27,5	37	2,0	11,0
Viper	27k	770	141	146	141	42,5	39,2
Viper	36k	7004	68	79	79	17,0	19,5
Sarah	16k	350	196	230	234	41,0	50,0

Tabelle 6.1: *Laufzeiten verschiedener Objekte in Frames Per Second (FPS) auf Windows XP.*

Objekt	Polygone	Tetraeder	CPU	GPU 1 Pass	GPU 2 Pass	Fell CPU	Fell GPU
Bunny	8k	630	183	199	N/A	20,0	N/A
BunnyTex	8k	630	157	182	N/A	19,0	N/A
Bunny	8k	11206	71	68	N/A	18,0	N/A
Bunny	64k	630	45	54	N/A	2,5	N/A
Bunny	64k	11206	24	32	N/A	2,0	N/A
Viper	27k	770	185	222	N/A	39,0	N/A
Viper	36k	7004	63	94	N/A	17,5	N/A
Sarah	16k	350	173	307	N/A	40,0	N/A

Tabelle 6.2: *Laufzeiten verschiedener Objekte in Frames Per Second (FPS) auf Linux.*

Zum aktuellen Zeitpunkt läuft das Programm nur mit einem Thread. Durch Erweiterung auf mehrere Threads würden die Volumenberechnung und die Berechnung der Grafikausgabe parallel ablaufen, wodurch sich die Framerate erhöhen könnte. Unter Linux sind die GPU-Varianten derzeit nicht lauffähig, da entsprechende Treiber der Hersteller fehlen.
Wie aus den beiden Tabellen zu erkennen, läuft die Darstellung unter Linux in der Regel mit einer höheren Framezahl. Dieses ist auf die schnellere Volumentetraederberechnung zurückzuführen.

[8] **per-pixel lighting:** Die Beleuchtungs- Berechnung wird für jedes Pixel durchgeführt.

7. Literaturverzeichnis

[1] A Multiresolution Approach for Real-Time Simulation of Deformable Objects, Joachim Georgii

[2] Dynamic Fur, Tomohide Kano
http://www.ati.com/developer/indexsc.html

[3] Hair & Fur, Anand Mathew

[4] Fur, Hugues Hoppe
http://research.microsoft.com/~hoppe/

[5] Lapped Textures, E. Praun, A. Finkelstein, H. Hoppe
http://www.cs.princeton.edu/gfx/proj/lapped_tex/