Interaktive Manipulation von Geometry-Images mit Hilfe der GPU

Studienarbeit

Vorgelegt von
Tobias Ritschel

UNIVERSITÄT KOBLENZ · LANDAU

Institut für Computervisualistik
Arbeitsgruppe Computergraphik

Betreuer und Prüfer:
Prof. Dr.-Ing. Stefan Müller

Oktober 2005

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit beschreibt ein System zur interaktiven Manipulation hoch aufgelöster virtueller Körper (sog. *Meshes*) in Echtzeit. Die dazu momentan verwendeten Methoden, sind technik-dominiert, erfordern viel Übung, dauern lange und skalieren oft schlecht. Dem gegenüber wird hier ein System beschrieben, das mit Hilfe der *GPU*, einer dafür besonders geeigneten Struktur zur Speicherung von Geometrie (*Geometry-Images*) und der Metapher des *Mesh-Painting* einen virtuellen Körper performant und einfach manipulieren kann.

Die immer größer werdenden Möglichkeiten der Grafik-Hardware ermöglichen es, immer detailliertere Modelle darzustellen, jedoch wird die Erstellung und Manipulation solcher Modelle (das sog. *Modelling*) ebenfalls zunehmen schwieriger. Die Komplexität der anfallenden Aufgaben, macht es bis jetzt oft unmöglich, auch diese auf der GPU durchzuführen. Es wurde daher eine Datenstruktur vorgeschlagen (GGH02), die Objekte so vereinfacht beschreiben kann, dass diese von einer GPU nicht nur dargestellt, sondern auch verarbeitet werden können: sog. *Geometry-Images*.

Hier soll kurz beschrieben werden, wie virtuelle Objekte gängigerweise erstellt werden. Die wichtigste Aufgabe ist die Definition der Oberfläche. Dazu wird meist eine Anzahl von Punkten (*Vertices*) und Flächen (*Faces*) erzeugt, wobei die Flächen entweder plan oder abgerundet (Bezier-Patches, B-Spline Surfaces, NURBS oder Subdivision Surfaces) sein können. Nutzer sind in allen Fällen für weite Teile effizienten und exakten Verwendung verantwortlich. Die Bedingungen für „exakte und effiziente Verwendung“ sind verschieden und auch verschieden schwer einzuhalten. So kann ein System z. B. zwar automatisch verhindern, dass Flächen doppelt erstellt werden, andere Bedingungen sind aber in gängiger Software nicht automatisiert oder auch nicht automatisierbar. Eine andere Bedingung besagt, dass an einer Stelle mit kleinen Details viele Vertices verwendet werden müssen. Auch ist es wichtig, dass die entstehenden Flächen akzeptable Proportionen zeigen: das

Verhältnis zwischen Umkreis und Inkreis eines Dreiecks sollte möglichst ausgewogen sein. Um allein diese Bedingungen zu vereinbaren, ist es nötig, beim Einfügen neuer Details nicht nur dort Vertices zu setzen wo diese benötig werden, sondern auch neue Kanten zu erzeugen und eventuelle alte zu entfernen oder neu zusammenzusetzen. Die anfallende Komplexität ist mindestens so hoch, wie z. B. die des Meshing bei Radiosity, bleibt aber den Anwendern überlassen.

Modellierung, ist weiter nicht nur auf die Geometrie der Oberfläche beschränkt. Es müssen andere *Features*, wie Parameter des Shading-Modells auf der Oberfläche definiert werden. Parameter des Shading-Modells sind z. B. die aus OpenGL bekannten Emissive-, Ambient-, Specular- und Diffuse-Farben - andere Attribute können z. B. Eigenschaften für physikalische Simulation (ZS00) wie die Dehnbarkeit des Stoffes in „Geris Game“ (Duf98) oder semantische Attribute (PP03) sein. Wie die Definition der Features auf die Oberfläche technisch realisiert ist, ist verschieden. So können Features in Texturen zusammen mit einer Abbildungsvorschrift von der Oberfläche in die Textur, dem sog. *Mapping* gespeichert werden. Alternativen dazu sind die Speicherung pro Eckpunkt (*per-vertex*), oder als vom Objekt losgelöste Werte regelmäßig oder adaptiv, z. B. hierarchisch (DGPR02) im Raum um das Objekt verteilt.

Am häufigsten ist jedoch z. Z., vor allem bei Echtzeitanwendungen die Nutzung von Texturen und Mappings. Beide sind manuell, und, was entscheidend ist, getrennt von der Geometrie zu erstellen (fast immer in einem anderen Programm), und vom Nutzer zu managen, was schwierig und relativ abstrakt ist. Zwar ist der Prozess des Mapping theoretisch automatisierbar, geübte Nutzer finden jedoch in nahezu allen Fällen durch ihre Vorstellung von einem Endergebnis, bessere manuelle Lösungen. Ein zentrales Problem bei der Bemalung ist weiterhin, dass wenn ein Mapping gefunden wurde und dazu, sei es 2-D oder 3-D eine Bemalung hergestellt wurde, das Modell streng genommen nicht mehr verändert werden kann (One-Way-System). Zwar sind globalere Modifikationen, die eher linear sind, wie z. B. durch FFDs teilweise möglich, andere jedoch nicht mehr ohne Probleme (Entfernen und Hinzufügen von Flächen, Wechsel der Topologie).

Mesh-Painting verbirgt diese Probelem indem es Nutzer den Eindruck gibt, direkt auf der Oberfläche zu malen. Es ist nicht Ziel dieser Arbeit die prinzipielle Handhabbarkeit von Mesh-Painting zu untersuchen. Der erfolgreiche Einsatz dieser Technik ist bekannt (Pix; Rig; Ali; Aut; Sen). Vielmehr soll folgenden Fragen nachzugehen sein

- Ist Mesh-Painting auf der GPU möglich?
- Welche Nachteile oder Vorteile ergeben sich?
- Insbesondere: Ist es schneller? Und: kann man daher größere bzw. detailliertere Meshes bearbeiten?

- Kann die Darstellung verbessert werden (Shader, Schatten, GI, HDRI)?
- Ergeben sich Einschränkungen für die Interaktion?
- Wie ist Mesh-Painting in einer Production-Pipeline einzuschätzen?

Das Nutzungs-Szenario kann wie folgt charakterisiert werden: es geht darum organische Körper mit vielen Details und vielen Features in Echtzeit mit hoher Darstellungsqualität zu bearbeiten. Ein denkbares Szenario ist das bearbeiten von Körpern, insbesondere Gesichtern. Dabei ist es wichtig, das schnelle und störungsfreie Zeichnen zu gewährleisten. Solche Nutzungsszenarien sind von einer extrem hohen Anzahl von Nutzerhandlungen bestimmt, ähnlich einer detailliert ausgearbeiteten Zeichnung, für die kein Bleistift mit 100 Funktionen nötig ist, sondern ein gut funktionierender Bleistift. Zur Abgrenzung ist zu erwähnen, dass z. B. konstruktive Objekte wie Maschinen oder Gebäude für die Bearbeitung durch Mesh-Painting ungeeignet sind.

Diese Arbeit ist wie folgt gegliedert: nach dieser Einleitung werden im zweiten Kapitel die verschiedenen Grundlagen des vorgeschlagenen Systems genannt, im dritten Kapitel wird die Implementierung beschrieben, deren Ergebnisse im Vierten Teil dargestellt werden. Die Arbeit schließt mit Überlegungen zu möglichen Verbesserungen oder Erweuterungen und einem Fazit.

Kapitel 2

Grundlagen

Dieses Kapitel beschreibt die theoretischen Grundlagen, geht auf dei verwendete Literatur ein, und betrachtet die relevanten jüngeren Entwicklungen der Grafik-Hardware.

2.1 Mesh-Painting

Der Begriff „Mesh-Painting“, beschreibt eine Werkzeug-Metapher, bei der die Veränderung eines Meshes durch Bemalung erreicht wird, und geht auf Häberli und Hanrahan (HH90) zurück. Dieser beschreibt das Bemalen eines virtuellen Objekts mit diffuser Farbe pro Vertex.

Grundsätzlich kann ein Ansatz entweder einen zweidimensionalen, oder einen dreidimensionalen Cursor verwenden. Hanrahan et al. nutzen einen zweidimensionalen oder projektiven Cursor bzw. Pinsel: das bedeutet u. A., dass alle Pixel die im Bild unter der Fläche des Pinsels liegen, bemalt werden. Ein dreidimensionaler Pinsel dahingegen, bemalt alle Punkte, die in der „Nähe“ des Pinselmittelpunkts liegen. Die „Nähe“ kann dabei die wirkliche Nähe im Raum sein, oder die Nähe in einer Parametrisierung, oder in der Oberfläche[1]. Dreidimensionale Pinsel verändern ihre Größe, mehr oder weniger stark, je nach Brennweite, da sie sich ja eigentlich durch den Raum bewegen, wenn sie über das Bild bewegt werden. Zweidimensionale Pinsel behalten ihre Größe an jeder Stelle des Bildes.

Keine der drei Möglichkeiten ist per se „besser“, es hängt von den Erwartungen der Nutzer ab. Vermutlich sind Anwender die geübte Nutzer eines Modellierungsprogramms sind, mit dreidimensionalen Cursorn eher vertraut, wohingegen ein Gele-

[1] Nähe in der Parametrisierung und Nähe auf der Oberfläche sind nicht notwendigerweise identisch, nicht einmal linear zueinander usw. Mehr dazu in Abs. 2.4.

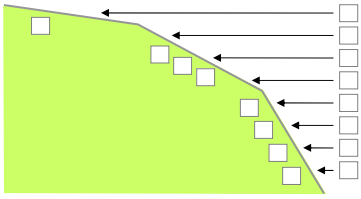


Abbildung 2.1: Eine tesselierte, gekrümmte Oberfläche (grün), wird regelmäßig abgetastet (Kästchen), was zu einer ungleichmäßigen Verteilung (4 : 3 : 1) der Samples auf der Oberfläche führt

genheitsnutzer die Nähe zum zweidimensionalen günstiger empfinden könnte.

Ein Problem dreidimensionaler projektiver Pinsel, tritt auf, wenn auf Flächen unter sehr flachen Winkeln gemalt wird (also z. B. immer am Rand von Bildern von Kugeln, siehe Abb. 2.1): die zu einem Schritt auf dem Bildschirm im Raum gehörende Strecke wird überproportional groß. Bei einfachen Modellen für Pinselstriche führt das zu diskontinuierlicher Qualität der Pinselstriche auf der Oberfläche. Projektive Systeme haben diesen Nachteil nicht, und in anderen Systemen kann es durch bessere Modelle für den Pinselstrich vermieden werden. Dieses Problem tritt vor allem dann auf, wenn ein diskontinuierliches Mapping vorliegt, und dessen Kanten mit einem flach erscheinenden Face, das bemalt wird, zusammenfallen.

Nach Häberli und Hanrahan folgen weitere Arbeiten in diesem Feld. Agrawala et al. (ABL95) beschreiben bereits 1995 erstmals ein System zum Zeichnen mit haptischem Feedback auf Meshes. Die verwendeten Meshes stammen aus einem 3D-Scanner und sind daher so hoch aufgelöst, dass Feature pro Vertex gespeichert werden. HapticFlow (DHQ04) ist ein System zur Modellierung mit haptischem Feedback. Dazu wird die Oberfläche exakt physikalisch simuliert. Gregory et al. (GEL00) beschreiben das System „InTouch“, das ein Mesh mit einem haptischen Eingabegerät (Phantom) bemalen und verformen kann. Typische Auflösungen diese Systems sind einige Zehntausend Flächen. Es werden Subdivision-Surfaces und Texturen mit Mapping verwendet. „ArtNova“ (FOL02), ist eine Weiterentwicklung von InTouch, die es ermöglicht, Detail-Texturen auf ein Modell zu malen. Anstelle von Farben, werden Zugehörigkeiten zu Texturen gemalt.

Igarashi und Cosgrove (IC01) schlagen ein System „Chameleon“ vor, das die Bemalung von Meshes durch ungeübte Nutzer unterstützt. Dazu werden zweidimensionale Pinsel verwendet und ein Mapping (eine Textur pro Face) automatisch, während dem Malen erzeugt. Anwender malen zweidimensional auf dem Objekt, bis sich die Kamera ändert, und die zweidimensionale Bemalung optimiert in eine Textur eingetragen wird. Das System erprobt einige neue Tools, so dass Zeichnen „hinter“ anderen Flächen sowie das Zeichnen im „Laser Mode“, bei dem Vorder- und Rückseite eines Meshes gleichzeitig bemalt werden.

Das System von Lawrence und Funkhouser (LF03) lässt Nutzer Teile von Oberflächen markieren, auf die Kräfte eine vom Nutzer wählbare Zeit wirken. Eine mögliche Kraft ist z. B. jene, die konstant in Richtung der Normale wirkt und dadurch zu „Wachstum“ führt. Während der Zeit, in der die Kräfte wirken, wird die Tesselierung des Meshes überprüft, und wenn Kanten entstehen, die länger als ein bestimmtes Threshold sind, werden diese gesplittet. Die entstehenden Flächen sind dadurch nie undersamplet.

Carr beschreibt (Car04) ein umfangreiches System, das einen hierarchischen Atlas managt. In (CH04b) beschreibt er weiter, wie ein solcher Atlas sich sogar Veränderungen des Modells anzupassen in der Lage ist. Diese System scheint das technisch am weitesten fortgeschrittene in diesem Gebiet zu sein. Es führt eine Vielzahl von Erscheinungen, wie SSS oder Mesh-Painting auf einen Atlas zurück.

Andere Systeme (GHQ04) bieten haptisches editieren von Point-Clouds, indem sie in die Punkt-Menge ein Feder-Masse-System einbetten, das einen FFD-Block steuert und durch das die Punkt-Komplexität und die Modellierungs-Komplexität entkoppelt werden.

Es existieren eine Anzahl kommerzieller Produkte, die Mesh-Painting unterstützen. Deep-Paint 3D (Rig) ist eine Applikation zum Erstellen von Mappings und dem Bemalen von Maps. Das Programm Z-Brush (Pix) ermöglicht es u. A. Objekte zu bemalen, aber auch Geometrie durch bemalen zu verändern. Z-Brush ist das kommerziell am weitesten entwickelte Produkt seiner Art. Das Produkt FreeForm (Sen) der Firma Sensable, die u. A. haptische Geräte produziert bietet, die haptische Modellierung virtuellen Tons. Das System arbeitet mit einer volumetrischen Darstellung des Tons, die es schwierig macht, feine Details anzuwenden (GEL00), vermutlich liegt das Auflösungsvermögen in der Größenordnung eines regulären 3D-Grids. Über die Auflösung von Features und Flächen liegen keine Angaben vor.

Bottleneck all dieser Verfahren ist, besonders bei großflächigen, schnellen Änderungen, die Übertragung an die GPU und die Neu-Erzeugung der Geometrie. Hierarchische Struktur von Subdivision-Surfaces stellen eine effiziente Möglichkeit das, Details nur dort zu speichern, wo diese benötigt werden, doch werden irreguläre Meshes dadurch weiter kompliziert, und schwerer zu verarbeiten und es werden letzten Endes Vertices und Faces in genau der gleichen Größenordnung verarbeitet, wie bei expliziter Speicherung in der feinsten Tesselierung. Einzig der Aufwand der Verarbeitung ist größer und irregulär nicht auf er GPU möglich. Irreguläre Meshes müssen weiterhin irgendwie Parametrisiert werden, wenn sie mehr als Per-Vertex-Attribute verwenden wollen, auch das ist kompliziert, aber möglich (CH04a). Weiterhin sind Normalen zu erzeugen. Normalen benötigen wieder alle Nachbarn, was auf der GPU schwer zu leisten ist. In allen Fällen bleibt aber die Übertragung auf dei GPU.

Eine kleine Rechnung zur Bus-Limitierung, zeigt dies: ein Vertex hat 12 Byte (drei Floats für x, y und z), mit allen Features hat er 12 Byte + 12 Byte Normal + 8 Byte Textur-Koordinate, 3 Byte (Diffuse-Color), 3 Byte Specular Color = 38 Byte, sind also für ein 1024×1024-Faces Mesh 38 MB. Dazu kommen, möglicherweise, je nach Vorgehen, eigentlich noch die Indizes, in der Größenordnung von $1024 \times 1024 \times 2 \times 3 \times 4 = 24$ MB. Demgegenüber stehen AGP $\times$ 8 mit einer theoretische Spitzen-Bandbreite von 2 GB / s (Wik05a), was bei 60 FPS, 30 MB / Frame bedeutet. Würden also CPU und GPU nicht anderes tun, als sich die Daten schicken, oder wären sie in der Lage, Verarbeiten und Versenden parallel zu tätigen (z. B. DMA), wären 60 Hz für ein Eine-Million-Faces-Mesh gerade so möglich. Realistisch ist aber eher eine Geschwindigkeit, die um Vielfache darunter liegt: CPU und GPU können die Daten weder erzeugen, noch zum Zeichnen verwenden, wenn diese übertragen werden, das Zeichnen und Verarbeiten selbst kostete Zeit, usw. Natürlich updaten viele Operationen das Mesh nur Teilweise, im Mittel vielleicht 10 %. Aber die Aufmerksamkeit dieser Arbeit gilt gerade globalen Operationen wie dem Smoothing oder Klonen großer Bereiche. Auch haben Änderungen auf niedrigeren LOD-Stufen globale Veränderungen zur Folge, die alle übertragen werden müssen. Man bedenke weiter, dass gerade das Speicherinterface der GPU zum Videospeicher mittlerweile um Größenordnungen schneller ist als der AGP-Port: NVIDIA gibt dessen Bandbreite hier mit bis zu 38GB / s an (NVi05b). Auch die Einführung von PCI-Express wird hier wenig ändern: dieser macht die Verbindung zwischen GPU und CPU zwar symmetrisch, ist aber mit 4 GB / s immer noch um Faktor 10 langsamer als die Verbindung von GPU und Videospeicher (Wik05b).

2.2 Surfaces

Zur Beschreibung von Körpern gibt es in der Computergrafik verschiedenen Techniken. Einige beschreiben die Oberflächen (engl. „Surfaces“), von Körpern, andere betrachten allgemeine Qualitäten, z. B. Volume-Daten, Point-Clouds oder Variatonal Models. In der vorliegenden Arbeit sind nur Beschreibungen von Flächen relevant, genauer, Flächen die als Polygon-Netze (engl. „polygonal Meshes“) vorliegen.

Polygon-, oder konkreter: Dreiecks-Netze, können als eine Menge von Vertices und einer Menge von Polygonen (engl. „Faces“), dargestellt als Indices in der Menge der Vertices, beschrieben werden. Meist werden eine Anzahl von Randbedingungen gestellt, um die Verarbeitung einfacher zu machen: keine Kante gehört zu mehr als zwei Faces (Rats Nest), keine Fläche ist doppelt vorhanden und Flächen sind konsistent orientiert, d.h. die Reihenfolge der Vertices einer Fläche immer einem gleichen Drehsinn folgt [2].

[2] B-Rep, Half-Edge oder Winged-Edge sind nur andere Formen der Beschreibung, die für bestimmte Aufgaben besser geeignet sind, für die das folgende auch gilt.

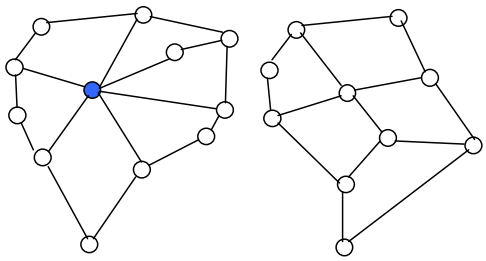


Abbildung 2.2: Ein reguläres und ein irreguläres Mesh

Diese Beschreibung ist in der Lage, Meshes beliebiger Topologie und Geometrie zu beschreiben, hat aber den Nachteil, dass sie Rundungen nur durch immer fein aufgelöstere Polygone beschreiben kann. Diese Limitierung lässt sich umgehen, indem die Vertices nicht als Stützstellen einer stückweise linearen, sondern einer z. B. stückweise kubischen Funktion angesehen werden. Diese Betrachtung führt zu Bezier- oder B-Spline-Patches und NURBS oder im allgemeineren Fall zu Subdivision Surfaces (ZS00). Für Geometry-Images ist diese Unterscheidung eher sekundär. Geometry-Images sind „stückweise interpolierte“ Funktionen, mit regulärer Aufteilung. Wie interpoliert wird, kann verschieden sein, und ist es im implementierten System auch. Die ist dort von Vorteil, da je nach Qualitäts-Anforderungen bestimmte Eigenschaften mit stückweiser Konstanz, andere mit bikubischer Interpolation verarbeitet werden können. Auch Interpolation durch Subdivision ist möglich und wird im System verwendet (Siehe Abs. 3.6.3).

Meshes dieser Art sind entweder irregulär, semi-regulär, oder regulär. Ein Mesh ist dann regulär, wenn die Vertex-Valenz (die Anzahl der eingehenden Kanten in einen Vertex, für den blauen Vertex in Abb. 2.2 z. B. 4, für alle anderen 3) für jedes seiner Vertices genau 4 ist. Ein Mesh ist semi-regulär, wenn die Valenz nicht 4 ist, aber ein Maximum hat, und irregulär, wenn beliebige Valenzen vorkommen.

Abb 2.2 zeigt rechts ein reguläres und links ein irreguläres Mesh: das linke enthält eine Vertex mit der Valenz 5. Es ist festzuhalten, dass nicht-reguläre Meshes vor allem zusammen mit Parametrisierungen und Texturen, ein großes Maß an Indirektion enthalten, was ihre Verarbeitung schwierig und damit langsam macht. So referenzieren Indizes erst Vertices und Texturkoordinaten und diese dann Pixel in einer Textur.

2.3 Parametrisierung

Parametrisierung meint i. A. die Beschreibung eines meist komplexen Sachverhaltes mit meist wenigen Parametern. Die Parametrisierung einer Oberfläche ist eine

Möglichkeit, diese mit nur zwei Parametern und einer Abbildung zu beschreiben. Diese Abbildungen ist ein gut untersuchtes und wichtiges Problem der Computergrafik. Eine Solche Abbildung nennt man „Parametrisierung“ der Fläche. Diese Arbeit betrachtet nur *endliche* Körper, mit endlicher Oberfläche die auf das Flächenstück $[0\ldots1)^2$ *bijektiv* abgebildet werden. Eine solche Parametrisierung wird hier „Atlas“ genannt. Sie wir beschrieben als

$$\mathcal{P}(x) = x \rightarrow s \in [0\ldots1)^2, x \in \mathbb{R}^3$$

Oder ihre Umkehrfunktion

$$\overline{\mathcal{P}}(s) = s \rightarrow x \in \mathbb{R}^3, s \in [0\ldots1)^2$$

Eine Parametrisierung für die $\overline{\mathcal{P}}$ nicht überall definiert ist, heißt „multi-chart“ oder „stückweise“ Parametrisierung. Die Stücke nennt man „Charts“.

Zahlreiche Algorithmen, sind auf eine solche Parametrisierung angewiesen, und es existiert eine Vielzahl unterschiedlicher Qualitäts-Kriterien, die beschreiben, wann eine Parametrisierung gut ist. Ein mögliches Kriterien ist der „Erhalt“ von Eigenschaften bei der Abbildung. So kann eine Parametrisierung z. B. versuchen, den Flächeninhalt, die Innenwinkel oder die Kantenlängen von Flächen möglichst gleich zu halten. Ein anderes Kriterium bewertet danach, wie gut Fläche von Multi-Charts ausgenutzt ist, denn ungenutze Flächen stellen ungenutzten Texture-Speicher. Auch ist eine geringe Anzahl von Kanten oder Bereichen von Multichart-Parametrisierungen günstig. Auch machen Diskontinuitäten mehr Texture-Vertices nötig, da sie den Vertex-Re-Use einschränken. [3] Das Problem der Raumausnutzung ist NP-vollständig (LPRM02), aber es existiert eine Vielzahl von heuristischen Verfahren, die es, da es auch in vielen anderen Bereichen auftritt, als gute Näherung zu lösen vermögen. Eine weitere Anforderung ist die Resistenz gegen MIP-Mapping, bzw. die Resistenz des MIP-Mappings gegen die Parametrisierung. So können nicht einfach beim Erzeugen von MIP-Stufen oder beim Smoothing allgemein zu jedem Pixel vier Nachbar-Pixel durch einen Box-Filter gemischt werden: es dürfen nur solche Nachbarn mit eingehen, die auch auf der Oberfläche Nachbarn sind. An den Grenzen der Charts, sind Nachbarn in der Parametrisierung keine Nachbarn im Raum bzw. auf der Fläche mehr, und müssen gesondert verarbeitet werden (ST04).

Es lässt sich zu jedem Mesh ein Atlas angeben, der zwar den Stretch beliebig steigen lässt, aber den Raum gut ausnutzt: wenn das Mesh o. B. d. A. $2n^2$ Dreiecke hat, kann man immer n^2 Paare finden die zusammen linear gemapt nahezu Quadrate im Mapping bilden, und dann n Zeilen mit n Spalten solcher Quads anlegen. Ein solcher Atlas ist genau dann gut, wenn Dreiecke möglichst gleichschenklig und gleichgroß sind. Wenn man jedes dieser Dreiecke als Geometry-Image ansieht, das man dann bereits genau so gut Displacement-Map nennen könnte, ergibt sich ein

[3] Ein in diesem Sinn besonders schlechtes Mapping, hätte dann dreimal so viel Texture-Vertices wie Faces, anstelle von ungefähr so viel Texture-Vertices wie Vertices.

ebenfalls attraktives Vorgehen, bei dem viele Eigenschaften der Geometry-Images und der dazu beschriebenen Verarbeitungsschritte erhalten bleiben. Diese einfache Vorgehen könnte auch in Echtzeit durchführbar und sein und auf Manipulationen reagieren (CH04a).

Es ist weiter denkbar, die Eigenschaften des Oberflächen-Signals, in die Parametrisierung einfließen zu lassen. So kann man eine diskrete Abtastung des diffusen Farbsignals über die Oberfläche, was eine Textur ja letztendlich ist, als Signal betrachten, und dort wo dieses höhere Frequenzen enthält, dadurch mehr Samples erzeugen, dass die Parametrisierung anders gewählt wird. Das kann auch erreicht werden, indem der Nutzer eine Gewichtung angibt (SWB98).

Neben der Parametrisierung existiert noch die Alternative, Daten nicht an die Oberfläche gebunden, sondern frei im Raum abzulegen., z. B. in Octrees (DGPR02), oder als Point-Clouds. Kniss et al. (KLS$^+$05) beschrieben unlängst ein System, das in der Lage ist, einen solchen Octree auf der GPU zu halten und Modelle zwischen 50 000 und 1 Million Faces zu bemalen. Während die Darstellung zwischen 15 und 80 FPS variiert, wird die Bearbeitungs-Geschwindigkeit mit „highly interactive“ beschrieben, also vermutlich zwischen 3 und 10 FPS.

2.4 Topologie

Topologie wird hier in *drei* Zusammenhängen verwendet: Topologie eines Meshes, Topologie eines Körpers, Topologie einer Parametrisierung. Es geht dabei nie um die streng mathematische Bedeutung, es sind nur einige Begriffe lose entliehen. Die exakte mathematische Betrachtung unterscheidet zwischen Körpern und Manigfaltigkeiten, Orientierbarkiet, Entwickelbarkeit, Offen- und Geschlossenheit, Endlich und Unendlichkeit, usw. (Weib).

2.4.1 Topologie eines Meshes

Die Topologie eines Meshes meint den Zusammenhang der Vertices. Dies hat nichts mit Raum zu tun, und ist das gleiche wie bei Knoten in anderen Graphen.

2.4.2 Topologie einer Fläche

Die Topologie einer Fläche ist ist dem mathematischen Begriff am nächsten: auf eine Fläche mit wie viel Henkeln lässt sich die Fläche durch verformen bringen ohne sie zu schneiden. Die Anzahl der Henkel, der *Genus* g ist dann z.B. $g = 0$

für eine Disc, $g = 1$ für einen Torus oder Teapot usw. Die praktische Umsetzung beschränkt g z. B. auf 1.

2.4.3 Topologie der Parametrisierung

Die Topologie der Parametrisierung beschriebt das Verhalten der Parametrisierung an ihren Rändern: wo, wenn überhaupt, liegen die Punkte jenseits von $[0\ldots1)^2$? Sie wird auch als „Tiling“ bezeichnet. Tiling ist also eine Funktion, die einen Punkt außerhalb von $[0\ldots1)^2$ auf einen Punkt innerhalb abbildet. Auf welche, hängt damit zusammen, wo die Oberflächen-Punkte liegen, die auf der Oberfläche neben den Oberflächen-Punkten liegen, die in der Parametrisierung am Rand liegen.

Sei $\mathcal{P}$ die Parametrisierung des Meshes. So ist das Tiling $\mathcal{T}$ eines Punktes s' ausserhalb der Parametrisierung auf einen Punkt s innerhalb ist definiert als:

$$\mathcal{T}(s') = s' \rightarrow s \in [0\ldots1)^2, \overline{\mathcal{P}}(s') = \overline{\mathcal{P}}(s), s' \in \mathbb{R}^2$$

Eine einfache Instanz dieses Prinzips ist das Tiling auf einer Kugel durch die Perdiodizität von Sinus und Cosinus.

Neben $\mathcal{T}$ ist es noch sinnvoll, eine Relation $\overline{\mathcal{T}}$ zu definieren, die einen Ort s *innerhalb* der Parametrisierung, auf die Menge aller Orte $\{s'_{0\ldots n}\}$ *außerhalb* der Parametrisierung abbildet, die durch durch $\mathcal{T}$ auf s abgebildet werden.

$$\overline{\mathcal{T}}(s) = s \rightarrow \{s' \in \mathbb{R}^2, \mathcal{T}(s') = s\}$$

$\overline{\mathcal{T}}$ soll hier „inverses Tiling“ genannt werden.

2.5 Geometry-Images

Ein Geometry Image ist ein Bild, dessen Farbwerte die Oberfläche eines Meshes beschreiben (GGH02). Pixel des Bildes kann man sich dazu als Vertices vorstellen, die bilineare (oder höhere) Interpolation dazwischen als die Flächen.

Ein Geometry-Image beschreibt ein Mesh bereits vollständig – es muss keine Parametrisierung angegeben werden, es existieren keine Textur-Koordinaten. Auch ohne die explizite Parametrisierung *besitzt* ein Geometry-Image eine solche: sie wird durch die Positionen selbst und deren Zusammenhang gebildet.

Auch ist es in einem Geometry-Image wesentlich einfacher als bei allgemeinen Meshes, Nachbarn zu finden. Jeder Vertex hat immer eine feste 4er bzw. 8er-Nachbarschaft. Verarbeitungsschritte auf Meshes werden damit so einfach (und schnell) wie auf zweidimensionalen Bildern. Eine der nützlichsten Anwendungen

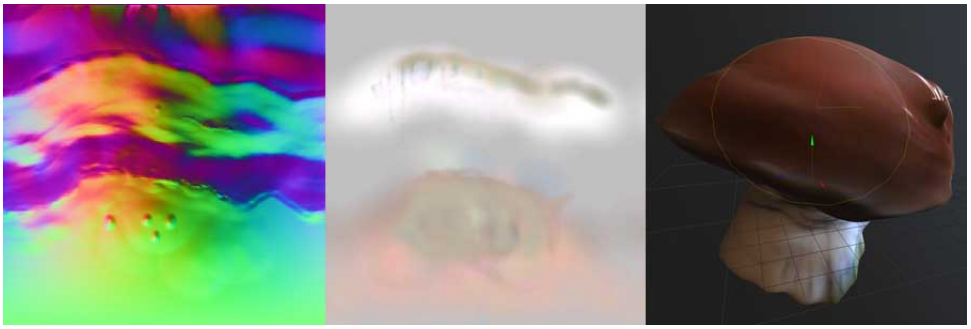


Abbildung 2.3: Positionen als XYZ→RGB und diffuse Farbe eines Geometry-Image-Beispiels

dieser Eigenschaft sind Kompression und LOD: Kompression wird der zweidimensionalen Bildkompression sehr ähnlich, oder gleich (DCT oder Wavelets), LOD entspricht dem MIP-Mapping des Geometry-Images. Neben einem Geometry-Image, werden zusätzliche Surface-Features wie Farben oder Normalen in einem anderen Bild (möglicherweise mit einer anderem Pixel-Auflösung) gehalten, das die gleiche Parametrisierung wie das Geometry-Image hat. Da ein Geometry Image keine unbenutzten Pixel übrig lässt, werden auch die Texturen für übrige Features perfekt ausgenutzt. Im Folgenden meint *das Geometry Image* wenn nicht anders vermerkt, *das Tupel* der Geometrie zusammen mit allen Features wie Farben.

$$g(s) = s \rightarrow \begin{pmatrix} x \\ y \\ z \\ r_{diffuse} \\ g_{diffuse} \\ b_{diffuse} \\ r_{specular} \\ \vdots \end{pmatrix} \in \mathbb{R}^n, s \in [0 \ldots 1)^2$$

Auch besitzt ein Geometry-Image keine Face-Indizes, diese sind ebenfalls implizit. Jedoch existiert in OpenGL zurzeit keine Methode, die diese Technik unterstützt. [4]

Es ist schwierig ein beliebiges gegebenes Meshes direkt auf eine Fläche abzubilden. Dies ist nur für Meshes möglich, die topologisch äquivalent zu einer Kreisscheibe sind. Für ein allgemeines Mesh wird dies bei Gu et al. dadurch erreicht,

[4] Man entfernt sich sogar weiter von dieser Sichtweise, da eines der wesentlichen Probleme bei der Einführung eines direkten Render-To-Vertex-Array immer noch der Umstand ist, das Vertices als Stream vorliegen, und Texturen als 2D-Array die in Hardware vollkommen anders organisiert sind (Ope05c). Aus dem Perspektive der Geometry-Images ist diese Punkt nicht mehr wichtig: zeichnen wird zu einem regulären Besuchen von Vertex-Nachbarschaften, egal in welcher Reihenfolge oder auch gleichzeitig und die Speicherorganisation für Texel scheint genau dann gut, wenn sie auch für Vertices gut ist.

dass ein Mesh an bestimmten Kanten aufgeschnitten wird, so dass ein Kreisscheiben-Äquivalent entsteht. Für ein Objekt des Genus g müssen mindestens $2g$ dieser sog. *Loops* gefunden werden, entlang derer geschnitten wird. Anschließend werden weitere Schnitte durchgeführt, so lange bis durch neue Schnitte keine Verbesserung mehr herbeigeführt wird. Diese zusätzlichen Schnitte, werden immer zwischen dem Punkt mit der größten Distortion und dem diesem Punkt nächsten Punkt auf dem Rand der Scheibe durchgeführt.

Diese Techniken werden im Weiteren in dieser Arbeit nicht weiter betrachtet: es wird wie in (PH03) von einem einfachen Objekt ausgegangen, das korrekt aufgeschnitten und parametrisiert ist: ein Rechteck, eine Kugel (Sphere) oder eine sog. Ge-oSphere, das Ergebnis der fortgesetzten Anwendung der Loop-Subdivision auf ein Oktahedron. Geometry-Images haben oft mehr Vertices, als ein äquivalentes irreguläres Mesh, vor allem wenn diese Optimiert wurde. Der Anteil, den jeder Vertex zur SNR des Gesamt-Mesh beiträgt ist geringer, als bei irregulären Meshes. Geometry-Images sind also eine Methode Kompaktheit der Darstellung gegen Einfachheit bei der Verarbeitung einzutauschen: mehr Vertices, diese dafür aber regulär. Diesen Handel einzugehen, ist durch die technische Entwicklung angezeigt: Vertex-Processing ist z. Z. weit weniger oft das Limit, als das zu große Komplexität andere Strukturen deren Verarbeitung unmöglich macht. Normale Index-Listen können von GPUs genauso genommen *gar nicht* [5] verarbeitet werden. Sie können zum zeichnen verwendet werden, aber alle Nachbarn zu einem Vertex zu wissen ist praktisch unmöglich, obwohl genug Algorithmen das nötig machen[6]. Man kann sich dem für semi-reguläre Fälle nähern, indem man z. B. die Vertex-Valenz nach oben, vielleicht auf 6 oder 8 beschränkt, aber das hat bereits die gleichen Nachteile wie regulär, nur weniger Vorteile.

Das größte Problem bei der Arbeit mit Geometry-Images ist die Behandlung der Ränder der Parametrisierung, besonders, wenn diese in der Oberfläche keine Ränder sind. Vertices die auf Rändern der Parametrisierung liegen sind Unstetigkeits-Stellen, sind „doppelt“, d.h. von Links und von Rechts kommend verschieden. Um ein in dieser Weise eigentlich aufgerissenes Mesh geschlossen zu halten, muss zusätzlicher Aufwand betrieben werden. Als Beispiel sei hier Kompression angeführt: würde das Geometry-Image einfach mit einer 8-Block DCT komprimiert werden, ohne auf die Ränder zu achten, würden am Schnitt im komprimierten Mesh Lücken entstehen, da der rechte und der linke Rand zu verschiedene Blöcken gehört. Also muss darauf geachtet werden, dass auch im komprimierten Mesh identische Rand-Vertices identisch bleiben. Dieses Problem tritt beim Malen auf Geometry-Images häufig auf und war eins der Haupt-Probleme bei der Implementierung des vorliegenden Programms.

[5] Keine Integer-Arithmetik. Indirektionen sind Texturen aus Texturkoordinaten mit Rundungsregeln. Es bleibt aber in jedemFall bei $1:1$ oder $1:n$ für sehr beschränkte n.

[6] Allein: Das Bilden von Vertex-Normalen als Mittel ihrer Nachbarn nachdem sich ein Vertex verändert hat.

Weitere Nachteile von Geometry-Images treten vor allem bei Objekte mit hohem Genus (z. B. „Buddha“ mit Genus 104) auf, hier würden Abbildungen auf mehrere Charts zu bessern Ergebnissen kommen. Durch die Beschränkung auf einen festen, kleinen Genus tritt das Problem im beschriebenen System aber nicht auf. Eine weitere Erweiterung von Geometry-Images, sind Geometry-Videos (BSM+03), die Verformungen von Objekten als Sequenz von Geometry-Images ausdrücken. Losasso beschreibt (LHSW03) eine Implementierung, die ein Geometry Image als das Base-Mesh einer Subdivision-Surface ansieht, und diese auf der GPU abrundet. Er nutzt dabei die reguläre Struktur um die Split- und Average-Schritte mit fixen Masken effizient in Hardware durchzuführen. Dabei ist es möglich, Detailing-Koeffizienten als Skalare in einem lokalen Frame in einer Textur zu speichern und auf der GPU anzuwenden. Die sog. Multi-Chart Geometry-Images (SWG+03) die nur stückweise regulär, dafür aber weniger distorted sind, scheinen für die Anwendung der GPU nicht geeignet, da sie ein globales „Zip-Locking“ zwischen den Vertices die am Rand der Charts liegen nötig machen. Interessant wird vor allem in Zukunft der Vergleich zu Point-Clouds, oder Splats sein, die in gewisser Weise den Geometry-Images vergleichbar, in vielem anderen aber auch vollkommen unterschiedlich sind.

2.6 OpenGL

Die Umsetzung basiert auf einer Anzahl neuere Hardware-Feature bzw. Erweiterungen (Extensions) der Schnittstelle OpenGL. Im Folgenden werden die wichtigsten davon kurz dargestellt.

2.6.1 GLSL

GLSL ist die Shading-Sprache von OpenGL 2.0 (KBR04). Sie Unterstützt Vertex (VP) und Fragment-Programme (FP), in C-ähnlicher Syntax.

GLSL verfügt über einen Preprocessor, der alle gängigen Befehle unterstützt. Dieser kann verwendet werden, um Varianten von Shadern zu erzeugen, die in Details abweichen, aber in weiten Teilen identisch sind, z. B. bei der Behandlung von Sonderfällen. Die Spezifikation sieht allerdings keinen Entry-Point vor, um Parameter an den Preprocessor zu übergegeben. Diese Funktion wird daher durch die GLSLProgramm-Klasse der ClassLib emuliert.

Die Zielhardware unterstützt zwar Texture-Read-Im-VP, diese Feature wird aber, obwohl es der Begriff „Geometry Image“ („Geometry“ entspricht VP, „Image“ entspricht Textur ...) nahe legt, nicht verwendet. Theoretisch ist die Nutzung zwar denkbar, aber durch PBO auch zu erreichen. Es wurden PBOs gewählt da davon

auszugehen ist (Ope05c, Issue 17), dass FBOs in näherer Zukunft eine Erweiterung für echte Render-To-Vertex-Array erfahren werden, und diese Umstellung für die Vertex- und Fragment-Programme transparent wäre.

2.6.2 Floating-Point-Texturen und Floating-Point-Frame-Buffer

Einige der verwendeten Texturen und Buffer arbeiten in Float-Präzision (Ope05a). Es stehen dazu 16- und 32-Bit-Präzision zur Verfügung. Die verwendete Hardware, eine NVIDIA GeFoce 6600 GT, stellt kein Blending und Filtering für 32-Bit Präzision bereit. Filtering, auch höherer Ordnung kann im FP einfach (bis auf die Randbehandlung) emuliert werden. Blending, bei genauerer Betrachtung, jedoch i. A. nicht! (Siehe Abs: 2.7).

Es ist mit MRT nicht möglich, gleichzeitig in Texturen verschiedener Bit-Tiefe zu rendern, so dass, wenn eine Textur 32 Bit verwendet, alle 32 Bit verwenden müssen. Das hat praktisch zur Konsequenz, dass Zeichnen immer auf zwei oder drei „Batches“ (Wlo03) verteilt wird: einer in 8, einer in 16 und einer in 32 Bit, um sich gegenseitig nicht auszubremsen. Dieses Vorgehen ist dann optimal, wenn der Mehraufwand zum doppelten Zeichnen, geringer ist als die Mehrkosten für die doppelte Datenrate bei 32 Bit. Je nach Umständen ist das der Fall oder nicht und ändert sich sicher je nach Hardware. Issue 49 der FBO-Extension (Ope05c, Issue 49), stellt hier aber eine weitere Extension in Aussicht, die diese Limitierung überwindet.

2.6.3 Vertex-Buffer-Object – VBO

VBOs sind eine Technik um Daten auf die GPU zu übertragen (Ope05b). Sie stellen damit eine alternative zu Displaylisten dar. Das Kompilat einer Displayliste ist GPU-Speicher, der mit Vertex-Daten wie Positionen, Farben, Normalen usw. gefüllt ist und State-Wechsel. VBOs gehen den direkten Weg: es werden einfach Vertex-Daten in den GPU-Speicher geschrieben. Dabei sind keine State-Changes möglich, wie zwischein einem `glBegin` und `glEnd`. Konkret arbeiten Buffer-Objects so, dass für eine bestimmte Menge von Funktionen (z. B. `glVertexPointer`, `glNormalPointer`, usw.) ein State gesetzt werden kann, indem diese bestimmte Parameter, die Pointer sind, nicht mehr als Pointer im Client (CPU, das Programm), sondern als Pointer im Server (OpenGL, der GPU) interpretiert werden. Solche Pointer sind durch spezielle Funktionen zu allokieren und freizugeben.

2.6.4 Pixel-Buffer-Object – PBO

PBOs (Ope04b) bauen auf VBOs auf. Sie bieten eine Möglichkeit, Pixel-Daten auf der GPU zu kopieren, vor allem aus dem Read-Buffer (der z. B. der `GL_BACK_BUFFER` ist, also das worin gezeichnet wird) in VBOs. Es wird dadurch möglich z. B. mit einem FP Vertices zu erzeugen, die Kosten dafür sind lediglich die, die Pixel aus dem Read-Buffer in den VBO zu kopieren, was aber auf der GPU wesentlich schneller ist und auch immer schneller wird. Mittelfristig, sollte diese Extension überflüssig werden und es direkt möglich sein, in einen VBO zu rendern. PBOs werden *nicht* verwendet, um in Texturen zu rendern. In jedem Fall bleibt die Verwendung von PBOs immer ein Kopieren.

2.6.5 Frame-Buffer-Object - FBO

FBO (Ope05c) sind eine umfangreiche Erweiterung von OpenGL, die es im Wesentlichen ermöglichen in Texturen zu rendern. Sie stellen dazu alle Funktionen von P-Buffern (Ope01) der Extension WGL_ARB_render_texture (RenderTexture, 2001) und noch wesentlich mehr bereit, ohne die mit P-Buffer verbundenen Probleme (plattformabhängig, jeder P-Buffer ist ein eigener GL-Kontext) zu zeigen.

Die aktuellen NVIDIA-Treiber in der Version 77.77 bieten zwar FBOs an (NVi05a), doch kommt es noch zu diversen Fehlern, z. B. im Zusammenhang mit Multi-Monitor-Systemen. Die Vorteile allerdings überwiegen bereits jetzt. In FBOs wird immer ohne Multisampling gerendert.

2.6.6 Multiple-Render-Targets - MRT

Normalerweise hat ein FP eine genau festgelegte Ausgabe: Eine Farbe, Alpha, und einen Z-Wert. Soll ein FP mehrere Ausgaben haben, muss es normalerweise in mehrere Programme zerlegt werden, die je nach Pass, ein anderen Teil-Ergebnis erzeugen. Die Extension `ARB_draw_buffers` ermöglicht es, die Ausgabe von mehreren Farben in mehrere sog. AUX-Buffer zu steuern (Ope04a).

```
gl_FragData[0] = position;
gl_FragData[1] = normal;
gl_FragData[2] = 3 * position + 2 * normal;
```

Dieses Beispiel hat als Ausgabe gleichzeitig die Position, die Normale und einen Berechnung aus beiden. Die Funktion `glDrawBuffers` steuert, welches Render-Target in welchen Buffer gelangt. Soll z. B. im obigen Beispiel `gl_FragData[0]` auf den Bildschirm, entspricht das dem Buffer `GL_BACK`. Sollen die beiden anderen Werte `gl_FragData[1]` und `gl_FragData[2]` später verarbeitet werden, ent-

spricht dies GL_AUX0 und GL_AUX1:

```
GLenum buffers[] = {
   GL_BACK,
   GL_AUX0,
   GL_AUX1};
glDrawBuffers(3, buffers);
```

Um Werte gezielt aus Buffern auszulesen wird glReadBuffer zusammen mit glReadPixels verwendet.

Die Verwendung von FBOs ist zur Verwendung von MRTs orthogonal: MRTs steuern, in welche Buffer die Ausgaben der FPs geschrieben werden – FBOs können diese Buffer auf Texturen umlenken. Insbesondere bedeutet das: soll in mehrere Texturen gerendert werden, müssen sowohl die Frame-Buffer als auch die MRTs eingestellt werden.

Die Performanz von MRTs müsste kritisch hinterfragt werden, es wurden jedoch keine Messungen über die Performanz eines Quads mit n MRTs vs. n Quads, ohne MRT durchgeführt. Der wirkliche Vorteil von MRTs kommt wahrscheinlich nur in zwei Fällen zum tragen

- Werden viele Faces gezeichnet, sind die Kosten für den Besuch der Pixel groß und werden mit MRT durch n geteilt.
- Es werden FPs ausgeführt, die komplexe Zwischenergebnisse erzeugen, von denen mehr als eine Ausgabe abhängt.

Nur der erste Punkt trifft für das beschrieben System zu: das Zeichnen der sichtbaren Geometrie hat hohe Kosten (Vertex-Count) für den Besuch der Pixel, und mehrere Ausgaben (Farbe, Normale, Textur-Koordinate).

2.7 GPGPU

GPGPU ist die Verwendung der GPU für allgemeine Zwecke, „General Pupose“, also nicht nur für Grafik. Verschiedene Probleme wie Matrix-Verarbeitung, Raytracing, physikalische Simulation, oder Computer Vision lassen sich mit Hilfe der GPU erfolgreich effizienter als mit der CPU lösen (OLG+05).

Um Daten auf der GPU effizient zu verarbeiten, muss das Problem so umformuliert werden, dass es als daten-parallele Transformation eines read-only Eingabe-

Streams durch einen Kernel auf einen write-only Ausgabe-Stream dargestellt werden kann. Der Kernel kann dabei zusätzliche Daten read-only, random-access und eine kleine, feste Anzahl von Register random-acces zum Lesen und Schreiben verwenden. Dieses abstrakte Modell der GPU ist so nicht vollständig, es wäre noch eine Vielzahl von Details zu ergänzen, die bei Bedarf erklärt werden.

In OpenGL realisiert sich das GPGPU-Modell konkret so, dass die Eingabe-Vertices und Texturen und die Ausgabe (vereinfacht) wieder andere Texturen sind. Dabei kann read-only auf weitere Texturen random zugegriffen werden. Alle diese Texturen sind verschieden und überlappen nie. Im einfachsten Fall handelt es sich bei den Eingabe-Vertices um die vier Ecken eines bildschirmfüllenden Quads. Es wird also jeder Ausgabe-Pixel mit jedem Eingabe-Pixel durch ein Fragment-Programm als Kernel transformiert. Der Ablauf eines GPGPU-Schrittes sieht dadurch so aus.

- Binde Eingabe-Texturen 0 bis n als normale GL-Texturen, für $n > 1$ durch Multi-Texturing.
- Binde Ausgabe-Texturen 0 bis m als FBOs und für $m > 1$ unter Beachtung von MRTs
- Aktiviere Fragment-Programm des Kernels unter Abgleich mit MRT und allen Texturen.
- Zeichen Full-Screen-Quad
- Verwende die Ausgabe-Texturen im nächsten Schritt

Die wichtigste Variante dieser Vorgehensweise, ist die, kein Full-Screen-Quad zu zeichnen, sondern andere Geometrie, mit anderen Features, um bestimmte Ausgabe-Pixel mehrmals oder gar nicht zu besuchen. Das sinnvolle Verarbeiten von arbiträrmehrfachen Besuchen ist nur mit Blending möglich, da immer nur eine Ausgabe gespeichert werden kann, oder eine Funktion der beiden, z. B. die Summe oder das Produkt, nicht aber z. B. das Maximum – denn es existiert kein solcher Blend-Mode.

In bestimmten Fällen lassen sich „synthetische Blend-Modes“, im FP erzeugen, indem eine Kopie der Ausgabe als Eingabe gebunden wird und funktional einfließt. Es ist jedoch nicht immer möglich, beliebige synthetische Blend-Modes (wie Maximum) im FP zu erzeugen. Es ist nur genau dann möglich, wenn oben Beschriebenes gerade nicht zutrifft, wenn jeder Pixel nur einmal besucht wird, wenn kein Over-Draw vorliegt, wenn Eingabe und Ausgabe noch identisch sind. Diese Unterscheidungen sind für bestimmte Floating-Point-Modi der Texturen und Frame-Buffer relevant, in denen kein Blending unterstützt wird.

Kapitel 3

Implementierung

Die Implementierung wurde in C++ mit Visual Studio 2003 unter Windows durchgeführt. Verwendete Tools: Maple, Photoshop, Excel und 3ds max. APIS waren OpenGL 2.0, glu, glut und glew.

Die Implementierung teilt das System in verschiedene Klassen

- `Editor` : Interface zum Anwender
- `Mesh` : Das Geometry Image mit seinen n Kanälen
- `MeshChanel` : Ein Kanal des Geometry Image
- `Tool` : Ein Werkzeug, das Kanäle transformiert
- `GPUStream` und `GPUKernel` : Stream und Kernel des GPGPU-Modells
- `Renderer`: Stellen das Geometry Image dar

Weitere Klassen, sind als Teil einer privaten Klassen-Bibliothek „ClassLib“ implementiert, die zum Abschluss kurz beschrieben wird. Der Editor umfasst ca. 5000, ClassLib ungefähr 4000 Zeilen Code.

3.1 Editor

Der Editor ist die zentrale Klasse des Programms. Er setzt die Befehle der Anwender in Änderungen der Geometrie um, er besitzt die einzige Instanz der Klasse `Mesh`, er erstellt das Fenster, lädt Shader, nimmt Eingaben durch Maus, Tasten und Menüs entgegen, zeichnet, misst Performanz, behandelt Fehler und hält zentral

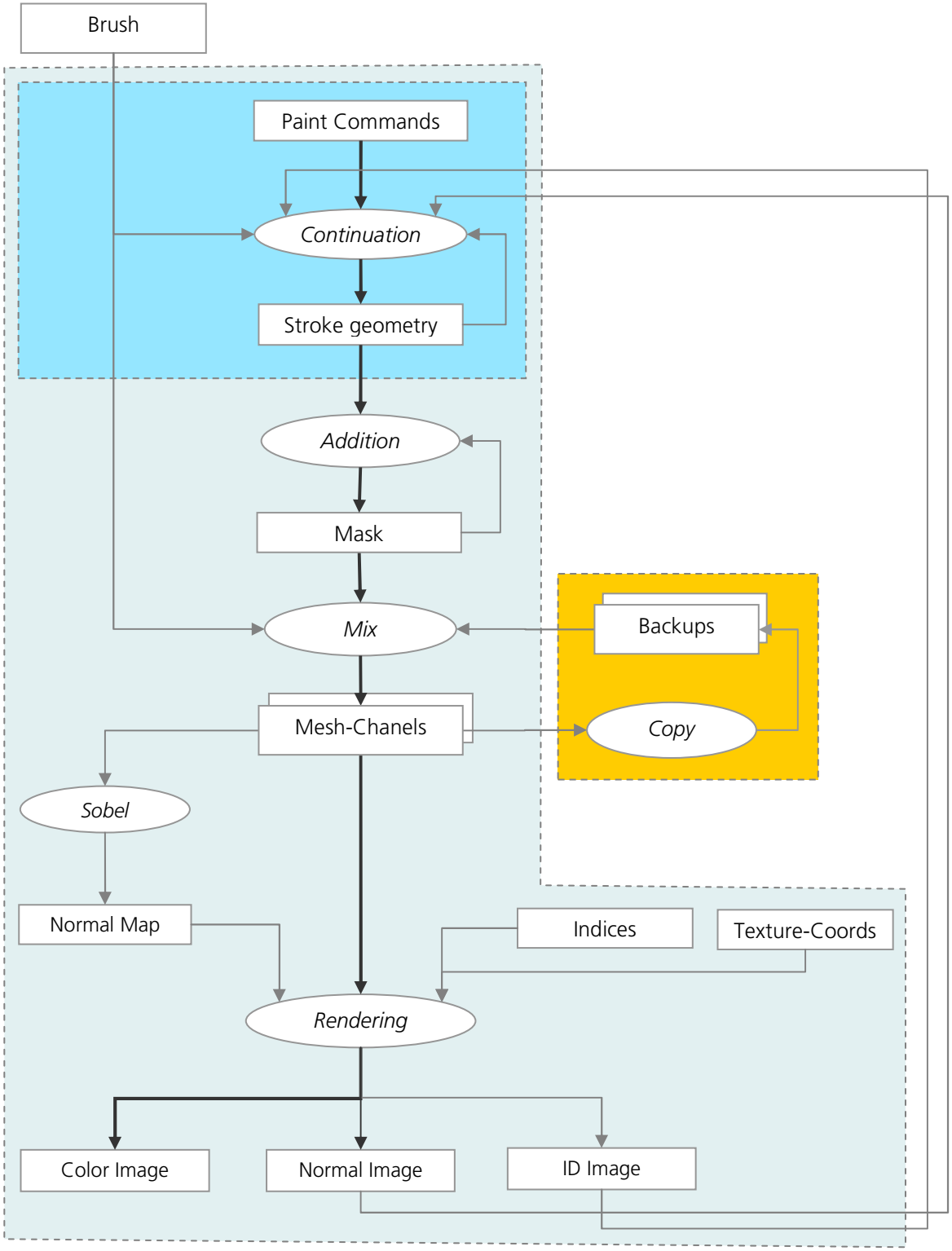


Abbildung 3.1: Datenfluss des Mesh-Editors

den State, z. B. die Darstellung. Editor ist als `Singleton<Editor>` implementiert. Abb. 3.1 zeigt seinen Datenfluss.

Es werden zunächst die Mauspositionen zu einem Zeichenstrich („Stroke“) zusammengesetzt (3.1.1). Der Stroke wird in eine Form auf der Oberfläche des Meshes umgesetzt (3.1.2). Danach wird dem Mesh die Gelegenheit gegeben auf den Stroke zu reagieren, was unter 3.2 beschrieben ist. Nach den Änderungen wird das Mesh dargestellt (3.8). Wurde die Mal-Taste nicht gedrückt, wird nur dargestellt.

3.1.1 Strokes

Die Interaktion ist in „Strokes“ zusammengefasst. Jeder Stroke kapselt die Sequenz der Punkte, über die ein Anwender das Eingabegerät bewegt hat, während er eine Taste gedrückt hält. Ein Stroke besteht aus drei Teilen: Start, Fortsetzung oder Ende. Jedes Frame wird der State von Maus und Tastaur *klassifiziert* und es kommt entweder zu Start und Fortsetzung, nur Fortsetzung oder Fortsetzung und Ende. [1]. Start und Ende finden *an einem Ort* auf dem Mesh statt, Fortsetzung über eine *Strecke* Die Fortsetzung eines Striches sind alle Punkte zwischen Start und Ende. Da die Maus verschieden schnell bewegt werden kann, aber kontinuierlich in der Zeit gesamplet wird, ergibt sich ein ungleich dichtes Sampling im Ort: eine schnell bewegte Maus erzeugt wenig Samples in großem Abstand, eine langsam bewegte Maus erzeugt viele Samples in dichtem Abstand. Für einfache Mal-Programme kann die ignoriert werden, man geht von einer gleichförmigen, relativ langsamen Bewegung des Eingabegeräts aus, erfahrenen Nutzer erwarten hier jedoch ein Verhalten wie Photoshop, bei dem das Sampling im Ort kontinuierlich wird. Das wird erreicht, indem nicht jedes Frame eine neue Position gesamplet wird, sondern Positionen abhängig davon gesamplet werden, wie weit das Eingabegerät bewegt wurde. Ist Δ_{max} der größte erlaubte und $\Delta = |x(t_{n-1}) - x(t_n)|$ der in diesem Frame aufgetretene Abstand zwischen zwei Maus-Positions-Samples $x(t_{n-1})$ und $x(t_n)$, dann kann es zu zwei Fällen kommen: Super-Sampling ($\Delta \leq \Delta_{max}$) und Sub-Sampling ($\Delta < \Delta_{max}$).

Supersampling liegt vor, wenn zu wenige Informationen vorhanden sind. Bei einer Mausbewegung tritt es auf, wenn die Maus zwischen zwei Frames weiter als Δ_{max} bewegt wurde. Es müßen so viele Zwischen-Samples eingeschoben werden, dass deren Abstand ihnen nicht größer als Δ_{max} wird: $m = \lfloor \frac{\Delta}{\Delta_{max}} \rfloor$ Stück. Diese werden an den Stellen $x(t_n) + \frac{k\Delta}{n}, k = 0 \ldots m-1$ erzeugt.

Bei *Sub-Sampling* liegen mehr Samples vor, als benötigt werden. Ein einfache

[1] Start und Ende sind die erste bzw. letzte Position bei der das Mesh unter der Mausposition lag und die Maustaste gedrückt wurde und die Maus das Fenster nicht verlassen hat. Die genaue Definition ist wichtig, da die Maus sich noch-gerdückt aus dem Fenster bewegen kann, da sie erst loslassen könnte, wo kein Mesh mehr liegt, wo der Ort undefiniert wäre, usw.

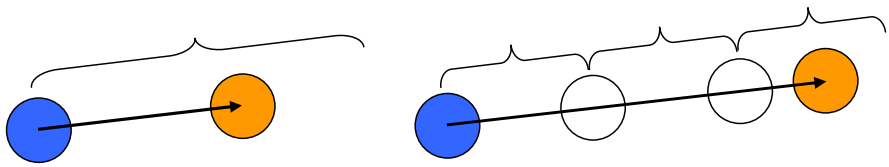


Abbildung 3.2: Sub- und Super-Sampling

Lösung läßt diese Samples einfach aus. Besser wäre es, eine Stroke-Geometrie zu haben, deren letzter Vertex immer der Mausposition entspricht und im Sub-Sampling-Fall nur verändert wird, während der Supersampling-Fall neue Vertices erzeugt.

Bei diesem Vorgehen ist wichtig, dass Δ_{max} zur Pinselgröße proportional gewählt wird, große Pinsel brauchen also weniger Samples. In der Umkehrung brauchen kleine Pinsel viele Samples und sind, wenn sie schnell bewegt werden so schnell, dass das System nicht mehr interaktiv bleibt [2]

In jedem Fall liegt nun eine Sequenz von Bildschirm-Koordinaten vor. Diese werden dann durch Picking (Siehe Abs. 3.1.2) in Koordinaten auf dem Mesh in eine möglicherweise kürzere, denn nicht an allen Bildschirmpositionen liegt auch ein Stück Mesh, Sequenz von Textur-Koordinaten transformiert. Es ist sogar möglich, das diese zweite Sequenz leer bleibt. In diesem Fall wurde definiert das noch kein Stroke begonnen hat.

Nach der Klassifikation, wird bzw. werden für das Mesh, ein bzw. zwei Callbacks aufgerufen, `onStrokeStart(const Vec2f& v)`, `onStrokeContinue(const vector-<Vec2f>& v)` und `onStrokeEnd(const Vec2f& v)`. Die Umsetzung der Strokes ist dem Mesh, dessen Tools und Chanels überlassen (Siehe 3.3).

Neben der Verarbeitung der Strokes, werden auch Callbacks für `onKeyDown(const char key)`, `onKeyUp(const char key)` an das Mesh weitergegeben.

Dieser Teil des Editors sollte perspektivisch verbessert werden. Er funktioniert weder in Gänze noch ist seine Definition optimal. Es existiert genug Literatur (2.1) die andere Vorgehensweisen vorschlägt. Idealerweise sollte dieser Teil austauschbar sein um verschiedene Strategien zu testen. Dazu müsste eine Schnittstelle zum Mesh definiert werden. Eine Sequenz von Koordinaten ist hier wohl nicht allgemein genug. Möglicherweise würden solche Stroke-Startegien als Eingabe alle Maus-Position bekommen und als Ausgabe bereits die Maske haben.

[2] Dabei sind die Kosten für die vielen einzelnen `glReadPixels` bestimmend, da die eigentliche Zeichenoperation von der Sampleanzahl nahezu unabhängig ist. Es macht *einen* Unterschied 200 statt einmal `glReadPixels` aufzurufen. Es macht *keinen* Unterschied 200 statt einem Quad zu zeichnen, wenn ihre Größe und Shader-Komplexität, wie hier, vernachlässigbar ist.

3.1.2 Picking

Um einer Bildschirm-Koordinate p ein Stück Mesh zuordnen zu können, wird „Picking“ benötigt. Zu einem Screen-Pixel wird gesucht:

- Textur-Koordinate
- Raum-Position
- Normale

Das Picking wird durch das Hovering *bei jeder Mausbewegung*, also fast in jedem Frame nötig, ist somit zeitkritisch. Eine gängige Technik bei der Umsetzung besteht darin, einen Strahl r_p (Pick-Ray) aus der Kameraposition und p zu konstruieren und diesen mit der ganzen Szene auf Schnitte zu testen um den nahesten Schnittpunkt auszuwählen. Liegen der CPU alle Informationen über die Geometrie der Szenen vor können diese Schnitte effizient mit Raytracing sublinear durch Raum-Aufteilungen, durchgeführt werden. Da verformende Geometrie aber auch einen Neuaufbau der Beschleunigungs-Strukturen des Raytracing bedeutet, was langsamer ist als der Strahltest (z. B. $O(n\,log(n))$, für BSP).

Für das beschriebene System scheidet Raytracing aus drei Gründen aus.

1. Die Geometrie *verformt* sich → man müsste z. B. einen BSP neu aufbauen.
2. Die Geometrie kann durch das VP *beliebig* verformt werden → es müsste das VP in Software emuliert werden.
3. Die Geometriedaten selber liegen im GPU-Speicher → man müsste für jedes Picking alles neu an die CPU übertragen.

Die vorgestellte Implementierung arbeitet anders: es werden für jeden Pixel alle potentiell relevanten Informationen mit gerendert und zusammen mit dem Pixel gespeichert. Da alle diese Informationen dem FP zur Darstellung sowieso übergeben werden ist dies ohne Aufwand möglich. Um mehr als eine Ausgabe für das FP zu nutzen werden MRTs verwendet. Kommt es zum Picking wird als Read-Buffer der AUX-Buffer gewählt, der die benötigte Information enthält. Aus diesem wird mit `glReadPixels` zurückgelesen. Dabei sind die Wertebereiche geeignet abzubilden, also von z. B. von $[-1\ldots1]$ auf $[0\ldots255]$.

3.2 Meshes

Hier wird beschrieben, wie die einzelnen Meshes erzeugt werden und wie ihre Parametrisierung aussieht. Ein Mesh besteht aus Vertices, Textur-Koordinaten und

Indices. Weiter hat es n Mesh-Chanels, die zusammen das Geometry Image bilden. Momentan ist $n = 3$ und die Kanäle sind Position, Diffuse- und Specular-Color. Das Mesh hält weiter eine Anzahl von VBOs die zur Darstellung verwendet werden. Es wurde eine abstrakte Basisklasse `Mesh` und drei Nachfahren: `Plane`, `Sphere` und `GeoSphere` implementiert.

3.2.1 Indices

Allen Mesh-Typen verwenden die gleichen Indizes. Diese werden nur benötigt, da OpenGL das Geometry-Image nicht nativ verarbeiten kann [3]. Da die Pixel des Geometry Image den Vertices und die Faces der Interpolation der Werte zwischen den Pixeln entsprechen, hat ein Geometry Image der Auflösung $w \times h$ genau $w - 1 \times h - 1$ Faces. Sie werden in einer Schleife über $i = 0 \ldots h-1$ und $j = 0 \ldots w-1$ erzeugt. Es können Quads und Quad-Strips verwendert werden. Für Quads werden die Indices wie folgt erzeugt:

$$quad_{i,j} = \begin{pmatrix} iw + j \\ iw + j + 1 \\ (i+1)w + j + 1 \\ (i+1)w + j \end{pmatrix}$$

Für Quad-Strips:

$$strip_{i,j} = \begin{cases} iw,\ iw+1 & j = 0 \\ iw + j + 1 & 1 < j < w-1 \\ iw + j + w,\ 2^{32} - 1 & j = w-1 \end{cases}$$

Quad-Strips verwenden die `NV_PRIMITIVE_RESTART`-Extension (Ope04c) mit dem Wert $2^{32} - 1$, der signalisiert, dass ein Strip neu beginnt. Entgegen der Erwartung [4] zeigen Strips *keine* bessere Performanz.

Es ist nicht Aufgabe der Indices, für die Geschlossenheit der Meshes zu sorgen, dafür sorgen die Vertices. Obwohl für bestimmte Flächen (Spheres) Quads mit anderen Indizes in Kombination mit Texture-Tiling ebenfalls das Problem zumindest für eine Dimension lösen könnten, ist dieser nahe liegende Ansatz nicht allgemein genug (z. B. nicht auf Geo-Spheres übertragbar) und wird daher nicht weiter verfolgt. Indizes werden so von der Topologie der Parametrisierung unabhängig, was am ehesten der Idee folgt, dass alle Information regulär im Image liegen sollte.

[3] Dabei wäre eine solche `GL_GRID`-Primitve als Extension vergleichsweise einfach: ein neuer State für die Breite und Höhe und ein neuer Primitive-Typ der immer h mal w Elemente wie in $quad_{i,j}$ und $strip_{i,j}$ beschrieben zeichnen. Dadurch liessen sich *alle* Indices einsparen; für 1 Million Faces immerhin 4 MB.

[4] Weniger Speicher und damit Bandbreite - Quads: 4×4 Byte pro Face, Quad-Strip: 4 Byte $\rightarrow$ 75 % Einsparung.

3.2.2 Vertices

Jedem Pixel des Geometry-Images wird beim Start eine Raum-Position zugeordnet. Diese erledigt eine virtuelle Funktion `virtual Vec3f evaluate(const Vec2f& p)`. Diese wird für alle Pixel $g_{i,j}$ mit dessen Koordinaten aufgerufen.

$$g_{i,j} = evaluate(\frac{i}{h}, \frac{j}{w})$$

Bemerkenswert ist hier, dass $\frac{i}{h}$ von 0 bis *inklusiv* 1 läuft. Dies garantiert die Geschlossenheit der Fläche da jetzt die Pixel $g_{i,0}$ und $g_{i,w-1}$ auf den gleichen Oberflächenpunkt abgebildet werden. Gleiches gilt für i und h. Es muss also immer zwischen solchen Texturkoordinaten unterschieden werden, die die Parametrisierung der Fläche betreffen und von 0 bis inklusiv 1 laufen und solchen die physische OpenGL-Koordinaten meinen die von 0 bis exklusiv 1 laufen. Diese Unterscheidung ist die wohl einfachste Form von Übersetzung logischer und physischer Koordinaten wie sie Lefohn (LKS+05) beschreibt. Eine weitere Formalisierung in dieser Richtung wäre auch für diese Arbeit anzustreben, z.B dadurch, dass Meshes GLSL und C++ virtuelle Funktionen [5] bereitstellen welche die Übersetzung leisten. Die verschiedenen Implementierungen für die verschiedenen Mesh-Typen folgen.

Die Funktion `evaluate` für Planes

$$evaluate_{plane}(s,t) = \begin{pmatrix} s - w/2 \\ d \\ t - h/2 \end{pmatrix}$$

und für Spheres

$$evaluate_{sphere}(s,t) = \begin{pmatrix} \cos(2\pi s)\,\sin(\pi t)\,r \\ \sin(\pi t)\,r \\ \sin(2\pi s)\,\sin(\pi t)\,r \end{pmatrix}$$

sind einfach polare und planare Koordinaten.. Für eine Geo-Sphere ist es allerdings nicht möglich eine geschlossene Formel anzugeben die s und t in den Raum abbildet. Hier wird `evaluate` durch einen Algorithmus beschrieben:

- Finde die zweidimensionale Fläche F, auf die s,t in der Parametrisierung des Basis-Oktahedrons abgebildet wird. Es gibt acht Flächen, $F_{i=0...7}$, alle sind Dreiecke. Für jedes F_i und seine Kanten $e_{i,j=0...2}$ wird das Skalar-Produkt $d_{i,j} = e_{i,j} \cdot (s,t)$ gebildet. Ist $d_{i,j}$ für alle j positiv, liegt der Punkt in F_i.

[5] Virtuelle GLSL-Funktionen geben ihren Quelltext als String zurück der dann mit dem Preprocessor einkompiliert wird.

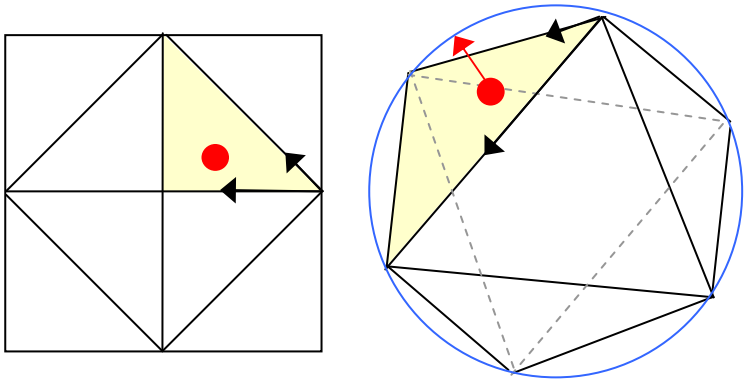


Abbildung 3.3: Geo-Sphere als Entwicklung eines Oktahedrons durch Sub-Division

- Es wurde i gefunden und wird als Index weggelassen. Drücke (s,t) nun als Linearkombination der beiden ersten Kanten e_0 und e_1 von F aus:

  $$(s',t') = \begin{pmatrix} ((s,t)-v_0)\cdot e_0^{norm} \\ ((s,t)-v_0)\cdot e_1^{norm} \end{pmatrix}$$

  Hier meint v_0 den ersten Vertex von F, und e_0^{norm} und e_1^{norm} die normalisierten Kanten e_0 bzw. e_1.

- Verwende diese Koordinaten um die beiden Kanten e_0' und e_1' der dreidimensionalen Fläche F' auf die F abgebildet wird zu kombinieren.

  $$(x,y,z) = v_0' + e_0' s' + e_1' t'$$

  Hier bedeutet v_0' den ersten Vertex der dreidimensionalen Fläche. Diese Linearkombination ergibt einen Punkt auf der Oberfläche des Oktahedrons.

- Dieser Punkt wird einfach durch Normalisierung und Skalierung mit r auf die Oberfläche der Kugel projiziert.

  $$(x',y',z') = \frac{r}{|(x,y,z)|}(x,y,z)$$

Sinn diese Vorgehens ist, es, eine Subdivision-Surface, die ja normalerweise rekursiv und auch auf nicht entwickelbare Flächen definiert ist, für den Sonderfall einer entwickelbaren Fläche nicht-rekursiv sondern geschlossen anzugeben. Dadurch wird eine Geo-Sphere, wie alle anderen Meshes, orthogonal durch `evaluate` von s und t beschrieben. Ob dies eine (und wenn „ja“: *welche*) Parametrisierung der bei Praun (PH03) beschriebenen Alternative darstellt, wurde nicht deutlich. Die Parametrisierung ist entscheidend besser als die der Sphere, da das Verhältnis von Flächen auf der Oberfläche zu Flächen in der Parametrisierung klein und nahe 1

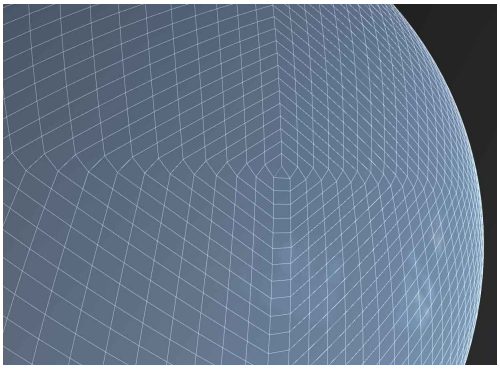


Abbildung 3.4: Anisotropie der Tesselierung

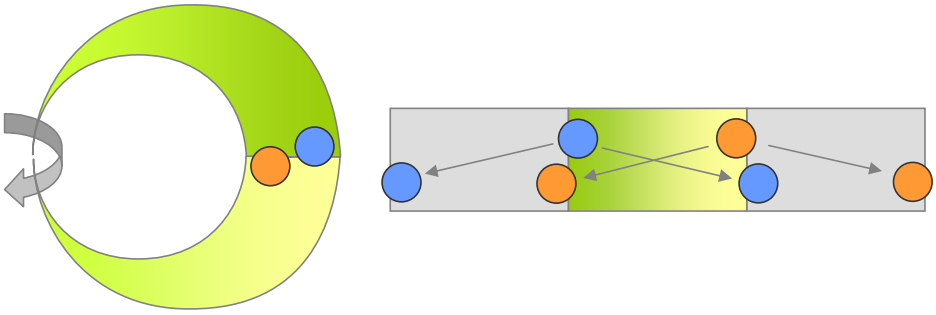


Abbildung 3.5: Tiling des roten und blauen Kreises auf dem Möbius-Band (links) und in der Parametrisierung (rechts). *Ein* Kreis auf dem Band, wird auf *drei* Kreise in der Parametrisierung abgebildet (Pfeile).

ist (bei einer Sphere ist dieses Verhältnis unbeschränkt groß). Ein Nachteil der Parametrisierung ist die Anisotropie der Tesselierung an den Stellen, der Oberfläche, die in der Parametrisierung an den Rändern liegen. Dies wird vor allem an den Stellen der Oberfläche deutlich an denen die Oberfläche in der Parametrisierung verschieden anisotrop ist, Kanten also an einer Stelle nach rechts und nach links verbogen einlaufen.

Es wäre möglich, die Dreiecke die das Oktahedron bilden auch als dreieckige Bezier-Patches mit geeigneten Tangenten aufzufassen. Laut Praun (PH03) folgt daraus aber eine ungleichmäßigere Tesselierung. Als Erweiterung sollte es hier möglich sein, verschiedene Schema für die Aufteilung zu testen.

3.2.3 Tiling

Ein Mesh kann die Topologie seiner Parametrisierung (Siehe Abschnit 2.4) dadurch ausdrücken das es $\mathcal{T}$ und $\overline{\mathcal{T}}$ als Methoden implementiert.

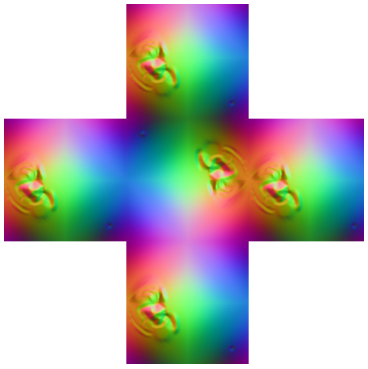


Abbildung 3.6: Tiling auf einer Geo-Sphere. Die Pixel ausserhalb sind um 180 Grad gedrehte Kopien der Pixel innerhalb.

Ziel dessen ist es, *Flächen* in der Parametrisierung so zeichnen zu können, dass sie deren Topologie folgen. Hierfür kann $\overline{\mathcal{T}}$ offensichtlich nicht die ganze (potentiell unendliche) Ergebnis-Menge zurückgeben. Es reicht aus, aller Werte in $[-1\ldots 2)^2$ zu liefern. Beispiel (Siehe Abb. 3.5): ein Kreis mit Radius r an der Stelle (s,t) auf einem Möbius-Band entspricht in der Parametrisierung der Vereinigung der Teile aus drei Kreisen, mit Radius r an den Stellen (s,t), $(s-1,1-t)$ und $(s+1,1-t)$, die wieder in $[0\ldots 1)^2$ liegen.

$$\begin{aligned}
&\{x \in [0\ldots 1)^2, |x-(s,t)| < r\} \cup \\
&\{x \in [0\ldots 1)^2, |x-(s-1,1-t)| < r\} \cup \\
&\{x \in [0\ldots 1)^2, |x-(s+1,1-t)| < r\}
\end{aligned}$$

Im Folgenden wird beschrieben, wie die unterschiedlichen Tilings der Meshes aussehen. Eine *Plane* hat die Topologie einer Disc, bildet also jeden Punkt auf sich selbst ab. Es findet also eigentlich kein Tiling statt.

Eine *Sphere* hat eine zylindrische Parametrisierung: an der „Datumsgrenze“ wird $mod 1$, also `frac` weiter geschritten, an den Polen gar nicht. Daher wird ein Punkt (s,t) auf $(s-1,t)$, auf (s,t) selbst und auf $(s+1,t)$ abgebildet.

Das Tiling auf einer *Geo-Sphere* ist komplizierter. Man könnte vermuten, es sei, nachdem die Sphere zylindrisch ist toroidal, dem ist aber nicht so. Stattdessen ist es (PH03) immer eine um 180 Grad gedrehte Fortsetzung (Siehe Abb., 3.6). Jeder Punkt wird also auf vier andere abgebildet. Die Drehung um 180 Grad entspricht auch der Spiegelung um beide Achsen (Symmetriegruppe), was eher der gewählten Implementierung entspricht. Man betrachte z. B. das Feld rechts neben dem Geometry Image: Die Abbildung lautet $(s,t) \rightarrow (2-s,1-t)$ - zwei Spiegelungen und eine Verschiebung. Insgesamt wird (s,t) auf fünf Positionen abgebildet:

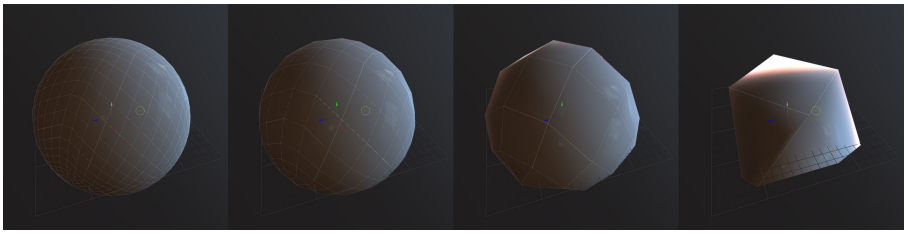


Abbildung 3.7: Sub-Sampling-Stufen 0, 1, 2 und 3 einer Geo-Sphere mit 32×32 Pixeln

$$\begin{aligned}\overline{\mathcal{T}}(s,t) = \{&(s,t),\\ &(2-s,1-t),\\ &(-s,1-t),\\ &(1-s,2-t),\\ &(1-s,-t)\}\end{aligned}$$

Zur Umsetzung: Jedes Mesh implementiert die pur-virtuellen Funktionen `virtual Vec2f tile(const Vec2f& s) = 0` und `virtual void inverseTile(const Vec2f& s, void (*callBack)(const Vec2f& s)) = 0`, die $\mathcal{T}$ und $\overline{\mathcal{T}}$ entsprechen. Die Funktion `tile` liefert einfach den getileten Wert zurück. Die Methode `inverseTile` wird eine Texturkoordinate `p` und eine Zeiger auf eine Call-Back-Funktion `callBack` übergeben, die dann für alle Positionen $\overline{\mathcal{T}}(s)$ aufgerufen wird. [6] So kann z. B. wenn ein Rechteck eines Pinselns im Texture-Space gemalt werden soll der Mittelpunkt des Pinsels und ein Zeiger auf die Zeichenfunktion an `inverseTile` übergeben werden, und der Pinsel wird getiled gezeichnet.

Es ist noch einmal zu motivieren, dass diese Verhalten nicht auf das Tiling in OpenGL zurückgeführt werden kann. Inverses Tiling ist prinzipiell nicht vorgesehen, und auch die spiegelnde Fortsetzung `GL_MIRROR_REPEAT` die GL bietet ist eine andere.

3.2.4 Sub-Sampling

Für viele Objekte sind zur Darstellung der Geometrie durch Faces wesentlich weniger Oberflächen-Informationen nötig als zum Shading. Daher kann das beschrieben System optional nur einen Anteil der Pixel des Geometry-Images verwenden

[6] Dieses Vorgehen ist eine einfache Möglichkeit, Methoden die eigentlich Relationen sind, also n Rückgabewerten haben, zu unterstützen. Die andere Alternative würde einen `vector` zurückgeben, der erst gefüllt werden muss, manchmal aber auch leer ist und über den erst zum Schreiben und dann noch einmal zum Lesen iteriert würde.

um tatsächlich Vertices zu erzeugen: das hier sog. *Sub-Sampling*. Der andere Teil wird nur zur Erzeugung von Normalen verwendet, die ins Shading eingehen. Sub-Sampling Stufe n bedeutet, dass nur für jeden horizontal und vertikal 2^n-ten Pixel im Geometry Image ein Vertex erzeugt wird. Stufe $n = 0$ entspricht ausgeschaltetem Sub-Sampling, da jeder $2^0 = 1$-te, also jeder Vertex verwendet wird. Im Verlauf der Codierung war es möglich, Sub-Sampling orthogonal zu fast allen Teilen des Systems zu machen. Die Ausnahmen sind: Erzeugung der Indizes, und Füllen der VBOs. Die Erzeugung der Indizes, ist dabei aber auch nahezu orthogonal: ein Geometry-Image der Größe m mit Subsampling n hat die gleichen Indizes wie ein Geometry-Image der Auflösung $m2^{-n}$. Für die VBOs wird anstelle die Positionen direkt zu verwenden, LOD-Stufe n verwendet.

3.2.5 VBOs eines Meshes

Sind alle Indizes und Vertices wie beschrieben erzeugt worden, wird das Geometry-Image auf die GPU übertragen und dort nur noch gelesen, bzw. durch kopieren zwischen VBOs und Texturen verändert. Für die Indizes wird ein 32-Bit-Integer (mehr als 2^{16} Vertices) VBO mit `GL_STATIC_DRAW_ARB` erzeugt, der einmal geschrieben, und dann nur noch von der GPU gelesen. Ein zweiter VBO (`GL_STATIC_DRAW_ARB`) wird mit den Koordinaten des Vertex in der Parametrisierung gefüllt; wieder von 0 bis inklusiv 1. Für die Vertices wird ein weiterer VBO angelegt (`GL_DYNAMIC_DRAW_ARB`, bedeutet: Daten ändern sich oft), der aber nicht die Original-Daten hält, sondern in den nur bei jeder Änderung des Positions-`MeshChanels` eine Kopie der darin enthaltenen Werte geladen wird.

3.3 Mesh-Chanels

Positionen sind nur ein Teil des Geometry-Images, auch Specular und Diffuse-Color können auf der Oberfläche gezeichnet werden. Daher wurde eine Abstraktion über Oberflächen-Attribute durchgeführt, die alle in der Klasse `MeshChanel` zusammengefasst werden. Sie bilden zusammen g wie in Abs. 2.5 beschrieben.

3.3.1 Streams

Ein `MeshChanel` besteht aus mehreren `GPUStreams`:

- `mStream`: aktuellen Zustand, als Pyramide
- `mBackupStream`: Zustand vor der Bemalung

- `mTransformedStream`: Zustand nach der kompletten Bemalung

Für den allgemeinen Fall ist dies leider unumgänglich und verbraucht viel Speicher. Es ist nicht möglich GPU-Streams zu managen und etwa zwischen Mesh-Chanels zu sharen: im Worst-Case wird in allen Kanälen gleichzeitig gemalt, werden alle modfiziert, usw.

Jeder Kanal entscheidet, welche LOD-Level er benötigt und pflegt eien Pyramide der nötigen Tiefe. Das LOD-Level wird entschieden, in Abhängigkeit vom Tool und vom Rendere. So kann der Renderer ein niedrigeres LOD-Level der Positionen wählen um diese in Vertices umzuwandeln. Ein Tool was für alle Kanäle niedrigere LOD-Level braucht wenn es angewandt wird, ist das Smooth-Tool.

3.3.2 Normalen

Die für die Darstellung benötigten Normalen, sind von den Positionen funktional abhängig und werden jedes Mal komplett neu berechnet. Dazu wird das aktuelle Geometry-Image als Textur gebunden und ein Fullscreen-Quad mit einem speziellen Shader gezeichnet, der die Normalen berechnet.

Die Normalen werden mit einem Sobel-Filter berechnet, der die beiden partiellen Ableitungen von g nach s und t, also zwei $\mathbb{R}^3$-Richtungs-Vektoren bestimmt. Auf diesen Richtungsvektoren steht die Normale senkrecht, ergibt sich also aus dem Kreuzprodukt. [7]

Für diesen Shader ist die Randbehandlung entscheidend. Es gibt für GPGPU grundsätzlich zwei Alternativen: Frühe oder späte Fallunterscheidung. *Frühe Fallunterscheidung* setzt vor der GPU an und zeichnet für jeden Fall Primitive mit einem speziellen Shader. Die *Späte Fallunterscheidung* verwendet `ifs` im Shader-Code. Beide Formen wurden implementiert und getestet. Die Ergebnisse müssen per Definition gleich sein, ihre Laufzeiten können jedoch unterschiedlich sich.

Späte Fallunterscheidung

Zunächst wurde die späte Randbehandlung implementiert, die weniger invasiv ist. Dazu muss die GPU Bedingte Ausführung unterstützen, was möglicherweise durch Conditional-Moves erreicht wird, d.h. die Kosten bleiben direkt proportional zur Länge des Programms, es werden sozusagen alle Pfade ausgeführt. Vor allem bei

[7] Die Normalen sind in World-Space. Da die Parametrisierung bijektiv ist, wird *kein* Tangent-Space benötigt. Tangent-Space ist nur nötig, wenn ein Stück Normal-Map in mehr als einem Face verwendet wird, da nur dann die Normalen nicht mehr in World-Space gespeichert werden können.

vielen `ifs` ist dieses Vorgehen daher teuer. Die Fallunterscheidung sieht ungefähr so aus:

```
// s in drei Gebiete einteilen
if(s < dS) {
   // Pixel am linken Rand -> kein linker Nachbar
} else if(s > 1 - dS) {
   // Pixel am rechten Rand -> kein rechter Nachbar
} else {
   // Sowohl linker als auch rechter Nachbar verfügbar
}
```

Wo in einem Mesh die Pixel jenseits der Ränder liegen, hängt vom Mesh-Typ ab, jeder Typ hat sein eigenes, implizites Mapping. Um jetzt nicht für jeden Mesh-Typ und jeden Shader komplett eigenen Code schreiben zu müssen, wird der GLSL-Preprocessor verwendet. Da die Fallunterscheidung jedoch in jedem Mesh auch grundsätzlich anders sein soll kann, existiert für jeden Mesh-Typ eine eigenen Fallunterscheidung, jedoch ist der Code zum Falten mit der Sobel-Maske und zur Berechnung der Normale bei allen Varianten gleich.

```
#ifdef MESH_TYPE_SPHERE
   // Fallunterscheidung für Spheres
   // ...
#elif MESH_TYPE_GEO_SPHERE
   // Fallunterscheidung für GeoSpheres
   // ...
#elif
# error No valid mesh type given
#endif
```

Ein Beispiel: für eine Sphere ist der vertikale Wiederholungsfall in beiden Richtungen interessant, da dieser mit einer Anomalie der Parametrisierung zusammenfällt: alle Pixel an den vertikalen Rändern werden zwar auf verschiedenen Vertices abgebildet, aber diese haben alle die gleiche Position, sind also gewisserweise auch identisch. Es wäre zumindest der Normale zuträglich das anzunehmen, deren Funktion wäre sonst am Pol nicht kontinuierlich. Was also tun? Eine Ebene durch alle Vertices im 1-Ring des Pols fitten? 1024 Mal ein 1022-fach überbestimmtes Gleichungssystem lösen? 1024 Mal denselben Mittelwert aus 1024 Face-Normalen bilden? Die Zeile zurück lesen und in Software verarbeiten? Hier wurde keine Lösung gefunden und alle Normalen an den Polen zeigen einfach hardgecodet nach $(0,0,1)$ bzw. $(0,0,-1)$.

Ein anderes Beispiel: horizontale Wiederholung, auch in beiden Richtungen, bei Spheres. Hier kann einfach mod_1 bzw. `frac` beim Zusammensetzen der 3er-Nachbarschaft für die Sobel-Maske gerechnet werden.

Frühe Fallunterscheidung

Die späte Randbehandlung hat keine Conditions im FP, zeichnet dafür aber so viele Quads wie es verschiedene Antworten auf alle Conditions geben kann. Für den Fall der Randbehandlung also drei mögliche Fälle vertikal, drei mögliche Fälle horizontal, mach also neun verschiedenen Kombinationen. Der Shader unterscheidet zunächst auf Preprocessor-Ebene, ob er frühe oder späte Fallunterscheidung verwenden soll. Wird frühe Fallunterscheidung gewählt, entscheidet der Preprocessor welcher Mesh-Typ vorliegt, und dann welcher der neun Fälle. Für späte Fallunterscheidung wird ebenfalls durch den Preprocessor der Mesh-Typ entschieden aber die Entscheidung über die neun Fälle muss pro Pixel getroffen werden.

Alternative Berechnung der Normalen „in-place“

Es bleibt zu erwähnen, dass es eine weitere, grundlegend andere Möglichkeit gibt, die Normalen neu zu berechnen. Es wäre möglich, beim Rendern selbst, die Normale aus dem Umfeld zu bestimmen. Dies ist mit einigen Nachteilen verbunden, könnte aber auch unter sich veränderten Hardware-Merkmalen konkurrenzfähig werden. Es wäre dabei möglich nach Bedarf zwischen „in-place“ und „stored“ zu wechseln: „in-place“ wird verwendet, wenn Normalen nur einmal gebraucht werden (während sich das Mesh ändert), „stored“ würde verwendet werden, wen die Normalen sich nicht mehr ändern. Der Trade-Off verhält sich hier ähnlich wie VP-Texture-Access vs. FBO – in beiden Fällen ist keine Technik grundlegend besser, nur ist die Hardware eben zurzeit so, dass das ein oder andere einfacher zu erreichen oder performanter ist.

3.3.3 Darstellung als Bild

Zur Darstellung von Geometry-Images als Bild wird ein Scale-and-Bias-Shader verwendet, der für verschiedene `MeshChanels` verschiedenen Scale- und Bias-Werte beim Zeichnen auf das Bild anwendet (eine einfache Transfer-Kurve) , um es in einen auf dem Bildschirm darstellbaren Wertebereich zu transformieren. Formen von Tone-Mapping wären eine Automatisierung dieses Vorgangs, doch handelt es sich nicht bei allen Chanels um Helligkeiten, was die Auswahl hier kompliziert.

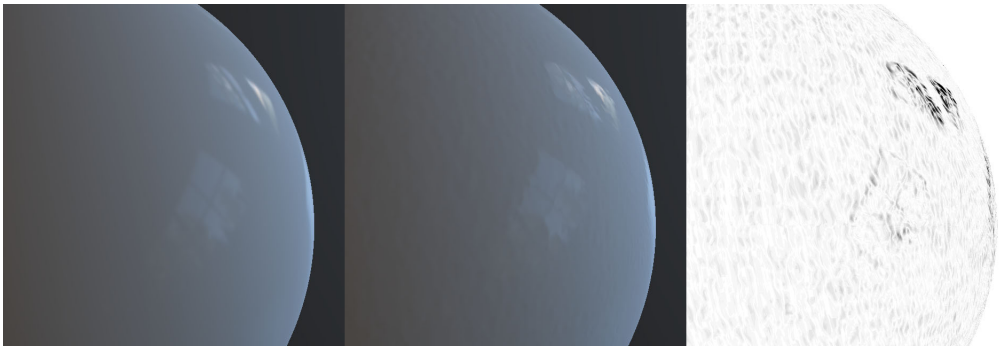


Abbildung 3.8: Ein Stück Kugel-Oberfläche (ohne Perturbation, vollkommen glatt) zeigt das Rauschen der Positionen, und damit der Normalen und des Lightings bei 32 Bit (*links*) und bei 16 Bit (*mitte*) und deren kontrastverstärkte Differenz (*rechts*). Der Unterschied wird bei der Spiegelung heller Stellen der Environment-Map deutlich. .

3.3.4 Un-Do und Re-Do

Weiter implementieren die `MeshChanels` *Redo* und *Undo*. Dazu wird ein `list<Image*>` geführt, an die bei jedem neuen Stroke eine Kopie des `GPUStreams` des aktuellen Zustands angehängt wird. Sind mehr als eine maximale Anzahl (z. B. 32) von Kopien vorhanden, werden ältere Kopien verworfen, bevor neue angehängt werden. Dieses Sichern, dauert vergleichsweise lange (AGP-Port), geschieht aber nur einmal pro Stroke und fällt kaum auf, obwohl es bei genauem Betrachten erkannt werden kann und auch messbare Zeit in Anspruch nimmt (Größenordnung: 30 ms).

3.3.5 Verwendung

Zurzeit werden folgende Einstellungen verwendet:

Mesh-Chanel	Auflösung	Präzision	Pyramide
Position	1-fach	32 Bit RGB	16 Bit RGB
Diffuse	2-fach	8 Bit RGB	8 Bit RGB
Specular	1-fach	8 Bit RGB	8 Bit RGB

Die 32-Bit-Präzision für Positionen wird nur benötigt, da die Normalen diese Präzision benötigen. Die Normalen sind besonders empfindlich gegen Quantisierungs-Rauschen, da bei der Beleuchtung bereits kleinste Änderungen für den spekularen Teil eine nahezu beliebig große Veränderung bewirken. Das wird umso stärker, je unsteter die Beleuchtung ist. Wird hier nur 16 Bit als Auflösung gewählt, ergeben sich ringförmige Artefakte auf der Oberfläche des Meshes. Diese erklären sich

wohl so, da bei 16 Bit, Werte die eigentlich verschieden sein müssten, gleich werden und damit auch die Ableitung über größere Gebiete gleich bleibt. An den Stellen, an denen dann bei 16 Bit doch ein Wechsel statt findet, ist dieser umso größer, umso größer auch die Veränderung des Look-Ups in der spekularen Environment-Map (Siehe Abb. 3.8).

3.4 Mesh-Tools

Ein Mesh-Tool transformiert einen Mesh-Chanel mit einem Stroke. Tools sind als Abbildung des Geometry-Images in ein anderes definiert, im einfachsten Fall unär.

`MeshTool` ist eine Basisklasse, von der alle Formen von Tool ableiten. Dazu werden zwei Funktionen implementiert: eine wird am Anfang des Strokes aufgerufen (`onStart`), eine bei seiner Fortsetzung (`onContinue`). Die Basisklasse bietet für beide Funktionen Standart-Implementierungen bereit.

Die Funktion `onStart` tut folgendes:

```
copy(meshChanel.backupStream, meshChanel.stream);
transform(
   meshChanel.transformedStream,
   meshChanel.stream,
   getKernel());
```

Die Funktion `onContinue` implementiert:

```
fill(mask, Vec3f(0, 0, 0)));
drawStroke(mask, stroke);
mix(
   meshChanel.stream,
   meshChanel.transformedStream,
   meshChanel.backupStream,
   mask);
```

Die abgeleiteten Klassen implementieren diese Funktionen bei Bedarf anders. So tut z. B. das Identity-Tool in beiden Funktionen nichts. Es ist zwar theoretisch möglich, mit jeder Art Tool jeden Kanal zu verändern, aber nicht alle Kombinationen machen Sinn, und werden vom UI zugelassen.

Mesh-Chanel	Mesh-Tools
Position	Identity, Scale, Smooth
Diffuse	Identity, Add, Scale, Smooth, Clone
Speular	Identity, Add, Scale, Smooth, Clone

Es ist nicht sicher, ob alle denkbaren Tools so realisiert werden können. Mittelfristig ist dieses Vorgehen wahrscheinlich aber nicht allgemein genug. Sie lässt z. B. kein „Bewegen“ durch das Eingabegerät mit dessen Bewegungsrichtung zu. Eine Abstraktion würde hier wohl aber noch höher anzusetzen sein.

Im Weiteren, müssen Mesh-Tools kommunizieren können, welches LOD-Level sie benötigen. Normalerweise benötigen Tools keine LOD-Level. Jedes Mesh-Tool hat einen Namen und kann seinen eigenen Cursor zeichnen. Die Standartimplementierung zeichnet dazu einen Kreis mit der Normalen als Orientierung. Im Folgenden werden die verschiedenen Tools beschrieben.

Das `IdentityMeshTool` Verändert einen Mesh-Chanel nicht und zeichnet auch keinen Cursor. So kann z. B. die Diffusfarbe alleine gemalt werden ohne die Position zu verändern, indem für sie Identity und für andere Chanels andere Tools gewählt werden.

Ein `PickingMeshTool` ist ein `MeshTool` mit der zusätzlichen Eigenschaft, dass es eine Farbe picken kann, wenn die Linke Maustaste zusammen mit der ALT-Taste gedrückt wird. In diesem Fall werden alle weiteren Kommandos geblockt und erst ein neuerliches Drücken der Maustaste ohne die ALT-Taste erzeugt wieder Zeichen-Kommandos. Das `PickingMeshTool` ist immer noch abstrakt.

Das `Add` und `ScaleMeshTool` sind `PickingMeshTools` und entsprechen der Addition bzw. Multiplikation mit einer gepickten Konstante.

Das `ExtrudeMeshTool` addiert ein Vielfaches der Normalen. Die Normale kann dabei optional aus einem tiefern LOD-Niveau kommen. Diese Vorgehen ist für Positionen wichtig, die man in Richtung ihrer Normalen verschieben will, unabhängig davon, wo sie sich befinden.

Das `CloneMeshTool` erzeugt eine translierte Kopie. Dazu kann bei gedrückter Strg-Taste ein „Anker“ gesetzt werden. Dieser ist der Ursprung der Kopie und wird angezeigt, wenn sich das Tool zeichnet.

Entgegen der ursprünglichen falschen Überlegung lassen sich dadurch *keine*, Kopien von Features auf der Oberfläche durchführen. Dazu wäre eine Art *lokales Koordinaten-Frame* nötig, wie es in z. B.: bei Sorkine et al. (SLCO$^+$04) verwendet wird. Deren Nutzung könnte auch möglich sein und event. auch von der regulären Struktur profitieren. [8]

[8] Ist es vielleicht nur ein normaler 2D-Laplace-Filter? Solle man neben den LOD-Stufen auch

Das `SmoothMeshTool` ist das komplexeste Tool: es ersetzt einen Mesh-Chanel mit einer niedrigeren LOD-Stufe seiner selbst. Als Rekonstruktions-Filter wird ein Bezier-Spline dritter Ordnung verwendet. Die Implementierung arbeitet für sich gesehen gut, lässt aber Probleme z. B. die anisotrope Tesselierung noch deutlicher sichtbar werden. Auch die Randbehandlung ist schwierig: eine 4er-Nachbarschaft hat noch viel mehr Sonderfälle als eine 2er-Nachbarschaft. Unklar bleibt, wieso Losasso (LHSW03) das abgerundete Mesh durch mehrere Split-and-Average-Schritte wie eine Subdivision-Surface erzeugt, obwohl das Geometry-Image doch gerade regulär ist.

3.5 Sampling

Alle Funktionen die das Geometry-Image samplen, sollten eine Anzahl zentraler Funktionen dazu verwenden [9]. Diese bieten eine Hierarchie von Qualitäten: Point-, bilineares, biquadratisches und bikubisches Sampling. Daneben existiert noch die Möglichkeit eine 3×3-Box zu sub-sampeln. Alle Funktionen rekurieren so weit wie möglich auf andere Funktionen, z. B. auf DeCasteljaus Algorithmus. GLSL macht daraus das was es ist: ein Paar Multiply-Adds und Subdraktione nacheinander.

Bemerkenswert ist, dass diese Funktionen in einer zentralen Datei abgelegt sind, die immer inkludiert wird, nachdem ihre Freiheitsgrade im Code festgelegt wurden: Welcher Mesh-Typ? Welche Randbehandlung? GLSL sieht keine `#include`-Preprozessor-Anweisung vor (KBR04), NVIDIA liefert sie vernünftigerweise trotzdem (NVi04).

```
// Bildet Texturkoordinate aus einer Textur in eine andere ab.
// Die Koordinate der Mitte des ersten Texels in OpenGL ist:
//
//    dS/2, dT/2,
//
// die des Letzten
//
//    1-dS/2, 1-dT/2.
//
```

Detailing-Stufen führen, auf deren Stufe man dann kopieren kann? Laplace und Box ist schon fast die Haar-Basis (Weia), vielleicht sollte überdacht werden, ob die vollständige Darstellung des Geometry Image als Wavelet günstiger sein könte. Wäre es möglich, die Details sparse zu führen? Wäre die Verarbeitung dann immer noch einfach? Wären dann noch feiner Details bei besserer Speichernutzung möglich?

[9] Aus Performanz-Gründen und Problemen mit der Orthogonalität, wird dies nicht konsequent eingehalten

```
float2 remapSamplePosition(float2 x, float2 dxFrom, float2 dxTo) {
   x -= dxFrom / 2;
   x /= 1 - dxFrom;
   x *= 1 - dxTo;
   x += dxTo / 2;
   return x;
}

float4 samplePoint(sampler2D sampler, float2 x, float2 delta, float2 dX) {
   x = x + delta * dX;
   float2 x2 = x;

   // [...] Randbehandlung hier.
   //
   // Da zwischen dem rechten Nachbarn von 0, 0 und dem linken Nachbarn
   // von 0, dS unterschieden wird, sollten später hier auch Multicharts
   // machbar sein, dort ist Nachbarschaft so allgemein.
   //
   return texture2D(sampler, x2);
}

// Berechnet Gewicht der beiden Nachbar-Werte im Abstand dX
float2 calculateWeight(float2 x, float2 dX) {
   return float2(mod(x.x, dX.x) / dX.x, mod(x.y, dX.y) / dX.y);
}

float4 deCasteljau1(float4 v0, float4 v1, float w) {
   return v0 + w * (v1 - v0);
}

float4 deCasteljau2(float4 v0, float4 v1, float4 v2, float w) {
   return deCasteljau1(
      deCasteljau1(v0, v1, w),
      deCasteljau1(v1, v2, w), w);
}

float4 deCasteljau3(float4 v0, float4 v1, float4 v2, float4 v3, float w) {
   return deCasteljau1(
      deCasteljau2(v0, v1, v2, w),
      deCasteljau2(v1, v2, v3, w), w);
}

float4 sampleBox(sampler2D sampler, float2 x, float2 dX) {
   float4 v00 = samplePoint(sampler, x, float2(-1, -1), dX);
```

```
   float4 v01 = samplePoint(sampler, x, float2(0, -1), dX);
   float4 v02 = samplePoint(sampler, x, float2(1, -1), dX);
   float4 v10 = samplePoint(sampler, x, float2(-1, 0), dX);
   float4 v11 = samplePoint(sampler, x, float2(0, 0), dX);
   float4 v12 = samplePoint(sampler, x, float2(1, 0), dX);
   float4 v20 = samplePoint(sampler, x, float2(-1, 1), dX);
   float4 v21 = samplePoint(sampler, x, float2(0, 1), dX);
   float4 v22 = samplePoint(sampler, x, float2(1, 1), dX);
   return (v00 + v01 + v02 + v10 + v11 + v12 + v20 + v21 + v22) / 9;
}

float4 sampleLinear(sampler2D sampler, float2 x, float2 dX) {
   float2 w = calculateWeight(x, dX);
   float4 v00 = samplePoint(sampler, x, float2(0, 0), dX);
   float4 v01 = samplePoint(sampler, x, float2(1, 0), dX);
   float4 v10 = samplePoint(sampler, x, float2(0, 1), dX);
   float4 v11 = samplePoint(sampler, x, float2(1, 1), dX);
   float4 v0 = deCasteljau1(v00, v01, w.s);
   float4 v1 = deCasteljau1(v10, v11, w.s);
   float4 v = deCasteljau1(v0, v1, w.t);
   return v;
}

float4 sampleCubic(sampler2D sampler, float2 x, float2 dX) {
   float2 w = calculateWeight(x, dX)
      * float2(0.333333, 0.333333) + float2(0.333333, 0.333333);
   float4 v00 = samplePoint(sampler, x, float2(-1, -1), dX);
   float4 v01 = samplePoint(sampler, x, float2(0, -1), dX);
   float4 v02 = samplePoint(sampler, x, float2(1, -1), dX);
   float4 v03 = samplePoint(sampler, x, float2(2, -1), dX);
   float4 v10 = samplePoint(sampler, x, float2(-1, 0), dX);
   float4 v11 = samplePoint(sampler, x, float2(0, 0), dX);
   float4 v12 = samplePoint(sampler, x, float2(1, 0), dX);
   float4 v13 = samplePoint(sampler, x, float2(2, 0), dX);
   float4 v20 = samplePoint(sampler, x, float2(-1, 1), dX);
   float4 v21 = samplePoint(sampler, x, float2(0, 1), dX);
   float4 v22 = samplePoint(sampler, x, float2(1, 1), dX);
   float4 v23 = samplePoint(sampler, x, float2(2, 1), dX);
   float4 v30 = samplePoint(sampler, x, float2(-1, 2), dX);
   float4 v31 = samplePoint(sampler, x, float2(0, 2), dX);
   float4 v32 = samplePoint(sampler, x, float2(1, 2), dX);
   float4 v33 = samplePoint(sampler, x, float2(2, 2), dX);
   float4 v0 = deCasteljau3(v00, v01, v02, v03, w.s);
   float4 v1 = deCasteljau3(v10, v11, v12, v13, w.s);
```

```
    float4 v2 = deCasteljau3(v20, v21, v22, v23, w.s);
    float4 v3 = deCasteljau3(v30, v31, v32, v33, w.s);
    float4 v = deCasteljau3(v0, v1, v2, v3, w.t);
    return v;
}
```

3.6 GPU-Abstraktion

GPGPU formalisiert die GPU als eine Maschine die Streams durch Kernel transformiert (Siehe Abs. 2.7). Wie gut dieses Modell auch auf die vorgestellte Applikation passt, ohne etwa Performanz zu zerstören, wurde erst im Lauf der Entwicklung deutlich. Die beschriebene Applikation implementiert dieses Modell in den Klassen `GPUStream`, `GPUKernel` und `GPUStreamPyramid`.

3.6.1 GPU-Stream

Ein `GPUStream` besteht aus einer Textur und einem FBO, an den die Textur attached ist, was zusammen ein Textur ergibt, in die gerendert werden kann. Weiter hat der Stream einen Namen, kann sich aus einem `Image` lesen und darin schreiben und implementiert eine Anzahl statischer Funktionen, um Streams zu tranformieren:

- 0-äre: $\{\} \rightarrow$ `GPUStream`
- unäre: `GPUStream` $\rightarrow$ `GPUStream`
- binäre: `GPUStream` $\times$ `GPUStream` $\rightarrow$ `GPUStream`
- tertiäre: (`GPUStream` $\times$ `GPUStream` $\times$ `GPUStream` $\rightarrow$ `GPUStream`

3.6.2 GPU-Stream-Kernel

Ein `GPUKernel` ist ein Shader mit einem Namen. Spezielle Filter sollten formal davon ableiten und ihre Parameter mit `set/get`-Funktionen zugreifbar machen, dies wurde aber auch für die Shader selbst versäumt zu implementieren. Häufig verwendete Transformationen sind zusammen mit ihren Kerneln als Abkürzungen definiert: mit-Konstanten-Füllen (0-är), Kopieren (unär), gewichtetes Mischen (trinär).

3.6.3 GPU-Stream-Pyramide

Eine Pyramide von GPU-Streams wird in der Klasse `GPUStreamPyramid` beschrieben. Diese ist allgemeiner als OpenGL-MIP-Maps, da sie die Topologie der Parametrisierung befolgt. Eine Pyramide kann dynamisch Level erzeugen und löschen und sich aus ihrer Basis konstruieren. Zur Konstruktion können verschiedenen Filter verwendet werden. Der in OpenGL bei MIP-Maps normalerweise verwendete Box-Filter ist für Geometry-Images problematisch da er den Daten eine Tendenz verleiht. Dies wird deutlich, wenn das Smooth-Mesh-Tool mit einem niedrigeren Level fortgesetzt mischt: die Faces wandern unerwünscht in die Richtung $\overline{\mathcal{P}}(0,0)$, dem Ort auf der Oberfläche der auf den Ursprung parametrisiert wird, anstatt nur abzurunden. Günstiger ist hier ein 3×3-Gauss-Kernel, der diese Tendenz aufhebt. Der Gauss-Filter ist zwar separabel, aber für 3×3 liegt darin kein ausreichender Vorteil.

Alle benötigten Level werden durch sukzessive Filterung rekursiv erzeugt:

```
for(int i = 1; i < maxLevel; i++) {
   transform(mLevel[i], mLevel[i - 1], downsamplingKernel);
}
```

Pyramiden werden momentan für das Sub-Sampling, für das Smooth-Tool und das ISL verwendet.

3.7 Dynamische Parametrisierung

Trotz der am Anfang nach verschiedenen Kriterien „guten“ Parametrisierung, kann sich diese durch Bearbeitungs-Schritte beliebig verschlechtern. Was also nötig wäre, ist eine Art „dynamische Parametrisierung“ die die Positionen auf der Oberfläche des Meshes so verschiebt, dass die Qualität erhalten bleibt. Andere Ansätze haben auf der CPU und bei irregulären Meshes hier natürlich die Möglichkeit neue Faces bzw. Vertices zu erzeugen – dies ist im beschrieben Fall aber gänzlich unmöglich.

Erstens: Die Qualität der Parametrisierung zu messen ist nicht weiter schwierig. Ein Kernel der das Geometry-Image in eine Qualitäts-Map transformiert, indem er misst, wie lang die Kanten im Mittel, oder maximal oder minimal sind. Alternativ kann er auch die Abweichung von der Soll-Länge, auch zum Quadrat, in Mittel, Minima oder Maxima bestimmen.

Zweitens: reguläres Re-Meshing ist, auch auf der GPU, hier kein Problem: einfach Geometry-Image als Textur laden und so viele Quads zeichnen, wie es Texel

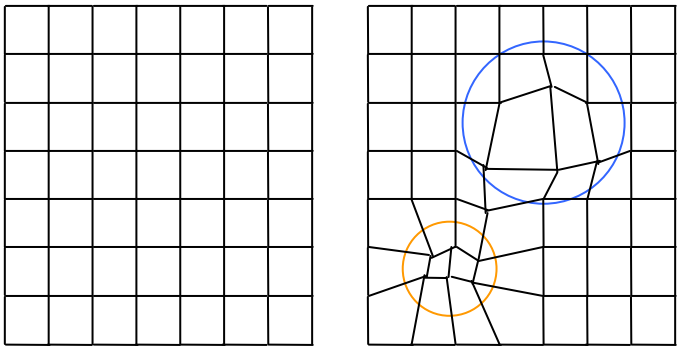


Abbildung 3.9: Warping des Texture-Space beim Resampling des Geometry-Image

bzw. Vertices gibt, aber mit Quads die Texturkoordinaten haben, die von den Koordinaten der Pixel (meist nur minimal) um ein Displacement abweichen (dem zweidimensionalen Image-Warping vergleichbar).

Lässt man dieses Displacement nun in die Richtung in der die Parametrisierung am schlechtesten ist, und mit einer Stärke proportional zu deren Qualität zeigen, würde sich die Parametrisierung verbessern. Einzig die Qualität des Signals leidet unter diesem Vorgang.

Carr verwendet zwar dynamische Re-Parametrisierung für seinen Atlas, diese arbeitet aber auf der CPU und ist dadurch AGP-port-limited, was er in (CH04a), beschreibt. Dort wird auch der Qualitätsverlust beim Re-Sampling beschrieben, der beim Warping auftritt.

3.8 Renderer

Renderer stellen ein Geometry Image auf dem Bildschrim da. Es wurde eine Basisklasse `Renderer` erstellt, die dies abstrakt leistet. Drei Klassen wurden davon abgeleitet: `ImageBasedLightingRendere`, `ImageSpaceLightingRenderer` und `ComicRendere`. Alle Klassen müssen zurzeit noch so rendern, dass Picking möglich ist, also den ID-Buffer usw. in die entsprechenden MRTs. Dies ist keine korrekte Trennung der Belange und sollte verbessert werden.

3.8.1 Image-Based-Renderer

Der Image-Based-Renderer ist der Standart-Renderer. Er bietet gute Darstellungs-Qualität bei guter Performanz. Er verwendet eine spekulare und eine diffuse Environmen-

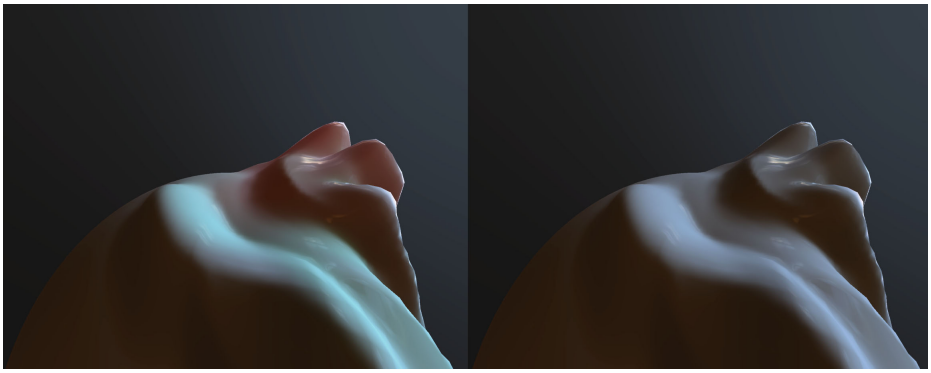


Abbildung 3.10: Vergleich zwischen IBL (*rechts*) und ISL *links*. ISL zeigt weiche Beleuchtung, vor allem im Rotkanal.

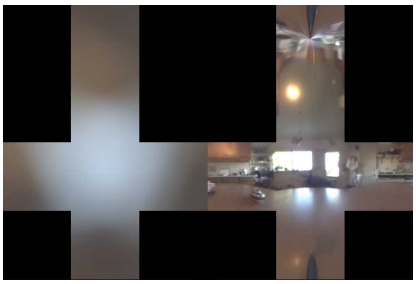


Abbildung 3.11: Rechts die diffus vor-gefilterte und links die spekulare Light-Probe „Kitchen“.

Map zur Beleuchtung. Beide stammen von der Seite Paul Debevecs (Deb). Die diffuse Environment-Map wurde aus der spekularen in HDRShop (TD) hergestellt und liegt im Format 64×64, die spekulare als 256×256 vor. Beide werden zur Laufzeit als kubische Environment-Maps in float-Präzision verwendet und wurden im Cross-Layout gespeichert.

Der Shader zur Darstellung verwendet als Eingabe die beiden Environment-Maps, die Normal-Map und die diffuse und spekulare Textur. Er indiziert die diffuse Environment-Map mit der Normale, und die spekulare mit dem an der Normalen reflektierten Eye-Vector. Die Ergebnisse werden mit der diffuse bzw. spekularen Textur gewichtet. Zusätzlich wird dann der Winkel gemessen, unter dem die Fläche betrachtet wird. Besonders flache Winkel verstärken die Beleuchtung, als einfache Annäherung an einen Fresnel-Term.

```
uniform sampler2D normalmap;
uniform samplerCube diffuseEnvmap;
uniform samplerCube specularEnvmap;
uniform sampler2D diffuseTexture;
```

```
uniform sampler2D specularTexture;
uniform float inverseTextureResolution;
varying vec3 viewerDirection;
varying vec3 position;

void main() {
   float3 normal = texture2D(normalmap, gl_TexCoord[0].st).rgb;
   float3 diffuseColor =
      1.0f * textureCube(diffuseEnvmap, normal) *
      texture2D(diffuseTexture, gl_TexCoord[0].st);

   float3 reflection = reflect(viewerDirection, normal);
   float3 specularColor =
      textureCube(specularEnvmap, reflection) *
      texture2D(specularTexture, gl_TexCoord[0].st);

   float edge =
      max(1, 100 * pow(1 - dot(normal, -viewerDirection), 8));

   float3 color = (edge * 0.004f * specularColor + diffuseColor);

   gl_FragData[0].rgb = color;
   gl_FragData[1].rg = gl_TexCoord[0].st;
   gl_FragData[1].b = 1;
   gl_FragData[2].rgb = position * 0.5 + 0.5;
   gl_FragData[2].a = 255;
   gl_FragData[3].rgb = normal * 0.5 + 0.5;
}
```

3.8.2 Image-Space-Renderer

Der Image-Space-Renderer (ISL-Renderer) zeigt einige Möglichkeiten des ISL am Beispiel von stark vereinfachtem SSS. Er arbeitet weitgehend genauso wie der IBL-Renderer nur mit zusätzlichen Komponenten. Es wird vor dem Rendern das Lighting einmal in eine Light-Map gerendert, zu der eine Pyramide, aufgebaut wird. In einem nächsten Pass, wird diese Lightmap noch einmal in einen weiteren Stream gerendert, diesmal aber, indem Rot, Grün und Blau aus verschiedenen Stufen der Lightmap-Pyramide bikubisch gesamplet werden. Dazu wird noch einmal das Lighting selbst gelesen, und abgezogen. Effekt ist, dass die Lightmap nun eine in verschiedenen Kanälen verschiedne stark geblurte Kopie des Lichtes enthält, abzüglich dem, was das direkte Licht dort bereits ausmacht. Diese direkte Licht

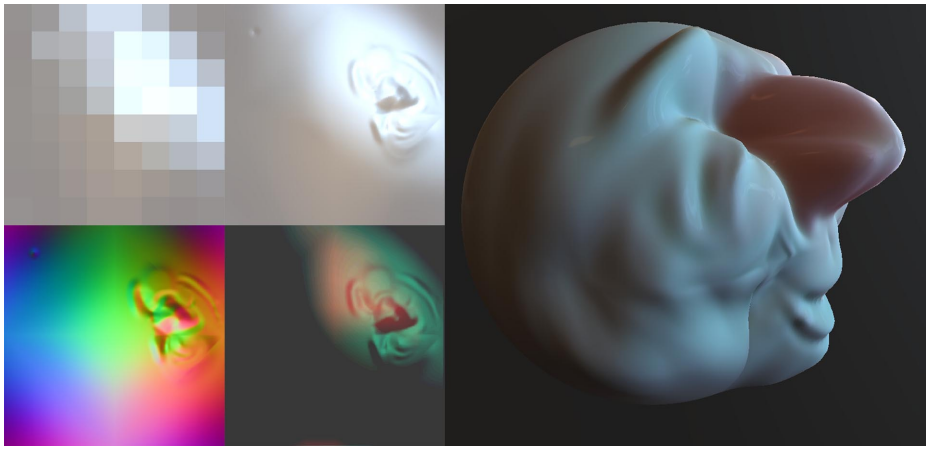


Abbildung 3.12: Kopf mit SSS. Das Licht der helleren, abgewandten Seite erhellt auch die dem Betrachter zugewandte Seite. Oben links zwei Level der Light-Map-Pyramide, darunter rechts das Geometry-Image und links die indirekte Komponente (verstärkt)

wird *genauso* wie bei IBL über Envmaps usw. berechnet und dargestellt, nur das die indirekte Komponente addiert wird.

Das Herrausrechnen der indirekten Komponente ist daher motiviert, dass bei der tatsächlichen Darstellung das Lighting per-pixel gesamplet wird und bei der indirekten Beleuchtung nur (wenn auch bikubisch) interpoliert wird. Ist die Texel[10]/Pixel-Rate nah an 1, ist das kein Unterschied. Kommen aber auf einen Texel viele Pixel ergibt sich ein Unterschied. Durch diese Unterscheidung kann eine Lightmap in viel geringerer Auflösung verarbeitet werden als das Geometry-Image, z. B. im Verhältnis 1 : 16.

Irgenwo in diesem System wird mit einem kleine Wert skaliert, der beschreibt, wieviel transportiert wird. An dieser Stelle müßte die BSSRDF stehen, deren Verhalten für den Look entschiedent ist.

3.8.3 Comic-Renderer

Um die Allgmeienheit des Renderers zu testen wurde ein Comic-Renderer implementiert. Der Comic-Shader verhält sich genau wie der Image-Based-Lighting-Shader, nur das er die Helligkeit als Index in einer Gradienten-Textur verwendet.

```
//
// ...
```

[10] Auch zu lesen: *Lumel*

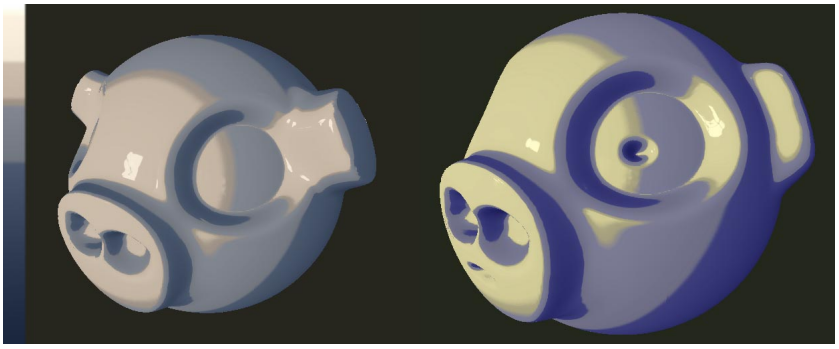


Abbildung 3.13: Zwei einfache Köpfe (< 1 Minute) mit Comic-Shader und ihr Gradient

```
//
float3 color = edge * specularColor + diffuseColor;
float a = clamp(
   color.r * 0.3 +
   color.g * 0.5 +
   color.b * 0.1, 0, 0.99);
float4 comicColor = texture2D(gradientTexture, float2(a, 0));
//
// ...
//
```

Abb. 3.13 zeigt zwei einfache Comic-Köpfe mit einem Gooch-Gradient (GGSC98).

3.9 ClassLib

Neben dem eigentlichen Programm zum editieren von Geometry-Images, wurde Code auch als Teil einer privaten Klassenbibliothek (ClassLib) erstellt um diesen wieder verwenden zu können. Einige der Funktionen decken sich mit den Funktionen z. B. aus OpenSG, das Layer ist jedoch viel dünner. Seine wesentliche Aufgabe ist eine flache Abstraktion von OpenGL und die Bereitstellung von Tools die häufig für Computer-Grafik-Anwendungen nötig ist, wie Vektor- und Matrix-Klassen.

3.9.1 GLContext

Es wird eine abstrakte Klasse für GL-Kontexte zur Verfügung, von der alle Kontexte abgeleitet werden die sichtbar sind, also ein Fenster haben, genauso wie jene,

die unsichtbar sind, wie P-Buffer. Kontexte mit Fenstern können wahlweise kompatibel mit glut erzeugt werden, oder direkt mit Win32. Die direkte Erzeugung mit Win32 ist nötig, wenn glut bestimmte Feature nicht zugänglich macht. So ist es nur mit einem `Win32WindowContext` möglich ein Mouse-Capture anzulegen, oder die Maus-Historie abzufragen. Auch zur Erzeugung der AUX-Buffer und zur Wahl der Mulit-Sampling-Einstellungen ist glut nicht ausreichend.

3.9.2 Bilder

Es existiert eine Klasse `Image` für Bilder. Diese kann Dateien laden, diese konvertieren, oder Testbilder erzeugen (Checker-Board, XOR, linearer und radialer Gradient). Um Bilder zu lesen und zu schreiben wurde die abstrakte Basis-Klasse `ImageIO` implementiert. Alle Nachfahren dieser Klasse sind Singletons und registrieren sich im Konstruktor bei einem weiteren Singleton, dem `ImageIOManager` der Dateiname anhand der Extension dem jeweiligen IO zuordnet. IOs sind prinzipiell in der Lage, Bilder verschiedener Kanal-Zahl und Quantisierung zu Lesen, nur sind nicht alle Pfade implementiert. Unterstütze Typen sind .RAW Float-RGBA und .TGA für Grey 8-, RGB 24- und RGBA-32 Bit. Die Umrechnung von verschiedenen Darstellungen von Environment-Maps (Cross, Longitude-Lattitude, sechs Einzelbilder) leistete die Klasse ebenfalls.

3.9.3 Texturen

Die Texturklasse ist als Mischung aus Template und abstrakter Basisklasse realisiert, das als Template-Parameter das OpenGL-Texture-Target nimmt. Es sind 2D-Texturen und Cube-Maps implementiert.

3.9.4 FBO

Die FBO-Extension wird in zwei Klassen `FrameBuffer` und `RenderBuffer` gekapselt. Frame-Buffer können angelegt und gebunden, so wie die Extension an und ausgeschaltet werden. Da FBOs eine spezielle Fehlerbehandlung benötigen, wurde diese Implementiert. Ein Frame-Buffer stellt weiter Funktionen bereit, um eine Textur oder einen Render-Buffer an einen Attachment-Point zu binden. Die beschriebene Applikation verwendet zu keiner Zeit Render-Buffer, da wenn in eine Textur gerendert wird, kein Stencil- oder Depth- benötigt wird.

3.9.5 GLSL

Es wurden Klassen erstellt, die GLSL-Programme (`GLSLProgram`) und GLSL-Shader (`GLSLShader`) abbilden. Für einen GLSL-Shader kann einen Vertex und ein Fragment-Programm gesetzt werden, das Programm kompiliert und gebunden werden. Es wurde eine Funktion zum Laden von Textfiles in einem bestimmten Schema erstellt, die ein Vertex- und ein Fragment-Programm aus einer Datei lädt, daraus ein GLSLProgramm erzeugt und linkt. Zum Erkennen und Behandeln von Fehlern im Shader-Code wurde eine Exception implementiert, die die einzelnen Fehlermeldungen kapselt.

Der GLSL-Preprocessor wird durch ClassLib um Funktionen erweitert. GLSL sieht zunächst keine Funktion vor, um Parameter an den Preprocessor zu übergeben. Um dieses nahe liegende Verhalten zu emulieren, werden alle gewünschten Parameter und eine Leerzeile einfach beim Laden der Programme vorne an den zu kompilierenden Code-String angehängt. Der Aufruf eines Meshes um seinen mNormalCalculationShader zu laden, sieht daher z. B. so aus:

```
mNormalCalculationShader = GLSLProgramm::loadFromFile(
   "normalCalculation",
   "#define meshTypeName = " + getMeshTypeName());
```

Die Funktion `getMeshTypeName` ist virtuell, und gibt z. B. `GeoSphere` oder `Sphere` zurück, das der Shader-Quelltext beim kompilieren verwenden kann. Das Linken der Shader sollte in keinem Fall, mit und ohne Preprocessor pro Frame stattfinden – alle möglichen Verwendungen werden vorkompiliert gehalten.

3.9.6 Error-Handling

Zur Behandlung von Fehlern wurden Funktionen implementiert, die die OpenGL und WGL-Status-Codes überprüfen und falls Fehler aufgetreten sind, die nötigen Fehlermeldungen erzeugen, und Exceptions geworfen. Insbesondere die Fehler der FBOs und GLSL bedurften dabei einer Besonderen Behandlung. Zusätzlich wurde eine Log-Klasse zum Sammeln von Fehler-, Warnungs- und Diagnose-Meldungen erstellt. Diese werden automatisch mit der Programmzeile und dem Namen der Quelldatei gelabelt und können nach Warnstufe gefiltert werden. Alle Checks sind als Preprocessor-Funktionen definiert, um bei Bedarf, z. B. im Release-Build nicht mehr aufgerufen zu werden.

3.9.7 Konsole

Die Konsole stellt eine Möglichkeit dar, proportionalen oder Text fixer Breite als Overlay-Grafik darzustellen. Dazu wird eine Textur geladen, die alle Buchstaben enthält, sowie eine Datei, in der die Buchstabengrößen gespeichert sind. Eine solche Technik arbeitet exakt und schnell und ist allen anderen Techniken (`TextOut()` kann kein Double-Buffering, `wglGlyph...` kann kein AA und hat geometrisches Aliasing und `glBitmapFont` verwendet `glPutPixels`) in jeder Hinsicht überlegen.

3.9.8 PerformanceTimer

Die Klasse PerformanceTimer wird verwendet um das Zeitverhalten von Codeteilen zur Laufzeit zu untersuchen. Es existiert dazu eine Anzahl anderer, z. B. Offline-Techniken (Profiler), die weder genau sind (nur Clock-Genau), noch irgendwie in der Lage Nutzerhandeln in Beziehung zum Geschwindigkeitsverhalten der Applikation zu setzen. Der Assemblerbefehl `rdtsc` lädt die aktuelle Anzahl an Cycles seit dem Systemstart als 64-Bit-Zahl und stellt somit eine exakte und zuverlässige Methode dar, Code cycle-genau und live zu timen. Die Klasse `PerformanceTimer` kapselt das Messen von Cycles zwischen zwei Punkten im Programmfluss. Solche Timer können hierarchisch verschachtelt werden und so anzeigen, welche Teile der Zeit eines Gesamtablaufs auf Teilabläufe entfallen. Daneben führen die Timer Mittelwerte und filtern die aktuelle Zeit aus einem kurzen History-Fenster vorheriger Messungen.

Kapitel 4

Ergebnisse

Als Ergebnis werden verschiedenen Nutzung-Episoden als Sequenzen gezeigt, die Performanz quantitativ aufgeschlüsselt und ein analytischer Vergleich zu anderen Systemen gezogen.

4.1 Beispiele für Nutzungs-Episoden

Einige lose Sequenzen von Anwendungen sollen die Möglichkeiten des Sytsems zeigen. Am einfachtsen sind Objekte die geometrisch nah an einer Kugel liegen zu modellieren. Diese Beschränkung ist teils durch den Mangel und die Qualität der Tools besitmmt, teils durch die Limitierungen des Ansatz. Alle Beispiele wurden durch den Autor ausgeführt, der mit dem System naheliegnderweise Übung erlangt hat, und auch i. A. einen professionellen Hintergrund in 3D-Modellierung besitzt. Es wurden kein Teste durch andere Nutzer durchgeführt, was mit dem existierende UI unrealistisch wäre.

Die Modelle wirken oft wie Ton oder bemaltes Holz - die Ausführung erscheint lose und skizzenhaft. Dieser Stil ist nicht „gewollt“ - er ergibt sich aus den Defiziten des Systems.

4.1.1 Fisch

Organische Objekte wie Fische (Siehe Abb. 4.1) die topolgisch einfach sind aber organisch differenziert und bunt sind sollten ein günstiger Anwendungsfall des Sytsems sein. Besonders die Bemalung sollte wesentlich einfacher sein als mit Texturen und Mappings. Konkret ist hier das größte Problem, dass das Displace-Tool

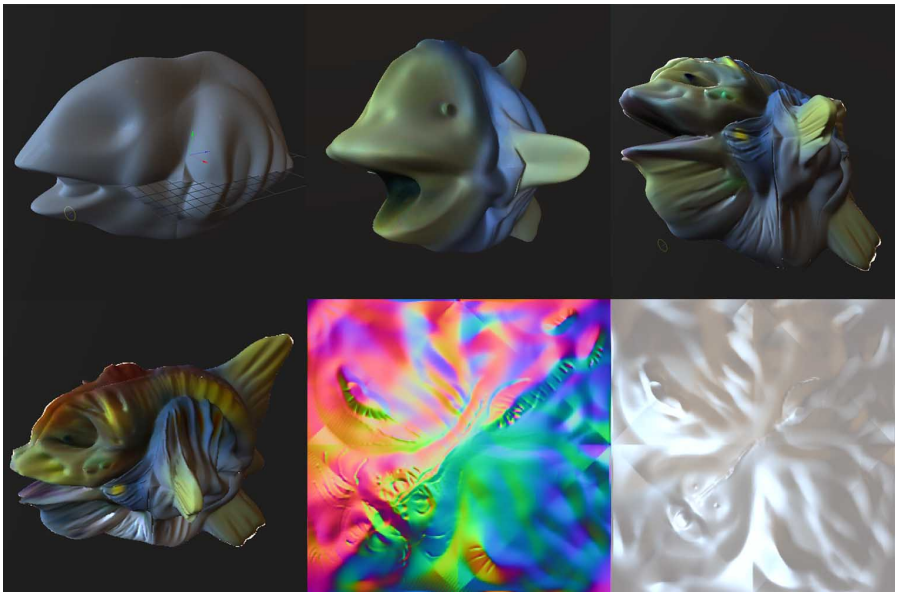


Abbildung 4.1: Ein tropischer Fisch. Vier Arbeitsschritte, Lightmap und Normal-Map

nicht korrekt funktioniert und mit dem Scale-Tool gearbeitet werden muss. Dadurch sind alle Größen nur Displacements, also ändern nicht die Richtung. Für den Kiefer z. B. wäre es wichtig, auch Vertices zuverlässig verschieben zu können. Gut funktionierte hier das Hinzufügen von organischen Details, wie Adern und Warzen, die der Struktur des Gewebes folgen. Es ist trivial z. B. die Struktur der Gräten/Rippen zu artikulieren, was mit normalem Modelling eher Aufwendig ist. Die Bearbeitungszeit betrug 20 Minuten; die Auflösung war 512.

4.1.2 Kürbis

Ein Kürbis (Siehe Abb. 4.2) ist einer Kugel sehr ähnlich und einfach zu modellieren. Es gab keine Probelme, für dieses einfache Modell sind die Tools ausreeichend. Es ist noch einmal zu motivieren, dass alleine die Bemalung in 2D nicht trivial ist: man hat immer Verzerrungen an den Polen *oder* Diskontinuitäten, was noch schwieriger (nahezu unmöglich) intuitiv zu handhaben ist. Die Bearbeitungszeit betrug ebenfalls 20 Minuten und die Auflösung war 512.

4.1.3 Kopf

Ein wichtiger Anwendungsfall ist die Modellierung von Gesichtern (Siehe Abb. 4.3). Dies ist nur Bedingt möglich. Die Modellierungszeit war ca. 1 Stunde und die

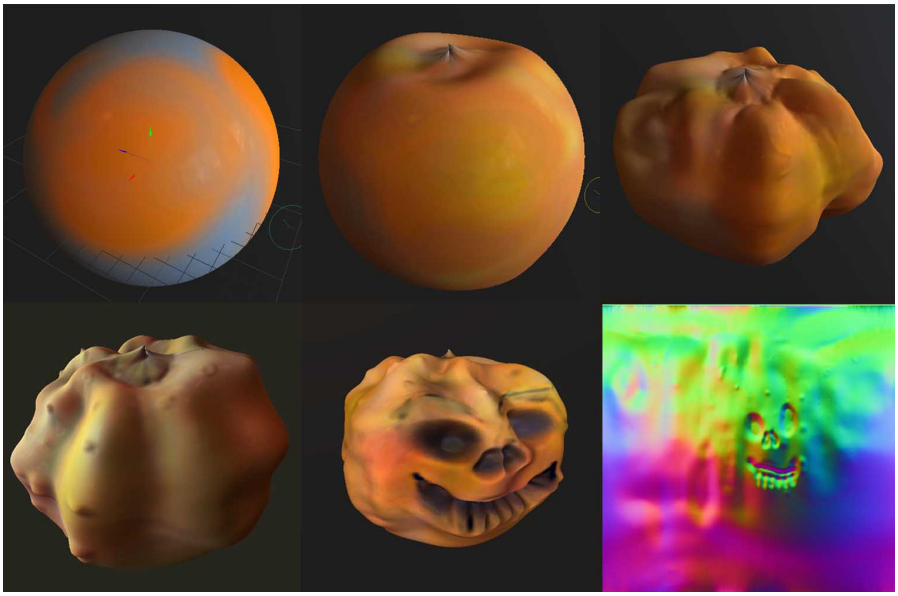


Abbildung 4.2: „Das System ist ausgezeichnet zum Modelliern von Zierkürbissen mit Mündern geeignet.“ - Fünf Arbeitsschritte, unten links mit SSS, und die Normal-Map unten rechts

Auflösung ebenfalls 512 mit 1-fachem Sub-Sampling.

Bei den feinen farbigen Abstufungen menschlichger Haut ist es wichtig, dass Farben exakt gepickt werden. Dies ist mit dem System so zur Zeit nicht möglich. Farben können zwar gepickt werden, aber nicht auf dem Mesh, ohne Lighting. Nutzer können die Farbe der Stirn picken um das Kinn genauso zu färben: es ist immer das Lighting dazwischen. Diese Problem tritt z. B. bei Photoshop auch auf, wenn mit meherer Ebenen gearbeitet werden soll. Einfach die Diffus-Frabe in der Map für einen Ort nachzuschlagen ist die diametrale Alternative. Aber auch das ist problematisch: es werden Farben gepickt, die nicht dem entsprechen was gesehen wird, und wenn diese von der Stirn gepickt am Kinn angewandt werden, sehen sie wieder anders aus. Auch wegen dieser Farb-Problematik konnte das SSS nicht getetstet werden, obwohl es doch u. A. für Haut implementiert wurde.

4.2 Performanz

Das Test-System ist ein Intel Pentium 3,4 GHz, mit 2 GB RAM und einer NVIDIA 6600 GT 8xAGP mit ausgeschaltetem Wait-For-Retrace. Alle Messungen wurden mit der Performance-Timer-Klasse von ClassLib durchgeführt.

Die zentrale Größe für die Qualität der Darstellung ist die Auflösung des Geometry

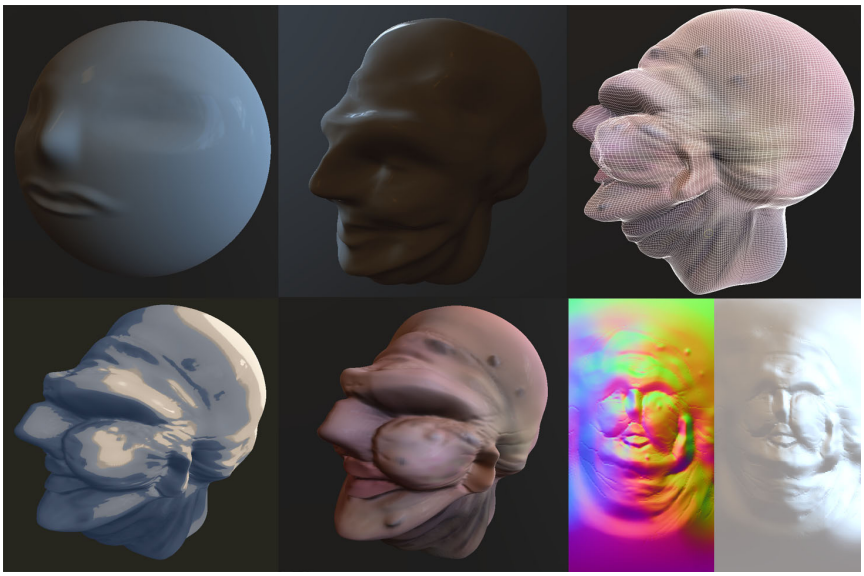


Abbildung 4.3: Kopf eines alten Mannes. Unten Rechts Normal- und Light-Map

Image.

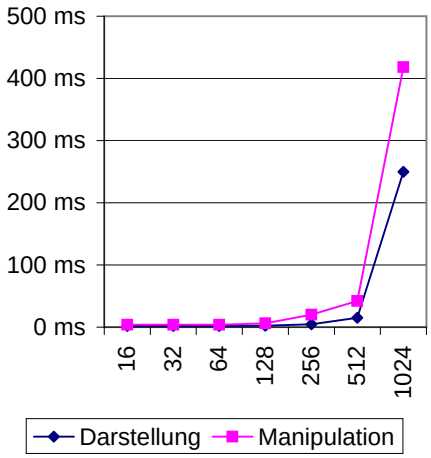
Die Tabelle 4.4 zeigt, dass Flächen mit bis zu 512×512 Elementen in Echtzeit, mit mehr als 20 Hz manipuliert werden können. Für Flächen mit 1024×1024 Elementen bietet das System nurnoch 4 bzw. 2 Hz und ist praktisch nicht mehr bedienbar.

Abb 4.5 zeigen, dass sich Sub-Sampling für die Darstellung lohnt: die Zeit für die Darstellung sinkt, für die Manipulation etwas weniger. Die Manipulation wird deswegen günstiger, weil sie weniger an die Darstellung übetragen muss. Manipulation ohne Darstellung bleibt bei Sub-Sampling konstant.

Die größeren Kosten sind jedoch jene für die Tool-Transformation des Geometry-Images: die Geschwindigkeit sinkt um die Hälfte. Günstig ist jedoch die Entkoppelung der Tool-Komplexität von der Darstellung: Manipulation ist bei allen Tools gleich schnell, nur der Start dauert verschieden lang. In der Praxis sind aber auch diese Kosten gering und nahezu konstant (Siehe Tbl. 4.6).

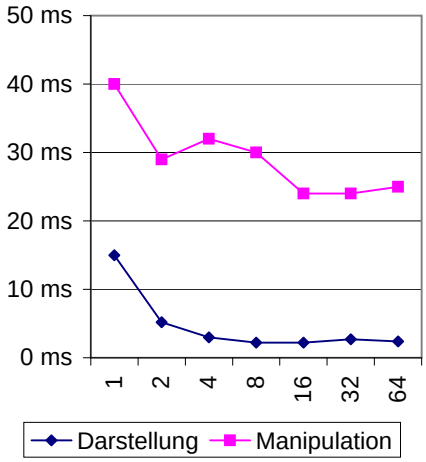
Werkzeug	Kosten
Identity	21 ms
Scale	25 ms
Smooth	27 ms
Dsiplace	28 ms

Abbildung 4.6: Zeit-Kosten der Werkzeuge beim Start eines Strokes



Auflösung	Darstellung	Manipulation
16	2 ms	4 ms
32	2 ms	4 ms
64	2 ms	4 ms
128	3 ms	6 ms
256	4 ms	20 ms
512	152 ms	42 ms
1024	250 ms	418 ms

Abbildung 4.4: Verarbeitungszeiten im Verhältnis zur Auflösung des Geometry-Image



Sub-Sampling	Darstellung	Manipulation
1:1	15 ms	40 ms
1:2	5 ms	29 ms
1:4	3 ms	32 ms
1:8	2 ms	30 ms
1:16	2 ms	24 ms
1:32	2 ms	44 ms
1:64	2 ms	25 ms

Abbildung 4.5: Verarbeitungszeit im Verhältnis zum Sub-Sampling eines Geometry-Image bei einer Auflösung von 512×512 Vertices

Die Performanz ist theoretisch von der Länge der Strokes abhängig, da diese jedes Frame komplett neu gemalt werden müssen. In der Praxis ist diese Komponente aber vernachlässigbar. Weiter sind Brushes immer schnell, unabhängig von ihrer Größe. Am Anfang jedes Strokes, entstehen kleine Pausen in Größenordnung 50 ms da hier das Geometry-Image einmal komplett tranformiert wird. Abschließend muss leider festgestellt werden, dass die Ziel-Auflösung von 1 Million Faces noch nicht schnell genug verarbeitet werden kann (Knick in Abb. 4.4 zwischen Stufe 6 und Stufe 7). Woran das liegt kann nur vermutet werden. Möglicherweise wird zuviel Video-Speicher verwendet und die Daten werden über den AGP-Port geschrieben oder gelesen.

Trotz der guten Performanz bleibet dbzgl. bedauerlich viel unklar:

- Was ist der Treiber?
- Was ist die Hardware?
- Was sind Fehler im Treiber?
- Wo kam der Stall her?
- Was passiert in welcher Reihenfolge?
- Was ist wann synchron, was wann asynchron?

Die einfachen und vernünftigen Techniken zur Optimierung von Grafik-Applikationen sind alle leicht anzuwenden und liefern Teilweise einen Beitrag, aber in anderen Fällen ist es unergründlich wo Zeit verschwindet. Zum besseren Verständnis der Performanz, wurde rudimentär `nvperfkit` (NVia) und `nvshaderperf` (NVib) verwendet. `Nvperfkit` arbeitet noch zu eingeschränkt (spezielle Treiber, usw.) und die Visualisierung der Ergebnisse durch die Windows-Ereignis-Anzeige ist schwierig zu handhaben. `Nvshaderperf` liefert aufschlussreiche Informationen über die Kosten der Shader, z. B. auch deren zu erwartender mittlerer oder maximaler Durchsatz auf einer bestimmten Ziel-Hardware. Eine genauer Analyse und Optimierung wäre hier möglich.

4.3 Probleme

Die Implementierung hat noch zahlreiche Probeleme. Die meisten sind bei der Wirkung der Tools festzustellen - dort kommt es zu oft zu Artefakten. Ohne die dynamische Parametrisierung, automatisch, halb-automatisch oder manuell, ist das System noch nicht einmal für allgemeine entwickelbare Flächen anwendbar - es zu Gebieten die so undersampelt sind, dass man sie nicht mehr bearbeiten kann.

Nicht alle Ränder in allen Varianten werden erfolgreich behandelt. Die Behandlung der Ränder stellt einen erheblichen Aufwand bei der Programmierung sowie zur Laufzeit dar.

Wird ein Tool angewandt, kann es passieren, dass kleine *Offsets* entstehen. Diese sind oft nur in der Größenordnung von $< \pm 1\,\%$ (der inversen Auflösung), entstehen bei Rundungsfehlern bei Textur-Zugriffen und optisch nicht zuerkennen. Durch wiederholtes anwenden der Tools, akkumuliert sich dieser Fehler und Features „wandern“ beim Bearbeiten über die Oberfläche. Das ein solcher Fehler wirk, fällt erst nach sehr vielen Schritten auf, oft erst so spät, dass man mehr als 32-Undo-Schritte zurückgehen müßte, und dieser wird damit irreparabel.

4.4 Vergleich

Ein praktischer Vergleich mit anderen Implementierungen fand nicht statt. Lediglich kommerzieller Software, die aber technisch nicht auf dem neusten Stand ist, wurde zum Vergleich herangezogen. So ist z. B. nicht getestet worden, wie schnell Methoden zur Manipulation von Point-Clouds sind und auch die praktische Anwendbarkeit konnte nicht getestet werden. Ein analytischer Vergleich mit Beschreibungen anderer Systeme allerdings ist möglich.

Discreet 3ds max bietet die Möglichkeit Vertex-Attribute auf einen Körper zu malen, ohne diesen zu verformen. Dabei kann es beliebige Körper bemalen, da die Attribute pro Vertex abgelegt werden. Solche Körper sind von der Auflösung bei ca. 50k Faces beschränkt, was aber an der Darstellung liegt, die nicht adäquat mit neuerer Hardware skaliert. Es ist nicht zu erkennen, wieso max an dieser Stelle mit der GPU skalieren sollte - es ist ja nicht einmal in der Lage die Darstellung für eine Größenordnung Faces weniger zu leisten.

Z-Brush arbeitet vergleichbar, kann jedoch besser mit Details und allgemeinen Flächen umgehen, da es von einem Basis-Mesh verschiedenen Flächen verschiedne Tief unterteilen kann. Das funktioniert so lange gut, wie keine globalen Manipulationen vorgenommen werden, die mehrere Teile betreffen oder wenn viele Details an vielen Stellen vorhanden sind oder zusammen manipuliert werden. Zwar ist Z-Brush in der Lage, Körper beliebigen Genus zu erzeugen, wenn diese aber manipuliert wurden, ist der Genus festgelegt. Dieses Vorgehen ist zwar praktisch hilfreich, man kann hier aber nicht mehr von der Manipulation beliebiger Körper sprechen. Mit einer Multichart-Parametrisierung könnte ein solches Verhalten durch Geometry-Images emuliert werden.

Chameleon (IC01), löst das Problem der Parametrisierung, die vom Programm erzeugt wird. Auch das beschriebene System leistet dies. Physikalisch getriebene Manipulation wie bei Lawrence und Funkhouser (LF03) ist nicht möglich, scheint

aber als Erweiterung denkbar.

Die Performance-Werte sind bedingt mit denen von Kniss et al. (KLS$^+$05) vergleichbar: die Darstellung ist unabhängig von der Bearbeitung und erfolgt in hoher Geschwindigkeit. Die Bearbeitung ist interaktiv ebenfalls in hoher Geschwindigkeit möglich. Ein wesentlicher Nachteil des beschriebenen Systems ist die Beschränkung auf entwickelbare Flächen bzw. auf eine Parametrisierung: Kniss et al. bemalen den Raum selbst, nicht nur Oberflächen. Die Beschränkung auf die Oberfläche ist dort nur eine künstliche Beschränkung. In wieweit ein Octree in der Lage wäre auf eine *verformende* Oberfläche, wie im beschriebenen System, zu reagieren ist schwer zu beurteilen.

Kapitel 5

Fazit

5.1 Erweiterungen und Verbesserungen

5.1.1 Umgang mit Ressourcen

Eine mögliche Verbesserung ist der konservativere Umgang mit Ressourcen. Es müsste nicht immer wenn ein Tool angewandt wurde gleich das ganze Geometry-Image einmal modifiziert werden und alle Normalen neu berechnet werden. Man könnte ein Dirty-Bereich führen, der erst wenn er wächst das Tool auf die neu hinzukommenden Bereiche anwendet. Das System geht durch diese Verhalten auch verschwenderisch mit Hardware-Resourcen um. Für z. B. eine mobile Anwendung würde man Performanz und Einfachheit gegen Komplexität und Strom-Sparen zu tauschen haben.

5.1.2 Andere Parametrisierung

Es wäre zu prüfen, ob nicht andere Formen der Geo-Sphere eine geringere Distortion liefert. Praun (PH03) vergleicht verschiedene Alternativen, auch für das Mesh-Painting einmal getestet werden sollten.

Es ist denkbar, dem Nutzer Einfluss auf die Parametrisierung zu geben. Dazu sind zwei Optionen denkbar. Erstens: ohne dynamische Parametrisierung, wäre ein einfaches Tool denkbar, dass die Dichte der Samples an der Oberfläche verändert, indem dort einfach ein Re-Sampling ausgeführt wird, entweder mit mehr, oder mit weniger Samples als bis jetzt dort vorhanden sind. Zweitens: Durch dynamisch Parametrisiert, würde dieses Verhalten wieder aufgehoben werden, da ja versucht wird, dass Sampling überall von konstanter Dichte zu erhalten. Man müsste da-

her hier einen weiteren skalaren Mesh-Chanel erzeugen, der angibt, wie lang die gewünschte Kantenlänge an einer Stelle des Meshes ist. Sloan (SWB98) beschreibt eine ähnliche Technik.

5.1.3 Detail-Maps

Zur Verbesserung der Darstellung wäre es interessant, für bestimmte Chanels nicht nur eine Textur zu verwenden, sondern die Textur als Gewichtung für andere Texturen zu verwenden, die sich wiederholen (Detail-Maps). Diese Technik könnte auch mit normalen Texturen gemeinsam verwendet werden. Der Shader zur Darstellung des Meshes müsste dann so flexibel (Preprocessor) gestaltet werden, dass er ohne und mit einer beliebigen Anzahl von Detail-Maps arbeiten könnte.

5.1.4 Darstellung

Zur Darstellung wird momentan wenige einfache Shader verwendet. Dabei bieten gerade die auf der GPU verfügbare Geometrie vollkommen neue Möglichkeiten.

Image-Space-Lighting

So ermöglicht z. B. das sog. Image-Space-Lighting (ISL) (nicht mit IBL zu verwechseln, weil auch gemeinsam auftretend), eine Annäherung an SSS zu liefern, oder auch allgemeiner Modell-Komplexität und Beleuchtungs-Komplexität über eine Parametrisierung zu entkoppeln (BL03; SGM04). Dazu wird das Lighting dynamisch in eine Textur gerendert (z B. mit IBL) , indem man in eine Textur rendert und Positionen und Texturkoordinaten beim Rendern vertauscht und trotzdem mit den Positionen lighted [1].

$$l(s) = lighting(g(s)), s \in [0 \ldots 1)^2$$

l wird im folgenden auch „Lightmap“ genannt. Danach wird das entstanden Bild geblurt, was einer Annäherung an den Licht-Transport unter der Oberfläche darstellt. Der Blur kann pro Vertex einstellbar (ein Beispiel für ein weiteres Mesh-Feature, einen weiteren Mesh-Chanel) sein und an den Stellen stark, wo viel SSS wirkt: Nase, Ohren, Lippen. Die so entstandene Textur wird beim Rendern anstelle der diffusen Komponente des Beleuchtungsmodell verwendet.

All diese Operationen unterstützt das vorgestellte System und zwar teilweise um Größenordnungen besser als irregulären Meshes:

[1] Diese Darstellungs wird bei die Praun et al. oder Gu et al. (PH03; GGH02) zur verbesserten 2D-Visualisierung des Geometry-Image verwendet.

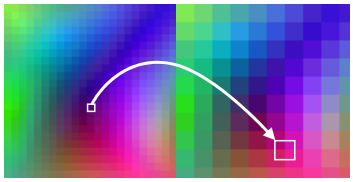


Abbildung 5.1: Formfaktor $F^{i,i'}_{(s,t),(s',t')}$ (weißer Pfeil) zwischen zwei Leveln (i rechts, i' links) und zwei Flächen (s,t und s',t') eines Geometry-Image. Die Kästchen stellen die zu den Vertices gehörenden Flächen dar.

- Die Diskontinuitäten der Parametrisierung ist minimal – es „fließt“ kein falsches Lighting an den Rändern in die Charts.
- Der Transport-Filter ist auf der GPU durchführbar
- Es müssen nicht alle Faces einzeln mit Position und Textur-Koordinate vertauscht gerendert werden – das Geometry-Image entspricht bereits diesem Schritt! Es muss lediglich *ein* Quad gezeichnet werden, dass im FP die Position aus dem Geometry-Image liest und diese gelighted zurück schreibt. Ohne Geometry-Images müssen so *alle* Faces in die Textur gezeichnet werden.

Es ist nun auch möglich, Berechnungen auf LOD-Leveln der Lightmap auszuführen. Dazu werden für die Lightmap $l(s)$ alle nötigen LOD-Level erzeugt $l^i(s)$. Auf diesen Stufen kann dann eine komplexere Beleuchtung durchführen, wie z. B. AO oder besseres SSS. Nachteil von ISL ist es, dass so auch Flächen gelighted werden, die nicht sichtbar sind.

Formfaktoren

Es ist denkbar, eine Art „Formfaktoren“ zwischen Flächen auf verschiedenen LOD-Stufen zu konstruieren: $F^{i,i'}_{(s,t),(s',t')}$, i und i' sind die Level und s,t und s',t' die Koordinate im Geometry-Image (Siehe Abb. 5.1).

Diese könnte man dann z. B. verwenden, um nach einen ISL-Schritt statt zu bluren, das transferierende Licht auf verschiedenen Hierarchiestufen in einer 3er-Nachbarschaft einzusammeln. Die Formfaktoren wären bei jeder Verwendung „in-place“ neu zu berechnen, da sie immer nur einmal gebraucht werden.

$$l^{i'}_{SSS,(s,t)} = \sum_{i=i'}^{i_{max}} \sum_{j=-1}^{1} \sum_{k=-1}^{1} l^{i}_{(s+k)2^{-i},(t+j)2^{-i}} F^{i',i}_{(s,t),(s+k,t+j)}$$

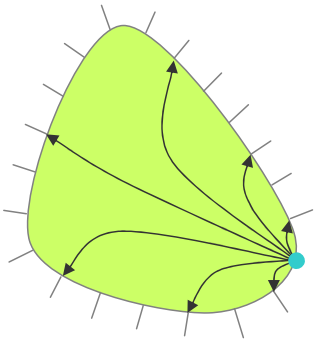


Abbildung 5.2: Hierarchischer Licht-Transport unter der Oberfläche. Patches die entfernet liegen werden zusammengefasst (oder: wurden zusammengefasst und liegen auf einem anderen Level der Lightmap-Pyramide), andere nicht.

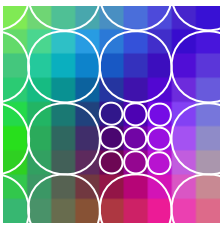


Abbildung 5.3: Geometry Image mit einem Pixel und den zusammengefassten Kreis-Scheiben seiner Umgebung.

Dies ist auf jeder, auch niedrigeren Stufe gegegen eine Auswahl von niedrigeren Stufen denkbar. Beispiel: ein 512×512-Geometry-Image wird in 256×256 diffus ISL-gelighted, und auf der Stufe 64×64 wird ein Transfer zu den Nachbarn in 16, 8 und 4 durchgeführt. Dies macht nur dann Sinn, wenn Flächen die in der Parametrisierung weit entfernt sind, eher mit einem gemeinsamen Formfaktor beschrieben werden können - keine wirklich abwegige Überlegung, aber für bestimmte Fälle beliebig falsch. Hier liegt ein Unterschied zu Ansätzen wie bei Mertens et al. (MKB+03), die u. A. die Formfaktor-Hierarchien akkurat umsetzen.

Ambience Occlusion

Auch Schatten wären eine erstrebenswerte Ergänzung, da sie die Raumwahrnehmung verbessern können (HLHS03). Eine einfache Möglichkeit dazu stellt Shadow-Mapping dar, eine andere, aufwendigere, die in GPU Gems 2 (Bun05) beschriebene Technik zur Darstellung von AO.

AO wäre eine Ideal Ergänzung und performant möglich, da sie sowieso auf Geometrie-Verarbeitung durch die GPU angewiesen ist. Hier ist das Vorgehen ähnlich

wie bei Formfaktoren. Soll für einen Punkt die Verschattung festgestellt werden, werden alle Faces als Kreisscheiben betrachtet und auf eine Halbkugel über dem Punkt projiziert. Die Verschattung wird proportional zum Anteil der projizierten Scheiben an der Gesamtfläche gewählt. Als Beschleunigung werden mehrere Faces, die weit weg liegen, mit einer Kreis-Scheibe angenähert. Für eine Pyramie aus Geometry-Images und die beiden Annahmen das im Ort entfernte Flächen auch in der Parametrisierung entfernt sind, und das Flächen die weit weg sind besser zusammengefasst werden können, wird die Auswahl der Kreis-Scheiben effizient möglich. Bei ISL wird zu jedem Vertex die 3er-Nachbarschaft auf mehrere, niedrigeren LOD-Stufen als Kreis-Scheibe betrachtet.

5.1.5 Physik

Es wäre weiter interessant zu untersuchen, ob Tools die physikalisches Verhalten simulieren möglich und sinnvoll sind. Ein Tool das die Oberfläche verändern kann als wäre diese aus Stoff, müsste das Geometry-Image lediglich als ein großes Feder-Masse-System ansehen. Solche Systeme lassen sich einfach und auf der GPU durch z. B. Verlet-Integration lösen (GEW05). Weiter ist denkbar, die Simulation von den Tools zu trennen, und nach jedem Bearbeitungsschritt einen Simulations-Schritt durchzuführen, bei dem verschiedene Constraints angewandt würden. Dabei wäre Cloth nur ein Sonderfall. Nutzer würden dann Einfluss auf die Zeit haben und könnten arbeiten wie in bei Lawrence und Funkhouser (LF03) beschrieben.

5.1.6 Haptisches Feedback

Die Anbindung eines Phantoms als haptisches Gerät wäre möglich. Die rotatorischen Freiheitsgarde wären zunächst ohne Beduetung, optional könnten die Tools sie verarbeiten. Wie auch immer das Feedback konkret arbeiten würde, es müssen Abstände zur Oberfläche festgestellt werden um Gegen-Kräfte erzeugen zu können. Da die Oberfläche, wie beim Picking (Siehe Abs. 3.1.2) auf der GPU vorliegt und sich beständig verformt, scheiden vorberechnete Informationen wie Distance-Fields, BSP, usw. aus.

Eine für GPU typische Brute-Force-Variante wäre als Näherung dennoch möglich. Der Abstand kann dabei im schlechtesten Fall nur so genau berechnet werden, wie die längste Kante im Mesh lang ist, was aber wenig ist, vor allem wenn die Tesslierung gepflegt wird. Dazu wird einfach der Abstand mit Vorzeichen (Signed Distance) des 3D-Curssrs zu jedem Vertex berechnet. Ist er negativ, ist er bereits eingedrungen, und die Richtung in der er am meisten eingedrungen ist, muss die Kraft wirken, um die Penetration aufzuheben. Sind mehrere Abstände negativ muss der kleinste verwendet werden. Es muss also aus allen berechneten Abständen das

Minimum gebildet werden und wenn es negativ ist, reagiert werden. Ein Minimum ist einer der *möglichen* Reduktionen (OLG+05). Es werden also in einem Kernel immer 4 Pixel mit fixem Code verglichen und nur das Minimum geschrieben. Das wird rekursiv fortgesetzt. Der letzte Pixel ist das Minimum. In dessen Richtung würde die Kraft wirken, wenn er negativ ist

5.1.7 Allgemeinere Flächen

Die in der Praxis größte Beschränkung des Systems ist die auf entwickelbare Flächen. Es wäre interessant, ob auch nur stückweise reguläre Geometry-Images (SWG+03) auf der GPU effizient verarbeitet werden können. Um Risse an den Rändern der Charts zu vermeiden, muss das Mesh vor dem Rendern mit einem Zip-Locking geschlossen werden – wie sollte dies auf der GPU geschehen? Und selbst wenn es möglich wäre, wie verhält sich ein Atlas, wenn sich die Geometrie ändert? Carr (CH04b), schlägt eine Erweiterung vor, die den Atlas dynamisch wieder aufbaut, und erreicht dabei für 100 000 Flächen 10 Hz, was eine Größenordnungen zu wenig ist. Auch sind die dort verwendeten Techniken eher Schwergewichte, und benötigen z. B. ca. 100 MB Speicher für diesen Fall!

5.2 Bewertung

Die in der Einleitung gestellten Fragen können nun teilweise beantwortet werden: Ja, Mesh-Painting auf der GPU ist möglich. Es gibt Nachteile, die aber mittelfristig lösbar scheinen, entweder durch weitere Verbesserungen der Hardware (z. B. schnelle 32-bit Floating-Point-Unterstützung, wie in der NVIDIA GeForce 7) oder zusätzlichen Aufwand in der Verarbeitung (Parametrisierung, Größenkonstanz). Geschwindigkeit ist die wesentliche Zielsetzung dieser Arbeit und man kann mit den Ergebnissen zufrieden sein: es ist auf einem handelsüblichen PC möglich, ein Mesh mit einer Million Faces zu bemalen, dass mit leichter Vereinfachung dargestellt wird. Im Weiteren skaliert die Geschwindigkeit mit der GPU – die Konsequenzen sind bekannt: die technische Entwicklung der GPU ist wesentlich schneller. Unklar ist, in wie weit die schlechten Tools die mittelmäßigen praktischen Ergebnisse bedingen, oder ob es doch konzeptionelle Probleme gibt. Diese Frage kann so noch nicht beantwortet werden.

Es ist möglich die Darstellung zu verbessern, obwohl es immer noch zu übersteuerter und undersampelnder Geometrie kommen kann, die negativ auffallen kann. Besonders an den Stellen, wo die Anisotropie groß ist, muss noch mit großer Vorsicht editiert werden, zu kleine und zu starke Pinsel, lassen die Mesh-Qualität schnell degradieren. Nicht alle Zielsetzungen der Studienarbeit wurden erreicht: Es existiert keine Funktionalität zum Laden und Speichern von Geometry-Images, es wurde

kein haptisches Eingabegerät angebunden, obwohl ein SensAble Phantom® zur Verfügung stand. Es können keine Empfehlungen zur weiteren Verarbeitung ausgesprochen werden, da keine Serialisierung implementiert wurde.

Perspektivisch ist davon auszugehen, dass die Manipulation von Geometry-Images auf der GPU für eine bestimmte Klasse von Modellierungs-Fällen zu einem produktiven Werkzeug ausgebaut werden kann, das die Modellierung von organischen Flächen mit vielen keinen, aber relevanten Features, die gleichzeitig eine hohe Darstellungs-Qualität wie AO oder SSS benötigen performant unterstützt.

Abkürzungen

AA Anti-Aliasing

AGP Accelerated Graphics Port

AO Ambience-Occlusion

FBO Frame Buffer Objects

FP Fragment Program

GLSL OpenGL Shading Language

GPU Geometry Procession Unit

IBL Image Based Lighting

ISL Image Space Lighting

MRT Multiple Render Targets

PBO Pixel Buffer Objects

SSS Sub-Surface Scattering

VBO Vertex Buffer Objects

VP Vertex Programm

Literaturverzeichnis

[ABL95] AGRAWALA, Maneesh ; BEERS, Andrew C. ; LEVOY, Marc: 3D painting on scanned surfaces. In: *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*. New York, NY, USA : ACM Press, 1995, S. 145–ff.

[Ali] ALIAS SYSTEMS CORP.: *Maya 7*. `http://www.alias.com/eng/index.shtml`

[Aut] AUTODESK MEDIA & ENTERTAINMENT: *Autodesk 3ds Max*. `http://www.autodesk.com`

[BL03] BORSHUKOV, George ; LEWIS, J. P.: Realistic human face rendering for "The Matrix Reloaded". In: *GRAPH '03: Proceedings of the SIGGRAPH 2003 conference on Sketches & applications*. New York, NY, USA : ACM Press, 2003, S. 1–1

[BSM+03] BRICENO, H. ; SANDER, P. ; MCMILLAN, L. ; GORTLER, S. ; HOPPE, H.: Geometry videos: A new representation for 3d animations. In: *Proceedings of ACM Symposium on Computer Animation 2003.*, 2003

[Bun05] *Kapitel* 14. In: BUNNELL, M.: *Dynamic Ambient Occlusion and Indirect Lighting*. Addison-Wesley, Reading, MA, 2005

[Car04] CARR, Nate: *Surface Processing on Graphics Hardware*, University of Illinois Urbana-Champaign, Diss., 2004

[CH04a] CARR, Nathan A. ; HART, John C.: Painting detail. In: *ACM Trans. Graph.* 23 (2004), Nr. 3, S. 845–852. – ISSN 0730–0301

[CH04b] CARR, Nathan A. ; HART, John C.: Two algorithms for fast reclustering of dynamic meshed surfaces. In: *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. New York, NY, USA : ACM Press, 2004. – ISBN 3–905673–13–4, S. 224–234

[Deb] DEBEVEC, Paul: *Light Probe Image Gallery*. `http://www.debevec.org/Probes/`

[DGPR02] DEBRY, David (grue) ; GIBBS, Jonathan ; PETTY, Devorah D. ; ROBINS, Nate: Painting and rendering textures on unparameterized models. In: *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 2002, S. 763–768

[DHQ04] DUAN, Ye ; HUA, Jing ; QIN, Hong: HapticFlow: PDE-based mesh editing with haptics. 15 (2004), Juli, Nr. 3–4, S. 193–200

[Duf98] DUFILHO, Karen: Geri's game. In: *SIGGRAPH '98: ACM SIGGRAPH 98 Electronic art and animation catalog*. New York, NY, USA : ACM Press, 1998, S. 131

[FOL02] FOSKEY, Mark ; OTADUY, Miguel A. ; LIN, Ming C.: ArtNova: Touch-Enabled 3D Model Design. In: *Proceedings of the IEEE Virtual Reality Conference* (2002), S. 119–126

[GEL00] GREGORY, Arthur D. ; EHMANN, Stephen A. ; LIN, Ming C.: inTouch: Interactive Multiresolution Modeling and 3D Painting with a Haptic Interface. In: *VR '00: Proceedings of the IEEE Virtual Reality 2000 Conference*. Washington, DC, USA : IEEE Computer Society, 2000, S. 45

[GEW05] GEORGII, Joachim ; ECHTLER, Florian ; WESTERMANN, Rüdiger: Interactive Simulation of Deformable Bodies on GPUs. In: *Simulation and Visualisation 2005*, 2005

[GGH02] GU, Xianfeng ; GORTLER, Steven J. ; HOPPE, Hugues: Geometry images. In: *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 2002, S. 355–361

[GGSC98] GOOCH, Amy ; GOOCH, Bruce ; SHIRLEY, Peter ; COHEN, Elaine: A Non-Photorealistic Lighting Model for Automatic Technical Illustration. In: *Proceedings of SIGGRAPH 98* (July 1998), S. 447–452

[GHQ04] GUO, Xiaohu ; HUA, Jing ; QIN, Hong: Touch-Based Haptics for Interactive Editing on Point Set Surfaces. In: *IEEE Comput. Graph. Appl.* 24 (2004), Nr. 6, S. 31–39

[HH90] HANRAHAN, Pat ; HAEBERLI, Paul: Direct WYSIWYG painting and texturing on 3D shapes. In: *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 1990, S. 215–223

[HLHS03] HASENFRATZ, Jean-Marc ; LAPIERRE, Marc ; HOLZSCHUCH, Nicolas ; SILLION, François: A survey of Real-Time Soft Shadows Algorithms. In: *Eurographics* Eurographics, Eurographics. – State-of-the-Art Report

[IC01] IGARASHI, Takeo ; COSGROVE, Dennis: Adaptive unwrapping for interactive texture painting. In: *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*. New York, NY, USA : ACM Press, 2001, S. 209–216

[KBR04] KESSENICH, John ; BALDWIN, Dave ; ROST, Randi: *The OpenGL Shading Language*

[KLS+05] KNISS, Joe ; LEFOHN, Aaron ; STRZODKA, Robert ; SENGUPTA, Shubhabrata ; OWENS, John D.: Octree Textures on Graphics Hardware. (2005)

[LF03] LAWRENCE, Jason ; FUNKHOUSER, Thomas: A Painting Interface for Interactive Surface Deformations. In: *Pacific Graphics* (2003), Oktober

[LHSW03] LOSASSO, F. ; HOPPE, H. ; SCHAEFER, S. ; WARREN, J.: Smooth geometry images. In: *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2003, S. 138–145

[LKS+05] LEFOHN, Aaron ; KNISS, Joe M. ; STRZODKA, Robert ; SENGUPTA, Shubhabrata ; OWENS, John D.: Glift: Generic, Efficient, Random-Access GPU Data Structures. In: *ACM Transactions on Graphics* (2005)

[LPRM02] LEVY, Bruno ; PETITJEAN, Sylvain ; RAY, Nicolas ; MAILLOT, Jerome: Least squares conformal maps for Automatic Texture atlas Generation. In: *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 2002, S. 362–371

[MKB+03] MERTENS, Tom ; KAUTZ, Jan ; BEKAERT, Philippe ; SEIDEL, Hans-Peter ; REETH, Frank V.: Interactive rendering of translucent deformable objects. In: *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2003, S. 130–140

[NVia] NVIDIA CORP.: *NVPerfKit 1.0 RC1*. `http://developer.nvidia.com/object/nvperfkit_home.html`

[NVib] NVIDIA CORP.: *NVShaderPerf*. http://developer.nvidia.com/object/nvshaderperf_home.html

[NVi04] NVIDIA CORP.: *Release Notes for NVIDIA OpenGL Shading Language Support*

[NVi05a] NVIDIA: *NVidia ForceWare Graphics Drivers: Release 75 Notes*. http://download.nvidia.com/Windows/77.77/77.77_ForceWare_Release_Notes.pdf. Version: 2005

[NVi05b] NVIDIA: *Nvidia GeForce 7800 Tech Specs*. http://www.nvidia.com/page/specs_gf7800.html. Version: 2005

[OLG+05] OWENS, John D. ; LUEBKE, David ; GOVINDARAJU, Naga ; HARRIS, Mark ; KRÜGER, Jens ; LEFOHN, Aaron E. ; PURCELL, Timothy J.: A Survey of General-Purpose Computation on Graphics Hardware. In: *Eurographics 2005, State of the Art Reports*, 2005, S. 21–51

[Ope01] OPENGL ARB: *WGL_ARB_pbuffer OpenGL extension*. http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt. Version: 2001

[Ope04a] OPENGL ARB: *ARB_draw_buffers*. http://oss.sgi.com/projects/ogl-sample/registry/ARB/draw_buffers.txt. Version: 2004

[Ope04b] OPENGL ARB: *ARB_pixel_buffer_object OpenGL extension*. http://oss.sgi.com/projects/ogl-sample/registry/ARB/pixel_buffer_object.txt. Version: 2004

[Ope04c] OPENGL ARB: *NV_primitive_restart*. http://oss.sgi.com/projects/ogl-sample/registry/NV/primitive_restart.txt. Version: 2004

[Ope05a] OPENGL ARB: *ARB_texture_float*. http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_float.txt. Version: 2005

[Ope05b] OPENGL ARB: *ARB_vertex_buffer_object OpenGL extension*. http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt. Version: 2005

[Ope05c] OPENGL ARB: *EXT_framebuffer_object OpenGL extension*. http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt. Version: 2005

[PH03] PRAUN, Emil ; HOPPE, Hugues: Spherical parametrization and remeshing. In: *ACM Trans. Graph.* 22 (2003), Nr. 3, S. 340–349. – ISSN 0730–0301

[Pix] PIXOLOGIC: *ZBrush 2*. http://pixologic.com/home/home.shtml

[PP03] PIERCE, Jeffrey S. ; PAUSCH, Randy: Specifying interaction surfaces using interaction maps. In: *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*. New York, NY, USA : ACM Press, 2003, S. 189–192

[Rig] RIGHT HEIMSPHERE: *Deep Paint 3D*. http://www.righthemisphere.com/products/dp3d/Deep3D_UV/index.htm

[Sen] SENSABLE TECHNOLOGIES: *FreeForm Concept*. http://www.sensable.com/products/3ddesign/concept/index.asp

[SGM04] SANDER, Pedro V. ; GOSSELIN, David ; MITCHELL, Jason L.: Real-Time Skin Rendering on Graphics Hardware. In: *GRAPH '04: Proceedings of the SIGGRAPH 2004 conference on Sketches & applications*. New York, NY, USA : ACM Press, 2004

[SLCO+04] SORKINE, Olga ; LIPMAN, Yaron ; COHEN-OR, Daniel ; ALEXA, Marc ; RÖSSL, Christian ; SEIDEL, Hans-Peter: Laplacian Surface Editing. In: *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing*, Eurographics Association, 2004, S. 179–188

[ST04] SIBLEY, Peter G. ; TAUBIN, Gabriel: Atlas-Aware Laplacian Smoothing. In: *VIS '04: Proceedings of the conference on Visualization '04*. Washington, DC, USA : IEEE Computer Society, 2004, S. 598.27

[SWB98] SLOAN, Peter-Pike J. ; WEINSTEIN, David M. ; BREDERSON, J. D.: Importance Driven Texture Coordinate Optimization. In: *Comput. Graph. Forum* 17 (1998), Nr. 3, S. 97–104

[SWG+03] SANDER, P. V. ; WOOD, Z. J. ; GORTLER, S. J. ; SNYDER, J. ; HOPPE, H.: Multi-chart geometry images. In: *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2003, S. 146–155

[TD] TCHOU, C. ; DEBEVEC, P.: *HDR Shop*. http://www.debevec.org/HDRShop

[Weia] WEISSTEIN, Eric W.: *'Haar Function', MathWorld*. http://mathworld.wolfram.com/HaarFunction.html

[Weib] WEISSTEIN, Eric W.: *'Manifold', MathWorld*. http://mathworld.wolfram.com/Manifold.html

[Wik05a] WIKIPEDIA: *Accelerated Graphics Port*. http://de.wikipedia.org/wiki/AGP. Version: 2005

[Wik05b] WIKIPEDIA: *PCI-Express*. http://de.wikipedia.org/wiki/PCI-Express. Version: 2005

[Wlo03] WLOKA, Matthias: *Batch, batch, batch - What does it really mean?* http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf. Version: 2003

[ZS00] ZORIN, Denis ; SCHRÖDER, Peter: *Subdivision for Modeling and Animation*. 2000. – Course Notes