Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

Diplomarbeit

Hardwaregestütztes Stippling von medizinischen Oberflächenmodellen

Alexandra Baer

Institut für Simulation und Graphik

Alexandra Baer:
Matrikelnummer: 15 76 82
Hardwaregestütztes Stippling von medizinischen Oberflächenmodellen
Diplomarbeit, Otto-von-Guericke Universität Magdeburg, 2005.

Hardwaregestütztes Stippling von medizinischen Oberflächenmodellen

Diplomarbeit

an der
Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von:	Alexandra Baer
geb. am:	17. August 1980
in:	Leipzig
Matrikelnummer:	15 76 82

1. Gutachter:	Prof. Dr.-Ing. Bernhard Preim
2. Gutachter:	Dr.-Inf. Bert Freudenberg
Betreuer:	Prof. Dr.-Ing. Bernhard Preim Dipl.-Ing. Christian Tietjen Dipl.-Ing. Martin Spindler

Zeit der Diplomarbeit:	3.03.2005 - 27.09.2005

Selbstständigkeitserklärung

Hiermit versichere ich, Alexandra Baer (Matrikel-Nr. 157682), die vorliegende Arbeit allein und nur unter Verwendung der angegebenen Quellen angefertigt zu haben.

Alexandra Baer

Magdeburg, 27.09.2005

Danksagung

An erster Stelle steht mein Familie, allen voran meine Eltern Antje und Ulrich Baer sowie Ute Pesselt. Ihre Liebe und Unterstützung hat es mir ermöglicht dieses Studium zu absolvieren und meine Ziele umzusetzen. Bei meinen Schwestern Lena und Sabine bedanke ich mich für all die schönen Stunden der Ablenkung und Aufmunterung. Ich bin euch unendlich dankbar, dass ihr auch wenn es einige schwere Zeiten gibt, immer hinter mir steht.

Ein besonderer Dank geht an meine Großmutter Rotraud Schmidt, die stets an mich geglaubt hat und mir in allen Lebenslagen hilfreich zur Seite stand.

Zu tiefsten Dank bin ich auch meinen Betreuern Prof. Dr. Bernhard Preim, Dipl.-Ing. Christian Tietjen sowie Dipl.-Ing. Martin Spindler verpflichtet, die mich während meiner Diplomzeit unterstützt und ihr Bestes gegeben haben, um mir bei meinen Fragen und Problemen weiterzuhelfen.

Für eine wunderbare Freundschaft, den Rückhalt sowie stets aufmunternde Worte möchte ich den beiden liebsten Freundinnen Kerstin Kellermann und Manuela Kuhn danken, die mich besonders im letzten Jahr stets unterstützt haben und zu mir standen. Vor allem werden mir die vielen Montagabende fehlen, ohne die ich ganz bestimmt nicht so entspannt die Zeit überstanden hätte. Auch ihr Zuspruch sowie ihre Mithilfe während der Diplomzeit hat mir geholfen und mich stets aufgebaut.

Daniel Eicke, Sandra Hartmann, Uta Hinrichs, Björn Meyer, Konrad Mühler, Christian Schulz, Heiko Seim und allen anderen Mitdiplomanden danke ich für die tägliche Motivation und Geselligkeit an manch endlosen Tagen und Nächten. In diesem Zusammenhang möchte ich auch Mathias Neugebauer nennen, der mich in nächtlicher Hilfsbereitschaft wieder auf den Pfad der objektorientierten Programmierung gelenkt hat. Für die kulinarischen Köstlichkeiten, dem fast täglichen Reaktionstraining, die abendlichen Aktivitäten zum mentalen Ausgleich sowie für die Rettung als Murphy erbarmungslos in letzter Minute zuschlug danke ich Anja Kuß. „Diplomzeit, schönste Zeit!“ auch wenn ich anfangs erst davon überzeugt werden musste.

Inhaltsverzeichnis

1 Einleitung

In der Computergraphik hat sich seit einigen Jahren die Richtung des nicht-photorealistischen Renderings (NPR) herausgebildet. Dieses Teilgebiet der Computergraphik beschäftigt sich unter anderem mit der Entwicklung neuer Visualisierungs- und Graphikstile sowie der Simulation künstlerischer Techniken. Das Ziel ist die Generierung von Bildern oder Animationen, die den Anschein erwecken, sie wären per Hand erstellt worden [Strothotte und Schlechtweg, 2002].

Wissenschaftliche Bücher, Atlanten oder technische Anleitungen verwenden oft derartige Darstellungsformen anstelle von Photographien, um komplexe Sachverhalte verständlicher abzubilden. Es werden bevorzugt *Pen-and-Ink*-Techniken wie Silhouetten, Merkmalslinien oder *Stippling* eingesetzt. Da die Generierung der Bilder per Hand ein sehr mühseliger und zeitintensiver Prozess ist, existieren zahlreiche Bestrebungen, die *Pen-and-Ink*-Methoden zu implementieren und Illustrationen computergeneriert zu erzeugen. Ferner wird ebenfalls versucht, diese Darstellungsweise in der Computergraphik für die Visualisierung von 3D-Objekten zu nutzen. NPR-Techniken können bei der Darstellung von 3D-Daten unter anderem Informationen über Form oder Orientierung der Strukturen vermitteln.

Als mögliche Anwendung für NPR-Techniken gilt auch die Generierung medizinischer Illustrationen, da bereits in den anatomischen Atlanten Abbildungen eingesetzt werden, die mit nicht-photorealistischen Stilmitteln Informationen übermitteln. Im Rahmen dieser Arbeit wird die *Stippling*-Technik zur Visualisierung von medizinischen 3D-Oberflächenmodellen genutzt. Unter Verwendung von *Shadern* werden Texturen auf die Modelloberfläche aufgebracht, um somit eine *Stippling*-Darstellung zu generieren. *Shader* ermöglichen eine schnelle Verarbeitung der 3D-Daten, einen beschleunigten Texturzugriff und demzufolge eine echtzeitfähige Visualisierung.

1.1 Zielsetzung

Ziel dieser Diplomarbeit ist eine echtzeitfähige Darstellung von medizinischen Oberflächen über die *Pen-and-Ink*-Technik *Stippling*. Hierfür ist es erforderlich, die charakteristischen Merkmale dieser Technik zu analysieren und in die

Visualisierung der Oberflächen zu integrieren. Für eine Analyse dienen neben einer Reihe von wissenschaftlichen Abbildungen auch die Illustrationen aus anatomischen Atlanten, die oft mit Hilfe verschiedenster *Pen-and-Ink*-Techniken erstellt werden.

Ferner existieren verschiedene Kriterien, die für Echtzeit-Visualisierung erfüllt sein müssen. Um die wesentlichen Anforderungen zu bestimmen und in der Visualisierung umsetzen zu können, werden bereits existierende Ansätze der Computergraphik zur Generierung von *Stippling*-Abbildungen untersucht. Weiterhin ist es erforderlich die verschiedenen Möglichkeiten zu betrachten, mit denen Objekte durch Punkte dargestellt werden können. Anhand der Vor- und Nachteile dieser Techniken kann die Ermittlung einer geeigneten Umsetzung erfolgen.

Programmierbare Graphik-Hardware ermöglicht eine beschleunigte Visualisierung und damit eine echtzeitfähige Darstellung. Vor allem bei der Verwendung von Texturen wird ein beschleunigter Texturzugriff durch *Shader* unterstützt. Alle auf den Daten auszuführenden Berechnungen sollen demnach direkt auf der Graphikkarte und ohne vorherige Vorverarbeitung des Geometriemodells ausgeführt werden. Dies geschieht mit Hilfe von *Shader*-Programmen. Die von NVIDIA bereitgestellte *Shader*-Entwicklungsumgebung RENDERMONKEY unterstützt hierbei den Entwurf und die Umsetzung dieser Programme. Ferner ermöglichen die *Shader*-Hochsprachen eine komfortable C-Sprachen-ähnliche Programmierung.

Die Diplomarbeit umfasst eine hardwaregestützte Echtzeit-Darstellung medizinischer Oberflächenmodelle über *Stippling*, wobei einerseits die wesentlichen Kriterien für *Stippling*-Illustrationen und andererseits die ermittelten Anforderungen an echtzeitfähige Visualisierungen erfüllt sein müssen. Bei der Verwendung von Texturen ist es nicht erforderlich eine exakt verzerrungsfreie Texturierung zu erreichen. Der Schwerpunkt liegt im Rahmen dieser Arbeit auf einer möglichst schnellen Visualisierung, ohne zusätzliche Bearbeitung des Objektgitters.

1.2 Gliederung der Arbeit

Die vorliegende Arbeit ist in sechs Kapitel gegliedert, welche folgende Themen beschreiben beziehungsweise untersuchen:

Kapitel 2 In diesem Kapitel werden zunächst die meist handgezeichneten Illustrationen in anatomischen Atlanten und die dafür verwendeten *Pen-and-Ink*-Techniken eingeführt und näher erläutert. Verschiedene bereits existierende Ansätze zur computergestützten Generierung derartiger Darstellungen werden vorgestellt und bezüglich ihrer Vor- und Nachteile für die Echtzeit-

Visualisierung von 3D-Modellen diskutiert. Ferner werden die für diese Arbeit benötigten Grundlagen für die Verwendung von Texturen beschrieben und die dabei auftretenden Verzerrungsprobleme untersucht. Abschließend erfolgt eine Einführung in das Konzept der programmierbaren Graphik-Hardware, die eine beschleunigte Visualisierung ermöglicht.

Kapitel 3 Basierend auf den betrachteten medizinischen Illustrationen und existierenden Ansätzen werden hier die Kriterien und Richtlinien einer Visualisierung von Oberflächenmodellen über *Stippling* definiert und für die Implementierung festgelegt. Ausgehend davon wird der Aufbau der benötigten *Stippling*-Texturen für eine beschleunigte, framekohärente Darstellung sowie die für eine verzerrungsfreie Abbildung erforderliche Parametrisierung konzipiert. Die erforderlichen *Shader*-Programme werden entworfen und eine notwendige Unterscheidung der Berechnungen pro *Vertex* und pro *Fragment* erarbeitet.

Kapitel 4 Eine hardwaregestützte Echtzeit-Visualisierung wird mit Hilfe der Entwicklungsumgebung RenderMonkey und der OpenGl *Shader*-Sprache realisiert. Die Berechnungen, die pro *Vertex* und pro *Fragment* auf den Daten ausgeführt werden sowie die Generierung der Texturen unter Beachtung der vorgestellten Kriterien wird hier ausführlich dargelegt. Weiterhin erfolgt hier die Präsentation und Auswertung der erzielten Resultate.

Kapitel 5 Abschließend werden die am Anfang der Arbeit herausgearbeiteten Anforderungen an Echtzeit-*Stippling*-Darstellungen mit den erzielten Ergebnissen verglichen und zusammengefasst. Die noch bestehenden Probleme werden erläutert und mögliche Lösungsansätze sowie Vorschläge für weiterführende Projekte aufgeführt.

2 Theoretische Grundlagen und Verwandte Arbeiten

In Rahmen dieser Arbeit wird eine hardwarebeschleunigte Echtzeit-Visualisierung medizinischer Oberflächenmodelle über die NPR-Technik *Stippling* betrachtet. In der Computergraphik wurden eine Reihe von NPR-Techniken entwickelt, die computergestützt *Pen-and-Ink*-Darstellungen nachahmen können. Aus diesem Grund wird auf bisherige Verfahren und Möglichkeiten zur Umsetzung einer Visualisierung eingegangen, wobei für die vorliegende Arbeit *Stippling*-Darstellungen relevant sind.

Die NPR-Darstellungen werden in den vorgestellten Verfahren entweder anhand der Objektgeometrie der Oberflächenmodelle generiert, basierend auf 2D-Graustufenbildern des 3D-Objektes erzeugt oder in bisherige Volumen-Rendering Verfahren integriert und somit direkt auf den Volumendaten ausgeführt. Ein bedeutender Ansatz für diese Arbeit ist die Generierung von NPR-Darstellungen mit Hilfe von Texturen. Hierfür werden wichtige Grundlagen, Verfahren und Probleme beim *Texture Mapping* erläutert. Es wird gezeigt, dass Texturen eine geeignete Alternative zu objekt-, bild- und volumenbasierten Ansätzen sind und welche Vorteile demnach von den einzelnen Verfahren ausgenutzt und welche Nachteile vermieden werden können.

Abschließend wird auf die Möglichkeiten und Vorteile programmierbarer Graphik-Hardware eingegangen, mit deren Unterstützung Echtzeit-Visualisierungen zeitsparend durchgeführt werden können.

2.1 Allgemeines

Für die Visualisierung medizinischer Daten müssen zunächst die herkömmlichen Illustrationen in Atlanten oder Lehrbüchern betrachtet werden. Sie enthalten unter anderem eine Reihe von handgezeichneten *Pen-and-Ink*-Abbildungen. Diese traditionellen Techniken werden dort zur besseren Darstellung eines medizinischen Sachverhaltes eingesetzt. Klassische Methoden, die in der Computergraphik

über NPR nachgeahmt werden, sind vor allem Silhouetten, Merkmalslinien und *Stippling*.

Eine Möglichkeit der Visualisierung medizinischer Daten ist das Volumen-Rendering. Neben dem traditionellen direkten und indirekten Volumen-Rendering gibt es auch Verfahren, die in diese herkömmlichen Verfahren die NPR-Techniken direkt integrieren.

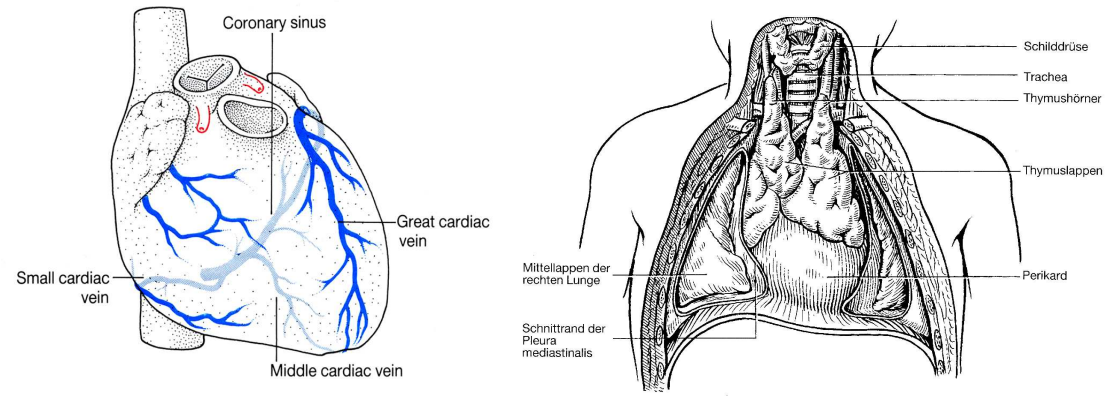
2.1.1 Visualisierung in medizinischen Atlanten

Von großer Bedeutung für die Visualisierung medizinischer Sachverhalte war 1543 das Erscheinen des ersten Anatomiebuches *De Humani Corpus Fabrica* von Andreas Versalius (1514-1564). Die darin enthaltenen Illustrationen über die menschliche Anatomie sind Holzschnitte. Bereits zu dieser Zeit galten medizinische Darstellungen als eine Form von Kunst und wurden bedingt durch die Drucktechnik als *Stippling*- oder Schraffur-Zeichnungen erstellt. Beide Techniken sind bis heute als Stilmittel erhalten geblieben und werden in vielen medizinischen Büchern verwendet.

Moderne anatomische Atlanten verwenden heutzutage neben Schwarz-Weiß-Zeichnungen auch farbige meist per Hand erstellte Zeichnungen. Im Gegensatz zu Photos sind diese Abbildungen in der Lage, viele Sachverhalte präziser und eindeutiger darzustellen und dabei trotzdem alle Einzelheiten abzubilden [Deussen et al., 2000]. Es gibt allerdings auch Atlanten, die anatomische Photos beinhalten auf denen Strukturen anhand präparierter menschlicher Leichen gezeigt werden oder apparative Aufnahmen wie Röntgen- oder Ultraschallbilder abgebildet sind.

Handzeichnungen sind für die Darstellung einiger Sachverhalte in Atlanten besser geeignet als Photos oder Bilder. Aufgrund der abstrakten Zeichenweise besitzen sie die Fähigkeit Informationen effektiv darzustellen, indem sie unwichtige Details weglassen oder mit geringerer Komplexität beziehungsweise Farbsättigung darstellen [Tietjen, 2004]. Da unwichtige Strukturen so dargestellt werden, das sie nicht störend wirken, wird die Aufmerksamkeit des Betrachters auf relevante Merkmale oder Details gelenkt. Hierfür werden auch Techniken wie *Stippling* und Silhouetten eingesetzt, die dezent zusätzliche Informationen übermitteln können. Abbildung 2.1 beispielsweise stellt die Blutversorgung des Herzens dar. Da hier der Schwerpunkt auf den koronalen Venen liegt, sind diese blau eingefärbt. Das Herz hingegen gibt nur die Kontextinformation und ist mit *Stippling*, Silhouetten und Merkmalslinien dargestellt. Simplifizierte Darstellungen sind optimal für schnelle Informationsverarbeitung.

Bei verdeckten Strukturen wird durch ihre Freilegung ein optimaler Blickwinkel gewährleistet (siehe Abbildung 2.1 (b)). Der Künstler erreicht durch das Weglassen irrelevanter Strukturen eine immer präzisere Darstellung [Koschatzky, 1993, S.257]. Weiterhin werden komplexe Strukturen zum besseren Verständnis in den Zeichnungen vereinfacht dargestellt. „*Für die Wahrnehmung gilt die Grundregel der größten Einfachheit. Jene Bedeutung wird zuerst wahrgenommen, die am einfachsten erkannt wird.*“ [Koschatzky, 1993, S. 257]. Eine selektive Darstellung gibt laut Winkenbach und Salesin [1994] Illustrationen eine höhere Aussagekraft als den Photographien. Techniken wie Schraffur, Sihouetten, Merkmalslinien und *Stippling* gehören zu den *Pen-and-Ink*-Methoden, die traditionell für medizinische und wissenschaftliche Illustrationen genutzt werden. Illustratoren haben mit diesen handgefertigten Zeichnungen eine Möglichkeit geschaffen, einfach und effizient möglichst viele Informationen zu vermitteln.



(a) Quelle: Rogers [1992] (b) Quelle: Bertolini et al. [1995]

Abbildung 2.1: (a) In dieser Illustration wurden Farbe, *Stippling*, Silhouetten und Merkmalslinien verwendet, um die Blutversorgung des Herzens darzustellen. (b) Der Künstler ermöglicht in dieser Illustration einen optimalen Einblick in den Brustsitus mit Herzbeutel und Thymus durch die Freilegung der relevanten Strukturen.

Tietjen [2004] erwähnt in seiner Arbeit, dass die Schraffur meist für Muskeln, Sehnen oder Bändern eingesetzt wird, um hier zusätzliche Informationen über Kontraktions- beziehungsweise Dehnungsrichtung zu vermitteln. *Stippling* dagegen ist für Strukturen geeignet, die eine Krümmung aber keine spezifische Richtung aufweisen. Die Objekte sind meist durch Silhouetten voneinander getrennt und lassen sich unter anderem anhand von Schraffur und *Stippling* voneinander differenzieren.

In der Computergraphik werden die *Pen-and-Ink*-Abbildungen mit Hilfe von NPR-Techniken nachgeahmt, um Sachverhalte für den Betrachter verständlicher darzustellen und Kontextinformationen dezent in der Darstellung zu integrieren.

Da Mediziner bereits mit diesen Illustrationen vertraut sind, bietet es sich an, diese auch für die Visualisierung von medizinischen Oberflächenmodellen zu verwenden.

2.1.2 Klassische NPR-Techniken

Ein Teilgebiet der nicht-photorealistischen Computergraphik beschäftigt sich mit der Nachahmung von Zeichnungen oder Bildern, die traditionell von Hand erstellt werden. Hierzu zählen beispielsweise *Pen-and-Ink*-Illustrationen, Kohlezeichnungen und mit Wasser- oder Ölfarben gemalte Bilder [Akenine-Moller et al., 2002]. In dieser Arbeit werden ausschließlich Techniken zur Generierung von *Pen-and-Ink*-Illustrationen betrachtet. Linienhafte Darstellungen und *Stippling* sind klassische NPR-Methoden, die Künstler zur Erstellung dieser Illustrationen einsetzen. Silhouetten und Merkmalslinien zählen zu den linienhaften Darstellungen, mit deren Hilfe abgebildete Objekte grob skizziert und umrissen werden können. Die Schraffur wird ebenso wie das *Stippling* zur Abbildung zusätzlicher Informationen über Textur, Volumen, Transparenz oder zur Steigerung des Realismus in der Szene eingesetzt. Weiterhin können durch eine Kombination der einzelnen Techniken die Illustrationen detailgetreuer gestaltet werden [Hodges, 1989].

Silhouetten und Merkmalslinien

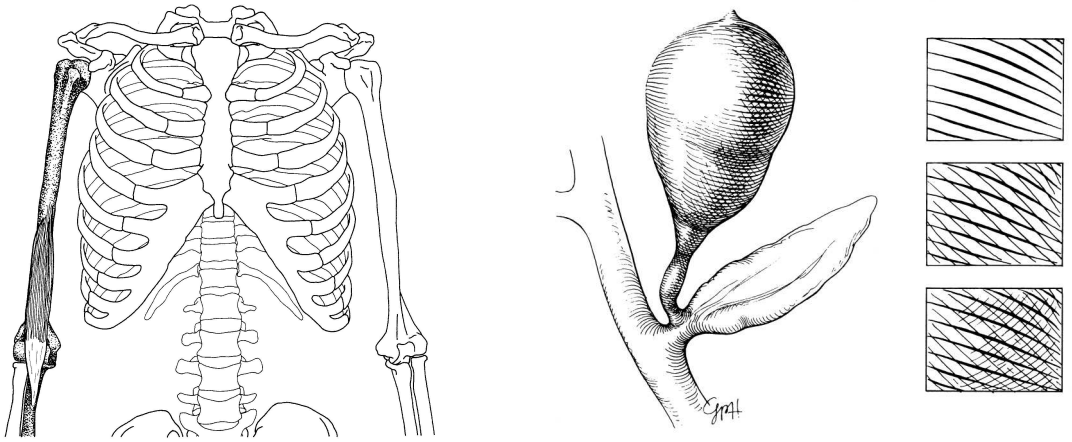
Eine Silhouette ist als eine Menge von Oberflächenpunkten definiert, für die gilt, dass ihr Normalen-Vektor senkrecht auf dem Vektor zum Betrachter steht. Basierend auf dieser Definition sind Silhouetten sichtabhängige Linienzüge, die den sichtbaren von dem unsichtbaren Bereich abtrennen. Sie gehören zu den wesentlichen Erkennungsmerkmalen eines Objektes [Pastor, 2003]. Konturen sind eine Untermenge der Silhouetten und beschreiben nur den Umriß eines Objektes. Die Konturlinie wird immer als zu dem Objekt gehörend betrachtet und grenzt es eindeutig vom Hintergrund ab [Koschatzky, 1993, S. 262].

Merkmalslinien dagegen sind sichtunabhängige Linien. Sie markieren charakteristische Merkmale der sichtbaren Oberfläche eines Objektes (siehe Abbildung 2.2 (a)). Diese Linien erlauben eine differenziertere Beschreibung und damit eine umfangreichere Darstellung [Koschatzky, 1993, S. 262]. Sousa und Prusinkiewicz [2003] geben Ränder, Falten, Vertiefungen und Ausbeulungen als verschiedene Typen von Merkmalslinien an, wobei:

Ränder alle Kanten sind, die nur an ein Polygon grenzen,

Falten und Kanten sich innerhalb der Kontur befinden und Bereiche starker Krümmung kennzeichnen und

Vertiefungen und Ausbeulungen Linien sind, die sich in konvexen und konkaven Oberflächenregionen befinden und diese verdeutlichen.



(a) Quelle: Stone und Stone [2000] (b) Quelle: Hodges [1989]

Abbildung 2.2: (a) Diese Illustration dient der Veranschaulichung des Oberarmmuskels. Während der betrachtete Arm durch *Stippling* und der dazugehörige Muskel mit Schraffur hervorgehoben ist, wurden Silhouetten und Merkmalslinien eingesetzt, um den Brustkorb und die anderen Oberarmknochen darzustellen. (b) Eine Kombination aus Eyelashing und Kreuzschraffur wird hier angewandt, um die Form und die unterschiedlichen Schattierungen zu repräsentieren.

Schraffur

Die Schraffur (*Hatching*) ist charakterisiert durch verschieden lange, äquidistante Linien beziehungsweise Striche, die unterschiedlich dichte Strichmuster formen. Eine Darstellung der Objekteigenschaften wie Farbe, Form, Material, Textur, der Beleuchtungseffekte und der lokalen Krümmung einer Oberfläche ist anhand dieser Linien möglich. Die Anzahl der Linien, deren Anordnung oder die Dicke der Linien können hierbei verschiedene Effekte für die Oberfläche erzeugen. Mit dieser Technik werden auch lokale Krümmungen der Objektoberfläche oder eine vorhandene Orientierung hervorgehoben. Spezielle Schraffur-Techniken sind die:

Kreuzschraffur(Crosshatching) , bei der sich die parallelen Linien kreuzen und somit das Schraffur-Muster erzeugen und das

Eyelashing , bei dem die Linienstärke variiert wird in Abhängigkeit von der Stärke der Beleuchtung oder der Krümmung der Oberfläche. Die Linien sind in der Mitte dicker als an den Enden (siehe Abbildung 2.2 (b)).

Stippling

Stippling (Punktiertechnik) ist eine sehr aufwändige aber vielseitig einsetzbare Technik, bei der die Objektform und -farbe allein durch Punkte dargestellt wird. Illustratoren haben diese Technik entwickelt um den Farbton eines Objektes mittels gleichmäßig verteilter Punkte zu repräsentieren, wodurch sehr klare Illustrationen erreicht werden [Secord, 2002].

Traditionelle *Stippling*-Zeichnungen erstellt ein Künstler per Hand, indem er die Punkte lokal setzt. Dabei erfolgt eine ständige Kontrolle der Punktverteilung, um makroskopische Muster zu vermeiden. Die Punkte müssen demnach sorgfältig platziert werden, damit keine ungewollten Muster entstehen und die Punkte dennoch zufällig auf dem Bild verteilt sind [Deussen et al., 2000]. Wie in Abbildung 2.3 dargestellt, werden durch Variationen von Anzahl und Dichte der Punkte Helligkeitsabstufungen repräsentiert.

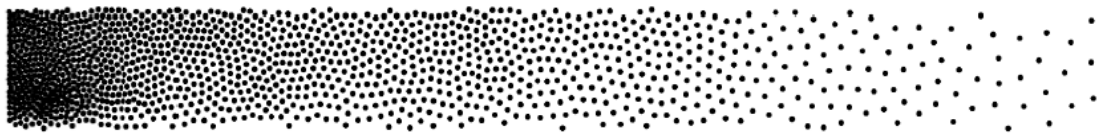


Abbildung 2.3: Darstellung verschiedener Helligkeiten anhand von Punkten, wobei die Anzahl und die Dichte der Punkte variiert wird, um die verschiedenen Graustufen darzustellen. Quelle: Strothotte und Schlechtweg [2002]

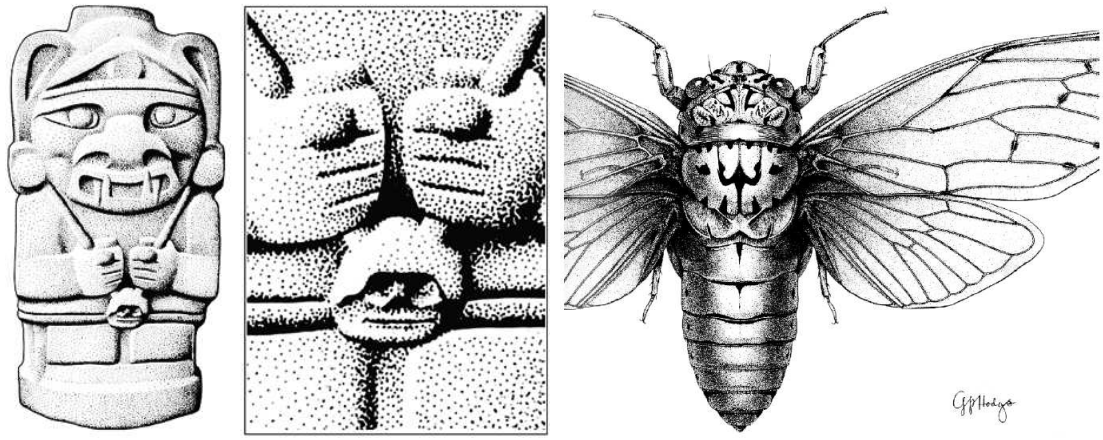
Laut Deussen et al. [2000] existieren bei dem traditionellen *Stippling* folgende Möglichkeiten, um individuelle Effekte zu erzeugen:

- **Abstand der Punkte**
  Alle Punkte sind zufällig verteilt aber nahezu mit äquidistanten Abständen zueinander angeordnet. Enger beieinander liegende Punkte repräsentieren dunklere Bereiche und größere Abstände zwischen den Punkten werden zur Darstellung von hellen Regionen genutzt (siehe Abbildung 2.4(a)). Es existieren aber auch Illustrationen, bei denen die Punkte entlang einer Hauptrichtung des Objektes orientiert sind oder Muster aufweisen, um spezielle Materialeigenschaften oder Texturen darzustellen.

- **Punktgröße**
  Im Allgemeinen ist die Punktgröße für eine Darstellung konstant. Verschiedene Größen können aber auch für hellere und dunklere Bereiche oder für unterschiedliche Materialien genutzt werden. Große Punkte vermitteln eine harte, womöglich starre Oberfläche, wobei kleine, eng nebeneinander platzierte Punkte eher einen weichen Eindruck der Oberfläche erzeugen.

- **inverse Illustrationen**
  Für eine Darstellung von Bildern mit sehr dunklen Regionen, bietet es sich an, die herkömmliche Darstellung von schwarzen Punkten auf weißem Hintergrund zu invertieren.

Mit diesen Möglichkeiten können verschiedene Schattierungen, Farbe, Objektformen, Oberflächeneigenschaften, Muster und Texturen dargestellt werden. *Stippling* bedeutet mehr als nur gleichmäßig verteilte Punkte auf einem Bild zu platzieren. Durch konkreten Einsatz der oben genannten Eigenschaften können Künstler neben Schattierungen und Objektform auch Objektgrenzen betonen und Details hervorheben (siehe Abbildung 2.4). Die Punkte werden entlang von Objektgrenzen, Silhouetten oder Merkmalslinien positioniert (siehe Abbildung 2.4 (b)). Ferner können noch Linien hinzugefügt werden, an denen sich die Punkte orientieren [Deussen et al., 2000].



(a) (b)

Abbildung 2.4: Zwei hangezeichnete *Stippling*-Darstellungen. (a) Der Götze von George Robert Lewis wurde mit Punkten und einer Kontur gezeichnet. Der vergrößert Ausschnitt zeigt den gleichmäßigen Abstand der Punkte. (b) In dieser Abbildung existieren zusätzlich Linien, an denen sich die Punkte orientieren. Hierdurch wird der Verlauf und die Form der Flügel und der Adern hervorgehoben. Quelle: Hodges [1989]

Stippling wird vor allem in wissenschaftlichen Illustrationen und für künstlerische Darstellungen eingesetzt [Pastor, 2003]. Laut Deussen et al. [2000] sind fast 80 % aller Illustrationen in Archäologie Büchern *Stippling*-Darstellungen. Diese Technik eignet sich sehr gut um glatte, gleichmäßig runde Objekte ohne scharfe Ecken darzustellen, wie sie oft bei medizinischen und archäologischen Sachverhalten vorkommen (siehe Abbildung 2.4 (b)).

Bei der Generierung derartiger Illustrationen sind einige Aspekte zu beachten. Kleinste Veränderungen der Dichte beispielsweise führen bereits zu dem Eindruck

von unterschiedlich hellen Regionen. Ferner müssen Berührungen der Punkte verhindert werden, da diese als Artefakte erscheinen. Hierbei ausgenommen sind sehr dunkle Bereiche, in denen sich solche Berührungen meist nicht vermeiden lassen [Deussen et al., 2000].

2.1.3 Medizinische Volumenvisualisierung

In der Medizin sind Volumendaten das Ergebnis von bildgebenden Verfahren, wie Computertomographie (CT) oder Magnetresonanztomographie (MRT). Die von diesen tomographischen Systemen gesammelten Daten liegen in Form von zweidimensionalen Schichtbildern mit kontanter Größe und konstantem Abstand vor. Eine Beurteilung räumlicher Strukturen und Beziehungen anhand einzelner Bilder ist sehr schwierig. Da 3D-Darstellungen der Daten für den Menschen besser beurteilbar sind, werden die Schichtbilder mittels entsprechender Verfahren in Volumina umgewandelt und visualisiert.

Volumen-Rendering ermöglicht die Darstellung aller drei Dimensionen und die Betrachtung der Daten als dreidimensionale Computergraphik-Objekte [Watt, 2002]. Das direkte und das indirekte Volumen-Rendering sind zwei große Verfahren, die sich für die Volumenvisualisierung herausgebildet haben. Ferner existieren Entwicklungen zur Integrierung von NPR-Techniken in die bisher bekannten Verfahren.

Direktes Volumen-Rendering

Das direkte Volumen-Rendering Verfahren ermöglicht eine detailgetreue Darstellung der Originaldaten ohne vorherige Segmentierung. Mit diesem Verfahren werden Volumeninformationen dargestellt. Für die Darstellung auf der Bildebene werden die Voxel unterschiedlich gewichtet. Der Farbwert eines Pixels ergibt sich dann durch die Überlagerung semitransparenter Voxel entlang eines betrachteten Sichtstrahles. Mit Transferfunktionen werden hierfür die Sichtbarkeit und die Färbung der abzubildenden Daten bestimmt. Neben Transferfunktionen, die nur gewisse Intensitätswerte auf Grauwerte abbilden, existieren auch Methoden, bei denen nur ein Voxel eines Sichtstrahles auf die Bildebene projeziert wird. Dieser besitzt entweder die maximale oder die minimale Intensität. Die Methoden werden unterschieden in *Maximum Intensity Projection* (MIP) und *Minimum Intensity Projection* (mIP). Das klassische Volumen-Rendering simuliert die Abschwächung eines Lichtstrahles durch ein Objekt und entspricht damit der natürlichen Erscheinung von durchsichtigen Objekten [Levoy, 1988].

Indirektes Volumen-Rendering (Iso-Surface-Rendering)

Indirekte Verfahren werden zur Extraktion von Oberflächen (Iso-Oberflächen) aus 3D-Volumendaten verwendet. Sie basieren meist auf dem *Marching Cubes* Algorithmus von Lorensen und Cline [1987]. Ein solcher Algorithmus betrachtet das Voxelgitter zunächst als einzelne, unabhängige Würfel, um mit deren Hilfe ein Dreiecksnetz zu bestimmen. Ein Schwellwert bestimmt die Würfel und deren Kanten, die zur Extraktion der Polygone beitragen. Mittels linearer Interpolation wird dann der gesuchte Intensitätswert auf den Kanten der entsprechenden Würfel lokalisiert. Anschließend erfolgt die Verbindung dieser Schnittpunkte und die Generierung des Dreiecksnetzes. Es existieren mittlerweile Methoden, wie die von Gerstner [2001], die mehrfache transparente Iso-Oberflächen in Echtzeit generieren können.

Da das indirekte Volumen-Rendering auf die Extraktion von Oberflächeninformationen reduziert ist, entfallen viele in den Daten enthaltene anatomische Informationen und die Strukturen können von ihrer eigentlichen Struktur bedeutend abweichen [Gerstner, 2001]. Das Verfahren arbeitet direkt auf den Volumendaten ohne vorherige Segmentierung, wodurch die Exaktheit der dargestellten Oberfläche täuschen kann. Ein weiteres Problem sind verrauschte Daten, wodurch es zur Trennung von eigentlich zusammengehörigen Strukturen oder zum Zusammenschmelzen von disjunkten Strukturen kommen kann.

NPR-Volumen-Rendering

Neben den herkömmlichen Volumen-Rendering Verfahren existieren Methoden, die einige aus Abschnitt 2.1.2 vorgestellte NPR-Techniken direkt auf den Volumendaten oder auf den aus diesen Daten gewonnenen Oberflächenmodellen ausführen. Die in Abschnitt 2.1.1 erläuterten Vorteile von NPR-Techniken in medizinischen Illustrationen können mit dieser Technik auch für die Volumenvisualisierung ausgenutzt werden.

Interrante et al. [1995] verwenden Linien zur besseren Illustration der Oberflächenformen in einem Volumendatensatz. Diese Linien sind in Falten und auf Ausbeulungen (*Ridge and Valley Lines*) von semitransparenten Oberflächen platziert, wodurch deren Form und Position deutlicher erkennbar ist. Ebert und Rheingans [2000] stellten einen Ansatz vor, bei dem NPR-Techniken mit Volumen-Rendering kombiniert werden, um Grenzen, Silhouetten und Orientierungen der Objekte zu betonen. Dong et al. [2003] entwickelten eine Schraffur-Methode für Oberflächen von Volumendaten. Sie konzentrieren sich auf die Darstellung von Muskeln, Eingeweiden und Gehirn. Hierbei wurde die Orientierung der Linien genutzt, um zusätzliche Informationen über die Strukturen abzubilden. Yuan und

Chen [2004] kombinieren alle Varianten des Volumen-Renderings. Zuerst werden die Oberflächen extrahiert, um anschließend ein direktes Volumen-Rendering durchzuführen und die Schraffur-Techniken zu integrieren. Das Rendern von Oberflächen und Volumen erfolgt separat, um die Vorteile beider Verfahren optimal auszunutzen.

Das Problem bei der Integration von NPR-Primitiven in eine Volumenvisualisierung ist die Menge der dargestellten Informationen. Hierbei kommt es leicht zu Überdeckungen. Es besteht die Gefahr einer überladenen Darstellung, die den Betrachter eher verwirrt, als das sie ihm zusätzliche Informationen übermittelt.

2.2 Stippling-Ansätze

In der Computergraphik existieren verschiedene Ansätze für die Visualisierung von Objekten über *Stippling*. Bei der Generierung einer derartigen Darstellung müssen verschiedene Charakteristika beachtet und integriert werden. Diese setzen sich einerseits aus den wesentlichen Kriterien für eine *Stippling*-Illustration und andererseits aus den Anforderungen an eine Visualisierung zusammen. Die folgenden Konzepte müssen für die Anwendung von 2D-Zeichenstilen auf 3D-Modelle beachtet werden:

- **Eigenschaften der Punkte**
  Die Punkte in einer *Stippling*-Darstellung müssen wie bereits in Abschnitt 2.1.2 erläutert, zufällig auf dem Objekt und mit möglichst äquidistantem Abstand zueinander verteilt werden.

- ***Frame*-Kohärenz**
  Für NPR-Animationen existiert eine Kohärenz auf Objekt-Ebene und eine auf Partikel-Ebene. Die Beibehaltung der Form von Objekten und ihrer Darstellung in einer Sequenz aufeinander folgender Bilder wird als *Frame*-Kohärenz der Objekte bezeichnet. In einer Szene müssen sich demnach die Objekte und ihre Eigenschaften (z.B. *Shading*) sanft über die *Frames* verändern, um den Erwartungen des Betrachters zu entsprechen [Pastor, 2003].

  Eine Kohärenz der Partikel verhindert die Entstehung von Rauschen oder störenden Artefakten während der Animation. Partikel sind auf der Objektoberfläche ansässige Punkte, die mit ihrer Position mögliche Orte für die NPR-Primitive angeben, welche die Farbe und Form des Objektes repräsentieren [Pastor, 2003]. Beim *Stippling* sind diese Primitive die Punkte.

- **Skalierbarkeit**
  Es besteht ein Unterschied zwischen der Skalierung von 2D-Illustrationen und 3D-Modellen. Die NPR-Primitive auf der Oberfläche können nicht gleichermaßen wie das Objekt vergrößert oder verkleinert werden, da ansonsten aus den Punkten der *Stippling*-Modelle entweder große Kreise oder winzige Punkte entstehen. Die Skalierung von Objekten mit NPR-Primitiven wird durch eine Vervielfachung oder Verminderung der Anzahl dieser Primitive auf dem Objekt erreicht.

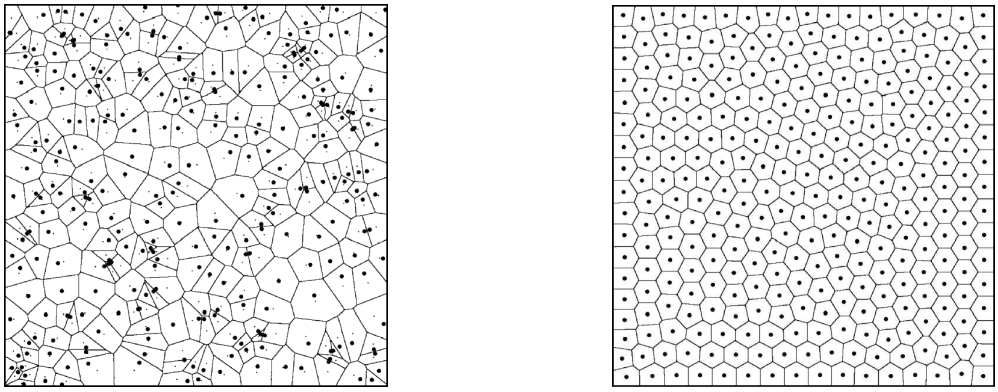
Im Folgenden werden nun die möglichen Verfahren der computergestützten Generierung von *Stippling*-Darstellungen vorgestellt, wobei diese unter den eben eingeführten Kriterien betrachtet werden.

2.2.1 Bildbasiertes Verfahren

Techniken, die *Stippling*-Darstellungen anhand von Graustufenbildern der Objekte, wie beispielsweise eingescannte Zeichnungen, Photos oder einzelne Bilder einer Animation generieren, werden bildbasierte Verfahren genannt. Eine zufällige Menge von Punkten wird zunächst ausgehend von den Graustufenbildern erzeugt und später mittels verschiedener Methoden gleichmäßig verteilt.

In dem Verfahren von Deussen et al. [2000] wird die Eingangsmenge der Punkte entweder von einem Benutzer interaktiv festgelegt oder mittels eines Referenzbildes und der Halbton-Methode (*Halftoning*) automatisch berechnet. Die regelmäßige Verteilung der Punkte (Relaxation) erfolgt durch die Implementierung des Algorithmus von Lloyd [1982], bei dem ein Voronoi-Diagramm aus den Eingangspunkten erstellt wird. Wie in Abbildung 2.5 (a) dargestellt, erfolgt eine Einteilung des gesamten Bildes in Voronoi-Regionen. Alle Punkte werden dann in den Schwerpunkt ihrer Region verschoben (siehe Abbildung 2.5 (b)). Die Iteration der Lloyd [1982] Methode erzeugt äquidistante Abstände zwischen benachbarten Punkten.

Der Prozess der Relaxation versucht die anfangs unregelmäßig angeordneten Punkte gleichmäßig zu verteilen. Da das zugrunde liegende Graustufenbild des darzustellenden Objektes keinen Einfluss auf die Relaxation der Punkte hat, entstehen Verwischungen und Unschärfe an den Kanten. Die wichtigsten Kanten bleiben jedoch durch eine von Deussen et al. [2000] integrierte Segmentierung des Bildes erhalten. Hierbei werden die unterschiedlichen Grauwertbereiche bestimmt, für die jeweils eine vordefinierte *Stippling*-Anzahl verwendet werden soll. Weiterhin bietet dieses Verfahren dem Benutzer die Möglichkeit weitere Punkte hinzuzufügen oder zu löschen um bestimmte Details besser hervorzuheben.



(a) (b)

Abbildung 2.5: (a) Ein Voronoi-Diagramm für eine zufällig verteilte Punktmenge. Die dicken Punkte kennzeichnen hierbei die Punkte, die die Unterteilung erzeugen und die dünnen den Schwerpunkt jeder Region. (b) Nach der Umordnung der Punkte an die Stelle des Schwerpunktes ihrer Region und Neuberechnung der Regionsgrenzen entsteht eine gleichmäßige Punktverteilung. Quelle: Secord [2002]

Secord [2002] entwickelte ein weiteres bildbasiertes Verfahren, welches auf dem Ansatz von Deussen et al. [2000] aufbaut. Bei diesem Verfahren existiert in Abhängigkeit vom Graustufenbild eine Einschränkung der Punktbewegung während der Relaxation. Die Methode von Secord [2002] realisiert die automatische Generierung einer *Stippling*-Zeichnung aus einem gegebenen Graustufenbild mit verbesserter Kantendarstellung.

Bildbasierte Verfahren sind nicht für die Erstellung von Animationen geeignet, da sie auf einzelnen Bildern arbeiten und auch nur jeweils ein Bild erzeugen. Die Punkte werden unabhängig von ihrer Position im vorhergehenden Bild gesetzt. Folglich kann es zu einer Änderung ihrer Position während der einzelnen *Frames* einer Animation kommen, ohne dass sich das Objekt ändert. Es entsteht ein Rauschen auf der Objektoberfläche oder der Eindruck einer nicht vorhandenen Objektbewegung wird erzeugt. Die Punkte wandern in einer Animation über das Objekt. Dieser Effekt wird auch als *Shower Door*-Effekt bezeichnet. Bildbasierte Verfahren garantieren folglich keine *Frame*-Kohärenz auf Partikel-Ebene. Die Eingangs erwähnte Verteilung der Punkte wird mittels der erwähnten Voronoi-Diagramme und dem Algorithmus von Lloyd [1982] erfüllt. Die Größe der Punkte ist auch bei einer Skalierung der Objekte gewährleistet, da für jede Entfernung des Objektes zum Betrachter ein neues Bild mit einheitlicher Größe der Punkte generiert wird, dadurch hat eine Skalierung keinen Einfluss auf die Punktgröße.

2.2.2 Objektbasiertes Verfahren

Objektbasierte Verfahren generieren aus 3D-Modellen *Stippling*-Darstellungen, wobei die Punkte direkt mit der Objektoberfläche verbunden sind. Pastor [2003] beschäftigte sich in seiner Promotion mit der *Stippling*-Darstellung von 3D-Modellen. In dieser Arbeit geht es vor allem um die Generierung von Animationen.

Pastor [2003] führt Punkthierarchien ein, um *Frame*-Kohärenz und Skalierbarkeit des Objektes zu garantieren und folglich eine Animation statischer und dynamischer Modelle zu ermöglichen. Die Punkte sind mit den Eckpunkten der Polygone verbunden und eine Unterteilung oder Vereinfachung des Polygonnetzes erzeugt die benötigte Anzahl an *Stippling*-Punkten. Nach jeder Veränderung des Objektgitters erfolgt die Relaxation der Eckpunkte mittels Voronoi-Diagrammen bis alle regelmäßig verteilt sind.

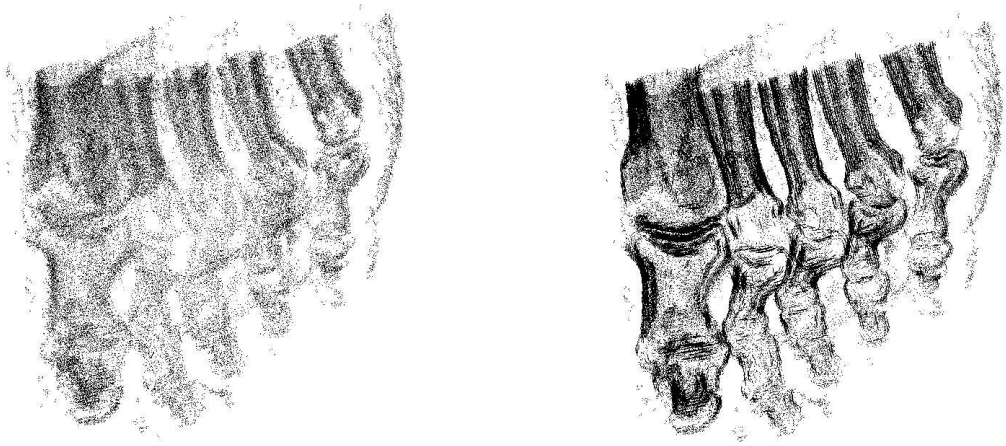
Die Hierarchie-Ebenen repräsentieren jeweils eine bestimmte Menge an *Stippling*-Punkten und damit einen bestimmten Farbton des Objektes. Der gewünschte Abstand zwischen benachbarten Punkten entscheidet, bis zu welchem Hierarchie-Level die Punkte gezeichnet werden. Weiterhin wird durch die verschiedenen Ebenen auch die Skalierbarkeit realisiert, indem die Anzahl der Punkte auf den Ebenen für unterschiedliche Objektgrößen verwendet wird. Bei der Skalierung erscheinen neue Punkte oder alte verschwinden.

Der bedeutende Vorteil von objektbasierten *Stippling*-Verfahren besteht in der *Frame*-Kohärenz. Die Punkte sind mit dem Objekt fest verbunden. Verändert das Objekt seine Position oder seine Form, verhalten sich die Punkte äquivalent zu dem Objekt. Daher entsteht kein Rauschen durch wandernde Punkte wie bei den bildbasierten Techniken. Es ist allerdings ein sehr aufwendiges Verfahren, da das Polygonnetz des Objektes vereinfacht oder unterteilt werden muss, um zusätzliche Eckpunkte zu erhalten. Eine Echtzeit-Visualisierung wird durch die zusätzlich zu bearbeitende Eckpunkte verlangsamt. Pastor [2003] gibt in seiner Arbeit an, dass der Teapot mit $2,256$ Polygonen inklusive jeglicher Operationen für den Aufbau der Punkthierarchie $41s$ und der Hase mit $69,459$ Polygonen $2min\ 31s$ benötigt.

Pastor et al. [2003] entwickelten auch einen hardwarebeschleunigten Algorithmus, basierend auf dem eben vorgestellten Konzept. Allerdings bleibt die Anzahl der dargestellten Punkte auf der Objektoberfläche weiterhin auf die Menge der vorhandenen Eckpunkte begrenzt. Demzufolge sind zeitaufwendige Polygonnetzmodifikationen zum Aufbau der Punkthierarchie und Generierung von weiteren Eckpunkten notwendig.

2.2.3 Stippling integriert im Volumen-Rendering

Lu et al. [2002] entwickelten ein direktes Volumen-Rendering Verfahren, mit dem wissenschaftliche oder medizinische Volumendaten über *Stippling* dargestellt werden (siehe Abbildung 2.6). Die erzeugten Bilder können laut Lu et al. [2002] für eine erste Erforschung oder eine Vorschau der Daten genutzt werden. Interessante Regionen werden mit dem Volumen-Rendering Verfahren bestimmt und hervorgehoben. Für eine genauere Betrachtung müssen dann andere Visualisierungstechniken gewählt werden.



(a) (b)

Abbildung 2.6: Der Volumendatensatz von einem Fuß, der mit dem Verfahren von Lu et al. [2002] dargestellt wurde. (a) Die Visualisierung des Datensatzes allein über *Stippling* und (b) als eine Kombination von *Stippling* entlang der Silhouetten. Quelle: Lu et al. [2002]

In dem Verfahren von Lu et al. [2002] werden die Merkmale anhand der Größe und der Richtung des Gradienten bestimmt. Die Regionen, die einen hohen Gradienten aufweisen, können über *Stippling* visualisiert werden. Hierbei handelt es sich vorwiegend um die Grenzregionen einer Struktur. In den von Lu et al. [2002] veröffentlichten Bildern sind nur Gebiete mit hohem Gradienten abgebildet. Dazu zählen die Grenzen zwischen Luft und Haut oder zwischen Gewebe und Knochen (siehe Abbildung 2.6). Grenzen zwischen Weichteilen dagegen können mit dieser Technik nicht oder nur unzureichend visualisiert werden, da dort der Gradient zu schwach ist. Ferner ist in verrauschten Volumendaten die exakte Gradientenberechnung problematisch, weswegen diese Technik hauptsächlich für CT-Datensätze verwendet wird [Lu et al., 2002].

In einem Vorverarbeitungschritt wird die maximale Anzahl an Punkten berechnet und gleichmäßig verteilt. Je nach Entfernung und Sichtbarkeit vom Betrachter wird dann die benötigte Anzahl an Punkten pro Voxel ausgewählt und dargestellt,

um die Helligkeitsinformationen sowie die Skalierung korrekt zu repräsentieren. Lu et al. [2002] gewährleisten auf diese Weise mit dem Verfahren die benötigte *Frame*-Kohärenz. Sie erweiterten das Verfahren, indem sie die *Stippling*-Punkte verstärkt zur Betonung von Silhouetten eingesetzten, wie in Abbildung 2.6 (b) dargestellt ist.

2.2.4 Stippling mit Texturen

Abschließend wird hier ein Ansatz betrachtet, der die NPR-Darstellung von Objekten mit Texturen realisiert, wobei auf die genaue Definition einer Textur und Textur-Abbildungen später in Abschnitt 2.3 eingegangen wird. Salisbury et al. [1994] erstellten in ihrer Arbeit neben Schraffur- auch *Stippling*-Illustrationen. Basierend auf 2D-Graustufenbildern werden diese mit Hilfe von Texturen erzeugt. Für Schraffur-Illustrationen entwickelten sie Texturen (*Prioritized Stroke Textures*), die verschiedene Graustufen repräsentieren. Hierbei werden nur die bedeutendsten Linien für helle Töne gesetzt und die weniger wichtigen für dunklere hinzugefügt. Der Benutzer kann interaktiv mit Hilfe der einzelnen Texturen die gewünschten Farbtöne erzeugen. Während die Methode von Salisbury et al. [1994] nur für 2D-Illustrationen konstruiert wurde, verwendeten Winkenbach und Salesin [1994] die gleiche Technik für 3D-Modellen. Es werden verschiedene Texturen generiert, um unterschiedliche Helligkeiten und Skalierungen des Objektes darstellen zu können. Die Techniken von Salisbury et al. [1994] und Winkenbach und Salesin [1994] erstellen nur einzelne Bilder und sind nicht für Echtzeit-Rendering konzipiert worden.

Lake et al. [2000] führten ein Echtzeit-Verfahren ein, bei dem je nach Helligkeit eines Eckpunktes aus einer Menge von Texturen die entsprechende ausgewählt wird. Falls notwendig werden hierbei die Polygone weiter unterteilt, was zu einem hohen Berechnungsaufwand führt. Praun et al. [2001] hingegen beschreiben wie durch eine Vorberechnung der *Tonal Art Maps* (TAM) die *Prioritized Stroke Textures* in Echtzeit erstellt werden können. TAMs sind spezielle Texturen, die Kohärenz zwischen verschiedenen Helligkeits- und Auflösungstufen besitzen. Diese bauen auf den *Art Maps* von Klein et al. [2000] auf und sind ferner eine Erweiterung der *Mip Maps* von Williams [1983]. Während *Art Maps* variierende Helligkeiten in einem Bild vereinen, bestehen TAMs aus einer Serie von *Art Maps*, in der jedes einzelne Bild einem Farbton entspricht. Klein et al. [2000] und Praun et al. [2001] veranschaulichen die TAMs anhand von Schraffuren, wobei die Dichte und Anzahl der Linien den Farbton definieren.

In Abbildung 2.7 sind horizontal die verschiedenen Helligkeitstufen und vertikal die unterschiedlichen Auflösungsstufen und damit das MIP-*Mapping* für eine Textur abgebildet. MIP steht als Abkürzung für das lateinische *multum in parvo*, was

soviel bedeutet wie „Viel auf kleinem Platz“ [Williams, 1983]. Beim *Texture Mapping* ist es problematisch, dass Objekte von sehr nah und von sehr weit entfernt betrachtet werden können. Es müssen demzufolge verschiedene Texturen vorliegen, die entsprechend der Skalierung des Objektes verwendet werden können. Durch Halbierung der Kantenlänge einer Ausgangstextur entstehen Texturen mit abnehmender Auflösung.

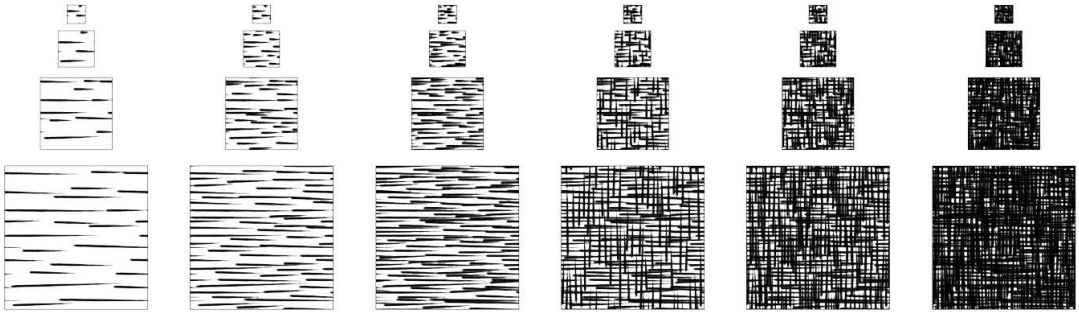


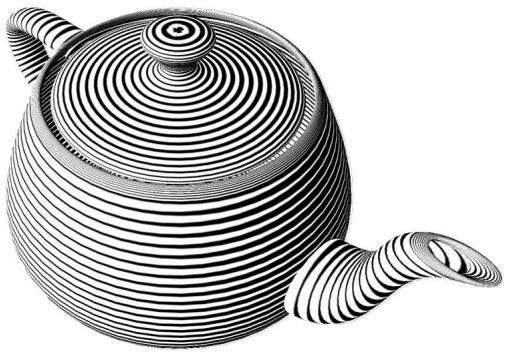
Abbildung 2.7: *Tonal Art Map* nach dem Modell von Praun et al. [2001]. Horizontal werden Helligkeits- und vertikal Auflösungsstufen einer Schraffur-Textur dargestellt. Quelle: Praun et al. [2001]

Das Konzept der TAMs gewährleistet die Kohärenz des Farbtones und der Auflösung zwischen den einzelnen Texturen. Alle Linien einer helleren Textur sind immer Teilmenge der dunkleren Textur. Diese Kohärenz verhindert störende Artefakte (Flackern) beim Überblenden der einzelnen Texturen. Ferner ist eine Textur mit geringerer Auflösung auch immer Teilmenge der Textur mit höherer Auflösung. Es werden folglich nur neue Linien eingeblendet oder vorher angezeigte verschwinden. Zur Texturierung eines Objektes werden in dem Verfahren von Praun et al. [2001] pro Eckpunkt zwei Texturen aus der TAM ausgewählt. Die Texturen der Eckpunkte werden dann über das jeweilige Polygon ineinander übergeblendet. Die Verwendung mehrerer Texturen pro Polygon erzeugt visuelle Kontinuität in interaktiven Systemen.

Im Gegensatz zu Praun et al. [2001], bei dem die Helligkeit durch die Dichte der Linien dargestellt wird, repräsentieren Freudenberg et al. [2002] diese anhand der Dicke der Linien (siehe Abbildung 2.8). Freudenberg et al. [2002] bauen auf der Halbton-Methode auf und ermöglichen eine Texturierung auf Pixelebene. Die Schraffur-Textur wird als Halbton-Muster kodiert, wobei die einzelnen Schraffur-Darstellungen als Layer in dieser Textur vereint sind. In den Texeln der Textur ist jeweils ein Grenzwert gespeichert, der angibt, wann ein Pixel seine Farbe ändern soll. Für jedes Pixel wird nun die von der Oberfläche reflektierende Intensität mit dem entsprechende Wert aus der Textur verglichen. Mittels einer Funktion wird ausgewertet, ob das Pixel weiß oder schwarz dargestellt wird. Im Gegensatz zur herkömmlichen Halbton-Methode verwenden Freudenberg et al.

[2002] hierfür eine lineare Funktion mit großem Anstieg. Auch Webb et al. [2002]

(a) Quelle: Praun et al. [2001]



(b) Quelle: Freudenberg et al. [2002]

Abbildung 2.8: (a) Ein Hand-Modell, das mit Hilfe von einer TAM über Schraffur dargestellt ist. (b) Der *Teapot* wurde auch mit Hilfe von Linien dargestellt, wobei hier die Dicke der Linien die Schattierung repräsentiert.

beschäftigten sich vorrangig mit Schraffur-Texturen und erweitern die Methode von Freudenberg et al. [2002], um verschiedene Helligkeiten durch Ein- und Ausblenden der Linien zu erhalten und Aliasing-Effekte zu vermeiden. Webb et al. [2002] verwenden die TAMs mit höherer Auflösung und führen Berechnungen pro Pixel aus, wodurch mehr Kontrolle über den dargestellten Farbton erreicht wird.

Die Verwendung von Texturen garantiert für aufeinander folgende Bilder *Frame*-Kohärenz, da die Texturen direkt mit dem Objekt verbunden sind. Bei der Generierung von *Stippling*-Texturen müssen entsprechende Verfahren eingesetzt werden, um die Verteilung der Punkte entsprechend der Anforderung an *Stippling*-Darstellungen zu erfüllen.

2.3 Texturen

Das Bestreben virtuelle Szenen möglichst realitätsnah zu gestalten, erfordert vor allem die detailgetreue Darstellung der darin enthaltenen Objekte. Meist werden die Objektoberflächen durch sehr kleine Strukturen charakterisiert. Die exakte Modellierung solcher Strukturen ist vor allem für eine Echtzeit-Darstellung zu aufwändig, da die einzelnen, geometrischen Details nachgebildet werden müssen und der Visualisierungsaufwand direkt von der Komplexität des zugrunde liegenden geometrischen Modells abhängt. Komplexe Eigenschaften einer Objektoberfläche werden deshalb meist mit Texturen realisiert. Die Oberflächeneigenschaften werden ersetzt und nicht modelliert, wodurch Modellier-, Speicher- und Laufzeit gespart

wird. Heckbert [1986, S.1] schrieb in seiner Arbeit[1]: "*Realism demands complexity, or at least the appearance of complexity.*" Objekte mit einfacher Geometrie können mittels Texturen aufwändige, komplexe Oberflächenstrukturen erhalten und damit die Szene realistischer erscheinen lassen. Da für die Abbildung von Texturen auf Oberflächen beliebige Muster oder Bilder verwendet werden können, ist es möglich jede Eigenschaft einer Oberfläche zu repräsentieren oder gezielt zu verändern.

Die Textur ist nach Heckbert [1986] ein mehrdimensionales Bild, dass in einen mehrdimensionalen Raum abgebildet wird. Es existiert eine Einteilung der Texturen in zwei Gruppen: zum einen die diskreten Texturen, die in Form von Bildern oder Mustern vorliegen und zum anderen die prozeduralen Texturen, welche durch prozedurale Funktionen erzeugt werden.

Diskrete Texturen In einer diskreten Textur sind diskrete Werte gespeichert, die Informationen über Intensität, Farbe oder andere Objekteigenschaften enthalten können. Der Vorteil liegt vor allem in der einfachen Erzeugung von komplexen, realitätsnahen Texturen. Digitalisierte Photos, Bilder, Zeichnungen oder computergenerierte Bilder gehören zur Gruppe der diskreten Texturen. Sie sind ohne größeren Aufwand beliebig auszutauschen und ermöglichen damit die Darstellung vieler verschiedener Oberflächeneigenschaften. Allerdings benötigen diskrete Texturen mit zunehmender Auflösung mehr Speicherplatz.

Prozedurale Texturen Diese Texturen werden durch prozedurale Funktionen erzeugt und liegen in Form eines Quellcodes vor. Bei jedem Zugriff auf ein Texturelement werden mathematische Funktionen oder Algorithmen ausgewertet [Ebert et al., 2003]. Folglich wird für jeden Punkt des Objektes der entsprechende Texturwert mit optimaler Genauigkeit berechnet. Im Gegenteil zu den Diskreten, benötigen prozedurale Texturen einen minimalen Speicherbedarf. Für eine Textur wird nur ein Quellcode abgespeichert, der die Berechnung jedes Texels definiert. Prozedurale Texturen repräsentieren nicht nur einen bestimmten Bereich des Objektes, sondern sind in ihrem Ausmaß unbegrenzt und ermöglichen die Texturierung eines Objektes ohne sichtbare Kanten oder Wiederholungen von Texturmustern [Ebert et al., 2003]. Der Entwurf und die Erstellung von prozeduralen Texturen ist allerdings mit großem Aufwand verbunden, da die benötigten Funktionen meist sehr komplex und selbst für erfahrene Entwickler schwierig zu implementieren sind. Ferner ist der Definitionsbereich der darstellbaren Texturmuster stark eingeschränkt. Es ist beispielsweise nicht möglich eine realistische Umgebung einfach als Textur zu erzeugen. Auch der Austausch von Texturen gestaltet sich schwieriger als

[1] Realismus erfordert Komplexität, oder zumindest den Anschein von Komplexität.

bei diskreten Texturen, da hier ein aufwändiger Programmcode verändert werden muss.

Laut Tönnies und Lemke [1994] werden Texturen auf drei verschiedene Arten repräsentiert. Entweder wird die Textur als Bild auf eine glatte Oberfläche abgebildet, durch Richtungsänderung (Perturbation) der Oberflächennormalen erzeugt oder als Modifikation durch einen 3D-Texturraum realisiert. Objekte lassen sich nicht nur „tapezieren“ oder mit Reliefs versehen, es können auch die Oberflächeneigenschaften wie Material, Farbe oder Transparenz an jedem Objektpunkt gesteuert werden. Es ist demzufolge möglich, mit Hilfe von Texturen beliebige Beleuchtungs-, Geometrie- und Materialeigenschaften eines Objektes zu simulieren.

Texture Mapping ist nach Heckbert [1986] die Abbildung einer Funktion auf eine Oberfläche im 3D-Raum. Der Definitionsbereich der Funktion kann dabei ein-, zwei- oder dreidimensional sein und entweder durch ein Feld oder eine mathematische Funktion repräsentiert werden. Jedem Punkt auf der Objektoberfläche wird somit ein Wert aus der Textur zugewiesen, der die neue Oberflächeneigenschaft repräsentiert.

2.3.1 Texture Mapping mit 2D-Texturen

2D-*Textur Mapping* ist die Projektion einer 2D-Textur auf ein virtuelles Objekt im 3D-Raum. Die Textur ist ein digitales 2D-Bild, dessen Grauwerte benutzt werden, um den diffusen Anteil der Reflexion zu modifizieren [Tönnies und Lemke, 1994]. Es wird von einer glatten Objektoberfläche ausgegangen, deren visuelle Struktur mit einer Textur verändert wird. Die einzelnen Elemente des Texturbildes heißen Texel. Ihre Koordinaten im Texturraum werden (s, t) Koordinaten genannt und liegen in dem Intervall [0,1].

Bevor die verschiedenen Arten der Textur-Abbildungen betrachtet werden, erfolgt zunächst die Definition der drei wichtigsten Begriffe:

- **Texturraum**
  Der Texturraum ist der Raum, in dem die Textur definiert ist. Bei einer 2D-Textur entspricht dies einer Ebene.
- **Objektraum**
  Im 3D-Objektraum werden die Körper modelliert, die später mit den Texturen versehen werden.
- **Bildraum**
  Der Bildraum beinhaltet die 2D-Szene, die auf dem Bildschirm dargestellt wird.

Die Abbildung 2.9 zeigt, dass das *Texture Mapping* konzeptionell ein zweistufiger Prozess und lässt sich in die aufeinander folgenden Abbildungen, Parametrisierung und Projektionstransformation unterteilen [Watt, 2002]. Als erstes wird eine 2D-

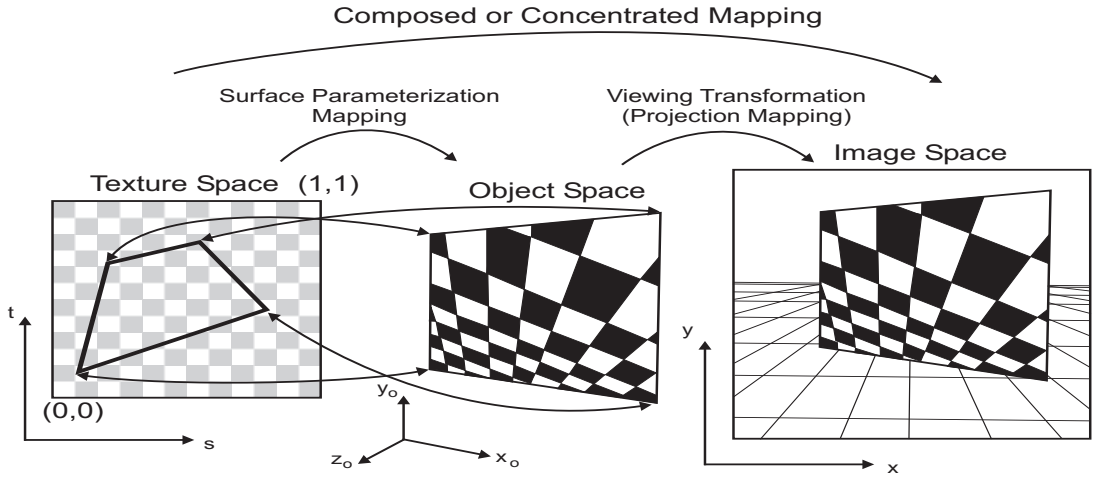


Abbildung 2.9: *Texture Mapping* ist ein zweistufiger Prozess bei dem zuerst vom Texturraum in den Objektraum und anschließend in den Bildraum abgebildet wird. Quelle: Lansdale [1991]

Textur auf die Oberfläche eines Objektes aufgebracht und somit in den Objektraum abgebildet. Diese Transformation stellt die Parametrisierung dar. Jedem Punkt auf der Objektoberfläche wird ein Texel der Texturebene zugeordnet. Hierbei ist es wichtig eine geeignete Funktion F zu finden, die jedem Punkt auf dem Objekt einen Wert zuweist: $(s,t) = F(x,y,z)$, wobei (x,y,z) die Koordinaten der Objektpunkte und (s,t) die Koordinaten der einzelnen Texturpunkte im Texturraum sind. Die Minimierung der Texturverzerrung auf dem Objekt ist ein wichtiges Kriterium zur Bestimmung dieser Abbildungsfunktion. Im Abschnitt 2.3.2 wird auf die Problematik der Texturverzerrung noch ausführlich eingegangen. Die zweite Abbildung ist dann die Projektion des Objektes aus dem 3D-Objektraum in den 2D-Bildraum.

Da das *Texture Mapping* während der Visualisierung erfolgt und zu diesem Zeitpunkt bereits feststeht, welcher Objektteil auf ein Pixel abgebildet wird, invertiert man die Abbildungsvorschrift [Tönnies und Lemke, 1994]. Ausgehend vom Bildraum muss für jedes Pixel zuerst die Fläche im Objektraum bestimmt werden, die auf das jeweilige Pixel abgebildet wird. Anhand der nun bekannten 3D-Koordinaten wird der zugehörige Ausschnitt der Textur im Texturraum festgelegt. Ein Pixel wird oft von mehren Texeln bedeckt, welche alle zur Pixelfarbberechnung mit einfließen [Foley et al., 1996].

Um ein möglichst gutes Ergebnis mit geringen Texturverzerrungen oder Artefakten zu erhalten, ist es beim *Texture Mapping* wichtig, eine gute Parametrisierung

des Objektes durchzuführen. Für Objekte wie Kugel, Zylinder oder Würfel ist es mittels einfacher mathematischer Formeln leicht eine Abbildungsfunktion zu definieren. Wenn die Geometrie der Objektoberfläche jedoch komplexer ist, wird die Zwei-Phasen-Texturprojektion angewandt. Hierbei wird das *Texture Mapping* in zwei Schritte unterteilt. Der folgende Abschnitt erläutert diese Texturprojektion, bei der die Textur zuerst auf ein Zwischenobjekt und ausgehend davon auf das eigentliche Objekt abgebildet wird.

Zwei-Phasen-Texturprojektion

Die Zwei-Phasen-Texturprojektion wurde von Bier und Sloan [1986] eingeführt und beschreibt die Abbildung von 2D-Texturen auf Objektoberflächen mit komplexer Geometrie. Wie bereits erwähnt wird hierfür die Textur zuerst auf die Oberfläche eines Zwischenobjektes abgebildet. Dieses Hilfsobjekt ist im 3D-Raum definiert und umgibt das zu texturierende Objekt. Zylinder, Würfel oder Kugel sind beispielsweise gut geeignet, da die jeweilige benötigte Abbildungsfunktion mathematisch einfach zu bestimmen ist. Der zweite Schritt umfasst die Abbildung von dem Zwischenobjekt auf das Zielobjekt, wie in Abbildung 2.10 dargestellt.

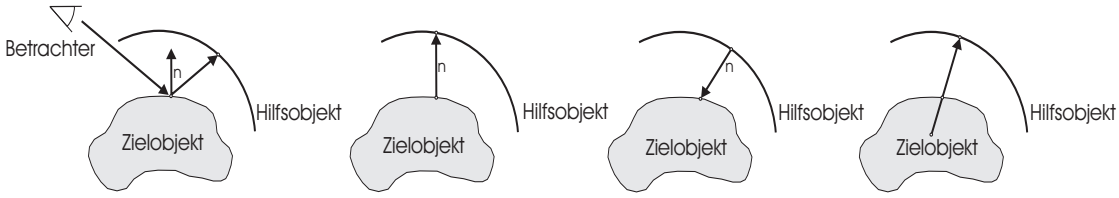


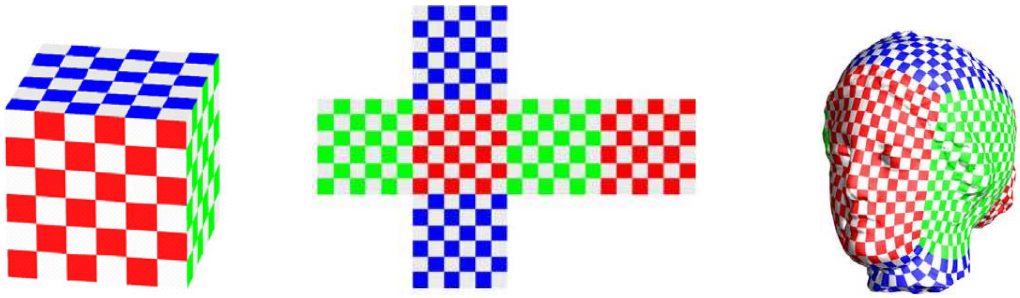
Abbildung 2.10: Bei der Zwei-Phasen-Texturprojektion von Bier und Sloan [1986] existieren verschiedene Varianten, um die Textur vom Zwischenobjekt auf das eigentliche Objekt abzubilden. Diese Darstellung zeigt einige typische Methoden.

Es ergeben sich eine Reihe verschiedener Möglichkeiten, mit denen eine Textur auf ein Objekt abgebildet werden kann. Die einfachste Veränderung, die mit Texturen erreicht wird, ist die Änderung der Objektfarbe. Es existieren aber auch Verfahren, die die Normalen-Vektoren der Oberfläche manipulieren (*Bump Mapping*), eine Umgebungsspiegelung auf dem Objekt (*Environment Mapping*) erzeugen oder andere Veränderungen beschreiben. Kugeln oder Würfel sind häufig verwendete Hilfsobjekte.

Sphere Mapping Bei dieser Variante wird die Textur auf die Oberfläche einer Kugel abgebildet. Im Gegensatz zum *Cube Mapping* wird hier nur eine Textur verwendet, die alle abzubildenden Informationen enthält. Das Problem bei diesem Verfahren besteht darin, eine rechteckige 2D-Textur auf die Kugeloberfläche abzubilden, da vor allem an den Polen ungewollte Verzerrungen

auftreten. Auf die Ursachen und mögliche Gegenmaßnahmen wird in Abschnitt 2.3.2 näher eingegangen.

Cube Mapping Das Objekt befindet sich in einem geschlossenen Würfel, auf dessen Oberfläche die Textur aufgebracht wurde (siehe Abbildung 2.11 (a)). Die *Cube Map*-Textur besteht, wie in Abbildung 2.11 (b) dargestellt, aus sechs gleich großen, quadratischen Bildern, die jeweils einer Seite des Würfels zugeordnet werden. In Abbildung 2.11 (c) ist die Texturabbildung der einzelnen Würfelseiten auf das Objekt veranschaulicht.



(a) (b) (c)

Abbildung 2.11: Eine *Cube Map*-Textur wird (a) auf einen Würfel abgebildet und besteht. (b) aus sechs gleich großen, rechteckigen 2D-Texturen, die jeweils eine Seite des Würfels repräsentieren. (c) Das Objekt wird innerhalb des Würfels platziert und ausgehend davon dann texturiert. Quelle: Praun und Hoppe [2003]

Cube Maps bieten im Gegenteil zu *Sphere Maps* durch die Würfelseiten sechs verschiedene Texturen. Der Nachteil ist der damit verbundene höhere Speicherplatzbedarf.

2.3.2 Texturverzerrung

Eine beliebige 3D-Oberfläche exakt mit einer 2D-Textur zu bedecken, ist meist sehr schwierig oder sogar ausgeschlossen. Vor allem für Oberflächen mit komplexer Geometrie ist es nicht möglich ohne Streckung, Stauchung, Wiederholung oder Zuschneiden der Textur, diese auf das Objekt abzubilden [Akenine-Moller et al., 2002]. Verzerrungen und Nähte sind dabei (*Seams*) die größten Probleme, die beim *Texture Mapping* auftreten.

Bei der Zuordnung von Texturen zu einzelnen Polygonen oder Gruppen von Polygonen können zwischen den verschiedenen, texturierten Polygonen sichtbare Nähte auftauchen. Texturverzerrungen dagegen enstehen wenn die Größe der Textur nicht mit der Größe des Objektes beziehungsweise Objektteils übereinstimmt, auf dessen

Oberfläche abgebildet werden soll. In diesen Fällen wird dann die Textur angepasst und demzufolge gestreckt oder gestaucht.

Für jeden Punkt auf der Oberfläche existiert ein korrespondierender Punkt auf der Textur und umgekehrt. Bei dem *Texture Mapping* erfolgt während des ersten Schrittes eine Parametrisierung des Objektes. Die Textur wird aus dem Texturraum auf das Objekt im Objektraum abgebildet, wobei eine Abbildungsfunktion definiert werden muss, die jedem Objektpunkt den zugehörigen Texel zuordnet. Geeignet sind die Funktionen, die eine mögliche Verzerrung der Textur auf der Oberfläche minimieren oder ausschließen. Zur Definition derartiger Funktionen wird eine bestmögliche Parametrisierung der Objektoberfläche benötigt.

Parametrisierung

Die Parametrisierung ist eine bijektive Abbildung einer Oberfläche auf eine andere Oberfläche [Floater und Hormann, 2004]. Das bedeutet, es ist eine eindeutig umkehrbare Abbildung. Die Parametrisierung wird als Abbildung der 3D-Oberfläche des Objektes auf eine planare 2D-Ebene betrachtet. Carr und Hart [2002] definieren die Parametrisierung wie folgt: Ist das *Texture Mapping* auf eine Oberfläche eine bijektive Funktion, so ist die Inverse dieser Funktion die Parametrisierung der Oberfläche.

Bei der Parametrisierung entstehen meist bedingt durch Skalierung, Stauchung oder Streckung Verzerrungen der Winkel oder der Flächen. *Mapping*-Techniken versuchen diese Verzerrungen zu minimieren, indem geeignete Abbildungsfunktionen verwendet werden. Nach Floater und Hormann [2004] wird daher für die Parametrisierung versucht eine Abbildung zu finden, die entweder konform, flächentreu oder isometrisch ist. Sei S eine Oberfläche und f die Abbildung von S auf eine zweite Oberfläche S^*. Die Parametrisierung x^* von S^* wird so definiert, dass die Koordinaten eines beliebigen Bildpunktes $f(p) \in S^*$ gleich denen des korrespondierenden Punktes $p \in S$ im Urbild sind. Eine Abbildung f ist zulässig, wenn die Parametrisierung x^* eindeutig ist [Floater und Hormann, 2004].

- **konforme Abbildung:**
  Eine zulässige Abbildung von S nach S^* ist konform oder winkeltreu, wenn der Winkel, unter dem sich zwei beliebige Kurven auf S in einem Punkt P schneiden, gleich dem Winkel ist, unter dem sich ihre korrespondierenden Bildkurven auf S^* im Bildpunkt P^* von P schneiden.

- **flächentreue Abbildung:**
  Eine zulässige Abbildung von S nach S^* ist flächentreu, wenn jedes Teilgebiet T auf S in ein Teilgebiet T^* auf S^* abgebildet wird, das den gleichen Flächeninhalt hat.

- **isometrische Abbildung:**
  Eine zulässige Abbildung von S nach S^* ist isometrisch oder längentreu, wenn die Länge jedes Kurvenstücks auf S mit der Länge seines Bildkurvenstücks S^* übereinstimmt.

Die isometrische Abbildung kann als die ideale Abbildung angesehen werden, da sie konform und flächentreu ist und somit Winkel, Flächen und Längen bewahrt werden. Da meist keine isometrische Abbildung möglich ist, wird beim *Texture Mapping* versucht eine Minimierung der Winkel- und Flächenverzerrung zu erreichen [Floater und Hormann, 2004].

Bei der Texturierung einzelner Polygonen ist die Parametrisierung am einfachsten, da die Texturkoordinaten hier den *Vertices* der Polygone zugewiesen werden [Heckbert, 1986]. Bei Objekten, deren geometrische Form allerdings komplexer aufgebaut ist, besteht das Problem eine geeignete Anpassung der Textur an die Objektform zu bestimmen.

Der traditionelle Ansatz der Oberflächen-Parametrisierung besteht in dem Zerschneiden der Oberfläche in kleine Stücke. Diese werden dann einzeln parametrisiert und texturiert. Es entstehen dann allerdings beim Zusammensetzen der Texturstücke auf dem Objekt sichtbare Nähte zwischen den Texturstücken. Praun et al. [2000] verwenden bei ihrer *Lapped Textures*-Technik kleine Ausschnitte von 2D-Texturen, die lokal auf planare Teilstücke (einzelne oder Gruppen von Polygonen) des 3D-Objektgitters aufgebracht werden, bis die gesamte Oberfläche mit einer Reihe überlappender Texturstücke bedeckt ist. Die einzelnen Texturen werden ineinander übergeblendet, um Kanten zwischen zwei aneinander stoßenden Texturen zu vermeiden.

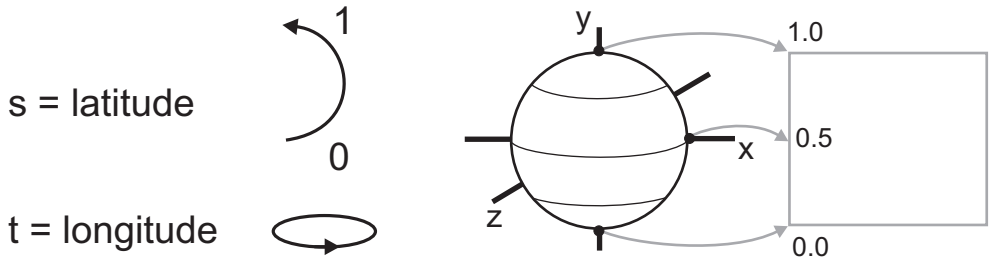
Einige Techniken nutzen auch die beschriebene Zwei-Phasen-Texturprojektion, um diese Objekte zu texturieren (siehe Seite 25). Es werden wie bereits erwähnt Objekte wie Kugel, Würfel und Zylinder als Hilfsobjekte eingesetzt. Ihre Oberfläche lässt sich durch mathematische Funktionen einfach beschreiben und ermöglicht eine Textur-Abbildung, die einer der eben genannten Anforderungen entspricht.

Kugel

Die Parametrisierung der Kugeloberfläche erfolgt mit Hilfe von Längen- (Longitude) und Breitengrad (Latitude) (siehe Abbildung 2.12). Wird eine rechteckige 2D-Textur auf eine Kugel abgebildet, treten vor allem an den Polen störende Verzerrungen auf. Das Bild wird über die Kugel gezogen und infolgedessen am Äquator gedehnt und an den Polen gestaucht [Praun und Hoppe, 2003]. Bei der

Verwendung eines rechteckigen Bildes ist es nicht möglich Texturverzerrungen zu vermeiden [Floater und Hormann, 2004].

Folglich existiert auch keine isometrische Abbildung von einer planaren 2D-Textur auf eine Kugel. Da eine Kugel aber laut Floater und Hormann [2004] beispielsweise mit der Mercator-Projektion[2] konform (winkeltreu) auf eine Ebene projeziert werden kann, ist es auch möglich eine Textur zu erstellen, die konform auf die Kugel aufgebracht wird. Die Textur muss folglich extra verzerrt werden,



(a) (b)

Abbildung 2.12: (a) Mit Longitude und Latitude (Längen- und Breitengrad) wird die Oberfläche einer Kugel (b) auf die (s,t) Texturebene abgebildet und demnach parametrisiert.

um Texturverzerrungen zu vermeiden. Miller und Hoffmann [1984] erstellten eine Umgebungstextur für das *Environment Mapping*, indem sie ein Bild einer glatten Kugel erzeugten, auf deren Oberfläche sich die Szenenumgebung spiegelt. Die Textur ist demnach sphärisch parametrisiert und kann verzerrungsfrei auf die Kugel und später auf das Objekt übertragen werden [Miller und Hoffmann, 1984].

Zylinder

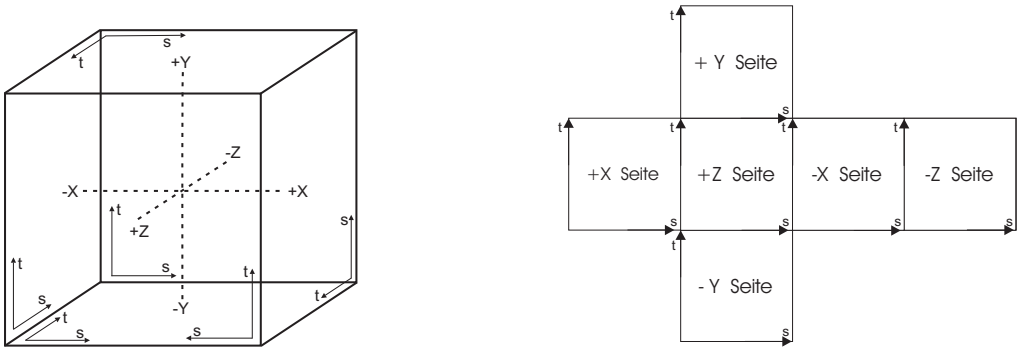
Die Abbildung einer Textur auf einen Zylinder ist isometrisch (längentreu) [Floater und Hormann, 2004]. Eine Textur wird an die Höhe des Zylinders angepasst und um diesen „herumgewickelt“. Die Koordinaten (s,t) werden entsprechend der Höhe und des Umfangs des Zylinders bestimmt. Die Textur muss ein fortlaufendes Muster repräsentieren, um nahtlos auf einen Zylinder abgebildet werden zu können. Desweiteren sind offene Zylinder nicht geeignet um kugelförmige Objekte zu texturieren, da die Texturabbildung von dem Zylinder auf das Objekt starke Verzerrungen an den Polen des Objektes erzeugt. Während auf eine Kugel und auf einen offenen Zylinder nur eine Textur abgebildet werden kann, besteht

[2]Gerhard Mercator (1514-1594) entwickelte eine Navigationskarte, die konform auf eine Kugel abgebildet werden kann.

bei einem geschlossenen Zylinder die Möglichkeit zwei separate Texturen für die obere und untere Kreisfläche zu verwenden.

Würfel

Bei der Verwendung eines Einheitswürfels können die 2D-Texturen planar auf die Seiten abgebildet werden, wobei die Eckpunkte einer Textur auf die Eckpunkte der Würfelseite abgebildet werden (siehe Abbildung 2.13 (a)). Die Texturen müssen nur auf die Größe der Seite skaliert werden. Da hier die Kanten des Würfels mit den Grenzen der Textur übereinstimmen und die *Cube Map* so konstruiert wird, dass sich die Texturkoordinaten kontinuierlich ändern, gibt es keine Probleme mit Nähten [Tarini et al., 2004]. Der Würfel ist achsenparallel ausgerichtet und befindet sich mit seinem Zentrum im Koordinatenursprung. Die (s, t) Koordinaten der Textur entsprechen den jeweiligen Achsen der betrachteten Würfelseite. Weiterhin können bedingt durch die sechs verschiedenen Seiten des Würfels auch sechs unterschiedliche Texturen verwendet werden (siehe Abbildung 2.13 (b)).



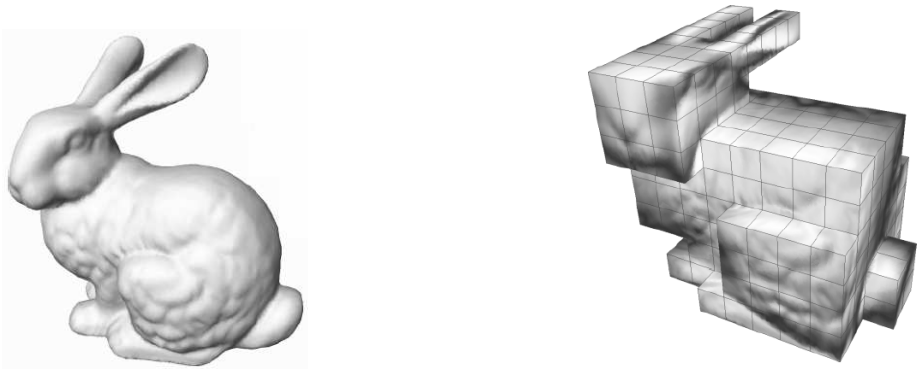
(a) (b)

Abbildung 2.13: (a) Der Würfel für die Textur ist achsenparallel und mit seinem Zentrum im Koordinatenursprung positioniert. (b) Jede Textur wird auf die entsprechende Seite abgebildet, wobei die (s, t) Koordinaten mit den jeweiligen Achsen übereinstimmen.

Ein bedeutender Vorteil ist die effiziente Speicherung der Textur. Da alle sechs Teiltexturen rechteckig und gleich groß sind, können sie in einer 2D-Textur ohne Platzverschwendung gespeichert werden. *Cube Mapping* wird im Zusammenhang mit einer Zwei-Phasen-Texturprojektion durchgeführt und realisiert für annähernd kugelförmige Objekte ein nahtloses *Texture Mapping* [Tarini et al., 2004]. Der Würfel sollte möglichst eng das Objekt umschließen. Eine *Bounding Box* ist als umgebender Würfel bei kugelförmigen Objekten geeignet. Je kleiner das Objekt im Bezug zum Würfel oder je komplexer die Topologie des Objektes, desto größer werden die auftretenden Verzerrungen.

Polycube-Technik

Tarini et al. [2004] erweiterten den bisherigen Ansatz und entwickelten ein Verfahren, welches auch für komplexe Objekte die Vorteile eines Würfels als Hilfsobjekt bei der Zwei-Phasen-Texturprojektion nutzen kann. Hierfür wird das gesamte Objekt in Einheitswürfel unterteilt, um dann lokal für die entsprechenden Teile des Objektes ein *Cube Mapping* durchführen zu können. Aus dem Objekt wird zuerst in einem Vorverarbeitungsschritt ein *Polycube* erstellt, der möglichst exakt die Objektform repräsentiert. Ein *Polycube* bezeichnet eine 3D-Form, die aus mehreren Einheitswürfeln zusammengesetzt ist, welche jeweils eine Seite gemeinsam haben [Tarini et al., 2004]. Da das Objekt nun aus vielen Würfeln besteht, ist es möglich, die Oberfläche problemlos zu parametrisieren und die Textur darauf abzubilden.



(a) (b)

Abbildung 2.14: (a) Tarini et al. [2004] generieren für ein Modell (b) den entsprechenden *Polycube*. Dieser ermöglicht eine Textur-Abbildung mit minimaler Verzerrung. Quelle: Tarini et al. [2004]

Der zweite Schritt beinhaltet die Abbildung der Textur von dem *Polycube* auf das Objekt. Für jeden Würfel wird lokal ein *Mapping* auf das Objekt durchgeführt. Die Größe der erzeugten Würfel ist abhängig von der Komplexität des Objektes. Je komplexer die Geometrie, desto kleinere Würfel werden benötigt, um Verzerrungen bei der Abbildung der Textur auf das eigentliche Objekt gering zu halten. Die jeweiligen Texturen für die Einheitswürfel werden in einer großen 2D-Textur gespeichert.

2.4 Programmierbare Graphik-Hardware

Das wachsende Interesse an echtzeitfähigen, virtuellen Animationen beeinflusste in den letzten Jahren bedeutend den technologischen Fortschritt auf dem Gebiet der Graphik-Hardware. Vor allem die Computerspiele- und Filmindustrie, die sich computergenerierten Bildern zuwandte, beschleunigte die Entwicklungen auf dem Gebiet der Computergraphik.

Rendering bezeichnet den Verarbeitungsprozess, der ein Modell aus einer 3D-Anwendung in ein 2D-Bild umwandelt und dieses dann auf dem Bildschirm abbildet. Die Implementierung in Soft- oder Hardware wird als Render-Pipeline bezeichnet [Foley et al., 1996]. Für Darstellungen in Echtzeit ist eine ausschließlich in Software implementierte Render-Pipeline hingegen zu langsam. Sie benötigt die Beschleunigung mittels Hardware. Der Nachteil einer Hardware-Implementierung besteht allerdings im beschränkten Einfluss des Entwicklers auf die Render-Pipeline. Seit einigen Jahren werden daher Graphikkarten mit programmierbaren Prozessoren entwickelt, die diese Grenzen aufheben. Vor der Einführung spezieller 3D-Graphik-Hardware[3] wurden die Aufgaben der Render-Pipeline vom Prozessor des Computers (CPU) durchgeführt und zur Darstellung der Daten an die Graphikkarte weitergeleitet. Komplexere, realitätsnahe Visualisierungen waren dementsprechend rechenintensiv und langwierig.

Zugunsten der Performance werden heutzutage spezielle Graphikoperationen nicht mehr von der CPU übernommen, sondern direkt in die Graphik-Hardware integriert. Die GPU ist ein Graphikprozessor, der auf jeder 3D-Graphikkarte vorhanden ist und rechenintensive Graphikberechnungen ausführt. Die modernen Graphikchips besitzen programmierbare *Vertex*- und *Fragment*-Prozessoren, die eine direkte Verarbeitung der Daten auf dem Chip ermöglichen [NVI, 2002]. Mittels parallel arbeitender Pipelines in der GPU-Architektur wird eine Steigerung der Performance erreicht. Folglich ist es möglich, eine große Anzahl an Daten gleichzeitig zu bearbeiten. Die parallele Bearbeitung von Daten und die Entlastung der CPU ermöglichen eine effiziente und schnellere Generierung von Echtzeit-Visualisierungen.

Die Vorteile programmierbarer Graphik-Hardware werden deutlich, wenn deren Einordnung in der 3D-Render-Pipeline näher betrachtet wird. Der konzeptuelle Aufbau einer Render-Pipeline und die Erweiterung dieser zur programmierbaren Pipeline wird in den folgenden Abschnitten behandelt.

[3] Graphikhardware, die auf die Berechnung von 3D-Graphiken spezialisiert ist

2.4.1 Echtzeit 3D-Render-Pipeline

Als Render-Pipeline beschreibt man die sequentiell auszuführenden Berechnungsschritte, um aus der Beschreibung eines Bildes das tatsächliche Bild zu generieren [Mark et al., 2003]. Der Ablauf ist durch verschiedene Abschnitte beziehungsweise Schritte charakterisiert, die die Algorithmen zur Transformierung der Geometriedaten nacheinander ausführen. Die Render-Pipeline generiert ein 2D-Bild anhand einer gegebenen virtuellen, 3D-Szene, bestehend aus Kamera, 3D-Objekten, Lichtquellen, Beleuchtungsmodellen und Texturen. In der Literatur wird diese Pipeline auch als Graphik-Pipeline oder feste Funktions-Pipeline (*Fixed Function* Pipeline) bezeichnet [Mark et al., 2003].

In Abbildung 2.16 sind die Schritte der festen Funktions-Pipeline auf der linken Seite untereinander dargestellt. Im Folgenden soll nun kurz jede Stufe dieser Render-Pipeline betrachtet werden, um anschließend die hinzugefügten programmierbaren Einheiten zu erläutern.

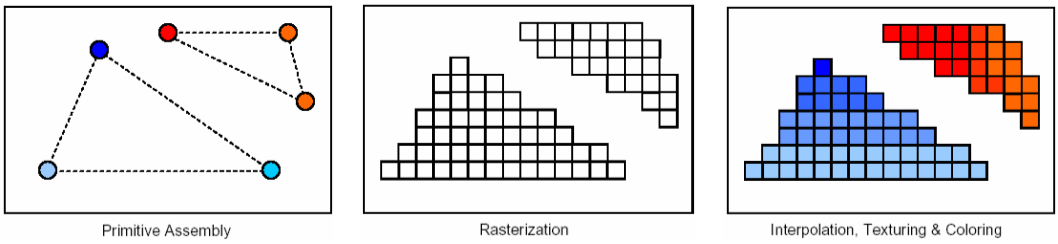
- **Geometrie spezifizieren**
  In einer Anwendung wird zunächst die gesamte Szene mit allen Informationen über Objekte, verwendete Materialien, Lichtquellen und allen anderen Szenenbestandteilen vom Benutzer festgelegt. Im Allgemeinen werden 3D-Objekte als eine Ansammlung von Polygonen betrachtet, die das Gittermodell eines Objektes repräsentieren. Die Eckpunkte der Polygone werden *Vertices* genannt.

- **Operationen pro Vertex**
  Zu den Operationen, die pro *Vertex* ausgeführt werden, zählen die Transformation der *Vertex*-Koordinaten von Objekt- in Kamerakoordinaten (*Model & View Transform*), die Beleuchtung sowie die Projektion der *Vertices*.

- **Primitive Assembly**
  Aus den transformierten und gefärbten *Vertices* werden Punkte oder sie werden zu Linien oder Polygonen zusammengesetzt (Abbildung 2.15 (a)).

- **Verarbeitung der Primitive**
  Auf dieser Stufe werden Operationen wie *Clipping* oder *Screen Mapping* ausgeführt. Alle in der Szene enthaltenen Primitiven werden an einem Einheitswürfel getestet, ob sie später auf dem Bildschirm vollständig oder teilweise sichtbar sind [Akenine-Moller et al., 2002]. Die Primitive, die nicht in dem Einheitswürfel liegen, werden entfernt. Das *Screen Mapping* ist die Transformation der 3D-Koordinaten in 2D-Bildschirmkoordinaten.

- **Rastern & Interpolieren**
  Die Graphikprimitive wie Linien oder Polygone werden gerastert und somit in Pixel konvertiert [Foley et al., 1994]. Alle Pixel, die von dem jeweiligen Primitiv bedeckt sind werden ermittelt und als *Fragmente* weitergeleitet (Abbildung 2.15 (b)). *Fragmente* sind die Elemente, die sich noch in der Pipeline befinden und die anschließend als Pixel auf dem Bildschirm dargestellt werden. Zur Darstellung eines Pixels auf dem Bildschirm können mehrere *Fragmente* zusammengefasst werden. *Fragmente* werden in der Literatur auch als „potentielle“ Pixel bezeichnet [Fernando und Kilgard, 2003]. Farbe, Tiefenwerte und Texturkoordinaten für ein *Fragment* werden durch lineare Interpolation der zugehörigen *Vertex*-Werte berechnet (siehe Abbildung 2.15 (c)).



(a) (b) (c)

Abbildung 2.15: (a) Aus den *Vertices* werden Punkte oder sie werden zu Primitiven zusammengesetzt. (b) Die Primitiven werden gerastert und anschließend erfolgt (c) die Interpolation der Attributwerte der jeweiligen *Vertices* über die zugehörigen *Fragmente*.

- **Operationen pro Fragment**
  Die wichtigste Operation, die an dieser Stelle der Pipeline ausgeführt wird, ist die Texturierung. Mit den Texturkoordinaten der *Fragmente* kann ein Texturzugriff erfolgen. Weiterhin werden hier Berechnungen zur Bestimmung der Farbe eines *Fragmentes* ausgeführt.

- **Rasteroperationen**
  Zu den Rasteroperationen zählt das *Clipping* auf Pixel- bzw. *Fragment*-Ebene und Berechnungen wie *Alpha*-, *Depth*- und *Stencil*-Test sowie viele weitere Operationen.

- **Anzeige**
  Abschließend wird die gesamte Szene auf dem Bildschirm dargestellt.

Aufgaben wie die Transformation der *Vertices* und das Füllen von Polygonen besitzen die Eigenschaft leicht parallelisierbar zu sein, folglich ist es möglich diese Schritte in der Hardware zu implementieren und somit die CPU zu entlasten.

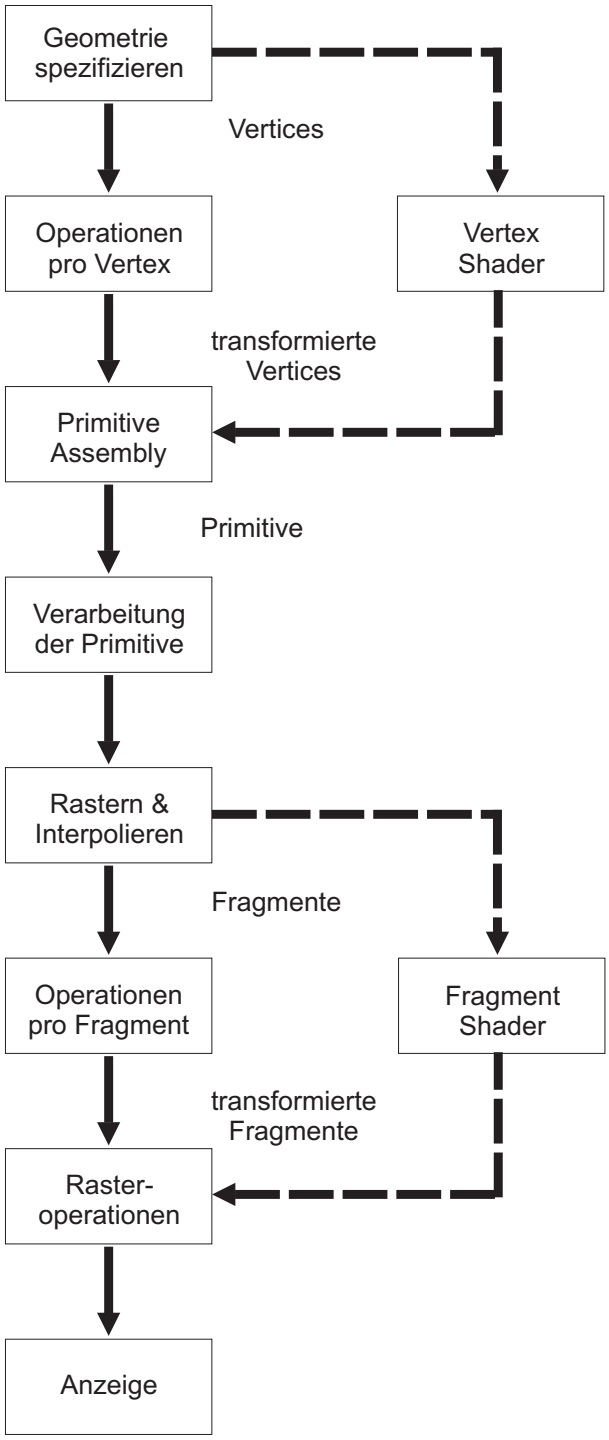


Abbildung 2.16: Auf der linken Seite ist die feste Funktions-Pipeline dargestellt. *Shader* ermöglichen die Ersetzung der Operationen pro *Vertex* und pro *Fragment*, der festen Funktions-Pipeline.

Mit spezieller Hardware lässt sich die parallele Bearbeitung einzelner Aufgaben verwirklichen. So können beispielsweise neue *Vertices* bereits transformiert werden, während die Vorgänger gerade eine Ebene weiter Beleuchtungsberechnungen durchlaufen. Moderne Graphik-Chips besitzen daher eine gewisse Anzahl an parallel arbeitenden *Vertex*- und *Fragment*-Pipelines und beschleunigen somit die Verarbeitung der Daten. Im folgenden Abschnitt wird nun der Aufbau programmierbarer Pipelines betrachtet.

Programmierbare Render-Pipeline

In den letzten Jahren fand eine partielle Ersetzung der beschriebenen festen Funktions-Pipeline statt, wodurch der Einfluss des Entwicklers auf die Graphikberechnungen vergrößert wurde. Nachdem anfangs nur die Operationen pro *Fragment* in Hardware implementiert waren, folgte darauf die Programmierbarkeit von *Vertex*-Prozessoren (Operationen pro *Vertex*). Der Fortschritt im Bereich der Graphik-Prozessor-Technologie gibt dem Benutzer an zwei Stellen direkte Kontrolle über die Verarbeitungsschritte pro *Vertex* und pro *Fragment*. Abbildung 2.16 zeigt rechts die Erweiterung der Pipeline durch die Verwendung von *Vertex*- und *Fragment Shader*[4].

Der Datenstrom von der CPU wird an die GPU übergeben und durchläuft die einzelnen Pipeline-Stufen. *Vertex*- und *Fragment*-Prozessoren sind zwei neue Funktionalitäten der GPU, die eine benutzerspezifische Manipulation des Datenstromes durch *Shader*-Programme in er Render-Pipeline gestatten [NVI, 2002]. Mit Hilfe von *Shader*-Programmen können Operationen pro *Vertex* oder pro *Fragment* definiert und dadurch von der festen Pipeline-Struktur abweichende Modifikationen der Daten vorgenommen werden. *Vertex Shader* verändern die geometrischen Informationen der *Vertices* und *Fragment Shader* arbeiten auf den interpolierten Daten der *Fragmente*. Wird kein *Shader*-Programm definiert, durchlaufen die Daten die Stufen der festen Funktions-Pipeline mit den in Abschnitt 2.4.1 erläuterten Transformationen (siehe Abbildung 2.16).

2.4.2 Shader

In der Computergraphik wird die Generierung realistischer Visualisierungen angestrebt. Die Realistätsnähe einer Darstellung wird allerdings nicht nur durch die Komplexität der Szene und der Geometrie, sondern auch durch das *Shading* bestimmt. Alle Operationen, die das Erscheinungsbild eines *Fragmentes* und damit

[4] *Fragment Shader* werden auch Pixel-*Shader* genannt

auch eines Pixels beeinflussen, werden unter diesem Begriff zusammengefasst. Dazu gehören beispielsweise Beleuchtungsberechnungen und Texturierung.

Shader sind kurze, abgeschlossene Programme, die pro *Vertex* oder pro *Fragment* vom jeweiligen Prozessor aufgerufen und ausgeführt werden. Damit ist es möglich, die zu einem *Vertex* oder *Fragment* gehörenden Parameter individuell zu manipulieren. *Shader* führen auf den einzelnen Elementen des Datenstromes Berechnungen aus und erzeugen dann einen Ausgabestrom, der in der Pipeline weitergeleitet wird.

Mittels *Shader*-Programmen wird eine einfache und hardwarebeschleunigte Implementierung von benutzerspezifischen Veränderungen der Objekte realisierbar [Mark et al., 2003]. Die Verwendung der *Shader* ist optional, was bedeutet, dass die gleiche Funktion auch der Pipeline überlassen werden kann. Weiterhin ist es möglich, *Vertex Shader* ohne *Fragment Shader* zu verwenden oder umgekehrt.

Ein bedeutender Vorteil sind die Verbesserungen der Texturierung von Objekten mittels *Shader*. Aufgrund schneller Texturzugriffe sind effiziente Darstellungen in Echtzeit möglich. Bei der Entwicklung von Computerspielen wird vor allem die schnelle Umsetzung von Effekten und Animationen ausgenutzt und erweitert, wodurch sich die Architektur und die Funktionalität der Graphik-Hardware ständig weiterentwickelt.

Vertex Shader

Vertex Shader sind Programme, die pro *Vertex* ausgeführt werden und unter anderem die in Abbildung 2.16 dargestellte Stufe der pro *Vertex* Operationen durch benutzerspezifische Operationen ersetzen können. Hierzu zählen:

- Transformation der *Vertices*
- Transformation der Normalen-Vektoren
- Beleuchtung
- Berechnung der Farbwerte

Wie bereits erwähnt, ist der Einsatz der *Shader*-Programme optional. Bei Verwendung eines individuellen *Vertex Shaders* sind die oben genannten Funktionen der festen Funktions-Pipeline inaktiv und müssen alle von dem *Shader*-Programm übernommen werden [Rost, 2004]. Es ist beispielsweise nicht möglich einen *Shader* zu schreiben, der nur die Beleuchtung oder Farbberechnung des *Vertex* übernimmt, ohne ebenfalls die Transformation im Programm definiert zu haben.

Zusätzlich können noch folgende Graphikoperationen ausgeführt werden:

- Normalisierung der Normalen-Vektoren
- Generierung der Texturkoordinaten
- Transformation der Texturkoordinaten

Vertex Shader erhalten als Eingabe einen *Vertex* mit zugehöriger Attributliste. Der Normalen-Vektor, Farbwert oder Texturkoordinaten sind Attribute, die mit einem *Vertex* abgespeichert sein können und über die Pipeline in den *Shader* gelangen. Es wird jeweils nur ein unbeleuchteter und untransformierter *Vertex* im *Shader* modifiziert. Der Ausgabestrom enthält mindestens die Position des *Vertices* in Bildschirmkoordinaten. Optional können noch weitere Werte im *Shader* berechnet werden, wie beispielsweise eine individuelle Beleuchtung des *Vertex*, Nebel- und Texturkoordinaten oder eigene benutzerspezifische Parameter. Diese werden dann zu der bisherigen Attributliste der *Vertices* hinzugefügt und interpoliert an den *Fragment Shader* weitergeleitet.

Ein *Vertex*, der den *Shader* durchläuft, muss allerdings nicht zwingend verändert werden. Es ist auch möglich nur *Vertices* mit bestimmten Eigenschaften zu verändern. *Vertices* können im *Shader* weder gelöscht noch erzeugt werden, ein *Vertex* als Eingabe ergibt auch einen *Vertex* als Ausgabe [Fernando und Kilgard, 2003]. Weiterhin liegen im *Shader* keine Informationen über die topologischen Zusammenhänge zwischen den einzelnen *Vertices* vor. Da jeder separat behandelt wird und kein Zugriff auf Vorgänger oder Nachfolger möglich ist, können weder Informationen über die Nachbarn des aktuellen *Vertex*, noch über seine Lage als Eckpunkt im Objektgitter vom *Shader* bestimmt werden. Standardmäßig können *Vertex Shader* keine Operationen in der Pipeline ersetzten, die Angaben über mehrere *Vertices* zu einem Zeitpunkt oder topologisches Wissen erfordern [Rost, 2004].

Wie Mitchell und Card [2002] und McGuire und Hughes [2004] gezeigt haben, lassen sich diese Grenzen umgehen, indem die *Vertex*-Attributlisten um Nachbarschaftsinformationen erweitert werden. Zu den bisherigen Attributen werden in einem Vorverarbeitungsschritt zusätzlich Informationen abgespeichert, die es ermöglichen, die zu einem *Vertex* gehörenden Kanten im *Shader* zu rekonstruieren. Der *Shader* bearbeitet weiterhin alle *Vertices* separat, kann aber anhand der Attribute nun topologische Informationen ableiten. Alle dafür benötigten Werte werden durch eine Vorverarbeitung der geometrischen Modelle gewonnen. Mitchell und Card [2002] speichern hierfür zu jedem *Vertex* die zugehörigen Kanten und die daran angrenzenden Flächen ab. McGuire und Hughes [2004] hingegen speichern alle *Vertices* ab, die an die zu dem *Vertex* gehörenden Kanten angrenzen.

Vertex Shader realisieren die Veränderung der Gestalt eines Objektes, Bewegungen, Spiegelungen oder Deformationen. Weiterhin können Oberflächenerscheinun-

gen, Farbwert, Beleuchtung und Texturkoordinaten des betreffenden *Vertices* generiert oder verändert werden. Der *Vertex Shader* ist besonders geeignet um Vorberechnungen durchzuführen, auf denen der *Fragment Shader* aufbaut. Die ausgegebenen Daten durchlaufen die nächsten Stufen der Pipeline und werden weiter modifiziert. Programmierer können dann wieder auf *Fragment*-Ebene die Daten individuell modifizieren.

Fragment Shader

Fragment Shader sind Programme, die pro *Fragment* aufgerufen und Operationen auf diesen Elmenten der Render-Pipeline übernehmen können. Dazu zählen Graphikberechnungen wie:

- benutzerspezifische Operationen auf den interpolierten Werten
- Texturzugriff
- Kombination der Farb- und Texturwerte
- Berechnung des Farbwerts für ein *Fragment*

Fragmente entstehen durch das Rastern der geometrischen Primitive. Da schon bei der Rasterung von Primitiven mit wenigen *Vertices* eine hohe Anzahl von *Fragmenten* entsteht, ist die Größe des Datenstromes und der damit verbundene Verarbeitungsaufwand um ein Vielfaches höher als im *Vertex*-Prozessor. Der vom *Fragment Shader* zu bearbeitende Datenstrom besteht nicht nur aus der Ausgabestruktur des *Vertex Shaders*, sondern auch aus den dazwischen interpolierten Werten.

Jedes *Fragment* wird vom *Shader* einzeln geladen und bearbeitet. *Fragmente* können weder gelöscht noch erzeugt werden. Weiterhin ist es nicht möglich ihre Position zu verändern, da diese bereits in Bildschirm-Koordinaten vorliegt. Die Ausgabestruktur ist einfacher als die des *Vertex Shaders*. Ein *Vertex Shader* muss die Position und kann Farbe, Texturkoordinaten oder andere Datenwerte ausgeben. Dagegen reduziert ein *Fragment Shader* alle Informationen auf einen Farb- oder Transparenzwert und aktualisiert damit den Bildspeicher. Bei Verwendung von Texturen ergibt sich die Farbe als Kombination des vorher berechneten Farbwertes mit dem Textur-Zugriffsergebnis.

2.4.3 Shader-Sprachen

Anfangs waren *Shader*-Programme assemblerähnliche Programme, da mit einem Assembler-Programm die Hardware direkt programmiert werden kann. Die

Assemblersprache ist eine maschinenorientierte (*low-level*) Programmiersprache, die von dem Befehlsvorrat des Prozessors abhängt und durch einen speziellen Compiler, den Assembler, in direkt ausführbare Maschinensprache umgewandelt wird. Die Assemblersprache ist hardware- und herstellerspezifisch und daher für jeden Prozessor unterschiedlich. Sie eignet sich lediglich zur Programmierung von kleinen, kurzen Programmen. Bei größeren Projekten wird die Programmierung durch den hohen Programmieraufwand und die Fehleranfälligkeit erschwert.

GPUs werden immer leistungsfähiger und unterstützen längere Programme, die auch Sprunganweisungen, Funktionen und Schleifen beinhalten können. Ein Assemblercode wird mit solchen Anweisungen sehr komplex und unübersichtlich und erschwert so die Programmierung und Fehlersuche.

Aufgrund fortschreitender Entwicklung der Graphikkartentechnologie ergab sich die Forderung nach Programmiersprachen, die es dem Entwickler ermöglichen, effizient und komfortabel Programme zu entwerfen. In den letzten Jahren entwickelten sich *Shader*-Sprachen, die einerseits Flexibilät und Leistung einer Assemblersprache und andererseits die Benutzerfreundlichkeit einer Hochsprache aufweisen.

Die Entwicklung der *Shader*-Hochsprachen (*high-level shading languages*) ist durch unterschiedliche Einflüsse geprägt. Zum einen wurden Syntax und Semantik von gängigen Hochsprachen wie C/C++ und Java übernommen. Zum anderen orientierte sich die Umsetzung einer *Shader*-Hochsprache an verfügbarer Hardware und den dominierenden 3D-Anwendungsprogrammierschnittstellen (API) OpenGL und DirectX [Fernando und Kilgard, 2003]. OpenGL ist ein offener Standard und wird vom *Architecture Review Board* (ARB)[5] entwickelt. Das ARB existiert seit 1992 und ist eine Institution, die auf SGI zurück geht und in der neben SGI viele weitere Firmen beteiligt sind. DirectX ist eine alternative Graphikbibliothek zu OpenGL und wurde von Microsoft entwickelt. Der Vorteil von OpenGL ist die Plattformunabhängigkeit.

Im Folgenden wird ein Überblick über die wichtigsten *Shader*-Hochsprachen für die Echtzeit-Programmierung gegeben.

Cg

NVIDIA, einer der größten Entwickler von Graphikprozessoren und Chipsätzen, veröffentlichte im Jahr 2002 die *Shader*-Sprache Cg, mit der die Programmierung der *Shader* und damit die Realisierung visueller Effekte in Echtzeit möglich

[5] Das ARB ist ein Zusammenschluss der Firmen 3Dlabs, Apple, ATI, Dell, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Matrox, NVIDIA und Sun. Microsoft, eines der Gründungsmitglieder, hat das ARB im März 2003 verlassen.

ist [Proudfoot et al., 2001]. Cg steht für *C for Graphics*. Ein bedeutender Fortschritt und wesentlicher Vorteil gegenüber anderen *Shader*-Sprachen ist die Compiler-Struktur, die *Vertex*- und *Fragment*-Programme für OpenGL- und DirectX-Umgebungen erlaubt [NVI, 2002].

Cg ist eine Programmiersprache, die einerseits die gleiche Flexibilität und Leistung einer Assemblersprache und andererseits die Benutzerfeundlichkeit einer Hochsprache wie C/C++ bietet. Das Programmieren mit Cg ist effizient und erlaubt daher, eine große Anzahl *Shader* innerhalb kurzer Zeit zu entwickeln. Weiterhin können einmal entwickelte Effekte für mehrere APIs und Plattformen kompiliert werden. Das für Cg entwickelte Profil-Konzept ermöglicht die Unterstützung unterschiedlicher Hardwarefähigkeiten einzelner GPU Generationen. Ein Profil fasst jeweils die Möglichkeiten der Hardware einer entsprechenden Generation zusammen. Falls das geschriebene Programm über die Möglichkeiten der benutzen Hardware hinausgeht, werden bereits beim Kompilieren entsprechende Fehlermeldungen angezeigt.

Cg ist mit einer umfangreichen Anzahl an mitgelieferten mathematischen Funktionen und Operationen ausgestattet und besitzt die für 3D-Graphik essentiellen Vektor- und Matrix-Operationen als grundlegenden Teil der Sprache [Fernando und Kilgard, 2003].

Grundlage für den Aufbau eines Programmes bildet der Datenstrom zwischen der API, dem *Vertex* und dem *Fragment Shader*. Ein *Shader*-Programm besteht aus der Definition von Ein- und Ausgabestrom und einer Hauptfunktion, die alle auszuführenden Operationen und Funktionen beinhaltet. Ein- und Ausgabestrom werden in Cg-Syntax über `structs` definiert und kennzeichnen alle Parameter, die eingelesen und ausgegeben werden sollen (siehe Listing 2.1). Die Ein- und Ausgabeparameter werden zur Speicherung vordefinierte Bezeichnungen, wie beispielsweise `POSITION`, `NORMAL`, `COLOR` oder `TEXCOORD0` bis `TEXCOORD7`, zugewiesen. Beim Aufruf eines Programms wird der Parameter dann mit den entsprechenden Eingabedaten initialisiert. Diese in Cg eingeführte Bindungssemantik dient zur Weiterleitung der Daten vom *Vertex* zum *Fragment Shader*.

Es existieren zwei Typen von Eingangsparametern: *Varying Inputs* und *Uniform Inputs*. *Varying Inputs* sind Werte, die für jeden betrachteten *Vertex* oder jedes *Fragment* variieren und an den jeweiligen *Shader* neu übergeben werden [NVI, 2002]. Beispielsweise sind in einem *Vertex Shader* die Normalen-Vektoren der *Vertices Varying Inputs*. *Uniform Inputs* hingegen sind Daten, die neben dem Datenstrom von der API an den *Shader* übergeben werden [NVI, 2002]. Diese ändern sich nicht mit jedem Element des Datenstroms, wie beispielsweise die Transformationsmatrix für einen *Vertex Shader* zur Umrechnung der *Vertex*-Position.

```
//Definition einer Struktur für die eingehenden Attribute aus der Anwendung
struct AppInput
{
       float4 Position : POSITION;
       float4 Normal : NORMAL;
};

//Definition der aus dem Vertex Shader auszugebenden Attribute
struct VertexOutput
{
       float4 vPosition : POSITION;
       float4 Color  : COLOR;
};

VertexOutput main(AppInput IN, uniform float4x4 ModelViewProj)
{...}
```

Listing 2.1: Beispiel eines Cg-Programmentwurfes für einen *Vertex Shader*. Quelle: NVI [2002]

HLSL

HLSL ist die Abkürzung für *High-Level Shader Language* und wurde von Microsoft in Zusammenarbeit mit NVIDIA 2002 entwickelt und veröffentlicht. Die Sprache ist syntaktisch und semantisch ähnlich der Sprache Cg. Der Unterschied besteht in der doppelten Abhängigkeit von HLSL, welche nicht für OpenGL, sondern nur für DirectX und auch nur mit Windows nutzbar ist [Rost, 2004]. HLSL besitzt ebenso wie Cg das Profil-Konzept und ermöglicht damit die *Vertex*- und *Fragment*-Programmierung für verschiedene Generationen von DirectX. Das Profil kann entsprechend der DirectX-Fähigkeit der verwendeten Graphikkarte direkt gewählt werden.

Aufgrund der Spezialisierung von HLSL besitzt die Sprache eine bessere Einbindung in DirectX als Cg und ist für Anwendungen, die ausschließlich auf DirectX basieren und auch nicht für andere Plattformen entwickelt werden sollen, eine komfortable Alternative zu Cg.

OpenGL Shading Language

Mit zunehmender Entwicklung der programmierbaren Pipeline veröffentlichte das ARB eine OpenGL-Spezifikation, die Erweiterungen enthält, mit denen eine Programmierung der GPU möglich wurde. Die ersten assemblerähnlichen *Vertex*-

und *Fragment*-Programme konnten den entsprechenden Teil der festen Funktions-Pipeline ersetzen und auf der Graphikkarte ausführen.

Die *OpenGL Shading Language* (GLSLANG) ist, wie die zuvor eingeführten Sprachen, eine Hochsprache und stellt den Nachfolger der in Assembler geschriebenen *Shader* dar. Mit dieser *Shading Language* können unter OPENGL *Vertex* und *Fragment Shader* in einer lesbaren, C-ähnlichen Hochsprache geschrieben werden, was deren Entwicklung stark vereinfacht. Eine hardware- und plattformunabhängige Programmierung von *Vertex* und *Fragment Shader*-Effekten wird damit möglich.

Der Programmaufbau ähnelt dem von CG. Wesentliche Bestandteile sind Ein- und Ausgabeparameter sowie eine `main`-Funktion und weitere individuelle vom Entwickler geschriebene Funktionen. Die Sprache unterstützt die gleichen Operationen und Anweisungen wie CG, die auf der Programmiersprache C basieren. Schleifen, bedingte Anweisungen, vordefinierte mathematische Operationen und Funktionen werden mitgeliefert beziehungsweise sind auch hier realisierbar.

Im Gegensatz zu CG werden Ein- und Ausgabeparameter in GLSLANG nicht mit semantischen Bindungen definiert, sondern mit Hilfe globaler Variablen, die entweder nur zum Schreiben oder Lesen gedacht sind [ROST, 2004]. Es existieren ebenfalls die Eingabeparameter *Varying* und *Uniform*. Für alle Variablen, die vom *Vertex* an den *Fragment Shader* weitergeleitet werden sollen, müssen passende Variablen im *Fragment Shader* deklariert sein. Die Definition dieser Variablen steht am Anfang des Programmes und kennzeichnet somit den Datenfluss.

2.5 Zusammenfassung

In diesem Kapitel wurden medizinische Atlanten betrachtet, in deren Illustrationen anatomische Strukturen zum besseren Verständnis über *Pen and Ink*-Techniken dargestellt sind. Die handgezeichneten Abbildungen können Informationen klar und präzise übermitteln, indem die Aufmerksamkeit des Lesers auf relevante Informationen gelenkt wird [WINKENBACH und SALESIN, 1994]. Strukturen, die für die Einordnung in den Kontext wichtig sind, werden dezent unter anderem mit der Hilfe von *Pen-and-Ink*-Techniken veranschaulicht. Da die Mediziner folglich bereits mit der *Stippling*-Technik vertraut sind, bietet es sich an, diese auch auf 3D-Datensätze anzuwenden.

In der Computergraphik existieren bereits verschiedene Ansätze, die eine Visualisierung der Daten über *Stippling* ermöglichen. Die Methoden können in objekt-, bild-, volumen- und texturbasierte Ansätze unterteilt werden. Für eine interaktive Echtzeit-Visualisierung müssen einerseits spezielle Anforderungen an die Skalierbarkeit der NPR-Primitive auf der Objektoberfläche sowie an die *Frame*-Kohärenz

erfüllt sein und andererseits wesentliche Merkmale einer *Stippling*-Illustration integriert werden. Texturen stellen eine geeignete Variante zur Visualisierung der Oberflächenmodelle da, wobei hier besonders bei der Texturabbildung auf die Minimierung der Verzerrungen geachtet werden muss. Das *Polycube*-Verfahren von Tarini et al. [2004] basiert auf dem traditionellen *Cube Mapping* und ermöglicht auch für komplexe Objekte eine Texturabbildung mit möglichst geringer Verzerrung.

Ferner besteht die Möglichkeit einer beschleunigten Visualisierung mit Hilfe von programmierbarer Graphik-Hardware. Moderne Graphikkarten ermöglichen die direkte Verarbeitung der 3D-Modelle auf dem Graphikchip. Der Entwickler hat die Möglichkeit individuelle Berechnungen auf den einzelnen *Vertices* oder *Fragmenten* auszuführen. Demzufolge können benutzerspezifische Modifikationen hardwarebeschleunigt durchgeführt werden. Im folgenden Kapitel wird nun, unter Berücksichtigung der Vor- und Nachteile der bereits vorgestellten Verfahren, ein Konzept für die hardwarebeschleunigte Echtzeit-Visualisierung von Oberflächenmodellen über *Stippling* entworfen.

3 Entwurf für Stippling-Visualisierungen von 3D-Datensätzen

In diesem Kapitel wird ein Konzept für die hardwarebeschleunigte Darstellung von dreidimensionalen, medizinischen Oberflächenmodellen über *Stippling* vorgestellt. Die Modelle sind triangulierte Objektgitter, die aus medizinischen Volumendaten generiert wurden. Die Objektgitter sind sehr komplex, da sie aus vielen kleinen Polygonen bestehen und dementsprechend viele Eckpunkte besitzen, die verarbeitet werden müssen. Für eine interaktive *Stippling*-Darstellung dieser Objekte werden zunächst anhand der in Kapitel 2 beschriebenen Grundlagen und existierenden Ansätze die charakteristischen Merkmale einer interaktiven NPR-Darstellung zusammengestellt und anschließend ein Konzept beschrieben, welches die wesentlichen Anforderungen integriert und eine Visualisierung ohne Vorverarbeitung der Geometrie oder zusätzliche Informationen über das Objektgitter hardwarebeschleunigt ermöglicht. Der Schwerpunkt liegt hierbei mehr in der echtzeitfähigen Realisierung und weniger in einer exakten Nachahmung von *Stippling*-Abbildungen.

3.1 Konzept einer interaktiven Stippling-Darstellung

Für eine interaktive, hardwarebeschleunigte *Stippling*-Darstellung müssen verschiedene Anforderungen erfüllt sein, um einerseits eine echtzeitfähige Visualisierung zu erreichen und andererseits die Merkmale einer traditionellen Illustration zu integrieren. In Abschnitt 2.2 wurden bereits existierende Ansätze zur bildbasierten, objektbasierten, texturbasierten oder in Kombination mit dem Volumen-Rendering erstellten *Stippling*-Visualisierung vorgestellt und ihre Vor- und Nachteile für eine interaktive Visualisierung aufgeführt. Sie erfüllen jeweils einzelne, der folgenden Anforderungen.

3.1.1 Anforderungen an interaktive Stippling-Darstellungen

Die untersuchten Verfahren haben gezeigt, dass vor allem *Frame*-Kohärenz, Skalierbarkeit auf Objekt- und NPR-Partikel-Ebene und die Darstellungsgeschwindigkeit wichtige Faktoren einer echtzeifähigen Visualisierung sind.

- **Frame-Kohärenz**
  Kohärenz der Objekte und der Punkte auf deren Oberfläche ist eine wesentliche Anforderung, die erfüllt sein muss, um störendes Rauschen oder ein Auftreten des *Shower Door*-Effektes zu verhindern.

- **Skalierbarkeit**
  Punkte müssen unabhängig von dem Abstand zwischen Betrachter und Objekt gleich groß gezeichnet werden. Bei einem geringeren Abstand des Objektes zum Betrachter sollten mehr Punkte sichtbar sein als bei einer größeren Distanz. Folglich dürfen die Punkte nicht wie das Objekt beim Skalieren vergrößert oder verkleinert werden, sondern müssen sich in ihrer Anzahl verändern.

- **Geschwindigkeit**
  Programmierbare Graphik-Hardware ermöglicht die beschleunigte Umsetzung von Graphikberechnungen und demzufolge eine schnellere Visualisierung der Oberflächenmodelle über *Stippling*. Ziel ist es, eine Visualisierung möglichst ohne jegliche Vorberechnungen auf der Objektgeometrie zu realisieren und so eine schnellere Darstellung zu erreichen.

3.1.2 Wesentliche Anforderungen an eine Stippling-Illustration

Die charakteristischen Merkmale einer *Stippling*-Illustration lassen sich aus den vorher untersuchten Verfahren und der in Abschnitt 2.1.2 vorgestellten, allgemeinen Definition des *Stipplings* entnehmen. Im Folgenden werden nun die speziell für diese Arbeit relevanten Anforderungen festgelegt.

- **konstante Punktgröße**
  Es wird von einer konstanten Größe der Punkte auf der Objektoberfläche ausgegangen. Bei der Verwendung variierender Punktgrößen kann der Eindruck unterschiedlicher Oberflächeneigenschaften erzeugt werden (siehe Abschnitt 2.1.2). Die Punkte werden nur zur Darstellung der Objektform genutzt und dürfen daher keine Informationen des Objektes übermitteln, die nicht vorhanden sind und nur aufgrund der Punktgröße entstehen.

- **äquidistante Punktabstände**
  Die Punkte müssen zufällig aber mit gleichmäßigen Abständen zueinander

platziert werden. Dunkle Regionen erhalten mehr Punkte und der Abstand zwischen diesen wird geringer als in helleren Regionen. Die Punkte dienen nur der Visualisierung des Objektes. Es dürfen demnach keine störenden Muster durch unpräzise Verteilung sichtbar werden.

- **Punktverteilung**
  Stippling-Punkte müssen so platziert werden, dass sie die Objektform und Schattierung auf der Oberfläche betonen. Die Position der Punkte ist somit einerseits abhängig von der Beleuchtung und andererseits von den Eigenschaften der Objektform.

3.1.3 Zusammenfassung und Schlussfolgerung

Basierend auf den bereits untersuchten Ansätzen in Abschnitt 2.2 soll die Visualisierung des Oberflächenmodells über *Stippling* entworfen werden, die die vorgestellten Anforderungen integriert. Die bisher diskutierten Ansätze erfüllen alle auf unterschiedliche Weise die Anforderungen an eine *Stippling*-Illustration, wobei allerdings basierend auf den genannten Bedingungen zwischen den einzelnen Verfahren unterschieden werden muss.

In Abschnitt 2.2.3 wurde ein Volumen-Rendering Verfahren vorgestellt, welches die *Stippling*-Generierung integriert. Dieses Verfahren kann nur für die Visualisierung bestimmter Regionen verwendet werden, da die Positionierung der Punkte anhand der Stärke des Gradienten erfolgt. Eine *Stippling*-Darstellung der Grenzregionen zwischen benachbarten Weichteilen ist beispielsweise nicht möglich, da der Gradient zu gering ist.

Bildbasierte Verfahren generieren einzelne Bilder unabhängig voneinander und erfüllen demnach nicht die für eine interaktive Darstellung wesentliche Anforderung der *Frame*-Kohärenz.

Bei den objektbasierten und den texturbasierten Verfahren sind die Punkte beziehungsweise die Texturen direkt mit dem Objekt verbunden. Die Punkte behalten daher in einer Sequenz aufeinander folgender Bilder ihre Position bezüglich der Objektoberfläche. Ferner ist in diesem Verfahren, wie auch bei den texturbasierten Methoden, die Skalierbarkeit speziell auf die Verwendung von NPR-Primitiven abgestimmt. Objektbasierte Techniken benötigen für die Realisierung der variierenden Punktanzahl Vorverarbeitungsschritte und sind auf die Eckpunktanzahl im Objektgitter beschränkt. Die Verwendung von Texturen hingegen ermöglicht die Darstellung beliebig vieler Punkte unabhängig von dem Aufbau des Objektgitters. Beide Techniken erfüllen die Anforderungen an Skalierbarkeit und *Frame*-Kohärenz, wobei die texturbasierten Methoden ohne vorherige Berechnungen auf der Objektgeometrie auskommen und durch einen

hardwaregestützten Texturzugriff beschleunigt werden können. *Stippling*-Texturen verwirklichen folglich die genannten Anforderungen an interaktive Darstellungen. Der Nachteil einer Texturierung, im Gegensatz zum objektbasierten Verfahren, sind Verzerrungen wie Stauchung oder Streckung, die jedoch durch die Wahl der Abbildungsfunktion sowie durch die Parametrisierung des Objektes verringert werden können.

Im folgenden Abschnitt wird ein Konzept für die Abbildung von 2D-*Stippling*-Texturen auf 3D-Modelle entworfen. Hier wird vor allem das Konzept der *Polycubes* aufgegriffen und modifiziert, um Texturverzerrungen zu verhindern.

3.2 Texturabbildung für 3D-Oberflächenmodelle

Die Texturabbildung auf das 3D-Objekt wird mittels Zwei-Phasen-Texturprojektion realisiert, da die verwendeten Modelle meiste eine sehr komplexe Geometrie besitzen. Als Hilfsobjekte können Kugel, Zylinder oder Würfel verwendet werden. Für die Abbildung auf eines dieser Zwischenobjekte ist es wichtig eine Funktion zu bestimmen, die mögliche Verzerrungen verhindert beziehungsweise minimiert.

Der Würfel hat für die im Rahmen dieser Arbeit benötigten Texturabbildung im Gegensatz zu einer Kugel oder einem Zylinder einige Vorteile. Eine rechteckige 2D-Textur kann konform auf eine Würfelseite abgebildet werden, da die Eckpunkte der Textur bei der Abbildung genau den Eckpunkten der jeweiligen Würfelseite entsprechen. Ein Würfel wird als Hilfsobjekt bevorzugt, da durch seine Geometrie das Konzept des *Polycube*-Verfahrens möglich ist. Hierbei wird die Texturabbildung auf mehrere Würfel verteilt und somit die Abbildung der Textur auf das Oberflächenmodell in viele kleine Abbildungen aufgeteilt und dadurch die Texturverzerrungen minimiert. Da sich ein Objekt problemlos in mehrere Würfel (*Polycube*) anstatt in Kugeln oder Zylinder unterteilen lässt, wird für die Texturabbildung die Technik des *Cube Mappings* verwendet.

3.2.1 Cube Mapping

Bei einem *Cube Mapping* wird die Textur zuerst auf einen Würfel und dann auf das Objekt abgebildet. Hierfür wird üblicherweise eine *Cube Map* benutzt, deren Texturen jeweils eine Würfelseite eines achsenparallelen Würfels repräsentieren (siehe Seite 26).

Die Textur auf dem Würfel ist in einem dreidimensionalen Raum mit (s, t, r) Koordinaten definiert. Der Texturzugriff erfolgt dann mit einer Menge von dreidimensionalen Texturkoordinaten (3D-Vektoren). Diese Vektoren sind beim

Cube Mapping Richtungsvektoren ausgehend vom Zentrum des Würfels. Für die Abbildung von dem Würfel auf das Zielobjekt kann eine der vorher in Abbildung 2.10 dargestellten Techniken verwendet werden.

Cube Mapping mit einem 2D-Texturbild

Bei der Verwendung von *Stippling*-Texturen erhält jede Würfelseite die gleiche Textur. Das *Cube Mapping* kann demzufolge abgeändert werden und allein mit einem rechteckigen 2D-Texturbild für einen Würfel realisiert werden. Anstatt einer *Cube Map* bestehend aus sechs Bildern, wird nur noch ein rechteckiges Bild pro Würfel abgespeichert. Der Texturzugriff wird folglich vereinfacht, da immer das gleiche Texturbild verwendet wird, unabhängig davon, welche Seite von dem betrachteten Vektor geschnitten wurde.

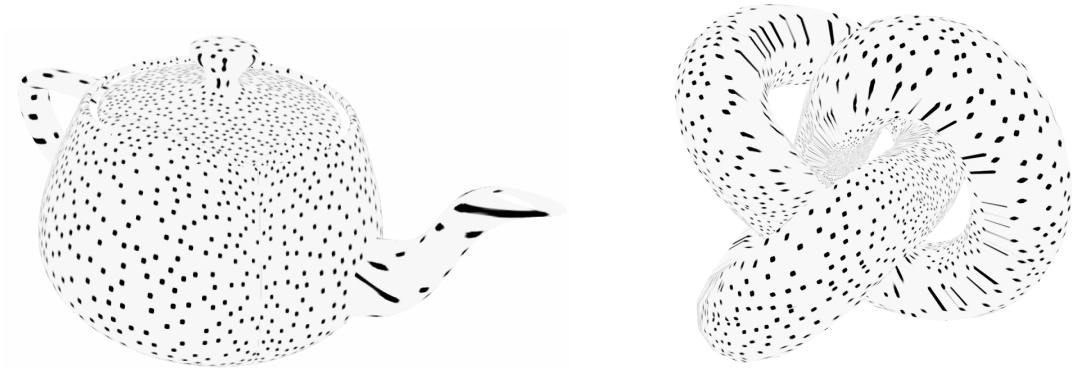
In den folgenden Abbildungen ist meist der *Teapot* dargestellt, da sich an diesem Modell die Anforderungen und Probleme der Texturabbildung gut erläutern lassen. Die Textur aus Abbildung 3.1 (a) wurde nach dem Prinzip des *Cube Mappings* auf das Objekt in Abbildung 3.1 (b) abgebildet. Das Objekt und der Würfel liegen bezogen auf den Objektraum im Koordinatenursprung. Somit kann für jeden Oberflächenpunkt der Richtungsvektor für den Texturzugriff verwendet werden (siehe Abbildung 3.1 (b)).

(a) (b)

Abbildung 3.1: Das *Cube Mapping* mit einer 2D-Textur. (a) Ein einzelnes Texturbild wurde für einen Würfel abgespeichert und (b) auf das Objekt abgebildet. Die Würfelkanten sind auf dem Objekt angedeutet, um das *Mapping* zu verdeutlichen.

Vor allem an der Tülle und an dem Henkel des *Teapots* sind die bei der Texturabbildung mit einem Würfel entstehenden Probleme sichtbar. Es erscheint, als wäre die Textur einer Würfelseite über das Objekt gespannt worden. An diesen Bereichen liegen mehrere Oberflächenpunkte hintereinander auf dem gleichen Vektor und erhalten demnach denselben Texel. Auf den zweidimensionalen Fall übertragen, enstehen dort Probleme, wo die Punkte beispielsweise eine gleiche y-Koordinaten

aber unterschiedliche x-Koordinaten haben. Diese Punkte bekommen alle denselben Texel und es entstehen langgezogene Punkte. In Abbildung 3.2 sind Objekte mit einer *Stippling*-Textur dargestellt. Die beschriebenen Texturverzerrungen werden vor allem sichtbar je komplexer die Oberfläche des Objektes ist (siehe Abbildung 3.2 (b)). Aus den einzelnen Punkten entstehen ovale Punkte oder sogar Linien beziehungsweise Liniensegmente.



(a) (b)

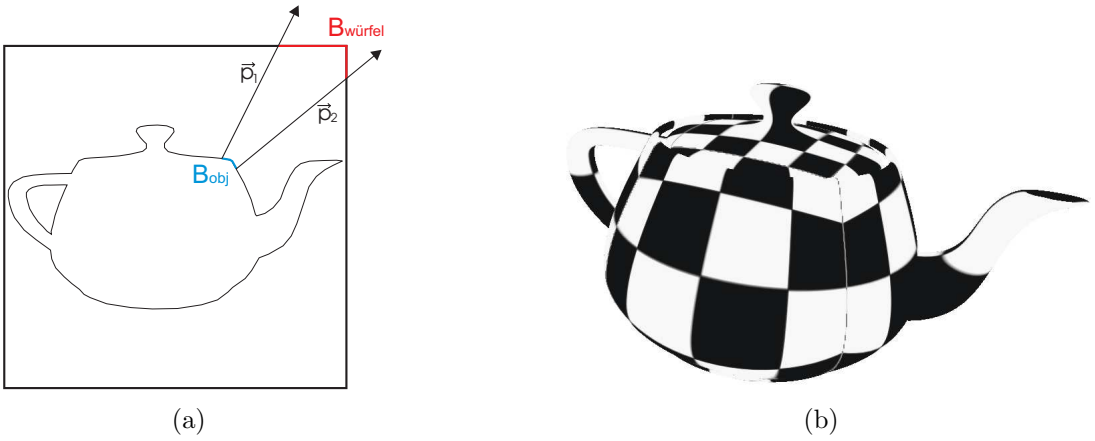
Abbildung 3.2: *Cube Mapping* mit einer *Stippling*-Textur. (a) Der *Teapot* und auch (b) der Torus weisen Verzerrungen der Textur auf, die zu Verzerrungen der Punkte führen. Je komplexer die Geometrie des Objektes, desto mehr wird die Textur gestaucht oder gestreckt.

Durch die Verwendung des Normalen-Vektors eines Punktes werden diese Verzerrungen minimiert. Der Vektor ist für jeden Punkt individuell und die entstehenden Probleme, durch mehrerer Punkte entlang eines Vektors, werden verringert.

Die schematische 2D-Darstellung in Abbildung 3.3 (a) zeigt die Ursache der Stauchung anhand zweier benachbarter Punkte. Der Bereich $B_{würfel}$ wird auf den Bereich B_{obj} abgebildet. Da der Bereich des Würfels größer ist, findet keine flächentreue Abbildung statt und die Textur wird auf dem Objekt gestaucht dargestellt (siehe Abschnitt 2.3.2). Analog wird die Textur auf dem Objekt gestreckt, wenn $B_{würfel} < B_{obj}$. Das Verhältnis $B_{würfel} : B_{obj}$ sollte daher möglichst $1 : 1$ sein, um sich einer flächentreuen Abbildung anzunähern und somit Texturverzerrungen auf dem Objekt zu vermeiden. Weiterhin treten auch Probleme bezüglich der Konformität auf. In Abbildung 3.3 (b) ist der *Teapot* mit einer Schachbrettmuster-Textur dargestellt. Die Form der Quadrate verändert sich entsprechend der Objektform und entspricht somit nicht mehr einer konformen Abbildung.

Die erläuterten Probleme können minimiert werden, indem der Abstand zwischen Objekt und Würfel verringert wird. Das Verhältnis $B_{würfel} : B_{obj}$ nähert sich dann

dem gewünschten 1 : 1 Verhältnis an. Da bei dem traditionellen *Cube Mapping* der Würfel allerdings das Objekt umgibt, ist der Abstand der Würfelseiten zum Objekt durch das Ausmaß des Objektes begrenzt. Das Konzept des *Polycube Mappings* hebt diese Beschränkung auf. In Abschnitt 2.3.2 wurde bereits die Idee diese Verfahrens beschrieben. *Cube Mapping* wird auf kleineren Teilen des Objektes ausgeführt, wodurch Verzerrungen minimiert werden.



(a) (b)

Abbildung 3.3: Es treten Stauchungen und Streckungen der Textur auf dem Objekt auf, wenn $B_{würfel} \neq B_{obj}$. (a) $\vec{p_1}$ und $\vec{p_2}$ sind hierbei die jeweiligen Normalen-Vektoren zweier Punkte, die zur Ermittlung der korrespondierenden Texel verwendet werden. Stauchungen entstehen wenn $B_{würfel} > B_{obj}$. (b) Der *Teapot* mit einer Schachbrettmuster-Textur zur Verdeutlichung der Verzerrungen. Die Textur wurde nicht konform auf das Objekt abgebildet.

3.2.2 Modifizierter Polycube-Ansatz

Cube Mapping mit einem Würfel, der das Objekt umgibt, führt bei Objekten mit komplexer Geometrie zu Problemen bezüglich einer flächentreuen und konformen Abbildung. Die Verzerrungen lassen sich durch Unterteilen des Objektes in mehrere Würfel (*Polycube*) minimieren. Abbildung 2.14 (b) zeigt den *Polycube* für das *Bunny*-Modell. Da dort die Position jeder einzelnen Textur wichtig ist, um das Modell korrekt mit der Textur zu versehen, muss das Objekt in Würfel unterteilt werden. Die *Stippling*-Texturen sind dagegen nicht an spezielle Objektbereiche gebunden. Die Umformung der Objektgeometrie in einen *Polycube* kann durch eine Einteilung des Objektraumes in Würfel, wie in Abbildung 3.4 (a) dargestellt, ersetzt werden. Ferner können alle Berechnungen zur Ermittlung der Texturkoordinaten an einem Würfel erfolgen, wodurch dieses Verfahren weiter vereinfacht wird.

Abbildung 3.4 (b) zeigt einen Ausschnitt eines Objektes mit zwei verschiedenen Würfelgrößen. Erfolgt eine Abbildung bezogen auf den äußeren großen Würfel,

entstehen Stauchungen auf dem Objekt, da die abgebildete Strecke zwischen $\vec{p_1}$ und $\vec{p_2}$ größer ist als die Entfernung der Punkte auf dem Objekt. Eine Verringerung der Würfelgröße führt dazu, dass die Vektoren auf die Würfelseite des kleinen Würfels zugreifen. Der Abstand der Schnittpunkte beider Vektoren auf dieser Würfelseite ist wesentlich geringer. Die Unterteilung des Objektraumes in kleinere Würfel ermöglicht eine Annäherung an eine flächentreue Abbildung, da die Entfernung der jeweiligen Punkte zur Würfelseite verringert wird und somit Stauchungen und Streckungen minimiert werden können.

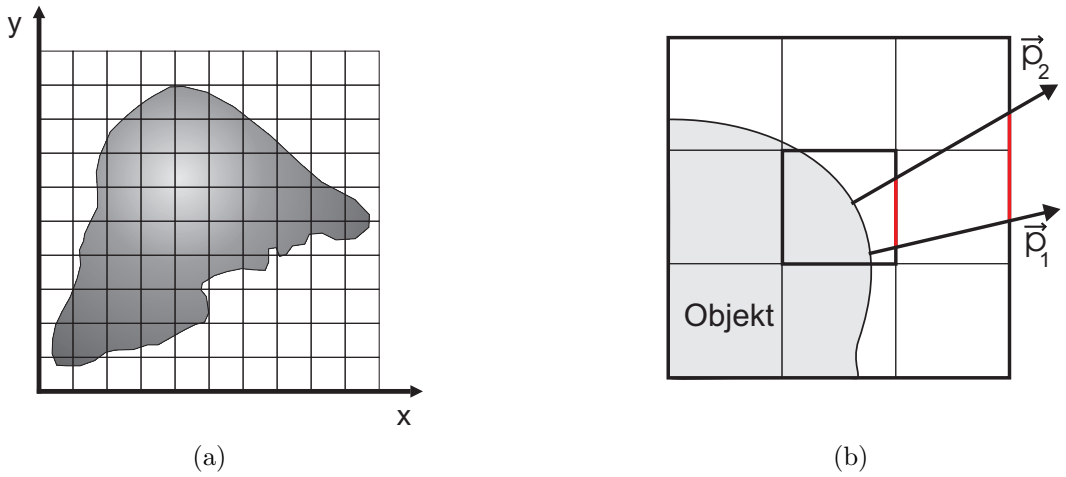


Abbildung 3.4: Der Texturzugriff unter Verwendung des Normalen-Vektors. (a) Die vorgesehene Unterteilung des Objektraumes. Jedes Quadrat entspricht im 3D-Raum einem Würfel. (b) Die Abbildung nähert sich einer flächentreuen Abbildung an, wenn der Abstand des Objektes zu der Würfelseite minimiert wird.

Die Texturabbildungen der einzelnen Würfel auf das Objekt orientieren sich an dem *Cube Mapping*-Prinzip. Es wird allerdings für einen Würfel nur ein Bild abgespeichert und die Seiten sind Ebenen, die durch einen Punkt und einen Normalen-Vektor aufgespannt werden. Entsprechend der Würfelseiten beim *Cube Mapping* werden die Ebenen achsenparallel im Raum positioniert. Die Umsetzung der Würfelseiten als Ebenen erleichtert die Unterteilung des Raumes in Würfel, da die Texturkoordinaten einfach auf das Intervall der Würfelgrenzen abgebildet werden können.

Unterteilung des Objektraumes in Würfel

Für eine Unterteilung des Objektraumes in Würfel wird angenommen, dass dieser aus gleich großen Würfeln besteht. Eine adaptive Unterteilung in unterschiedlich große Würfel würde zu Verzerrungen der Textur führen, da nur eine Texturgröße verwendet wird. Die Punkte wären unterschiedlich groß, da die Texturen und

folglich auch die Punkte auf großen Würfeln gestreckt und auf kleinen gestaucht werden. Variierende Würfelgrößen benötigen entsprechende Texturen. Weiterhin müssten zusätzliche Informationen in der Attributliste der *Vertices* gespeichert werden, um für jeden die korrespondierende Würfelgröße für die Berechnungen in den *Shader*-Programmen zu erhalten. Da mit einer einheitlich großen Textur gearbeitet wird und keine Nachbarschaftsinformationen ohne die gewünschte Vorverarbeitung der Objektgeometrie vorliegen, werden gleich große Würfel betrachtet.

Da die Textur nicht an spezielle Bereiche des Objektes gebunden ist, können die Berechnungen für die Texturabbildung eines Punktes in einem beliebigen Würfel im Raum anhand eines definierten Referenzwürfels ausgeführt werden. Zunächst wird, basierend auf der Helligkeit des Punktes, eine Textur ausgewählt, die der Würfel erhält und anschließend erfolgt die Ermittlung der Texturkoordinate. Dieser Referenzwürfel ist im Koordinatenursprung mit $W_{min} = (0, 0, 0)$ und einem beliebigen W_{max} definiert und besitzt achsenparallele Seiten. Wird nun für einen Punkt die Texturkoordinate ermittelt, muss dessen Position bezüglich des Referenzwürfels betrachtet werden, um den korrespondierenden Texel ermitteln zu können. Seine Koordinaten werden folglich mit der Gleichung 3.1 für die Texturabbildung in den Würfel im Koordinatenursprung transformiert. Dort kann dann der Schnittpunkt des zu dem Punkt gehörenden Vektors mit dem Würfel berechnet und somit der Texel bestimmt werden.

$$Punkt\ im\ Würfel\ :\qquad P_{würfel} = \frac{P_{orginal} - W_{min}}{W_{max} - W_{min}} \tag{3.1}$$

Die Transformation des Punktes bewirkt eine Modulo-Rechnung der Koordinaten mit der Würfelgröße und erzeugt so eine Unterteilung des Raumes in Würfel. Die Position des Punktes in Bezug auf den Würfel ist der ganzzahlige Anteil der jeweiligen Vektorkomponenten von $P_{würfel}$. Für die Berechnung der Texturkoordinate eines Punktes wird im folgenden Abschnitt immer von dieser Position des Punktes im Referenzwürfel ausgegangen.

Texturverzerrungen können durch die Verwendung vieler Würfel bei Objekten mit komplexer Geometrie der Oberfläche reduziert beziehungsweise auf kleinere Bereiche des Objektes begrenzt werden. Die Würfelgröße darf allerdings nicht zu klein gewählt werden, da mit kleineren Würfelseiten die Textur minimiert werden muss und demzufolge weniger und kleinere Punkte auf dieser Textur und auf dem Objekt gezeichnet werden können. Analog treten Probleme auf, wenn die Textur zu klein für den verwendeten Würfel ist. In diesem Fall wird die Textur auf die Größe der jeweiligen Würfelseite angepasst und alle Punkte auf der Textur werden skaliert. Diese Probleme können, durch eine auf die Würfelgröße abgestimmte

Textur verringert werden. Faktoren wie die Texturgröße, die Punktanzahl und die Punktgröße auf der Textur müssen bei der Generierung entsprechend der verwendeten Würfelgröße gewählt werden.

3.2.3 Vorgehensweise bei der Texturabbildung

Die Bestimmung der Texturkoordinate für einen Oberflächenpunkt P erfolgt mit einem Vektor $\vec{p}$, der das Hilfsobjekt in einem Punkt schneidet. Demzufolge muss mit dem Vektor einerseits die zu schneidende Würfelseite und andererseits die (s, t) Koordinaten der jeweiligen 2D-Textur auf dieser Seite beziehungsweise Ebene berechnet werden. Alle Gleichungen gelten unter der Annahme, dass der verwendete Vektor $\vec{p}$ sich im selben Koordinatensystem wie das Hilsfobjekt befindet.

Ermittlung der Würfelseite

Zunächst wird die Achse bestimmt, für die eine Ebene aufgespannt und die Schnittpunktberechnung mit dem Vektor erfolgen muss. Dies geschieht wie bei dem traditionellen *Cube Mapping*. Alle Komponenten des Vektors $\vec{p}$ werden bezüglich ihrer absoluten Werte verglichen. Die größte Komponente gibt die Hauptrichtung des Vektors und somit auch die zu betrachtende Würfelseite beziehungsweise Ebene an. Falls $p_x = p_y = p_z$ gilt, wird p_x als Hauptrichtung angenommen. Bezogen auf die kleinen Würfel, gibt die Richtung von $\vec{p}$ nicht immer die Seite an, die zuerst geschnitten wird.

In Abbildung 3.5 (a) wird anhand der Vektoren $\vec{p_1}$ und $\vec{p_2}$ ein Texturzugriff benachbarter Punkte dargestellt. Würde für jeden Vektor die Seite betrachtet werden, die vom ihm zuerst geschnitten wird, dann wäre für $\vec{p_1}$ der gesuchte Schnittpunkt x_1 und für $\vec{p_2}$ der Punkt y_1. In diesem Fall treten allerdings Stauchungen auf, da der Bereich B_2 wesentlich größer ist, als der Objektbereich auf den er abgebildet wird. Basierend auf der Hauptkomponente wird von beiden Vektoren eine x-Seite geschnitten und der Bereich B_1 auf das Objekt abgebildet. Der Vektor $\vec{p_2}$ schneidet demnach ebenfalls die x-Seite. Die Abbildung nähert sich somit einer flächentreuen an. Der ermittelte Schnittpunkt x_2 von $\vec{p_2}$ befindet sich außerhalb des betrachteten Würfels und muss deshalb noch auf die Würfelgrenzen abgebildet werden. Für eine Objektoberfläche mit einer geringen Abweichung vom Ebenenverlauf (Krümmung) ist die Bestimmung der Würfelseite anhand der Vektor-Richtung vorteilhafter.

Abbildung 3.5 (b) zeigt im Gegensatz dazu ein Beispiel einer stärker gekrümmten Oberfläche, bei der die kleineren Würfel für die Texturierung hilfreich sind und

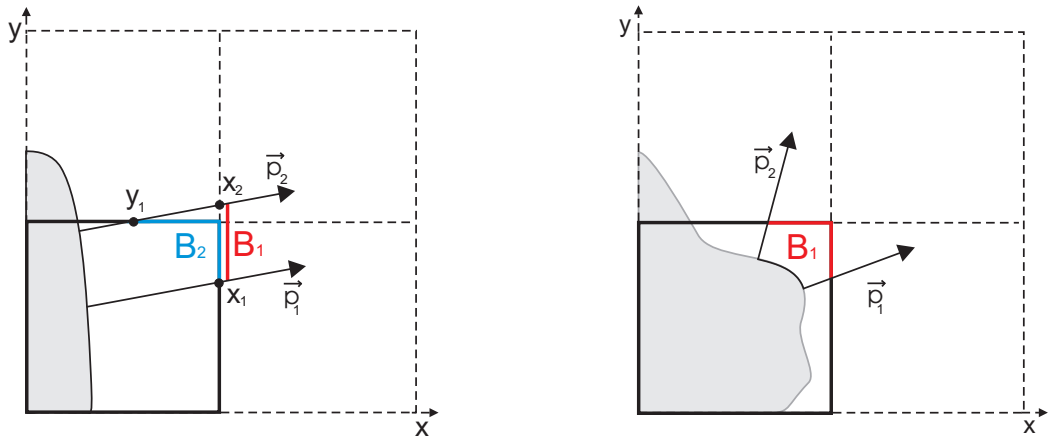


Abbildung 3.5: Anhand der Hauptrichtung der Vektoren $\vec{p_1}$ und $\vec{p_2}$ wird die zu schneidende Seite bestimmt. (a) Für Objekte mit geringer Krümmung erfolgt daher eine kontinuierliche Texturierung und Verringerung möglicher Verzerrungen. (b) Bei Objekten mit stärkerer Krümmung wird durch die Richtung die Seite angegeben, die eine möglichst flächentreue Abbildung ermöglicht.

genutzt werden. Da sich die Richtungen beider Vektoren stark unterscheiden, wird durch die Verwendung eines kleineren Würfels die Texturabbildung verbessert. Vektor $\vec{p_1}$ schneidet die x-Seite und $\vec{p_2}$ die y-Seite. Der Bereich B_1 ist dem Bereich auf der Objektoberfläche ähnlich. Folglich verbessert sich die Abbildung wenn die Würfelseite, wie bisher beim *Cube Mapping*, durch die Richtung des Vektors bestimmt wird.

Berechnung der 2D-Texturkoordinaten

Nachdem die Würfelseite bestimmt wurde, müssen die (s,t) Texturkoordinaten ermittelt werden. Beim *Cube Mapping* werden die restlichen zwei Komponenten des Vektors $\vec{p}$ durch die maximale Vektorkomponente dividiert und somit in das Intervall $[-1.0, 1.0]$ überführt. Der 3D-Vektor wird durch die Verwendung der zwei Komponenten in einen 2D-Vektor umgewandelt, dessen Elemente nun noch auf das Intervall der Texturkoordinaten $[0.0, 1.0]$ abgebildet werden müssen, um in Form einer (s,t) Koordinate auf die Textur zugreifen zu können. In Abbildung 3.6 sind diese Schritte veranschaulicht. Der Vektor $\vec{p}$ schneidet in Abbildung 3.6 (a) im Punkt A den Einheitswürfel. Hierbei handelt es sich um die positive x-Seite des Würfels. Zur Ermittlung der (s,t) Texturkoordinaten muss r von $\vec{p}$ ermittelt werden (siehe Abbildung 3.6 (b)). Die Koordinate s errechnet sich durch die Gleichung $s = r_y/r_x$. Da s nun im Intervall $[-1.0, 1.0]$ liegen kann, wird auf das Ergebnis noch 1 addiert, um auf das Intervall $[0.0, 2.0]$ abzubilden. Anschließend

wird das Ergebnis durch 2 dividiert, um dann einen Wert zwischen $[0.0, 1.0]$ zu liefern. s, t berechnen sich folglich durch $s = ((r_y/r_x)+1)/2$ und $t = ((r_z/r_x)+1)/2$.

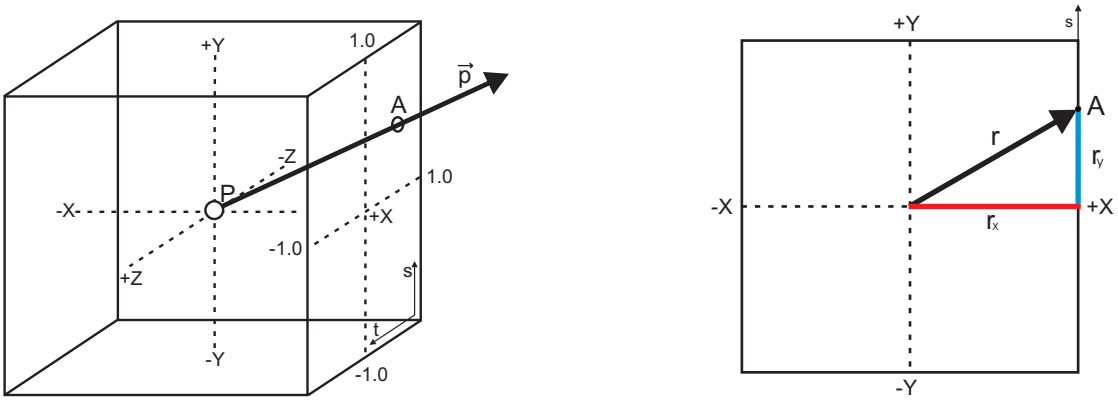


Abbildung 3.6: (a) Beim *Cube Mapping* ist der für einen Punkt P zugehörige Texel der Schnittpunkt A des Vektors $\vec{p}$ mit einer Würfelseite. (b) Für die Berechnung der Texturkoordinate s müssen r, r_y und r_x ermittelt werden. Die Berechnungen für t erfolgen dann analog.

Diese Berechnungen werden nun auf die Verwendung von Ebenen angepasst. Der Vektor $\vec{p}$ bestimmt, wie bereits beschrieben, die Seite beziehungsweise die Achse, für die eine Ebene aufgespannt wird. Gleichung 3.2 definiert die Ebene mit einem Punkt p, dem Normalen-Vektor $\vec{n_e}$ und dem Abstand d zum Koordinatenursprung. Der Normalen-Vektor einer Ebene ist ein Einheitsvektor, da die Ebenen angelehnt an den Würfel achsenparallel sind. Für die Schnittpunktberechnung des Vektors mit der Ebene wird eine Geradengleichung aufgestellt, wobei $\vec{p}$ der von dem betrachteten Punkt P ausgehende Vektor und p ein Punkte auf dieser Geraden ist.

$$\begin{aligned} &\textit{Ebenengleichung} \quad : \quad d = p * \vec{n_e} \\ &\textit{Geradengleichung} \quad : \quad p = P + k * (\vec{p}) \quad \Rightarrow \quad d = (P + k(\vec{p})) * \vec{n_e} \end{aligned} \tag{3.2}$$

Der Schnittpunkt entspricht in Abbildung 3.6 dem Punkt A, der auf der Geraden und in der Ebene liegt. Folglich kann dieses Gleichungssystem durch das Einsetzen der Geradengleichung in die Ebenengleichung für den Punkt p gelöst werden (siehe Gleichung 3.2). Die resultierende Gleichung wird nun nach k umgestellt, um den Schnittpunkt zu finden.

Alle anderen Variablen sind bekannt oder können mit der Ebenengleichung berechnet werden. Die Variable d ist das Produkt eines beliebigen Punktes auf

der Ebene und dem Normalen-Vektor. Für die Schnittpunktberechnung mit einer positiven Würfelseite wird hierbei der Punkt W_{max} und bei einer negativen Seite W_{min} eingesetzt (siehe Abbildung 3.7). Die Koordinaten für den Schnittpunkt A ergeben sich anschließend aus der Geradengleichung 3.3 mit dem zuvor berechneten k.

$$k = \frac{d - P * \vec{n_e}}{\vec{p} * \vec{n_e}} \qquad \Rightarrow \qquad p = P + (\frac{d - P * \vec{n_e}}{\vec{p} * \vec{n_e}}) * (\vec{p}) \qquad (3.3)$$

Der Schnittpunkt ist noch nicht zwingend innerhalb der Würfelgrenzen (siehe Abbildung 3.5 (a)). Demnach muss A zusätzlich noch auf die Intervallgrenzen $[W_{min}, W_{max}]$ abgebildet werden, um auf die Textur zugreifen zu können. Die (s,t) Koordinaten werden aus den zwei Komponenten von A berechnet, die den Achsen der Texturkoordinaten entsprechen (siehe Abbildung 2.13). Für die in Abbildung 3.7 dargestellte Texturabbildung, in der die positive x-Seite geschnitten wird, sind demzufolge $s = (y_S - y_{Wmin}) \,/\, (y_{Wmax} - y_{Wmin})$ und $t = (z_S - z_{Wmin}) \,/\, (z_{Wmax} - z_{Wmin})$. Folglich werden alle Komponenten (p_x, p_y, p_z) des Vektors $\vec{p}$ für die Schnittpunktberechnung betrachtet. Für die Ermittlung der (s,t) Koordinaten fällt, wie beim *Cube Mapping*, die Komponente weg, die die zu schneidende Seite angibt.

3.2.4 Schlussfolgerung

Die Texturabbildung für ein Objekt setzt sich aus dem Konzept des *Cube Mappings* und des *Polycube*-Ansatzes zusammen. Folgende Vorteile und Möglichkeiten lassen sich daher umsetzen und ausnutzen:

- Ein rechteckiges 2D-Bild kann als Textur genutzt werden.
- Die Textur muss vorher nicht verzerrt beziehungsweise dem Hilfsobjekt angepasst werden.
- Eine Textur wird konform (winkeltreu) auf den Würfel abgebildet.
- Die Vorteile des *Polycube*-Konzeptes sind integrierbar und folglich werden Texturverzerrungen minimiert und die Abbildung nähert sich einer flächentreuen und konformen Abbildung an (siehe Abschnitt 2.3.2 auf Seite 27).

Alle Berechnungen für einen Oberflächenpunkt können anhand eines Referenzwürfels ausgeführt werden. Dieser Würfel besitzt, abhängig von dem für den Punkt zu repräsentierenden Helligkeit, eine *Stippling*-Textur. Alle Texturen sowie deren Generierung und Speicherung werden im folgenden Abschnitt erläutert. Die Texturkoordinate für einen Oberflächenpunkt berechnet sich aus dem Schnittpunkt

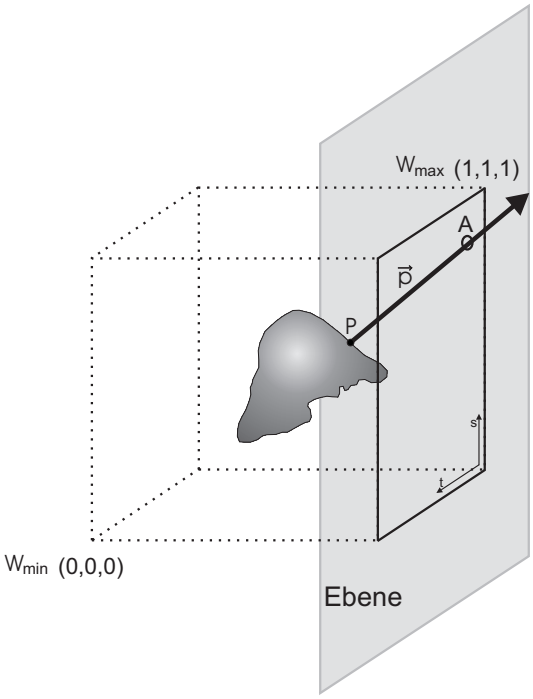


Abbildung 3.7: Die Texturabbildung am Beispiel eines Würfels, wobei die einzelnen Würfelseiten durch Ebenen repräsentiert werden und A der Schnittpunkt mit der Ebene ist, der im Intervall $[W_{min}, W_{max}]$ liegt.

der Normalen mit der entsprechenden Seite. Weiterhin wird die Würfelseite basierend auf der Richtung der Normalen bestimmt, wodurch bei weniger gekrümmten Oberflächen die Unterteilung des Raumes nicht beachtet wird und somit auch dort eine verzerrungsfreiere Abbildung möglich ist (siehe Abbildung 3.5 (a)).

3.3 Stippling-Texturen

Die medizinischen Oberflächenmodelle werden mittels Texturen über *Stippling* dargestellt. Basierend auf der beschriebenen Abbildungstechnik ist es erforderlich zweidimensionale, quadratische Texturen mit *Stippling*-Punkten zu erstellen. Der Aufbau der Texturen orientiert sich an dem Prinzip der TAM, um unter anderem die Skalierbarkeit und *Frame*-Kohärenz der Punkte entsprechend der verlangten Anforderungen aus Abschnitt 3.1 zu ermöglichen. Ferner ist es erforderlich, eine Verteilung der Punkte auf der Textur zu entwerfen, die möglichst wenig oder keine störenden Muster auf dem Objekt entstehen lässt.

Eine *Stippling*-Textur besteht aus einem weißen Hintergrund mit unterschiedlich vielen schwarzen Punkten. Die einzelnen Punkte entsprechen Kreisen, die durch ihren Mittelpunkt und einen Radius definiert sind.

3.3.1 Generierung verschiedener Helligkeitsstufen

Die *Stippling*-Texturen müssen verschiedene Helligkeitstufen repräsentieren. Bei den *Stippling*-Illustrationen werden diese Abstufungen durch die Variation der Punktanzahl und der Abstände zwischen den Punkten erreicht. Allgemein entspricht eine zunehmende Anzahl von Punkten in einem Gebiet einem dunkleren Farbton. Zur Repräsentation verschiedener Helligkeitsstufen werden demzufolge unterschiedlich stark gepunktete Texturen erzeugt. Da der Aufbau der einzelnen Texturen nach dem TAM-Prinzip erfolgt, ist für die verschiedenen Auflösungs- und Helligkeitstufen eine bestimmte Reihenfolge bei der Generierung zu beachten. Eine TAM wird beginnend mit der hellsten Textur erstellt. Da es in einer *Stippling*-Illustration auch Gebiete gibt, die keine Punkte haben, muss die hellste Stufe eine weiße Textur sein, die keine Punkte besitzt. Ausgehend von dieser weißen Grundtextur werden je nach Farbton die erforderliche Menge an Punkten hinzugefügt.

Die erste Textur wird mit $Textur_0$ bezeichnet und ist ausschließlich weiß. Alle weiteren Texturen erhalten eine fortlaufende Nummerierung von 1 bis n. $Textur_n$ bezeichnet die dunkelste Textur mit der maximalen Anzahl an Punkten. Eine Textur wird aufbauend auf der vorhergehenden Textur generiert. Sie besitzt demzufolge neben den neu hinzukommenden Punkten auch alle Punkte ihrer Vorgängertextur und repräsentiert somit einen dunkleren Farbton. Die Menge der neu hinzukommenden Punkte ist bei allen Texturen konstant und lässt sich berechnen, indem die Anzahl der Punkte der $Textur_n$ durch die Anzahl der verwendeten Texturen geteilt wird. $P_{Offset} = P_{max}/n$. P_{max} entspricht hierbei der Punktanzahl von $Textur_n$, n der Gesamtanzahl der Texturen und P_{Offset} der Menge zusätzlicher Punkte für eine Textur. Folglich berechnet sich die Punktanzahl einer Textur mit der Gleichung 3.4.

$$Punkte\ einer\ Textur: \ \forall\, i \in \{0...n\} : P_{Textur_{i+1}} = P_{Textur_i} + P_{Offset} \quad (3.4)$$

Die Definition der maximalen Punktanzahl P_{max} von $Textur_n$ kann entweder der maximal möglichen Punkte auf der Textur oder einem vom Benutzer individuell festgelegten Wert entsprechen. Erstere Variante berechnet diesen Wert anhand der Flächeninhalte von Textur und Punkte: $P_{max} = A_{Grundfläche}/A_{Punkt}$, wobei davon

ausgegangen wird, dass bei der $Textur_n$ die Punkte dicht aneinander gereiht sind und die Punkte sich nur berühren und nicht überlappen (siehe Abbildung 3.8). Die verwendete Punktanzahl entspricht dann dem ganzzahligen Wert von P_{max}.

In Abbildung 3.8 ist die $Textur_n$ unter Verwendung dieses Ansatzes dargestellt. Das Ausnutzen der gesamten Texturfläche führt zu einer Punktanordnung, die sichtbare Muster erzeugt und diese auf das Objekt übertragen könnte. Wird die Möglichkeit der überlappenden Punkte hinzugezogen, ist die dunkelste Textur mit maximaler Anzahl an Punkten schwarz. Die Helligkeit einer Textur muss dann in Abhängigkeit der Flächeninhalte der Punkte und der Textur berechnet werden. Vorher wird festgelegt wie das Verhältnis von schwarz und weiß auf der Textur sein muss, um eine Helligkeitsstufe zu repräsentieren. Dann werden zu einer Textur so lange Punkte hinzugefügt, bis dieser Wert erreicht ist. Bei beiden Ansätzen ist die dunkelste Textur vordefiniert durch die maximal mögliche Anzahl an Punkten. Dadurch wird automatisch die Helligkeit der gesamten Darstellung festgelegt.

Es wird ein Ansatz gewählt, bei dem die Punkte sich maximal berühren können und der Benutzer die Menge an Punkten angibt, die die dunkelste Textur besitzen darf. $Textur_n$ ist somit nicht beschränkt auf die maximal mögliche Anzahl. Einerseits ist es möglich die Helligkeit der gesamten *Stippling*-Illustration zu steuern, indem die dunkelste Textur individuell definiert wird. Andererseits können die Punkte auf der Textur, wenn nicht die maximal mögliche Anzahl gewählt wurde, ausgeglichener verteilt und somit störende Muster verhindert werden.

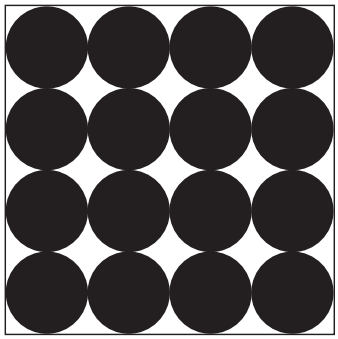


Abbildung 3.8: Der Aufbau einer *Stippling*-Textur mit maximaler Punktanzahl, wenn die Punkte sich höchstens berühren dürfen.

Die Gesamtzahl verwendeter Texturen n wird ebenfalls vom Benutzer festgelegt. Die Texturen repräsentieren bei der Texturabbildung bestimmte Grauwertbereich. Dies entspricht der Abbildung der Texturen auf die Grauwerte, wobei $Textur_0 \rightarrow 255$ weiß und $Textur_n \rightarrow 0$ schwarz zugeordnet wird. Alle dazwischen liegenden Texturen entsprechen äquivalent den Grauwerten zwischen 0 und 255.

Repräsentation in einer 2D-Textur

Da die verschiedenen Texturen aufeinander aufbauen, können sie in einer Textur vereint und somit effizient abgespeichert werden. Das Konzept wird zunächst an einem Beispiel erläutert. In Abbildung 3.9 (a) sind alle Texturen anhand eines Quaders mit Säulen im dreidimensionalen Raum dargestellt. Die Höhe des Quaders repräsentiert die verschiedenen Helligkeitsstufen und die Säulen entsprechen den Punkten der Texturen. Es existieren inklusive der weißen Ausgangstextur sechs Texturen. Mit $Textur_0$ wird die Textur bezeichnet, die keine Punkte besitzt. Alle folgenden Texturen werden mit $1..n$ gekennzeichnet und bekommen jeweils einen weiteren Punkt hinzu. Folglich besitzt eine $Textur_3$ drei Punkte. Der Quader enthält von oben nach unten die einzelnen Texturen, die wiederum verschiedene Farbtöne durch die Anzahl ihrer enthaltenen Punkte widerspiegeln. Wird ein Punkt in einer Textur hinzugefügt, ist er automatisch in allen folgenden Texturen enthalten. Da der erste Punkt ab $Textur_1$ auch in den folgenden dunkleren Texturen enthalten ist, wird diese Säule über die gesamte Höhe des Quaders definiert.

Eine Textur entspricht nun einem horizontalen Schnitt durch den Quader (siehe Abbildung 3.9 (b)). Alle Säulen, die von diesem Schnitt betroffen sind, werden durch einen Punkt auf der Textur dargestellt. Die resultierende Textur entspricht in diesem Beispiel der $Textur_4$ (siehe Abbildung 3.9 (b)), die vier Punkte besitzt.

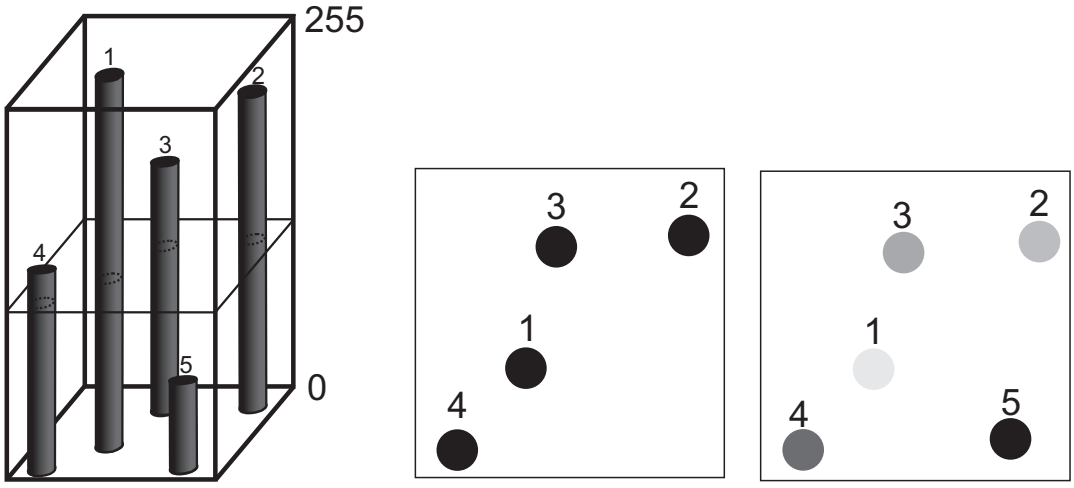


Abbildung 3.9: (a) Die verschiedenen Texturen als 3D-Darstellung, wobei der Boden des Quaders der dunkelsten Textur entspricht. Eine Textur ist das Ergebnis eines horizontalen Schnittes durch den Quader. Alle Säulen die davon betroffen sind, (b) werden als schwarze Punkte auf der Textur abgebildet. (c) Eine 2D-Textur kann alle Texturen repräsentieren, indem die Reihenfolge der Punkte im Grauwert kodiert ist.

Das Konzept kann auf eine 2D-Textur übertragen werden, wenn diese Textur alle Punkte enthält. Der Grauwert der Punkte kodiert dann die Höhe der Säulen aus der 3D-Darstellung. Zunächst wird dafür das Intervall $[0, 255]$ durch die Anzahl der benötigten Texturen n geteilt: $GW_{Offset} = 255/n$. Der Wert GW_{Offset} wird dann jeweils von dem Grauwert der Vorgängertextur subtrahiert. Je dunkler ein Punkt gezeichnet wird, desto später wurde er generiert. Das bedeutet der dunkelste Punkt wird zu der letzten und dunkelsten Textur hinzugefügt. Alle Punkte die einen größeren Grauwert haben, sind ebenfalls in dieser Textur enthalten, da sie früher generiert wurden. Die Reihenfolge der generierten Punkte entspricht demzufolge ihrem Grauwert (siehe Abbildung 3.9 (c)). Eine Textur wird nun anhand eines zulässigen Grauwertes (Schwellwert) aus der abgespeicherten Textur extrahiert. Alle Texel der Textur, die entweder Grauwert 255 haben oder unter dem Schwellwert liegen, werden weiß die anderen schwarz gezeichnet. Folglich lässt sich die Textur aus Abbildung 3.9 (b) mit der Textur in Abbildung 3.9 (c) beschreiben.

3.3.2 Punktverteilung auf einer Textur

Ein charakteristisches Merkmal einer *Stippling*-Illustration ist die Anordnung der Punkte. Sie sind zufällig verteilt, besitzen aber gleichmäßige Abstände zueinander. In computergenerierten Illustrationen werden die Punkte vorwiegend mit dem Algorithmus von Lloyd [1982] zur Generierung eines Voronoi-Diagrammes verteilt. Wie bereits in Abschnitt 2.2.1 erläutert, wird hierbei die gesamte Punktmenge betrachtet und die endgültige Position der *Stippling*-Punkte entsprechend der Schwerpunkte ihrer Voronoi-Regionen bestimmt. Das Ergebnis sind gleichmäßig angeordnete Punkte mit äquidistanten Abständen zueinander. Unter Berücksichtigung der im vorhergehenden Abschnitt beschriebenen Generierung und Speicherung der Texturen lässt sich der Algorithmus von Lloyd [1982] nicht ohne Modifikationen auf dieses Konzept übertragen. Es existieren zwei Möglichkeiten, ein Voronoi-Diagramm für die Punkte zu erstellen und diese dann gleichmäßig auf der Texturfläche zu verteilen.

- **1. Variante:**
  Die Punkte werden nacheinander erzeugt und der Textur hinzugefügt (siehe Abbildung 3.9 (c)). Ein Voronoi-Diagramm könnte somit für jede einzelne Textur aufgestellt werden. Hierbei ist zu beachten, das alle Punkte für die Generierung eines Diagrammes mit einfließen, aber nur die neuen Punkte verschoben werden dürfen. Alle Punkte, die auch in einer vorhergehenden Textur enthalten sind, müssen ihre Position beibehalten, um später *Frame*-Kohärenz zwischen den einzelnen Texturen zu gewährleisten. Werden neue Punkte hinzugefügt, ändern sich die Voronoi-Regionen der bereits vorhandenen Punkte und somit auch die Schwerpunkte. Da die alten Punkte

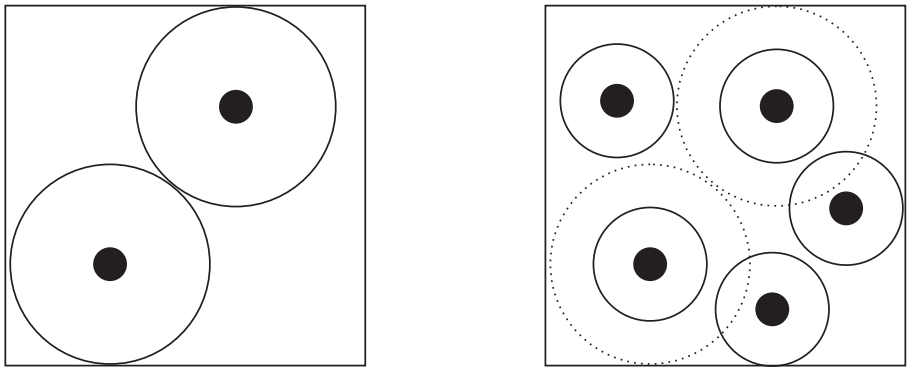
ihre Position aber nicht mehr ändern dürfen, kann die mit der Methode von Lloyd [1982] gewünschte Anordnung der Punkte nicht ausreichend erreicht werden.

- **2. Variante:**
  Eine andere Möglichkeit besteht in der Erzeugung eines Voronoi-Diagrammes basierend auf der letzten Textur, welche alle Punkte enthält und alle Texturen in einer vereint. Dies entspricht dem Konzept von Lloyd [1982], da alle Punkte die Erzeugermenge des Diagrammes darstellen und auch alle ihre Position entsprechend verändern können. Die Verteilung der Punkte gilt allerdings nur für diese Textur. Bei der Betrachtung hellerer Texturen, die aus weniger Punkten aufgebaut sind, werden größere und kleinere Abstände zwischen den einzelnen Punkten sichtbar.

Es existiert eine weitere Methode für die Verteilung der Punkte, die simultan mit der Generierung der Punkte und Texturen ausgeführt werden kann und dabei die *Frame*-Kohärenz zwischen den Texturen bewahrt. Die hierarchische *Poisson Disk*-Verteilung von McCool und Fiume [1992] vereint das Prinzip der *Poisson*-Verteilung und des *Dart-Throwing*-Algorithmus von Cook [1986]. Ein Punkt wird zu einer Menge von Punkten nur dann hinzugefügt, wenn dieser zu allen anderen einen definierten Mindestabstand einhält. Der Abstand wird ausgehend vom Mittelpunkt des *Stippling*-Punktes definiert und als ein kreisrundes Gebiet um diesen Punkt herum betrachtet.

Abbildung 3.10 veranschaulicht dieses Konzept. Es wird innerhalb eines Kreises um den jeweiligen Punkt herum der Abstand zu den anderen getestet. Die Kreise dürfen sich maximal in einem Punkt berühren. Das ist genau dann der Fall, wenn die Entfernung dieser Punkte dem Mindestabstand entspricht. Ein Punkt wird demnach mit einer zufälligen Position generiert und anschließend getestet, ob diese Position zulässig in Abhängigkeit der bereits gezeichneten Punkte ist. Folglich kann sofort entschieden werden, ob der Punkt erlaubt ist oder nicht. Werden anfangs wenige Punkte generiert, ist es erforderlich einen großen Abstand zwischen den Punkten zu definieren, um Clusterbildung zu vermeiden (siehe Abbildung 3.10 (a)). Je mehr Punkte gezeichnet werden, desto geringer muss die Distanz zwischen den Punkten gewählt werden. Es ist daher wichtig, den Abstandswert dynamisch zu gestalten, um sich der variierenden Punktmenge anzupassen (siehe Abbildung 3.10 (b)). Wurde nach einer festgelegten Anzahl an Versuchen kein Ort für den Punkt gefunden, muss der Abstandswert geringfügig heruntergesetzt werden. Die hierarchische *Poisson Disk*-Verteilung passt sich der wachsenden Punktmenge an und kann während der Texturerzeugung ausgeführt werden, ohne die bereits gezeichneten Punkte zu beeinflussen.

Die Wahl des Abstandes hat Auswirkungen auf die Verteilung der Punkte. Es muss ein Radius für die ersten Punkte in Abhängigkeit der Texturgröße gewählt werden,



(a) (b)

Abbildung 3.10: Die Verteilung der Punkte nach dem hierarchischen *Poisson Disk*-Prinzip. (a) Jeder Punkte besitzt einen definierten Abstand, indem sich kein anderer Punkte befinden darf. (b) Der Radius des Umkreises wird bei zunehmender Anzahl an Punkten minimiert.

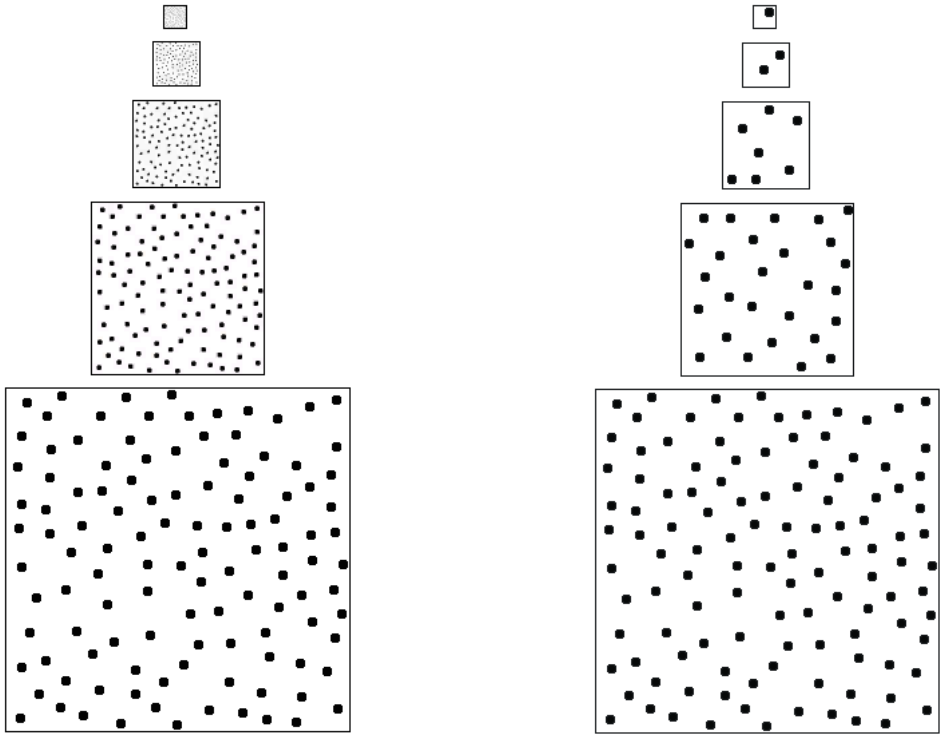
um den gesamten Platz der Textur auszunutzen und eine Anhäufung der Punkte in bestimmten Bereichen zu verhindern. Wird der Radius zu gering gewählt, kann es zu sichtbaren Mustern kommen. Der Anfangsradius wird gesenkt, wenn keine zulässige Position für den Punkt gefunden wurde.

3.3.3 MIP-Maps

Eine MIP-*Map* ist definiert, als eine Folge von Bildern mit abnehmender Auflösung bis zu einer Größe von 1×1 Pixel. Dabei ist die Kantenlänge jedes Bildes halb so groß, wie die des Vorgängerbildes. Üblicherweise werden MIP-*Maps* erstellt, indem für jedes Pixel des verkleinerten Bildes der Mittelwert der vier korrespondierenden Pixel des Ausgangsbildes berechnet wird. Dies wird rekursiv für alle weiteren Stufen durchgeführt. In dieser Arbeit besteht eine MIP-*Map* aus einer Folge von Bildern einer Textur mit abnehmender Auflösung.

In Abbildung 3.11 ist die MIP-*Map* für eine *Stippling*-Textur dargestellt. Das Zusammenfassen mehrerer Pixel führt dazu, dass aus den schwarzen Punkten durch Interpolation mit dem weißen Hintergrund graue Punkte werden. Die *Stippling*-Punkte verändern sich von schwarz zu grau und später entsteht aus der *Stippling*-Textur eine einheitlich graue Texturfläche ohne eindeutige Punkte. Abbildung 3.11 (a) zeigt die verschiedenen Auflösungsstufen einer Textur von 256×256 bis 16×16. Hier ist gut zu erkennen, dass mit abnehmender Texturgröße auch eine Veränderung der Punktform auftritt. Die anfangs runden *Stippling*-Punkte erscheinen durch die Verringerung der Textur und die damit zusammenhängende Änderung ihrer Farbe eckig. Texturen ab einer Auflösung von 32×32 besitzen keine erkennbaren Punkte mehr. Bei der Verwendung dieser MIP-*Map* für die

Texturabbildung würden weiter entfernte Objektbereiche nicht mehr punktiert dargestellt werden, sondern einen grauen Farbton erhalten. Bei dem Aufbau einer MIP-*Map* wird nicht explizit die *Frame*-Kohärenz der Punkte beachtet. Weiterhin werden die wesentlichen Merkmale von MIPs entsprechend der TAM für NPR-Illustrationen vernachlässigt. Da allerdings mit abnehmender Auflösung der einzelnen Bilder die Punkte nicht mehr konkret als schwarze abgegrenzte Kreise erkennbar sind, entstehen keine störenden Artefakte oder Rauschen während der Visualisierung.



(a) (b)

Abbildung 3.11: Die MIP-Stufen einer *Stippling*-Textur von 256×256 bis 16×16 Pixel, wobei die einzelnen MIP-Maps (a) durch Mittelwertbildung von vier korrespondierenden Pixeln und (b) nach dem TAM-Prinzip ausgehend von der kleinsten Auflösung erstellt wurden.

In Abbildung 3.11 (b) ist der Aufbau einer MIP-*Map* nach dem TAM-Konzept dargestellt, bei dem zuerst die niedrigsten Auflösung einer Textur erstellt wird. Alle Punkte, die in den kleiner aufgelösten Texturen enthalten sind, müssen auch in den größeren vorhanden sein, um so eine Kohärenz auf Punktebene zwischen den einzelnen Auflösungsstufen zu ermöglichen. *Stippling*-Punkte werden zunächst für die kleinste Auflösungsstufe erzeugt und bezüglich ihrer Position getestet, wie im vorherigen Abschnitt beschrieben. In der nächst größeren Textur müssen die Punkte ebenfalls enthalten sein und deren Position an die Texturgröße angepasst

werden. Da die Kantenlänge der Textur verdoppelt wird, kann die Punktposition in Relation zur Texturgröße durch Verdopplung der x- und y-Koordinate beibehalten werden. Weiterhin wird durch eine Vervierfachung der Punktanzahl bei steigender Auflösung die zu repräsentierende Helligkeit der Textur bewahrt. Da die Punkte einer Darstellung immer gleich groß sein sollen, unabhängig von der Größe des Objektes, wird nur die Texturgröße variiert.

3.4 Shader-Programme

Die Texturierung der medizinischen Daten erfolgt aus Effiziensgründen unter Verwendung von *Shader*-Programmen. Alle Schritte, die für die Textur-Abbildung notwendig sind, können so während des Renderings von der Graphikkarte ausgeführt und damit beschleunigt realisiert werden. Im Folgenden wird nun kurz erläutert, welche mathematischen Berechnungen pro *Vertex* oder pro *Fragment* ausgeführt werden müssen, um eine Stippling-Darstellung mittels Texturierung zu erhalten.

Ein *Shader*-Programm arbeitet auf den einzelnen Elementen des Datenstromes. Dementsprechend müssen die Programme so entworfen werden, dass die Modifikationen für jedes Element ausführbar sind und den gewünschten Effekt erzeugen. Weiterhin ist es erforderlich, bei Einsatz eines *Shaders* die jeweiligen Transformationen der Funktions-Pipeline für *Vertices* und *Fragment* vollständig zu übernehmen (siehe Seite 37-39).

3.4.1 Berechnungen pro Vertex

Im *Vertex Shader* muss zunächst die Umwandlung der Weltkoordinaten eines *Vertices* in Kamerakoordinaten und die Projektionsberechnungen erfolgen, um die Bildschirmkoordinaten zu bestimmten und an den *Fragment Shader* weiterleiten zu können. In Gleichung 3.5 sind beide Schritte aufgelistet, wobei M die Matrix zur Umwandlung in Kamerakoordinaten und P die Projektionsmatrix darstellt. Die Transformationen werden bei der Verwendung eines *Shaders* nicht von der Render-Pipeline ausgeführt und müssen demnach in dem Programm realisiert werden. Weiterhin können im *Vertex Shader* auch Berechnungen ausgeführt oder Variablen deklariert werden, deren Werte erst später im *Fragment*-Programm wichtig sind.

$$
\begin{aligned}
Kamerakoordinaten &: \begin{pmatrix} x_k \\ y_k \\ z_k \end{pmatrix} = M * \begin{pmatrix} x \\ y \\ z \end{pmatrix} \\
Projektion &: \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = P * \begin{pmatrix} x_k \\ y_k \\ z_k \end{pmatrix}
\end{aligned}
\tag{3.5}
$$

Die Texturkoordinaten werden basierend auf den Weltkoordinaten der *Vertices* und Normalen-Vektoren ermittelt, da dort das Objekt und die einzelnen Würfel für die Texturabbildung definiert sind und die Texturabbildung folglich unabhängig von der Bewegung der Kamera ist. Die untransformierten Koordinaten jedes *Vertices* und der zugehörigen Normalen werden daher zusätzlich abgespeichert, der Attributliste hinzugefügt und in der Pipeline weitergeleitet. Wenn der jeweilige *Vertex* oder der Normalen-Vektor transformiert in die Texturberechnungen eingeht, würde sich bei jeder Kamerabewegung die Texturkoordinate eines betrachteten Oberflächenpunktes ändern.

Weiterhin wird für den Texturzugriff im *Fragment Shader* die Stärke der Beleuchtung an einem Oberflächenpunkt benötigt, die ein Kriterium für die Wahl der Textur ist. Je dunkler der Oberflächenpunkt erscheint beziehungsweise je geringer die Lichtintensität an dieser Stelle ist, desto dunkler muss auch die verwendete Textur sein. Es existieren zwei Möglichkeiten, um den Wert im *Fragment*-Programm für die Texturierung zur Verfügung zu stellen. Einerseits kann anhand der Normalen die Intensität der Beleuchtung pro *Vertex* berechnet und an das *Fragment*-Programm übergeben werden. Die Werte zwischen den einzelnen *Vertices* werden dann interpoliert. Andererseits ist es auch möglich, die Normale zu übergeben und im *Fragment Shader* mit dem interpolierten Normalen-Vektor die Intensität für jedes *Fragment* zu ermitteln. Erstere Variante wird bevorzugt, da der Aufwand der Berechnung pro *Fragment* aufgrund des umfangreicheren Datenstromes höher ist. Da die Intensität eines Oberflächenpunktes nur für die Auswahl der Textur verwendet wird, ist es ausreichend diese pro *Vertex* zu bestimmen und später für die Werte dazwischen zu interpolieren.

3.4.2 Berechnungen pro Fragment

Die Texturkoordinaten werden im *Fragment Shader* pro *Fragment* berechnet und ermöglichen im Gegensatz zu einer Bestimmung pro *Vertex* eine genauere Bestimmung der Texel. Ein Texturzugriff, basierend auf den Texturkoordinaten der *Vertices* würde die Texel zwischen zwei *Vertices* interpolieren. Folglich muss

auch die zu betrachtende Würfelseite für ein *Fragment* in diesem Programm bestimmt werden. Wie bereits beschrieben, wird hierfür der Normalen-Vektor eines jeden *Fragmentes* bezüglich der Hauptkomponenten überprüft. Für die so ermittelte Koordinatenachse wird nun eine Ebene mit der Gleichung 3.2 aufgespannt. Der Schnittpunkt mit dem Normalen-Vektor wird berechnet und die Texturkoordinaten, wie in Abschnitt 3.2.2 entworfen, bestimmt.

Ferner wird auch die Textur, abhängig von der im *Vertex Shader* erfolgten Beleuchtungsberechnung, ermittelt werden. Für ein *Fragment* wird die Textur ausgewählt, die dessen Grauwert repräsentiert. Dies geschieht anhand der berechneten Lichtintensität für das jeweilige *Fragment*. Wie in Abschnitt 3.3.1 beschrieben, entspricht dieser Wert einem Schwellwert für die Bestimmung der zugehörigen Textur. Die abgespeicherte Textur enthält grauwertkodiert alle anderen Helligkeitsstufen. Anhand des errechneten Intensitätswertes lässt sich die korrespondierende Textur extrahieren. Alle Texel der Textur, die entweder genau den Grauwert 255 besitzen oder unter dem ermittelten Wert liegen, werden weiß gezeichnet, alle anderen schwarz. Da die *Stippling*-Punkte der einzelnen Texturen entsprechend ihrer Zugehörigkeit zu einer Textur grauwertkodiert sind, wird so die korrespondierende Textur für jedes *Fragment* ermittelt.

3.5 Zusammenfassung

Basierend auf den betrachteten Anforderungen an eine interaktive Visualisierung sowie den bereits existierenden Verfahren wird im Rahmen dieser Arbeit eine *Stippling*-Darstellung medizinischer Oberflächenmodelle über 2D-Texturen realisiert. Die wesentlichen Merkmale der *Stippling*-Technik wurden bei dem Entwurf der Texturen berücksichtigt sowie das Konzept der TAMs zur Darstellung unterschiedlicher Helligkeits- und Auflösungsstufen integriert und in einer Textur umgesetzt. Die Texturabbildung setzt sich aus dem *Cube Mapping*- und dem *Polycube*-Ansatz zusammen, um mögliche Texturverzerrungen zu minimieren. Die Verfahren werden so modifiziert, dass für ein *Cube Mapping* eine 2D-Textur ausreichend ist und anstelle des Objektes der Objektraum in Würfel unterteilt werden kann. Ferner sind aufgrund der verwendeten Textur alle Berechnungen für die Texturabbildung anhand eines Referenzwürfels durchführbar. Für einen Oberflächenpunkt wird zunächst durch die Beleuchtungsberechnung der Grauwert ermittelt und somit die zu verwendende Textur bestimmt. Der Normalen-Vektor des Punktes definiert die zu betrachtende Seite, wodurch eine Unterscheidung zwischen gering und stark gekrümmten Oberflächen erfolgt und somit eine verbesserte Texturierung möglich ist. Der Schnittpunkt der Normalen mit der Würfelseite entspricht dem korrespondierenden Texel. Stauchungen und

Streckungen der Textur auf dem Objekt können mit der Unterteilung in mehrere Würfel sowie durch die Betrachtung des Normalen-Vektors für die Bestimmung der Würfelseite minimiert werden. Es wird eine Annäherung an eine flächentreue und konforme Abbildung erreicht. Die Umsetzung der Texturabbildung erfolgt hardwarebeschleunigt unter Verwendung von *Vertex* und *Fragment Shadern*.

4 Implementierung

Dieses Kapitel beschreibt die Implementierung der Texturabbildung und der Texturgenerierung. Die *Stippling*-Texturen werden hardwarebeschleunigt mittels *Shader* auf die Objekte abgebildet. Die verwendeten Werkzeuge sowie die einzelnen Programme werden im folgenden Abschnitt vorgestellt und kurz erläutert. Ferner werden die erzielten Ergebnisse und eine Auswertung der Resultate präsentiert und diskutiert.

4.1 Umsetzung der Texturabbildung

Die Implementierung der *Shader* erfolgt auf einem Intel Pentium 4 Prozessor mit 3,2 GHz und anfänglich auf einer ATI RADEON 9600 Graphikkarte. Da hier allerdings aufgrund der beschränkten Anzahl an erlaubten Instruktionen die Berechnung der Texturkoordinaten nicht möglich war, wurden dann eine ATI RADEON X800 Graphikkarte mit 256 MB, die *Vertex-* und *Fragment Shader* 2.0 unterstützt, verwendet. Als Entwicklungsumgebung für den Entwurf der *Shader* wurden MeVisLab[1] und RenderMonkey 1.6 (*RenderMonkey Integrated Development Environment*)[2] von ATI gewählt.

4.1.1 MeVisLab

MeVisLab ist eine Entwicklungsplattform für die Bildverarbeitung und Visualisierung medizinischer Datensätze, die vom Centrum für Medizinische Diagnosesysteme und Visualisierung (MeVis) in Bremen entwickelt wurde. Für die Entwicklung von neuen Anwendungen werden in MeVisLab mittels *Drag & Drop* individuelle Netzwerke aus Modulen erstellt. Module sind Einheiten, die verschiedene Funktionalitäten für die Verarbeitung, Visualisierung oder Interaktion mit den Daten bereitstellen. MeVisLab nutzt OpenInventor und übernimmt dessen Szenengraphkonzept. OpenInventor ist eine Graphkbibliothek zur objektorientierten Programmierung von interaktiven 3D-Anwendungen, die auf OpenGL aufsetzt.

[1] www.mevislab.de

[2] www.ati.com/developer/rendermonkey/index.html

Alle Objekte einer 3D-Szene werden hierarchisch in einer Baumstruktur (Szenengraphmodell) verwaltet, um eine einfache Modellierung und Animation zu ermöglichen.

Für eine Implementierung der *Shader*-Programme können die als OpenInventor Knoten erstellten `SoShaderProgramm`, `SoVertexShader` und `SoFragmentShader` Komponenten verwendet werden. Das Modul `SoShaderProgramm` erhält von `SoVertexShader` und `SoFragmentShader` die dort entworfenen Modifikationen, die dann pro *Vertex* oder *Fragment* auf dem verwendeten Datensatz ausgeführt werden (siehe Abbildung 4.1).

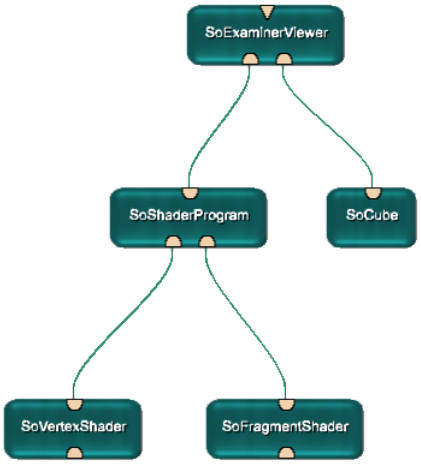


Abbildung 4.1: Ein Beispielnetzwerk in MeVisLab, wobei hier durch die *Shader* ein Würfel modifiziert werden kann.

Für die Entwicklung der *Shader* gibt es jedoch Beschränkungen in MeVisLab. Einerseits kann nur eine 2D-Textur eingelesen werden, was bei dem anfänglichen Entwurf und der Umsetzung der Texturierung problematisch war, da die einzelnen Texturen noch nicht in einer einzigen integriert waren. Andererseits existiert in MeVisLab keine automatische Zuweisung der MIP-Texturen zu den unterschiedlichen Skalierungen des Objektes. Folglich wird eine Textur auf das Objekt abgebildet und mit diesem skaliert. Die Zuweisung einzelner MIP-Texturen muss hier manuell erfolgen. Es wurde eine Entwicklungsplattform benötigt, die auf die Entwicklung von *Shader*-Programmen spezialisiert ist und ferner stabilere Programmabläufe gewährleistet.

4.1.2 RenderMonkey

RenderMonkey ist eine frei verfügbare *Shader*-Entwicklungsumgebung (*Shader Development Tool*), die das Erstellen und Verwalten von 3D-Echtzeit-

Effekten mittels *Vertex* und *Fragment Shader* unterstützt. DirectX (9.0c) sowie OpenGl (1.5) *Shader*-Sprachen können für die Implementierung der einzelnen Programme genutzt werden. Ferner können die *Shader* für die Weiterverwendung in einer anderen 3D-Software exportiert werden.

Die graphische Benutzeroberfläche von RenderMonkey besteht aus einem Arbeitsbereich (*Workspace*), dem *Shader Editor*, dem *Preview*-Fenster sowie einem Ausgabefenster. Im Arbeitsbereich wird das gesamte Projekt als hierarchische Baumstruktur angezeigt, deren Knoten die benutzerspezifischen Parameter, Texturen, Modelle und die verschiedenen Effekte beinhalten. Die Variablen und Texturen sind später über `uniform`-Parameter im *Shader*-Programm verfügbar. Der *Shader Editor* beinhaltet den Quellcode der einzelnen *Shader*-Programme, der dort individuell bearbeitet werden kann. Mittels des integrierten *Shader*-Compilers sowie des *Preview*-Fensters ist es möglich, die entwickelten Effekte sofort anzuzeigen und demnach die Fehlersuche erheblich zu erleichtern.

Die medizinischen Objekte werden als 3D-Studio (.3ds) Modelle geladen, dargestellt und durch *Shader*-Effekte individuell modifiziert. 2D-Texturen können in verschiedenen Bildformaten benutzt werden, wobei RenderMonkey hier die benötigten MIP-Texturen automatisch berechnet. Mit dem vorwiegend verwendeten Format DirectDrawSurface(.dds) ist es mögliche die MIP-Texturen manuell dem Programm zu übergeben. Für den Entwurf oder die Bearbeitung von Texturen und den zugehörigen MIP-*Maps* existiert ein DDS *Plug-in* von Nvidia[3], welches dem Entwickler erlaubt in Adobe Photoshop oder Jasc Paint Shop Pro Bilder im DDS-Format zu öffnen oder abzuspeichern.

4.1.3 Shader-Programme

Vertex und *Fragment Shader* wurden mit der *OpenGl Shading Language* 2.0 realisiert. Zwischen der *OpenGl Shading Language* 2.0 und der Sprache Cg besteht für die Implementierung dieses Konzeptes kein Unterschied. In der *OpenGl Shading Language* 2.0 können die Variablen von *Vertex* zu *Fragment Shader* über beliebig definierte *varying*-Variablen weitergeleitet werden und unterliegen nicht wie bei Cg der Bindungssemantik (siehe Abschnitt 2.4.3 Seite 40). Da dies eine für die Programmierung der *Shader*-Programme „elegantere“ Lösung darstellt, wurden alle *Shader* in der *OpenGl Shading Language* 2.0 geschrieben.

[3] http://developer.nvidia.com/object/photoshop_dds_plugins.html

Vertex Shader

Im *Vertex Shader* werden, wie bereits im Entwurf beschrieben, zunächst die Koordinaten jedes *Vertices* in Kamerakoordinaten transformiert, um in der Render-Pipeline weiter verarbeitet werden zu können. Ferner werden hier bereits Berechnungen für die im *Fragment Shader* stattfindende Texturabbildung ausgeführt. Die Auswahl der Textur im *Fragment Shader* ist abhängig von der reflektierten Intensität des *Fragmentes*. Hierfür müssen Beleuchtungsberechnungen durchgeführt werden, die bereits pro *Vertex* erfolgen können.

Demzufolge wird für jeden *Vertex* die reflektierte Intensität (`lightIntensity`) ermittelt, an die Attributliste angehängt und zum *Fragment Shader* geleitet. Dort stellt dieser Wert den Schwellwert dar, der aus der abgespeicherten die benötigte Textur extrahiert.

```
vec3 VertexPos     = vec3 (gl_ModelViewMatrix * gl_Vertex);
vec3 transNormal   = normalize(gl_NormalMatrix * gl_Normal);
vec3 Light         = normalize(vec3(gl_LightSource[0].position) - VertexPos);
float lightIntensity = max(dot(Light, transNormal), 0.0);
```

Fragment Shader

Die Berechnung der Texturkoordinaten sowie die Zuweisung der ermittelten Texel zu den *Fragmenten* findet im *Fragment Shader* statt. Wie bereits in Abschnitt 3.2.3 beschrieben, muss die Achse, für die die Ebene aufgespannt wird, bestimmt werden und die Schnittpunktberechnungen mit dem Normalen-Vektor für die Texturabbildung erfolgen. Pro *Fragment* wird zunächst die maximale Komponente des Normalen-Vektors ermittelt, um dann die Gleichungen 3.2 und 3.3 zur Bestimmung der Texturkoordinaten auszuführen. Mit `if`-Anweisungen wird zwischen den drei möglichen Achsen x, y und z unterschieden. Das Vorzeichen der ermittelten maximalen Vektorkomponente ist nur für den Punkt zur Aufstellung der Ebenengleichung wichtig und wird später abgefragt. Die benötigten Berechnungen zur Ermittlung der (s, t) Texturkoordinaten können für alle Achsen in Form einer Funktion ausgeführt werden, da sich nur die Vektoren für die Ebenengleichung, je nach ermittelter x, y oder z Achse, unterscheiden.

Die Funktion `TextureCoordinates` realisiert diese Berechnungen. Zunächst wird das Vorzeichen der betrachteten Achse ermittelt, um entsprechend den maximalen oder minimalen Würfeleckpunkt für die Ebenengleichung zu verwenden. Dann werden, gemäß der im Kapitel 3 vorgestellten Gleichungen 3.2 und 3.3, die

Schnittpunktberechnungen durchgeführt und alle möglichen Texturkoordinaten bestimmt. Die Variablen `vPos` und `vNormal`, die die untransformierten Koordinaten der Position beziehungsweise des Normalen-Vektors speichern, werden von dem *Vertex* an den *Fragment Shader* weitergeleitet.

```
//Funktion zur Berechnung der Texturkoordinaten
vec3 TextureCoordinates (vec3 planeNormal, float Cubeside)

{
  vec3 TexCoord = vec3(0.0,0.0,0.0);

  //Punkt auf der Ebene bestimmen
  if(Cubeside >= 0.0)
  {
    vec3 planePoint = Cubemax;

  } else {
        planePoint = Cubemin;
        }

  // Schnittpunktberechnungen
  vec3 D      = planePoint * planeNormal;
  vec3 t      = (D - (vPos * planeNormal)) / (vNormal * planeNormal);
  vec3 iPoint = vPos + k * vNormal;

  //Texturkoordinaten
  TexCoord.x = (iPoint.x - Cubemin.x) / Cubemax.x - Cubemin.x);
  TexCoord.y = (iPoint.y - Cubemin.y) / Cubemax.y - Cubemin.y);
  TexCoord.z = (iPoint.z - Cubemin.z)  / Cubemax.z - Cubemin.z);

  return TexCoord;
}
```

Diese Funktion liefert einen 3D-Vektor (`TexCoord`) zurück. In den einzelnen `if`-Abfragen werden dann, je nach Koordinatenachse, die benötigten Texturkoordinaten auf den s, t Komponenten von `TexCoord` abgespeichert, die für den Texturzugriff benötigt werden. Die *OpenGl Shading Language* bietet die vordefinierte Funktion `texture2D`, die als Eingabewerte eine 2D-Textur und die zugehörigen (s, t) Texturkoordinaten bekommt und somit den Texel dem betrachteten *Fragment* zuweist. Die *Stippling*-Textur wird in Form eines 2D-Texturknotens in die Baumstruktur des Projektes in RenderMonkey eingefügt und als `uniform sampler2D` Variable an den *Fragment Shader* übergeben. Nach Aufruf der Funk-

tion `TextureCoordinates` erfolgt mit den berechneten Koordinaten, die auf `TexCoord` gespeichert sind, der Zugriff auf die Textur über `texture2D`. Anhand der `lightIntensity` wird eine entsprechende Textur ausgewählt.

- **1.Variante:**
  Für die Verwendung von automatisch generierten MIP-*Maps* muss der Zugriff manuell erfolgen, da die geringer aufgelösten Texturen keine differenzierbaren Grauwerte enthalten, anhand derer die Schwellwert-Methode ausgeführt werden kann. Folglich müssen die Texturen einzeln abgespeichert und im *Shader* gelesen werden. Es muss bereits bei der Programmierung des *Shaders* die genaue Anzahl der Texturen bekannt sein, da `for`- oder `while`-Schleifen von der Graphik-Hardware nicht unterstützt wurden.

- **2.Variante:**
  Die manuell erstellten Texturen erlauben die Extraktion der Textur, basierend auf dem, durch die reflektierende Intensität definierten, Schwellwert. Es wird getestet, ob der Grauwert des ermittelten Texels dazu führt, dass dieser weiß oder schwarz gezeichnet wird. Es ist möglich anstatt schwarz jede beliebige Farbe zu definieren und somit die *Stippling*-Darstellung, um Farbinformationen zu erweitern. Der Parameter `lightIntensity` bestimmt den Schwellwert, der die entsprechende Helligkeitsstufe einer Textur aus der abgespeicherten Textur extrahiert. Liegt der Grauwert des Texels über dem Schwellwert (`lightIntensity`) wird dieser schwarz gezeichnet, da er dann zu einer helleren Textur gehört, ansonsten wird er weiß. Die Funktion **vec4** step (**vec4** edge, **vec4** x) liefert für *edge* als Schwellwert und x als Grauwert des Texels, den Wert 0 und somit schwarz zurück, wenn $x <= edge$ ansonsten 1 (weiß). Da die Grauwerte der gespeicherten *Stippling*-Punkte auf der Textur die verschiedenen Helligkeiten repräsentieren und die Punkte der hellen Texturen hohe Grauwerte besitzen, wird der Schwellwert auf $edge = 1 - edge$ gesetzt und so die endgültige Farbe eines *Fragmentes* bestimmt.

  Die Texturen werden hardwarebedingt bei variierender Entfernung des Betrachters zum Objekt skaliert. Demzufolge weichen die Grauwerte an den Rändern der *Stippling*-Punkte durch Aliasing von dem eigentlichen Grauwert in geringem Maße ab. Bei der Verwendung einer `step`-Funktion entstehen folglich eckige Punkte, da nur ein Schwellwert beachtet wird und die Ränder der Punkte abweichende Grauwerte besitzen. Die Darstellung kann mit einer `smoothstep`-Funktion verbessert werden, die zwischen zwei Grenzwerten interpoliert. Die Grenzen werden auf $edge1 - a$ und $edge2 + a$ gesetzt, wobei durch a eine Umgebung um den Grenzwert definiert wird. Für $a = 0.2$ wurden bereits gute Resultate erzeugt. Mit `smoothstep` wird demnach ein Intervall von Grauwerten um den Schwellwert herum berücksichtigt.

4.2 Generierung der Stippling-Textur

Die *Stippling*-Texturen werden separat erzeugt. Hierfür wurde ein Programm implementiert, welches die Texturen generiert und dabei die im vorherigen Kapitel beschriebenen Anforderungen an die Verteilung und die Größe sowie an die Abstände der Punkte erfüllt. Der Aufbau der Helligkeits- und Skalierungsstufen folgt dem in Abschnitt 3.3.1 beschriebenen Konzept. Die Punkte sind demnach auf jeder Textur sowie jeder Auflösungsstufe gleich groß und die verschiedenen Helligkeiten sind in den Grauwerten der einzelnen Punkte kodiert und können daher in einer Textur gespeichert werden. Die Implementierung der Texturgenerierung setzt sich aus der Positionierung der einzelnen Punkte auf den verschiedenen MIP-Stufen und der Integration der verschiedenen Helligkeitsstufen in einer Textur zusammen.

Zur Darstellung einer *Stippling*-Textur werden zunächst alle Punkte der einzelnen Skalierungs- sowie der Helligkeitsstufen generiert und anschließend entsprechend ihrer Zugehörigkeit zu einer Textur und Helligkeit gezeichnet. Die Anzahl der Punkte der dunkelsten Textur, der Radius und der anfängliche Mindestabstand der Punkte sowie die Anzahl der gewünschten MIP-Stufen und Helligkeiten werden zu Beginn vom Benutzer individuell festgelegt.

Die zur Texturgenerierung benötigten Funktionen und Berechnungsschritte sind in Java implementiert worden. Die entwickelten Klassen `CreateStipples`, `Dot` und `DrawTexture` enthalten die wesentlichen Algorithmen zur Erstellung einer Textur und werden im Folgenden näher erläutert.

4.2.1 Verteilung der Stippling-Punkte

Ein *Stippling*-Punkt ist ein Objekt vom Typ `Dot` und durch seine (x, y) Position definiert. Diese Koordinaten werden für die Abstandsberechnungen benötigt und sind ausreichend für die Speicherung eines Punktes. Alle Punkte sind zunächst einzelne (x, y) Positionen, die später in `DrawTexture` mit dem vorgegebenen Radius als *Stippling*-Punkte gezeichnet werden. `DrawTexture` berechnet dafür eine Kreisgleichung, wobei die (x, y) Koordinaten als Mittelpunkt und der Radius genutzt werden, um die innerhalb dieses Kreises liegenden Texel ebenfalls in dem entsprechenden Grauwert darzustellen.

Die Klasse `CreateStipples` erzeugt alle Punkte einer Textur. Ein Punkt entspricht anfangs einem Kandidaten dessen (x, y) Koordinaten als Pseudozufallszahlen mit der Java-Klasse `Random` generiert werden. Dieser Kandidat muss imklusive seines Radius innerhalb der Texturgröße liegen und den vorher definierten Minimalabstand zu den bisherigen Punkten einhalten. Hierfür wird der Euklidische Abstand des Kandidaten zu den anderen Punkten berechnet und daher eine

Poisson Disk-Verteilung erzielt. Erfüllt ein generierter Kandidat eines dieser Tests nicht, wird solange ein neuer Kandidat generiert bis entweder die benötigte Anzahl der Punkte oder die maximale Anzahl an Versuchen einen Punkt zu erzeugen erreicht ist. Bei Erreichen der maximalen Versuchsanzahl wird zunächst der einzuhaltende Mindestabstand der Punkte verringert, da mit zunehmender Punktanzahl die zulässigen Abstände zueinander minimiert werden müssen (siehe Abbildung 3.10).

Besteht ein Kandidat alle Tests wird er in einer `LinkedList` abgespeichert. Diese Liste enthält alle Punkte, wobei die Punkte zusätzlich einen Index erhalten, der ihre Zugehörigkeit zu einer Helligkeitsstufe repräsentiert und später für die Berechnung der Grauwerte verwendet wird. Die Speicherung in einer `LinkedList` berücksichtigt die variable Punktanzahl einer Textur. Können nicht alle Punkte im Rahmen der festgelegten Versuchsanzahl gefunden werden, wird die Textur mit den bisher gefunden Punkten gezeichnet. Für die Implementierung wurden gute Resultate mit einer maximalen Anzahl von bereits 1000 Versuchen erzielt.

4.2.2 Berechnung des Grauwertes eines Punktes

Die Grauwerte der Punkte entsprechen ihrer zu repräsentierenden Helligkeit. Je größer der Grauwert eines Punktes ist, desto heller ist die Textur zu der er gehört. Bei der Generierung der Punkte wird ein Index abgespeichert, der ihre Textur und somit ihren Grauwert bestimmt. Es wird vorausgesetzt, dass der maximal mögliche Grauwert < 255 ist ($255 =$ Weiß). Dieser Grenzwertert kann beliebig gewählt werden, muss aber geringer als 255 sein, um sich vom Texturhintergrund zu unterscheiden. Abhängig von der verwendeten Anzahl an Helligkeitsstufen wird ein Offset-Wert berechnet, der den Grauwert eines *Stippling*-Punktes angibt (siehe Abschnitt 3.3.1). Je nach Index eines Punktes erhält dieser seinen entsprechenden Grauwert, mit dem er auf der Textur gezeichnet wird.

4.2.3 Implementierung der MIP-Stufen

Wie bereits erwähnt legt der Benutzter die Anzahl an MIP-Stufen fest. Die kleinste Textur ist durch das von RenderMonkey verwendeten Format auf die Größe 1×1 Pixel festgelegt. Für jede MIP-Stufe, angefangen bei der kleinsten Textur, wird die angegebene Anzahl an Punkten in der Klasse `CreateStipples` generiert und in einer Liste abgespeichert. Für die nächst größere Textur werden die bereits generierten Punkte einer kleineren Textur übernommen und ihre Koordinaten an die neue Texturgröße angepasst. Die x und die y Koordinate eines Punktes werden hierfür verdoppelt und behalten daher bezüglich der Texturgröße ihre Position.

Kann aufgrund der angegebenen Punktgröße oder aufgrund der maximal möglichen Anzahl an Versuchen einen Punkt zu positionieren kein Punkt auf dieser Textur gesetzt werden, bleibt die zugehörige Punkt-Liste leer und es wird eine weiße Textur gezeichnet. Die Anzahl der Punkte für eine MIP-Textur wird von einer kleineren zu einer größeren Textur um den Faktor vier erhöht. Da sich die Größe der Textur vervierfacht, wird durch diese Erhöhung der Punktanzahl eine konstante Helligkeit erreicht (siehe Abbildung 3.11 (b)).

Die verschiedenen Texturen werden ebenfalls lediglich durch Listen von Punkt-Positionen realisiert, die am Ende der Berechnungen durch die Klasse `DrawTexture` als *Stippling*-Punkte auf einer weißen Textur dargestellt werden. Eine Textur mit allen MIP-Stufen ist intern als eine Liste von Listen implementiert, wobei eine Listen ein MIP-Level repräsentiert.

4.2.4 Speicherstruktur

Die Speicherstruktur der *Stippling*-Textur sowie der MIP-Stufen ist durch RenderMonkey vorgegeben. Die einzelnen MIP-Texturen werden nur korrekt verwendet, wenn ein DDS-Format vorliegt, die Größe der einzelnen Texturen Zweierpotenzen sind, die Anordnung der MIP-Texturen von links nach rechts beginnend mit der größten Textur eingehalten wird und die kleinste Textur 1×1 Pixel groß ist. Die einzelnen Texturen werden, wie in Abbildung 4.2 dargestellt, aneinander gehängt, wobei links die Größte und rechts die Kleinste ist.

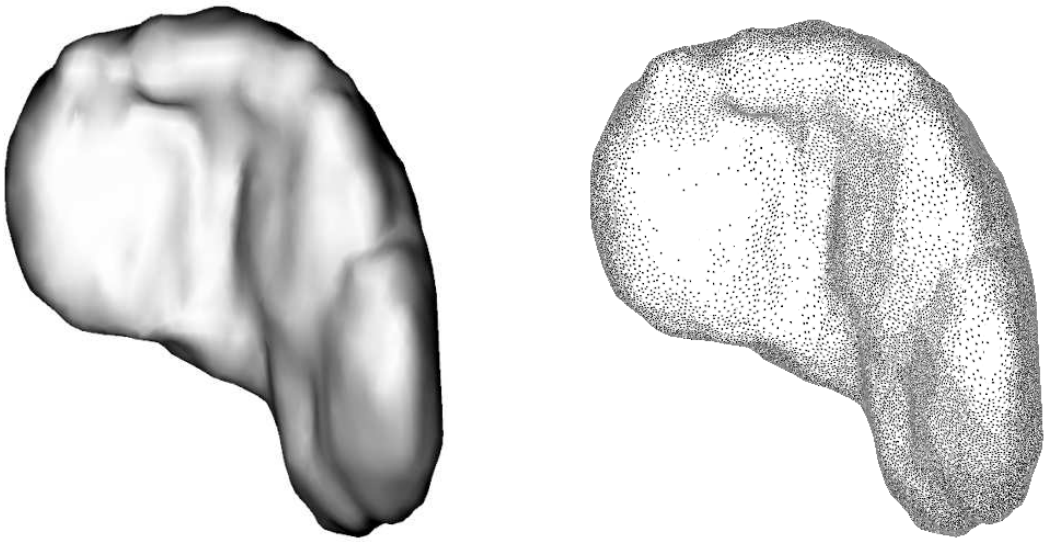
Da in Java nur eine Speicherung der Textur in einem gängigen Bildformat mit dem Java-Package `Imaging I/O` möglich ist, muss die Textur anschließend mit Hilfe des DDS-Konverters in Adobe Photoshop oder Jasc Paint Shop Pro umgewandelt werden. Die MIP-Stufen können dann von RenderMonkey erkannt und angewandt werden.

Abbildung 4.2: Die Speicherstruktur der einzelnen Texturen. Die Kleinste ist 1×1 und die Größte 128×128 Pixel groß.

4.3 Resultate und Auswertung

Es wurden *Stippling*-Texturen auf medizinische Oberflächenmodelle abgebildet und somit eine *Stippling*-Darstellung dieser Objekte generiert. Die im vorherigen Kapitel entworfene Texturabbildung wurde ohne Vorverarbeitung oder Vorwissen über die Geometrie realisiert. Die benötigten Texturkoordinaten werden im *Fragment Shader* berechnet und für den Texturzugriff verwendet. Je nach reflektierter Lichtintensität wird die benötigte Helligkeitsstufe einer Textur ausgewählt. Die verschiedenen Helligkeitsstufen der Texturen unterscheiden sich in der Anzahl ihrer Punkte. Diese variierende Anzahl kann für eine *Stippling*-Darstellung die wahrgenommene Lichintensität der reflektierten Intensität angepasst werden.

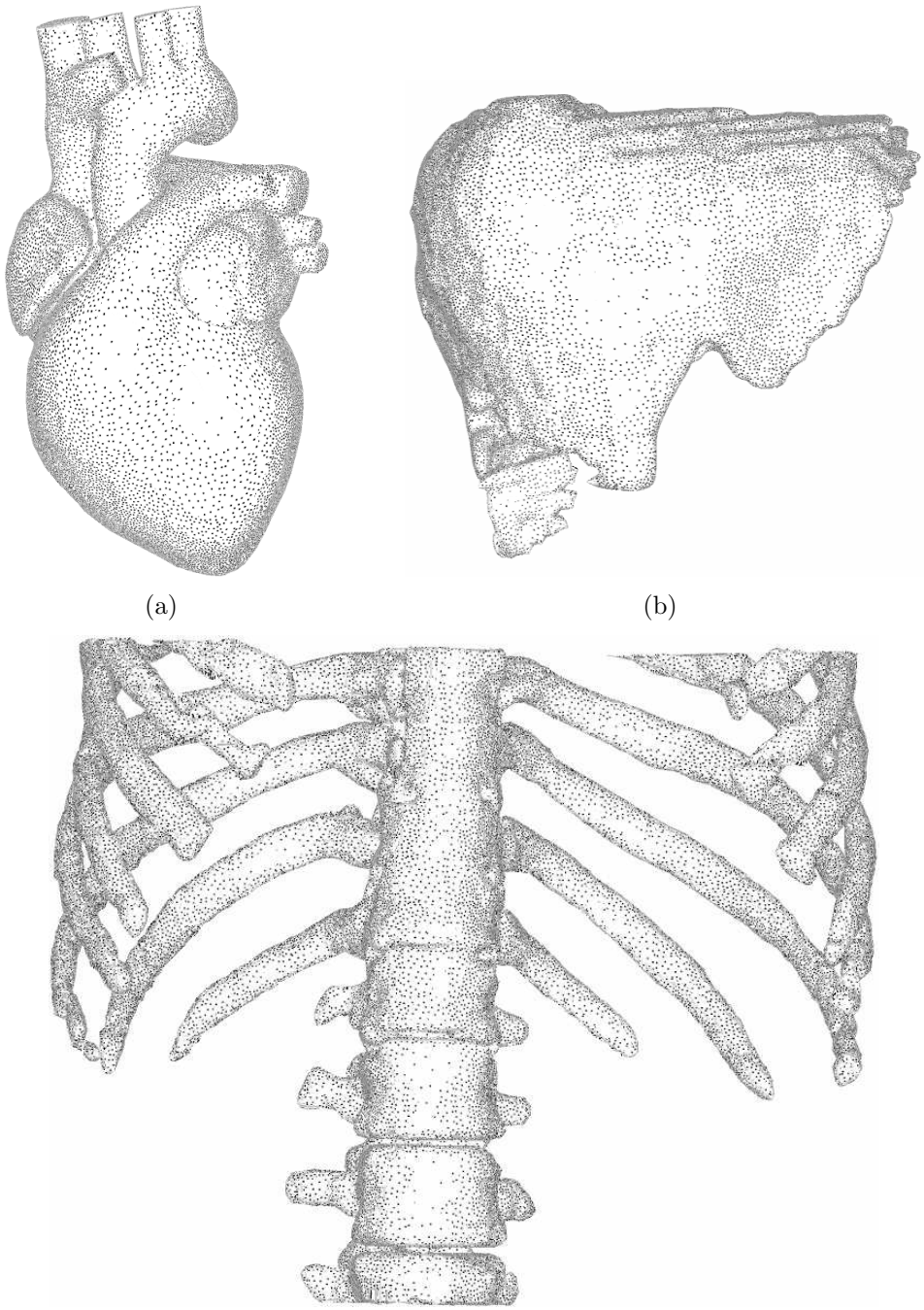
Für die Texturabbildung wurden zunächst, mittels der Unterteilung des Objektraumes in Würfel, die Verzerrungen behoben. Weiterhin erfolgte die Texturierung abhängig von der reflektierten Intensität. Die in Abbildung 4.3 (a) dargestellte Milz ist nur mit der im *Vertex Shader* ausgeführten Beleuchtungsberechnung generiert worden. Abbildung 4.3 (b) zeigt die dazu äquivalente Darstellung mit *Stippling*-Texturen.



(a) (b)

Abbildung 4.3: Die Darstellung der Milz (a) wobei nur im *Vertex Shader* die Beleuchtungsberechnung ausgeführt wird. (b) Die Texturabbildung abhängig von der reflektierten Intensität.

Im Folgenden werden nun einige Modell gezeigt, deren Oberfläche über *Stippling* repräsentiert wird. Hierbei besteht die Möglichkeit, die Objekte ausschließlich durch Punkte, wie in Abbildung 4.4 darzustellen oder Silhouetten hinzuzufügen, um entweder einige Objekte von anderen besser abzugrenzen oder ihre Form



(a) (b)

(c)

Abbildung 4.4: Diese *Stippling*-Darstellungen von (a) einem Herz, (b) einer Leber und (c) dem linken und rechten Rippenbogen sowie der Wirbelsäule wurden unter Verwendung von 7 Helligkeitsstufen der Textur generiert, wobei die dunkelste Textur 382 *Stippling*-Punkte besitzt und 512×512 Pixel groß ist.

zu betonen. In Abbildung 4.5 sind die Organe, die nur zur Darstellung des Kontextes verwendet werden mit Punkten dargestellt. Die Objektform von Niere, Milz und Knochen wird durch die Punktverteilung hervorgehoben, ohne von den wesentlichen Dingen abzulenken. Fokusnahe Objekte werden dagegen farbig visualisiert, wie die Leber mit den intrahepatischen Strukturen aus Abbildung 4.5. Weitere Modelle sind im Anhang A abgebildet.

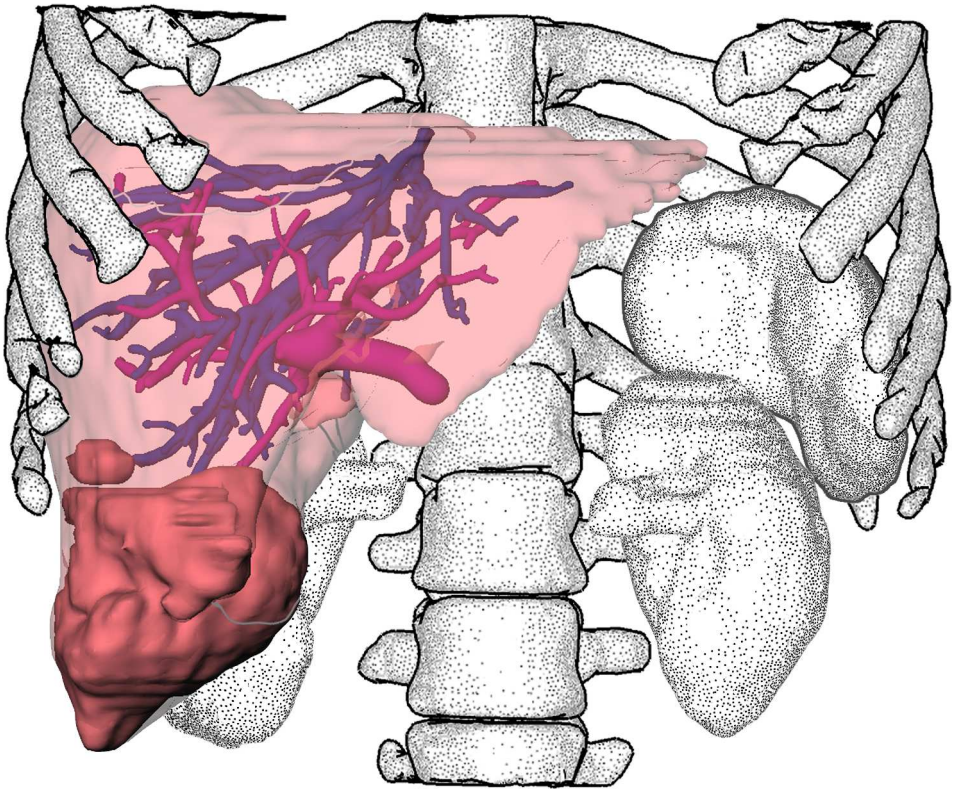


Abbildung 4.5: Diese Visualisierung integriert *Stippling*-Darstellungen zur Veranschaulichung von Kontextinformationen. Während die Milz und die Nieren ausschließlich mit Punkten visualisiert sind, werden die Knochen zusätzlich mit Silhouetten verstärkt. Die Leber mit intrahepatischen Strukturen wurden dagegen farbig dargestellt und repräsentieren fokusnahe Strukturen.

Es wurden verschiedene medizinische Modelle visualisiert und die Geschwindigkeit ihrer Darstellung in *Frames per Seconds* (fps) gemessen. Die Tabelle 4.1 zeigt die gemessene Geschwindigkeit der Objekte ohne und mit einer Textur sowie die Anzahl der Eckpunkte, als ein Maß für die Komplexität des Objektgitters beziehungsweise die Menge der zu bearbeitenden *Vertices* und dementsprechend der *Fragmente*. Der Stanford Bunny wurde als ein weiteres nicht medizinisches Objekt gewählt, um einen Vergleich zu anderen im Abschnitt 2.2 vorgestellten Verfahren zu bekommen. Der objektbasierte Ansatz von Pastor [2003] erreicht bei diesem Objekt 56 fps bei 120,000 generierten Punkten mit einer Nvidia

GeForce 4 Graphikkarte und benötigt für den gesamten Prozess $2min\ 31s$, wobei vorher eine Punkthierarchie aufgebaut werden muss. Eine *Stippling*-Darstellung dieses Objektes mit Texturen kann nun unabhängig von der Menge der Punkte, die dargestellt werden sollen, erfolgen. Die Anzahl der Punkte wird für die Textur festgelegt und wirkt sich demzufolge nicht nachteilig auf die Darstellungsgeschwindigkeit aus. Alle Werte wurden auf der ATI RADEON X800 Graphikkarte unter MeVisLab gemessen. Da die fps abhängig von der verwendeten Graphik-Hardware sind, zeigt diese Tabelle nur eine Verhältnis zwischen der Darstellung ohne *Shader* und der Visualisierung mit der Texturabbildung.

Modell	ohne Textur fps	mit Textur fps	Anzahl der Eckpunkte
Knochen	27	20	22,795
Leber	30	25	19,225
einzelne Niere	36	31	2950
Milz	31	26	3302
Stanford Bunny	34	30	34,834

Tabelle 4.1: Die Darstellungsgeschwindigkeit dieser Modelle wurde in MeVisLab mit dem Modul `SoFramesPerSecond` gemessen und geben ein Verhältnis der Geschwindigkeit zwischen untexturierten und texturierten Objekten an.

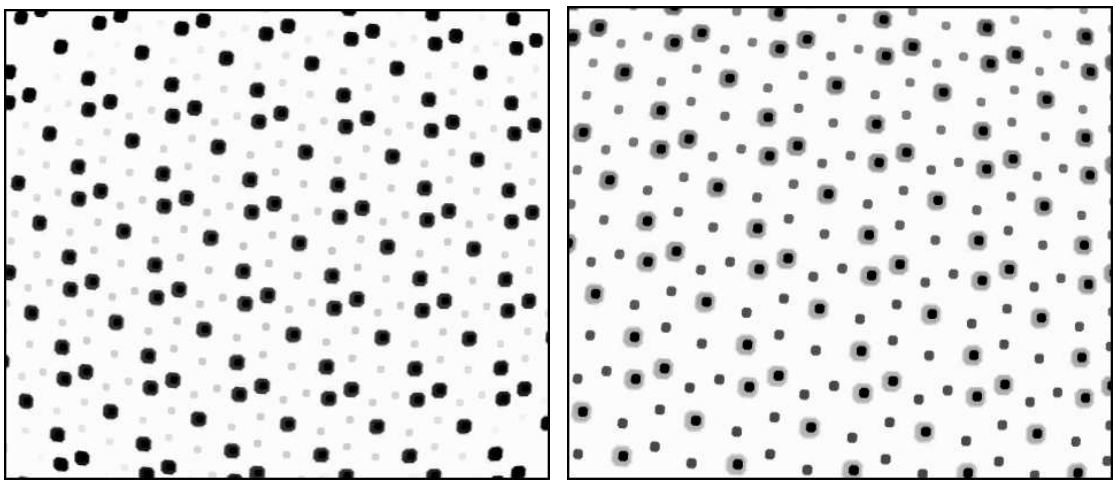
Die bisher gezeigten Resultate verwenden alle MIP-Texturen, die automatisch von dem DDS-Konverter erzeugt werden (siehe Abbildung 3.11 (a)). Diese Darstellungen genügten jedoch noch nicht den im Konzept untersuchten Anforderungen an eine *Stippling*-Illustration, bei der unter anderem konstante Punktgrößen vorliegen müssen. Daher wurden die einzelnen MIP-Stufen der Textur nach dem TAM-Prinzip generiert. Skalierungsstufen einer Textur besitzen demzufolge Punkte konstanter Größe.

Es existieren zwei Varianten für die Umsetzung einer *Stippling*-Visualisierung mit Texturen. Einerseits die bisher gezeigten Darstellungen, bei denen die MIP-Stufen einer Textur automatisch erzeugt werden und andererseits die Umsetzung mit den manuell generierten MIP-Stufen einer Textur nach dem TAM-Prinzip. Bei der Verwendung dieser Texturen werden jedoch einige Probleme sichtbar, die aufgrund einiger Hardwarebeschränkungen beziehungsweise internen Realisierungen bei der Zuweisung der benötigten MIP-Textur entstehen. Weiterhin erfolgt eine ungewollte Skalierung der Texturen, die bei den automatisch erzeugten MIP-Texturen nicht auffallen, da sich der Punktradius mit abnehmender Texturgröße ebenfalls verringert. Im Folgenden soll nun kurz die Ursache dieser Probleme erläutert werden.

4.3.1 Hardwarebedingte Probleme

Bei der Verwendung der manuell generierten MIP-*Map*-Texturen aus Abbildung 3.11 (b) auf Seite 65 werden interne Realisierungen der Graphik-Hardware im Umgang mit MIP-Texturen sichtbar und erzeugen unerwünschte Nebeneffekte. Beim Vergrößern eines Objektes wird die Textur entsprechend skaliert und dann in die nächst größere Textur übergeblendet (siehe Abbildung 4.6). Da die *Stippling*-Punkte auf den manuell erstellten MIP-Texturen immer die gleiche Größe besitzen, um eine konstante Punktgröße bei unterschiedlichen Skalierungen des Objektes zu gewährleisten, ist die Vergrößerung der Punkte vor allem beim Überblenden sichtbar, wie in Abbildung 4.6 (b) in einem Ausschnitt der Objekoberfläche zu sehen ist.

Die hierfür benötigten Algorithmen für ein bilineares und trilineares Filtern werden zugunsten der Performancesteigerung in der Graphik-Hardware gegen das brilineare Filtern eingetauscht. Diese Mischung der beiden traditionellen Methoden soll auf Kosten der Qualität schnellere Visualisierungen ermöglichen. Im Folgenden wird nun kurz die bilineare und trilineare sowie die von der Graphik-Hardware eingesetzte brilinearen Filterung vorgestellt und erläutert.



(a) (b)

Abbildung 4.6: Dieser Ausschnitt verdeutlicht anhand der manuell generierten MIP-Texturen die hardwarebedingten Probleme. (a) Eine Textur inklusive der darauf enthaltenen Punkte wird über einen gewissen Bereich vergrößert. Die Punkte der nächst größeren Textur werden langsam sichtbar. (b) Probleme entstehen dort, wo ein Punkt auch auf der nächsten Textur enthalten ist. Hier taucht innerhalb eines vergrößerten grauen Punktes ein kleinerer schwarzer auf.

bilineares Filtern Bei der Texturabbildung wird der Farbwert eines Pixels durch den korrespondierenden Texel bestimmt. Beim bilinearen Filtern fließen die vier umgebenden Texel für die Bestimmung eines Pixel-Farbwertes mit ein

und erhöhen somit die optische Qualität. Bei der bilinearen Filterung in Kombination mit MIP-Mapping wird allerdings nur eine Textur beachtet, was zu dem Effekt des MIP-*Banding* führt (siehe Abbildung 4.7). MIP-*Banding* ist eine Unstetigkeit, die bei bilinear gefilterten Texturen am Übergang zwischen zwei MIP-Stufen entsteht [3DC, 2001]. Dort wo zwei MIP-Stufen aneinander treffen ändert sich der Schärfegrad der Textur abrupt.

Abbildung 4.7: Drei farbig dargestellte MIP-Texturen mit bilinearer Filterung. Hier entsteht MIP-*Banding*. Quelle: [3DC, 2001]

trilineares Filtern Die trilineare stellt eine Erweiterung zur bilinearen Filterung dar. Es werden zwei Texturwerte benachbarter MIP-Texturen, die sich durch bilineare Interpolation ergeben, berechnet und anschließend linear interpoliert. Folglich wird der MIP-*Banding*-Effekt verhindert und es treten aufgrund der Interpolation zwischen benachbarten MIP-Texturen keine sichtbaren Skalierungen der einzelnen Texturen auf (siehe Abbildung 4.8).

Abbildung 4.8: Die trilineare Filterung der einzelnen MIP-Texturen. Quelle: [3DC, 2001]

brilineares Filtern Der Bereich, in dem die Texturen ineinander übergeblendet werden, ist hier wesentlich geringer. Große Bereiche werden nur bilinear gefiltert [3DC, 2001]. Folglich wird beim Vergrößern des Objektes die Textur über einen gewissen Abstand skaliert und erst zum Schluss auf die nächste Textur übergeblendet. Übertragen auf die *Stippling*-Texturen bedeutet das eine Skalierung der einzelnen Punkte auf der Textur, die durch den Wechsel zu der nächsten Textur sichtbar wird. Der Einsatz des brilinearen Filters erhöht so wieder die Sichtbarkeit der einzelnen Detailstufen.

Bei der trilinearen Filterung vervierfacht sich folglich der Berechnungsaufwand und es müssen acht Texel pro Pixel gelesen werden. Um diesen Zugriff pro Pixel zu reduzieren, wird der brilineare Filter verwendet. Für die Texturierung der Oberflächenmodelle entsteht hierdurch allerdings eine Vergrößerung der Punkte

Abbildung 4.9: Die brilineare Filterung der einzelnen MIP-Texturen. Quelle: [3DC, 2001]

beim Skalieren der ersten Textur und ein auftauchen der kleineren Punkte der zweiten Textur, die noch nicht vergrößert wurden. Da die Punkte über die MIP-Stufen ihre Position beibehalten, taucht bei dem Überblenden ein kleiner schwarzer Punkt in der Mitte eines größeren Punktes auf (siehe Abbildung 4.6 (b)). Die Punkte auf den automatisch generierten MIP-Texturen vergrößern sich entsprechend einer höheren Texturauflösung, da sie zur Erzeugung der kleineren MIP-Stufen bilineare gefiltert wurden. Folglich sind keine störenden Übergänge bei einer Vergrößerung des Objektes mit der Textur sichtbar.

Ein weiters Problem ergibt sich für stark geneigte Flächen im Bezug zur Bildfläche. Dadurch ist der Bereich der Texturfläche auf den ein Pixel zugreift sehr groß und vor allem bei einer perspektivischen Verzerrung nicht mehr rechteckig. Wie in Abbildung 4.10 dargestellt, fallen in y Richtung mehr Texel auf einen Pixel als in x Richtung. Dementsprechend muss nicht wie bei der bilinearen und trilinearen Filterung in alle Richtungen gleich (isotrop), sondern anisotrop gefiltert werden.

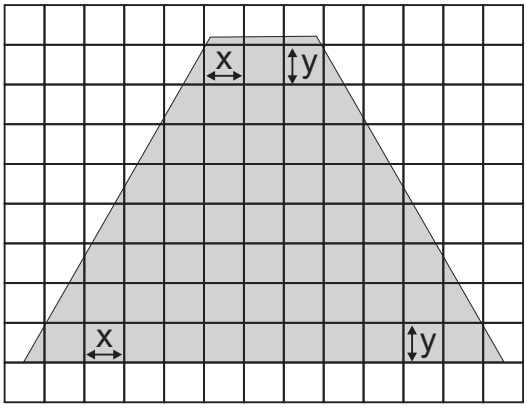


Abbildung 4.10: Bei stark geneigten Flächen und einer perspektivischen Verzerrungen muss die Textur in x und y Richtung unterschiedlich stark gefiltert werden. Quelle: [3DC, 2001]

Der anisotrope Filter arbeitet adaptiv. Je stärker die Fläche geneigt ist, desto mehr Texel bis zu einem eingestellten Maximalwert werden zur Darstellung eines Pixels eingelesen. Bei einer Abtastung von ca 16×16 Texeln und einem starken Neigungswinkel des Polygones wird, zur Vermeidung einer Überfilterung und der daraus resultierenden unschärferen Texturen, eine kleinere MIP-Textur

ausgewählt. Problematisch ist diese Technik, wenn das Objekt noch sehr nah am Betrachter positioniert ist. Da kleinere Texturen auch wenige Punkte besitzen, werden diese gestreckt und mögliche Muster durch Texturwiederholung sichtbar. Die Punkte erscheinen überdimensional groß und leicht verzerrt. Diese Nachteile sind bei den automatisch generierten MIP-Texturen nicht sichtbar, da kleinere MIP-Stufen einen fast einheitlich grauen Farbton besitzen beziehungsweise keine klar abgegrenzten Punkte mehr vorhanden sind.

Daraus resultiert, dass die Variante mit den automatisch erstellten MIP-Texturen aufgrund der internen Realisierung der Graphik-Hardware eine sehr gute Alternative darstellt. Obwohl hier nicht die Anforderungen an ein *Stippling*-Illustration mit konstanter Punktgröße umgesetzt werden können, eignet es sich dennoch zur Darstellung von medizinischen Oberflächenmodellen über *Stippling*. Die Form der Modelle wird durch die Punkte betont und erlaubt dem Betrachter eine genau Vorstellung der Ausdehnung und Lage der Organe im Bezug zu den umgebenden Strukturen.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine hardwarebeschleunigte Visualisierung medizinischer Oberflächenmodelle über *Stippling* entwickelt und implementiert. Die Anforderungen an traditionelle *Stippling*-Illustrationen und an interaktive NPR-Darstellungen wurden berücksichtigt und in der Umsetzung integriert.

Traditionelle *Pen-and-Ink*-Techniken sowie bereits existierenden Verfahren zur Generierung computergestützter *Stippling*-Illustrationen wurden untersucht, um die wesentlichen Anforderungen an eine interaktive Darstellung zu definieren. Aufbauend darauf konnte ein texturbasierter Ansatz entwickelt werden, der eine hardwaregestützte Darstellung der Oberflächenmodelle über *Stippling* erreicht. Texturen eignen sich besonders gut für eine framekohärente interaktive Visualisierung. Eine beliebige Menge an *Stippling*-Punkten kann, unabhängig von der Anzahl der Polygone oder Eckpunkte des Objektgitters und ohne den Berechnungsaufwand der gesamten Darstellung zu beeinflussen, auf der Oberfläche dargestellt werden. Es sind keine zeitaufwändigen Vorverarbeitungsschritte des Objektes notwendig, da die Anzahl der Punkte auf den Texturen festgelegt wird.

Die Abbildung der Texturen basiert auf dem Ansatz der *Polycubes*. Dieser konnte aufgrund der Beschaffenheit der *Stippling*-Texturen modifiziert werden. Eine Unterteilung des Objektraumes in Würfel erspart die aufwändigen Bearbeitungsschritte zur Umwandlung des Objektes in einen *Polycube* und erzielt die gleichen Vorteile für die Parametrisierung des Modelles. Die Berechnungen für die Texturabbildung erfolgen direkt auf der GPU und damit hardwarebeschleunigt. *Vertex* und *Fragment Shader* wurden hierfür in RenderMonkey mit der *OpenGL Shading Language* implementiert und können in jeder beliebigen *Shader*-Entwicklungsumgebung genutzt werden.

Zur Verteilung der *Stippling*-Punkte wurde eine *Poisson Disk*-Verteilung basierend auf zufällig generierten Punkten erzeugt. Gemäß den untersuchten Illustrationen erzielt diese Verteilung eine gleichmäßige Anordnung der Punkte mit annähernd äquidistanten Abständen. Texturen mit variierender Anzahl an Punkten repräsentieren verschiedene Helligkeiten des Objektes und betonen dementsprechend die Objektform. Der Aufbau der einzelnen Texturen orientiert sich an den TAMs von Praun et al. [2001] und gewährleistet die Kohärenz der Punkte zur Vermeidung störender *Shower Door*-Effekte während einer interaktiven Darstellung. Ferner

wurde die Integration der verschiedenen Helligkeitsstufen in einer Textur sowie die manuelle Generierung der dazugehörigen Skalierungsstufen umgesetzt. Aufgrund einer hardwareinternen Vereinfachung der Filter-Algorithmen werden beim Einsatz manuell erstellten MIP-*Maps* störende Artefakte sichtbar. Aus diesem Grund wurde zusätzlich eine Umsetzung mit automatisch erzeugten Skalierungsstufen implementiert, bei der die Probleme vernachlässigbar sind. Obwohl diese Texturen nicht alle Anforderungen einer *Stippling*-Darstellung erfüllen, wurden gute Ergebnisse erzielt, die eine echtzeitfähige Visualisierung von 3D-Oberflächenmodellen über *Stippling* ermöglichen.

5.1 Weiterführende Ansätze und Verbesserungsvorschläge

Ein Ansatz zur Verbesserung der Darstellung der entwickelten Texturen und der dazugehörigen Skalierungsstufen, ist die manuelle Selektion der MIP-Texturen. Hierdurch können die bei der automatischen Zuweisung durch die Hardware aufgetretenen Probleme verringert werden. In der *OpenGL Shading Language* ist es bisher nur möglich dies im *Vertex Shader* durch die Angabe des entsprechenden *Level of Deatils* in der Funktion `texture2DLod` vorzunehmen. Ein Texturzugriff im *Fragment Shader* liefert allerdings bessere Ergebnisse, da pro *Fragment* ein Texel bestimmt wird und dementsprechend genauere Werte berechnet werden können. Es ist folglich notwendig, die Berechnung des Pixelfarbwertes zu modifizieren und manuell zu realisieren.

Weiterhin wäre auch die Minimierung der in den *Shader*-Programmen ausgeführten Berechnungen reizvoll, die eine noch schnellere Verarbeitung und Visualisierung der Objekte erzielen kann. Die Berechnung der Texturkoordinaten außerhalb der *Shader* trägt zu einer Verringerung der Operationen und zu einer Beschleunigung der Darstellung bei.

Für eine Erweiterung der *Stippling*-Darstellung ist die Integration weiterer Kriterien zur Positionierung der Punkte wünschenswert. In traditionellen Illustrationen wird die Menge der dargestellten Punkte nicht nur abhängig von der reflektierten Intensität bestimmt. Entlang von Silhouetten und Merkmalslinien werden zusätzliche Punkte gesetzt, die die Objektform deutlicher hervorheben. Zusätzliche Informationen über die Krümmung der Objektoberfläche wären dafür hilfreich. Diese Daten könnten als Attribut eines jeden *Vertices* gespeichert werden und wären so für die einzelnen *Shader* verfügbar, um in Bestimmung der verwendeten Textur mit einzufließen.

Die Platzierung der Punkte auf den Texturen wird derzeit über eine *Poisson Disk*-Verteilung geregelt. Für eine hohe Anzahl an Punkten ergeben sich in einigen Fällen Muster auf dem Objekt. Eine Möglichkeit wäre die zusätzliche Generierung eines Voronoi-Diagrammes auf den bereits platzierten Punkten und die Implementierung des Algorithmus von Lloyd [1982]. Da die bereits durch eine *Poisson Disk*-Verteilung platzierten Punkte dann noch anhand ihrere Voronoi-Regionen verteilt werden, erfolgt eine Verbesserung und Annäherung an äquidistante Abstände. Allerdings sind, wie bereits im Konzept beschrieben, einige Änderungen zu integrieren, um die Kohärenz der Punkte zwischen den Texturen beizubehalten (siehe Seite 62). Ferner müsste getestet werden, welcher der beiden beschriebenen Ansätze die besseren Ergebnisse im Bezug auf die gleichmäßige Verteilung der Punkte liefert.

Weiterführende Untersuchungen im Bezug auf die spezifische Verwendung von *Stippling*-Darstellungen für bestimmte anatomische Strukturen, die Auswahl der Punktgröße sowie die Kombination mit Farbe sind wichtig, um die Vorteile dieser Darstellungsweise weiter auszubauen. Mit dem vorgestellten *Fragment*-Programm wurde bereits die Möglichkeit geschaffen, nicht nur schwarze, sondern beliebig farbige Punkte zu setzen. Zusätzliche Analysen über die Vor- und Nachteile einer farbigen *Stippling*-Visualisierung wären ebenfalls sehr interessant.

Stippling-Illustrationen werden in anatomischen Atlanten zur Visualisierung von Kontextinformationen verwendet und liefern in einer 3D-Darstellung zusätzliche Informationen über die Ausdehnung und Lage der Objekte, ohne dabei von den wesentlichen fokusnahen Strukturen abzulenken. In dieser Arbeit wurde eine hardwaregestützte *Stippling*-Darstellung von 3D-Oberflächenmodelle umgesetzt, die unabhängig von der Komplexität des Objektgitters ist. Die erzielten Ergebnisse sowie die weiterführenden Möglichkeiten und Ansätzen zeigen, dass es lohnenswert ist, diese Darstellungsmethoden auszubauen und weiterzuentwickeln.

Abbildungsverzeichnis

Literaturverzeichnis

[3DC 2001] *3DCenter : Grafik-Filter*. Dezember 2001

[Akenine-Moller et al. 2002] Akenine-Moller, Tomas ; Moller, Tomas ; Haines, Eric: *Real-Time Rendering*. A. K. Peters, Ltd., 2002. – ISBN 1568811829

[Bertolini et al. 1995] Bertolini, Rolf ; Leutert, Gerald ; Rother, Paul ; Scheuner, Georg ; Wendler, Dietmar ; Bertolini, Rolf (Hrsg.): *Systematische Anatomie des Menschen*. Bd. 5. Ullstein Mosby GmbH & Co. KG, 1995

[Bier und Sloan 1986] Bier, Eric A. ; Sloan, Ken R.: Two-Part Texture Mapping. In: *IEEE Computer Graphics and Applications* 6 (1986), September, Nr. 9, S. 40–53

[Carr und Hart 2002] Carr, Nathan A. ; Hart, John C.: Meshed atlases for real-time procedural solid texturing. In: *ACM Transactions on Graphics* 21 (2002), Nr. 2, S. 106–131. – ISSN 0730–0301

[Cook 1986] Cook, Robert L.: Stochastic sampling in computer graphics. In: *ACM Transactions on Graphics* 5 (1986), Nr. 1, S. 51–72. – ISSN 0730–0301

[Deussen et al. 2000] Deussen, Oliver ; Hiller, Stefan ; Overveld, Cornelius van ; Strothotte, Thomas: Floating Points: A Method for Computing Stipple Drawings. In: *Computer Graphics Forum* 19 (2000), Nr. 3, S. 40–51

[Dong et al. 2003] Dong, Feng ; Clapworthy, Gordon J. ; Lin, Hai ; Krokos, Meleagros A.: Nonphotorealistic Rendering of Medical Volume Data. In: *IEEE Computer Graphics and Applications* 23 (2003), Nr. 4, S. 44–52

[Ebert und Rheingans 2000] Ebert, David ; Rheingans, Penny: Volume Illustration: Non-Photorealistic rendering of Volume Models. In: *VIS '00: Proceedings of the conference on Visualization 2000*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2000. – ISBN 1–58113–309–X, S. 195–202

[Ebert et al. 2003] Ebert, David S. ; Musgrave, F. K. ; Peachey, Darwyn ; Perlin, Ken ; Worley, Steven: *Texturing & Modeling: A Procedural Approach*. Morgan-Kaufmann, 2003

[Fernando und Kilgard 2003] Fernando, Randima ; Kilgard, Mark J.: *The Cg Tutorial :The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003

[Floater und Hormann 2004] Floater, Michael S. ; Hormann, Kai: Surface Parameterization: a Tutorial and Survey. In: Dodgson, N.A. (Hrsg.) ; Floater, Michael S. (Hrsg.) ; Sabin, M.A. (Hrsg.): *Advances in Multiresolution for Geometric Modelling*, Springer Verlag, 2004, S. 157–186

[Foley et al. 1996] Foley, James D. ; Dam, Andries van ; Feiner, Steven K. ; Hughes, John F.: *Computer Graphics principles and practice (2nd Edition in C)*. Addison-Wesley Longman Publishing Co., Inc., 1996. – ISBN 0–201–84840–6

[Foley et al. 1994] Foley, James D. ; Dam, Andries van ; Feiner, Steven K. ; Hughes, John F. ; Phillips, Richard L.: *Grundlagen der Computergraphik, Einführung, Konzepte, Methoden*. Addison-Wesley, 1994. – FOL j 94:1 1.Ex. – ISBN 3–89319–647–1

[Freudenberg et al. 2002] Freudenberg, Bert ; Masuch, Maic ; Strothotte, Thomas: Real-Time Halftoning: A Primitive for Non-Photorealistic Shading. In: *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2002. – ISBN 1–58113–534–3, S. 227–232

[Gerstner 2001] Gerstner, Thomas: Fast Multiresolution Extraction of Multiple Transparent Isosurfaces. In: Peikert, Ronald (Hrsg.) ; Eber, David S. (Hrsg.) ; Favre, Jean M. (Hrsg.): *VisSim 01: Proceedings of Data Visualization 2001*, 2001, S. 35–44

[Heckbert 1986] Heckbert, Paul S.: Survey of Texture Mapping. In: *IEEE Computer Graphics and Applications* 6 (1986), November, Nr. 11, S. 56–67

[Hodges 1989] Hodges, Elaine: *The Guild Handbook of Scientific Illustration*. John Wiley and Sons, 1989

[Interrante et al. 1995] Interrante, Victoria ; Fuchs, Henry ; Pizer, Stephen: Enhancing Transparent Skin Surfaces with Ridge and Valley Lines. In: *VIS '95: Proceedings of the 6th conference on Visualization 1995*. Washington, DC, USA : IEEE Computer Society, 1995. – ISBN 0–8186–7187–4, S. 52

[Klein et al. 2000] Klein, Allison W. ; Li, Wilmot W. ; Kazhdan, Michael M. ; Correa, Wagner T. ; Finkelstein, Adam ; Funkhouser, Thomas A.: Non-Photorealistic Virtual Environments. In: Akeley, Kurt (Hrsg.): *SIGGRAPH '00: Computer Graphics Proceedings*, ACM Press, Addison Wesley Longman, 2000, S. 527–534

[Koschatzky 1993] Koschatzky, Walter ; Albertina, Graphische S. (Hrsg.): *Die Kunst der Zeichnung*. Residenz Verlag, 1993

[Lake et al. 2000] Lake, Adam ; Marshall, Carl ; Harris, Mark ; Blackstein, Marc: Stylized Rendering Techniques for Scalable Real-Time 3D Animation. In: *NPAR '00: Non-Photorealistic Animation and Rendering 2000*. Annecy, France, June 2000

[Lansdale 1991] Lansdale, Robert C.: *Texture Mapping and Resampling for Computer Graphics*. Toronto,Canada, Department of Electrical Engineering University of Toronto, Diplomarbeit, Januar 1991

[Levoy 1988] Levoy, Marc: Display of Surfaces from Volume Data. In: *IEEE Computer Graphics and Application* 8 (1988), Nr. 3, S. 29–37. – ISSN 0272–1716

[Lloyd 1982] Lloyd, Stuart P.: Least squares quantization in PCM. In: *IEEE Transactions on Information Theory* 28 (1982), Nr. 2, S. 129–136

[Lorensen und Cline 1987] Lorensen, William E. ; Cline, Harvey E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 1987. – ISBN 0–89791–227–6, S. 163–169

[Lu et al. 2002] Lu, Aidong ; Morris, Christopher J. ; Ebert, David S. ; Rheingans, Penny ; Hansen, Charles: Non-Photorealistic Volume Rendering using Stippling Techniques. In: *VIS '02: Proceedings of the conference on Visualization 2002*. Washington, DC, USA : IEEE Computer Society, 2002. – ISBN 0–7803–7498–3, S. 211–218

[Mark et al. 2003] Mark, William R. ; Glanville, R. S. ; Akeley, Kurt ; Kilgard, Mark J.: Cg: A System for Programming Graphics Hardware in a C-like Language. In: *SIGGRAPH '03: Proceedings of ACM*, 2003

[McCool und Fiume 1992] McCool, Michael ; Fiume, Eugene: Hierarchical Poisson Disk Sampling Distributions. In: *Proceedings of the conference on Graphics Interface '92*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1992. – ISBN 0–9695338–1–0, S. 94–105

[McGuire und Hughes 2004] McGuire, Morgan ; Hughes, John F.: Hardware-determined Feature Edges. In: *Non-Photorealistic Animation and Rendering 2004: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–887–3, S. 35–147

[Miller und Hoffmann 1984] Miller, Gene S. ; Hoffmann, C. R.: Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments. In: *SIGGRAPH '84: Advanced Computer Graphics Animation courses notes*, 1984

[Mitchell und Card 2002] Kap. 3: Pixel Shader Tricks In: Mitchell, Jason L. ; Card, Drew: *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware Publishing Inc., 2002, S. 319–333

[NVI 2002] NVIDIA: *Cg Toolkit : A Developer's Guide to Programmable Graphics*. 2002

[Pastor 2003] Pastor, Oscar Ernesto M.: *Frame-Coherent 3D Stippling for Non-Photorealistic Computer Graphics*, Otto-von-Guericke-Universität Magdeburg, Diss., 2003

[Pastor et al. 2003] Pastor, Oscar Ernesto M. ; Freudenberg, Bert ; Strothotte, Thomas: Real-Time Animated Stippling. In: *IEEE Computer Graphics and Applications* 23 (2003), Nr. 4, S. 62–68

[Praun et al. 2000] Praun, Emil ; Finkelstein, Adam ; Hoppe, Hugues: Lapped Textures. In: *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000, S. 465–470

[Praun und Hoppe 2003] Praun, Emil ; Hoppe, Hugues: Spherical Parametrization and Remeshing. In: *ACM Transactions on Graphics* 22 (2003), Nr. 3, S. 340–349. – ISSN 0730–0301

[Praun et al. 2001] Praun, Emil ; Hoppe, Hugues ; Webb, Matthew ; Finkelstein, Adam: Real-Time Hatching. In: Fiume, Eugene (Hrsg.): *SIGGRAPH '01: Computer Graphics Proceedings*, 2001, S. 579–584

[Proudfoot et al. 2001] Proudfoot, Kekoa ; Mark, William R. ; Tzvetkov, Svetoslav ; Hanrahan, Pat: A Real-Time Procedural Shading System for Programmable Graphics Hardware. In: *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 2001. – ISBN 1–58113–374–X, S. 159–170

[Rogers 1992] Rogers, Andrew W.: *Textbook of Anatomy*. Churchill Livingstone, 1992

[Rost 2004] Rost, Randi J.: *OpenGL(R) Shading Language*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 2004. – ISBN 0321197895

[Salisbury et al. 1994] Salisbury, Mike ; Anderson, Sean ; Barzel, Ronen ; Salesin, David: Interactive Pen-and-Ink Illustration / University of Washington Department of Computer Science and Engineering. Seattle, Washington 98195, April 1994. – Forschungsbericht

[Secord 2002] Secord, Adrian: Weighted Voronoi Stippling. In: *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, ACM Press, 2002. – ISBN 1–58113–494–0, S. 37–43

[Sousa und Prusinkiewicz 2003] Sousa, Mario C. ; Prusinkiewicz, Przemyslaw: A Few Good Lines: Suggestive Drawing of 3D Models. In: *Computer Graphics Forum* 22 (2003), Nr. 3, S. 381–390

[Stone und Stone 2000] Stone, Robert J. ; Stone, Judith A.: *Atlas of Skeletal Muscles*. Bd. 3. McGraw-Hilligher Education, 2000

[Strothotte und Schlechtweg 2002] Strothotte, Thomas ; Schlechtweg, Stefan: *Non-Photorealistic Computer Graphics - Modelling, Rendering, and Animation*. Morgan Kaufmann, 2002

[Tarini et al. 2004] Tarini, Marco ; Hormann, Kai ; Cignoni, Paolo ; Montani, Claudio: PolyCube-Maps. In: *ACM Transactions on Graphics* 23 (2004), Nr. 3, S. 853–860. – ISSN 0730–0301

[Tietjen 2004] Tietjen, Christian: *Evaluierung und Modifikation von Methoden zur Generierung von Liniengrafiken in der medizinischen Visualisierung*, Institut für Simulation und Graphik, Otto-von-Guericke Universität Magdeburg, Diplomarbeit, 2004

[Tönnies und Lemke 1994] Tönnies, Klaus D. ; Lemke, Heinz U.: *3D-Computergrafische Darstellungen*. Oldenbourg Verlag, 1994

[Watt 2002] Watt, Alan: *3D-Computergrafik*. 3. Addison-Wesley, 2002

[Webb et al. 2002] Webb, Matthew ; Praun, Emil ; Finkelstein, Adam ; Hoppe, Hugues: Fine Tone Control in Hardware Hatching. In: *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–494–0, S. 53–ff

[Williams 1983] Williams, Lance: Pyramidal Parametrics. In: *SIGGRAPH '83: Computer Graphics Proceedings* 17 (1983), Juli, Nr. 3, S. 1–11

[Winkenbach und Salesin 1994] Winkenbach, Georges ; Salesin, David H.: Computer-Generated Pen-and-Ink Illustration. In: *Computer Graphics Forum* 28 (1994), Nr. Annual Conference Series, S. 91–100

[Yuan und Chen 2004] Yuan, Xiaoru ; Chen, Baoquan: Illustrating Surfaces in Volume. In: *VisSym'04: Proceedings of Joint IEEE/EG Symposium on Visualization*, the Eurographics Association, 2004. – ISBN 3–905673–07–X, S. 9–16, color plate 337

A Resultate

A.1 Teapot

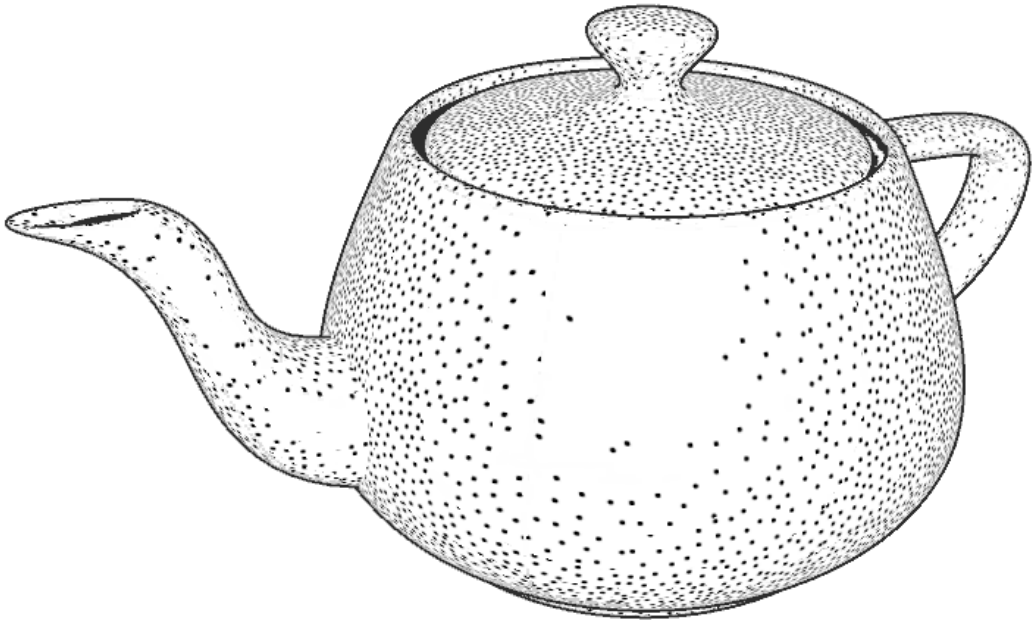
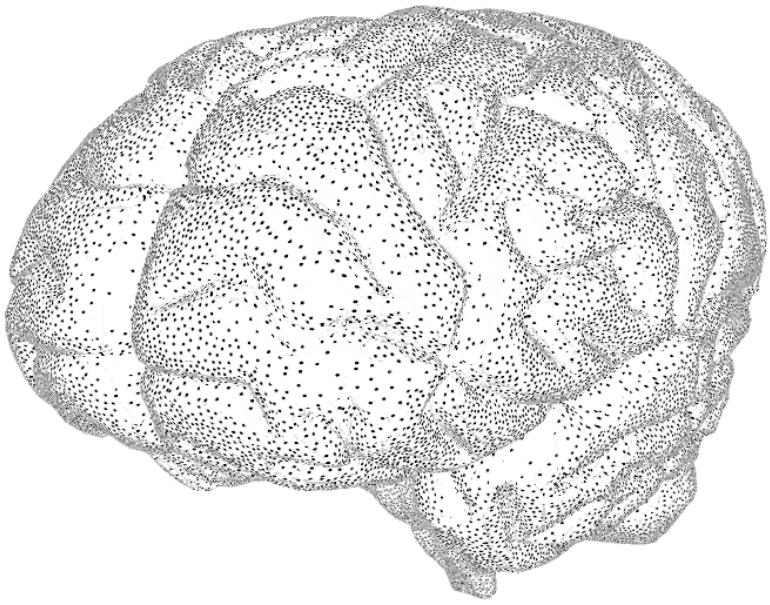
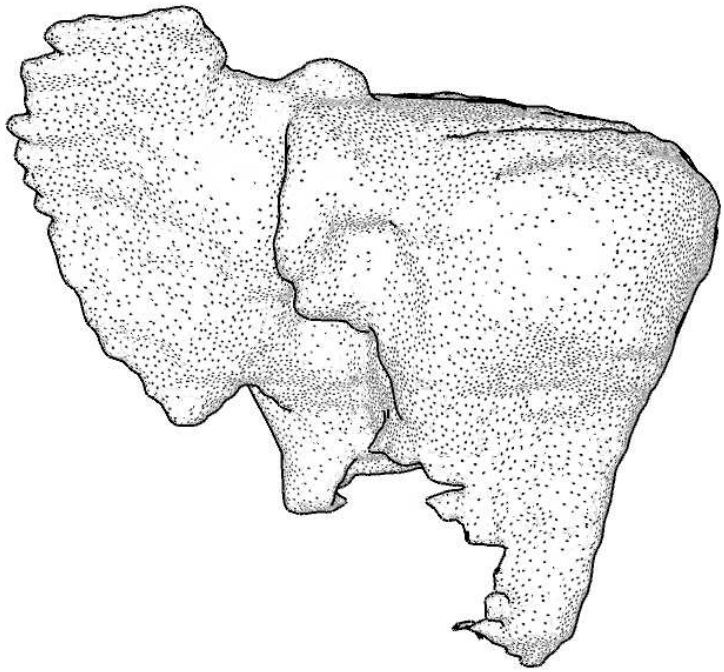


Abbildung A.1: Der *Teapot* hier mit Silhouetten, wodurch die Objektgrenzen hervorgehoben werden.

A.2 Medizinische Modelle



(a)



(b)

Abbildung A.2: (a) Die *Stippling*-Darstellung des Gehirnes (b) sowie der Leber mit 7 Texturstufen und einer Größe von 512×512 Pixeln.