FACHHOCHSCHULE **TRIER**
Hochschule für Technik, Wirtschaft und Gestaltung
University of Applied Sciences
Informatik

Displacement Mapping in DirectX 11

Displacement mapping with DirectX 11

Daniel Dewald

Bachelor Abschlussarbeit

Betreuer: Prof. Dr.-Ing. Christoph Lürig

Trier, 02.04.2011

Danksagung

Ich möchte folgenden Personen für ihre Unterstützung beim Erstellen dieser Ausarbeitung danken:

- Meiner Freundin Lea Kaufmann für das Korrekturlesen, den seelischen Beistand und nicht zuletzt die Rücksichtnahme während der Zeit meiner Arbeit.

- Meiner Familie, ohne deren Rückhalt diese Arbeit nie entstanden wäre.

- Prof. Dr.-Ing. Christoph Lürig für die Betreuung dieser Arbeit.

Kurzfassung

In der modernen Computergrafik spielt die realistisch wirkende Wiedergabe komplexer Objekte eine große Rolle. Displacement Mapping erlaubt eine wesentlich verbesserte Darstellung solcher Objekte. Im folgenden Dokument soll diese Rendertechnik genauer untersucht, sowie ihre Stärken und Schwächen aufgezeigt werden. Es wird gezeigt, wie sich mit DirectX 11 Funktionen der Großteil der Schwächen beseitigen lässt und das Echtzeit-Rendering von großen detaillierten Flächen möglich wird. Außerdem wird anhand eines praxisnahen Beispiels erläutert, wie sich mit weiteren Direct X 11 Funktionen die Praxistauglichkeit von Displacement Mapping weiter erhöhen lässt. Abschließend wird eine Gesamtlösung präsentiert, welche die erläuterten Rendertechniken zur Simulation einer Insel in einer Wasserlandschaft zusammenfasst.

In modern computer graphics the realistic looking rendering of complex objects is very important. Displacement mapping allows an drastically improved rendering of such objects. In the following document this render technique will be analyzed and its positive and negative spects disclosed. It will be shown how using DirectX 11 can eliminate most of the negative aspects and real time rendering of big detailed surfaces is made possible. Furthermore using a practical example it will be shown how other DirectX 11 features further increase the suitability for daily use of displacement mapping. Finaly a complete solution for rendering an island in a waterscenery using the introduced techniques will be presented.

Inhaltsverzeichnis

Abbildungsverzeichnis

1

Einleitung

Displacement Mapping selbst ist keine neue Technik. Sie wurde bereits 1982 bei Special Effects in Filmen verwendet. Die Technik selbst ist relativ simpel: Für jeden Punkt eines zu verformenden Polygongitters wird eine Graustufentextur, welche als Höhenkarte dient, mit ihren entsprechenden Texturkoordinaten abgefragt. Danach wird der Punkt einfach nach dem Wert der abgefragten Texturfarbe entlang seiner Normalen verschoben.

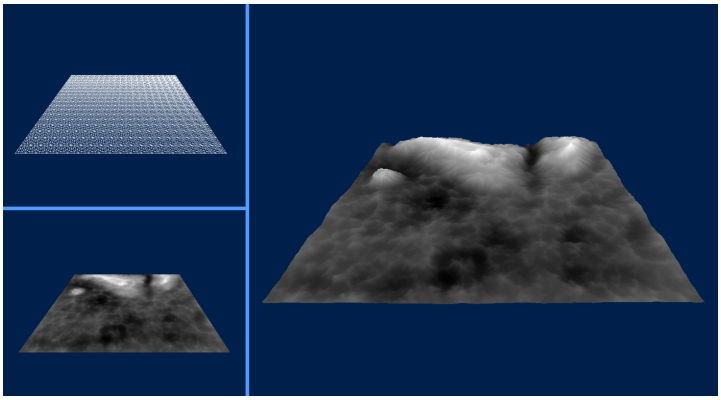

Abbildung 1.1. Displacment Mapping

Die größte Stärke von Displacement Mapping, das Verformen dichter Polygonnetze, ist jedoch auch die größte Herausforderung bei dessen Umsetzung in Echtzeit-Rendering-Umgebungen. Aus diesem Grund wird es bisher hauptsächlich bei Softwaresystemen eingesetzt, welche nicht in Echtzeit rendern oder bei denen die zu erreichende Render-Geschwindigkeit eine untergeordnete Rolle spielt. Desweiteren wird häufig zusätzlich auf LOD-Techniken zurückgegriffen, welche die hohen Polygonzahlen auf jene Ober-

flächenbereiche beschränken, die gerade im direkten Blickfeld und in der unmittelbaren Nähe des Betrachters liegen. Im optimalen Fall ließe sich jeder Pixel einer Höhenkarte auf einen Punkt im Polygongitter abbilden. Damit hätte man die höchste Detailstufe mit der vorhandenen Höhenkarte erreicht. Um solch dichte Polygongitter zu erhalten, ohne tausende von Polygonen pro Objekt vorzuhalten, wird in der Regel Tessellation[1] verwendet.

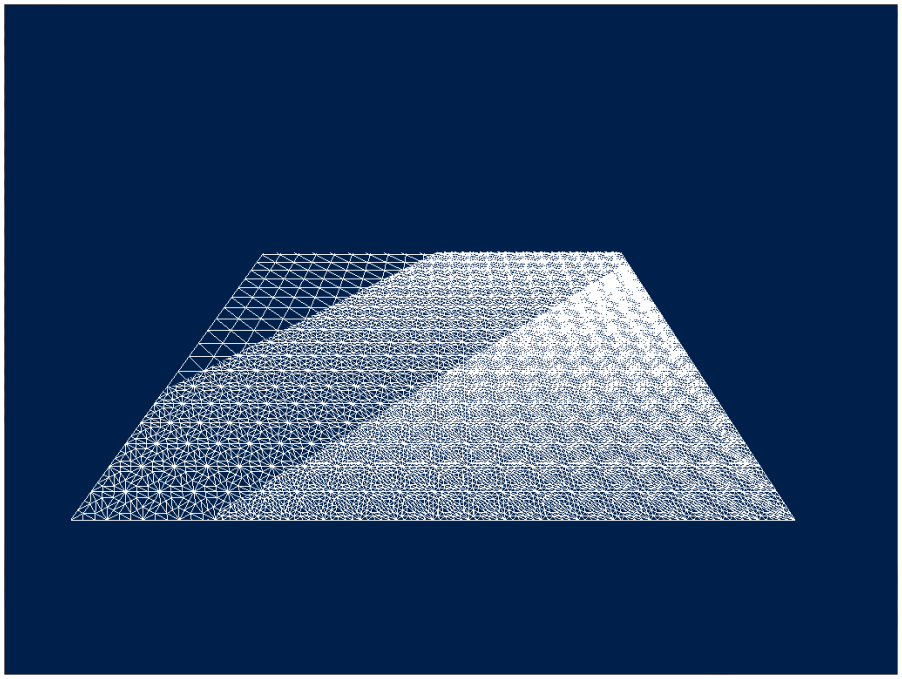

Abbildung 1.2. Tessellation

Bisher nur als Vorstufe zum eigentlichen Render-Vorgang in Software möglich, bieten alle Direct X 11 GPU's diese Technik nun als vollintegierte Stufe in der Renderpipeline an. Dies ermöglicht das Senden von Low-Polygon-Objekten an die GPU, welche von dieser ohne nennenswerten Geschwindigkeitsverlust und hardwarebeschleunigt soweit tesselliert werden, bis ein hochdetailliertes Displacement möglich ist. Hierdurch wird eines der problematischsten Hindernisse bezüglich des Einsatzes von Displacement Mapping beseitigt.

[1] Zerteilen eines Polygons in mehrere Teilpolygone

2

Problemstellung

Als Alternativen zum Displacement Mapping dienten bisher im Wesentlichen Bump Mapping, Normal Mapping und Parallax Occlusion Mapping, bei denen dem Betrachter nur eine komplexe Geometrie vorgespielt wird. Bei flachen Betrachtungswinkeln und genauer Inspektion des Objektes fällt die Täuschung jedoch relativ schnell auf. Bei den genannten Verfahren sind zudem einige modernere Rendertechniken[1] nicht oder nur schwer möglich, da sich die Geometrie des Objektes durch diese Techniken nicht verändert, sondern nur ihre Wahrnehmung im Licht.

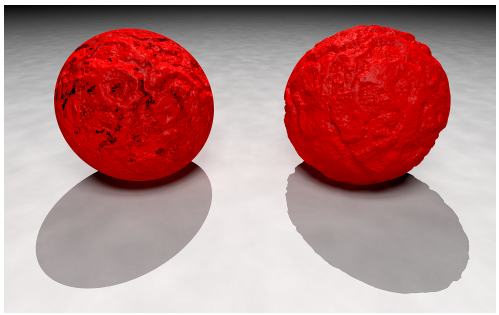

Abbildung 2.1. Bump Mapping / Displacement Mapping (Quelle: Wikipedia)

Um Displacement Mapping in Echtzeit-Systemen verwenden zu können, ohne den vorhandenen GPU Bus und Speicher auszulasten, ist es daher nötig, eine Low-Polygon-Oberfläche in der GPU in eine High-Polygon-Oberfläche umwandeln zu können. Hierzu kann Hardware-Tessellation verwendet werden, welches es erlaubt, ein Dreieck in der GPU in eine beliebige Menge kleinerer Teildreicke aufzuspalten, ohne dabei GPU Bandbreite oder Speicher zu ver-

[1] z.B. realistischer Schattenwurf

schwenden. Zusätzlich steht dem Entwickler seit DirectX 11 eine generalisierte Schnittstelle zur Berechnung komplexer Formeln in der GPU[2] zur Verfügung: Der Compute Shader.

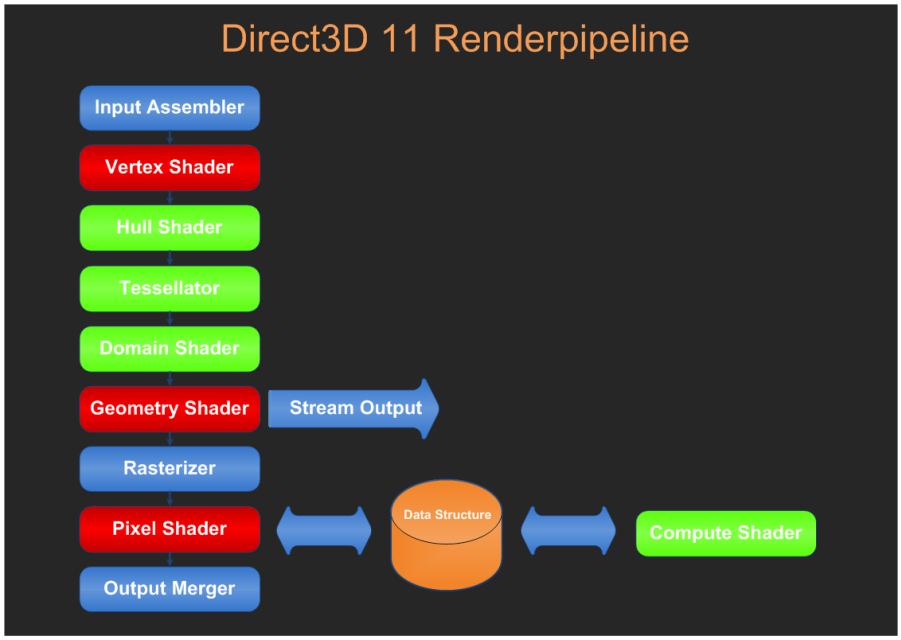

Abbildung 2.2. DirectX 11 Render Pipeline

Mithilfe von Compute Shadern ist es möglich, mit wenigen Daten komplexe Objektstrukturen in der GPU zu berechnen. Die hier erzeugten Daten lassen sich verwenden, um ein Low-Polygon-Objekt mit Tessellation und Displacement Mapping in der GPU in ein hochdetailliertes dynamisch veränderbares Objekt umzuwandeln. Hierdurch wird die CPU erheblich entlastet. Dies schafft Freiraum für Rendertechniken, welche bisher als zu CPU hungrig angesehen wurden.

[2] GPGPU

3

Aufgabenstellung und Zielsetzung

Zunächst soll eine Analyse von Displacement Mapping erfolgen, um die Vor- bzw. Nachteile der Technik zu klären und diese in den korrekten Kontext zur eigentlichen Arbeit zu bringen. Ziel der Analyse sollte es sein, jene Punkte zu finden, welche sich mithilfe der in den Kapiteln 1 und 2 beschriebenen Techniken[1] aus der DirectX 11 Pipeline verbessern bzw. eliminieren lassen. Hierzu sollen ebenfalls die benannten Techniken genauer untersucht und beschrieben werden. Zusätzlich soll untersucht werden, welcher gängige Algorithmus zur realistischen Simulation von Wasseroberflächen zur Umsetzung im Compute Shader geeignet ist. Auf diesen Analysen aufbauend soll eine Insel in einer dynamisch generierten Wasserumgebung in einer DirectX 11 3D-Anwendung gerendert werden. Ziel ist ein möglichst hoher Detailgrad bei beiden Oberflächen[2]. Hierzu sollen große Teile der Berechnungen in der GPU durchgeführt werden. Der gesamte Render-Vorgang muss in Echtzeit ablaufen[3]. Optional kann ein Pixel Shader für eine realistischere Land- und Wasserdarstellung entworfen und implementiert werden.

[1] Hardware Tessellation und Compute Shader
[2] Land und Wasser
[3] real-time rendering

4

Displacement Mapping

Die grundlegende Funktionsweise von Displacement Mapping wurde in Kapitel 1 bereits erläutert. Hier soll diese Funktionsbeschreibung vertieft werden und auf die Vor- und Nachteile der Methode genauer eingegangen werden.

4.1 Theorie

Wie jede andere Technik kommt auch Displacement Mapping nicht ohne theoretische Überlegungen aus. Sei $\vec{s}_1(x, y)$ ein Punkt der zu verformenden Oberfläche S_1 an der Stelle (x, y) und $\vec{n}(x, y)$ die Normale an dieser Stelle. Sei außerdem H eine Höhenkarte, welche jedem Punkt $\vec{s}_1(x, y) \in S_1$ einen eindeutigen Höhenwert $h \in \mathbb{R}^+$ zuordnet. Dann erhält man die verformte Oberfläche S_2, indem man für jedes $\vec{s}_1(x, y) \in S_1$ folgende Formel berechnet:

$$\vec{s}_2(x, y) = \vec{s}_1(x, y) + \vec{n}(x, y) * h(x, y)$$

Da die Höhenwerte in der Praxis üblicherweise aus einer Textur ausgelesen werden, liegen die Zahlenwerte hier in der Regel im Intervall $[0.0, 1.0]$. Um eine nennenswerte Verformung erreichen zu können, wird der Höhenwert daher in der Praxis mit einem statischen Verformungsfaktor $v \in \mathbb{R}$ multipliziert:

$$\vec{s}_2(x, y) = \vec{s}_1(x, y) + \vec{n}(x, y) * (h(x, y) * v)$$

Durch ein negatives v lassen sich Dellen in der S_1 Oberfläche erzeugen, durch ein positives v entstehen entsprechend Erhebungen. Der Verformungsfaktor kann sogar je nach Punkt $\vec{s}_1(x, y) \in S_1$

unterschiedlich groß sein bzw. unterschiedliche Vorzeichen aufweisen. Er erlaubt somit eine individuelle Anpassung der Verformung an die Gegebenheiten der Aufgabe.

Es ist wichtig anzumerken, dass durch die Verformung der Oberfläche deren Normalen ebenfalls verändert werden müssen. Anderfalls würde sich die unverformte, statt der verformten Oberfläche, in weiteren Berechnungen, welche die Normale verwenden, wiederspiegeln und z.B. eine unnatürlich wirkende Ausleuchtung der Oberfläche verursachen. Die Berechnung der Normalen kann bei statischer Höhenkarte bereits vor der Verformung berechnet und in einer Normal Map gespeichert werden. Sie muss dann lediglich für jeden Punkt der Oberfläche über eine Texturabfrage abgerufen werden. Bei dynamischer Höhenkarte müssen die Normalen pro Punkt aus den Werten der aktuellen Höhenkarte errechnet werden. Hierzu wird das Kreuzprodukt zwischen zwei Vektoren des aktuellen Punktes und seinen Nachbarpunkten genommen und normalisiert.

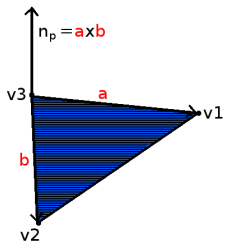

Abbildung 4.1. Normale durch Kreuzprodukt

Die beiden Vektoren $\vec{a}$ und $\vec{b}$ erhält man durch Subtraktion des Punktvektors v_1 von v_2 und v_3. Daraus ergibt sich folgende Formel zur Berechnung des Vektors $\vec{n}_{pu}$ eines Patches:

$$\vec{n}_{pu} = \begin{pmatrix} (v_{2y} - v_{1y})(v_{3z} - v_{1z}) - (v_{2z} - v_{1z})(v_{3y} - v_{1y}) \\ (v_{2z} - v_{1z})(v_{3x} - v_{1x}) - (v_{2x} - v_{1x})(v_{3z} - v_{1z}) \\ (v_{2x} - v_{1x})(v_{3y} - v_{1y}) - (v_{2y} - v_{1y})(v_{3x} - v_{1x}) \end{pmatrix}$$

Abschließend erhält man den Normalenvektor $\vec{n}_p$ des Patches durch Normalisieren des vorher berechneten Vektors $\vec{n}_{pu}$:

$$\vec{n}_p = \frac{1}{|\vec{n}_{pu}|} * \vec{n}_{pu}$$

4.2 Vorteile

Nach dem Verformen der Oberflächen steht ein detailliertes Objekt in der restlichen Renderpipeline zur Verfügung. Gegensätzlich zu anderen Verfahren, welche auf Tricks zurückgreifen, um nicht detaillierten Objekten ein detailliertes Aussehen zu verleihen, können so formabhängige Berechnungen direkt an Objektdaten durchgeführt werden. Komplizierte Schattenberechnungen benötigen beispielsweise keine kostspieligen Vorberechnungen, um Fehler, welche durch falsche Geometrie des Objektes entstehen würden, auszugleichen. Ebenfalls fallen Tiefeneffekte, welche bei anderen Techniken in flachen Betrachtungswinkeln unschön bemerkbar sind, hier nicht ins Gewicht, da das Objekt echte Tiefe besitzt. Ein nicht unwesentlicher Faktor ist außerdem, dass alle Techniken, welche keine Objektverformung vornehmen, nicht mit Virtual Reality Systemen kompatibel sind. Würde man eine Oberfläche, auf welche Bump Mapping angewendet wurde, mit einer 3D Brille auf einem kompatiblen Monitor betrachten, so würde diese direkt unangenehm ins Auge stechen. Oberflächen, die bislang Bump Mapping oder Normal Mapping verwendet haben, lassen sich mit Displacement Mapping wesentlich realistischer darstellen. Außerdem ist es möglich, durch Displacement Mapping ein Basismodell[1] mehrfach, aber mit unterschiedlichem Aussehen, zu verwenden. Das Modell selbst bleibt hierbei gleich. Es werden lediglich die angewandten Displacement Maps und die aufgebrachten Texturen verändert. So könnten z.B. zwei optisch vollkommen unterschiedliche Charaktere in einem Spiel dasselbe Basismodell verwenden, welches dann unterschiedlich verformt wird. Eine ähnliche Technik wird häufig herangezogen, wenn aus einfachen Höhenkarten Landschaften generiert werden. Die Entwickler schreiben hierzu zunächst einen Algorithmus zur Erzeugung eines flachen Polygongitters. Die eigentliche Landschaft wird dann durch die jeweils gewählte Displacement Map und die damit verbundenen Texturen bestimmt. Modelle und Landschaften lassen sich mithilfe von Displacement Mapping einfacher und schneller erstellen, da für die meisten Strukturarten bereits Höhenkarten zu finden sind. Zudem gibt es für real existierende Landschaften Höhenkarten günstig oder kostenfrei im Internet[2].

[1] z.B. Das Modell einer Person
[2] z.B. ETOPO unter [ETO]

4.3 Nachteile

In Kapitel 1 wurde bereits angesprochen, dass Ergebnisse mit Displacement Mapping sehr stark von der Dichte des zugrundeliegenden Polygonnetzes abhängig sind. Ein dichtes Polygonnetz in einem Objekt gewährleistet ein wesentlich detailreicher verformtes Objekt. Leider ist die nötige Dichte, vor allem bei dynamischen Höhenkarten, bei der Erstellung der Oberflächen nicht immer abzusehen. Es gilt daher, den goldenen Mittelweg zu finden oder die Polygondichte von Oberflächen dynamisch an die Gegebenheiten anzupassen. Dies erfordert entweder zusätzlichen Rechenaufwand oder verschwendet GPU Bandbreite[3]. Möchte man dynamische Displacement Maps verwenden, so ist es nötig, bei jedem Frame Daten in eine Textur zu laden. Das Locking[4] einer Textur serialisiert allerdings den Zugriff auf diese zwischen CPU und GPU. Folglich kann der jeweils schnellste Teil (GPU oder CPU) vom langsameren ausgebremst werden. Operationen, welche pro Punkt ausgeführt werden, sind auf verformten Oberflächen in der Regel erheblich aufwendiger, da die Anzahl der Punkte stark gesteigert werden muss. Vertex Shader sollten bei dichten Polygongittern also simpler gehalten werden. Komplexe Operationen müssen deswegen entweder in die CPU ausgelagert werden oder man muss ganz darauf verzichten. Desweiteren müssen bei dynamischen Höhenwerten nach dem Displacement für jeden Patch die Normalenvektoren neu berechnet werden. Geschieht dies in der CPU, kostet es hier Leistung. Bei Berechnung in der GPU[5] müssen zusätzlich zu jedem Punkt die Daten von dessen Nachbarn im Patch bereitgestellt werden. Zuletzt sei noch erwähnt, dass Displacement Mapping das Texture Mapping erschweren kann, da durch die Verformung des Objektes teilweise andere Texturen nötig werden, um ein realistisches Aussehen des Objektes nicht zu trüben.

Im folgenden Kapitel 5 wird darauf eingegangen, wie sich die meisten der hier genannten Nachteile von Displacement Mapping mithilfe von neuen DirectX 11 Funktionen beseitigen lassen.

[3] Sollten Objekte dichter sein als eigentlich nötig
[4] siehe [Micf]
[5] In der Regel im Vertex Shader

5

DirectX 11 Funktionen

Veröffentlicht am 22. Oktober 2009, wird die DirectX 11 API die Basis zukünftiger High-End-Spieletitel bilden. Die wesentlichen Änderungen zur Vorgängerversion 10.1 sind:

- Shader Modell 5.0 mit 3 neuen Shader Typen
  - Hull Shader (Per-Patch)
  - Domain Shader (Per-Control Point)
  - Compute Shader (vereinheitlichte GPGPU Programmierung)
- Hardware Tessellation
- Optimierung der Renderpipeline auf Threads

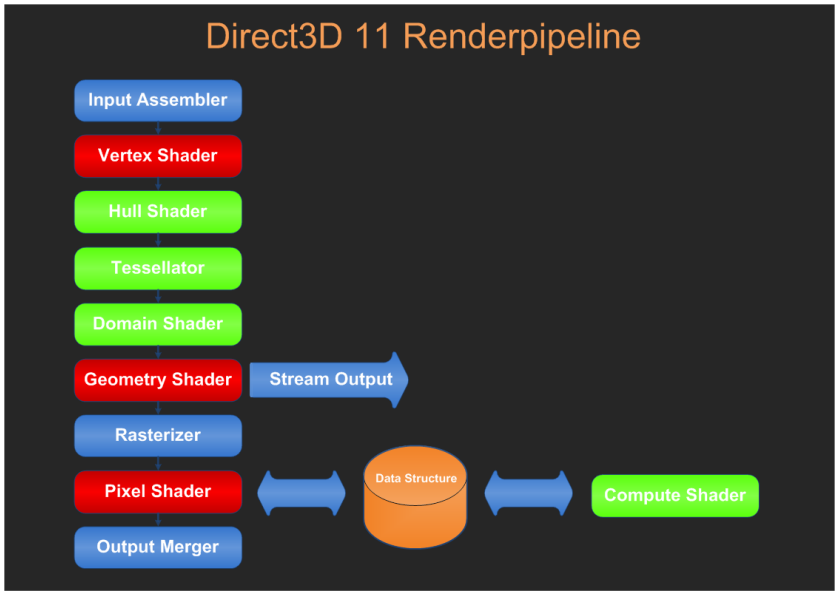

Abbildung 5.1. Direct X 11 Render Pipeline

Wie man an der Grafik erkennt, sitzen die Tessellationseinheiten hinter dem Vertex Shader und werden nach diesem ausgeführt. Der Compute Shader ist außerhalb der Renderpipeline angesiedelt. Er hat allerdings begrenzten Zugriff auf die Daten des Grafikspeichers. In den folgenden Kapiteln soll vor allem auf die hardwarebeschleunigte Tessellation (Hull und Domain Shader), die neue GPGPU Unterstützung (Compute Shader) und die Nutzung dieser Techniken für Displacement Mapping eingegangen werden. Shader Beispielcode wird prinzipiell in der Shader Sprache HLSL dargestellt. Er lässt sich aber natürlich auch in andere Sprachen (beispielsweise CG) übertragen.

5.1 Hardware Tessellation

Als Tessellation bezeichnet man die Zerteilung einer großen Fläche in eine Menge kleinerer Flächen, welche zusammen dieselbe Fläche bilden. Tessellation wird vor allem zur dynamischen Steigerung der Polygondichte in bestimmten Bereichen[1] von Oberflächen eingesetzt. Durch die hardwarebeschleunigte Tessellation in DirectX 11 wird diese Funktion ein essentielles Hilfsmittel zur Verbesserung der visuellen Ergebnisse beim Displacement Mapping. Oberflächen, die vorher mit bereits hochdichten Polygonnetzen an die GPU gesendet werden mussten, können nun mittels Tessellation in dieser verdichtet werden. Dies spart nicht nur GPU Bandbreite, sondern ermöglicht es, viele Stufen in der Entwicklung von detaillierten Objekten stark zu vereinfachen. Im Spiele- und 3D-Segment kann davon ausgegangen werden, dass Dreiecke, Vierecke oder Linien tesselliert werden sollen. So ist es nicht verwunderlich, dass die hardwareunterstützte Tessellation in DirectX 11 genau die Tessellation dieser 3 Typen beschleunigt. Um die Tessellation von Flächen zu ermöglichen, ist zunächst eine andere Übertragung der Flächendaten in die GPU nötig. Bisher von Dreiecks-Streifen[2], Dreiecks-Listen[3] und die Linienentsprechungen[4] geprägt, werden nun patchorientierte Datentypen verwendet. Diese Patchtypen haben im Gegensatz zu den vorherigen Typen eine variable Anzahl von Kontrollpunkten, welche pro Patch übertragen werden

[1] In der Regel im Blickfeld des Betrachters
[2] D3DPT_TRIANGLESTRIP
[3] D3DPT_TRIANGLELIST
[4] D3DPT_LINELIST, D3DPT_LINESTRIP

können. Konnte beispielsweise bei einer Dreiecks-Liste ein Dreieck nur aus 3 Punkten bestehen, so besteht bei den neuen *Control Point* Datentypen die Möglichkeit, zwischen einem und bis zu zweiunddreißig Kontrollpunkten zu wählen. Zunächst wird nun in der GPU für jeden Kontrollpunkt der Vertex Shader aufgerufen[5]. Dies ist sowohl bei der Wahl der Menge der Kontrollpunkte, als auch bei der Konzeption des Vertex Shaders zu beachten. Wie die GPU aus diesen Kontrollpunkten ein verwertbares Dreieck (bzw. ein Viereck oder eine Linie) formt, bestimmt der Hull Shader.

5.1.1 Hull Shader

Die Eingabe des Hull Shaders sind die erwähnten Kontrollpunkte. Die Ausgabe, die an den Tessellator weitergeleitet wird, besteht aus Ausgabekontrollpunkten und Patchkonstanten.

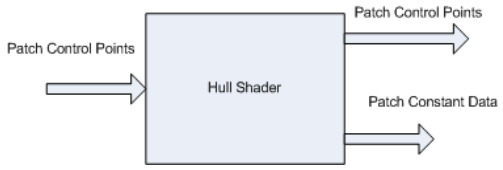

Abbildung 5.2. Hull Shader als Black Box

Ein gültiger Hull Shader besteht aus 3 Kernelementen:

Teil 1: Statische Konfigurationsdaten für den Tesselator

Der Tessellator ist im Gegensatz zur sonstigen Render Pipeline eine fixe Funktionseinheit. Er kann nicht vom Entwickler programmiert werden. Daher existiert auch kein Tessellation Shader. Um ihm dennoch eine gewisse Flexibilität zu verleihen, kann er aber mithilfe von statischen Definitionen im Hull Shader konfiguriert werden. Außerdem können ihm noch Tessellationsfaktoren übergeben werden, welche bestimmen, wie stark ein Patch zerteilt werden soll. Hier sind nur die wichtigsten Definitionen aufgezählt. Außerdem wird darauf eingegangen welche Werte für diese Definitionen gültig sind und was die einzelnen Konfigurationsparameter bewirken.

[5] Siehe hierzu Abbildung 5.1

Folgende Definitionen sind im Hull Shader gültig:

- **domain("x")** Mit *tri*, *quad* oder *isoline* für *x*.
- **partitioning("x")** Mit *integer*, *fractional_even*, *fractional_odd* oder *pow2* für *x*.
- **outputtopology("x")** Mit *line*, *triangle_cw* oder *triangle_ccw* für *x*.
- **outputcontrolpoints("x")** Mit $x \in \{1, .., 32\}$.
- **patchconstantfunc("x")** Mit x als Name der Funktion.
- **maxtessfactor("x")** Mit $x \in \{1, .., 64\}$

domain

Die domain Konfiguration bestimmt, in welcher Domain die Patches liegen. Sie legt also fest, welcher Typ von Patches (Dreiecke, Vierecke oder Linien) in den Hull Shaders eingespeist wird. Die Anzahl der Kontrollpunkte ist hier nicht entscheidend, sondern nur wie diese im Hull Shader verarbeitet werden sollen.

partitioning

Dieser Parameter legt fest, wie der Tessellator die Tessellationsfaktoren zu interpretieren hat. Wobei *integer* bedeutet, dass die Tessellationsfaktoren auf Ganzzahlwerte gerundet werden. *fractional_even* und *fractional_odd* bedeutet entsprechend, dass die Tessellationsfaktoren auf gerade bzw. ungerade Gleitkommawerte gerundet werden. Bei *pow2* werden die Faktoren immer auf Zweierpotenzwerte hochgerechnet.

outputtopology

Dieser Faktor gibt an, welche Art von Objekt der Tessellator ausgeben soll. Zur Auswahl stehen hier normale Linien (*line*), Dreiecke mit im Uhrzeigersinn angegebenen Koordinaten (*triangle_cw*) und Dreiecke mit gegen den Uhrzeigersinn angegebnen Koordinaten (*triangle_ccw*). Dass keine Vierecke angegeben werden können, hat den einfachen Grund, dass Vierecke von der GPU in der Randerpipeline nach dem Tessellator nicht mehr verarbeitet werden können.

outputcontrolpoints

Diese Konfigurationsdefinition entscheidet darüber, wie oft die Hull Shader Funktion pro Patch aufgerufen wird und wieviele Kontrollpunkte vom Hull Shader ausgegeben werden. Gegensätzlich zum Vertex Shader wird der Hull Shader nicht pro Kontrollpunkt der **Eingabe**, sondern pro Kontrollpunkt der **Ausgabe** aufgerufen. Dies ermöglicht z.B. die Eingabe von zweiunddreißig Kontrollpunkten in den Hull Shader, jedoch die Ausgabe von weniger Kontrollpunkten in den Tessellator. Hierdurch können Berechnungen im Tessellator und den folgenden Renderstufen beschleunigt werden, ohne die Kalkulationen im Hull Shader mit weniger Daten durchführen zu müssen. Der Tessellator wird einmal pro Ausgabekontrollpunkt aufgerufen.

patchconstantfunc

Diese Konstante legt den Namen der Funktion fest, welche einmalig pro Patch aufgerufen wird, und Konstanten für diesen Patch berechnet. Die Konstanten, welche von dieser Funktion berechnet werden sollen, sind die bereits erwähnten Tessellationsfaktoren. Sie bestimmen die Menge an Unterteilungen, die vom Tessellator vorgenommen werden soll. Der Wertebereich dieser Faktoren ist abhängig von der in **partitioning** angegebenen Faktorvariante und muss unbedingt eingehalten werden:

- **fractional_odd** $[1, 63]$
- **fractional_even** $[2, 64]$
- **integer** $[1, 64]$
- **pow2** $[1, 64]$

Die Menge der zu berechnenden Faktoren wiederum ist abhängig von der gewählten Domäne:

- **tri** Vier Faktoren (3 x Kante, 1 x Mitte)
- **quad** Sechs Faktoren (4 x Kante, 2x Mitte)
- **isoline** Zwei Faktoren (1 x Detail, 1x Dichte)

Hier bildet die Linientessellation einen Spezialfall. Da ein Tessellationsfaktor nicht > 64 sein darf, dies aber zu gering für eine feine Tessellation wäre, werden für Linien zwei Faktoren verwendet. Der

erste Faktor (Detail) gibt an, wie oft die Tessellation durchgeführt werden soll. Der zweite Faktor (Dichte) ist der eigentliche Tessellationsfaktor. Er gibt an, in wieviele Teile eine Linie tesselliert werden soll. Ein Tessellationsfaktor von (64, 64) würde also einem Faktor von $64 * 64 = 4.096$ entsprechen.

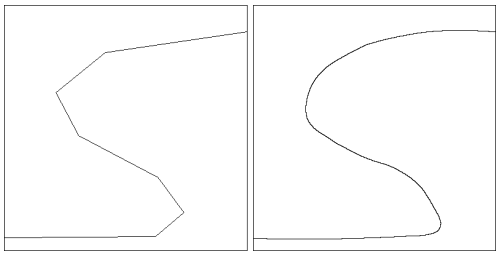

Abbildung 5.3. Tessellation von Linien

maxtessfactor

Dieser gibt den maximal möglichen Tessellationsfaktor für diese Operation an. Gültige Werte müssen im Wertebereich des gewählten *partitioning* liegen und dürfen folglich nicht größer als 64 sein.

Ein gültiger Konfigurationsabschnitt für einen Hull Shader könnte folgendermaßen aussehen

```
1  [domain("tri")]
2  [partitioning("fractional_odd")]
3  [outputtopology("triangle_cw")]
4  [outputcontrolpoints(3)]
5  [patchconstantfunc("cHullShader")]
6  [maxtessfactor(15.0)]
```

Es werden Dreiecke in den Hull Shader eingegeben. Der Tessellationsfaktor ist auf ungerade Gleitkommawerte zu runden. Es werden Dreiecke mit Koordinaten im Uhrzeigersinn ausgegeben. Für jedes Dreieck (Patch) werden drei Kontrollpunkte ausgegeben (entsprechend wird der Hull Shader für jedes Dreieck dreimal aufgerufen). Die pro Patch aufzurufende Funktion heißt “cHullShader”. Und der maximal zulässige Tessellationsfaktor ist 15.

Teil 2: Funktion zur Berechnung von Patch-Konstanten

Die Funktion zur Berechnung von Patch-Konstanten wird im Hull Shader mit der Konfigurationsdefinition *patchconstantfunc* definiert. Ihre Aufgabe ist es, pro Patch Konstanten zu berechnen und diese an den Tessellator zu übergeben. Es laufen parallel mehrere Berechnungen ab. Ebenso laufen andere Berechnungen des Hull Shaders parallel zur Berechnung der Konstanten. Einzig für den Tessellator relevante Konstanten sind die Tessellationsfaktoren. Um diese Aufgabe bewältigen zu können, werden der Funktion der Patch selbst und dessen ID übergeben. Der Patch wird dem Shader als ein n-dimensionales Array[6] zur Verfügung gestellt. Zusätzlich wird der Funktion eine Patch ID über das Semantic *SV_PrimitiveID* übergeben. Diese ID ist später auch im Pixel und Domain Shader verfügbar und ermöglicht es, dort Punkte einem Patch zuzuordnen. Unabhängig von der Eingabe, sind in der Konstantenfunktion zusätzlich alle globalen Shader Konstanten[7] abrufbar. Als Rückgabe der Funktion wird mindestens die benötigte Anzahl Tessellationsfaktoren[8] in ihren Registern erwartet. Die entsprechenden Register sind *SV_TessFactor* für die Kantenfaktoren und *SV_InsideTessFactor* für die inneren Faktoren. Ist einer der Faktoren ≤ 0, so werden die entsprechenden Kontrollpunkte vom Tessellator verworfen. Hiermit lässt sich Culling[9] im Hull Shader implementieren. Sind die Faktoren eines Patches alle 1.0, so wird die Tessellatorstufe für diesen Patch umgangen und die Daten direkt an den Domain Shader weitergeleitet. Dem Tessellator können weitere Konstanten übergeben werden, welche dieser allerdings ignoriert und an den Domain Shader weiterreicht. So müssen Berechnungen, welche für alle Punkte in einem Patch gleich sind, nur einmal pro Patch durchgeführt werden.

[6] n ist abhängig von der Anzahl der Kontrollpunkte
[7] im sogenannten Constant Buffer
[8] Abhängig von der konfigurierten domain
[9] siehe [Wikd]

Eine gültige Patch-Konstantenberechnung könnte folgendermaßen aussehen:

```
 1  struct inp
 2  {
 3      float3 WPosition   : WORLDPOS;
 4      float4 SVPosition  : SV_POSITION;
 5      float4 Normal      : NORMAL0;
 6      float2 TextureUV   : TEXCOORD0;
 7  };
 8
 9  struct outc
10  {
11      float EdgeFactors[3] : SV_TessFactor;
12      float InsideFactor   : SV_InsideTessFactor;
13  };
14
15  outc cHullShader( InputPatch<inp,3> patch,
16                    uint patchID:SV_PrimitiveID )
17  {
18      outc OUT = (outc)0;
19
20      OUT.EdgeFactors[0] = 3.5f;
21      OUT.EdgeFactors[1] = 6.2f;
22      OUT.EdgeFactors[2] = 5.7f;
23      OUT.InsideFactor   = 10.3f;
24
25      return OUT;
26  }
```

Die Faktoren werden in diesem Beispiel statisch und für jeden Patch gleich festgelegt. In der Praxis werden diese natürlich dynamisch und abhängig von verschiedenen Gegebenheiten[10] berechnet. Wichtig ist lediglich, dass sie im vorher konfigurierten Bereich (*maxtessfactor*) liegen.

Teil 3: Funktion zur Berechnung von Kontrollpunkten

Wie bereits erläutert, wird die Funktion zur Berechnung der Kontrollpunkte für jeden Ausgabekontrollpunkt einmal aufgerufen. Als Eingabe stehen auch ihr alle Eingabekontrollpunkte des Patches zur Verfügung. Und es wird die ID des aktuell zu berechnenden Patches über das Semantic *SV_OutputControlPointID* übergeben. Die Ausgabe der Funktion sollte alle Daten des aktuellen Kontrollpunktes beinhalten, welche in späteren Stadien der Pipeline noch benötigt werden. Hierzu können alle in HLSL gültigen Semantics und Datentypen verwendet werden[11]. Als nächste Stufe ist hier

[10] z.B. Entfernung zur Kamera

[11] Eine Liste gültiger Semantics findet sich unter [Micm]

der Domain Shader anzusehen, da der Tessellator die Daten nicht verändert, sondern nur neue Punkte zum Patch hinzufügt. Eine gültige Funktion zur Berechnung der Kontrollpunkte könnte folgendermaßen aussehen:

```
1  struct inp
2  {
3      float3 WPosition   : WORLDPOS;
4      float4 SVPosition  : SV_POSITION;
5      float4 Normal      : NORMAL0;
6      float2 TextureUV   : TEXCOORD0;
7  };
8
9  struct outp
10 {
11     float3 Position    : WORLDPOS;
12     float4 Normal      : NORMAL0;
13     float2 TextureUV   : TEXCOORD0;
14 };
15
16 outp hull( InputPatch<inp, 3> patch,
17             uint patchID : SV_OutputControlPointID )
18 {
19     outp OUT = (outp)0;
20
21     OUT.Position  = patch[patchID].WPosition;
22     OUT.Normal    = patch[patchID].Normal;
23     OUT.TextureUV = patch[patchID].TextureUV;
24
25     return OUT;
26 }
```

In diesem einfachen Beispiel werden keine Kontrollpunkte verändert oder entfernt. Für jeden Ausgabepatch werden die Daten des entsprechenden Eingabepatches einfach übernommen. Selbst wenn ein so simpler Shader keinen direkten Nutzen hätte, so wäre er dennoch nötig, um die Tessellatorstufe zu aktivieren. Wird kein Hull Shader gebunden, so wird die komplette Hardware Tessellation (Hull Shader, Tessellator, Domain Shader) deaktiviert. Es ist nicht möglich, den Domain Shader ohne Hull Shader zu betreiben. Der Hull Shader wird mit der Funktion *CreateHullShader*[Mici] erstellt und anschließend mit *HSSetShader*[Micj] an die Renderpipeline gebunden.

5.1.2 Tessellator

Da die Tessellatorstufe nur konfigurierbar, aber nicht programmierbar ist, sind die Möglichkeiten der Einflussnahme in diese Stufe begrenzt. Alle wichtigen Einstellungen wurden bereits im

vorhergehenden Kapitel beschrieben und müssen im Hull Shader vorgenommen werden. Die Tessellation besteht aus zwei Stufen:

1. Stufe: Runden der Tessellationsfaktoren, wenn nötig, und Behandlung von sehr kleinen Tessellationsfaktoren mittels Fließkommaarithmetik.
2. Stufe: Basierend auf der gewählten Partitionierung (*partitioning*) werden baryzentrische Koordinaten für neue Punkte im Patch berechnet. Dies ist die Hauptarbeit des Tessellators und geschieht hardwarebeschleunigt.

Der Tessellator wird entsprechend einmal pro Patch aufgerufen. Nachdem dieser seine Arbeit beendet hat, wird der Domain Shader einmal pro Kontrollpunkt des neuen Patches aufgerufen.

5.1.3 Domain Shader

Damit der Domain Shader seine Arbeit verrichten kann, muss ihm in einer Konfigurationsdefinition mitgeteilt werden, in welcher Domäne er arbeitet. Dies geschieht wie im Hull Shader mit der Definition *domain(x)*[12]. Der Tessellator stellt dem Domain Shader bei jedem Aufruf sowohl seine ursprünglichen Eingaben (Konstanten und Patch Kontrollpunkte), als auch baryzentrischen Koordinaten des aktuellen Kontrollpunktes zur Verfügung. Die baryzentrischen Koordinaten sind über das Semantic *SV_DomainLocation* abrufbar. Anhand dieser Koordinaten und den Kontrollpunkten des Patches lassen sich die Weltkoordinaten des Punktes im Raum berechnen. Seien $\vec{x_1}, \vec{x_2}$, und $\vec{x_3}$ die Kontrollpunkte eines Dreiecks und b_x, b_y und b_z die baryzentrischen Koordinaten eines Punktes $\vec{p}$, dann lassen sich die Weltkoordinaten des aktuellen Punktes wie folgt berechnen:

$$\vec{p} = b_x * \vec{x_1} + b_y * \vec{x_2} + b_z * \vec{x_3}$$

Die gleiche Vorgehensweise kann bei anderen Koordinatentypen (z.B. Texturkoordinaten, Normale) angewandt werden. Sobald die Koordinaten von Normale und Kontrollpunkt vorliegen, kann im Domain Shader das Displacement Mapping durchgeführt werden. Hierzu wird, wie in Kapitel 4 beschrieben, der Punkt entlang seiner berechneten Normalen verschoben. Der Domain Shader wird mit der Funktion *CreateDomainShader*[Mich] erstellt und anschließend mit *DSSetShader*[Mice] an die Renderpipeline gebunden. Ein

[12] Gültige Werte für x siehe Abschnitt 5.1.1

einfacher Domain Shader für Displacement Mapping könnte folgendermaßen aussehen:

```
1  Texture2D heightMap  : register( t0 );
2
3  struct inp
4  {
5      float3 Position    : WORLDPOS;
6      float4 Normal      : NORMAL0;
7      float2 TextureUV   : TEXCOORD0;
8  };
9
10 struct inc
11 {
12     float EdgeFactors[3] : SV_TessFactor;
13     float InsideFactor   : SV_InsideTessFactor;
14 };
15
16 struct outp
17 {
18     float4 Position       : SV_POSITION;
19     float4 Normal         : NORMAL0;
20     float2 TextureUV      : TEXCOORD0;
21 };
22
23 [domain("tri")]
24 outp domainShader( inc constIn,
25                    float3 bCoords : SV_DomainLocation,
26                    const OutputPatch<inp,3> patch )
27 {
28     outp OUT = (outp)0;
29
30     float4 Position = float4( bCoords.x * patch[0].Position +
31                               bCoords.y * patch[1].Position +
32                               bCoords.z * patch[2].Position, 1.0f );
33
34     float2 TextureUV =        bCoords.x * patch[0].TextureUV +
35                               bCoords.y * patch[1].TextureUV +
36                               bCoords.z * patch[2].TextureUV;
37
38     float4 Normal =           bCoords.x * patch[0].Normal +
39                               bCoords.y * patch[1].Normal +
40                               bCoords.z * patch[2].Normal;
41
42     Normal = normalize( Normal );
43
44     float height = heightMap.SampleLevel( samplerLinear, TextureUV, 0 );
45     Position += Normal * ( height * displacmentFactor );
46
47     OUT.Position = mul( mul( Position, View), Projection);
48     OUT.Normal = Normal;
49     OUT.TextureUV = TextureUV;
50     return OUT;
51 }
```

Deutlich zu erkennen sind in den Zeilen 30, 34 und 38 die Berechnungen der Position, Texturkoordinaten und der Normalen des Kontrollpunktes mithilfe seiner baryzentrischen Koordinaten. Nachdem die Normale wieder normalisiert wurde, wird der Höhenwert mittels der Texturkoordinaten aus der Höhenkarte ausgelesen (Zeile 44). In Zeile 45 findet dann das eigentliche Displacement Mapping statt. Die Variable *DisplacementFactor* wird in diesem Beispiel über einen *Constant Buffer*[13] geladen und ist deswegen im Shader global verfügbar. Nachdem die neue Position berechnet wurde, wird sie zur weiteren Verwendung mit der Ansichts- und Projektionsmatrix multipliziert (Zeile 47). Wie man erkennen kann, entspricht die Ausgabe des Domain Shaders der Ausgabe eines üblichen Vertex Shaders, da die nächste Stufe in der Renderpipeline der Pixel Shader ist.

5.2 Compute Shader

Seit einiger Zeit unter dem Begriff GPGPU bekannt, erfreut sich die Programmierung der GPU für normale Zwecke einer immer weiter steigenden Beliebtheit. Berechnungen auf der GPU haben mehrere Vorteile gegenüber denen in der CPU. So rechnet die GPU ihre Aufgaben stark parallelisiert aus, was mit einem entsprechenden Algorithmus bestimmte Berechnungen stark beschleunigen kann. Außerdem ist die Anbindung der GPU an ihren Speicher schneller als die der CPU an deren. Der Speicher selbst läuft in der Regel auch in wesentlich höheren Takten als dies bei der CPU der Fall ist. Deshalb ist mittlerweile der GPU Markt die treibende Kraft bei der Entwicklung neuer Speichertechnologien. GPGPU Berechnungen haben schon lange ihr Nischendasein beendet und sind heute eine feste Größe bei der Berechnung komplexer Daten in der Wissenschaft. Musste man in den Anfangszeiten noch komplizierte Umwege gehen, um Daten parallelisiert in der GPU berechnen zu können, so gibt es mittlerweile eine Vielzahl von APIs, welche einem dabei behilflich sind. Die beiden meist verwendeten APIs in diesem Sektor sind bisher CUDA (Nvidia) und OpenCL (Khronos Group). Mit DirectX 11 bringt Microsoft nun ebenfalls eine standardisierte Möglichkeit von Berechnung auf der GPU. Der Compute Shader[14] soll die GPGPU Berechnungen mit

[13] siehe [Micn]
[14] Von Microsoft auch Direct Compute genannt

HLSL vereinheitlichen und an DirectX angliedern. Hierdurch verspricht sich Microsoft eine bessere Anbindung von den in der GPU berechneten Daten zu deren Verwendung in der GPU. Außerdem muss der Shader von allen GPUs, welche DirectX 11-fähig sind, unterstützt werden[15]. Das ermöglicht Anwendern eine garantierte Kompatibilität der Hardware zur eingesetzten API. Die Verwendung des Compute Shaders ähnelt sehr stark den anderen HLSL Shader Typen. Jedoch ist die Ein- und Ausgabe anders geregelt, da es keine nächste und keine vorherige Stufe in der Renderpipeline gibt. Außerdem kann die Anzahl der Threads, mit welcher der Shader aufgerufen wird, bei der Entwicklung vom Programmierer selbst bestimmt werden. Zusätzlich stehen einige neue Funktionen zur Verfügung, welche das synchronisieren von Threads erlauben[16].

5.2.1 Anwendungsgebiete

GPGPU Berechnungen finden vor allem dort Anwendung, wo sich Algorithmen stark parallelisieren lassen. Insbesondere, wenn auf einer größeren Datenmenge dieselbe Operation mit verschiedenen Datensegmenten durchgeführt werden soll, kann GPGPU seine Stärken ausspielen. So ist es nicht verwunderlich, dass große Mengen an GPU's mittlerweile Verwendung in wissenschaftlichen Bereichen finden. Aber auch die Spieleindustrie hat ein Auge auf GPGPU geworfen. Lagert man beispielsweise die Berechnung dynamischer Texturen in die GPU aus, so muss das Ergebnis nicht erst wieder in den Grafikspeicher kopiert werden. Hierdurch verhindert man die Serialisierung der GPU mit der CPU, was andernfalls zu Geschwindigkeitsverlusten führen würde. Bei der Berechnung solcher Daten kann die Eingabe einmalig erfolgen bzw. zumindest stark reduziert werden. Beispiele für eine Anwendung im Spielesegment wären:

- Berechnung von Height Maps (z.B. für Wasseroberflächen)
- Berechnung dynamischer Texturen
- Berechnung von Normal Maps aus vorliegenden Daten

[15] Eine partitielle Unterstütztung existiert auch bei neuen DirectX 10 Karten

[16] Man sollte, wenn möglich, auf sie verzichten, da sie die Stärken des GPGPU teilweise zunichte machen

5.2.2 Verwendung

Zur Verwendung des Compute Shaders muss dieser, genau wie alle anderen Shader, zunächst kompiliert und gebunden werden. Das Kompilieren kann wie üblich mit dem Befehl *D3DX11CompileFromFile* durchgeführt werden. Durch *CreateComputeShader*[Micg] entsteht aus dem binären Blob danach ein *ID3D11ComputeShader*. Als Eingabe kann dem Compute Shader eine normale Textur oder ein *StructuredBuffer* dienen. Im Falle des *StructuredBuffer* muss der Datentyp der Eingabedaten angegeben werden. Dies kann mit der in C üblichen Typendefinition geschehen:

$StructuredBuffer < MyStructure > mySB;$

MyStructure ist hierbei durch einen gültigen Datentypen[17] zu ersetzen. Die Shader Ressource muss vor ihrer Verwendung mit der Funktion *CSSetShaderResources*[Micb] an den Shader gebunden werden, damit dieser auf die Daten zugreifen kann. Hierfür benötigt man ein *Shader Resource View* für den Puffer. Dieser lässt sich mit *CreateShaderResourceView*[Mick] erstellen. Hat man diese Schritte befolgt, so hat der Compute Shader nun im Texturregister Zugriff auf den Puffer. Bevor man den Compute Shader nun starten kann, benötigt dieser noch einen Ausgabepuffer. Dieser kann entweder eine beschreibbare Textur sein (z.B. $RWTexture2D$) oder wieder ein $StructuredBuffer$. In beiden Fällen muss der Datentyp mit korrekter Größe angegeben werden. Hierzu wird wieder die Typendefinition verwendet. Wichtig ist, dass für jeden erstellten Puffer (ob Texture oder StructuredBuffer) ein *Unordered Access View* erstellt und gebunden werden muss. Andernfalls kann der Shader nicht in den Puffer schreiben. Erstellt werden kann dieser mit dem Befehl *CreateUnorderedAccessView*[Micl]. Danach wird er mit *CSSetUnorderedAccessViews*[Micc] an den Compute Shader gebunden. Nachdem man alle nötigen Puffer gebunden hat[18], wird der richtige Compute Shader mit *CSSetShader*[Mica] selektiert. Zu guter letzt wird der geladene Shader mit der Funktion *Dispatch*[Micd] gestartet. *Dispatch* akzeptiert dabei drei Parameter vom Typ Integer (x, y, z). Jeder dieser Parameter gibt die Anzahl an Threadgruppen, welche für die entsprechende Richtung gestartet werden sollen. Stellt man sich die GPU als dreidimensionales Array vor, so gibt es Threadgruppen in x-, y- und

[17] Sowohl Selbstdefinierte als auch Intrinsische
[18] Es kann auch mehr als ein Ein- bzw. Ausgabepuffer gebunden werden

z-Richtung. Jede dieser Gruppen kann wiederum maximal 1.024 Threads haben, wobei die maximale Anzahl von Threads in einem Shader nicht 1.024 übersteigen darf. Würde man z.B. die Funktion *Dispatch* mit den Werten (10, 10, 10) aufrufen, so gäbe es insgesamt 1.000 Threadgruppen. Die Anzahl der Threads pro Gruppe wird im Shader selbst über die **numthreads(x, y, z)** Konfigurationsdefinition festgelegt. Der Shader Funktion werden dabei verschiedene Parameter übergeben, welche es ihr erlauben, festzustellen, in welcher Threadgruppe sie liegt und in welchem Thread sie aufgerufen wurde. Ein einfacher Compute Shader zum parallelen Addieren von Zahlen könnte folgendermaßen aussehen:

```
1  Texture2D          inputData1 : register( t0 );
2  Texture2D          inputData2 : register( t1 );
3  RWTexture2D<float> outputData : register( u0 );
4
5  uint2 getCurrentPosition( uint3 groupID, uint3 threadID )
6  {
7      uint2 Pos;
8      Pos.x = ( groupID.x * 32 ) + threadID.x;
9      Pos.y = ( groupID.y * 32 ) + threadID.y;
10     return Pos;
11 }
12
13 [numthreads( 32, 32, 1 )]
14 void compute( uint3 gThreadID : SV_GroupThreadID, uint3 gID : SV_GroupID )
15 {
16     uint2 Pos = getCurrentPosition( gID, gThreadID );
17     outputData[Pos] = inputData1[Pos].x + inputData2[Pos].x;
18 }
```

Wie in Zeile 13 zu sehen ist, werden sowohl in x- als auch in y-Richtung jeweils 32 Threads pro Gruppe (insgesamt 1.024) gestartet. Durch die hochparallelisierte Verarbeitung der GPU, wäre eine solche Berechnung selbst bei sehr großen Texturen sehr schnell. Die Parameter für die *Dispatch* Funktion könnten z.B. $(width/32, height/32, 1)$ mit width und height für die Breite und Höhe der Eingabetextur sein. Bei einer Textur von 1.024 x 1.024 würden dann 32 x 32 Threadgruppen mit jeweils 32 Threads gestartet. Jede dieser Threadgruppen würde seine 32 Threads ausführen und dann terminieren. Insgesamt ergäbe dies 32 x 32 x 32 x 32 = 1.048.576 = 1.024 x 1.024 Additionen. Die Ergebnisse würden danach in der Outputtextur zur Verfügung stehen.

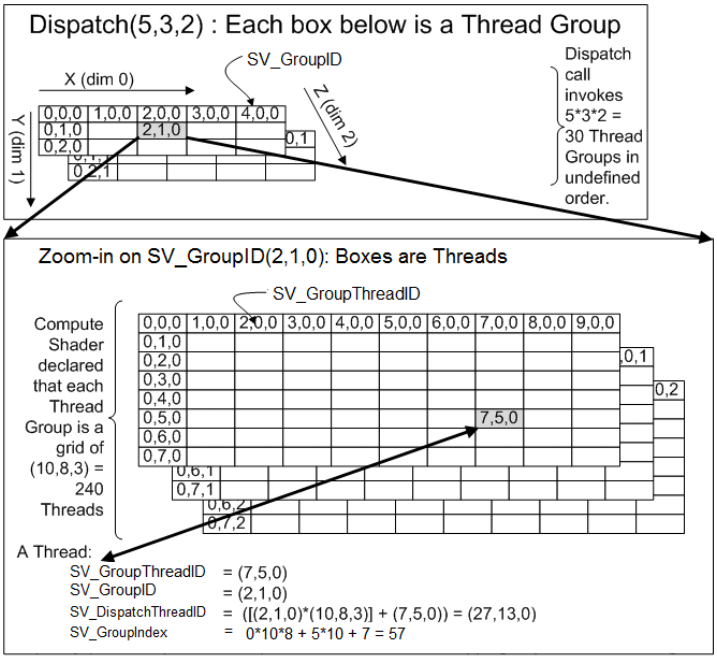

Abbildung 5.4. Threadgruppen

5.2.3 Grenzen

Der Einsatz des Compute Shaders macht, wie alle GPGPU Berechnungen, hauptsächlich bei parallelisierbaren Funktionen Sinn. Lässt sich eine Funktion nicht parallel abarbeiten, so muss man zwischen Ein- und Ausgabekosten und einem eventuellen Geschwindigkeitsgewinn durch die GPU abwägen. Auch wenn eine parallele Bearbeitung möglich ist, so ist in jedem Fall zu verhindern, dass sich die Threads gegenseitig behindern. Auch wenn DirectX 11 Funktionen zur Synchronisation von Threads in Compute Shadern bereitstellt, sollte man sehr sparsam davon Gebrauch machen. Andernfalls droht Geschwindigkeitsverlust in der Abarbeitung der Threads. Wird die GPU noch für andere Aufgaben außer der Berechnung verwendet, so ist außerdem zu beachten, dass man genug Rechenzeit für die anderen Aufgaben einplant. Funktionen, welche stark auf Schleifen und andere Flusskontrollen (z.B. if- oder switch-statements) setzen, müssen vor dem Einsatz im Compute Shader unter Umständen angepasst werden, um die optimale Geschwindigkeit erreichen zu können. Ein großes Problem bei der Berechnung von großen Datenmengen ist auch weiterhin der noch recht geringe Speicher moderner GPUs.

6

Wasser & Displacement Mapping

Die Simulation von realistischem Wasser gilt in der Spielebranche als eine der schwierigsten Aufgaben. Auf der anderen Seite hinterlässt eine detailierte Wasseroberfläche in einer Szene einen tiefen Eindruck. Aufgrund des äußeren Erscheinungsbildes von großen Wasser als geschlossene Oberfläche lässt es sich sehr gut mit Displacement Mapping realisieren. Hierzu wird zunächst ein flaches Polygongitter erstellt, welches dann mit dynamisch berechneten Höhenwerten verformt wird. Die nötige Dichte des Polygongitters ist dabei abhängig vom Detailgrad der errechneten Höhenkarte. Sie lässt sich mit einem low-polygon Gitter durch die in Kapitel 5.1 beschriebene Hardwaretessellation erreichen und im Domain Shader (siehe Kapitel 5.1.3) mittels Displacement Mapping entsprechend verformen. Die Höhenkarte selbst kann, wie in Kapitel 5.2 beschrieben, ebenfalls in der GPU berechnet werden. Für das hier vorgestellte Beispiel wird auf eine interaktive Wasseroberfläche verzichtet.

6.1 Höhenkarte berechnen

Die beiden bekanntesten Arbeiten im Bereich der Wassersimulation sind [Tes04] und [Gom02]. Da sich die Arbeit von Miguel Gomez auf die Simulation von interaktivem Wasser spezialisiert, konzentriert sich die vorliegende Arbeit auf die Umsetzung der Tessendorf-Methode im Compute Shader. In seinem Artikel "*Simulating Ocean Water*" beschreibt Jerry Tessendorf die Berechnung von Höhenkarten, welche sich an statistischen Beobachtungen von Meereswellen orientieren. Anhand dieser Grundlagen werden zunächst Startwerte und Dispersionen von Wellen errechnet,

welche danach, abhängig von einem Zeitwert und unter Zuhilfenahme der inversen schnellen Fourier Transformation (kurz: iFFT), in eine brauchbare Höhenkarte überführt werden. Um diese Höhenkarte in der GPU berechnen zu können, gilt es, zwei Compute Shader zu entwickeln. Der erste Shader berechnet anhand der Startwerte und eines Zeitindexes die Frequenzwerte der Wellen. Der zweite Shader überführt mittels iFFT diese Frequenzwerte in eine Höhenkarte. Da alle Berechnungen in der GPU und mit Daten im GPU-Speicher durchgeführt werden, steht abschließend dem Domain Shader im GPU-Speicher eine Höhenkarte zum Anwenden von Displacement Mapping zur Verfügung.

6.2 Tessendorf im Compute Shader

Zur Berechnung der zeitabhängigen Frequenzwerte müssen zunächst einmalig zufällige Startwerte berechnet werden. Es handelt sich bei diesen Werten um komplexe Zahlen, welche aus zwei Komponenten[1] bestehen. Die Berechnung geschieht, wie in [Tes04] beschrieben, mit dem Phillips Spektrum, einigen statischen Konfigurationsdaten und zwei Gauss'schen Zufallszahlen. Zusätzlich zu den Startwerten wird für jeden Punkt die positionsabhängige Dispersion und ein Skalierungsfaktor[2] berechnet. Um diese Daten später einfacher in den Compute Shader übertragen zu können, werden sie als *float* Werte in einer RGBA Textur gespeichert. Hierbei sind R & G die Werte der komplexen Zahl an der entsprechenden Position, der Wert in B ist die Dispersion und A ist der Skalierungsfaktor. Da die Berechnung nur abhängig von vorberechneten Werten und der Zeit ist, können alle Ausgangswerte gleichzeitig und parallel berechnet werden. Der Algorithmus eignet sich also optimal zur Berechnung mit dem Compute Shader. Die Ausgangswerte lassen sich, zur weiteren Verarbeitung, in einer 2D-Textur speichern. Die vorberechneten Werte zur Skalierung der Höhendaten nach der iFFT werden dabei einfach in die Ausgangstextur weitergereicht. Es ist wichtig, die Threadmenge nicht zu dicht am Limit zu halten, da zu viele parallele Speicherzugriffe mit sinkender Performance bestraft werden können. Da jede Berechnung sowohl lesen, als auch schreiben auf jeweils eine 2 Texturen im GPU-Speicher zugreifen, gilt es, eine ausgewogene Threadanzahl zu finden. Die-

[1] Real und Imaginär
[2] Da während der Berechnung der iFFT die Ausgabe skaliert wird

se lässt sich am besten durch praktische Versuche mit steigender Threadanzahl ermitteln. Ein Compute Shader zur Berechnung der zeitabhängigen Frequenzwerte lässt sich folgendermaßen realisieren:

```
1  Texture2D           inp  : register(t0);
2  RWTexture2D<float4> outp : register(u0);
3
4  uint2 getCurrentPosition(uint3 groupID, uint3 threadID)
5  {
6      uint2 Pos;
7      Pos.x = (groupID.x * 16) + threadID.x;
8      Pos.y = (groupID.y * 16) + threadID.y;
9      return Pos;
10 }
11
12 [numthreads(16, 16, 1)]
13 void tessendorf(uint3 gThreadID : SV_GroupThreadID, uint3 gID : SV_GroupID)
14 {
15     uint2 Pos = getCurrentPosition(gID, gThreadID);
16     uint2 negPos;
17     negPos.x = gridSize - Pos.x - 1
18     negPso.y = gridSize - Pos.y - 1;
19
20     float2 aPos = inp[Pos].xy;
21     float2 aNeg = inp[negPos].xy;
22     float dispersion = inp[Pos].z;
23
24     float sinw, cosw;
25     sincos(dispersion * displayTime, sinw, cosw);
26
27     float4 values;
28     values.x =   ( aPos.x + aNeg.x ) * cosw
29                - ( aPos.y + aNeg.y ) * sinw;
30     values.y =   ( aPos.y - aNeg.x ) * sinw
31                + ( aPos.x - aNeg.y ) * cosw;
32
33     values.z = inp[Pos].z;
34     values.w = 1.0f;
35
36     outp[Pos] = values;
37 }
```

In Zeile 20 - 21 werden die Eingabewerte in separate Variablen eingelesen, um übermäßigen Speicherzugriff zu verhindern. Mit dem Befehl *sincos* wird in Zeile 25 der Sinus und Cosinus der zeitabhängigen Dispersion gleichzeitig berechnet. Dies geschieht in modernen GPUs hardwarebeschleunigt und ist ensprechend performant. Die Werte *displayTime* und *gridSize* werden zur Berechnung über einen *Constant Buffer* in den Shader geladen. Dieser Puffer wird bei jedem Frame mit den ensprechenden Daten aktualisiert. Die eigentliche Berechnung findet in den Zeilen 28 bis

31 statt. In Zeile 33 wird der Skalierungsfaktor aus der Eingabe in die Ausgangstextur weitergereicht. Zeile 34 ist der Tatsache geschuldet, dass HLSL keine Zuweisung in Ausgabetexturen zulässt, wenn nicht alle Werte einer Variable definiert sind. Da DirectX keine RGB Texturen als Ausgabe in Compute Shadern akzeptiert, muss hier ebenfalls eine RGBA Textur verwendet werden.

6.3 Fourier im Compute Shader

Eine zweidimensionale Fourier-Transformation lässt sich berechnen, indem zuerst horizontal für jede Zeile und danach vertikal über jede Spalte ihre eindimensionale Fourier-Transformation berechnet wird. Weil die Spalten- und Zeilenberechnungen unabhängig voneinander sind, lassen sie sich parallel berechnen. Als Basis für die hier gezeigte HLSL iFFT-Implementierung dient [Bou93]. Da die Implementierung stark von Schleifen und anderen Flusskontrollen abhängig ist, kann sie möglicherweise noch stark optimiert werden. Die gezeigte Vorgehensweise sollte entsprechend als Ansatz zu einer Lösung gedacht sein und nicht als endgültig betrachtet werden. Der iFFT Compute Shader ist in vier Schritte unterteilt:

1. Bitumkehr Horizontal
2. Zeilenweise eindimensionale iFFT
3. Bitumkehr Vertikal
4. Spaltenweise eindimensionale iFFT

Die nötigte Bitumkehr kann auf allen Daten parallel ablaufen:

```
1  Texture2D           inputData : register( t0 );
2  RWTexture2D<float4> OUT       : register( u0 );
3
4  [numthreads( 16, 16, 1 )]
5  void revBitsH( uint3 tIDInG : SV_GroupThreadID, uint3 gID : SV_GroupID )
6  {
7      uint2 Pos = getCurrentPosition(groupID, tIDInG);
8      uint xPos = reversebits( Pos.x << MAXSHIFT );
9      uint2 revPos = uint2(xPos, Pos.y);
10
11     float4 values;
12     values.x = inputData[Pos].x;
13     values.y = inputData[Pos].y;
14     values.z = inputData[revPos].z;
15     values.w = 1.0f;
16
17     OUT[revPos] = values;
18 }
```

Zunächst wird in Zeile 7 die Position der Daten abgefragt. Die Funktion *getPosition* funktioniert hierbei wie in den vorherigen Shadern. Da es sich hier um den Shader zur Bitumkehr in Horizontaler Richtung handelt, wird in Zeile 8 diese Bitumkehr an der x-Position durchgeführt. Die Zeilen 11 bis 17 speichern schließlich die Daten in der neu errechneten Position. Der Shader zur vertikalen Bitumkehr funktioniert analog in y-Richtung und wird daher nicht extra vorgestellt. Als nächstes muss die eindimensionale iFFT Funktion erstellt werden:

```
1  void iFFT(inout double2 workData[WORKSIZE])
2  {
3      double c1 = -1.0;
4      double c2 = 0.0;
5      int l2 = 1;
6
7      for (int l = 0; l < int(passes); l++)
8      {
9          int l1 = l2;
10         l2 <<= 1;
11         double u1 = 1.0;
12         double u2 = 0.0;
13
14         for (int j = 0; j < l1; j++)
15         {
16             for (int i = j; i < int(gridSize); i += l2)
17             {
18                 int i1 = i + l1;
19                 double t1 = u1 * workData[i1].x - u2 * workData[i1].y;
20                 double t2 = u1 * workData[i1].y + u2 * workData[i1].x;
21                 workData[i1].x = workData[i].x - t1;
22                 workData[i1].y = workData[i].y - t2;
23                 workData[i].x += t1;
24                 workData[i].y += t2;
25             }
26
27             double z =  u1 * c1 - u2 * c2;
28             u2 = u1 * c2 + u2 * c1;
29             u1 = z;
30         }
31
32         c2 = sqrt(float(1.0 - c1) / 2.0);
33         c1 = sqrt(float(1.0 + c1) / 2.0);
34     }
35 }
```

Die Funktion orientiert sich stark an Bourkes Algorithmus. Lediglich die Verwendung einiger HLSL spezifischer Datentypen lässt den Unterschied erkennen. *WORKSIZE* ist eine Präprozessordefinition, welche zur Kompilierzeit mit dem echten Wert ersetzt wird. *passes* und *gridSize* werden über einen *Constant Buffer* geladen.

Zum Schluss wird noch der iFFT Compute Shader selbst benötigt, der die oben vorgestellte Funktion mit den korrekten Daten aufruft:

```
1  [numthreads( 1, 16, 1 )]
2  void computeFFTH(uint3 tIDInG : SV_GroupThreadID, uint3 gID : SV_GroupID)
3  {
4      uint2 Pos = getCurrentPosition(gID, tIDInG);
5      double2 workData[WORKSIZE];
6
7      for (uint x = 0; x < uint(gridSize); x++)
8      {
9          workData[x] = inputData[uint2(x, Pos.y)].xy;
10     }
11
12     iFFT(workData);
13
14     for (uint b = 0; b < uint(gridSize); b++)
15     {
16         float4 values;
17         values.x = float(workData[b].x);
18         values.y = float(workData[b].y);
19         values.z = inputData[uint2(b, Pos.y)].z;
20         values.w = 1.0f;
21         OUT[uint2(b, Pos.y)] = values;
22     }
23 }
```

Die Zeilen 4 bis 10 laden die Daten für diesen Thread in einen Zwischenspeicher. In Zeile 12 wird die iFFT-Funktion mit diesem aufgerufen. Schließlich wird er in den Zeilen 14 bis 22 in die Ausgangstextur geschrieben.

6.4 Abschließende Berechnungen

Nachdem die iFFT im Compute Shader die Höhenwerte der Wasseroberfläche ermittelt hat, sind einige abschließende Berechnungen nötig. So muss für jeden Punkt anhand seiner Nachbarpunkte die Normale der Fläche berechnet werden[3]. Außerdem muss der Skalierungsfaktor auf die Höhenwerte angewendet werden. Als Endergebnis steht dem Domain Shader dann im GPU-Speicher eine Höhenkarte für Displacement Mapping zur Verfügung. Der Compute Shader für alle abschließenden Berechnungen lässt sich folgendermaßen implementieren:

[3] siehe 4.1

```
1  Texture2D           inp  : register(t0);
2  RWTexture2D<float4> outp : register(u0);
3
4  uint2 getCurrentPosition(uint3 groupID, uint3 threadID)
5  {
6      uint2 Pos;
7      Pos.x = (groupID.x * 16) + threadID.x;
8      Pos.y = (groupID.y * 16) + threadID.y;
9      return Pos;
10 }
11
12 [numthreads(16, 16, 1)]
13 void normals(uint3 gThreadID : SV_GroupThreadID, uint3 gID : SV_GroupID)
14 {
15     uint2 Pos = getCurrentPosition(gID, gThreadID);
16     uint2 Pos2 = uint2(Pos.x + 1, Pos.y);
17     uint2 Pos3 = uint2(Pos.x, Pos.y + 1);
18
19     float weight = inp[Pos].z;
20     float height = inp[Pos].x * weight;
21
22     float3 ta, tb, tc;
23     ta.x = float(gridRatio);
24     ta.y = 0.0f;
25     ta.z = float(inp[Pos2].x - inp[Pos].x) * weight;
26     tb.x = 0.0f;
27     tb.y = float(gridRatio);
28     tb.z = float(inp[Pos3].x - inp[Pos].x) * weight;
29
30     tc = cross( ta, tb );
31
32     float4 output;
33     output.r = tc.x;
34     output.g = tc.y;
35     output.b = tc.z;
36     output.a = height;
37
38     outp[Pos] = output;
39 }
```

Zeilen 19 und 20 skalieren den Höhenwert auf normale Werte zurück. In Zeile 30 wird mit dem Kreuzprodukt von Vektoren der Nachbarpunkte die Normale berechnet. Diese Normale wird in den RGB Werten der Ausgangstextur gespeichert. Der Höhenwert für das Displacement Mapping wird im Alphawert der Textur untergebracht, wo er später vom Domain Shader ausgelesen werden kann.

7

Pixel Shader

Um eine realistische Darstellung von Land & Wasser in der Simulation erreichen zu können, ist die Implementierung eines entsprechenden Pixel Shaders unerlässlich. Dabei unterscheidet sich der Pixel Shader für die Wassersimulation grundlegend von der für das Rendering der Landschaft. Gemein haben beide die Basisformeln zur Berechnung von Lichtverhältnissen. Hierbei wird auf das Blinn-Phong-Beleuchtungsmodell [Wikb] zurückgegriffen, welches eine realistische Darstellung natürlicher Lichtverhältnisse anhand weniger Berechnungen erlaubt.

7.1 Pixel Shader für Land

Die Berechnung der Pixelfarbe für die Landfläche ist relativ simpel gehalten. Wie bereits erwähnt, werden für jeden Pixel mithilfe des Blinn-Phong-Beleuchtungsmodells die Itensitäten von Umgebungslicht, Streulicht und reflektiertem Licht errechnet. Danach wird anhand einer Texturkarte die Stärke von drei unterschiedlichen Texturen als RGB Wert an dem aktuellen Punkt abgelesen. Abschließend werden die Texturfarben addiert, mit den Lichtintensitäten multipliziert und das Ergebnis als Pixelfarbe ausgegeben. Die Intensität des reflektierten Lichts spielt eine Sonderrolle, da ihre Farbe nicht abhängig von der Farbe des Untergrunds ist, sondern durch die Farbe der Lichtquelle bestimmt wird. Sie wird daher zur Endfarbe einfach hinzu addiert. Beispielcode für die geschilderten Berechnungen in HLSL ist im Anhang B.1 zu finden.

7.2 Pixel Shader für Wasser

Die Berechnungen im Wasser Pixel Shader ähneln denen des Land Pixel Shaders. Durch die Transparenz des Wassers, seine Oberfläche und die daraus resultierende Refraktion und Reflektion der Umgebung und des Lichtes gibt es allerdings signifikante Unterschiede. Zusätzlich dazu muss beachtet werden, dass die Transparenz des Wassers aufgrund von Mikropartikeln mit zunehmender Tiefe abnimmt. Eine physikalisch korrekte Simulation dieser Effekte würde die Kapazitäten moderner GPUs auslasten. Um sie wenigstens realitätsnah wiedergeben zu können, sind einige zusätzliche Anstrengungen nötig.

7.2.1 Transmission

Zunächst muss die Transmission[1] des Mediums Wasser an der Position des Pixels berechnet werden. Diese Werte sind abhängig von der Wellenlänge des ausgestrahlten Lichts, der Dichte des Materials (hier Meerwasser), welches durchschienen wird und der Tiefe, welche vom Licht durchdrungen werden muss. Nach [Wike] lässt sich die optische Durchlässigkeit I_1 eines Mediums mit der Formel:

$$I_1 = I_0 \ \mathrm{e}^{-\epsilon^* cd}$$

bestimmen. Wobei I_0 die Intensität des einfallenden Lichtes darstellt. ϵ^* ist der spektrale Absorptionskoeffizient der einfallenden Wellenlänge. c ist die Konzentration des absorbierenden Materials im Medium[2]. Und d schließlich ist die Tiefe des Mediums, welche durchdrungen werden muss[3]. Der wellenlängenabhängige Absorptionskoeffizient von Wasser lässt sich nach [Wika] mit der Formel:

$$\epsilon^* = \frac{-\log T}{w}$$

berechnen. Wobei T die dunkelste Farbe für dieses Spektrum darstellt. Und w die größt mögliche Tiefe des Mediums[4]. Der Absorptionskoeffizient muss für jede Farbe (RGB) separat berechnet werden. Der Wert c ist statisch und d lässt sich pro Pixel im Shader berechnen. Die Transmission lässt sich später mit der Pi-

[1] Lichtdurchlässigkeit
[2] Menge der Verunreinigungen im Wasser
[3] In diesem Fall die Tiefe des Wassers an dieser Stelle *2
[4] Maximal erreichbare Wassertiefe

xelfarbe des Untergrunds an dieser Stelle multiplizieren, um den tiefenabhängigen Transparenzeffekt zu erzeugen.

7.2.2 Refraktion

Refraktion, also Lichtbrechung, findet statt, wenn Licht von einem Medium in ein anderes mit unterschiedlicher Dichte wechselt. In der GPU lässt sich dieser Effekt mit dem Verschieben von Texturkoordinaten simulieren. Auch wenn dies keine physikalisch korrekte Darstellung von Refraktion erzeugt, ist es für Wassersimulation ein ausreichend realistisch wirkender Effekt. Die genaue Vorgehensweise ist in [Sou05] erläutert und soll deswegen hier nur grob skizziert werden. Zunächst muss eine Refraktionstextur erzeugt werden, welche ein genaues Abbild des Terrains unter Wasser darstellt. Um eine solche Textur erzeugen zu können, wird die Szene ohne die Wasseroberfläche in eine Textur gerendert. Mit *Projective Texture Mapping*[5] werden nun projezierte Texturkoordinaten berechnet. Da die Kamera beim Mapping als Projektor dient, können diese mit den selben Sicht- und Projektionsmatrizen berechnet werden wie die restliche Szene. Die Normale der Oberfläche an jeder Stelle wird nun zunächst mit einem statischen Faktor skaliert. Um Artefakte von Objekten oberhalb der Wasseroberfläche zu vermeiden, muss eine *Clipping Plane*[6] dafür sorgen, dass diese Objekte nicht in die Refraktionstextur gerendert werden. Abschließend werden die Texturkoordinaten der Oberfläche mit der skalierten Normale addiert und die so gewonnenen Koordinaten als Texturkoordinate auf der Refraktionstextur angwendet, um den Refraktionsteil der Wasserfarbe zu erhalten.

7.2.3 Reflektion

Reflektionen an der Wasseroberfläche lassen sich durch eine ähnliche Technik wie die Refraktion erzeugen. Entscheidender Unterschied ist die Position der Kamera beim Erzeugen der Reflektionstextur. Um die korrekte Position zu berechnen, wird zunächst die aktuelle Kameraposition an der Wasseroberfläche gespiegelt. Zusätzlich zu der neuen Kameraposition muss die Betrachtungs- und Projektionsmatrix erzeugt werden, da sich die Textur sonst

[5] siehe [Wikf]
[6] siehe [Wikc]

nicht korrekt auf die Wasseroberfläche projezieren lässt. Die so gewonnenen Texturkoordinaten werden dann ebenfalls an der Wasserlinie gespiegelt, um das Spiegelbild zu erhalten. Abschließend werden die Texturkoordinaten auf der Reflektionstextur angewendet, um den Reflektionsteil der Wasserfarbe zu erhalten.

7.2.4 Fresnel Term

Als Fresnel Term wird eine von Augustin Jean Fresnel aufgestellte Formel bezeichnet, welche das Reflektionsverhalten von elektromagnetischen Wellen an ebenen Oberflächen beschreibt. Der Term beschreibt dabei das Verhältnis zwischen transmissivem und reflektivem Teil des auftreffenden Lichts. Die konkrete Formel lautet:

$$F = \frac{1}{2}\frac{\sin(\alpha-\beta)^2}{\sin(\alpha+\beta)^2} + \tan(\alpha-\beta)^2 * \tan(\alpha+\beta)^2$$

Mit α als Winkel zwischen Normale der Oberfläche und des Betrachtungsvektors und β als Winkel zwischen Normale und refraktiertem Betrachtungsvektor. Es ist ersichtlich, dass die Berechnung dieser Formel für jeden Pixel einen enormen Rechenaufwand erzeugen würde. Daher wird der Fresnel Term in der Computergrafik mit folgender Formel approximiert:

$$F = 1.0 - \langle \vec{B}, \vec{N} \rangle^p$$

wobei $\vec{B}$ der Betrachtungsvektor, $\vec{N}$ die Normale der Oberfläche und $\langle\ ,\ \rangle$ das Skalarprodukt darstellt. Der Faktor p kann mehr oder weniger frei gewählt werden. In der Praxis hat sich $p = 5.0$ durchgesetzt, was eine relativ realistische Annäherung an den physikalisch korrekten Fresnel Term darstellt. Hat man den Fresnel Term berechnet, lässt sich mit seiner Hilfe in einer linearen Interpolation zwischen dem berechneten Refraktions- und Reflextionsanteil der Wasserfarbe die Endfarbe des Pixels errechnen.

Beispielcode für die geschilderten Berechnungen in HLSL ist im Anhang B.2 zu finden.

8

Gesamtlösung

Die in den vorhergehenden Kapiteln vorgestellten Lösungen lassen sich auf modernen GPUs mit einer Insel in einer dynamisch generierten Wasserumgebung veranschaulichen. Wie im Kapitel 6 beschrieben, wird dazu im Compute Shader eine Höhenkarte berechnet. Eine Oberfläche, welche mit Tessellation aus Kapitel 5.1 verdichtet wurde, kann im Anschluss durch Displacement Mapping im Domain Shader anhand dieser Höhendaten verformt werden. Eine zweite ebenfalls tessellierte Oberfläche bildet zusammen mit einer statischen Textur und Displacement Mapping die Landmasse. Um der Simulation den letzten Schliff zu verpassen, sorgen die Pixel Shader aus Kapitel 7 für eine entsprechende Einfärbung. Zusätzlich bildet ein Pixel Shader die physikalischen Lichtspiele in flachem bis mitteltiefen Wasser realitätsgetreu nach.

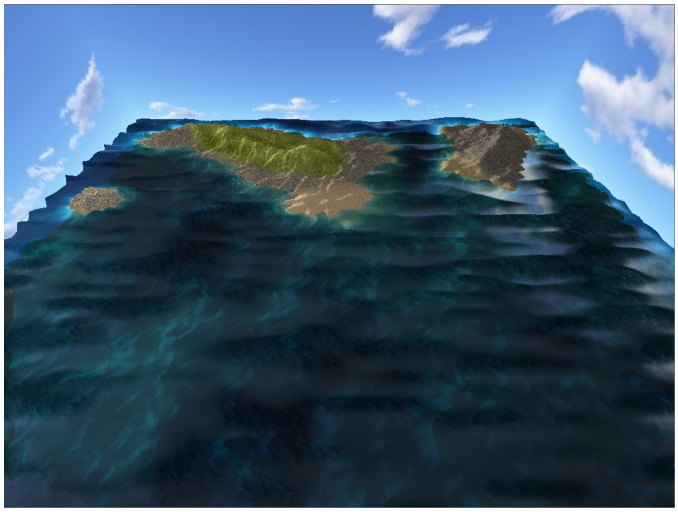

Abbildung 8.1. Simulation von Land und Wasser

9

Fazit

Konnte Displacement Mapping in der Vergangenheit wegen zu hoher GPU-Last kaum in Echtzeitsystemen eingesetzt werden, werten die neuen DirectX 11 Funktionen sie zu einer wahren Killerapplikation auf. Displacement Mapping kann mithilfe von Tessellation auf nahezu jeder Oberfläche eingesetzt werden, um deren Aussehen markant zu verbessern. Selbst Objekte, welche in einer Szene vorher als unwesentlich angesehen wurden, können nun mit diesen Techniken nahezu ohne Geschwindigkeitsverlust erheblich zum Realismus beitragen. Insbesondere die Tatsache, dass Displacement Mapping reale Objekte und nicht nur real wirkende Objekte erzeugt, stellt dabei einen besonderen Reiz dar. Dies führt dazu, dass man bei vielen Techniken auf komplizierte Berechnungen zur Simulation von physikalischen Effekten verzichten kann und spart weitere kostbare GPU-Zeit und -Bandbreite. Weiterhin kann Displacement Mapping mit DirectX 11 dynamische Objekte erzeugen, indem ihre Höhenkarten über den Compute Shader berechnet werden. Außerdem erlaubt der Tessellator das dynamische Culling von Polygonen über den Hull Shader (mit Tessellationsfaktoren ≤ 0). Weitere, in diesem Dokument nicht vorgestellte Techniken, wie z.B. dynamisches Berechnen von Tessellationsfaktoren, erhöhen den praktischen Nutzen der Technik zusätzlich[1]. Die hier gezeigte praktische Anwendung der vorgestellten Techniken deckt somit nur einen Teil der Möglichkeiten von Displacement Mapping, Tessellation und Compute Shadern ab. Dennoch erlaubt sie bereits einen Einblick in zukünftige Möglichkeiten moderner Computergrafik.

[1] siehe hierzu `http://www.nvidia.com/object/tessellation.html`

10

Aussichten

Zu den neuen Funktionen existieren bereits einige Referenzimplementierungen von Grafikkartenherstellern und Microsoft. Auch wenn neue Techniken in der Spielebranche üblicherweise nur zögerlich und schleppend aufgenommen werden, hat Tessellation besonders im Zusammenhang mit Displacement Mapping bereits erste Anwendung gefunden. So wird es bereits von einigen Spieleengines[1] in der Liste der unterstützten Funktionen geführt. Andere Unternehmen[2] haben für neue Versionen ihrer Engines bereits Unterstützung angekündigt oder arbeiten daran. Nahezu alle diese Unternehmen setzen dabei primär auf Displacement Mapping unter Einsatz von Tessellation, um Objekte in der Spieleumgebung realistischer wirken zu lassen. Aber auch auf andere Neuerungen der elften Version von DirectX haben Engine- und Spieleentwickler bereits ein Auge geworfen. So ist es abzusehen, dass der Compute Shader rasch Einzug in die Liste der branchenüblichen Techniken halten wird. Erlaubt er Spieleentwicklern doch die nahezu immer knappen Resourcen wie CPU-Zeit, Speicher und GPU Busbandbreite zu schonen und Berechnungen auf vorhandenen Hochleistungs-GPUs auszulagern. Techniken, wie Displacement Mapping, die bis vor kurzem noch als zu komplex für großflächige Effekte angesehen wurden, werden so bald schon Einzug in den Spielealltag halten. Die Verwendung dieser neuen Techniken wird nicht zuletzt dazu beitragen, dass die Branche ihrem Ziel der photorealistischen Echtzeitanwendungen einen Schritt näher gekommen ist.

[1] z.B. Unigine Engine `http://unigine.com/products/unigine/#technology`
[2] z.B. Crytek (CryEngine 3), Trinigy (Vision Game Engine)

Literatur

Bou93. Bourke, Paul: *Fast Fourier Transform*, 1993. http://paulbourke.net/miscellaneous/dft/.

ETO. *ETOPO1 Global Relief Model*. http://www.ngdc.noaa.gov/mgg/global/global.html.

Gom02. Gomez, Miguel: *Spieleprogrammierung*, Band Gems 1, Seiten 188–194. mitp-Verlag, 2002.

Mica. Microsoft: *ID3D11DeviceContext::CSSetShader Method*. http://msdn.microsoft.com/en-us/library/ff476402%28v=vs.85%29.aspx.

Micb. Microsoft: *ID3D11DeviceContext::CSSetShaderResources Method*. http://msdn.microsoft.com/en-us/library/ff476403%28v=vs.85%29.aspx.

Micc. Microsoft: *ID3D11DeviceContext::CSSetUnorderedAccessViews Method*. http://msdn.microsoft.com/en-us/library/ff476404%28v=vs.85%29.aspx.

Micd. Microsoft: *ID3D11DeviceContext::Dispatch Method*. http://msdn.microsoft.com/en-us/library/ff476405%28v=vs.85%29.aspx.

Mice. Microsoft: *ID3D11DeviceContext::DSSetShader Method*. http://msdn.microsoft.com/en-us/library/ff476420%28v=vs.85%29.aspx.

Micf. Microsoft: *ID3D11DeviceContext::Map Method*. http://msdn.microsoft.com/en-us/library/ff476457%28v=vs.85%29.aspx.

Micg. MICROSOFT: *ID3D11Device::CreateComputeShader Method*. http://msdn.microsoft.com/en-us/library/ff476503%28v=vs.85%29.aspx.

Mich. MICROSOFT: *ID3D11Device::CreateDomainShader Method*. http://msdn.microsoft.com/en-us/library/ff476508%28v=vs.85%29.aspx.

Mici. MICROSOFT: *ID3D11Device::CreateHullShader Method*. http://msdn.microsoft.com/en-us/library/ff476511%28v=vs.85%29.aspx.

Micj. MICROSOFT: *ID3D11Device::CreateHullShader Method*. http://msdn.microsoft.com/en-us/library/ff476511%28v=vs.85%29.aspx.

Mick. MICROSOFT: *ID3D11Device::CreateShaderResourceView Method*. http://msdn.microsoft.com/en-us/library/ff476519%28v=vs.85%29.aspx.

Micl. MICROSOFT: *ID3D11Device::CreateUnorderedAccessView Method*. http://msdn.microsoft.com/en-us/library/ff476523%28v=vs.85%29.aspx.

Micm. MICROSOFT: *Semantics (DirectX HLSL)*. http://msdn.microsoft.com/en-us/library/bb509647%28v=vs.85%29.aspx.

Micn. MICROSOFT: *Shader Constants (DirectX HLSL)*. http://msdn.microsoft.com/en-us/library/bb509581%28v=vs.85%29.aspx.

Sou05. SOUSA, TIAGO: *GPU: Techniques for Graphics and Compute-Intensive Programming*, Band Gems 2, Seite Chapter 19. mitp-Verlag, 2005.

Tes04. TESSENDORF, JERRY: *Simulating Ocean Water*, 2004. http://tessendorf.org/papers_files/coursenotes2004.pdf.

Wika. *Wikipedia - Absorptionskoeffizient*. http://de.wikipedia.org/wiki/Absorptionskoeffizient.

Wikb. *Wikipedia - Blinn - Beleuchtungsmodel*. http://de.wikipedia.org/wiki/Blinn-Beleuchtungsmodell.

Wikc. *Wikipedia - Clippingebene.* http://de.wikipedia.org/wiki/Clippingebene.

Wikd. *Wikipedia - Culling.* http://de.wikipedia.org/wiki/Culling.

Wike. *Wikipedia - Lambert - Beersches Gesetz.* http://de.wikipedia.org/wiki/Lambert-Beersches_Gesetz.

Wikf. *Wikipedia - Projective Texture Mapping.* http://en.wikipedia.org/wiki/Projective_texture_mapping.

Index

A

Glossar

API	**A**pplication **P**rogramming **I**nterface (Programmierschnittstelle)
CG	**C** for **G**raphics (Shader Sprache)
CPU	**C**entral **P**rocessing **U**nit (Prozessor)
FFT	**F**ast **F**ourier **T**ransformation (Schnelle Fourier-Transformation)
FPS	**F**rames **p**er **S**econd (Bilder pro Sekunde)
GPU	**G**raphics **P**rocessing **U**nit (Grafikkarte)
GPGPU	**G**eneral-**P**urpose computation on **G**raphics **P**rocessing **U**nits (Allgemeine Berechnungen auf Grafikkarten)
HLSL	**H**igh **L**evel **S**hading **L**anguage (Shader Sprache)
iFFT	**i**nvers **F**ast **F**ourier **T**ransformation (Inverse Schnelle Fourier-Transformation)
LOD	**L**evel **o**f **D**etail (Detailgranulierung)
RGB	**R**ed **G**reen **B**blue (Farbraum)
RGBA	**R**ed **G**reen **B**blue **A**lpha (Farbraum)

B

Beispielcode

B.1 HLSL Land Pixel Shader

```
1  Texture2D textureMap   : register(t0);
2  Texture2D baseTexture1 : register(t1);
3  Texture2D baseTexture2 : register(t2);
4  Texture2D baseTexture3 : register(t3);
5
6  SamplerState sAnsitropic : register(s0);
7  SamplerState sLinear     : register(s1);
8
9  struct inp
10 {
11     float4 WPosition  : WORLDPOS;
12     float4 SVPosition : SV_POSITION;
13     float4 Normal     : NORMAL0;
14     float3 TextureUVW : TEXCOORD0;
15 };
16
17 float4 terrainPixelShader(in inp IN) : SV_Target
18 {
19     float2 tUVScaled = IN.TextureUVW.xy * 13.0f;
20
21     float3 base1 =
22     baseTexture1.SampleLevel(sAnsitropic, tUVScaled, IN.TextureUVW.z).rgb
23     * textureMap.SampleLevel(sLinear, IN.TextureUVW.xy, IN.TextureUVW.z).r;
24
25     float3 base2 =
26     baseTexture2.SampleLevel(sAnsitropic, tUVScaled, IN.TextureUVW.z).rgb
27     * textureMap.SampleLevel(sLinear, IN.TextureUVW.xy, IN.TextureUVW.z).g;
28
29     float3 base3 =
30     baseTexture3.SampleLevel(sAnsitropic, tUVScaled, IN.TextureUVW.z).rgb
31     * textureMap.SampleLevel(sLinear, IN.TextureUVW.xy, IN.TextureUVW.z).b;
```

In den Zeilen 19 bis 31 werden die Farbintensitäten der drei Basistexturen anhand der Texturkarte ermittelt.

```
1       float3 pixelToLight = normalize(Sun.Position.xyz - IN.WPosition.xyz);
2       float  distanceToLight = distance(Sun.Position.xyz, IN.WPosition.xyz);
3       float3 pixelToEye = normalize(Camera.EyePosition.xyz - IN.WPosition.xyz);
4       float3 halfWayVector = normalize(pixelToLight + pixelToEye);
5       float  fade = pow(distanceToLight, 2);
6
7       float3 Ambient =
8       World.terrainMaterial.AmbientColor.rgb * Sun.AmbientIntensity / fade;
9
10      float  intensity = saturate(dot(pixelToLight, IN.Normal.xyz));
11      float3 Diffuse =
12      intensity * World.terrainMaterial.DiffuseColor.rgb
13                        * Sun.DiffuseIntensity / fade;
14
15      intensity = pow(saturate(dot(IN.Normal.xyz,halfWayVector)), 2.0f);
16      float3 Specular = intensity * World.terrainMaterial.SpecularColor.rgb
17                         * Sun.SpecularIntensity / fade;
18
19      return float4((base1+base2+base3)*(Ambient+Diffuse)+Specular,1.0f);
20  }
```

Die Zeilen 1 bis 17 dienen der Berechnung der Lichtintensitäten anhand des Blinn-Phong-Beleuchtungsmodells. Die Intensitäten und Farbstärken werden in Zeile 19 zur Endfarbe zusammengefasst. Die Variablen *Sun*, *Camera* und *terrainMaterial* werden mit einem *Constant Buffer* in den Shader übertragen und sind statisch.

B.2 HLSL Wasser Pixel Shader

```
1   Texture2D heightMap        : register( t0 );
2   Texture2D terrainHeightMap : register( t1 );
3   Texture2D refractionMap    : register( t2 );
4   Texture2D reflectionMap    : register( t3 );
5
6   SamplerState sAnsitropic  : register(s0);
7   SamplerState sLinear      : register(s1);
8
9   struct inp
10  {
11      float4 WPosition      : WORLDPOS;
12      float4 SVPosition     : SV_POSITION;
13      float4 Normal         : NORMAL0;
14      float3 TextureUVW     : TEXCOORD0;
15      float3 refrTextureUVW : TEXCOORD1;
16      float3 reflTextureUVW : TEXCOORD2;
17  };
```

Die Zeilen 1 bis 17 zeigen die Eingabedaten des Wasser Pixel Shaders. Es ist zu beachten, dass neben den Höhendaten des Wassers auch die des Terrains nötig sind. Zusätzlich zu den gezeigten Daten stehen der *Constant Buffer* des Land Pixel Shaders zur Verfügung.

```
1  float4 waterPixelShader(in inp IN) : SV_Target
2  {
3      float groundHeight =
4      terrainHeightMap.SampleLevel(sLinear, IN.TextureUVW.xy, 0).r
5      * World.terrainDisplacmentFactor;
6
7      float waterSurfaceDepth = Water.maxDepth - groundHeight;
8
9      float displacementFactor = World.waterDisplacmentFactor
10     * ((waterSurfaceDepth - 0.5f) / Water.maxDepth);
11
12     float waterRealDepth = waterSurfaceDepth
13     + (heightMap.SampleLevel(samplerLinear, IN.TextureUVW.xy, 0).a
14     * displacementFactor) + 0.3f;
15
16     clip(waterRealDepth);
```

Die Zeilen 1 bis 14 dienen dem Berechnen der echten Wassertiefe. Durch Zeile 16 werden alle Pixel mit einer Tiefe < 0 verworfen.

```
1      float3 pixelToLight = Sun.Position.xyz - IN.WPosition.xyz;
2      float  distanceToLight = distance(Sun.Position.xyz, IN.WPosition.xyz);
3      float3 pixelToLightN = normalize(pixelToLight );
4      float3 pixelToEyeN = normalize(Camera.EyePosition.xyz - IN.WPosition.xyz);
5      float3 halfWayVector = normalize(pixelToLightN + pixelToEyeN);
6
7      float  fade = pow(distanceToLight, 2);
8
9      float3 Ambient = (Water.material.AmbientColor.rgb
10                       * Sun.AmbientIntensity) / fade;
11
12     float  intensity = saturate(dot(pixelToLightN, IN.Normal.xyz));
13     float3 Diffuse = (intensity * Water.material.DiffuseColor.rgb
14                      * Sun.DiffuseIntensity) / fade;
15
16     intensity = pow(saturate(dot(IN.Normal.xyz, halfWayVector)),
17                     Water.SpecularHardness);
18     float3 Specular = (intensity * Water.material.SpecularColor.rgb
19                       * Sun.SpecularIntensity) / fade;
20
21     float3 waterColor = Ambient + Diffuse + Specular;
```

In den Zeilen 1 bis 5 werden Lichtvektoren erzeugt. Die Zeilen 7 bis 21 dienen der Berechnung von Intensitäten der Lichtanteile.

```
1      float2 projTexCrds;
2      projTexCrds.x = IN.refrTextureUVW.x / IN.refrTextureUVW.z / 2.0f + 0.5f;
3
4      projTexCrds.y = -IN.refrTextureUVW.y / IN.refrTextureUVW.z / 2.0f + 0.5f;
5
6      float2 disTexCrds = projTexCrds
7      + float2(IN.Normal.x * 0.02f, IN.Normal.y * 0.02f);
8
9      float3 refractedColor =
10     refractionMap.SampleLevel(sAnsitropic, disTexCrds, IN.TextureUVW.z).xyz;
```

Mithilfe von *Projected Texture Mapping* werden in den Zeilen 1 bis 4 projezierte Texturkoordinaten berechnet. Die Zeilen 6 und 7 verschieben die Texturkoordinaten entlang der Normalen der Wasseroberfläche. Abschließend wird in den Zeilen 9 und 10 die Bodenfarbe mit den verschobenen Texturkoordinaten abgefragt.

```
1       float2 reflTexCrds;
2       reflTexCrds.x =
3       IN.reflTextureUVW.x / IN.reflTextureUVW.z / 2.0f + 0.5f;
4
5       reflTexCrds.y = IN.reflTextureUVW.y / IN.reflTextureUVW.z / 2.0f + 0.5f;
6
7       float3 reflectedColor =
8       reflectionMap.SampleLevel(sAnsitropic, reflTexCrds, IN.TextureUVW.z).xyz;
9
10      float fresnelTerm = 1.0f - pow( dot(-pixelToEyeN, IN.Normal.xyz), 5.0f);
```

Wie bei der Refraktion wird in den Zeilen 1 bis 8 *Projected Texture Mapping* angewendet. In Zeile 10 wird der Fresnel Term errechnet.

```
1       float3 transmittance;
2       transmittance.r =
3       exp((-Water.absorbtion.r) * Water.density * waterRealDepth * 2.0f);
4
5       transmittance.g =
6       exp((-Water.absorbtion.g) * Water.density * waterRealDepth * 2.0f);
7
8       transmittance.b =
9       exp((-Water.absorbtion.b) * Water.density * waterRealDepth * 2.0f);
10
11      float3 groundColor = refractedColor * transmittance;
12      float3 effectsColor = lerp(reflectedColor, groundColor, fresnelTerm);
13
14      return float4(waterColor + effectsColor, 1.0f);
15  }
```

Die Zeilen 1 bis 9 berechnen die Transmission des Wassers für die einzelnen Farbspektren des Lichts. Zeile 11 berechnet die Farbe des Untergrunds, gesehen durch Wasser. In Zeile 12 wird die Untergrund- und Reflektionsfarbe mit linearer Interpolation zusammengeführt. Zeile 14 berechnet aus der Effektfarbe und der Wasserfarbe die Endfarbe des Wassers am aktuellen Pixel und gibt sie zurück.