Diplomarbeit

Bumpmapping-Verfahren und deren Weiterentwicklung

eingereicht bei

Prof. Dr.-Ing. Detlef Krömker,

Professur für Graphische Datenverarbeitung

von

Sebastian Schäfer

Eingereicht am:
21.03.2007

JOHANN WOLFGANG GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

Betreuung durch
Dr. Tobias Breiner

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Frankfurt am Main, den 21. März 2007

_____

(Sebastian Andreas Schäfer)

Zusammenfassung

Das Bumpmapping-Verfahren, eine Methode zur realistischen Darstellung rauer Oberflächen, existiert schon seit 30 Jahren, aber erst durch aktuelle Entwicklungen der Hardware lässt es sich in Echtzeitumgebungen einsetzen. Die aktuellen Verfahren ermöglichen viele darüber hinausgehende Effekte, jedoch haben sie auch mit Problemen zu kämpfen. Das Ziel dieser Diplomarbeit ist die Weiterentwicklung der Verfahren zu betrachten. In dieser Arbeit werden die Grenzen der aktuellen Bumpmapping-Algorithmen aufgezeigt und nach neuen Wegen geforscht. Das erste Verfahren erzeugt durch ein Multipassrendern fraktale Landschaften im Shader. Die darin verwendeten Methoden lassen sich für einen weiteren Algorithmus nutzen, mit dem feine Unebenheiten der Oberfläche an jedem Pixel ausgewertet werden. So können anisotrope Materialien wie gebürstete Metalle oder Mikropartikellacke simuliert werden. Den Abschluss bilden zwei neue Verfahren für prozedurale Shader. Die zu imitierende Oberfläche wird im Modell nachgebildet und per Raytracing für jeden Oberflächenpixel ausgewertet. Durch diese Methode werden viele Probleme texturbasierter Verfahren komplett umgangen.

Abstract

The bumpmapping algorithm produces realistically looking rendered images of wrinkled surfaces. It was invented almost 30 years ago. Only quite recently though the algorithm could be implemented on realtime rendering hardware. State of the Art implementations are capable of creating various additional effects but they are not free of problems. The main goal of this work is to explore further enhancements of bumpmapping algorithms. The first presented method creates fractal landscapes in a multi-pass render. Parts of this algorithm can be used in a second shader which creates a sub-pixel surface per pixel and uses it's normals to create anisotrop material effects such as brushed metal or microparticle paint. The last algorithms shown use an alternative way for procedural shaders: the surface of the material is modeled inside the shader and raytraced per pixel. This method avoids almost all problems common with texture-based algorithms.

Danksagung

Mein Dank gilt allen, die mich in meinem Studium unterstützt und gefördert haben.

Zuallererst bedanke ich mich bei meinen Eltern und meiner Familie. Sie standen mir jederzeit mit Rat und Tat zur Seite und haben mir das Studium erst ermöglicht.

Bedanken möchte ich mich auch bei allen, die die endlosen Mühen des Korrekturlesens auf sich genommen haben: Sarah, Daniel, Daniel, Frederik, Gero, Lucas und Patrick.

Mein weiterer Dank gilt Dr. Tobias Breiner für seine Betreuung und Ratschläge, sowie Prof. Dr.-Ing. Detlef Krömker, der diese Arbeit erst ermöglicht hat.

Nicht zuletzt bedanke ich mich bei meiner Freundin Hana – für einfach alles!

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

„Phantasie ist wichtiger als Wissen, denn Wissen ist begrenzt."

Albert Einstein

Schon immer hat der Mensch versucht, die in seiner Phantasie entstandenen Bilder und Geschichten mit anderen zu teilen. Es ist daher nicht verwunderlich, dass seitdem das erste Bild mit einem Computer erzeugt wurde, die Computergrafik eines der wichtigsten Antriebsmittel in der Entwicklung des Computers ist. Zu groß ist das Verlangen, der Drang, mittels eines Computers seiner Phantasie Ausdruck zu verleihen, als dass diese Möglichkeit ungenutzt bleiben dürfte. Unzählige Verfahren wurden entwickelt, um der Rechenmaschine immer glaubwürdigere Bilder zu entlocken. Schon seit einiger Zeit ist es möglich, täuschend echte - „photorealistische" - Bilder zu rendern, wie dies unzählige Hollywood-Filme belegen. Durch die zunehmend steigende Leistungsfähigkeit der Computer rücken diese Möglichkeiten immer mehr in den Bereich normaler Anwender vor. Viele Desktoprechner verfügen schon jetzt über die geeignete Hardware, um sehr realistische Bilder in einer interaktiven Umgebung zu produzieren.

Das Bumpmapping-Verfahren trägt wesentlich zu diesem Ergebnis bei. Mit ihm existiert schon seit 1978 ein Verfahren, das schnell arbeitet und den Realismus von Bildern durch die korrekte Beleuchtung von unebenen Oberflächen deutlich erhöht.

Diese Diplomarbeit untersucht die Weiterentwicklungsmöglichkeiten des Bumpmapping-Verfahrens. Nachdem die Grundlage für die weitere Arbeit gelegt

wurde, werden verschiedene aktuelle Bumpmapping-Verfahren in der State of the Art-Analyse verglichen.

Die eigenen Verfahren versuchen, das Bumpmapping um verschiedene Aspekte zu erweitern. Die erste Methode verdeutlicht grundsätzliche Probleme der aktuellen Bumpmapping-Algorithmen, indem sie ein State of the Art Verfahren mit einem prozedural generierten Backsteinmuster verbindet. Um diesem und anderen Materialien eine rauere Oberfläche zu verleihen, beschäftigt sich das zweite Verfahren mit der Erzeugung von realistischen Landschaftsstrukturen. Die angewandten Methoden werden in einem dritten Verfahren benutzt, um anisotrope Materialien darzustellen. Die grundsätzlichen Fehler der aktuellen Algorithmen eliminiert eine vierte Herangehensweise, die nicht auf Texturen, sondern auf strukturellen mathematischen Beschreibungen basiert und so Oberflächenunebenheiten darstellt.

Alle eigenen Verfahren werden abschließend bewertet und in einem Ausblick auf weitere Entwicklungen hin betrachtet.

Kapitel 2

Grundlagen

„Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.“

Douglas Adams

Das zweite Kapitel erklärt die für die weitere Arbeit notwendigen Grundlagen und Techniken.

Zunächst wird die Renderpipeline kurz vorgestellt. Die darin enthaltenen Techniken Texturemapping und verschiedene Beleuchtungsmodelle werden im Detail besprochen. Anschließend werden Shader erklärt und ihre Stellung in Hinblick auf die Renderpipeline verdeutlicht. Den Abschluss bildet die Vorstellung von einem Algorithmus zur Generierung von fraktalen Höhenzügen und Landschaften.

2.1 Die Renderpipeline

Die Renderpipeline (wie in [Wat02], [Ros04] oder [FDHD90] erörtert) beschreibt den Weg aller zu rendernden Geometriedaten bis hin zu den Pixeln des finalen Bildes. Sie besteht aus verschiedenen Transformations- und Berechnungsphasen, die grob in zwei Gruppen eingeteilt werden können: Methoden für den Objektraum und Methoden für den Bildraum.

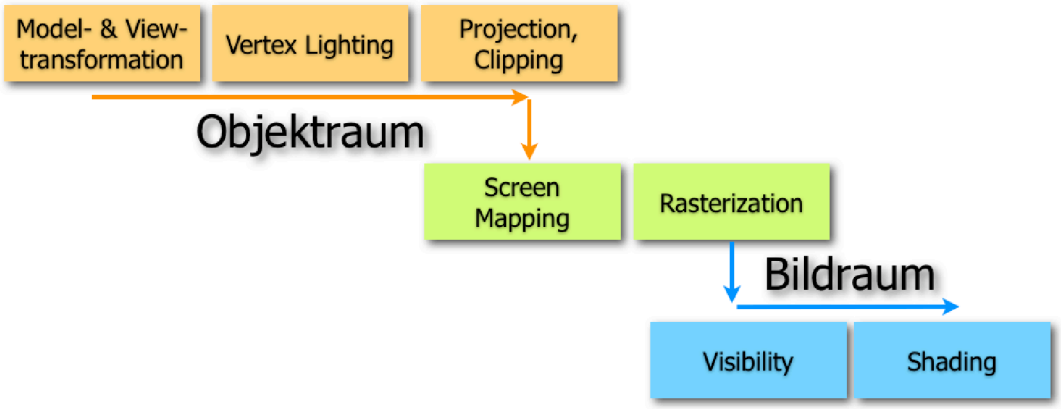

Abbildung 2.1: Renderpipeline

Die Methoden im Objektraum behandeln die Vertices eines Objektes. Model- und Viewtransformation sorgen für die Positionierung der Modelle an der Zielposition im Raum. Im Vertex–Lighting-Schritt wird die Beleuchtungsrechnung für jeden Vertex durchgeführt. Schließlich wird in der Projektionsphase jeder Vertex mit der Projektionsmatrix multipliziert und beim Clipping die nicht sichtbaren Geometrieteile entfernt.

Nachdem im Screenmapping der sichtbare Bildschirmbereich bestimmt wurde, werden die von Vertices begrenzten Flächen im Rasterisierer in Pixel überführt.

Alle folgenden Methoden behandeln den Bildraum. Die Visibility-Stufe entfernt verdeckte Pixel. Anschließend wird im Shading für jeden sich noch in der Renderpipeline befindlichen Pixel eine Farbe berechnet – je nach Textur, Beleuchtung oder sonstigen Merkmalen.

2.2 Texturen

Eine Textur ist eine diskrete Speicherstruktur für diskrete Werte. Die Textur wird in Dimensionen aufgeteilt und jeder Vertex bekommt eine entsprechend gleichdimensionale Texturkoordinate. Beispielsweise benötigt eine zweidimensionale Textur auch zweidimensionale Texturkoordinaten. Nach Konvention haben alle Achsen im Koordinatensystem eine Länge von eins, beginnend bei null. Üblicherweise speichern Texturen ein zweidimensionales Bild mit einem roten, grünen und blauen Kanal - die Semantik einer Textur ist durch den Benutzer frei bestimmbar.

2.2.1 Texturemapping

Das Texturemapping von Catmull [Cat74] und Blinn [BN76] ist ein Verfahren, um ein Muster auf eine Oberfläche zu bringen. Der Prozeß kann mit dem Anbringen eines Aufklebers verglichen werden. Die Textur wird nach einer bestimmten Regel auf die Oberfläche projiziert. Dieser Vorgang wird „Mapping“ genannt.

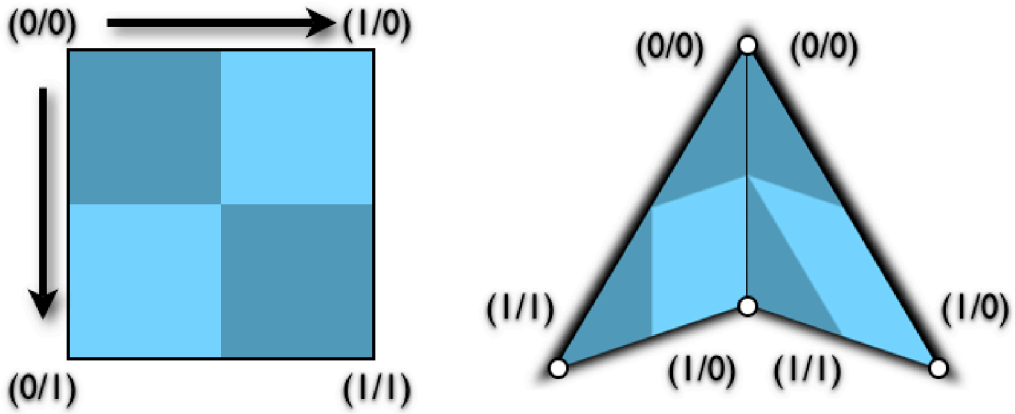

Abbildung 2.2: Texturemapping

2.2.2 Cubemapping

Eine interessante Variante des Texturemappings ist das Cubemapping. Die Textur besteht hier aus sechs Stücken, die zusammengefügt einen Würfel ergeben, dessen Kanten – ohne sichtbaren „Riss“ - aneinander anliegen. Anwendung findet das Cubemapping im von Voorhies und Foran [VF94] entwickelten „Environment Cubemapping“. Der Würfel umhüllt die Szene (der Betrachter befindet sich also innen) und ist so ausgerichtet, dass die Normalen entlang der Hauptachsen zeigen. Auf die Cubemap wird über den an der Oberfläche reflektierten Sichtstrahl zugegriffen. Formel 2.1, deren Vektoren in Abbildung 2.4 erklärt werden, dient der Berechnung dieses reflektierten Vektors. Die dominierende Komponente bestimmt, welche der sechs Cubemapteile benutzt wird, die anderen beiden bestimmen den Texel. Als Texel bezeichnet man den Bildpunkt einer Textur

$$R \quad = \quad 2 * N * (N \cdot C) - C$$

Formel 2.1: Reflektanzvektor R

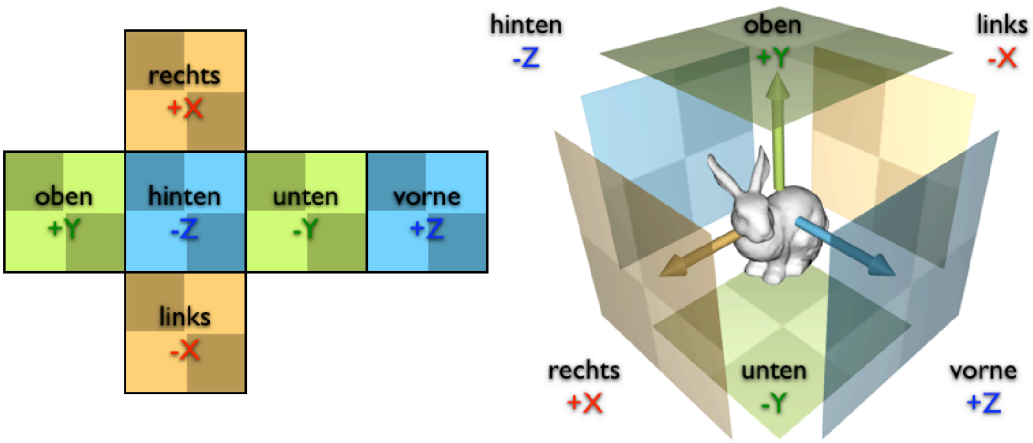

Abbildung 2.3: Cubemap und Cubemapping

2.3 Beleuchtungsrechnung

Bei polygonalen Renderern ist die Beleuchtungsrechnung üblicherweise in drei verschiedene Lichtarten aufgeteilt: ambientes, diffuses und spiegelndes Licht. Im Folgenden werden die Beleuchtungsrechnungen von OpenGL, wie sie in [WDS99] beschrieben sind, erklärt; Direct3D benutzt identische Gleichungen. Emittierendes Licht wird in den weiteren Betrachtungen vernachlässigt, da dies keine Beleuchtungsform, sondern wie eine Materialeigenschaft modelliert wird. Jede Farbe wird durch einen dreidimensionalen Vektor repräsentiert, dessen Werte den roten, grünen und blauen Farbanteil abspeichern. Die Formeln benutzen die im englischen Sprachraum gebräuchlichen Bezeichnungen aus Gründen der Konsistenz mit der Literatur.

Die gesamte Beleuchtung eines Objektes ergibt sich aus der Addition aller benutzten Beleuchtungsmethoden:

$$farbe_{total} = min(1, farbe_{ambient} + farbe_{diffuse} + farbe_{specular})$$

Formel 2.2: Addition der Teilbeleuchtungen

Alle Formeln des folgenden Abschnitts basieren auf den in Abbildung 2.4 dargestellten Vektoren.

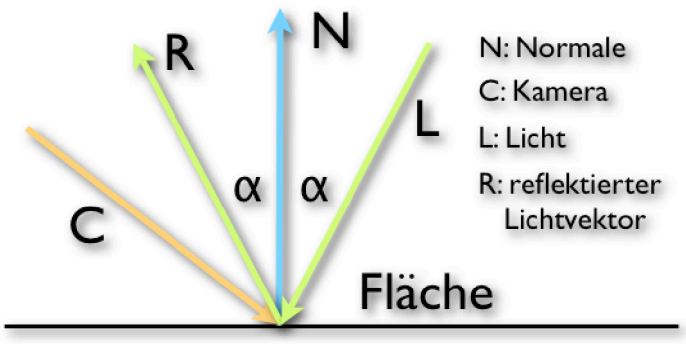

Abbildung 2.4: Vektoren der Beleuchtungsverfahren

2.3.1 Ambiente Beleuchtung

Die ambiente Beleuchtung steht für den Teil des Lichtes, der allgegenwärtig ist und von keiner bestimmten Quelle ausgeht – das Umgebungslicht. Die Bestimmung des ambienten Farbanteils eines Objektes ergibt sich aus:

$$farbe_{ambient} = Lichtfarbe_{ambient} * Materialfarbe_{ambient}$$

Formel 2.3: Formel für ambientes Licht

2.3.2 Diffuse Beleuchtung

Gerichtetes Licht, das aus einer bestimmten Richtung auf ein Objekt trifft, wird durch die diffuse Beleuchtung modelliert. Sie ist auch als das Lambertsche Beleuchtungsmodell bekannt.

$$farbe_{diffuse} = Lichtfarbe_{diffuse} * Materialfarbe_{diffuse} * max(0, L \cdot N)$$

Formel 2.4: Formel der diffusen Beleuchtung

2.3.3 Glanzlichter

Spiegelndes Licht wird von einem einzelnem Punkt, einer Punktlichtquelle, ausgesandt, die sich auf dem Material spiegelt. Im Gegensatz zu den beiden vorherigen ist die Formel für Glanzlichter von empirischer Natur und nicht durch die Physik begründet und stammt von Phong [Pho75].

$$farbe_{specular} = Lichtfarbe_{specular} * Materialfarbe_{specular} * max(0, R \cdot C)^{shininess}$$

Formel 2.5: Phong'sche Beleuchtungsformel

In beiden APIs findet Beleuchtungsrechnung im Objektraum pro Vertex statt. Glanzeffekte sind nicht auf den Oberflächen selbst möglich – dort wird nur der an den benachbarten Vertices berechnete Wert interpoliert (Abbildung 2.5).

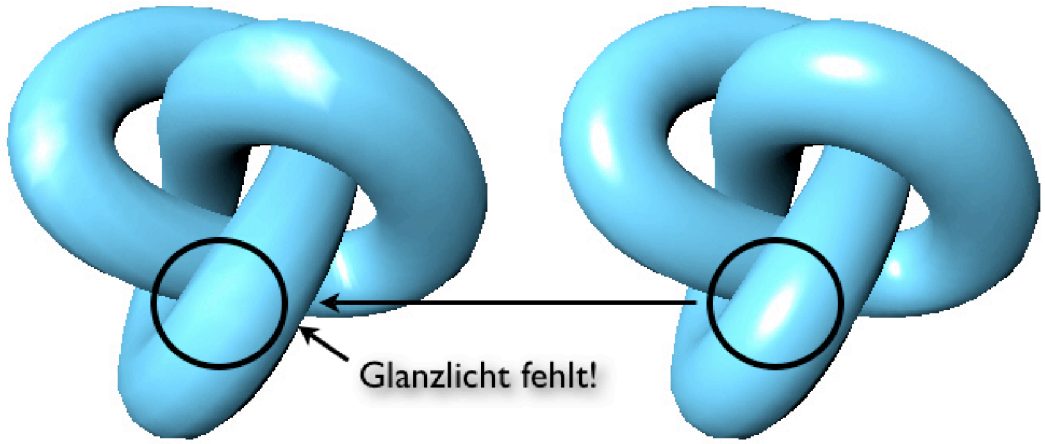

Abbildung 2.5: Vertexlighting und Pixellighting

2.4 Shader

Der Begriff Shader stammt von R. Cook aus dem Jahre 1984 [Coo84] und bezeichnet kleine Programme, die Teile der Renderpipeline ersetzen. Das Rendersystem, in dem Shader zuerst eingesetzt wurden, ist Pixars RenderMan [Pix88], ein sehr leistungsfähiges Offline-Rendersystem, das in vielen Hollywood-Filmproduktionen nach wie vor eingesetzt wird.

Innerhalb dieser Diplomarbeit werden ausschließlich Shader betrachtet, die in den aktuellen Echtzeit–3D-APIs Direct3D und OpenGL verwendet werden können. Sie wurden im Jahre 2000 mit dem Erscheinen von DirectX 8.0 eingeführt [Hor00] und seitdem stetig weiterentwickelt. Da die Entwicklung der Shader von der Hardwareumsetzung stark abhängig ist, haben beide APIs Shader (beinahe) gleichzeitig in ihre Sprachdefinition aufgenommen.

Shaderprogramme werden in die GPU geladen und dort an entsprechender Stelle innerhalb der API-Renderpipeline ausgeführt. Bei der GPU handelt es sich um einen SIMD-Prozessor (nach [Fly95]), der Vertices und Pixel massiv parallel bearbeitet. Ferner ist die GPU ein Vektorprozessor, dessen Variablen aus mehreren Komponenten bestehen und dessen Befehle meist mehrere Komponenten beeinflussen. Operationen wie Vektoraddition, -Subtraktion und -Multiplikation können daher besonders schnell ausgeführt werden. Spezielle Vektorbefehle vereinfachen die Programmierung zusätzlich.

Es gibt zwei verschiedene Typen von Shadern: Vertex- und Pixelshader. Diese werden zunächst vorgestellt und anschließend wird auf deren Programmier-sprachen eingegangen.

2.4.1 Vertexshader

Vertexshader werden für jeden zu rendernden Vertex aufgerufen und operieren im Objektraum der Renderpipeline.

Zu den Aufgaben eines Vertexshaders zählen gemäß Rost [Ros04]:

- Vertextransformationen (Projektion, Manipulation und anderes),
- Normalentransformation,
- Texturkoordinatengenerierung bzw. deren Manipulation und
- Beleuchtungsvorberechnungen.

Die Eingabe für jeden Vertexshader ist mindestens die Position des Vertex, daneben können noch andere Parameter übergeben werden. Diese sind sowohl vom Modell (Normale, Texturkoordinaten, Tangente, Binormale, etc.) als auch vom Programm (Projektionsmatrix, Lichtposition und andere frei bestimmbare Werte) abhängig.

Als einzige obligatorische Ausgabe fällt die Angabe der Finalposition des Eingabevertex an. Daneben können aber weitere, in Grenzen[1] frei wählbare Werte berechnet und als Eingabe für Pixelshader verwendet werden.

2.4.2 Pixelshader

Ein Pixelshader wird für jeden gerasterten Pixel aufgerufen. Er arbeitet somit direkt nach dem Rasterisierer im Bildraum der Renderpipeline. Seine Aufgabe ist es, die Einfärbung jedes Pixels zu berechnen. Neuere Versionen des Pixelshaders erlauben neben dem direkten Rendern in eine Textur auch die Ausgabe mehrerer Werte, welche gleichzeitig in verschiedene Texturen gespeichert werden können (Multiple-Render-Target).

1 Die Anzahl der Parameter, die zwischen Vertex- und Pixelshader ausgetauscht werden können, ist begrenzt, jedoch nicht deren Semantik.

Die Aufrufparameter jedes Pixelshaders sind die Ausgabewerte des Vertexshaders. Da sie pro Vertex, nicht aber pro Pixel berechnet wurden, werden sie vom Rasterisierer interpoliert. Daneben besteht – genau wie im Vertexshader – die Möglichkeit, dem Shader vom ausführenden Programm Parameter zu übergeben.

2.4.3 Shadersprachen

Während die ersten Shader noch mit einer LowLevel Assembler Sprache sehr hardwarenah programmiert werden mussten, haben sich seit 2002 drei verschiedene C-ähnliche Hochsprachen entwickelt: GLSL,[2] HLSL[3] und NVIDIA Cg.[4] Da alle die gleiche SIMD-Hardware, die GPU, benutzen, überrascht es nicht, dass die Sprachen einander sehr ähnlich sind. Die Unterschiede sind nur marginal, so dass die Programme leicht von einer in die andere Shadersprache portiert werden können. Da zum jetzigen Zeitpunkt die Sprache GLSL jedoch leichte Nachteile gegenüber HLSL und Cg hat (insbesondere ist das später betrachtete Parallax Occlusion Mapping nicht in GLSL implementierbar[5]), wurde die Sprache für weitere Betrachtungen nicht herangezogen. Da HLSL-Programme direkt nach Cg exportiert werden können, wird diese Sprache ausschließlich im weiteren Text verwendet.

Listing 1 zeigt als Beispiel die Implementierung der Beleuchtungsrechnung im Pixelshader in der Shadersprache HLSL.

2 OpenGL Shading Language, 2002 von Baldwin/OpenGL ARB vorgestellt [Bal02]
3 High-Level Shader Language, 2002 von Microsoft vorgestellt [Ros04]
4 C for graphics, 2002 von NVIDIA vorgestellt [Ros04]
5 Zum Befehl tex3Dlod gibt es kein Äquivalent in GLSL. Zwar existiert eine ATI-Extension, die eine vergleichbare Funktion bietet, allerdings wäre damit das Programm nicht mit NVIDIA-Hardware ausführbar.

```
float4 main( PS_INPUT In) : COLOR
{
   float3 vViewTS  = normalize(In.viewTS);
   float3 vLightTS = normalize(In.lightTS);

   float3 vNormalTS  = float3(0, 0, 1);

   float  fNDotL = dot(vNormalTS, vLightTS);
   float  fRDotV = max( 0.0f,
                    dot(reflect(-vLightTS, vNormalTS), vViewTS));

   float4 fvTotalAmbient   = cBaseColor * flAmbient;
   float4 fvTotalDiffuse   = cBaseColor * flDiffuse * fNDotL;
   float4 fvTotalSpecular  = flSpecular
                            * pow( fRDotV, fSpecularExponent);

   return (saturate(fvTotalAmbient
                  + fvTotalDiffuse
                  + fvTotalSpecular));
}
```

Listing 1: Beleuchtungsrechnung im HLSL Pixelshader

2.4.4 LOD-Systeme in Shadern

LOD-Systeme (LOD = Level of Detail) sind ein sehr wichtiges Werkzeug in der Computergrafik, um den Bildeindruck zu verbessern und die Erzeugung der Bilder zu beschleunigen. Sie helfen (allgemein) durch die der Vermeidung unnötiger Details Abtast- bzw. Aliasingprobleme zu verhindern. LOD-Systeme können auch durch Veränderung des Detailgrads von Geometriemodellen helfen, die Geschwindigkeit zu verbessern. Allerdings ist dieses Verfahren in einem Shader nicht realisierbar und daher hier uninteressant.

In Rost [Ros04] wird ein Pixelshader vorgestellt, der ein Schachbrettmuster erzeugt, das durch einen adaptiven analytischen Vorfilter sichtbare Artefakte verhindert. Alle drei Shadersprachen stellen entsprechende Methoden zur Verfügung, mit denen die Änderung eines Parameters im Bildraum ermitteln werden kann. Wird die Funktion mit den Texturkoordinaten als Parameter aufgerufen, so gibt das zurückgegebene Wertepaar an, wie stark sich diese in X- und

Y-Richtung an einem Punkt ändert (Abbildung 2.6, rot ist die Änderung in X-Richtung, grün in Y-Richtung).

Abbildung 2.6: Änderung der Texturkoordinate

Dieser Wert bestimmt die maximal darstellbare Frequenz, die ein prozeduraler Shader erzeugen darf, damit es nicht zu Aliasingeffekten kommt. Je stärker sich die Texturkoordinaten in einem Pixel ändern, desto größer ist der Bereich der Textur, mit dem der Pixel bedeckt werden muss. Ist die Frequenz zu hoch, so kann der Pixel mit einer Mittelwertfarbe gefärbt werden (Abbildung 2.7).

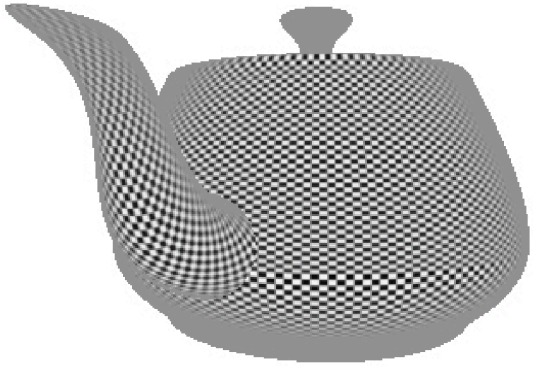

Abbildung 2.7: Antialiasing eines Schachbrettmuster

2.5 Fraktale Landschaftsgenerierung

In der Computergrafik haben sich vor allem zwei Verfahren zur Generierung von fraktalen Landschaften etabliert und durchgesetzt: MidPoint-Displacement und DiamondSquare.

2.5.1 MidPoint-Displacement

Das MidPoint–Displacement-Verfahren nach [FFC82] ist ein rekursiver Algorithmus zum Erzeugen realistisch aussehender fraktaler Höhenzüge. Er basiert auf der rekursiven Halbierung einer Strecke und der zufälligen Verschiebung des erzeugten Mittelpunktes:

```
Solange Verfeinerung noch nicht erreicht:
   Wiederhole für jede Linie:
      halbiere Linie
      ermittle Zufallswert rndY
      verschiebe Mittelpunkt in Höhe um rndY
      verkleinere Wertebereich des Zufallswertes
```

Listing 2: MidPoint-Displacement

Abbildung 2.2 zeigt die Vorgehensweise des Algorithmus (Listing 2) grafisch.

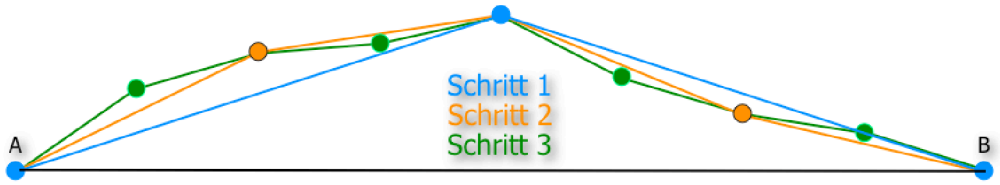

Abbildung 2.8: MidPoint Displacement

Je nach Einschränkung des Intervalls, aus dem die Zufallszahl „gezogen“ wird, kann die subjektive Glattheit des gesamten Zuges gesteuert werden. Üblich ist, den Wertebereich pro Iterationsschritt zu verkleinern.

2.5.2 DiamondSquare

Um nicht nur Höhenzüge, sondern ganze Landschaften generieren zu können, wurde der MidPoint–Displacement-Algorithmus auf drei Dimensionen erweitert [FFC82]. Seinen Namen weist auf das charakteristische Aussehen seiner zwei Aufteilungsschritte: eine Raute (engl. Diamond) und ein Quadrat (engl. Square) hin.

Zur Verdeutlichung des Algorithmus dienen Abbildung 2.9 und 2.10 [Mar06]:

```
Diamond:
    Berechne Mittelpunkt jeder Raute
    Verschiebe Mittelpunkt um Zufallswert

Square:
    Berechne Mittelpunkt jedes Quadrats
    Verschiebe Mittelpunkt um Zufallswert

DiamondSuqare:
    Initialisiere Startquadrat mit Höhenwerten
    Solange Verfeinerung nicht erreicht:
        Führe Schritt Diamond aus
        Führe Schritt Square aus
        verkleinere Wertebereich der Zufallswerte
```

Listing 3: DiamondSquare-Algorithmus

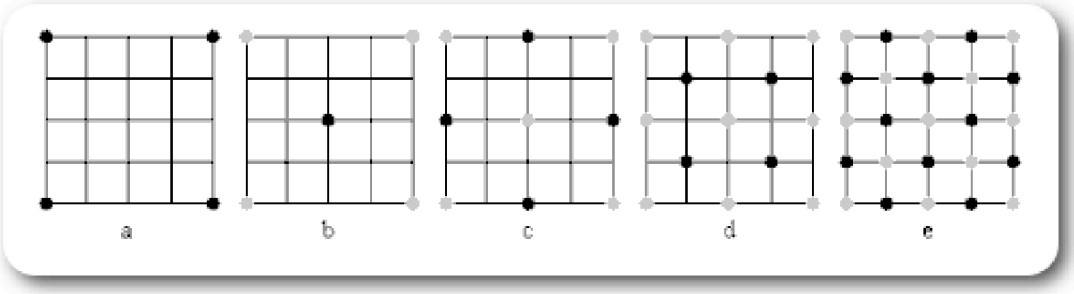

Abbildung 2.9: DiamondSquare Algorithmus, 5 Schritte

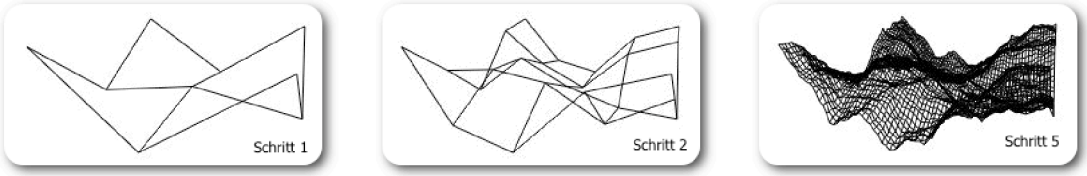

Abbildung 2.10: DiamondSquare Landschaft nach 1, 2 und 5 Schritten

Kapitel 3

Bumpmapping-Verfahren

„Illusion is the first of all pleasures."

Oscar Wilde

Das Bumpmapping-Verfahren wurde von Blinn 1978 veröffentlicht [Bli78]. Es hat zum Ziel, die Darstellung unebener Oberflächen in computergenerierten Bildern zu verbessern, ohne die Geometrie zu verändern. Das Verfahren arbeitet somit rein im Bildraum und erzeugt die Illusion einer detaillierter modellierten Geometrie.

Zunächst wird Blinns ursprünglich vorgeschlagenes Verfahren und im Anschluss daran werden verschiedene aktuelle Versionen vorgestellt. Den Abschluss dieses Kapitels bildet eine Beurteilung der vorgestellten Verfahren.

3.1 Bumpmapping

Das Bumpmapping-Verfahren wird beim realistischen Rendern von rauen Oberflächen eingesetzt. Es werden in einer Textur gespeicherte Bumps[6] auf eine Oberfläche projiziert. Es gibt zwei Verfahren, die eben dies leisten: Bumpmapping und Displacementmapping. Sie unterscheiden sich jedoch in einem wesentlichen Punkt. Bumpmapping findet im Bildraum und Displacementmapping im Objektraum statt.

3.1.1 Blinns Bumpmapping

Anstatt eine einzige (zum Pixel interpolierte) Normale pro Fläche in der Beleuchtungsrechnung einzusetzen, schlägt Blinn vor, an jedem Pixel eine Normalenablenkung aus einer Bumpfunktion zu lesen, diese in die Normale einzurechnen und anschließend mit der perturbierten Normalen die Beleuchtungsrechnung durchzuführen. Die Bumpfunktion wurde aufgrund der beschränkten Hardwareleistugen durch eine vorberechnete Bumpmap, eine Textur, ersetzt.

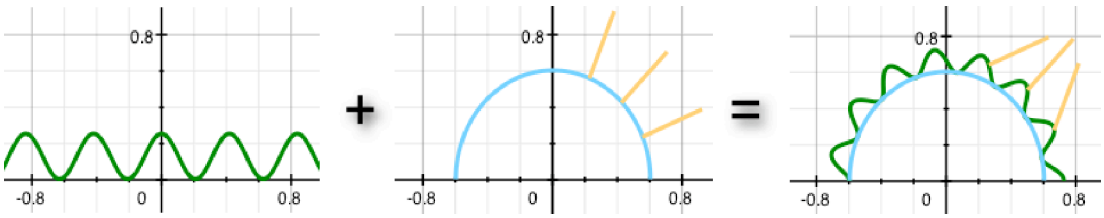

Abbildung 3.1: Bumpfunktion, Objektnormalen und perturbierte Objektnormalen

Das Verfahren liefert deutlich realistischere Ergebnisse als alle damals aktuellen Verfahren (Phongbeleuchtung und Texturemapping), kann jedoch nicht alle Probleme beseitigen. Da die Geometrie nicht verändert wird, ist die Silhouette unverändert gegenüber dem nicht mit Bumpmapping gerenderten Objekt.

6 Bump (engl.) = Unebenheit, Erhebung, Beule

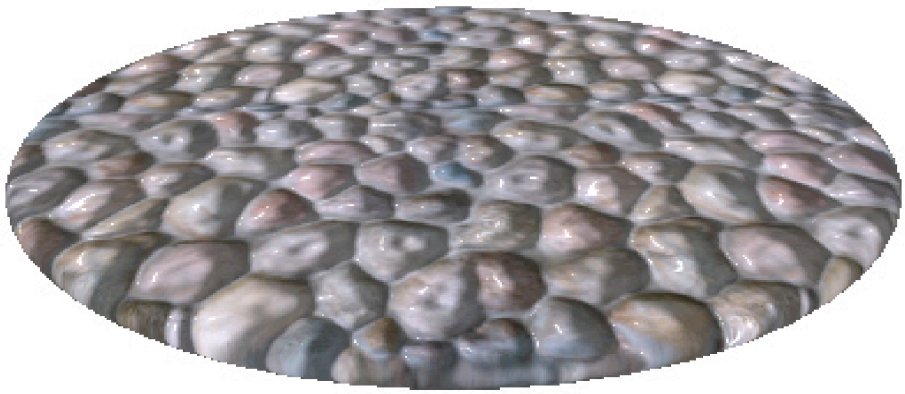

Abbildung 3.2: Bumpmapping mit Normalmapping

Ein weiteres Problem des Verfahrens ist die fehlende Selbstverdeckung und Selbstschattierung der veränderten Geometrie. Auch hier bietet das Bumpmapping nach Blinn keine Lösung.

3.1.2 Displacementmapping

Das Displacementmapping von Cook [Coo84] arbeitet nicht im Bildraum, sondern verschiebt die Vertices gemäß einer Displacement-Textur und arbeitet somit im Objektraum. Die Auflösung des Modells reicht unter Umständen nicht aus, so dass hier ggf. Vertices sogar erzeugt werden müssen. Das Verfahren führt zu sehr realistischen Ergebnissen, allerdings ist es mit aktueller Hardware (noch) nicht in Echtzeit implementierbar.

In eine ähnliche Richtung geht auch Breiners Hydrostatic Bump Mapping [Bre06], das die Vertices anhand des hydrostatischen Drucks gemäß mehrerer maximaler Deformationstexturen verschiebt und damit biologische Simulationen ermöglicht.

3.2 Pre-Shader-Bumpmapping

Aufgrund von technischen Limitierungen konnte das Verfahren lange Zeit nicht in Echtzeitsystemen eingesetzt werden, da die Beleuchtungsrechnung nur auf Vertexbasis stattfand. Allerdings wurden andere Verfahren entwickelt, die einen vergleichbaren Eindruck vermitteln sollen.

3.2.1 Embossedmapping

Das als Embossedmapping bekannte Verfahren, beschrieben von Schlag 1994 [Sch94], greift Blinns Idee der veränderten Lichtberechnung pro Pixel auf, setzt diese aber mit den Mitteln damaliger Hardware um.

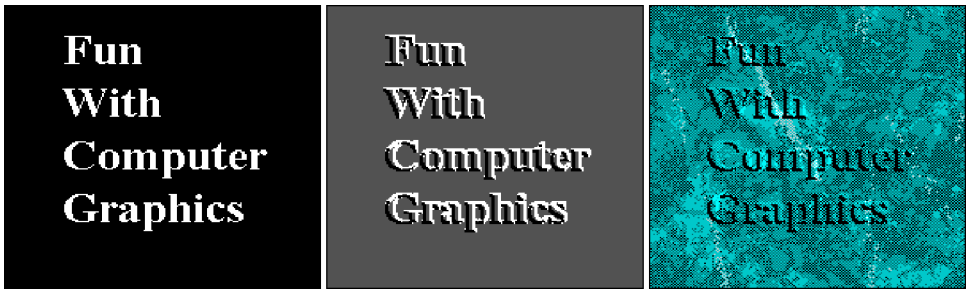

Abbildung 3.3: Heightmap, grauer Hintergrund, gemusterter Hintergrund

Anstatt die Lichtberechnung wirklich zu beeinflussen, wird nur eine Heightmap[7] vom Licht weg verschoben und vom originalen Bild abgezogen (Formel 3.1). Dadurch entsteht der Eindruck einer unterschiedlich beleuchteten Oberfläche. Mathematisch wird dabei folgende Operation durchgeführt:

$$farbe = (height_{original} - height_{verschoben}) * farbe_{diffuse} * farbe_{ambient}$$

Formel 3.1: Shadingformel des Embossedmappings

7 Textur, die das Höhenprofil eines Materials speichert. Um eine bessere Auflösung der Höhenwerte zu ermöglichen, können die Farbkanäle miteinander kombiniert werden.

Die ersten Implementierungen mussten mit Multitexturing arbeiten: Die Textur wird zweimal gerendert - einmal original und einmal verschoben. Vor dem Renderpass der verschobenen Textur wird als Blendmode „subtractive“ (abziehend) ausgewählt. Sobald Register-Combiner zur Verfügung standen, war es möglich, Embossing in einem Durchgang zu erzeugen. Register-Combiner ermöglichen das Ablegen von max. 4 Werten pro Pixel, die mit recht beschränkten mathematischen Methoden verbunden werden können. Das Verfahren hilft zwar einen groben Eindruck zu vermitteln, ist aber nicht in der Lage, wirklich realistische Bilder zu erzeugen (Abbildung 3.3[8]).

3.2.2 DOTPRODUCT3-Bumpmapping

Mit dem Aufkommen von Register-Combinern konnte die erste echte Verwirklichung des Blinnschen Bumpmappings implementiert werden, die unter dem Namen „DOTPRODUCT3 Bump Mapping“ bekannt wurde [Kil00].

Da das Vektorprodukt ist eines der Register-Combiner Operationen, kann ein pixelweises Bumpmapping mit berechnet werden:

- Eine Normalmap wird im ersten Register A abgelegt.
- Der Vektor zum Licht wird im zweiten Register B gespeichert.
- Im dritten Register C wird das Vektorprodukt der ersten beiden Register abgespeichert.

Pro Pixel wird somit das Vektorprodukt von Normale und Vektor zum Licht im Register gespeichert. Da dieser Wert die Grundlage für die diffuse Beleuchtungsrechnung ist, kann Bumpmapping erfolgreich berechnet werden. Die Berechnung des glänzenden Lichtanteils erfolgt auf gleichem Weg mit Hilfe einer weiteren Cubemap-Textur.

8 Quelle: http://web.cs.wpi.edu/~matt/courses/cs563/talks/bump/bumpmap.html , Zugriff: 12.11.2006, 12:35MEZ

3.2.3 Environment-Mapped Bumpmapping

Das Environment-Mapped Bumpmapping (kurz: EMBM) basiert auf dem Environment-Mapping von Blinn [BN76]. Die Erweiterung nähert sich dem Bumpmapping-Verfahren von einer anderen Seite: Anstatt die perturbierte Normale in der Lichtberechnung zu nutzen, wird sie für die Ermittlung des Cubemapping-Vektors benutzt (Bestimmung des reflektierten Sichtstrahls). In der Cubemap wird die Umgebung gespeichert, wodurch der Eindruck entsteht, dass das Material stark reflektierend sei. Wird das Verfahren pro Pixel durchgeführt und die Normale aus einer Normalmap gelesen, entsteht aufgrund der gut sichtbaren Reflektanzunebenheiten der Eindruck einer unebenen, reflektierenden Oberfläche.

Abbildung 3.4: EMBM (NVIDIA Demo)

3.3 Normalmapping

Das Normalmapping ist ein als Shader implementiertes Bumpmapping-Verfahren. Anstatt eine Perturbation in die Normale einzurechnen, wird direkt die veränderte Normale in der Normalmap gespeichert. Diese kann entweder in einem Vorverarbeitungsschritt erzeugt oder aus einer Heightmap zur Laufzeit errechnet werden.

Normalmapping benötigt sowohl ein spezielles Vertex- als auch ein Pixelshaderprogramm. Im Vertexshader werden Kamera- und Lichtkoordinaten in den lokalen Vektorraum (auch „Tangentspace“ genannt) des Vertex transformiert und dem Pixelshader als Parameter übergeben. Innerhalb des Vektorraums befindet sich der Vertex im Koordinatenursprung und die Normale zeigt in Richtung der positiven Z-Achse. Dadurch kann die Beleuchtungsrechnung homogen für alle gerasterten Pixel erfolgen. Im Pixelshader wird mit Hilfe der Texturkoordinaten die perturbierte Normale aus der Normalmap gelesen, in den richtigen Wertebereich verschoben und anschließend eine „lokale“ Beleuchtungsrechung durchgeführt. Die Normalenskalierung ist notwendig, da die Farben einer Textur dem Wertebereich [0;1] entstammen und eine Normale Werte aus dem Bereich [-1;1] besitzt. Da eine Textur mindestens drei Farbkanäle und eine Normale drei Komponenten hat, können die Werte direkt als solche interpretiert werden.

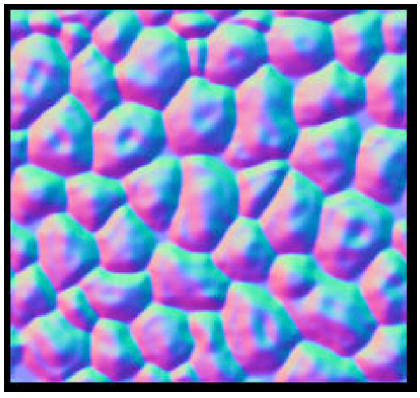

Abbildung 3.5: Normalmap, ATI

Des weiteren ist darauf zu achten, dass die aus dem Vertexshader übergebenen Licht- und Kameravektoren normalisiert werden. Dies ist nötig, da zum einen die Werte im Rasterisierer interpoliert werden und somit keine Einheitslänge haben und zum anderen nur die Richtung, nicht aber die Länge für die anschließende Berechnung wichtig ist.

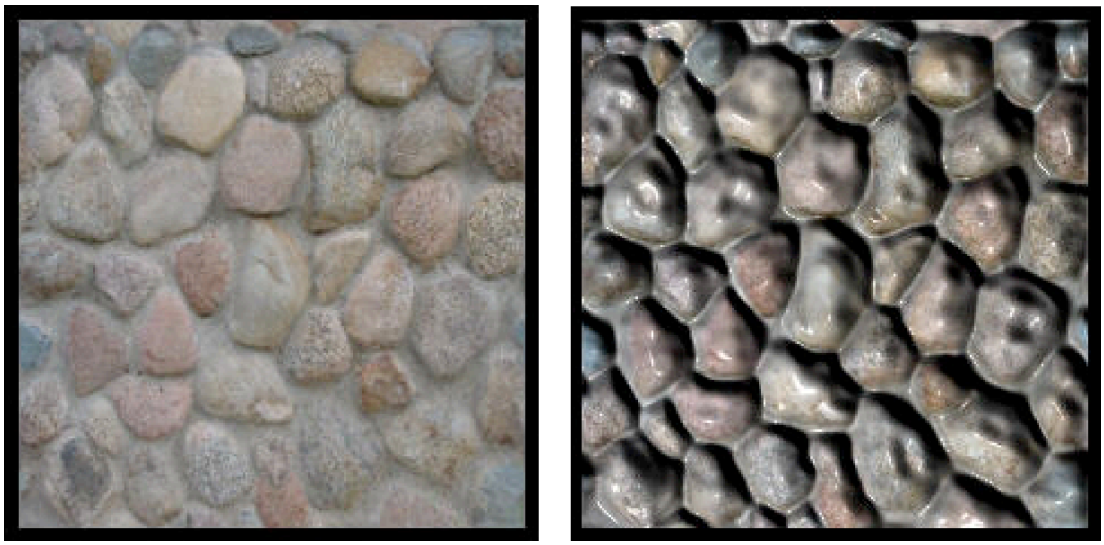

Abbildung 3.6: Originale Textur und Textur mit Normalmapping

Auch wenn sich mit Normalmapping-Verfahren deutlich bessere Ergebnisse erzeugen lassen, bleiben die Probleme des Blinnschen Bumpmappings (Selbstschattierung, Selbstverdeckung und fehlender Tiefeneffekt) bleiben erhalten.

3.4 Heightmap-basierte Verfahren

Die aktuellste Gruppe von Bumpmapping Verfahren hat Blinns Idee um eine entscheidende Komponente erweitert: Sie verfolgen den Sichtstrahl „durch“ das Material und schneiden ihn mit dem Höhenprofil, welches in einer Heightmap gespeichert ist. Der ermittelte Schnittpunkt ist der korrekte Punkt der Oberfläche, welcher gesehen wird.

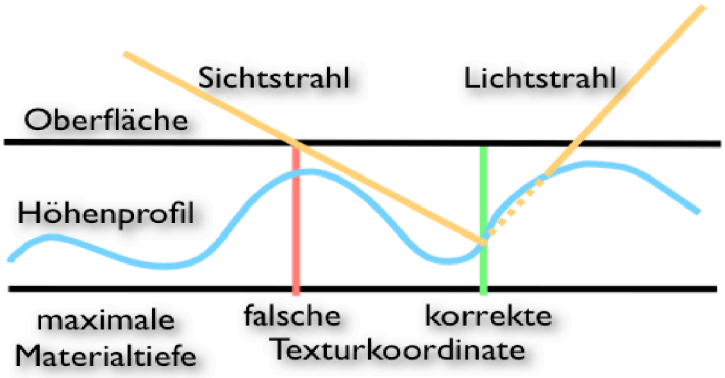

Abbildung 3.7: Prinzip des Heightmap-Tracings

Anstatt die Oberfläche als flaches, zweidimensionales Objekt aufzufassen, wird sie von der Heightmap um eine dritte Komponente erweitert. Die (zweidimensionalen) Texturkoordinaten bilden nach wie vor X- und Y-Achse, der in

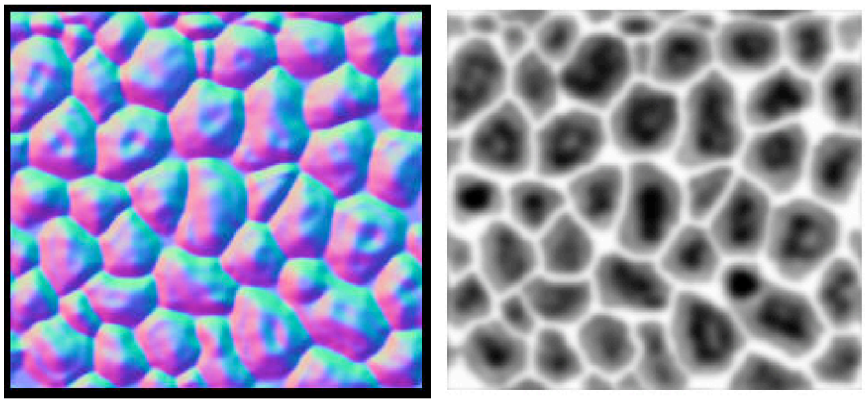

Abbildung 3.8: Normalmap und Heightmap

der Heightmap gespeicherte Wert die positive Z-Achse des Höhenprofils. Der Höhenwert beginnt nach Konvention an der Oberfläche mit Z=0.

Diese Vorgehensweise, welche mit dem Begriff „Heightmap-Tracing“ beschrieben werden kann, behebt die zuvor bemängelten Punkte von Blinns Algorithmus. Selbstschattierung, Selbstverdeckung und Tiefeneindruck entstehen nun völlig automatisch: Anstatt der falschen (herkömmlich ermittelten) Texturkoordinate wird die per Schnitt von Sichtstrahl und Heightmap ermittelte korrekte Koordinate für die weitere Lichtberechnung benutzt. Anhand des Schnittpunktes und der Lichtposition, kann Selbstschattierung berechnet werden [CD03]. Da das Verfahren an jedem Pixel einen neuen Betrachtungswinkel ermittelt, wird auch über große Flächen hinweg der korrekte Schnittpunkt berechnet.

Eine weitere Verbesserung des Verfahrens betrifft die Objektsilhouette: Trifft ein Sichtstrahl nicht auf den Höhenzug, so trifft der Sichtstrahl nicht auf die Geometrie. Somit gehört das Pixel nicht zum Objekt und wird vom Renderprozess ausgeschlossen (in GLSL direkt,[9] in HLSL über den Alphakanal). Dieses von Policarpo in 2005 vorgeschlagene Verfahren [PO05] liefert beeindruckende Ergebnisse (Abbildung 3.9, linker Würfel), hat aber auch mit Problemen zu kämpfen. Die Silhouette wird nicht erweitert, sondern nur „beschnitten“. Betrachtet man eine einfache Fläche von der Seite, erscheint sie immer noch eben (rechter Würfel, obere Seite).

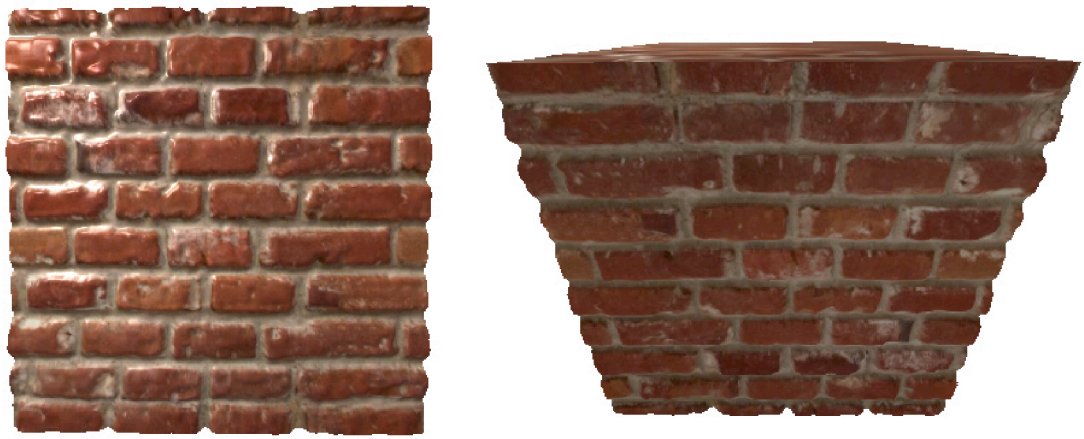

Abbildung 3.9: Silhouette und ebene Fläche

Die Schwierigkeit des Heightmap-Tracings ist das Berechnen des korrekten Schnittpunktes. Aktuell gibt es zwei parallel entstandene Verfahren, die das Problem auf verschiedene Arten lösen: Relief-Mapping von Policarpo et al. [OBM00], [POC05], [PO05] und [PO06] und Parallax-Occlusion-Mapping von Tatarchuk [BT04], [MM05] und [Tar06].

3.4.1 Relief-Mapping

„Relief Mapping“ erweitert Oliveira's „Relief Texture Mapping“ [OBM00], so dass es auf beliebige polygonale Oberflächen angewandt werden kann.

Die Schnittpunktberechnung erfolgt dabei in zwei Stufen: Zunächst wird mit binärer Suche der grobe Auftreffbereich bestimmt, anschließend mit linearer Suche über den ermittelten Bereich der genaue Punkt ausgetestet.

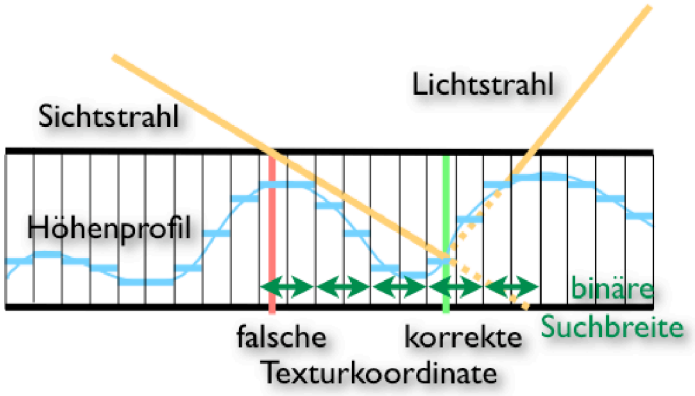

Abbildung 3.10: Schema Relief-Mapping

Durch die Bestimmung der Schrittweite in beiden Suchvorgängen kann die maximale Laufzeit sehr gut eingeschränkt werden. In Abbildung 3.10 ist die Breite der binären Suche 2 Texel (bei 10 Texel Eindringungsbreite also 5 Tests), die der linearen Suche 1 Texel. Je nach Erfordernis ist so eine feine Abstufung zwischen Geschwindigkeit und Genauigkeit möglich. Die binäre Suche verkürzt die Zeit, die

eine komplette lineare Suche über das gesamte Höhenprofil ansonsten erfordern würde.

Die aktuellste Version des Verfahrens beschäftigt sich mit mehrseitig aufgebrachten Heightmaps: eine Heightmap für die Vorderseite und eine für die Rückseite. Die so beschriebenen Objekte können Lücken haben und von beiden Seiten gerendert werden, wie etwa das folgende Holzmaschengitter:

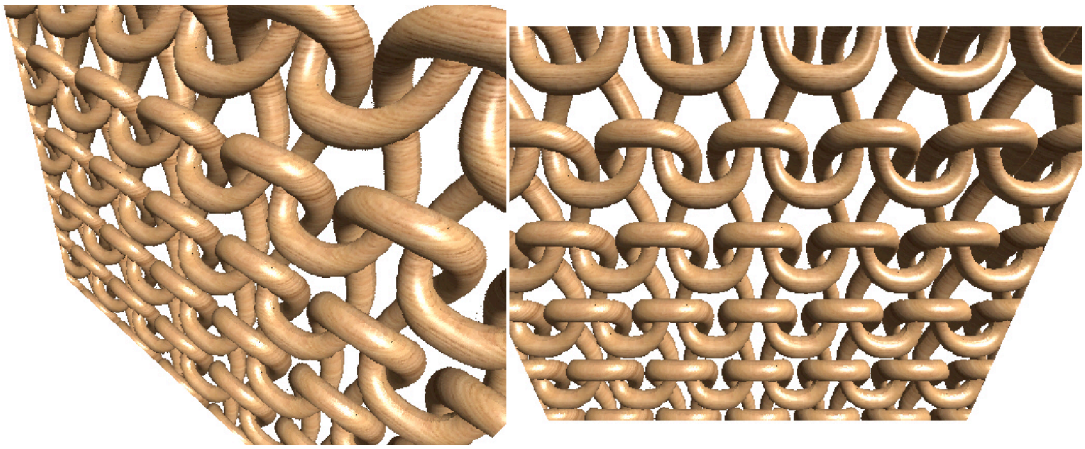

Abbildung 3.11: Relief-Mapping of Non-Height-Field Surface Details

3.4.2 Parallax-Occlusion-Mapping

Tatarchuks „Parallax Occlusion Mapping“ (kurz POM) greift die Idee von Reliefmapping auf, benutzt aber eine andere Methode der Schnittpunktbestimmung. Das Material wird komplett mit linearer Suche durchlaufen. Ist der Texel des Schnittpunktes bestimmt, wird der eigentliche Schnittpunkt per Interpolation aus den benachbarten Punkten bestimmt. Dadurch wird für den Schnittpunkt nicht alleine der Wertebereich der Heightmapdaten (meistens 8 Bit) genutzt, sondern der gesamte Wertebereich der Testvariablen (üblich sind hier 32 Bit Floats). Diese erhöhte Berechnungsgenauigkeit ermöglicht ein korrektes Darstellen weit entfernter Texel unter flachen Sichtwinkeln, die nicht wie beim Reliefmapping abflachen (Horizontflachheit).

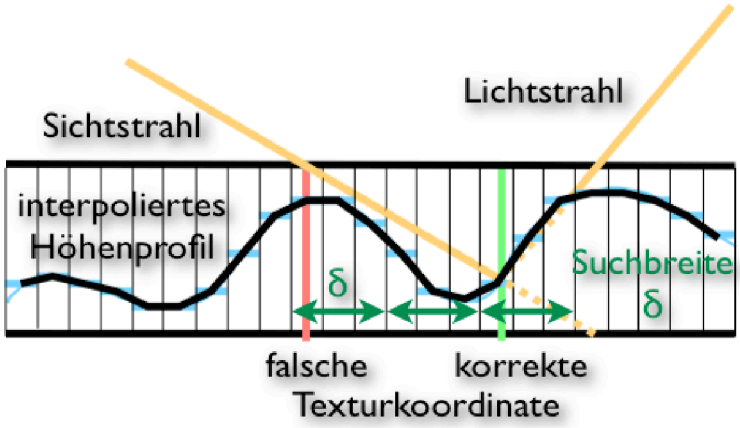

Abbildung 3.12: Schema POM

Zusätzlich wird noch auf das entsprechende Mipmaplevel[10] der Textur innerhalb der linearen Suche zugegriffen. Die Abtastbreite der linearen Suche wird dynamisch im Shader je nach Blickwinkel (innerhalb vom Benutzer definierbarer Grenzen) angepasst (In Abbildung 3.12 ist die Suchbreite $\delta = 3$). Je flacher der Blickwinkel das Material trifft, desto mehr Texel schneidet der Sichtstrahl, bis er den Boden berührt.

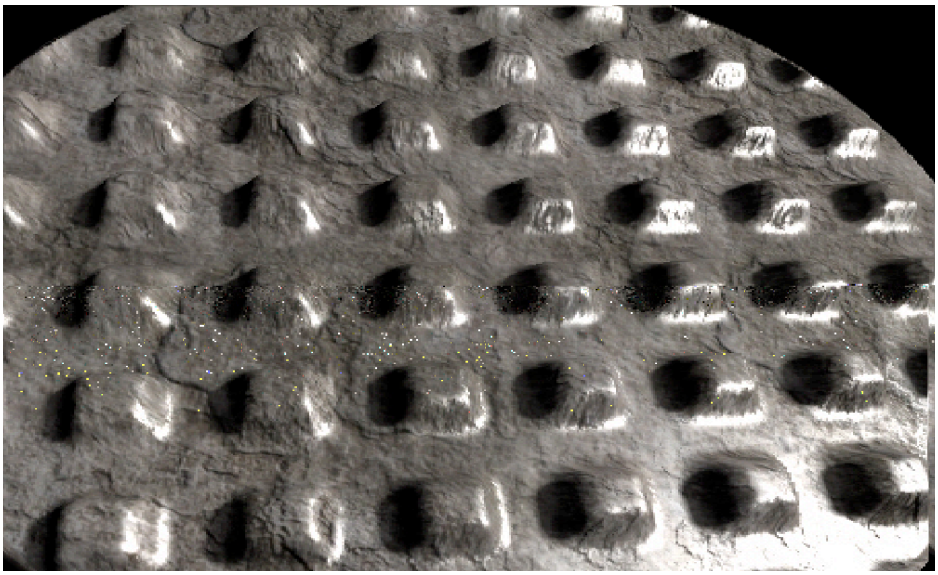

Abbildung 3.13: Oberfläche, gerendert mit POM

10 Mipmapping bedeutet, dass zu jeder Textur mehrere Texturen unterschiedlicher Größe existieren. Dadurch müssen die Texturen nicht übermäßig verkleinert werden und Aliasingartefakte werden vermindert.

3.4.3 Vergleich

Beide Heightmap-Tracing-Ansätze bringen erstaunliche Ergebnisse im Vergleich zum reinen Bumpmapping, bringen gleichzeitig aber auch Probleme mit sich.

	Relief-Mapping	POM
Schnittmethode	binär + linear	linear
Mipmaplevel/LOD	✗	✓
Abtastbreite	Statisch	Dynamisch
Laufzeit	„schnell“	„langsam“
keine Horizontflachheit	✗	✓
Zweiseitige Strukturen	✓	✗

Das Parallax-Occlusion-Mapping scheint das fortgeschrittenere Verfahren zu sein. Zwar beherrscht Reliefmapping die Darstellung zweiseitiger Strukturen, die Technik ist jedoch auf das POM übertragbar. Der Vorteil von POM ist, dass es ein dynamisches Verfahren ist, das durch den Einsatz von Mipmaps ein einfaches LOD-System bietet. Durch die Interpolation zwischen den Höhenwerten ist es zudem weniger anfällig für Aliasingartefakte.

Aufgrund des Vergleiches wurde für die weitere Arbeit das POM Verfahren als Grundlage für die Entwicklungen von prozeduralen Shadern und Erweiterungen benutzt.

3.5 Fazit

Aktuelle Bumpmapping-Verfahren und ihre Verbesserungen erreichen einen hohen Realitätsgrad. Sie liefern überzeugende Ergebnisse und gehen sogar weit über das von Blinn vorgeschlagene Verfahren hinaus. Mit ihnen sind vor allem folgende Effekte besonders eindrucksvoll:

- Parallaxeffekte,
- Selbstschattierung,
- Selbstverdeckung und
- Silhouettenbildung.

Allerdings sind die einzelnen Verfahren auch nicht problemfrei.

3.5.1 Scheibchenbildung

Auffälligstes Problem ist die Schnittpunktberechnung von Sichtstrahl und Höhenzug unter flachen Betrachtungswinkeln. Bei entsprechend eingestellter Materialhöhe kommt es bei allen Verfahren zu starker Artefaktbildung (Abbildung 3.14).

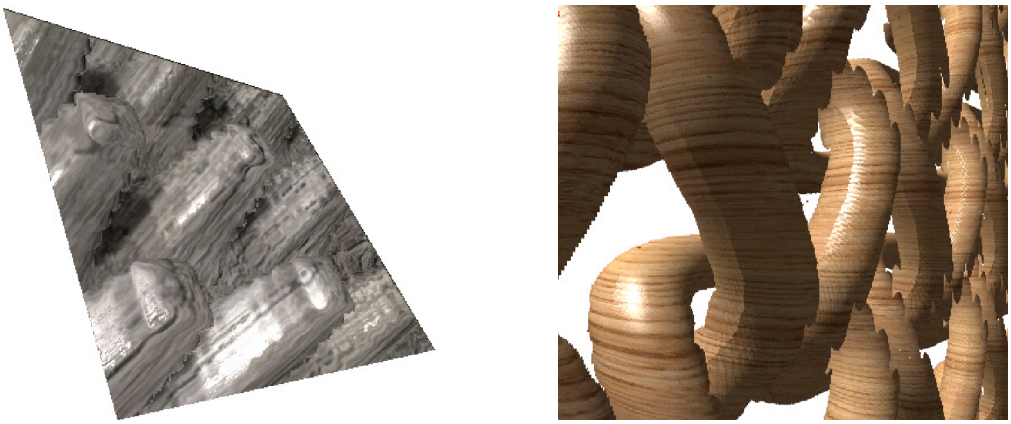

Abbildung 3.14: Scheibchenbildung

Die falschen Schnittpunkte entstehen aus der Abtastung der diskreten, pixelbasierten Heightmap. Sie sind verfahrensbedingt und werden bei entsprechender Auflösung des Materials selbst bei einer Erhöhung der Heightmap nicht verschwinden. Je mehr abzutastende Texel auf einen Pixel fallen, desto wahrscheinlicher sind Treppenkanten und Abtastfehler.

Eine Folge der Fehlabtastungen sind „verlorene Pixel“. Besitzt das Material dünne Spitzen, so werden bei entsprechenden Betrachtungswinkel die zusammenhaltenden Pixel nicht abgetastet und die Enden schweben scheinbar frei in der Luft.

3.5.2 Klötzchenbildung

Ein weiteres Problem, das von keinem der genannten Verfahren gelöst wird, ist die Klötzchenbildung. Sie wird in verschiedenen Formen deutlich. Sei es der stark pixelnde (selbst erzeugte) Schatten, die Silhouette mit sichtbaren Pixelblöcken, die scheinbar in der Luft schweben, oder deutlich erkennbare Treppenkanten bei vergrößerten Höhenzügen. Der Grund liegt immer in einer unzureichenden Texturauflösung – ein Texel wird auf mehrere Pixel abgebildet. Die Klötzchen erinnern ein wenig an die von Voxelspace-Grafik gewohnten Bilder. Beide Verfahren sind sich sehr ähnlich und Voxelspace ist für dieses Manko bekannt.

Abbildung 3.15: Klötzchenbildung

3.5.3 Texturauflösung

Grundlage aller Verfahren ist eine Textur, die den Höhenzug des Materials speichert. Eine Textur ist immer eine diskrete Speicherstruktur - die Anzahl der Speicherzellen und die darin gespeicherten Werte sind genau festgelegt und limitiert. Bisher hat sich mit zunehmenden Grafikkartenspeicher auch die maximale Texturauflösung verbessert. Dadurch werden einige Probleme aber nur verschoben (Selbstschattierung, Selbstverdeckung und Materialdetail), andere aber werden bleiben: Alle Verfahren, die auf diskreten Speicherstrukturen aufbauen, werden immer mit Abtastproblemen (Abtasttheorems [Wat02]) zu kämpfen haben.

Abbildung 3.16: Begrenzte Texturauflösung

Kapitel 4

Eigene Verfahren

„The first problem is simply finding something that hasn't been done."

Jim Blinn

In diesem Kapitel werden eigene Verfahren und Weiterentwicklungen des Bumpmapping-Verfahrens vorgestellt.

Den Anfang bildet eine Anforderungsanalayse sowie eine Beschreibung der Mittel, die benutzt wurden, um die Aufgabe zu lösen. Der erste Schritt ist die Erstellung eines prozeduralen Shaders für die Oberfläche einer Backsteinmauer. Daran anschließend werden die Möglichkeiten der Erzeugung von fraktalen Landschaften in Shadern besprochen und deren Einsetzbarkeit in prozeduralen Shaderfunktionen analysiert. Den Abschluss bildet ein neu entwickelter Ansatz, um Materialien und deren Oberflächen prozedural zu generieren.

4.1 Arbeitsgrundlagen

In diesem Abschnitt wird zunächst die Aufgabenstellung näher betrachtet und eine Anforderungsanalyse aufgestellt. Danach wird die Implementierungs- und Bewertungsgrundlage beschrieben.

4.1.1 Anforderungsanalyse

Die Aufgabenstellung dieser Diplomarbeit ist die Betrachtung der Weiterentwicklungsmöglichkeiten des Bumpmapping-Verfahrens. Unter einer Weiterentwicklung kann verstanden werden:

1. Verbesserung der grafischen Qualität
2. Vereinfachung in der Handhabung

   Eine Vereinfachung des Verfahrens kann etwa ein prozeduraler Shader sein, der im Einsatz keine (oder weniger) Texturen benutzt. Das auf diese Weise erzeugte Material ist flexibler (durch Parameter des Shaders regelbar) und kann einfacher eingesetzt werden, da die Texturensets gar nicht erst erzeugt und ggf. nachbearbeitet werden müssen.
3. Neue Methoden

   Fehlerhafte Darstellungen, die durch das Prinzip der ursprünglichen Herangehensweise entstehen, können so vermieden entstehen.

Alle erarbeiteten Verfahren müssen folglich mindestens einen der genannten Punkte abdecken.

4.1.2 Konzept- und Implementierungsgrundlage

Alle Programme werden in Form von Shadern in der Sprache HLSL implementiert. Mit HLSL können alle momentan in Shaderhardware

implementierten Funktionen genutzt werden. Zudem können Shader in HLSL nach NVIDIA Cg konvertiert werden (vgl. Kapitel 2.4.3).

Da Shadersprachen im Vergleich zu anderen Hochsprachen noch relativ limitiert sind, hat dies auch Folgen für die Konzept- und Implementierungsabschnitte: Das Konzept muss in Shadern umsetzbar sein und die begrenzten Programmiermöglichkeiten berücksichtigen.

4.1.3 Bewertungsgrundlage

Alle Shader wurden auf folgendem System getestet: AMD Athlon 2800+, 1 GB DDR2 RAM, NVIDIA 6600GT mit 256 MB RAM. Installiert waren Windows XP SP2 und ForceWare-Treiber in der Version 93.71. Als Entwicklungsumgebung wurde RenderMonkey 1.62[11] zum Ausführen und Testen der erstellten Verfahren benutzt. Die Shader lassen sich sowohl in andere Anwendungen exportieren.

Alle Frameraten sind mit folgenden Einstellungen gemessen worden:

- Der Shader wurde in RenderMonkey geladen und angezeigt.
- Die Renderfläche hatte eine Auflösung von 1280x1024 Pixeln (Vollbild).
- Als einziges angezeigtes Objekt diente das ATI Modell „Disc2.3ds“ aus den Beispielen von RenderMonkey.
- Die Kamera befand sich an Position (200, 0, 0), blickte in Richtung Ursprung, hatte einen Upvektor von (0, 1, 0) und ein Field of View von 45°.

11 Erhältlich unter: http://www.amd.com oder http://www.3dlabs.com

4.2 Ein prozeduraler Backsteinshader

Der zuerst untersuchte Aspekt der Weiterentwicklung von Bumpmapping-Verfahren, betrifft die prozedurale Generierung von Oberflächen. Dies bedeutet, dass zur Laufzeit pro Pixel ein Algorithmus ausgeführt wird, der das Aussehen des Oberfläche an diesem Punkt bestimmt. Dazu gehören die Ermittlung seiner Farbe, des Reflektanzverhaltens und alle anderen für die korrekte Berechnung der Schattierung notwendigen Details.

Die Methode hat sowohl Vor- als auch Nachteile: Auf der einen Seite bedeutet die Auswertung eines Algorithmus die (weitestgehende) Freiheit von Texturen. Dadurch verschwinden viele Probleme, die mit dem Abtasten verbunden sind: Aliasing, „Aufpixelung“ der Oberfläche und dergleichen. Insbesondere könnte die prozedurale Erzeugung von Oberflächen helfen, die Probleme der aktuellen Bumpmapping-Verfahren zu beheben. Auf der anderen Seite ist die Entwicklung eines prozeduralen Shaders nicht immer einfach: Es muss eine allgemeine Berechnungsvorschrift gefunden werden, die die Oberfläche treffend beschreibt. Im Falle eines Schachbrettmusters mag dies noch trivial erscheinen [Ros04], aber nicht jede Oberfläche hat eine derart nahe liegende Beschreibungsmöglichkeit, wie etwa gemusterter Stoff. Im Folgenden wird gezeigt, dass bereits ein einfaches Muster, eine mit Backsteinen belegte Oberfläche, verbunden mit dem Heightmap-Tracing Probleme bereiten kann.

4.2.1 Konzept

Das Erscheinungsbild einer Backsteinmauer, aus dem ein kachelbares[12] Teilstück nachmodelliert werden soll, hängt von vielen Parametern ab:

- Breite und Tiefe[13] der Backsteine,

12 Kachelbar heißt, dass mehrere Stücke nahtlos aneinander gelegt werden können.
13 Um Konfusionen mit dem später erzeugten Höhenwert zu vermeiden, bezeichnet die X-Achse die Breite, die Y-Achse die Tiefe und die Z-Achse die Höhe des Materials.

- Dicke des Mörtels zwischen den Steinen,
- Farbe der Backsteine und
- Höhenunterschied zwischen Backsteinen und Mörtel.

Eine zweidimensionale Texturkoordinate pro Pixel legt die auszuwertende Position des Materials fest. Per Konvention hat die vom Shader erzeugte Backsteinkachel in den Texturkoordinaten eine Breite und Tiefe von eins. Als geeignete Parameter haben sich folgende herausgestellt:

- Anzahl der Steine in der Breite der Textur
- Anzahl der Steine in der Höhe der Textur
- Verhältnis von der Größe der Backsteine zu den Zwischenräumen

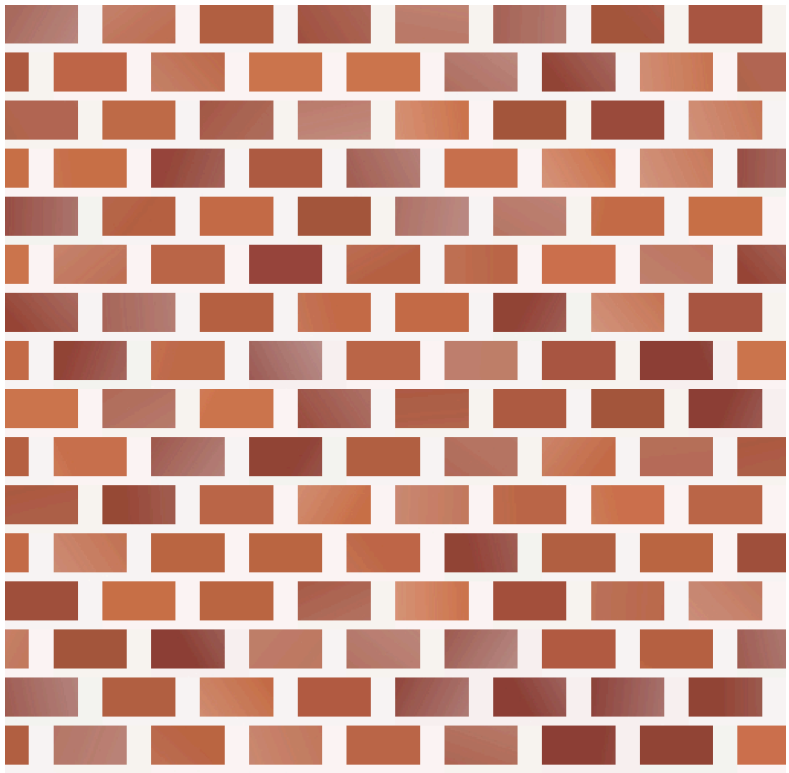

Abbildung 4.1: Backsteinshader-Struktur

Um eine realistischere Darstellung der Backsteinwand zu ermöglichen, werden zwei weitere Parameter benutzt:

- maximale Farbvarianz eines Backsteins von der Grundfarbe und
- maximaler Versatz und Rotation eines Backsteins.

Die Parameter verändern nur indirekt das Backsteinmuster: Sie bestimmen die maximale Einwirkung eines zufälligen Wertes. Dieser sollte aber uniform pro Backstein erzeugt werden, so dass er etwa für jeden Stein eine einheitliche Farbe berechnet. Deswegen wurden die Backsteine durchnummeriert und aus der Backsteinnummer ein Zufallswert erzeugt. Das Ergebnis der Farbvarianz wird in Abbildung 4.1 gezeigt.

4.2.2 Implementierung

Die Generierung der Backsteine ist vergleichsweise einfach, da sie ein regelmäßiges Muster bilden:

```
float4 ps_main(PS_INPUT Input) : COLOR0

{
   float4 color = {0, 0, 0, 1};

   //size of (brick + mortar) in texCoors
   float2 totalSize = 1.0 / brickCount;


   float2 paddingSize = totalSize * brickPadding / 100.0;
   float2 brickSize = totalSize - paddingSize;

   //every 2nd line is a wrap-around line
   if (mod_(texCoord.y, 2.0*totalSize.y) >= totalSize.y){
      texCoord.x += (totalSize.x * 0.5);
   }

   float2 texCoordMod = mod_(texCoord, totalSize);

   // it's a brick, if the mod'd texCoord < brickSize
   bool isBrick = lessAll(texCoordMod, brickSize);
```

Listing 4: Generierung des Backsteinmusters im Pixelshader

In aktuellen Shaderprogrammen gibt es allerdings keine Möglichkeit, Zufallswerte zu erzeugen. Es existiert zwar eine Noisefunktion, die auf Perlins Noise-Verfahren basiert ([Per85], [Per02]), jedoch ist diese zur Zeit nur auf 3Dlabs Hardware implementiert. Auf anderer Hardware liefert die Funktion konstant 0.5 zurück. Von der Alternative, die Perlin-Noise Werte direkt im Shader zu erzeugen [Har01], musste abgesehen werden, da deren Implementierung nicht nur ein aufwändiges Multipassverfahren für den Shader wäre, sondern es eine wesentlich effizientere Lösung gibt, die Vergleichbares leistet: Texturen mit generierten Perlin-Noise und (unabhängigen) Zufallswerten. Bei beiden Texturen handelt es sich um dreidimensionale Texturen, welche etwa in Beispielen von RenderMonkey benutzt werden.

Die Rotation soll jeweils über einen Stein hinweg erfolgen. Für jeden Pixel des Materials muss – sofern das Pixel einen Backstein darstellt – ermittelt werden, welche Nummer der Stein hat. Diese Information wird im blauen Farbkanal abgelegt (in Abbildung 4.2 zur Veranschaulichung in Graustufen gezeigt). Die Zufallsvariable für Rotation und Farbvarianz basiert auf der Steinnummer, so dass sie immer den gleichen Wert für den gleichen Stein unabhängig von der Texturkoordinate liefern kann.

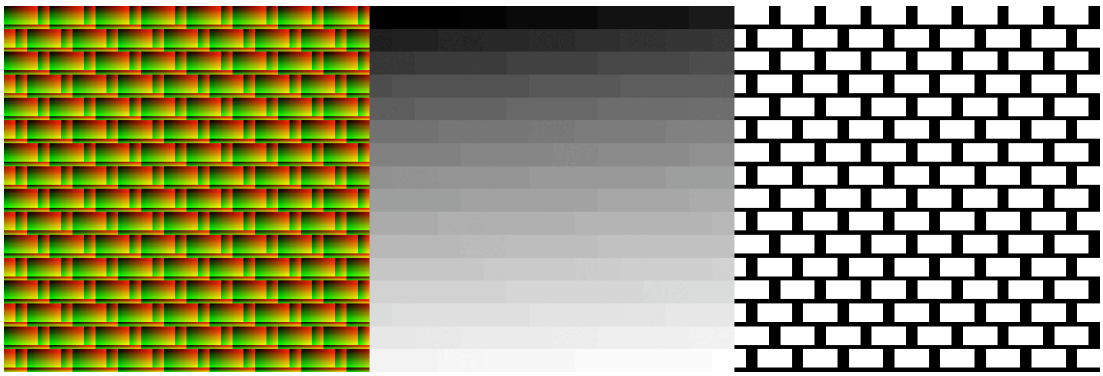

Abbildung 4.2: Zusammenhangsinformationen des Backsteinshaders

Die Rotation eines Backsteins erfordert noch weitere Zusammenhangsinformationen: Um die Neigung über die Fläche eines einzelnen Steins korrekt für

jeden Pixel zu berechnen, muss das Pixel auch seine Position innerhalb des Steins kennen. Zu diesem Zweck wird für jeden Stein eine Texturkoordinate im roten und grünen Kanal erzeugt, die für jeden Stein die Werte von 0 bis 1 annimmt. Im Alphakanal kann die Höhe jedes Steins gespeichert werden. Abbildung 4.2 verdeutlicht die bis jetzt erstellten Zusammenhangsinformationen, Listing 5 zeigt den restlichen Quellcode.

```
color.a = isBrick;

if (!isBrick){ //assign texCoords to mortar pieces
    if (texCoordMod.y > brickSize.y){   //below
        color.r = (texCoordMod/totalSize).x;
        color.g = ((texCoordMod - brickSize) / paddingSize).y;
    }else{                              //right
        color.r = ((texCoordMod - brickSize) / paddingSize).x;
        color.g = (texCoordMod/brickSize).y;
    }
}else{
    color.rg = (texCoordMod/brickSize);
}

float numBricks = ceil(brickCount.x) * ceil(brickCount.y);

color.b = (texCoordDiv.y * float(brickCount.x) +
                                    texCoordDiv.x) / numBricks;
```

Listing 5: Generierung der restlichen Variablen

Das Verfahren wird jetzt mit dem Parallax-Occlusion-Mapping zusammengebracht. Anstatt des Texturzugriffs auf die Heightmap wird die geschriebene Funktion aufgerufen. Dies funktioniert leider aufgrund der Instruktionsgrenze des Shaders nur mit dem reinen Backsteinmuster, die Variationen sprengen den möglichen Rahmen. Daher wurde zusätzlich noch ein Render-To-Texture-Pass hinzugefügt, der die Zusammenhangsinformationen in eine Textur rendert.

Ist die korrekte Texturkoordinate in Bezug auf den Sichtstrahl ermittelt, wird die Funktion ein weiteres Mal aufgerufen, um die Shadingparameter zu ermitteln. Die Normale wird gemäß des in Kapitel 4.3 diskutierten Verfahrens berechnet.

4.2.3 Bewertung

Verbunden mit dem Parallax-Occlusion-Mapping entstehen Ergebnisse, die in Farbvarianz, Rotation und Höhenversatz reale Backsteinmauern ähneln:

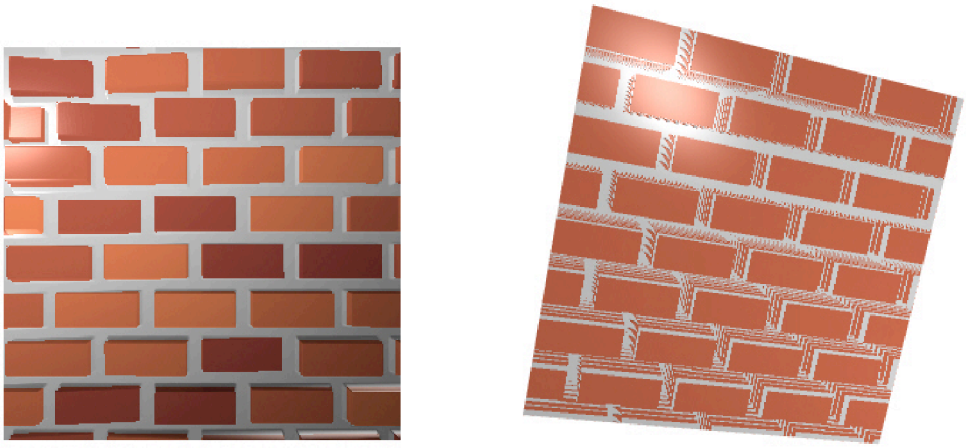

Abbildung 4.3: Backsteinshader und POM: Textur und Funktion

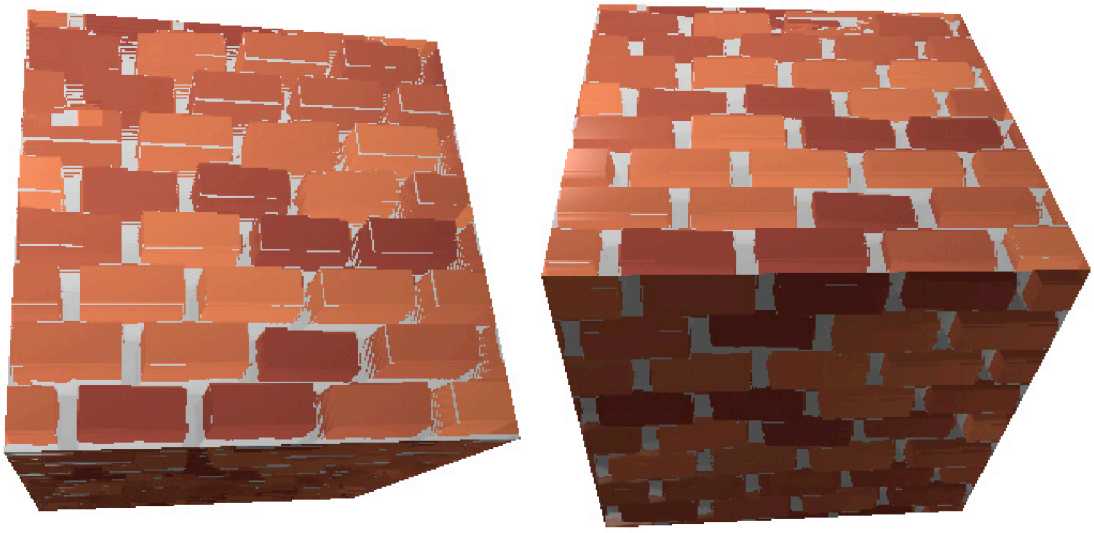

Abbildung 4.4: Backsteinshader und POM: Aliasing

Allerdings werden auch mehrere Probleme deutlich. Zum einen fehlt es der Oberfläche der Backsteine an Details, weshalb die Steine glatt wirken. Zum anderen sieht man einen weißen Saum an der Kante der Backsteine sowie backsteinfarbene Striche im Mörtel. Sie stammen von der Heightmapabtastung des POM. Dies wird deutlich, wenn man das rechte Bild der Abbildung 4.3 betrachtet. Das Bild wurde nicht mit einer Heightmap, sondern mit der Höhenfunktion direkt erzeugt. Da es die gleichen sichtbaren Artefakte aufweist, liegt das Problem in der Schnittpunktbestimmung: der Abtastung. Der Unterschied der Höhenwerte ist an den Kanten sehr hoch. Aus Abbildung 4.2 geht hervor, dass die Höhenwerte nicht langsam ansteigen, sondern direkt von 0 auf 1 springen. Es muss also eine sehr hohe Frequenz abgetastet werden. Da alle unter Kapitel 3.4 vorgestellten Verfahren diese Frequenz auf die gleiche Weise abtasten müssen, ist der Fehler verfahrensbedingt und nur durch das Verringern der Frequenz abzumildern. Immer kommt es zu Rundungsfehlern, Fehlabtastungen und schließlich den sichtbaren Artefakten. In Kapitel 4.5 und 4.6 werden Verfahren vorgestellt, die raue Oberflächen darstellen, ohne Texturen auszuwerten und diese Probleme daher nicht aufweisen.

Die Erzeugung des Backsteinmusters und der erörterten Parameter wirken sich nicht wesentlich auf die Laufzeit aus. Das POM-Verfahren bietet zwei Einstellmöglichkeiten, die direkten Einfluss auf die Laufzeit haben: minimale und maximale Samples für die Schnittfindung.

	Max. Samples	Frames	Messzeit [ms]	Min. [FPS]	Max. [FPS]	Avg. [FPS]
POM	23	140	11984	11	12	11.7
	50	120	11558	10	11	10.4
Backsteinshader	23	157	12127	12	23	13.0
	50	126	14231	8	10	8.9

Das Ergebnis für maximal 23 Samples überrascht, da der Backsteinshader das reine POM-Verfahren schlägt.

4.3 Normalengenerierung auf Höhenfeldern

In den nächsten Kapiteln werden Höhenfelder erstellt, die im Shader benutzt werden. Nicht nur die Höhe, sondern vielmehr die Normale jedes Höhenwertes ist dabei von Interesse – ist es doch die veränderliche Normale, die das Bumpmapping vom herkömmlichen Shading unterscheidet. Die State of the Art-Verfahren benutzen sowohl eine Heightmap als auch eine Normalmap. Sie können also auf vorberechnete Ergebnisse zurückgreifen. Die Verfahren in dieser Arbeit erzeugen die Höhenfelder dynamisch im Shader, so dass die Normale ebenfalls dynamisch berechnet werden muss. Im Folgenden werden daher geeignete Verfahren vorgestellt und evaluiert, so dass das geeignetste Verfahren weiter benutzt werden kann.

4.3.1 Konzept

Die Normale einer Ebene ist der Vektor, der rechtwinklig auf einer Ebene steht. Insbesondere gilt:

$$Normale \quad = \quad Tangente \times Binormale$$

Formel 4.1: Normale als Kreuzprodukt

Im Folgenden bezeichnet die Tangente den Vektor, der entlang der X-Achse einer Ebene zeigt. Die Binormale zeigt analog in die Richtung der Y-Achse.

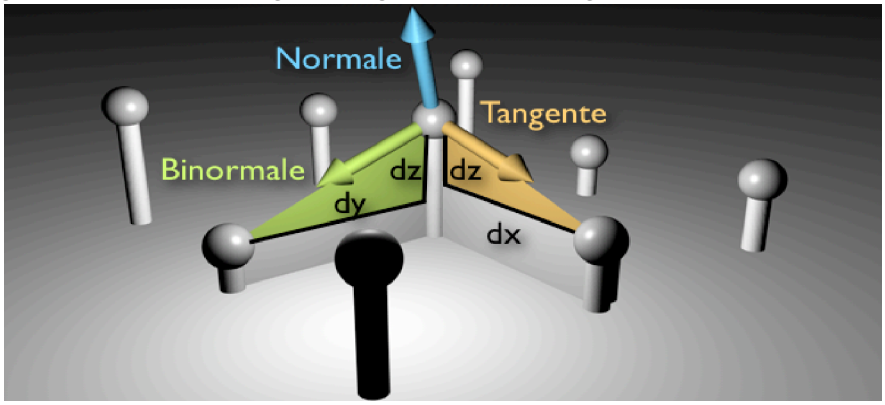

Abbildung 4.5: Normalengenerierung aus dem Höhenfeld

Da die Heightmap in einer Textur vorliegt, kann über eine zweidimensionale Texturkoordinate eindeutig auf einen Punkt der Heightmap zugegriffen werden.

Der Abstand zu den umliegenden Höhenfeldpunkten ist fix. Durch das Auslesen dieser Höhenwerte kann die Tangente und Binormale bestimmt werden (Abbildung 4.5) . Das Kreuzprodukt liefert gemäß Formel 4.1 die Normale.

Das 2er-Verfahren liefert eine Normale, die nur in Abhängigkeit von zwei benachbarten Punkten steht. Ein „innerer“ Punkt eines Höhenfeldes hat aber vier direkte Nachbarn. Für einen solchen Fall wird das Verfahren vier mal gedreht und auf die anderen Nachbarn angewandt („4er normal“). Die so berechneten Normalen werden addiert und normiert. Ebenso kann das Verfahren sogar auf die indirekten Nachbarn („4er rotiert“) erweitert werden (somit insgesamt acht Nachbarn). Abbildung 4.6 verdeutlicht die dabei nötigen Höhenfeldzugriffe:

Abbildung 4.6: Höhenfeldzugriffe der Normalengenerierung

4.3.2 Implementierung

Damit die Verfahren ihre Leistungsfähigkeit zeigen können, werden sie für optimale Bedingungen entwickelt, haben also freien Zugriff auf alle Höhenwerte. Um auf das nächste Texel zuzugreifen, muss die Texturauflösung bekannt sein. Ist b die Breite der Textur, so hat ein Texel eine Breite von $1/b$ im „Texturespace“, in dem jede Textur nach Konvention eine Breite von eins hat. Im Folgenden wird die Implementierung des normalen 4er-Verfahrens gezeigt, das leicht abgewandelt auch für das 8er-Verfahren und 2er-Verfahren angewandt werden kann:

```
float3 calcNormalFrom4Samples(float3 r, float3 l,
                              float3 a, float3 b,
                              float h, float scale){
    r.z = (r.z - h)*scale;//scaling has to be done
    l.z = (l.z - h)*scale;//'cause the distance to
    a.z = (a.z - h)*scale;//adjacent texel of the
    b.z = (b.z - h)*scale;//texturemap isn't 1

    r = normalize(r);
    l = normalize(l);
    a = normalize(a);
    b = normalize(b);

    return normalize(+cross(r, a)
                     -cross(l, a)
                     -cross(r, b)
                     +cross(l, b))
           * float3(1, -1, 1); // invert Y-Axis
}

float3 calcNormalHeight(float2 texCoord){
    const float delta = 1.0 / 512.0;//width of a Texel
    const float scale = 64.0;

    //define 4 vectors to ease coding
    const float2 v2r = float2( 1, 0);
    const float2 v2l = float2(-1, 0);
    const float2 v2a = float2( 0, 1);
    const float2 v2b = float2( 0,-1);

    //access the heightmap here
    float h = getHeight(texCoord);
    float r = getHeight(texCoord + v2r*delta);
    float l = getHeight(texCoord + v2l*delta);
    float a = getHeight(texCoord + v2a*delta);
    float b = getHeight(texCoord + v2b*delta);

    return calcNormalFrom4Samples(  float3(v2r, r),
                                    float3(v2l, l),
                                    float3(v2a, a),
                                    float3(v2b, b),
                                    h, scale         );
}
```

Listing 6: Normalengenerierung mit 4er-Verfahren

4.3.3 Evaluation

Die Kriterien, nach denen das geeignetste Verfahren bestimmt wird, sind Geschwindigkeit und Qualität der Ergebnisse.

Die durchzuführenden mathematischen Operationen sind minimal und werden direkt in der GPU durchgeführt. Sie fallen also deutlich weniger ins Gewicht als die Zugriffe auf die Textur, die das Maß für die Geschwindigkeit angeben.

	2er-Verfahren	4er-Verfahren	8er-Verfahren
Texturzugriffe	3	5	9
Normalisierungen	3	5	11
Kreuzprodukte	1	4	8

	Frames	Messzeit [ms]	Min. [FPS]	Max. [FPS]	Avg. [FPS]
2er-Verfahren	3991	14247	279	281	280.1
4er-Verfahren	2091	13688	152	154	152.8
8er-Verfahren	951	12676	74	76	75.0

Um die Qualität beurteilen zu können, wurde ein Texturset von NVIDIA gewählt, zu dem sowohl Heightmap und Normalmap existieren. Die implementierten Verfahren greifen auf die Heightmap zurück und die erzeugten Normalen können direkt mit den originalen Normalen verglichen werden.

In der ersten Reihe von Abbildung 4.7 sind die originale Heightmap und Normalmap abgebildet. Die zweite Reihe zeigt die Ergebnisse des Verfahrens für zwei, vier und acht umliegende direkte Nachbarn. Die letzte Bildreihe zeigt pixelweise die Differenz zur Originaltextur. Je heller die Farbe, desto größer die Differenz der entsprechenden Komponente des Normalenvektors. Da sie bereits im Fall für vier Nachbarn gering ist, wurden alle Differenzen um den Faktor 4,2 skaliert. Das Ergebnis wurde mit einem neunfachen Renderpass-Shader erzeugt.

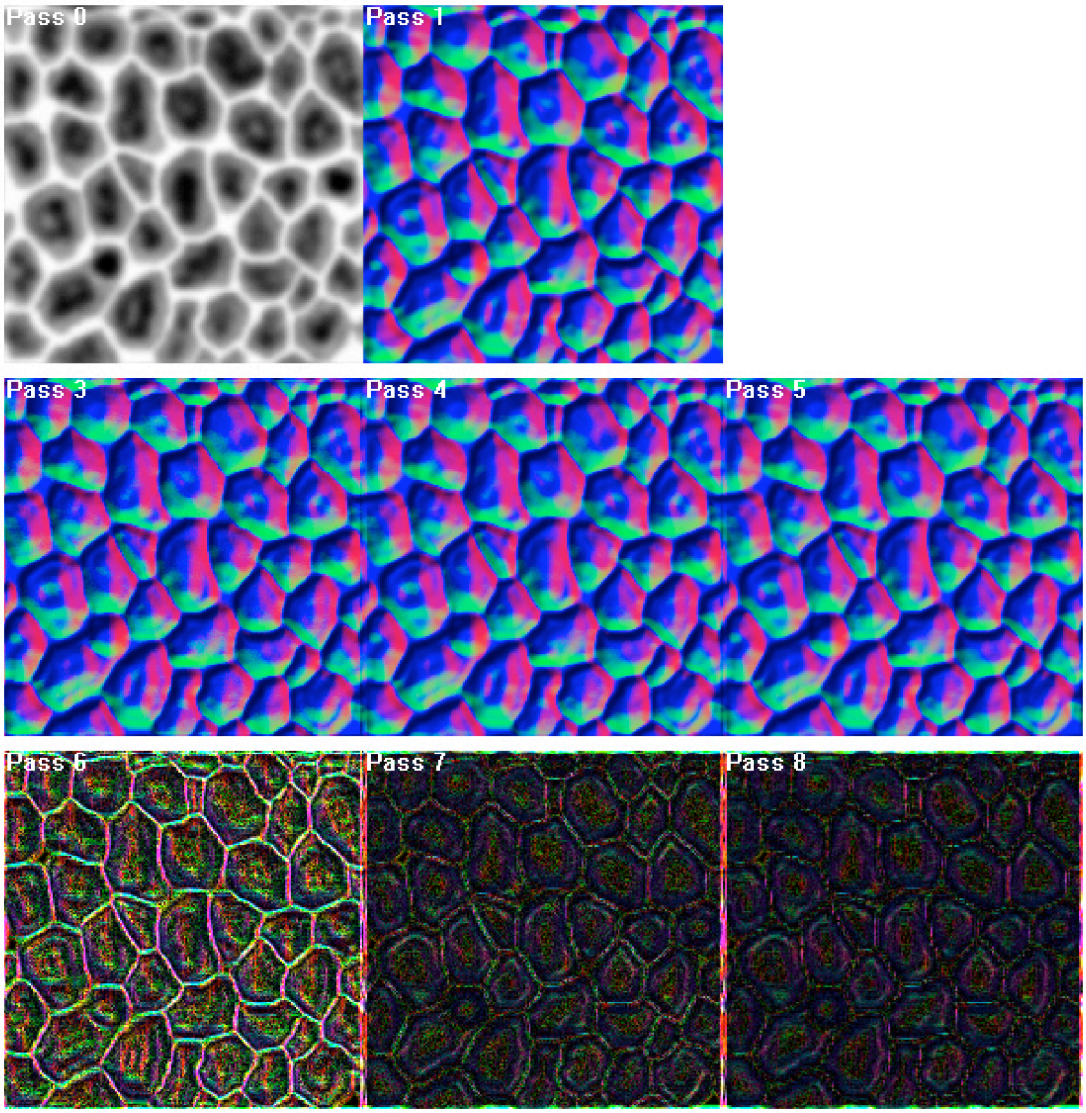

Abbildung 4.7: Normalengenerierungen im Vergleich

4.3.4 Fazit

Der Vergleich zeigt, dass das 2er Verfahren die deutlichsten Unterschiede zur originalen Normalmap hat. Zwischen dem 4er und 8er besteht nur ein kleiner Unterschied. Da das 8er-Verfahren allerdings fast doppelt so viele Texturzugriffe und Berechnungen benötigt, wird im Folgenden das 4er Verfahren verwendet.

4.4 Fraktale Landschaften im Shader

Der in 4.2 beschriebene Shader erstellt die Oberfläche einer Backsteinmauer. Sie besitzt jedoch keine Details, wenn die Mauer aus der Nähe betrachtet wird: Den Backsteinen fehlt es an Struktur. Um dies zu ändern, wird im Folgenden versucht, eine fraktale Landschaft – eine glaubwürdige Oberflächenstruktur – auf den Backstein zu übertragen. Gelingt es, ein solches Muster zu erzeugen, so lässt es sich auch in vielen anderen Situationen anwenden.

Für das Erzeugen von fraktalen Landschaften wird der in Kapitel 2.5 vorgestellten DiamondSquare-Algorithmus benutzt.

4.4.1 Konzept DSPatchmaps

Jeder Verfeinerungsschritt basiert auf Werten, die im Schritt zuvor berechnet wurden. Die Werte sind nicht nur an einen einzigen Pixel gebunden, sondern bauen auf den umliegenden Pixel auf. Da GPUs als Vertreter der SIMD-Prozessoren ihre Shader parallel ausführen, besteht keine Möglichkeit, Daten zwischen zwei Pixel auszutauschen. Es gibt keinen globalen Zwischenspeicher – jedes Pixel muss alle benötigten (Zwischen-)Werte selbst berechnen. In Konsequenz heißt das, dass nur ein Multipassverfahren größere DiamondSquare-Stücke erzeugen kann. Im folgenden wird daher ein Multipassverfahren beschrieben, das in jedem Renderpass das erzeugte Muster verfeinert.

Multipassverfahren für DiamondSquare

Jedes „Patch“,[14] das von einem Renderpass erzeugt wird, ist ein Höhenfeld der Größe $n \times n, n = 1 + 2^m, m \in \mathbb{N}$. Um das Patch effizient generieren zu können, wurden mehrere DiamondSquare-Stufen zusammengefasst. Pro Renderpass werden zwei Verfeinerungen erzeugt (Diamond und Square), die in eine Textur gerendert

14 Patch (eng.) = Flicken, Pflaster.

werden. Diese wird im nächsten Pass als Basis für die zwei folgenden Stufen benutzt.

Erster Renderpass

Im ersten Pass wird ein Patch der Größe 9x9 erstellt. Der äußere Rand hat eine festgelegte Höhe von 0. Alle anderen Werte werden gemäß eines leicht abgewandelten DiamondSquare-Algorithmus erzeugt, der sich besser im Shader implementieren lässt. Mit externen Parametern lässt sich der Einfluss der Zufallswerte auf den Mittelwert in jedem Schritt steuern.

Weitere Renderpasses

In jedem weiteren Renderpass wird nun das Patch in seiner Größe (fast) vervierfacht. Das Verfahren basiert auf dem des ersten Renderpasses, benutzt aber statt den Zufallswerten auf die Ergebnisse des vorherigen Passes.

4.4.2 Implementierung

Die Implementierung erfolgt, wie geschildert, in einem mehrstufigen Verfahren. Jeder Pass hat eine Iterationstiefe von drei Schritten.

Erster Renderpass

Zunächst wird die Texelkoordinate bestimmt. Sie entscheidet, welche der drei möglichen Stufen auszuwerten ist, wie in Abbildung 4.8 gezeigt.

1. Die Höhe ist fixiert auf 0 (Randgebiete) oder wird rein von einem zufälligen, skalierten Wert bestimmt (innere Felder). Im originalen Verfahren wird nur ein Teil dieser Werte derart fixiert. Die Belegung des Patchs entspricht der nach den ersten vier Phasen des DiamondSquare-Algorithmus. Sie wurden zusammengezogen, damit ein geeignet großes Patch erstellt werden kann, wie in Schritt 3 beschrieben.
2. Es wird ein Diamond-Verfeinerungsschritt durchgeführt. Entsprechend der Gitterbelegung sind die benötigten Werte alle rein zufallsbasiert, so dass sie

durch das Auswerten der umliegenden Zufallswerte ermittelt werden können. Die zweite Stufe greift auf fünf Zufallswerte zurück.

3. Beim anstehenden Square-Verfeinerungsschritt muss zwischen zwei Fällen unterschieden werden. Zwei der vier zu betrachtenden Werte entstammen der ersten Stufe, zwei der zweiten.

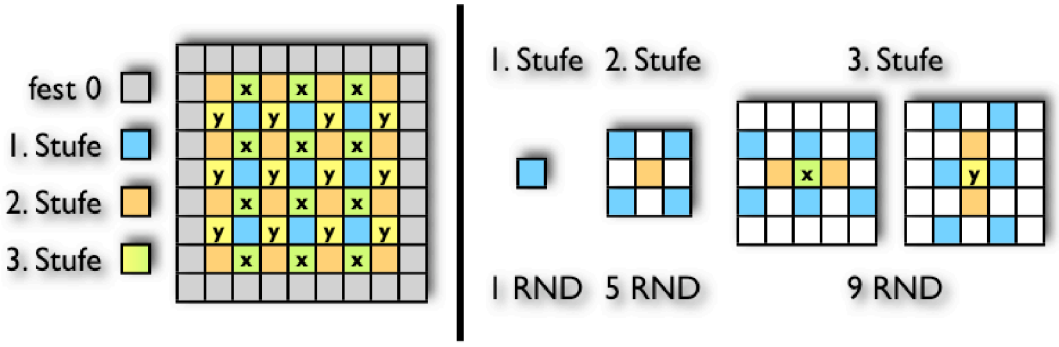

Abbildung 4.8: DSPatch-Generierung: Renderpass 1

Die Berechnung der ersten beiden Stufen kann problemlos mit maximal fünf Zufallswerten erfolgen. Die erste Stufe ist nicht direkt ersichtlich und wird im Folgenden anhand von Abbildung 4.9 näher erklärt.

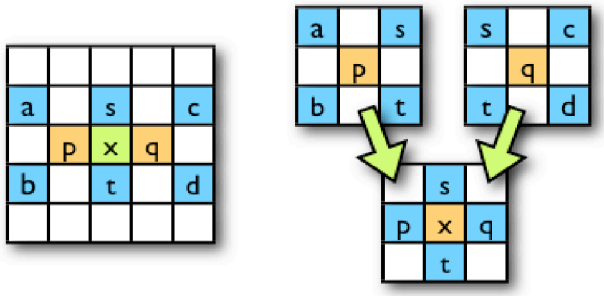

Abbildung 4.9: DSPatch-Generierung: Renderpass 1, Stufe 3, Fall x

Die Berechnung des Werts für x hängt von vier Werten ab: s und t sind direkt gelesene Zufallswerte, p und q ergeben sich aus der zweiten Stufe. Da diese Ergebnisse aber weder existieren, noch darauf zugegriffen werden darf, müssen sie für solche Texel möglichst effizient neu berechnet werden.

$$
\begin{aligned}
p &= \frac{a+b+s+t}{4} + rnd(pos_p) \\
q &= \frac{c+d+s+t}{4} + rnd(pos_q) \\
x &= \frac{p+q+s+t}{4} + rnd(pos_x) \\
\Rightarrow & \\
x &= \frac{a+b+c+d+6s+6t}{16} + \frac{rnd(pos_p)+rnd(pos_q)}{4} + rnd(pos_x)
\end{aligned}
$$

Formel 4.2: Herleitung der Höhenformel für Punkt x

Die Auswertung der Formel 4.2 zeigt, dass neun Zufallswerte bzw. Textur-zugriffe ausreichen. In dem ursprünglichen DiamondSquare-Algorithmus ergeben sich viel größere Abhängigkeiten und die Anzahl der sich daraus ergebenden Texturzugriffe verhindern eine Umsetzung.

Da die Anzahl der Texturzugriffe eines Pixelshaderprogramms beschränkt ist, ist es nicht möglich, ein größeres Patch zu generieren. Da aktuelle Shaderprogramme keine Rekursion zulassen, muss für jeden Iterationsschritt eine eigene Funktion geschrieben werden.

Weiterer Renderpass

Der Algorithmus entspricht dem des ersten Renderpasses. Der einzige Unterschied ist die Entstehung der Werte der ersten Stufe: Anstatt hier auf Zufallswerte zuzugreifen, werden die Werte aus der im vorigen Renderpass erzeugten Textur gelesen. Alle anderen Zufallswerte werden abhängig von der Texelkoordinate aus der Zufallstextur gelesen.

In Listing 7 ist die Kernmethode der Erzeugung der DSPatchmap mit Patchgröße 9x9 ausgeführt:

```
float3 calcDiamondSquare(int2 texel, int pSize, int pCount){
    float3 res = {0, 0, 0};
    const int2 patchSize2_shift = int2(pSize, pSize) - 1;
    const int2 patchCount2_shift = int2(pCount, pCount) - 1;
    const int2 tMod = mod(texel, int2(pSize, pSize));
    const int2 tDiv = div(texel, tMod,int2(pSize, pSize));
    if(!greaterAny(tDiv, patchCount2_shift)){
        if (   !equalsAny(tMod, int2(0,0)) &&
               !equalsAny(tMod, int2(patchSize2_shift))){
            if( isEven(tMod.x) && isEven(tMod.y)){
                res = getRND_1(tMod, tDiv);
            }else{
                float height = 0; float sum = 0;
                if (isEven(tMod.x + tMod.y)){
                    height = rnd_2(texel);
                    sum += getRND_1(tMod - int2( -1, -1), tDiv);
                    sum += getRND_1(tMod - int2(  1, -1), tDiv);
                    sum += getRND_1(tMod - int2(  1,  1), tDiv);
                    sum += getRND_1(tMod - int2( -1,  1), tDiv);
                    sum /= 4.0;
                }else{
                    height = rnd_3(texel);
                    if (isEven(tMod.x)){
                        sum += getRND_1(tMod - int2( -2, -1), tDiv);
                        sum += getRND_1(tMod - int2( -2,  1), tDiv);
                        sum += getRND_1(tMod - int2(  2, -1), tDiv);
                        sum += getRND_1(tMod - int2(  2,  1), tDiv);
                        sum += getRND_1(tMod - int2(  0, -1), tDiv)*6;
                        sum += getRND_1(tMod - int2(  0,  1), tDiv)*6;
                        sum += 4.0 *  intensity_p1.x *
                               (rnd_2(texel - int2(-1, 0))*2.0 - 1.0);
                        sum += 4.0 *  intensity_p1.x *
                               (rnd_2(texel - int2( 1, 0))*2.0 - 1.0);
                    }else{
                        sum += getRND_1(tMod - int2( -1, -2), tDiv);
                        sum += getRND_1(tMod - int2(  1, -2), tDiv);
                        sum += getRND_1(tMod - int2( -1,  2), tDiv);
                        sum += getRND_1(tMod - int2(  1,  2), tDiv);
                        sum += getRND_1(tMod - int2( -1,  0), tDiv)*6;
                        sum += getRND_1(tMod - int2(  1,  0), tDiv)*6;
                        sum += 4.0 *  intensity_p1.x *
                               (rnd_2(texel - int2( 0, -1))*2.0 - 1.0);
                        sum += 4.0 *  intensity_p1.x *
                               (rnd_2(texel - int2( 0,  1))*2.0 - 1.0);
                    }   sum /= 16.0;
                }   res = height + sum;
            }}} return res;}
```

Listing 7: Erzeugung der 9x9 DSPatchmap

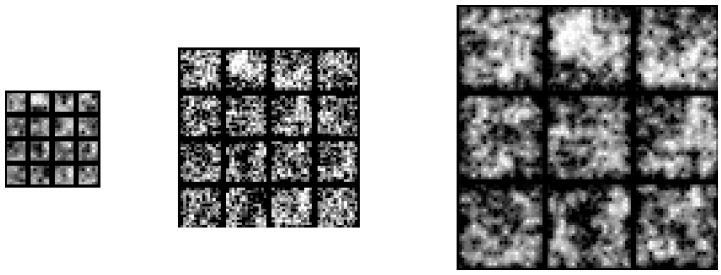

Abbildung 4.10: DSPatchmaps: 9x9, 17x17 und 33x33

4.4.3 Einsatz von DSPatchmaps

Die erzeugten DSPatchmaps können auf zwei verschiedene Arten eingesetzt werden: Als Variation einer bestehenden Heightmap oder bei der Erzeugung einer neun Heightmap. In beiden Fällen wird auf die DSPatchmap wie auf eine Heightmap zugegriffen. Da sich mehrere Patchs auf einer Patchmap befinden, muss pro Pixel bestimmt werden, welches der Patchs adressiert werden soll. Dies kann entweder über Zusammenhangsinformationen (gleicher Stein = gleiches Patch) oder aufgrund anderer Parameter erfolgen.

Anwendungsfall 1: Variation bestehender Heightmaps

Die DSPatchmap wird benutzt, um einer Heightmap zusätzliche Oberflächendetails hinzuzufügen. Bei der Bestimmung des Höhenwertes wird sowohl der Wert der originalen Heightmap wie auch der Wert der Patchmap in eine Variable gelesen. Je nach Grad des Einflusses der Patchmap wird der gelesene Wert skaliert. Da der Wert vor der Skalierung in eine Shadervariable eingelesen wird, ändert sich der Wertebereich von 8 Bit in den Wertebereich einer Float-Variable, üblicherweise 32 Bit.

Das Verfahren erreicht zwar schon gute Ergebnisse, allerdings ist es unter Umständen nicht immer gewollt, dass die gesamte Heightmap mit Details versehen wird. Oftmals soll nur ein gewisser Bereich variiert werden: Der Mörtel soll etwa flach bleiben. Damit die Patchmaps konsistent auf einer Oberfläche verteilt werden können, ist es ferner sinnvoll, bestimmte Areale der Heightmap mit einem gleichen Patch zu variieren. Für beide Zwecke müssen sie als Zusammenhangsinformationen

gespeichert werden, die im Shader ausgelesen werden können. Es ist nahe liegend, auch hier auf eine Textur zurückzugreifen.

Anstatt die Option variieren – nicht variieren als Boolesche Entscheidung in einem eigenen Farbkanal zu speichern, kann der Parameter Intensität der Variation aufgefasst werden.

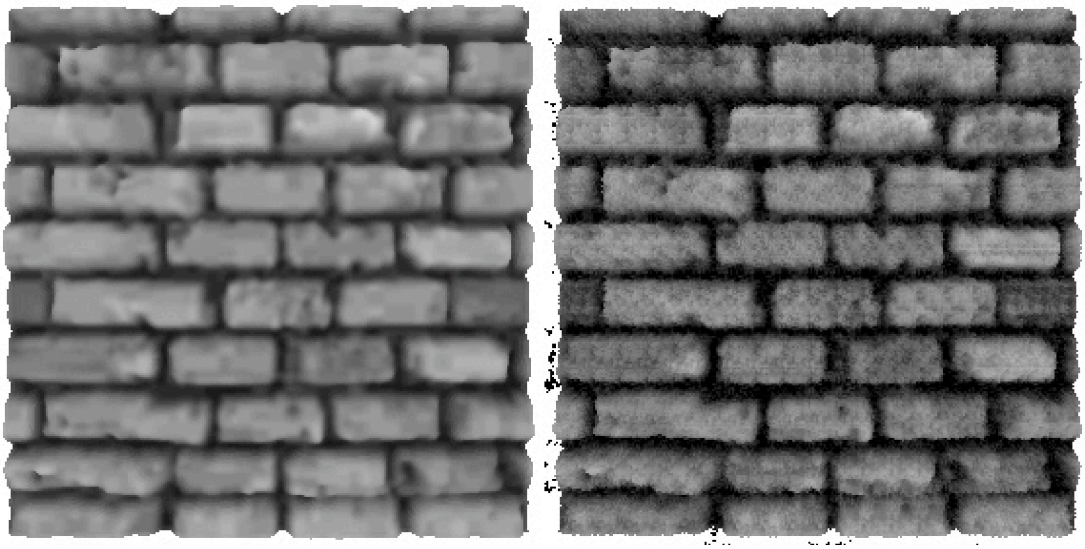

Abbildung 4.11: Originale und variierte Heightmap

Abbildung 4.12 zeigt ein originales Texturenset der Firma ATI, das mit dem POM-Verfahren gerendert wurde. Das Ergebnis in Abbildung 4.13 benutzt variierte Normalen mit der originalen Heightmap. In Abbildung 4.14 wurden zusätzlich noch die Höhenwerte fraktalisiert.

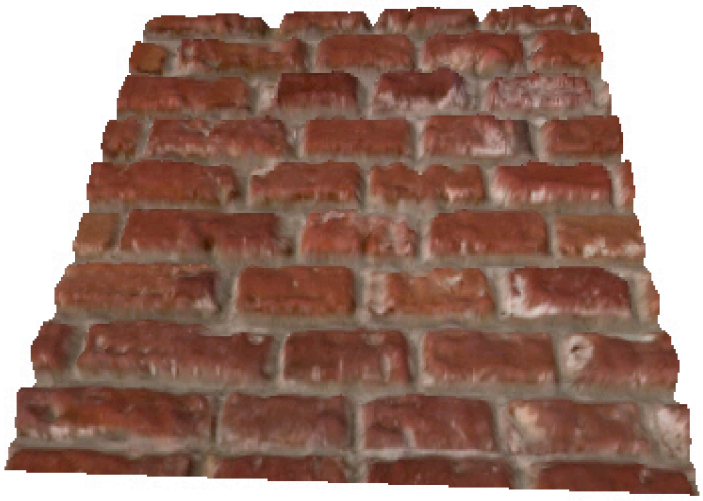

Abbildung 4.12: Backsteine: Originales Texturenset

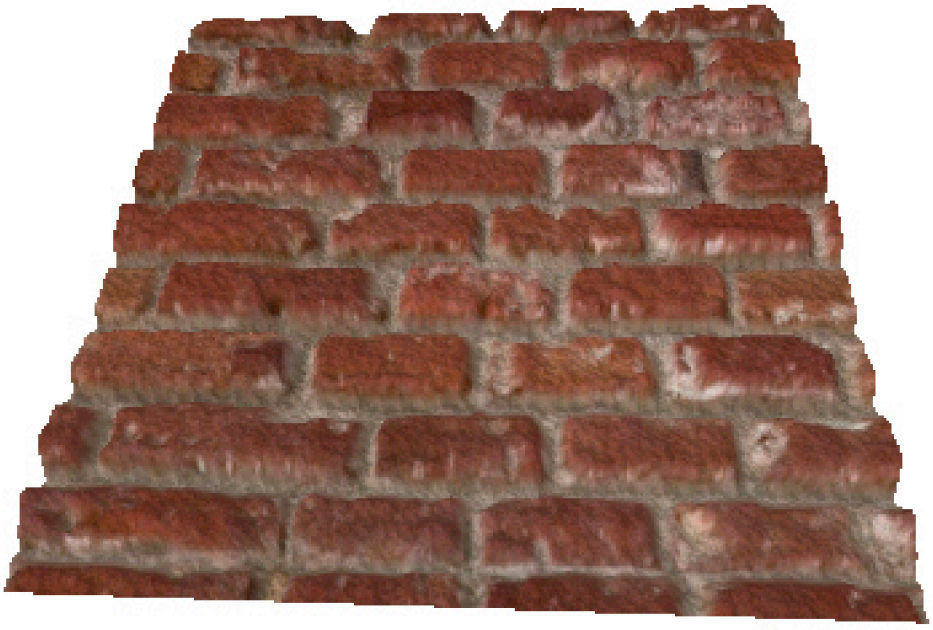

Abbildung 4.13: Backsteine: variierte Normalen

Abbildung 4.14: Backsteine: Normale und Heightmap variiert

Anwendungsfall 2: Erzeugung einer neuen Heightmap

Die Erzeugung einer Heightmap nur mit den DSPatchmaps ist eine weitere Anwendungsmöglichkeit. Da allerdings die einzelnen Patchs eine beschränkte Größe haben, ist die Generierung einer Oberflächenstruktur alleine aus Patches mit Problemen verbunden. Die Stücke müssen derart aneinander angelegt werden, dass keine sichtbaren Übergänge vorhanden sind. Dies kann zwar mit weichen Übergängen und mehreren Ebenen erreicht werden. Jedoch ist der Aufwand allerdings nicht unerheblich und das Heightmap-Tracing-Verfahren produziert Artefakte.

Die Patchs sind von begrenzter Größe. Ein komplexes Material aus begrenzten Teilstücken derart zusammenzufügen, ist zwar möglich, aber nicht effektiv in Hinblick auf ein LOD-System. Zwischen den verschiedenen Stufen der bereits generierten DSPatchs kann zwar interpoliert werden, eine glaubwürdige Vergrößerung ist aber nur schwer erreichbar.

Hinzu kommt, dass die Anzahl der Patchs und deren Aussehen „limitiert“ sind. Patchs, die größer als 9x9 Pixel sind, können nur über das Multipassverfahren erzeugt werden, müssen also in einer Textur gespeichert werden. Somit sind die erzeugten Stücke begrenzt. Globale Parameter können zwar in den Multipass-prozess eingreifen und etwa den Seed-Wert des Zufallsgenerstors und damit alle Patchs ändern. Aber sobald eine Stufe erzeugt wurde, kann die nächste Stufe nur auf die zuvor erzeugte DSPatchmap zurückgreifen. Zwar gibt es auch hier Mittel und Wege, die Anzahl, bzw. die Variation zu erhöhen, jedoch beeinflusst dies nicht die Systematik hinter dem Problem: Der DiamondSquare-Algorithmus kann aufgrund seiner benötigten Zusammenhangsinformationen zwischen den Pixeln nicht in beliebiger Verfeinerung im Shader implementiert werden. Das kann sich ändern, wenn die Shader mächtiger werden und geeignete Methoden zur Verfügung stellen.

4.4.4 Bewertung

Die Erzeugung der DSPatchmaps erfordert erheblichen Rechenaufwand, sofern ein einzelner Patch eine brauchbare Größe hat. Daher wurde die DSPatchmap in einem Vorbereitungsschritt erzeugt, abgespeichert und als statische Textur benutzt.

	Frames	Messzeit	Min. FPS	Max. FPS	Avg. FPS
DSPatch 9x9	1094	11923	91	92	91.8
DSPatch 9x9, 17x17	446	14625	30	31	30.5
DSPatch 9x9, 17x17, 33x33	110	16491	6	8	6.7

Die Anwendung von DSPatchMaps zur Aufwertung bestehender Texturensets bietet interessante Ergebnisse. Texturen, die unter künstlich wirkender „Glattheit“ leiden, bekommen besonders bei näherer Betrachtung echt aussehende Rauheit. Allerdings hat das Verfahren auch Grenzen: Der neu gewonnene Detailgrad wirkt sich nur auf die Heightmap aus. Die farbgebende Textur muss weiterhin mit einer Interpolation auskommen. Hier hilft die neu gewonnene Normale mit einer anderen Beleuchtung. Diese kann eine raue Oberfläche simulieren und in Szenen mit dynamischen Lichtern auch glaubhafte Bilder generieren.

Die Höhenwerte zu variieren bringt zusätzliche Probleme in den Algorithmus. Die in Kapitel 3.5 erörterten sichtbaren Aliasingartefakte werden durch feine Strukturen nur noch verstärkt. Insbesondere „verlorene Pixel“ treten vermehrt auf, da die Abtastung zusammenhängenden Pixelspitzen fehlerhaft ist.

	Max. Samples	Frames	Messzeit [ms]	Min. [FPS]	Max. [FPS]	Avg. [FPS]
Nur Normale	23	70	17433	2	6	4.0
	50	61	15799	2	6	3.9
Heightmap + Normale	23	21	60000	0	5	0.4
	50	11	27684	0	1	0.4

Grund für die schlechten Laufzeit ist der aufwändige Rechenvorgang, der beim Anbringen eines DSPatchs durchgeführt werden muss. Um den richtigen DSPatch aus der DSPatchmap auszuwählen, werden viele DIV- und MOD-Operationen benötigt.

Der alleinige Einsatz von DSPatchmaps kann nicht sinnvoll mit dem prozeduralen Generieren von komplexen Oberflächenstrukturen verbunden werden. Im Folgenden wird untersucht, inwiefern sich dieses Verfahren dennoch eignet, um einfach zu beschreibende Oberflächenstrukturen generieren.

4.5 SubPixelSurface mit DSPatchs

Eine weitere Anwendung der DSPatchs ist das SubPixelSurface-Verfahren (kurz SPS). Pro Pixel wird ein (kleines) Patch im Shader erzeugt, das die Oberfläche eines Materials für einen einzelnen Pixel imitiert. Mit Hilfe von Texturkoordinaten und (mathematischen) impliziten und polynomiellen Funktionen können Muster, die größer als ein Pixel sind, realisiert werden. Das System kann auch mit einem LOD-System verbunden werden, so dass der Einfluss des SubPixelSurface um so größer wird, je näher sie am Betrachter liegen.

Das Verfahren basiert nicht mehr auf einem Heightmap-Tracing-Verfahren, sondern greift die ursprüngliche Idee des Bumpmappings auf. Statt einer (diskreten) Bumpmap wird eine Bumpfunktion ausgewertet. So werden einige der beobachteten negativen Effekte vermieden. Die im Folgenden beschriebenen Methoden lassen sich mit Heightmap-Tracing-Algorithmen kombinieren, jedoch bedeutet dieser Schritt auch die Inkaufnahme eben jener Artefakte.

4.5.1 Konzept

Für jedes Pixel wird ein kleiner Patch erzeugt, der einen kleinen Oberflächenausschnitt imitiert. Die Normalen jedes Patchpunktes werden beim Shading eingesetzt.

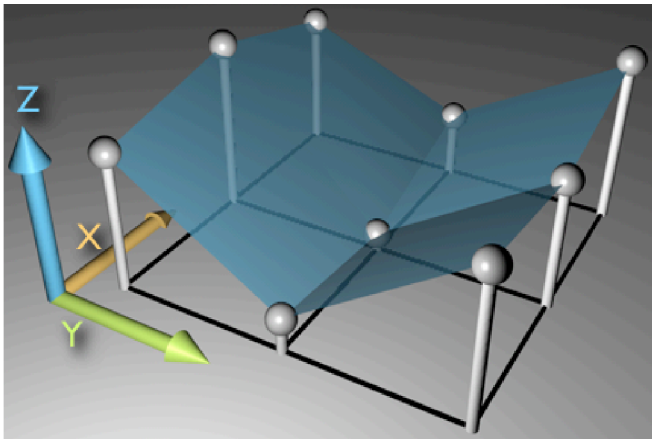

Abbildung 4.15: Oberfläche eines SPS-Patch

Das Patch weicht leicht von den in Kapitel 4.4.1 beschriebenen DSPatchs ab, da es andere Anforderungen erfüllen muss:

- Die Größe des DSPatchs wird für eine einfachere Verarbeitung auf 3x3 Höhenwerte reduziert. Somit ist ein Patch in einer float3x3-Variable speicherbar. Dabei fällt die Beschränkung weg, dass der Rand flach (= Höhe 0) sein muss. Das Patch ist dadurch einfacher zu berechnen und kommt mit weniger Texturzugriffen aus. Somit ist es auch im Shader, der die Oberfläche erzeugt, direkt generierbar. Er benötigt keine „Gedächtnistextur“ mehr - für jedes Pixel kann ein individuelles Patch erstellt werden.
- Das DSPatch muss in seiner Ausrichtung anpassbar sein. Während sich ein originales Patch in alle Richtungen gleich „entwickelt“, muss hier die Richtung steuerbar sein. Bestimmte Richtungen sollen gegenüber anderen favorisiert werden. Dadurch werden bestimmte Effekte, wie etwa eine Materialausrichtung, ermöglicht.

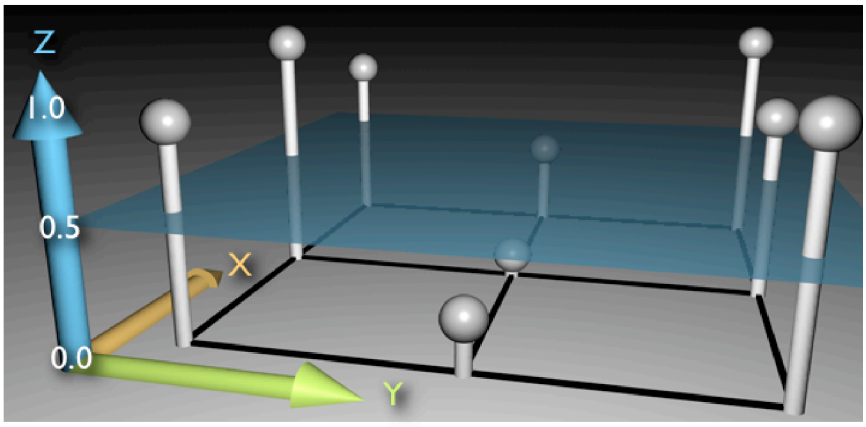

Abbildung 4.16: Aufbau eines SPS-Patch, gerichtet entlang der X-Achse

4.5.2 Implementierung

Listing 8 zeigt die Generierung eines Patchs, das entlang der X-Achse ausgerichtet ist:

```
float3x3 createPositiveHeightPatch( float3 seed,
                                    float intensityFactor){
    float3x3 h = 0;
    seed *= fStrokeWidth;

//  R X  Y
//            rnd returns [0;1] avg: 0.5
    h[0][0] = rnd(seed * 07.21);
    h[1][0] = rnd(seed * 09.33);
    h[2][0] = rnd(seed * 11.19);
    h[0][1] = rnd(seed * 2.98);
    h[1][1] = rnd(seed);
    h[2][1] = rnd(seed * 4.38);
    h[0][2] = rnd(seed * 3.57);
    h[1][2] = rnd(seed * 1.72);
    h[2][2] = rnd(seed * 9.61);

//  1st Row: [ 0; 1]
    h[0][0] *=  1.0;  h[1][0] *=  1.0;  h[2][0] *=  1.0;
//  2rd Row: [-1; 0]
    h[0][1] *= -1.0;  h[1][1] *= -1.0;  h[2][1] *= -1.0;
//  3rd Row: [ 0; 1]
    h[0][2] *=  1.0;  h[1][2] *=  1.0;  h[2][2] *=  1.0;
//  scale back to [0;1]
    h = h*0.5+0.5;
//  apply desired intensity
    h *= intensityFactor;

    return h;
}
```

Listing 8: Erzeugung eines SPS-Patchs

Der nächste Schritt ist die Berechnung der Normalen des Höhenpatchs. Die Höhenwerte werden nur für die strukturellen Informationen benötigt. Da das Verfahren aber nur Normalen benutzt, werden diese aus dem Höhenpatch (vgl. Kapitel 4.3) erzeugt.

4.5.3 SubPixelSurface Lighting

Mit einem so erzeugten Patch lassen sich mit dem SubPixelSurface-Lighting (kurz: SPS-L) anisotrope[15] Oberflächen imitieren . Die Beleuchtungsrechnung wird nicht für eine einzige Normale durchgeführt, sondern für jede der neun Normalen. Durch Wiederholen des Beleuchtungsvorgangs können auch mehr als neun Normalen pro Pixel ausgewertet werden. Dies betrifft allerdings nicht den ambienten Lichtanteil, da dieser unabhängig von der Normalen ist. Für besondere Fülleffekte kann die Berechnung des diffusen und glänzenden Anteils mit getrennten Patchs erfolgen.

4.5.4 Implementierung

Für die Kombination der berechneten Beleuchtungswerte wurden verschiedene Möglichkeiten implementiert:

- Skalierte Beleuchtungswerte

  Je nach Anzahl der Normalen wird der Wert entsprechend skaliert. Bei etwa neun Normalen wird der berechnete Wert addiert und durch neun dividiert.

- Gesättigte Beleuchtungswerte

  Alle errechneten Werte werden addiert und bei dem Maximum (pro Kanal jeweils 1.0) abgeschnitten.

- Maximum

  Das Maximum aller berechneten Werte wird zurückgegeben.

Zwar sind noch andere Möglichkeiten denkbar (wie etwa die Multiplikation aller Werte), sie haben sich aber im Praxisvergleich als nicht brauchbar herausgestellt. Insbesondere die skalierten Werte haben glaubwürdige Ergebnisse erzielt.

Der Algorithmus wird in Listing 9 gezeigt:

15 Kamera- und Beleuchtungswinkel abhängiges Reflektanzverhalten.

```
float4 anisotropyLighting(float2 texCoord,
                          float3 light,
                          float3 viewer,
                          float2 preRotCalc,
                          bool bUseAnisotropy){
    float4 res = float4(0,0,0,1);
    float4 ambient = res;
    float4 diffuse = res;
    float4 specular = res;
    float3 normal = float3(0,0,1);

    texCoord = texCoord * fTexRepeat;
    texCoord.x *= fStrokeType;

    float3x3 h1 = createPositiveHeightPatch(float3(texCoord, 0),
                                            (fAPIntensity / 10));
    float3x3 h2 = createPositiveHeightPatch(float3(texCoord, 0),
                                            (fAPIntensity / 8));

    ambient =  getAmbient(color);

    if (bUseAnisotropy){
        float n = 7.0;
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                normal = calcNormalFrom4SamplesHeight(h1, i, j);
                normal = rotZAxis(normal, preRotCalc);
                specular +=(getSpecular(normal, light, viewer));

                normal = calcNormalFrom4SamplesHeight(h2, i, j);
                normal = rotZAxis(normal, preRotCalc);
                diffuse += getDiffuse(normal, light, color);
            }
        }
        res = ambient + diffuse/n + specular/n;
    }else{
        res = doSpecularLighting(normal, light, viewer, color);
    }

}
```

Listing 9: SPS-Lighting

Anwendungsbeispiel: Gebürstetes Metall

Mit den oben beschriebenen Methoden wurde gebürstetes Metall imitiert. Die Riefen des Bürstens können sehr gut von Patchmaps simuliert werden – an jedem beobachteten Punkt wird Licht von allen Seiten der Bürstrichtung reflektiert. Dies fällt insbesondere bei Glanzlichtern stark auf (Abbildung 4.17).

Abbildung 4.17: Gebürstetes Metall: ein Ikea Kochtopf

Der Shader besitzt folgende Parameter:

- Bürstrichtung

  Die Richtung, in der gebürstet wurde, ist unter Angabe eines Winkels relativ zur X-Achse der Texturkoordinaten frei wählbar.

- Stärke der Bürstung

  Der Grad der Bürstung, also die maximale Stärke der Ablenkung der Normalen relativ zur X-Achse, ist frei bestimmbar.

- Einfluss der Anisotropie

  Mehr für die Verdeutlichung des Effektes als für die Simulation des Materials kann zwischen dem originalen Beleuchtungsmodell (Ambient-,

Labert-, Phong-Beleuchtung) und dem SPS-L in beliebigen Schritten überblendet werden.

Die Rotation der Bürstrichtung wurde durch eine einfache Rotation um die Z-Achse erreicht (Formel 4.3). Alle Normalen des Patchs werden mit Hilfe der Matrixrotation entsprechend des Winkels gedreht und dann in der Berechnung genutzt.

Um maximale Effizienz zu erreichen, wird die Rotationsmatrix für die

$$\mathbf{R_z}(\phi) = \begin{pmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Formel 4.3: Matrix der Rotation um Z-Achse

Bürstrichtung zu Beginn des Verfahrens berechnet und dann (mehrfach) auf die vorberechneten Werte zurückgegriffen.

Das Verfahren ist auch geeignet, um andere Materialien zu simulieren, deren Beleuchtung von „Mikrostrukturen“ abhängt. So gibt es etwa Lacke, die mit Mikropartikeln versehen sind. Die Lichtreflektanz ist nicht uniform über die Oberfläche, sondern wirkt eher zufällig. Mit dem SubPixelSurface Lighting kann dieses Verhalten imitiert werden.

4.5.5 SubPixelSurface Environmentmapping

Die Normalen sind nicht nur für die Beleuchtungsrechnung nutzbar. Sie können auch für das Environmentmapping benutzt werden, um auf eine Cubemap zuzugreifen. Das Verfahren entspricht dem SubPixelSurface Lighting, nur dass mit dem Höhenpatch keine Normalenbeleuchtung, sondern ein Cubemapping durchgeführt wird. Ergebnis ist das SubPixelSurface Environmentmapping (kurz: SPS-EM).

Das Verfahren entspricht weitestgehend dem des SPS-L. Der Wert der Environmentmap muss gelesen und (wenn gewünscht) mit dem Beleuchtungswert zusammengerechnet werden. Da die Environmentmap-Zugriffe sehr teuer sind, muss das SPS-EM an zwei Punkten eingeschränkt werden:

- Es wird nur ein Environmentmap-Wert bestimmt.

  Alle Normalen werden aufaddiert und das normalisierte Ergebnis anschließend für Cubemapping benutzt.

- Es wird nur eine Spalte des SPS-Patchs ausgewertet.

  Die äußere Schleife entfällt und die Indexvariable i wird fix auf 1, die mittlere Spalte, gesetzt.

Listing 10 zeigt die Erweiterungen zum SPS-Lighting:

```
float4 anisotropicLightingEM( [...], float3x3 mTangentToWorld){
   [...]
   float4 env = float4(0,0,0,1);

   if (bUseAnisotropy){
      int i = 1;
      float n = 3;
      float3 normalAvg = normal;
      for(int j=0; j<3; j++){
         normal = calcNormalFrom4SamplesHeight(h1, i, j);
         normal = rotZAxis(normal, preRotCalc);
         specular +=(getSpecular(normal, light, viewer));

         normalAvg += normal;
         [...]
      }
      env += (getTSCube(normalize(normalAvg), viewerTS,
                        mTangentToWorld) );
      res = (ambient + diffuse/n)
            *lerp(env, 1, saturate(fEMIntensityFactor.y));
      res += specular/n;
   }else{
   [...]
}
```

Listing 10: SPS-EM

4.5.6 Gesteuerte SubPixelSurface Verfahren

Beide SubPixelSurface-Methoden lassen sich pro Texturkoordinate in Intensität und Ausrichtung steuern, um etwa eine bestimmte Maserungen oder Muster einer Oberfläche zu imitieren. Dies ist besonders effektiv, wenn das Muster aus einer Funktion heraus generiert wird, denn eine Funktion kann beliebig genau ausgewertet werden. Artefakte, die beim Vergrößern einer Textur erzeugt werden, können so nicht entstehen. Hinzu kommt, dass Funktionen mit den Antialiasingmethoden für Pixelshader verbunden werden können.

4.5.7 Implementierung

In einer ersten Version wurde die Sinusfunktion ausgewertet, die den Bereich maskiert, der mit gebürstetem Metall versehen ist. Die Sinusfunktion ist in ihrer Frequenz und Bandbreite anpassbar. Im Shader wird pro Pixel anhand der Texturkoordinate die Funktion ausgewertet (Listing 11):

```
bool checkSine(float2 texCoord){
    texCoord.x += fTime0_1*fSineSpeed;
    texCoord.y = (texCoord.y - 0.5) * 2.0;

    float sinFunc = sin(texCoord.x * fSineFrequency * 2 * pi);

    float base = sinFunc * fSineAmplitude;

    float dst = (texCoord.y - base);

    float width = fSineWidth / 2.0;

    return (abs(dst) < width);
}
```

Listing 11: Gesteuertes SPS-Lighting

4.5.8 Bewertung

Die mit dem SPS-L gerenderten Bilder weisen die typischen Reflektanzcharakteristika anisotroper Materialien (Abbildung 4.19, 4.20 und 4.21) auf. Dies fällt besonders im Vergleich zum regulär beleuchteten Modell auf (Abbildung 4.18).

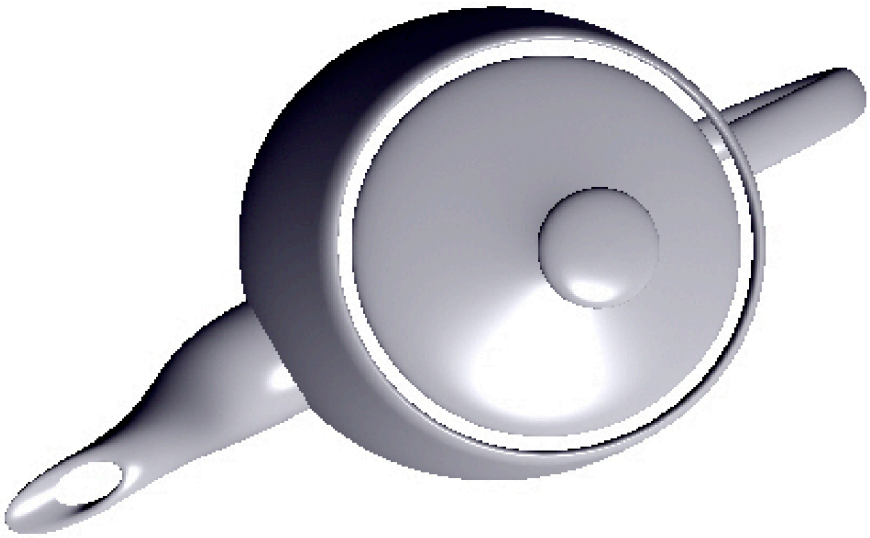

Abbildung 4.18: Utah-Teapot mit per Pixel Beleuchtung

Abbildung 4.19: Utah-Teapot mit SPS-Lighting; 208° Rotation

Abbildung 4.20: Utah-Teapot mit SPS-Lighting; 0° Rotation

Abbildung 4.21: Utah-Teapot mit SPS-Lighting; 90° Rotation

Benutzt man anstelle der neun SPS-Patch-Normalen nur drei für die Beleuchtungsrechnung, ändert sich das Erscheinungsbild der Reflektionen. Sie wirken weniger subtil, da weniger unterschiedliche Beleuchtungswerte miteinander vermischt werden (Abbildung 4.22).

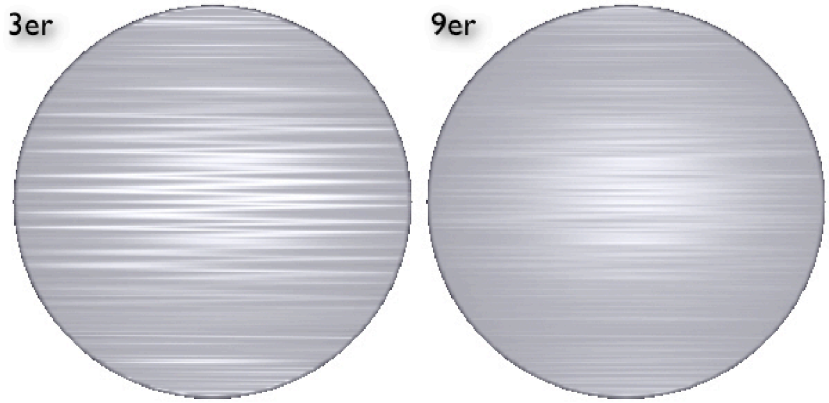

Abbildung 4.22: SPS-Lighting: 3 Samples und 9 Samples

Auch das SPS-EM Verfahren erreicht ein glaubwürdiges Ergebnis. Die Abbildung 4.23 zeigt eine Kombination aus SPS-L und Environmentmapping mit originaler Normale. Dagegen wurde für Abbildung 4.24, die Normale aus dem SPS-Patch gemittelt und für Cubemapping genutzt. Durch Bestimmen der perturbierten Normalen für das Cubemapping entstehen die typischen Verwischeffekte. Leider ist es aufgrund der aktuellen Hardwarebeschränkung nicht möglich, im Verfahren selbst pro Pixel mehrmals auf die Cubemap zuzugreifen. Dies würde den Effekt der diffusen Spiegelungen (wie sie auf dem Kochtopf aus Abbildung 4.17 zu erkennen sind) noch steigern. Mit einer einzelnen Normalen wirken die Striche noch grob.

Abbildung 4.23: SPS-EM, keine Normalenablenkung

Abbildung 4.24: SPS-EM mit Normalenablenkung

Die leichte Handhabung macht die Methoden universell einsetzbar. Intensitäts-, Mischungs- und Ausrichtungsparameter sind im Shader steuerbar und können etwa von prozeduralen Methoden zur Laufzeit kontrolliert werden. Dies wurde für eine einfache Materialienbeschreibung – eine Sinus-Kurve – im gesteuerten SPS-Lighting umgesetzt:

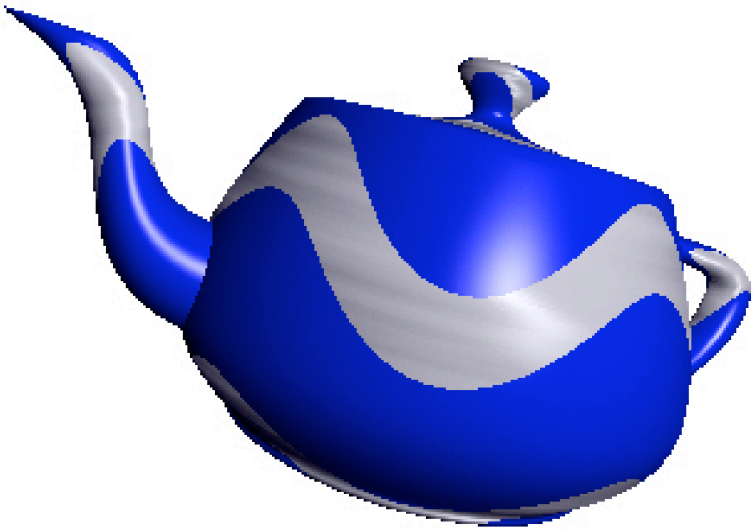

Abbildung 4.25: Gesteuertes SPS-Lighting

	Patch Samples	Frames	Messzeit [ms]	Min. [FPS]	Max. [FPS]	Avg. [FPS]
Phong-Lighting	-	4034	11498	349	353	350.8
SPS-Lighting	3	178	12310	14	15	14.5
	9	49	13031	2	5	3.8
SPS-EM	3	158	13083	11	14	12.0
	9	34	13906	0	4	2.4
Gesteuertes SPS-Lighting	3	271	12417	21	22	21.8

Die Geschwindigkeitsanalyse zeigt, dass die Auswertung des Patchs starken Einfluss auf die Performance hat. Dies fällt schon beim Vergleich der Laufzeiten für die zwei SPS-Lighting-Verfahren auf. Besonders deutlich wird der Unterschied aber, wenn man die Ergebnisse des gesteuerten SPS-L mit dem reinen SPS-L vergleicht. Der Performancegewinn von fast 50% ist nur darauf zurückzuführen, dass deutlich weniger Patchs ausgewertet werden.

4.6 Bumpmapping ohne Texturen

Im vorherigen Kapitel wurde bereits ein Verfahren gezeigt, das Muster einer Oberfläche darstellt, ohne eine Textur als Grundlage des Materials zu verwenden. Das Muster beschränkt sich allerdings auf zweidimensionale Funktionen.

Wie in Kapitel 3.5 erörtert, bringt der Einsatz von Texturen immer bestimmte Probleme mit sich. Da die Probleme zu Lasten der erzeugten Grafikqualität gehen, stellt sich die Frage, ob es nicht andere Methoden gibt, die Ähnliches (oder gar Besseres?) leisten, aber nicht mit diesen Nachteilen behaftet sind. Die Methoden werden immer noch unter dem Aspekt des Bumpmappings untersucht, aber sie benutzen keine „Maps“ bzw. Texturen. Daher wird der Begriff „Bumping“ eingeführt, um den Unterschied zu verdeutlichen. Bumping-Verfahren versuchen, Details einer Oberfläche darzustellen und lesen das Muster der Oberfläche nicht aus einer Textur. Sie arbeiten nur im Bildraum, erzeugen also keine Geometrie. Das Problem der Silhouette kann also nur eingeschränkt gelöst werden.

4.6.1 Konzept der BumpPattern

Ein Weg, die Probleme von texturbasierten Verfahren zu umgehen, ist die Vermeidung von Texturen. Wie in Kapitel 4.5.6 vorgestellt, gibt es Alternativen, die vektorielle Beschreibungen, polynomielle sowie implizite Funktionen zum Erzeugen von artefaktfrei skalierbaren „Texturen“ nutzen. Mit Texturen (im eigentlichen Sinne) haben die Verfahren allerdings wenig gemeinsam: Sie nutzen lediglich die Texturkoordinaten eines Modells, um die aktuelle relative Position des von ihnen beschriebenen Materials auswerten zu können. Sofern die Ausrichtung der Textur auf dem Modell eine wichtige Komponente des Materials ist, wird zusätzlich noch die Tangente und Binormale des Pixels benötigt.

Im Folgenden werden nun zwei Methoden vorgestellt, die auf Patterns[16] statt auf Texturen basieren. Die Patterns beschreiben die Struktur eines Materials nicht mit

16 Pattern (en.) = Muster, Schema, Ornament, Dekor.

absoluten, sondern mit relativen Angaben. Dadurch können die beschriebenen Muster in einem beliebigen Detailgrad ausgewertet werden. Sowohl Vergrößerung als auch Verkleinerung stellt kein Problem dar. Vergrößerung ist durch die detaillierte Auswertung, die durch die Texturkoordinate bedingt ist, gegeben. Für die Verkleinerung kann die Auswertung eines Patterns mit dem in 2.4.4 schon vorgestellten adaptiven analytischen Vorfiltern kombiniert werden: Ist die anzuzeigende Frequenz zu groß für den zur Verfügung stehenden Bereich, kann entweder die Frequenz angepasst werden (= zu kleine Muster entfallen) oder durch entsprechende Alternativen ersetzt werden (= Ersatzmuster + Ersatznormale).

4.6.2 Konzept der 2D-BumpPattern

Die Möglichkeit, implizite und polynomielle Funktionen, vektorielle Beschreibungen oder parametrische Kurven im Shader auszuwerten, wie in Kapitel 4.5.6 erläutert, ist nur eingeschränkt für ein Bumping-Verfahren nutzbar. Alle Methoden basieren auf 2D-Funktionen. Überdeckungen von Mustern, Formen oder Flächen sind nur auf Umwegen möglich. Nicht berücksichtigt werden dadurch Selbstschattierungs- und Verdeckungseffekte. Auch Parallax-Verschiebungen kann die Methode nicht erzeugen. Das Verfahren ist daher mit dem Normalmapping vergleichbar. Dennoch sind sie für einfache Muster nutzbar, wie im Folgenden gezeigt wird.

Die Idee ist es, aus impliziten und polynomiellen Funktionen das gewünschte Muster nachzubilden. Jede Funktion bildet ein Ornament, also jeweils ein Muster. Aufgrund der Texturkoordinate wird die Funktion ausgewertet und so bestimmt, welches Ornament unter dem Texel liegt. Jedem Ornament wird ein bestimmtes Material zugewiesen, das in der Beleuchtungsrechnung benutzt wird. Die Lage des Texels innerhalb des Ornaments wird benutzt, um die Normale zu bestimmen. Zwar sind über Z-Ebenen die Ornamente sortierbar, jedoch können einzelne Ornamente nicht mehr als eine Ebene einnehmen. Die Ornamente schneiden sich also nicht.

Zu jedem Ornament existieren:

- eine implizite oder polynomielle Funktion, die die Form festlegt,
- eine Initialposition,
- ein Wert, der die Z-Ebene angibt,
- eine Materialbeschreibung mit Farb- und/oder Texturattributen
- und eine Normalenfunktion, die die Normale jedes Punktes des Ornaments beschreibt.

4.6.3 Implementierung der 2D-BumpPattern

Zunächst wird der Schnittpunkt ermittelt, anschließend das entsprechende Shading durchgeführt.

Die Schnitttestfunktionen liefern einen 4-komponentigen Vektor zurück, dessen Entstehung die Abbildung 4.27 veranschaulicht. Abbildung 4.26 illustriert die Struktur des Materials.

Der erste unterste Ebene ist das Lochgitter. Außen ist ein Kegel, der in das Material hineinzeigt. Im Inneren ist der Kegel durch einen Kreis mit konstanter Farbe begrenzt. Die Radien beider Umrisse sind frei wählbar. Die Sinuskurve ist in ihrer Frequenz, Breite, Offset und Amplitude steuerbar.

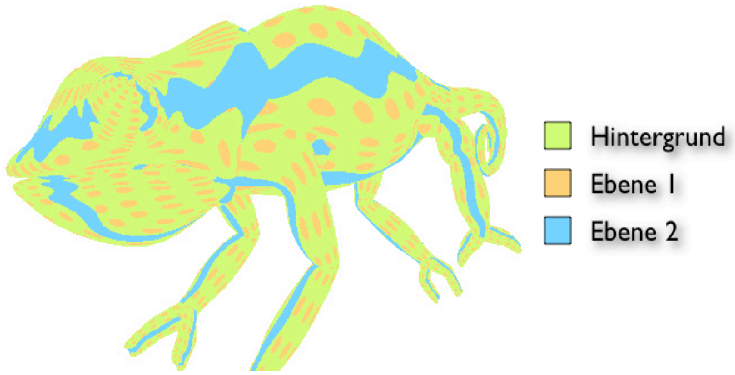

Abbildung 4.26: Struktur eines 2D-BumpPatterns

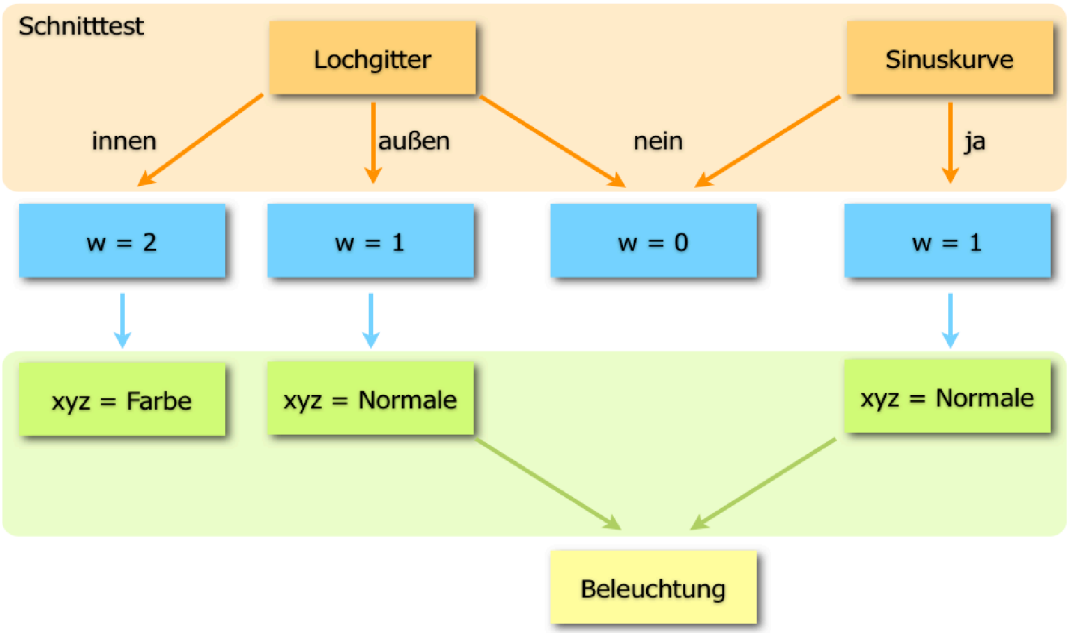

Abbildung 4.27: Rückgabewert der Schnittfunktionen

Der Schnitttest und Shadingcode der 2D-BumpPattern ist in Listing 12 zu finden. Zuerst wird das Lochgitter getestet. Kommt es zu einem Schnitt, werden die entsprechenden Paramater für die Beleuchtungsrechnung festgelegt (Kegel) oder eine fixe Farbe bestimmt. Danach wird die Sinuskurve getestet und bei Schnitt die Werte für das Shading festgelegt.

```
float4 checkHole(float2 texCoord){
    float4 res = 0;

    texCoord = frac(texCoord * fHolesCount) - 0.5;
    float r = sum(sqr(texCoord));
    if (r < holesRad.x){
        res.w = 1;
        res.xyz = normalize(float3(-normalize(texCoord), 0.5));
        if (r < holesRad.y){
            res.rgb = cDarkColor.rgb;
            res.w = 2;
        }
```

```
float4 checkSine(float2 texCoord){
    float4 res = 0;
    texCoord.x += fTime0_1*fSineSpeed;
    texCoord.y = (texCoord.y - 0.5) * 2.0;
    float width = fSineWidth / 2.0;
    float base = sin(texCoord.x * fSineFrequency * 2 * pi)
               * fSineAmplitude;
    float dst = (texCoord.y - base);
    if (abs(dst) < width){
        res.w = 1.0;
        res.y = (dst / width) * fSineHeight;
        res.z = cos(asin(t));
        res.xyz = normalize(res.xyz);
    }
    return res;
}
//shading here
float4 calcPixel2D(float2 texCoord, float3 lightTS,
                   float3 viewerTS, float3x3 mTangentToWorld){
    float4 res        = 1;
    float3 normal     = float3(0, 0, 1);
    bool   bDoLighting = true;
    float4 hole = checkHole(texCoord);
    if (hole.w > 0){
        bDoLighting = true;
        normal = hole.xyz;
        if (hole.w > 1){
            res.rgb = hole.rgb;
            bDoLighting = false;
        }
    }
    float4 sine = checkSine(texCoord);
    if (sine.w > 0){
        bDoLighting = true;
        normal = sine.xyz;
    }

    if (bDoLighting){
        res  = doDiffuseLighting(normal, lightTS, viewerTS,
                                                    cBaseColor);
        res *= getTSCube(normal, viewerTS, mTangentToWorld);
        res += getSpecularLighting(normal, lightTS, viewerTS,
                                                     cBaseColor);
    }
    return res;
}
```

Listing 12: 2D-BumpPattern: Shading und Schnitttest

4.6.4 Konzept der Raytraced-BumpPattern

Anstatt die (flachen) Ornamente um eine zweite 2D-Funktion auszustatten, erscheint es sinnvoller, eine geeignete Funktion zur Beschreibung der Oberflächenobjekte zu finden. Der nahe liegende Schritt ist die Erweiterung der Funktion um eine dritte Dimension. Mit 3D-Funktionen lassen sich Körper, wie etwa Kugel, Kegel, Zylinder oder Tori, beschreiben. Da die Objekte alle über drei Dimensionen beschrieben werden, sind somit auch beliebige Überlappungen möglich.

Mit der Erweiterung muss auch das Konzept überarbeitet werden. Eine zweidimensionale Texturkoordinate kann nicht alleine ausreichen, um eine dreidimensionale Funktion auszuwerten. Hier bietet es sich an , wie bei den Heightmap-Tracing-Verfahren aus Kapitel 3.4, den Sichtstrahl zu nutzen. In Verbindung mit der Texturkoordinate kann die Auftreffposition des Sichtstrahls (die Texturkoordinate erweitert um eine Z-Komponente mit Wert 0) festgelegt werden.

Ein Verfahren, das einen Sichtstrahl (bestehend aus Richtung und Startpunkt) nutzt, um ihn mit Objekten der Szene zu schneiden, wurde bereits 1968 von Appell unter dem Namen „Raytracing“ vorgestellt [App68]. Es gilt noch heute als das Verfahren, das die realistischsten dreidimensionalen Bilder im Computer generieren kann. Durch eine Erweiterung von Whitted [Whi80] waren Reflexion, Refraktion und transparenten Materialien darstellbar. Aktuelle Arbeiten setzen das Verfahren als Shader um [CHH02] und versuchen es durch verschiedene Speed-up-Techniken in Echtzeit zu realisieren, so etwa die Arbeiten von Christen [Chr05], Foley und Sugerman [FS05] oder Carr, Hoberock, Crane und Hart [CHCH06].

Die Idee hinter den Raytraced-BumpPattern (kurz: RBP) ist folgende: Die zu imitierende Oberflächenstruktur wird mit 3D-Körpern nachmodelliert. Jeder Körper besitzt:

- Eine Formel, die den Körper beschreibt,
- ein Positionsvektor,
- eine Normalenfunktion, die für einen Schnittpunkt P die Normale berechnen kann
- und eine Materialeigenschaft, die für die Beleuchtungsrechnung benutzt wird. Dies können:
    - Farben für die Beleuchtung

    und/oder

    - eine Textur zusammen mit einer Texturkoorrdinaten erzeugenden Funktion sein.

4.6.5 Implementierung der Raytraced-BumpPattern

Der Shader, der Raytraced-BumpPatterns berechnet, unterscheidet sich nur minimal von der Implementierung der 2D-BumpPattern. Daher wird im Folgenden nur auf die Unterschiede eingegangen.

Die Auswertung, welches Objekt getroffen wurde, geschieht anhand des Sichtstrahls, der aus einem Startpunkt und Richtungsvektor besteht. Startpunkt ist die Texturkoordinate erweitert um den Z-Wert 0; Richtungsvektor ist der Kameravektor im lokalen Koordinatensystem des Pixels.

Die Ornamente werden zu Objekten und von einer dreidimensionalen Funktion beschrieben. Da Funktionen in Shaderprogrammen nicht rekursiv aufgerufen werden können, sind zur Zeit noch keine Reflektionen oder Selbstschattierungen umgesetzt.

Das Verfahren kann auch mit dem Silhouettenbilden der Heightmap-Tracing-Verfahren kombiniert werden und somit „semireale“ Silhouetten erzeugen. Parallaxeneffekte sind genauso wie Selbstverdeckung schon jetzt möglich.

4.6.6 Bewertung

Das 2D-BumpPattern-Verfahren ist für grobe, flache Muster gut geeignet. Der durch die gebumpten Normalen erzeugte Tiefeneindruck ist sehr realistisch (Abbildung 4.28). Wird die Methode für kleine, oft wiederholte Muster angewandt, fallen die Beschränkungen nur unwesentlich auf. Das funktional beschriebene Muster kann sogar beliebig stark vergrößert werden, ohne dass es zu einer Aufpixelung kommt.

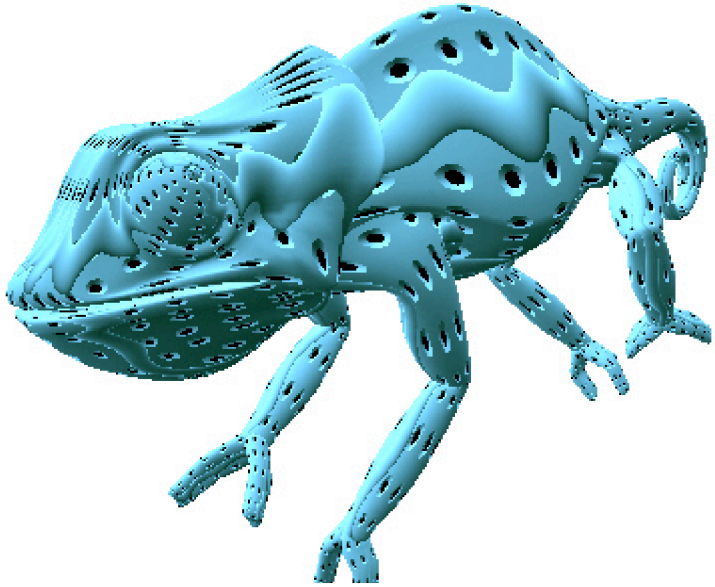

Abbildung 4.28: Lambertsche Beleuchtung mit 2D-BumpPattern

Die berechneten Normalen lassen sich auch für das Environmentmapping benutzen (Abbildung 4.29). Die Oberflächenstruktur wirkt besonders bei größeren Unebenheiten plastisch. Bei großen Normalenvariationen zwischen benachbarten Pixeln ist oft nicht gut erkennbar, welcher Teil der Umgebung reflektiert wird.

Abbildung 4.29: Cubemapping mit 2D-BumpPattern

Die Raytraced-BumpPattern bieten durch das beliebige Anordnen von „echten“ Körpern im Raum noch mehr Flexibilität. So lassen sich viele Muster als Oberfläche erzeugen und größere Strukturen sind pixelübergreifend überzeugend realisierbar (Abbildung 4.30).

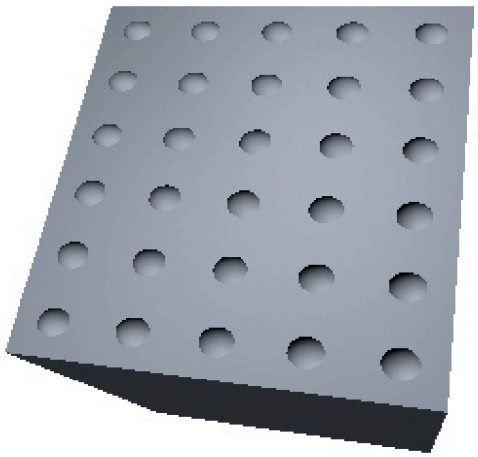

Abbildung 4.30: RBP-Kugeln

Der Raytracer enthält noch keinerlei Beschleunigungsstrukturen. Da eine (einfache) Oberflächenstruktur nachmodelliert werden soll, fällt dies noch nicht weiter ins Gewicht. Dennoch kann schon jetzt jedem Oberflächenobjekt ein Schattierungsmuster zugewiesen werden (Abbildungen 4.31, 4.32 und 4.33).

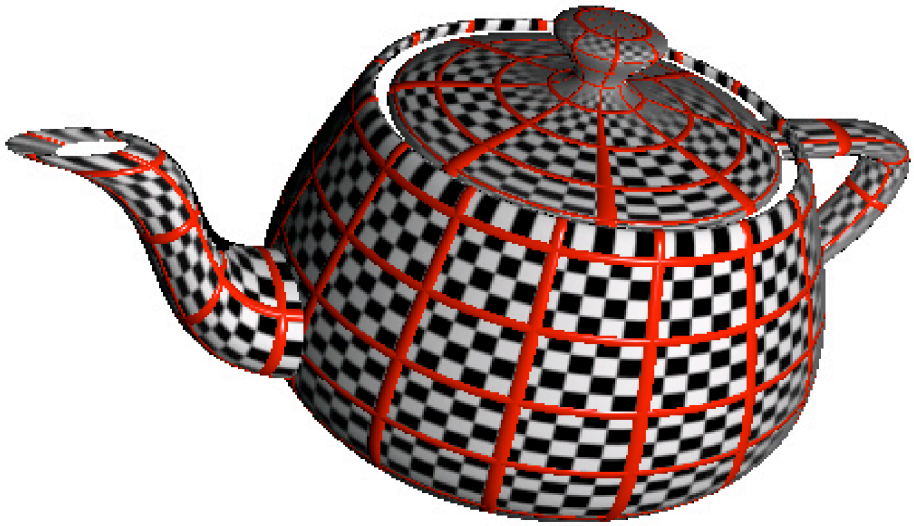

Abbildung 4.31: RBP auf dem Utah Teapot

Abbildung 4.32: RBP auf einem Chamäleon (TexRepeat = 2)

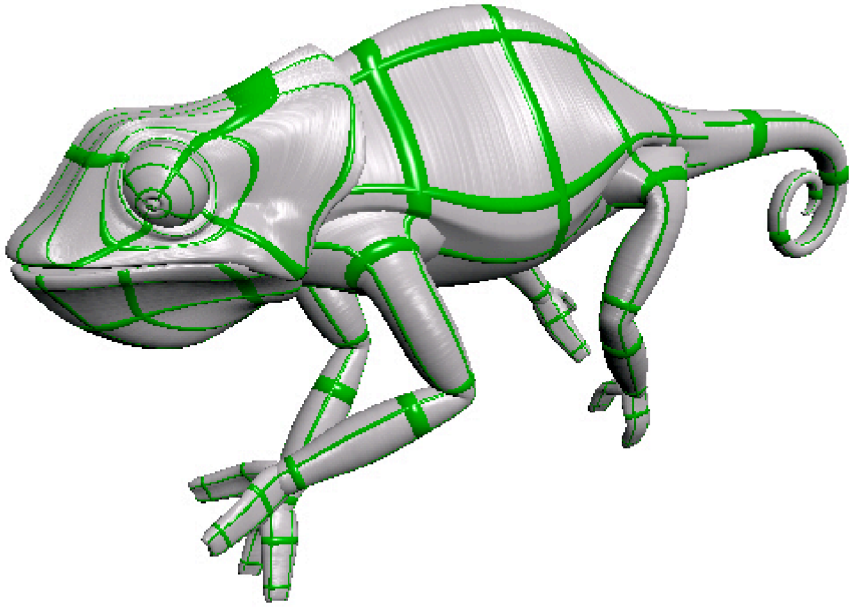

Abbildung 4.33: RBP und SPS-L auf einem Chamäleon (TexRepeat = 4)

	Methode	Frames	Messzeit [ms]	Min. [FPS]	Max. [FPS]	Avg. [FPS]
Untexturiert	Phong	4034	11498	349	353	350.8
2DBump-Pattern	Diffus	1405	12121	115	117	115.9
	Phong + EM	673	11724	57	58	57.4
RBP	Diffus	1957	11515	169	171	170.0
	Phong	523	11882	43	45	44.0

Die Geschwindigkeitsauswertung zeigt deutliche Unterschiede zum regulären Beleuchtungsverfahren. Der Grund dafür liegt vor allem im rechenintensiven Schnitttest. Dennoch sind die Laufzeiten vielversprechend: Schon jetzt sind Frameraten erreicht, die interaktive Anwendungen bei hohen Auflösungen ermöglicht.

4.7 Zusammenfassung

Kritischer Punkt aktueller Verfahren ist die Verwendung von Texturen. Durch sie entstehen sichtbare Artefakte, die nicht behoben werden können. Hinzu kommt, dass es Situationen gibt, in denen die Heightmap-Tracing-Verfahren zusätzliche Artefakte verursachen. Dies ist als wichtiges Ergebnis des Backsteinshaders aus Kapitel 4.2 hervorgegangen. Ist der Sprung des Höhenwerts benachbarter Texel zu hoch, kommt es zu Fehlabtastungen. Das bedeutet im Umkehrschluss aber auch, dass alle Oberflächen, die mittels Heightmap-Tracing gerendert werden, keine harten Kanten aufweisen dürfen, sofern sie fehlerfrei dargestellt werden sollen.

Die Generierung der Oberflächen ohne eine Basistextur ist allerdings auch problematisch. Begrenzungen der zur Verfügung stehenden Hardware machen vielen auf der CPU problemlos implementierbaren Verfahren „einen Strich durch die Rechnung“. Das Kapitel 4.4 beschreibt dennoch eine Möglichkeit, den DiamondSquare-Algorithmus auf der GPU auszuführen. Die Limitierung des Multipass-Verfahrens und die komplizierte Einsetzbarkeit stellen allerdings Hürden dar. Die DSPatchs werten das erzeugte Bild auf, in dem sie für jede Pixel eine berechnen. Die (notwendige) Glattheit der Heightmaps kann so nach Schnittberechnung noch eine Rauheit erzeugen – gewissermaßen eine Bumpmap für die Bumpmap.

Als weiteres Einsatzgebiet der Patchs wurde das SPS-Verfahren entwickelt, das in Kapitel 4.5 vorgestellt wurde. Mit ihm werden pro Pixel mehrere Normale erzeugt, die eine Oberfläche innerhalb des Pixels imitieren. Da die Struktur jedes SPS-Patchs (individuell) kontrollierbar ist, sind mit ihnen anisotrope Materialien darstellbar. Das komplizierte Reflektanzverhalten ergibt sich so von selbst durch Einbeziehung aller Normalen in das weitere Shading. Das Beispiel der gebürsteten Metalle zeigt, dass das Verfahren realistische Ergebnisse liefert. Sowohl mit regulärer Beleuchtung als auch mit dem Environmentmapping verbunden, überzeugen die generierten Bilder. Mit einer Steuerung der SPS-Patch-Parameter sind auch gemusterte

anisotrope Materialien darstellbar. Ein einfaches Beispiel – eine Sinuskurve – ist im Kapitel 4.5 gezeigt worden. Denkbar ist eine Kombination mit anderen Verfahren, wie etwa den in [LB05], [RNCL05] oder [Gus06] veröffentlichten Methoden, vektoriell beschriebene Flächen darzustellen. So könnten auch komplexere Muster realisiert werden.

Als Alternative zum Bumpmapping hat sich eine andere Art von Verfahren gefunden, ein Bumpfunktion-Verfahren. Es bezieht seine Oberflächenstruktur nicht aus einer Textur, sondern aus einer Berechnung, die beliebig genau ausgewertet werden kann. Ein weiterer Vorteil ist die Geschwindigkeit der Methoden: Texturzugriffe sind teuer und GPUs werden immer leistungsfähiger. So ist eine komplexe Formel schneller ausgewertet, als dass die notwendigen Texturzugriffe erfolgt sind. Die Bumpfunktion-Methoden sind in zwei Gruppen eingeteilt: zweidimensionale und dreidimensionale Oberflächenbeschreibungen.

Das 2D-BumpPattern-Verfahren benutzt zweidimensionale implizite und polynomielle Funktionen, um die Oberfläche nachzubilden. Jedes so beschriebene Ornament ist von außen steuerbar und kann ein individuelles Shading erhalten. Zwischen regulärer Beleuchtungsrechnung und Environmentmapping gibt es keine Beschränkung. Bei regelmäßigen, einfachen Mustern ist das Ergebnis glaubwürdig.

Universeller ist das dreidimensionale Raytraced–BumpPattern-Verfahren. Die Oberfläche wird mit dreidimensionalen Objekten beschrieben. Auch hier sind die Parameter extern oder im Shader selbst kontrollierbar. Die Benutzung eines Raytracers bedeutet auch, dass die Objekte frei im Raum platziert werden und sich beliebig schneiden können. Dadurch sind auch alle von Heightmap-Tracing-Verfahren bekannten Effekte (Selbstschattierung, Selbstverdeckung und Pseudo-Sihouettenbildung) ohne umfangreiche Erweiterungen realisierbar.

Kapitel 5

Bewertung und Ausblick

„It is far better to foresee even without certainty than not to foresee at all."

Henri Poincare

Das letzte Kapitel bewertet die entwickelten Verfahren und beleuchtet deren Möglichkeiten der Weiterentwicklung und liefert somit einen Ausblick auf zukünftige Anwendungen.

5.1 Bewertung

Innerhalb der Arbeit wurden verschiedenen Methoden, die das Bumpmapping-Verfahren weiterentwickeln, vorgestellt.

Das wichtigste Ergebnis ist das Raytraced-BumpPattern-Verfahren. Eine Oberfläche wird von mehreren Objekten beschrieben, die per Raytracing dargestellt wird. Es stellt einen Zwischenschritt auf dem Weg von polygonalen Renderern zu Raytracern dar: Anstatt einer kompletten Szene kann so die Oberfläche einzelner Objekte mit einem Raytracer im Shader gerendert werden.

Das größte Problem ist, dass das Verfahren auf prozeduralen Methoden basiert. Es wertet implizite und polynomielle dreidimensionale Funktionen aus. Diese Funktionen müssen aber erst erstellt werden. Da es sich um rein mathematische Formeln handelt, wendet es sich eher an Programmierer als an Modellierer. Stellt man sich komplexere Muster, wie etwa einen Teppich oder Stoffmuster vor, wird schnell klar, dass es schwierig wird, eine geeignete Funktion zu finden. Will man ein Holzobjekt darstellen, ist es einfacher, ein Foto von einem Stück Holz zu machen und es als Textur einzubinden, als sich erst mit prozeduralen Generierung des gewünschten Holzmaterials auseinanderzusetzen. Hinzu kommt, dass nicht nur die Erzeugung, sondern auch der Umgang mit Texturen für Designer und Modellierer wesentlich einfacher ist.

Hier müssen noch geeignete Schnittstellen geschaffen werden, die die Methode leichter und handhabbar macht. Ein erster Schritt wurde schon gemacht: der Shader kann mit verschiedenen 3D-Körpern umgehen, die auf der Oberfläche angeordnet werden können. Hier gibt es noch deutliches Potential für Verbesserungen.

Andererseits wurde das Verfahren auch nicht geschaffen, um Texturen abzulösen. Das Verfahren bietet sich eher als Ergänzung für texturbasierte Verfahren an: Bestimmte Oberflächen einer Szene lassen sich besser mit Texturen, andere mit Raytraced-BumpPattern darstellen.

Ein zweites Themengebiet der eigenen Verfahren ist die prozedurale Erstellung von landschaftsähnlichen, fraktalen Oberflächenstrukturen. Die Ergebnisse sind allerdings gemischt.

Auf der einen Seite sind komplexe Strukturen nur schwer erstellbar. Aufgrund der Limitierung von Shadern ist die Größe der erstellten Oberflächenstücke stark begrenzt. Dennoch hat sich ein Weg gefunden, diese zu nutzen: als Anrauung der oft glatt wirkenden Heightmap-Texturen. Sie werden gewissermaßen als Bumpmap für Bumpmaps eingesetzt. Allerdings müssen sie kontrolliert eingesetzt werden, da sie die Probleme von Heightmap-Tracing-Verfahren verstärken.

Auf der anderen Seite überzeugen die Patch genannten Höhenfelder, wenn man sie im kleinen einsetzt: wertet man pro Pixel ein Höhenfeld mit mehreren Normalen aus, lassen sich anisotrope Beleuchtungseffekte imitieren. Dieses SubPixelSurface Lighting genannte Verfahren funktioniert für reguläre Beleuchtungsrechnung und für Environmentmapping gut. Da die Patchs frei ausrichtbar sind, lässt sich so das Reflektanzverhalten von z.B. gebürsteten Metallen nachahmen. Die Auswertung eines kompletten Patchs ist recht aufwändig und im Falle des Environmentmappings nur stark eingeschränkt möglich. Allerdings ist dies eine Frage der Rechenleistung, die in den letzten Jahr(zehnten) stetig zugenommen hat.

5.2 Die Zukunft des Bumpmappings

Das Bumpmapping-Verfahren hat sich in den letzten Jahren deutlich weiterentwickelt. Insbesondere die Heightmap-Tracing-Verfahren haben wesentlich zur Steigerung des Realitätsgrads beigetragen. Allerdings sind die Verfahren limitiert:

- Es wird immer Abtastartefakte geben.
- Die Auflösung der Texturen wird sich erhöhen, aber dennoch nie groß genug sein, um alle Details erfassen zu können.
- Mehr als eine „Pseudosilhouette“ kann nicht erzeugt werden.

5.2.1 GeometryShader

Mit der Einführung von DirectX 10 wurde ein neuer Shadertyp eingeführt, der so genannte GeometryShader [Bly06]. Dieser sitzt in der API-Pipeline nach dem Vertexshader und kann neue Vertices erzeugen. Da er auch auf Texturen zugreift, kann so die Heightmap direkt in den GeometryShader geladen und dort die entsprechenden Vertices generiert werden. Das Verfahren entspricht dann dem DisplacementMapping. Da die Körpergeometrie somit verändert wird, ist es wahrscheinlich, dass es das Heightmap-Tracing ablösen wird.

Dadurch wird dem Bumpmapping die Möglichkeit gegeben, sich weg von einem „Universalverfahren“ hin zu seiner eigentlichen Idee – dem realistischen Darstellen einer Oberflächenstruktur – zurück zu bewegen. Eine solche Methode ist selbst in Raytracern von Nöten, die aufgrund steigender Hardwareleistungen [Wal05] oder dedizierter Hardware [WSS05] immer schneller werden und Echtzeitfähigkeit damit immer besser realisierbar wird.

5.3 Weiterentwicklung der eigenen Verfahren

Die gezeigten Methoden liefern viel versprechende Ansätze. Hervorzuheben sind das SubPixelSurface-Verfahren und die BumpPattern. Mit ihnen können brauchbare Ergebnisse erzielt werden. Insbesondere die BumpPattern bieten jedoch noch Raum für Verbesserungen.

5.3.1 Fraktale Landschaften

Der kritische Faktor beim Erzeugen fraktaler Landschaften pro Pixel im Shader war das Zusammenhangs- bzw. Gedächtnisproblem. Da künftige Chips eine ähnlich hohe Ausgabeleistung durch die parallele Ausführung der Pixelshader erzielen werden, ist nicht davon auszugehen, dass der Algorithmus wesentlich besser implementiert werden kann. Allerdings könnte mit steigender Anzahl erlaubter Texturzugriffe die Anzahl der DiamondSquare-Schritte erhöht werden.

Daraus folgt, dass andere Ideen zur Erzeugung von Landschaftsstrukturen herangezogen werden müssen. Fraktale Methoden sind nicht geeignet, da auf die durch die Rekursivität fraktaler Algorithmen benötigten Zwischenwerte nicht zugegriffen werden kann.

5.3.2 SubPixelSurface

Von der steigenden Hardwareleistung profitiert wohl das SubPixelSurface Verfahren am meisten. Größere Patchs erlauben eine feinere Mikrostruktur, die noch deutlichere Effekte erlaubt. Zudem könnte mit leistungsfähigeren Shadern auch das SPS-EM voll implementiert werden, bei dem mehre Cubemap-Zugriffe durchgeführt werden.

Denkbar ist auch die Verbindung des Verfahrens mit verschiedenen Phänomenen der optischen Physik: Refraktion und Interferenz sind Effekte, die zur Steigerung der optischen Qualität viel beitragen können.

Ein weiterer Aspekt, der noch näher betrachtet werden kann, ist die Steuerbarkeit des Verfahrens mit prozeduralen Mitteln. Sehr interessant ist auch die Frage, wie weit das Verfahren mit Methoden des BumpPattern-Verfahrens kombinierbar ist.

5.3.3 BumpPattern

Das Raytraced-BumpPattern birgt noch viel Erweiterungspotential in sich. Die Frage, welche Möglichkeiten zur Beschreibung von Pattern umsetzbar sind, ist praktisch noch unbeantwortet. Innerhalb der Arbeit wurde das Problem nur für einfache Materialbeschreibungen in Form von Objektanordnung beantwortet. Mit Zunahme der Beschreibungsmöglichkeit steigt auch der Nutzen der Methode. Durch Steigerung der Hardwareleistungsfähigkeit werden Shader flexibler und schneller in ihrer Ausführung, so dass sogar sehr mächtige Werkzeuge herangezogen werden können. Denkbar ist etwa einen kompletten Erzeugungsmechanismus voranzustellen. L-Systeme oder andere Grammatiken sind hier Optionen, die in Betracht gezogen werden könnten.

Literaturverzeichnis

[App68] Appel, Arthur: Some Techniques for Shading Mashine Renderings of Solids. In: *Spring Joint Computer Conference* (1968).

[Bal02] Baldwin, David, OpenGL 2.0 Shading Language White Paper, Version 1.2. In: *3Dlabs, http://www.3dlabs.com/support/developer/ogl2*, 2002.

[Bli78] Blinn, James F.: Simulation of Wrinkled Surfaces. In: *Proc of Annu Conf on Comput Graph and Interact Tech (SIGGRAPH '78), 5th*, Seite 286-292, 1978.

[Bly06] Blythe, David: The Direct3D 10 System. In: *Proceedings of the ACM SIGGRAPH 2006*, Seite 724-734, 2006.

[BN76] Blinn, James; Newell, Martin: Texture and Reflection in Computer Generated Images. In: *Communications of the ACM* (1976).

[Bre06] Breiner, Tobias: Dreidimensionale virtuelle Organismen (Dissertation). Johann-Wolfgang-Goethe-Universität Frankfurt, 2006.

[BT04] Brawley, Zoe; Tatarchuk, Natalya: Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapppi. In: *Shader X3*, Seite 135-154, 2004.

[Cat74] Catmull, Edwin. E., A Subdivision Algorithm for Computer Display of Curved Surfaces (Phd-Thesis). Dept. of CS, U. of Utah, 1974

[CD03] Chan, E.; Durand, F.: Rendering fake soft shadows with smoothies. In: *Eurographics Symposium in Rendering Proceedings*, Seite 208-218, 2003.

[CHCH06] Carr, Nathan; Hoberock, Jared; Crane, Keenan; Hart, John: Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. In: *To appear in Proceedings of Graphics Interface*, Seite , 2006.

[CHH02] Carr, Nathan; Hall, Jesse; Hart, John: The Ray Engine. In: *Graphics Hardware*, Seite 1-10, 2002.

[Chr05] Christen, Martin: Ray Tracing on GPU (Diplomarbeit). University of Applied Sciences Basel (FHBB), 2005.

[Coo84] Cook, Robert L.: Shade Trees. In: *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, Seite 223-231, 1984.

[FDHD90] Foley, James; vanDam, Andries; Feiner, Steven; Hughes, John:

Computer Graphics - Principles and Pratices. Addison Wesley, 1990. - ISBN: 0-201-84840-6

[FFC82] Fournier, Alain; Fussel, Donald; Carpenter, Loren: Computer Rendering of Stochastic Models. In: *Commun. ACM*, 1982.

[Fly95] Flynn, Michael: Computer Architecture: Pipelined and Parallel Processor Design. Jones and Bartlett Publishers, Inc., 1995. - ISBN: 086720204137

[FS05] Foley, Tim ; Sugerman, Jeremy: KD-Tree Acceleration Structures for a GPU Raytracer. In: *Proceedings of the ACM SIGGRAPH/Eurographics*, Seite 15-22, 2005.

[Gus06] Gustavson, Stefan, Beyond the pixel: towards infinite resolution textures. In: *Linköping University, ITN , Internal Report*, 2006.

[Har01] Hart, John C.: Perlin noise pixel shaders. In: *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, Seite 87-94, 2001.

[Hor00] Horne, Dave, DirectX 8 GraphicsOverview. In: *NVidia Tech Report*, 2000.

[Kil00] Kilgard, Mark, A practical and robust bump-mapping technique for today's gpu's. In: *GDC 2000 / NVIDIA*, 2000.

[LB05] Loop, Charles; Blinn, James: Resolution Independent Curve Rendering Using Programmable Graphics Hardware. In: *Siggraph Proceedings*, Seite 1000-1010, 2005.

[Mar06] Martz, P., Generating Random Fractal Terrain. In: *http://www.gameprogrammer.com/fractal.html, Zugriff: 12.11.2006, 12:34MEZ*, 2006.

[MM05] McGuire, Morgan; McGuire, Max, Steep parallax mapping. In: *I3D 2005 Poster*, 2005.

[OBM00] Oliveira, Manuel; Bishop, Gary; McAllister,David: Relief texture mapping.. In: *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, Seite 359-368, 2000.

[Per02] Perlin, Ken: Improving noise. In: *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, Seite 681-682, 2002.

[Per85] Perlin, Ken: An image synthesizer. In: *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, Seite 287-296, 1985.

[Pho75] Phong, Bui T.: Illumination for computer generated pictures. In:

Commun. ACM, 1975.

[Pix88] Pixar, RenderMan Interface Specification, V. 3.0. In: *https://renderman.pixar.com/products/rispec/index.htm*, 1988.

[PO05] Policarpo, Fabio; Oliveira, Manuel, An Efficent Representation for Surface Details. In: *UFRGS Technical Report*, 2005.

[PO06] Policarpo, Fabio; Oliveira, Manuel: Relief mapping of non- height-field surface details. In: *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, Seite 55-62, 2006.

[POC05] Policarpo, Fabio; Oliveira, Manuel; Comba, João: Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. In: *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, Seite 155-162, 2005.

[RNCL05] Ray, Nicolas; Neiger, Thibaut; Cavin, Xavier; Levy, Bruno, Vector Texture Maps. In: *Tech Report*, 2005.

[Ros04] Rost, Randi: OpenGL(R) Shading Language. Addison Wesley Longman Publishing Co., Inc., 2004. - ISBN: 0321197895

[Sch94] Schlag, John: Fast embossing effects on raster image data. In: *Graphics gems IV*, Seite 433-437, 1994.

[Tar06] Tatarchuk, Natalya: Dynamic parallax occlusion mapping with approximate soft shadows. In: *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, Seite 63-69, 2006.

[VF94] Voorhies, Douglas; Foran, Jim: Reflection vector shading hardware. In: *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, Seite 163-166, 1994.

[Wal05] Wald, Ingo: The OpenRT-API. In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, Seite 21, 2005.

[Wat02] Watt, Alan: 3D-Computergrafik. Pearson Studium, 2002. - ISBN: 3827370140

[WDS99] Woo, Mason; Davis, Tom; Shreiner, Dave: OpenGL Program- ming Guide: The Official Guide to Learning OpenGL, V. 1.2. Addison-Wesley Longman Publishing Co., Inc., 1999. - ISBN: 0-201-60458-2

[Whi80] Whitted, Turner: An improved illumination model for shaded display. In: *Communications ACM* (1980).

[WSS05] Woop, Sven; Schmittler, Jörg; Slusallek, Philipp: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In: *Proceedings of ACM SIGGRAPH 2005*, Seite 434-444, 2005.