Simulation und Visualisierung von Wasseroberflächen

Studienarbeit

Vorgelegt von

Timo Wallrath

UNIVERSITÄT KOBLENZ · LANDAU

Institut für Computervisualistik
Arbeitsgruppe Computergraphik

Betreuer: Dipl.-Inform. Matthias Biedermann
Prüfer: Prof. Dr.-Ing. Stefan Müller

März 2005

Inhaltsverzeichnis

1 Einleitung **1**

2 Der Szenegraph **2**
2.1 OpenSG 2
2.2 Die Szene 4

3 Simulation **5**
3.1 Noise-Funktionen 5
3.2 Einfache Wellenfunktionen 8
3.3 Gerstner-Wellen 8
3.4 Optische Eigenschaften 10

4 Softwaretechnischer Entwurf **13**

5 Implementierung **15**
5.1 Das Mesh 15
5.2 Die Water-Strategies 17
5.2.1 FlatWaterStrategy 20
5.2.2 PerlinNoiseStrategy 21
5.2.3 SumSinusStrategy 22
5.2.4 GerstnerStrategy 23
5.2.5 GerstnerPerlinStrategy 24
5.3 Shader 26
5.3.1 Die allgemeinen Shader 27
5.3.2 Die Shader der GerstnerPerlinStratey 31
5.4 Ereignisbehandlung und die Display-Methode 35

6 Ergebnisse **38**

7 Fazit und Ausblick **42**
7.1 Fazit 42
7.2 Ausblick 42

1 Einleitung

Die realistische Darstellung von Naturphänomenen ist, trotz der immer schneller wachsenden Leistungsfähigkeit von PC-Hardware, nach wie vor eine der größeren Herausforderungen der Computergrafik. Die optischen Eigenschaften, aber vor allem die Dynamik, erschweren speziell die Darstellung von Wasser. Je nach Umgebung verhält es sich zudem völlig unterschiedlich: ein reißender Fluss bietet ein völlig anderes Bild als ein begradigter Kanal, ein offener Ozean oder ein Glas Wasser. Mit der heutigen Hardwaregeneration ist man an einem Punkt, an dem eine ansprechende, realitätsnahe Darstellung von Wasser in Echtzeit möglich ist. So bieten aktuelle, programmierbare Grafikkarten umfangreiche Möglichkeiten in der visuellen Gestaltung von Wassereffekten, die bis vor einigen Jahren noch teurer Spezialhardware vorbehalten waren.

Gegenstand dieser Arbeit ist die Darstellung eines offenen Ozeans. Mit Hilfe des Szenegraph-Systems OpenSG (Kapitel 2.1) soll eine Anwendung, im Folgenden *OceanApp* genannt, erstellt werden, die verschiedene Techniken der Wassersimulation (Kapitel 3) beinhaltet. Zu diesem Zweck muss ein softwaretechnischer Entwurf erstellt werden (Kapitel 4), der die Grundstruktur des Programms festlegt. Zur Umsetzung des Entwurfs ist eine Einarbeitung in C++, OpenSG und die Shadersprache Cg erforderlich, die zur Programmierung der Grafikhardware verwendet wird. Kapitel 5 handelt von verschiedenen Aspekten der Implementierung, während in Kapitel 6 die Ergebnisse der Arbeit anhand illustrierender Screenshots aus *OceanApp* vorgestellt werden. Nach einem kurzen Fazit werden abschließend Möglichkeiten zur Erweiterung der Anwendung diskutiert (Kapitel 7).

2 Der Szenegraph

2.1 OpenSG

Bei OpenSG handelt es sich um ein auf OpenGL basierendes Szenegraph-System, das als Open Source für verschiedene Plattformen (u. a. Linux, Windows) verfügbar ist. Die Bibliothek dient als Schicht zwischen Anwendung und OpenGL-Code und kapselt die meisten OpenGL-Funktionalitäten mit eigenen Methoden ab. Wie die Bezeichnung Szenegraph impliziert, verwaltet OpenSG die darzustellende Szene in einer Graphstruktur, die dann, zur Darstellung optimiert, an OpenGL weiter gereicht wird. So werden unter anderem durch so genanntes *View Frustum Culling* nicht sichtbare Teiläste des Szenegraphen vor der Übergabe an OpenGL aussortiert und die übergebene Geometrie wird so angeordnet, dass die Anzahl der zeitintensiven State-Changes von OpenGL minimiert wird.

Ein OpenSG-Graph besteht in erster Linie aus Knoten (Nodes), die wiederum einen Kern (Core) enthalten. Der Kern legt die Eigenschaften des Knotens fest. Die Topologie des Graphen wird somit durch untereinander verbundene, hierarchisch angeordnete Knoten aufgebaut, während die Cores seine Funktionalität festlegen. Einige der für diese Arbeit wichtigen Core-Typen werden im Folgenden kurz vorgestellt.

- Der einfachste Core-Typ ist der **Group-Core**. Er besitzt keine eigene Funktionalität, sondern fasst lediglich unter ihm liegende Teilbäume zusammen.
- Die **Geometry-Cores** enthalten die Geometrie, also die sichtbaren Elemente der Szene. Neben der Möglichkeit eigene Geometrie zu definieren, enthält OpenSG auch einige einfach zu erstellende Standard-Geometrien wie Kugeln oder Quader.
- **Transform-Cores** beinhalten eine Transformationsmatrix, die auf den darunter liegenden Teilbaum angewendet wird.
- Will man nicht jeder Geometrie einzeln Materialeigenschaften zuweisen, kann man diese Arbeit mit Hilfe einer **Material-Group** bündeln. Alle Geometrien unterhalb eines solchen Knotens werden mit den Materialeigenschaften der Material-Group belegt.

Ein zusätzlicher Knotentyp wird zur Behandlung von Lichtquellen eingeführt: das Beacon. Es werden zwei Knoten benötigt, um eine Lichtquelle in die Szene einzufügen. Ein Knoten mit einem **Light-Core** bestimmt, was von der Lichtquelle beleuchtet wird. Nur die unter dem Lichtknoten hängenden Teiläste werden von diesem auch beleuchtet. Um Position und Ausrichtung des Lichtes festzulegen dient das oben erwähnte Beacon. Dabei handelt es sich um einen Knoten mit einem Transform-Core, dessen Transformationsmatrix die Position des Lichtes festlegt. Analog zum Licht-Beacon wird auch die Positionierung der Kamera über das Beacon-Konzept gelöst.

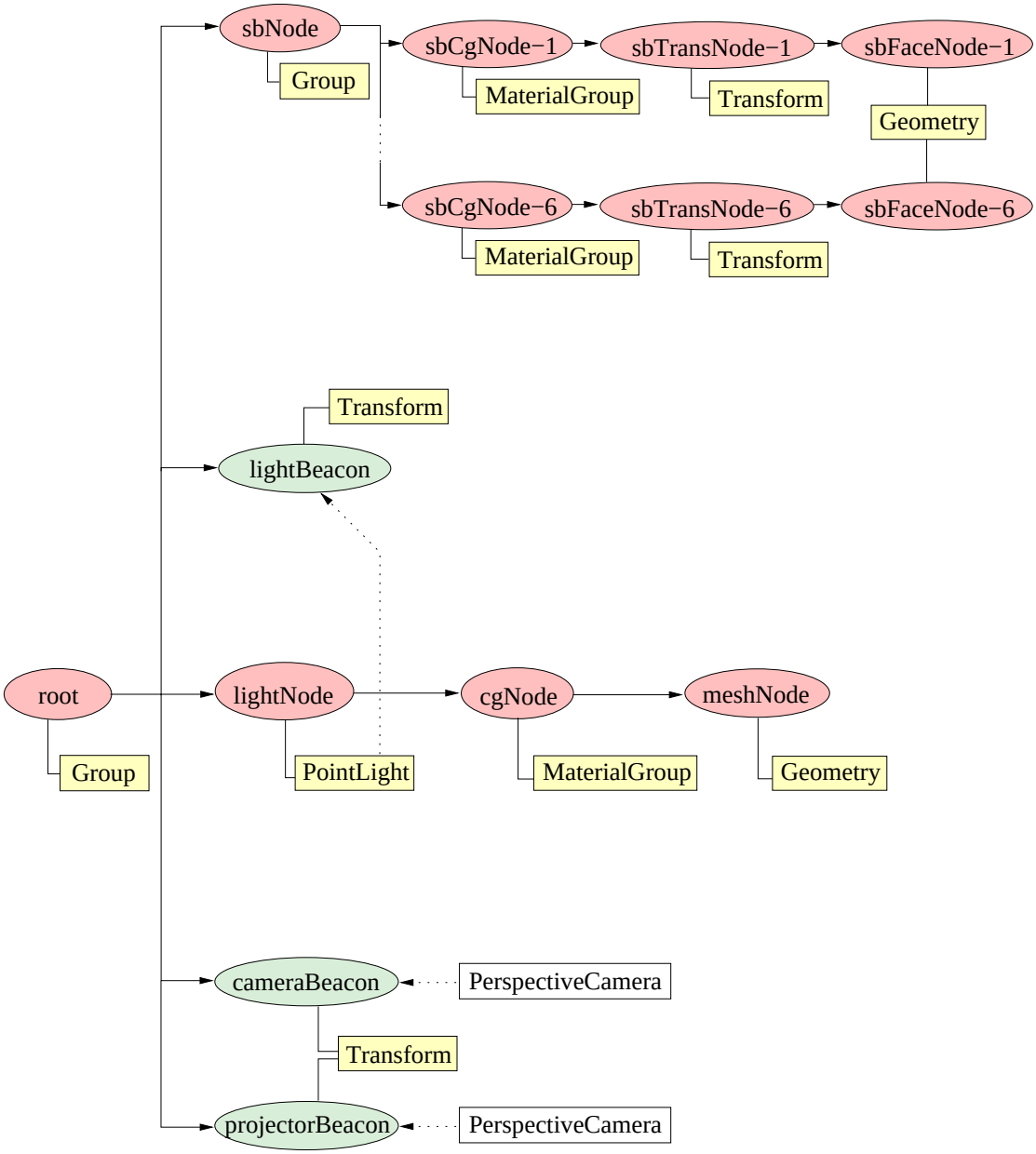

Abbildung 1: Der Szenegraph von *OceanApp*. Nodes sind rot, Beacons grün und Cores gelb gekennzeichnet.

2.2 Die Szene

Abbildung 1 zeigt den Szenegraph von *OceanApp*. Unter dem Wurzelknoten *root*, einem einfachen Gruppenknoten, hängen die beiden Beacons für Kamera und Licht, der Knoten *lightNode*, der für die Beleuchtung der kompletten Szene sorgt, sowie ein weiterer Gruppenknoten. Die Positionierung der Beacons direkt unter dem Wurzelknoten ist üblich, um Licht und Kamera unabhängig von anderen Einflüssen steuern zu können. Die Material-Group des Knotens *cgNode* setzt das Material für die darunter liegende Wassergeometrie. Der Name macht deutlich, dass über die Materialeigenschaft die Verbindung zu den Shader-Programmen hergestellt wird, die Einfluss auf die Wasseroptik haben sollen. OpenSG stellt zu diesem Zweck die Klassen `CGChunk` und `CGParameterChunk` zur Verfügung. Am Ende des Graphen steht der Knoten *meshNode* in dessen Geometry-Core die eigentliche Wassergeometrie aufgebaut wird.

Der Gruppenknoten *sbNode* leitet den für die Skybox zuständigen Teilast des Szenegraphen ein. Eine Skybox trägt entscheidend zum Gesamteindruck der Szene bei, indem sie die Kamera mit einer unendlich weit entfernten Umgebung umgibt. Im Falle von *OceanApp* besteht diese Umgebung nur aus einem leicht bewölkten Himmel (siehe Abbildung 2). Die Skybox besteht aus sechs texturierten Ebenen, die zusammen einen Würfel bilden. Die Kamera befindet sich in der Mitte dieses Würfels, so dass der Abstand zwischen Kamera und den Seitenwänden der Skybox konstant ist. Einer Auf- und Abwärtsbewegung der Kamera kann die Skybox allerdings nicht folgen, da sich in diesem Fall der Wasserspiegel unnatürlich verschieben würde. Aus diesem Grund wurde die maximale Höhe der Kamera begrenzt, um den nötigen Abstand zur oberen Ebene der Skybox zu bewahren.

Die sechs Ebenen der Knoten *sbFaceNode* werden durch darüber liegende Transformationsmatrizen in den Knoten *sbTransNode* zu einem Würfel zusammengesetzt. Ihre Materialeigenschaften erhalten sie durch die Material-Groups der Knoten *sbCGNode*, die entsprechende Shader zur Texturierung der Geometrie aufrufen.

An den Geometrie-Knoten der Skybox lässt sich ein weiteres Konzept von OpenSG aufzeigen: gemeinsame Cores. Um Speicherplatz zu sparen, besitzt jeder *sbFaceNode* den selben Geometry-Core, der lediglich die Grunddefinition einer Ebene beinhaltet. Position und Materialeigenschaften werden erst durch darüberliegende Knoten festgelegt.

Das gleiche Konzept kommt beim letzten Knoten zum Einsatz. Das *projectorBeacon* hat nahezu die gleichen Eigenschaften wie die darstellende Kamera und teilt sich mit ihr den Transform-Core. Somit stimmen auch Position und Ausrichtung mit der Kamera überein. Der Projektor dient der Positionierung der Wasseroberfläche. Auf seine Funktion wird in Kapitel 5.1 genauer eingegangen.

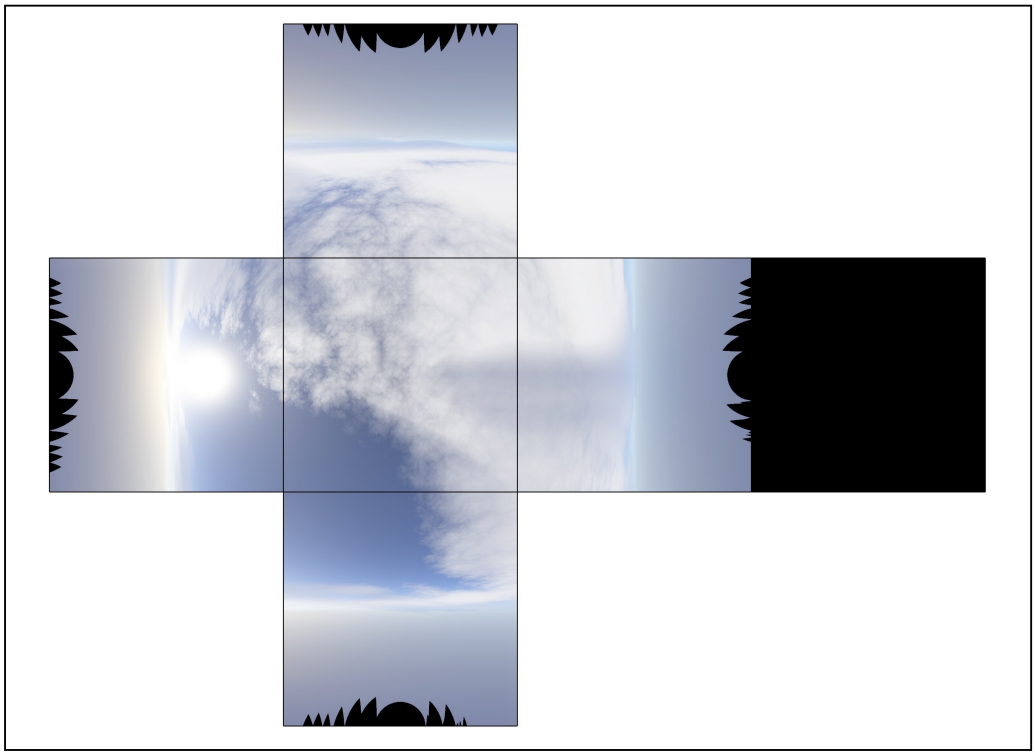

Abbildung 2: Die sechs auseinandergefalteten Seiten, der mit *Terragen* [3] gerenderten Skybox. Die Wasseroberfläche liegt auf mittlerer Höhe der Box, weshalb die untere Würfelhälfte niemals sichtbar ist. Daher wurde auf ihre Berechnung verzichtet, was zu den schwarzen Auswüchsen im unteren Teil des Würfels führt.

3 Simulation

Es existieren verschiedene Ansätze zur Simulation von Wasseroberflächen, wobei man zwischen verschiedenen Wasserformen unterscheiden muss. Ein physikalisch korrektes System aus Navier-Stokes-Gleichungen, mit dem sich das Strömungsverhalten in Flüssigkeiten beschreiben lässt, liefert überzeugende Ergebnisse für ein relativ begrenztes Wasservolumen [7]. Die Simulation eines kompletten Ozeans wäre allerdings mit aktuellen Rechenkapazitäten nicht zu realisieren. Im folgenden werden die in *OceanApp* integrierten Möglichkeiten der Wellenerzeugung vorgestellt und miteinander verglichen.

3.1 Noise-Funktionen

Noise-Funktionen werden häufig verwendet, um Objekte durch eine gewisse Unregelmäßigkeit natürlich wirken zu lassen. Eine spezielle Noise-Funktion ist der so genannte Perlin-Noise [9], der unter anderem zur Terrain-Erzeugung oder Textur-Generierung häufig zum Einsatz kommt. Noise-Funktionen sind sehr flexibel und

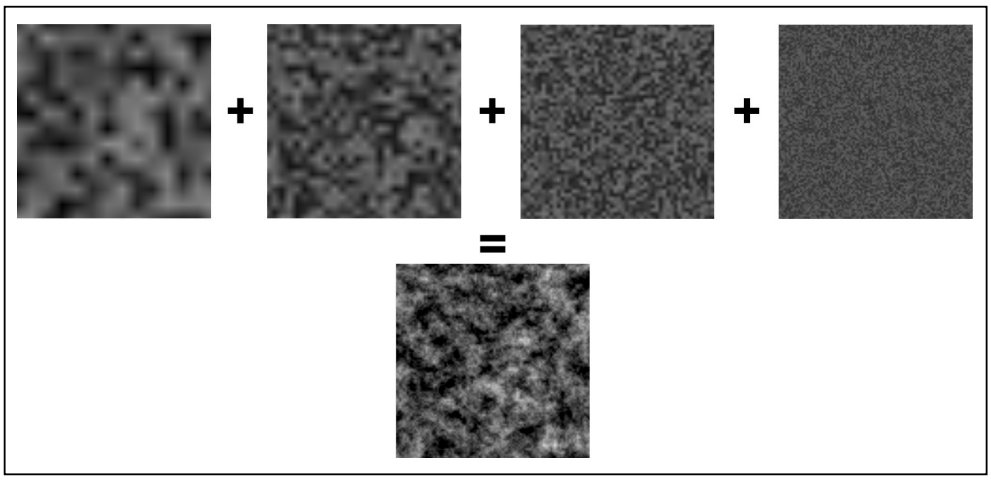

Abbildung 3: Vier Noise-Oktaven und das durch Überlagerung der Oktaven entstandene Ergebnis.

so lassen sich auch Wasserbewegungen mit einem Perlin-Noise darstellen.

Perlin-Noise, benannt nach seinem Erfinder Ken Perlin, besteht aus mehreren übereinander gelagerten, geglätteten Noise-Schichten, so genannten Oktaven, die sich durch Frequenz und Amplitude voneinander unterscheiden. Abbildung 3 zeigt vier Noise-Oktaven und die resultierende Summe.

Die Grundlage des Perlin-Noise bilden Pseudozufallszahlen, die zu einem ganzzahligen Eingabewert ein scheinbar zufälliges, jedoch sicher reproduzierbares, Ergebnis liefern.

$$y = noise(x) \tag{1}$$

Diese Einzelwerte lassen sich durch Interpolation zu einem kontinuierlichen Graphen verbinden. Anstatt einer einfachen linearen Interpolation kann man hierzu eine Sinus- oder Cosinusinterpolation verwenden, die einen guten Kompromiss zwischen Geschwindigkeit und Optik darstellt (siehe Abbildung 4). Somit bilden die reellen Zahlen den Definitionsbereich des Perlin-Noise. Für Eingabewerte x ohne Nachkommastelle ergibt sich das Ergebnis direkt aus dem Noisewert von Funktion 1, für alle anderen x wird das Ergebnis interpoliert.

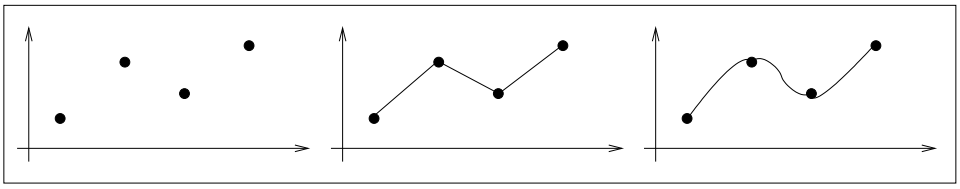

Abbildung 4: Einzelne Rückgabewerte einer eindimensionalen Noise-Funktion (links), interpoliert durch lineare Interpolation (mitte) und eine Cosinusinterpolation (rechts).

Neben der Interpolation lässt sich der Noise durch Glättung weiter visuell aufbereiten. Betrachtet man nicht nur den Funktionswert y, sondern bezieht seine Umgebung mit ein (z.B. Durchschnittswert von y und aller Werte, die an y angrenzen), erscheint der Noise natürlicher und weniger willkürlich.

Um den in Abbildung 3 gezeigten zweidimensionalen Noise zu erzeugen, wird die Noisefunktion um eine Dimension erweitert.

$$y = noise(x,z) \tag{2}$$

Die Interpolations- und Glättungsfunktionen müssen entsprechend angepasst werden. Da Wasser ständig in Bewegung ist, bedarf es eines animierten Noise. Dies kann man erreichen, indem man eine weitere Dimension hinzufügt und dieser eine Zeit t zuweist.

$$y = noise(x,z,t) \tag{3}$$

OceanApp verwendet zur Noise-Erzeugung den von Perlin eingeführten *Improved Perlin Noise* [11]. Als Interpolationsfunktion dient hier die Funktion $6t^5 - 15t^4 + 10t^3$, die gegenüber einer Sinus-/Cosinusfunktion, oder der von Perlin in der Vergangenheit verwendeten kubischen Funktion $3t^2 - 2t^3$, den Vorteil hat, dass sie sowohl in der ersten, als auch in der zweiten Ableitung für $t = 0$ und $t = 1$ eine Nullstelle besitzt. Diese Eigenschaft bietet unter anderem Vorteile beim Einsatz einer so erzeugten Noise-Texture als Bump-Map, da es die Artefaktbildung bei der Darstellung verringert [12].

Die Erzeugung von Noise benötigt relativ viel Rechenzeit und muss bei der Simulation von Wasser in Echtzeit erfolgen. Da es aus Gründen der Performance nicht möglich ist, eine einzige, große Noise-Textur zur Abdeckung der Wasseroberfläche zu erzeugen, muss die Möglichkeit der Kachelung des Noise gegeben sein. Das entsprechende Verfahren wird in [13] genauer beschrieben und erreicht durch mehrmaliges Aufrufen der Noise-Funktion und linearem Interpolieren der Ergebnisse einen Gesamtnoise, der nahtlos in alle Richtungen kachelbar ist. Für einen eindimensionalen Noise sind zwei Aufrufe der Noise-Funktion notwendig, damit die Gesamtfunktion nach einem festgelegten Intervall am Ausgangswert endet. Einen nahtlos kachelbaren, zweidimensionalen Noise, der sich alle w Einheiten in x und alle h Einheiten in y Richtung wiederholt, erhält man durch folgende Gesamtfunktion, die vier Aufrufe der Noise-Funktion beinhaltet:

$$\begin{aligned} noise_{tileable}(x,y) \quad = \quad & \Big(noise(x,y) \cdot (w-x) \cdot (h-y) + \\ & noise(x-w,y) \cdot (x) \cdot (h-y) + \\ & noise(x-w,y-h) \cdot (x) \cdot (y) + \\ & noise(x,y-h) \cdot (w-x) \cdot (y) \Big) \\ & /(wh) \end{aligned} \tag{4}$$

Der Rechenaufwand vervierfacht sich dementsprechend gegenüber einem nicht kachelbarem Noise.

3.2 Einfache Wellenfunktionen

Eine nahe liegende Möglichkeit, Wellen darzustellen, bieten Sinus- oder Cosinusfunktionen. Wählt man die Parameter Wellenlänge, Amplitude und Wellenrichtung mit einer gewissen Varianz um festgelegte Grundwerte und überlagert eine Anzahl dieser Wellen, ergibt sich zumindest ein wasserähnlicher Wellenzug. Folgende Formel ergibt sich für eine Summe von Sinuswellen, die durch das Heigth-Field H visualisiert werden. Das Wasser liegt dabei in Ruhestellung in der x/z-Ebene eines Koordinatensystems.

$$H(x,z,t) = \sum a_i \cdot sin(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i) \tag{5}$$

Für i Wellen ergibt sich so zu jeder Position (x,z), mit der Amplitude a, dem Richtungsvektor D und einer Phasenverschiebung φ ein Höhenwert im Height-Field. D steht dabei parallel zur unbewegten Wasseroberfläche und senkrecht zum zugehörigen Wellenzug und die Phasengeschwindigkeit c, die Frequenz f und die so genannte Wellenzahl k sind definiert durch

$$c = \sqrt{\frac{g \cdot l}{2\pi}} \tag{6}$$

$$f = \frac{c}{l} \tag{7}$$

$$k = \frac{2\pi}{l} \tag{8}$$

mit der Wellenlänge l und der Erdbeschleunigung $g = 9,81 m/s^2$.

Sinuswellen haben allerdings eine sehr runde Form, was für einen ruhigen Wellengang ausreichend sein kann, doch je rauer und aufgewühlter die darzustellende See ist, desto künstlicher und unrealistischer wirken die Sinuswellen.

3.3 Gerstner-Wellen

Eine verwandte Wellenform der in 3.2 beschriebenen trigonometrischen Funktionen sind die Gerstner-Wellen. In der Computergrafik wurden sie erstmals von Fournier und Reeves zur Wasserdarstellung verwendet [4].

Im Gegensatz zu den anderen hier vorgestellten Ansätzen, wird nicht nur die Höhe der Wasseroberfläche berechnet, sondern alle Punkte dieser Oberfläche sind auch in horizontaler Richtung in Bewegung. Zusammengefasst ergibt sich für den Ausgangspunkt an der Position (x_0, z_0):

$$\begin{aligned} x &= x_0 - \sum(a_i \cdot D_i.x \cdot sin(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)) \\ y &= \sum(a_i \cdot cos(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)) \\ z &= z_0 - \sum(a_i \cdot D_i.z \cdot sin(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)) \end{aligned} \tag{9}$$

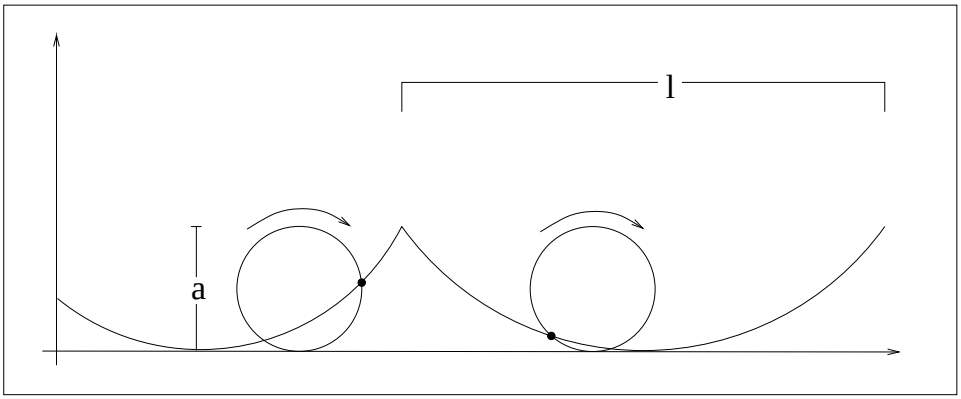

Abbildung 5: Durch kreisförmige Bewegung der Wasserpartikel entsteht eine Welle der Wellenlänge l und mit einer Amplitude a, die dem Kreisdurchmesser entspricht.

Die Berechnung der Wellenhöhe y erfolgt wie schon in 3.2 durch eine trigonometrische Funktion, nur wird dieses Mal analog zu Tessendorf [10] eine Cosinus-Funktion verwendet. Die Sinus-Funktion kommt dafür zur Berechnung der horizontalen Bewegung zum Einsatz, die ansonsten fast identisch mit der Höhenberechnung ist. Neu ist lediglich die jeweilige Komponente des normierten Richtungsvektors D, die für die Bewegung in die entsprechende Richtung sorgt.

Es ergibt sich eine trochoide Wellenform, die einer kreisförmigen Bewegung der Wasserpartikel um ihre Ruheposition entspringt (siehe Abbildung 5). Für kleine Amplituden entspricht die entstehende Welle einer Sinuswelle, doch mit zunehmender Amplitude läuft der Wellenberg immer spitzer zusammen, bis zu dem Punkt, an dem sich die Kurve selbst überschneidet und eine Schleife entsteht. Diese Überschneidung ist physikalisch nicht mehr plausibel und repräsentiert den Punkt, an dem die Welle brechen würde. Das Kriterium für eine brechende Welle ist sehr einfach zu kontrollieren. Ist die Bedingung $k \cdot a > 1$ erfüllt, muss der Brechungsvorgang eingeleitet werden.

Da sich die Wasserpartikel in alle Richtungen bewegen und es zu Überlappungen der Partikel kommen kann, lässt sich die Wassergeometrie nicht mehr durch ein einfaches Height-Field repräsentieren, das darauf ausgelegt ist, zu einer x/z-Position genau einen Höhenwert zurückzuliefern.

Die Bewegung der Partikel ist derart angelegt, dass sie sich zu den Wellenbergen hin zusammenziehen (siehe Abbildung 6). So werden zusätzliche Details dort ermöglicht, wo sie am dringendsten benötigt werden, während sich in den lang gezogenen Wellentälern der Partikelabstand vergrößert.

Die Beschreibung der Wellen durch eine explizite Funktion vereinfacht die Berechnung von Normale, Tangente und Binormale. Durch eine einfache Ableitung der Gerstner-Funktion nach x und z erhält man Binormale und Tangente. Das Kreuzprodukt der beiden Vektoren bildet die Normale der Wasseroberfläche (Glei-

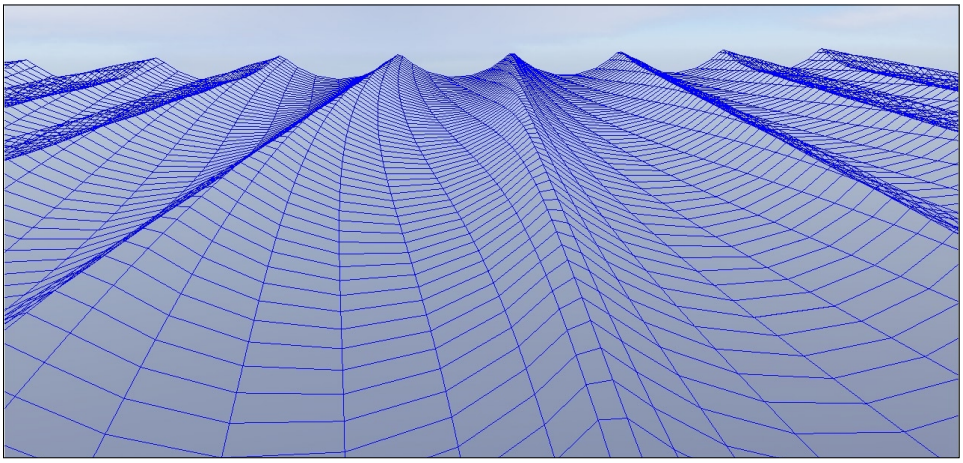

Abbildung 6: Gerstnerwellen in einer Wireframeansicht. Man erkennt deutlich ein Zusammenziehen der Mesh-Punkte am Wellenberg.

chungen 10-12).

$$B = \begin{pmatrix} 1 - \sum(a_i \cdot D_i.x^2 \cdot k_i \cdot cos(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)), \\ -\sum(a_i \cdot D_i.x \cdot k_i \cdot sin(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)), \\ -\sum(a_i \cdot D_i.x \cdot D_i.z \cdot k_i \cdot cos(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)) \end{pmatrix} \tag{10}$$

$$T = \begin{pmatrix} -\sum(a_i \cdot D_i.x \cdot D_i.z \cdot k_i \cdot cos(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)), \\ -\sum(a_i \cdot D_i.z \cdot k_i \cdot sin(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)), \\ 1 - \sum(a_i \cdot D_i.z^2 \cdot k_i \cdot cos(k_i \cdot D_i \circ (x,z) - f_i \cdot t \cdot c_i + \varphi_i)) \end{pmatrix} \tag{11}$$

$$N = T \times B \tag{12}$$

3.4 Optische Eigenschaften

Neben den oben beschriebenen Möglichkeiten der Wellenerzeugung, wird das Erscheinungsbild von Wasser durch seine speziellen optische Eigenschaften geprägt.

Auf eine Wasseroberfläche auftreffendes Licht wird zu bestimmten Teilen reflektiert und transmittiert (siehe Abbildung 7). Während reflektierte Strahlen für die Spiegelung der Umgebung auf der Wasseroberfläche sorgen, unterliegen die transmittierten Strahlen dem Gesetz der Refraktion (Brechung). Da sich die Brechungsindizes von Wasser ($n_w = 1,33$) und Luft ($n_l = 1$) unterscheiden, kommt es beim Eintritt in das Wasser zu einer Richtungsänderung des Lichtsrahls. Wie groß diese Ablenkung ausfällt, lässt sich durch das so genannte Snelliussche Brechungsgesetz (oft auch als Snells Gesetz bezeichnet) berechnen [2]:

$$n_1 \cdot \sin(\Theta_1) = n_2 \cdot \sin(\Theta_2) \tag{13}$$

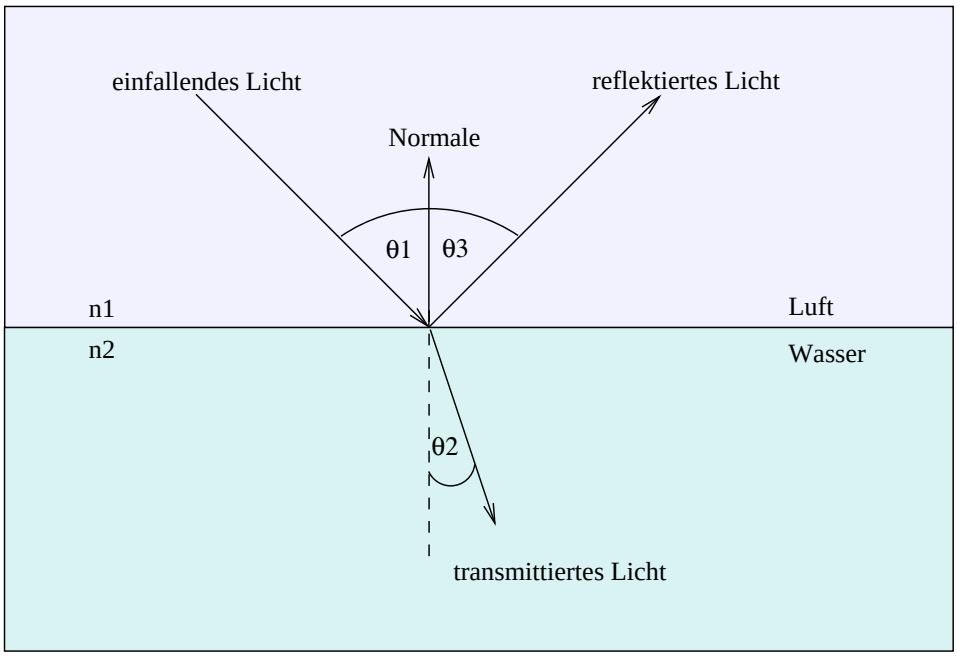

Abbildung 7: Ein unter dem Winkel Θ_1 einfallender Lichtstrahl, transmittiert unter dem Winkel Θ_2 in das Wasser und wird unter dem Ausfallswinkel Θ_3 von der Oberfläche reflektiert. n_1 und n_2 bezeichnen die Brechungsindizes von Luft und Wasser.

Snells Gesetz gibt dabei lediglich an, wie stark ein transmittierter Strahl abgelenkt wird. Das Verhältnis zwischen transmittiertem und reflektiertem Licht erhält man mit Hilfe der Fresnel-Formel [1]. Der Reflexionsgrad von unpolarisiertem Licht für einen Einfallswinkel Θ_1 und einen Transmissionswinkel Θ_2 errechnet sich demnach durch:

$$r = \left(\frac{\sin(\Theta_1 - \Theta_2)}{\sin(\Theta_1 + \Theta_2)}\right)^2 + \left(\frac{\tan(\Theta_1 - \Theta_2)}{\tan(\Theta_1 + \Theta_2)}\right)^2 \tag{14}$$

Der Grad der Transmission ergibt sich aus:

$$t = (1 - r) \tag{15}$$

Blickt man senkrecht auf die Wasseroberfläche ist der Reflexionsgrad Null und das Wasser verhält sich komplett transparent. Je weiter der Sichtstrahl jedoch von der Oberflächennormale abweicht, desto stärker reflektiert die Wasseroberfläche, bis sie bei besonders flachen Blickwinkeln ($\Theta_1 \approx 90°$) einem perfekten Spiegel gleicht. Da der Berechnungsaufwand des Reflexionsgrades relativ hoch ist, wird er in der Implementierung durch einen approximierenden Term ersetzt. Details hierzu finden sich in Kapitel 5.3.

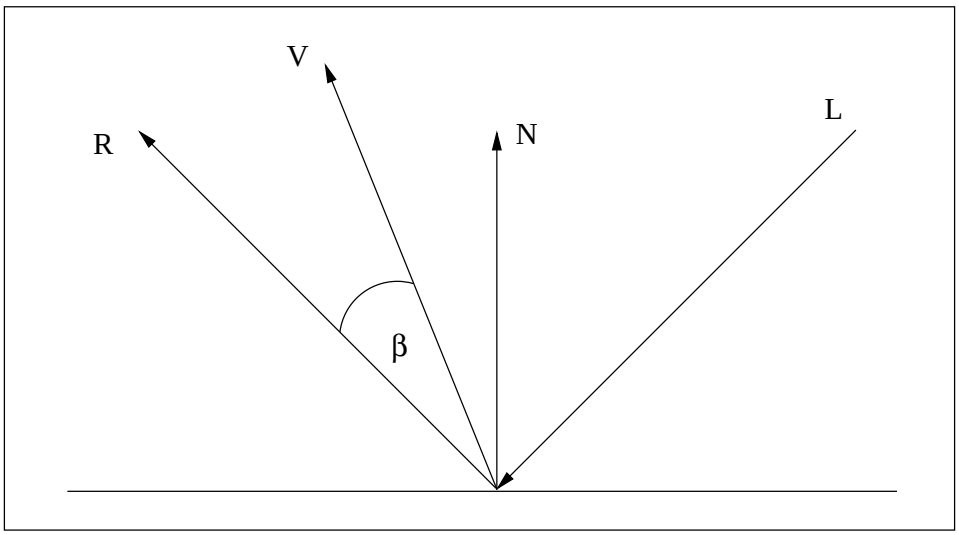

Abbildung 8: Die normalisierten Vektoren zur Berechnung der spekularen Komponente des Phong-Models: Der einfallende Lichtvektor L, die Reflexion R, der Blickvektor der Kamera V, sowie die Oberflächennormale N.

Zur Berechnung der Reflexion dient die allgemeine Reflexionsformel, die für eine Oberflächennormale und einen Vektor vom Reflexionspunkt zur Kamera folgendes Ergebnis liefert:

$$R = 2(E \circ N)N - E \tag{16}$$

OceanApp verwendet den Ergebnisvektor R für ein Lookup in einer Cube-Texture, die identisch mit der Skybox ist. So wird eine Spiegelung der, durch die Skybox dargestellten, Umgebung auf der Wasseroberfläche erreicht.

Im Gegensatz zur Reflexion kann auf eine Berechnung des Transmissionsvektors verzichtet werden. Da große Wassertiefen simuliert werden, ist ein Meeresgrund, auf den ein transmittierter Strahl treffen könnte, nicht sichtbar. Refraktionseffekte wären somit auf lokale Unterwasserobjekte, wie Fische, beschränkt, die in *OceanApp* nicht berücksichtigt werden. Anstatt über einen Transmissionvektor einen Farbwert für nicht reflektierende Lichtstrahlen zu bestimmen, wird ein konstanter Farbwert, die Grundfarbe des Wassers, festgelegt.

Ein weiterer optischer Effekt ist die direkte Reflexion von Sonnenlicht auf der Wasseroberfläche. Eine Möglichkeit zur Darstellung der so entstehenden Glanzlichter ist die Verwendung der spekularen Komponente des Phong-Shading-Models. Abbildung 8 zeigt die an der Berechnung beteiligten Vektoren. Je kleiner der abgebildete Vektor β ausfällt, desto mehr Licht wird direkt in die Kamera reflektiert und desto heller erscheint das entstehende Glanzlicht. Der Winkel ergibt sich aus dem Skalarprodukt der beiden einschließenden Vektoren. Die Gesamtformel zur Berechnung der spekularen Komponente lautet:

$$specular = K_s \cdot \text{lightColor} \cdot (R \circ V)^n \tag{17}$$

mit der Materialfarbe K_s und der Farbe des Lichtes *lightColor*. Der Exponent n steuert den Glanz des Objektes. Ein niedriger Exponent sorgt für ein größeres, diffuseres Glanzlicht, wohingegen ein hoher Wert für n ein kleineres, scharfes Glanzlicht entstehen lässt.

4 Softwaretechnischer Entwurf

Vor der eigentlichen Implementierung wurde ein softwaretechnischer Entwurf erstellt, der die beteiligten Klassen und deren Beziehungen zueinander festlegt. Ziel ist ein objektorientiertes Design, das zum einen die Elemente des Szenegraphen (siehe Abbildung 1) umsetzt und zum anderen eine Struktur schafft, die eine einfache Implementierung der in Kapitel 3 vorgestellten Simulationsformen ermöglicht. Außerdem wird ein Wechsel der Simulationsalgorithmen zur Laufzeit möglich sein. Abbildung 9 zeigt den Grundaufbau von *OceanApp*.

Die Klasse `WaterApplication` bildet das Zentrum von OceanApp. Sie enthält Referenzen zu den anderen beteiligten Klassen und ist für deren Initialisierung verantwortlich. `WaterApplication` wurde nach dem Singleton-Pattern konzipiert [8], so dass sichergestellt ist, dass genau eine Instanz der Klasse existiert.

`WaterScene` ist für die Abbildung des Szenegraphen verantwortlich und die Knoten des Graphen aus Abbildung 1 finden hier ihre Entsprechung. Mit den von OpenSG zur Verfügung gestellten Cg-Datentypen stellt sie außerdem die Schnittstelle zu den Shadern der Grafikkarte dar und sorgt so für die Parameterübergabe zwischen *OceanApp* und den Shaderprogrammen.

Die Klasse `MeshData` dient als Datenstruktur zur Speicherung und Verwaltung der Wasseroberfläche. Sie speichert die 3D-Koordinaten der Oberflächenpunkte und aktualisiert deren Position in Abhängigkeit vom Blickfeld der Kamera. Einzelheiten zur Implementierung der Klasse finden sich in Kapitel 5.1.

Ein weiteres Design Pattern kam beim Entwurf der `WaterStrategy`-Klassen zum Einsatz, die für die Umsetzung der in Kapitel 3 beschriebenen Möglichkeiten der Wassersimulation verantwortlich sind. Die Anwendung des in Abbildung 10 dargestellten Strategy-Patterns liefert eine klar definierte, einheitliche Schnittstelle für alle Simulationsstrategien und ermöglicht den Wechsel zwischen ihnen zur Laufzeit. Die Klasse `Strategy` definiert hierbei die gemeinsame Schnittstelle aller Strategien. Die von ihr abgeleiteten `ConcreteStrategy`-Klassen bedienen sich dieser Schnittstelle und implementieren die eigentlichen Strategien zur Wassersimulation. In Kapitel 5.2 werden die Implementierung des Patterns sowie alle implementierten Water-Strategies genauer beschrieben.

Die Klasse `ParameterContainer` dient, wie der Name schon andeutet, als Containerklasse für diverse Parameter, die anwendungsweit zur Verfügung stehen. Durch die Abkapselung in ein Objekt wird die objektorientierte Struktur gewahrt

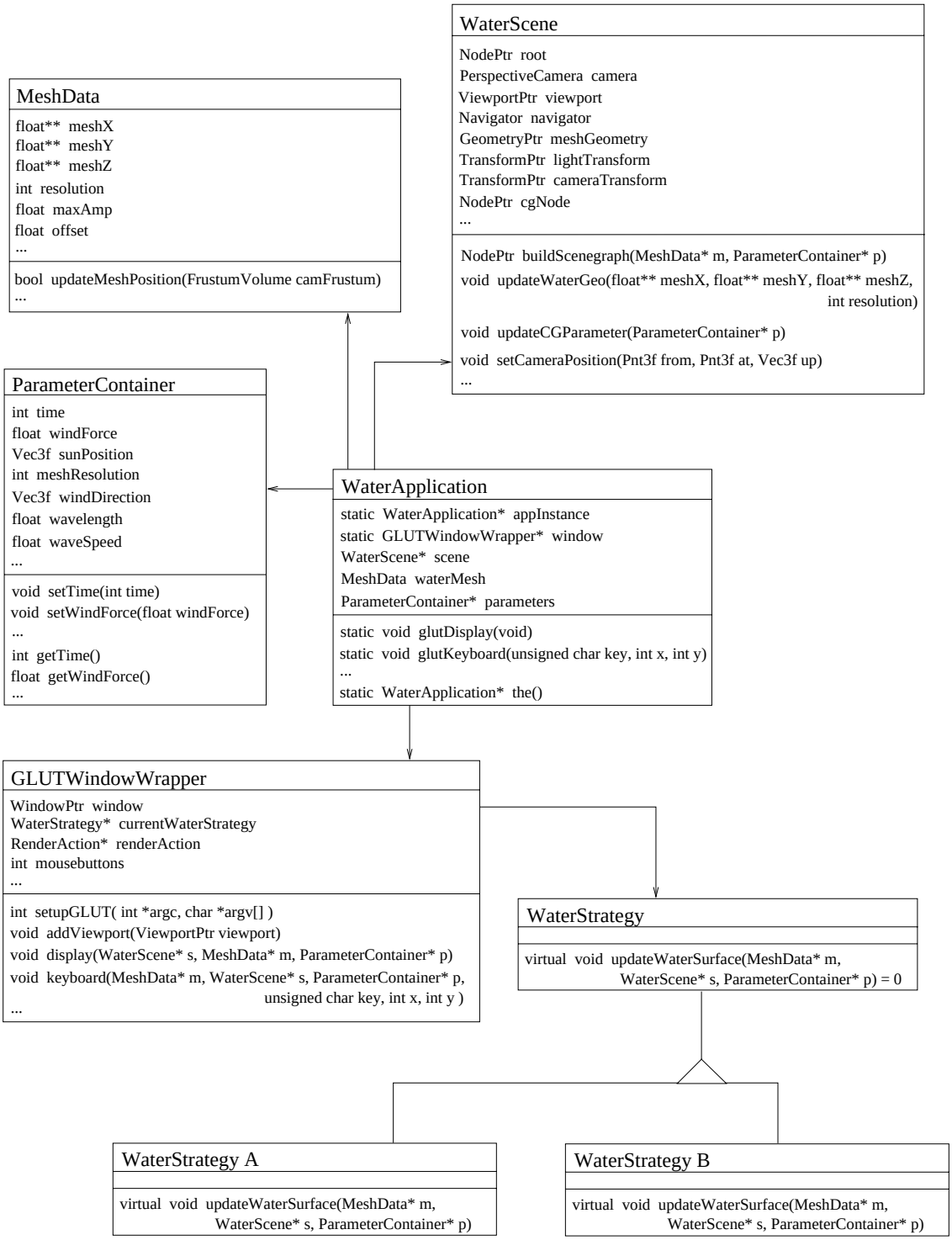

Abbildung 9: Das Klassendiagramm

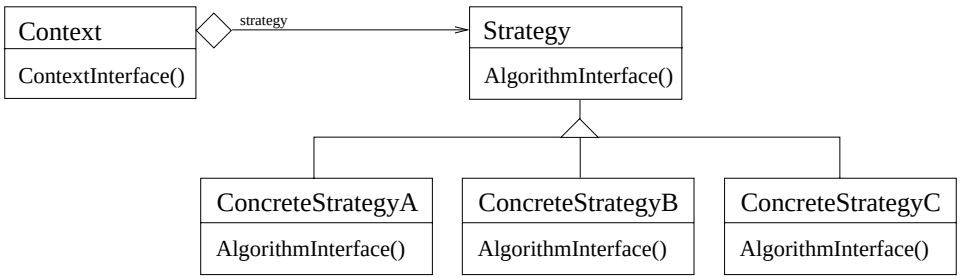

Abbildung 10: Das Strategy-Pattern

und der Austausch der Parameter zwischen den verschiedenen Klassen wird vereinfacht. Um auf die im Container gehaltenen Parameter zugreifen zu können, reicht die Übergabe eines Zeigers auf ein Objekt der Klasse `ParameterContainer`.

5 Implementierung

5.1 Das Mesh

In diesem Abschnitt wird die Datenstruktur des eigentlichen Wassers näher erläutert. Das Wasser wird durch ein Netz repräsentiert, dessen Knotenpunkte die einzelnen Vertices der Wassergeometrie bilden. Die dafür zuständige Klasse `MeshData` speichert und verwaltet diese Knotenpunkte und ist nicht mit der eigentlichen Geometrie zu verwechseln, die als Knoten im Szenegraph hängt und letztendlich dargestellt wird. `MeshData` aktualisiert diese Geometrie auf Framebasis und sorgt so für die korrekte Lage des Wasser-Meshs.

Die 3D-Punkte dieses Meshs werden in drei zweidimensionalen Arrays gehalten, jeweils eines für die x-, y- und z-Komponente des 3D-Punktes. Die Simulation eines offenen Ozeans mit seiner unüberschaubar großen Wasseroberfläche macht den Einsatz eines Meshs mit fest definierter Position und Größe unmöglich. Stattdessen wird ein dynamisches Mesh generiert, das seine Position und Ausdehnung in jedem Frame an Kameraposition und Blickrichtung des Benutzers anpasst. Diese Aufgabe übernimmt die Methode

```
bool updateMeshPosition(Frustum frustum),
```

der als Eingabewert ein Kamerafrustum dient.

Die Methode definiert zunächst zwei Hilfsebenen, die jeweils parallel zur Basisebene liegen, welche der Wasseroberfläche in Ruhestellung entspricht. Eine der Hilfsflächen liegt über dieser Ebene, die andere darunter und ihr Abstand beträgt jeweils die maximale Amplitude der Wellenbewegung (siehe Abbildung 11). Durch die Ebenen wird ein vertikaler Sicherheitsabstand definiert, der verhindern soll, dass die Ränder des Meshs ins Sichtfeld der Kamera gelangen, sobald es in Bewegung gesetzt wird. Durch den folgenden Ablauf werden Lage und Ausdehnung des Meshs bestimmt:

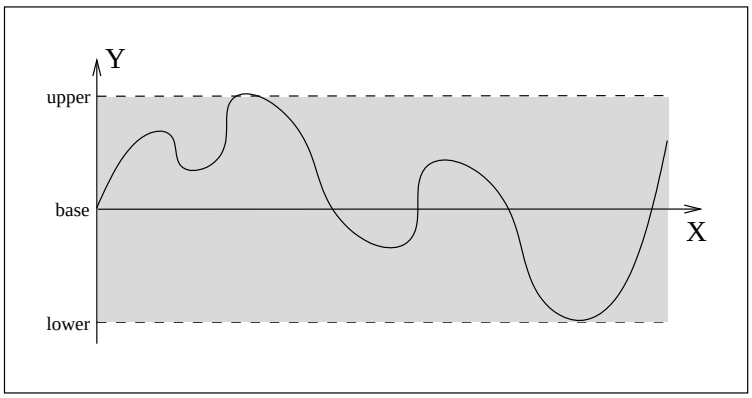

Abbildung 11: Die zwei begrenzenden Hilfsebenen *lower* und *upper* sorgen dafür, dass die Ränder des Wasser-Meshs bei Wellengang nicht ins Blickfeld gelangen. Die *base*-Ebene liegt auf der Höhe des ruhenden Wassers ($y = 0$).

1. Führe einen Schnitttest zwischen den acht Kanten des View-Frustums und den drei Ebenen durch.
2. Bestimme, ob Eckpunkte des Frustums im Raum zwischen den Ebenen liegen und füge diese der Menge der Schnittpunkte hinzu.
3. Ist die Menge der Schnittpunkte leer, ist das Mesh nicht sichtbar und muss nicht gerendert werden. Die Methode wird verlassen und liefert als Rückgabewert `false` zurück.
4. Projiziere alle Schnittpunkte auf die Basisebene (siehe Abbildung 12).
5. Bestimme Minimum und Maximum der Schnittpunkte in x- und z-Richtung. Die gefundenen Werte legen die Grenzen des Meshs fest.
6. Dehne das Mesh durch Addition/Subtraktion eines Offset-Wertes in alle Richtungen aus.
7. Durch Interpolation zwischen den neuen Minimal- und Maximalwerten wird das Mesh aufgespannt.

Der Offset-Wert aus Punkt 6 dient dem gleichen Zweck wie die Definition der zwei Hilfsebenen. Während die Ebenen einen vertikalen Puffer definieren, sichert der Offset-Wert das Mesh gegen horizontale Bewegungen der Vertices ab, wie sie beispielsweise bei Gerstner-Wellen (siehe Kapitel 3.3) entstehen. Er muss daher nur aufaddiert werden, wenn eine solche Bewegung möglich ist. Der Offset-Wert kann mit der Methode

```
void setSafetyOffset(float offset)
```

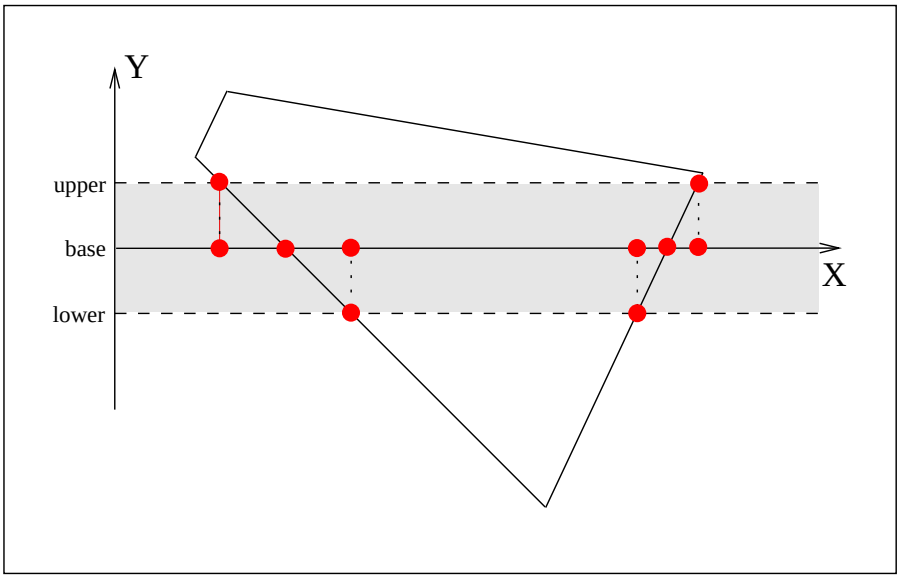

Abbildung 12: Zweidimensionale Ansicht des Schnitttests zwischen Frustum und den drei Ebenen, mit anschließender Projektion der Schnittpunkte auf die Basisebene.

gesetzt werden. Für Gestner-Wellen bietet sich die Summe der Wellenamplituden an, da diese ebenfalls den maximalen Ausschlag in horizontaler Richtung festlegt.

Es liegt nahe, als Eingabefrustum der Update-Methode das Frustum der aktuellen Kamera zu verwenden, doch würde dies zu einem sehr unvorteilhaften Mesh führen. Um die weit entfernte Skybox-Geometrie der Szene darstellen zu können, muss der Abstand der Far-Plane des Kamerafrustums entsprechend hoch angesetzt werden. Lage und Ausdehnung des Meshs hängen allerdings direkt von der Ausdehnung des Eingabefrustums ab, so dass für eine Kameraausrichtung, bei der die Far-Plane des Frustums eine Begrenzung des Meshs definiert (z.B. Blick in Richtung Horizont), ein sehr großes und somit schlecht aufgelöstes Mesh entstehen würde. Die Lösung dieses Problems ist der bereits in Kapitel 2.2 erwähnte Projektor. Es handelt sich hierbei um ein Kameraobjekt, das sich lediglich in einem Parameter von der eigentlichen Kamera unterscheidet: die Far-Plane des Projektors liegt deutlich näher am Betrachter. Das View-Frustum des Projektors dient nun als Eingabefrustum der Update-Methode und sorgt für eine sinnvolle Größe und Auflösung des Meshs.

Abbildung 13 zeigt den Gesamtaufbau der Klasse `MeshData`.

5.2 Die Water-Strategies

Wie in Kapitel 4 beschrieben, wurden die verschiedenen Algorithmen zur Wassersimulation unter Verwendung des Strategy-Patterns implementiert. Die dadurch gewonnene Möglichkeit, die aktuelle Strategie zur Laufzeit ändern zu können, wird

```
class MeshData{

    const int MAX_MESH_RESOLUTION;
    const int DEFAULT_MESH_RESOLUTION;

public:
    MeshData(int resolution = 0);
    ~MeshData();

    int getResolution() const;

    float** getMeshX() const;
    float** getMeshY() const;
    float** getMeshZ() const;

    float getStepsizeRatioX() const;
    float getStepsizeRatioZ() const;

    void setMeshX(int i, int j, float value);
    void setMeshY(int i, int j, float value);
    void setMeshZ(int i, int j, float value);

    float getMaxAmp() const;

    void setResolution(int resolution);
    void setMaxAmp(float amp);
    void setSafetyOffset(float offset);

    bool updateMeshPosition(FrustumVolume camFrustum);

private:
    int _resolution;     // resolution of the mesh

    float** _meshX;      // two-dimensional arrays
    float** _meshY;      // to store the x/y/z-comonent
    float** _meshZ;      // of the mesh-vertices

    float _maxAmp;       // maximum amplitude of waves

    float _offset;       // safety offset, if vertices are
                         // oscillating in x/z-direction

    float _initialStepsizeX;     // stepsize to interpolate the
    float _initialStepsizeZ;     // vertices from intersection
    float _stepsizeRatioX;       // points. Is needed to
    float _stepsizeRatioZ;       // scale texture-coordinates.
};
```

Abbildung 13: Die Klasse MeshData.

durch das gemeinsame Interface aller Strategy-Klassen realisiert. Definiert wird dieses Interface durch die Klasse `WaterStrategy` mit ihrer virtuellen Methode `updateWaterSurface` (siehe Abbildung 14). Jede konkrete Strategy-Klasse muss von ihr abgeleitet sein und die virtuelle Methode implementieren.

```
/**
 * Strategy-class of the strategy pattern.
 * Provides an interface for the concrete strategies,
 * which provide different algorithms for creating waves.
 */
class WaterStrategy{
public:

    virtual void updateWaterSurface(MeshData* mesh,
                                    WaterScene* scene,
                                    ParameterContainer* par) = 0;

protected:
    WaterStrategy();
};
```

Abbildung 14: Die Klasse WaterStrategy

Das Null-Setzen der Methode kennzeichnet sie als rein virtuell. Sie wird in `WaterStrategy` lediglich deklariert, eine Implementierung in den abgeleiteten Klassen ist somit zwingend erforderlich.

Der Hauptvorteil des Strategy-Patterns, das Wechseln der Strategien zur Laufzeit, kommt in der Klasse `GLUTWindowWrapper` zum Tragen, die als Membervariable einen Zeiger vom Typ der Basisklasse auf die aktuelle Strategie hält:

```
WaterStrategy* _currentWaterStrategy;
```

Im Konstruktor wird der Zeiger auf eine Instanz einer konkreten Strategy-Klasse gelegt, welche die virtuelle Methode implementiert und so das Wasserverhalten festlegt.

```
_currentWaterStrategy = new FlatWaterStrategy();
```

Zum Einsatz kommt die aktuelle Strategie beim Aktualisieren der Wasseroberfläche. Einmal pro Frame wird sie über das durch die Basisklasse definierte Interface aufgerufen.

```
_currentWaterStrategy->updateWaterSurface(mesh, scene, par);
```

Die Entscheidung, auf welchen Objekttyp `_currentWaterStrategy` zeigt, fällt dabei nicht zur Compile-, sondern zur Laufzeit. Ein Wechsel der Strategie ist somit jederzeit möglich. Zuerst wird der Speicherplatz des alten Strategy-Objektes freigegeben, um dann den Zeiger, der nach wie vor vom Typ WaterStrategy ist, auf ein Objekt der neuen Klasse zu legen.

```
delete _currentWaterStrategy;
_currentWaterStrategy = new PerlinNoiseStrategy();
```

Durch das von `WaterStrategy` geerbte, gemeinsame Interface ändert sich am Aufruf der Update-Methode nichts und ein Wechsel der Strategie beschränkt sich auf den Austausch des Strategie-Objektes.

Die folgenden Abschnitte beschreiben die Funktionalität der einzelnen Strategie-Klassen.

5.2.1 FlatWaterStrategy

Die FlatWaterStrategy dient lediglich dem Zurücksetzen der Wassergeometrie auf den Ausgangslevel. Die Update-Methode läuft in einer verschachtelten for-Schleife über alle Punkte des Meshs und setzt ihre y-Komponenten auf Null. Danach wird die eigentliche Wassergeometrie der Szene anhand der neuen Mesh-Daten aktualisiert.

```
void FlatWaterStrategy::updateWaterSurface(MeshData* mesh,
                                           WaterScene* scene,
                                           ParameterContainer* par)
{   // get resolution of the mesh
    int resolution = mesh->getResolution();

    // set the surface to base-level
    for (int i=0; i < resolution; i++)
        for (int j=0; j < resolution; j++)
            mesh->setMeshY(i, j, 0.0f);

    scene->updateWaterGeo( mesh->getMeshX(), mesh->getMeshY(),
                           mesh->getMeshZ(), resolution );
    scene->updateNormals();
}
```

Abbildung 15: Die Update-Methode der FlatWater-Strategie.

Anhand dieser, in Abbildung 15 dargestellten, einfachen Methode lässt sich ein struktureller Nachteil des Strategy-Patterns aufzeigen. Obwohl der Parameter-Container innerhalb der Methode nicht verwendet wird, muss er trotzdem mit übergeben werden, da das Interface der Basisklasse dies verlangt. Andere Strategien könnten Parameter aus dem Container benötigen, und das Interface muss, da es für alle Fälle allgemein definiert ist, alle Möglichkeiten berücksichtigen. Es kommt zu einem Daten-Overhead, der in diesem Fall allerdings noch unkritisch ist, da es sich zum einen nur um einen Zeiger auf ein schon bestehendes Objekt handelt und zum anderen die FlatWaterStrategy die Ausnahme bildet – alle anderen Strategien bedienen sich Werten des Parameter-Containers.

Die letzten beiden Aufrufe von Methoden der Klasse `WaterScene` schließen jede Update-Methode einer Strategie ab. Sie übertragen die aktualisierten Mesh-Daten auf die Wassergeometrie des Szenegraphen und aktualisieren die dazuge-

hörigen Normalen. Ausnahmen sind Strategien, die Normalen der Wasseroberfläche komplett durch ein Shader-Programm berechnen lassen. Dann kann auf den entsprechenden Methodenaufruf verzichtet werden. Ein Beispiel für eine solche Strategie ist die GerstnerPerlinStrategy aus Kapitel 5.2.5.

5.2.2 PerlinNoiseStrategy

Die PerlinNoiseStrategy verwendet den in Kapitel 3.1 beschriebenen Perlin-Noise zur Manipulation des Wassers auf Vertex-Ebene. Den Noise erzeugt ein Objekt der Klasse `ImprovedPerlinNoise`, dessen Methode

```
static float noise(float x, float y, float z);
```

einen dreidimensionalen Perlin-Noise realisiert. Die PerlinNoiseStrategy instanziiert dazu im Konstruktor ein Objekt dieser Klasse und weist es der dafür vorgesehenen Membervariable zu, um in der Update-Methode darauf zugreifen zu können.

```
PerlinNoiseStrategy::PerlinNoiseStrategy()
     : WaterStrategy()
{
   _noise = ImprovedNoiseStrategy();
}
```

Die Auflösung des Noise entspricht der Mesh-Auflösung, und als Eingabewerte der Noisefunktion dienen die x-/z-Koordinaten der Mesh-Knotenpunkte, sowie die Zeit als dritte Dimension. In einer Schleife über alle Punkte werden in einer inneren Schleife alle Noise-Oktaven aufsummiert. Das Ergebnis stellt direkt den Höhenwert des aktuellen Punktes dar und wird diesem zugewiesen. Folgender Pseudo-Code aus der Update-Methode der PerlinNoiseStrategy soll den Ablauf verdeutlichen:

```
Für alle Punkte (x, y, z) des Wasser-Meshs
{
   Noise      = 0;
   Persistenz = p;

   Für alle Noise-Oktaven i
   {
      Noise = Noise + noise(x * Frequenz,
                            z * Frequenz,
                            Zeit) * Amplitude
      Frequenz  = 2^i
      Amplitude = Persistenz^i
   }
```

```
   y = Noise
}
```

Der Begriff der Persistenz gibt im Zusammenhang des Perlin-Noise den Einfluss an, den höhere Frequenzen auf den Gesamtnoise ausüben. Er wird üblicherweise mit einem Wert ≤ 1 belegt, so dass die Amplituden höherer Frequenzen abgeschwächt werden. *OceanApp* verwendet standardmäßig einen Persistenzwert von $p = 0,75$.

5.2.3 SumSinusStrategy

Wellen durch die in Kapitel 3.2 beschriebenen einfachen Wellenfunktionen werden durch die SumSinusStrategy implementiert. Die Methode summiert eine vom Benutzer festgelegte Anzahl von Sinuswellen, die sich in den Parametern Wellenlänge, Amplitude, Richtung und Phase unterscheiden. Die Abweichungen liegen in einem gewissen Intervall um einen gegebenen Richtwert, der aus dem Parameter-Container bezogen wird. Die Initialisierung der Wellenparameter findet in der Methode

```
void SumSinusStrategy::initWaveParameters(int index,
                                          ParameterContainer* par);
```

statt, mit `index` als Index der zu bearbeitenden Welle. Die Aufgabe der Update-Methode besteht aus zwei Teilen. Zuerst wird geprüft, ob die Lebenszeit einer Welle abgelaufen ist. Um das Erscheinungsbild des Wassers dynamischer und abwechslungsreicher zu gestalten, besitzt jeder Wellenzug als weiteren Parameter einen Todeszeitpunkt, der ihm bei der Initialisierung zugewiesen wird. Ist dieser Zeitpunkt erreicht, wird die Welle langsam ausgeblendet, indem ihre Amplitude pro Frame um einen kleinen Betrag verkleinert wird. Fällt die Amplitude unter Null, ist die Welle ausgeblendet und obige Init-Methode wird mit ihrem Index aufgerufen, um sie mit neuen Werten zu initialisieren. Um einen sauberen Übergang zu erreichen, wird die neue Welle nicht abrupt „eingeschaltet“, sondern mit der gleichen Amplitudentechnik des Ausblendens sanft eingeblendet. Aus diesem Grund hält die Strategie zwei Amplitudenwerte pro Welle: die aktuelle Amplitude, sowie die zu erreichende Ziel-Amplitude.

Die zweite Aufgabe der Update-Methode ist die eigentliche Wellenerzeugung. Wie in den vorangegangenen Strategien läuft eine Schleife über alle Mesh-Punkte. Eine innere Schleife läuft über alle Wellen, liest ihre Parameter aus, berechnet die darauf aufbauenden Größen (Gleichungen 6-8) und summiert dann alle Sinuswellen nach Gleichung 5. Der zusätzlich eingeführte Faktor $\frac{1}{500}$ verlangsamt dabei die Wellenbewegung, die andernfalls unrealistisch schnell ablaufen würde.

```
for (int index=0; index < _numberOfWaves; index++){

   Vec2f dir = /* Richtungsvektor (x,z) der aktuellen Welle */
```

```
   Vec2f pos = /* aktueller Mesh-Punkt (x,z) */

   float c = sqrt( (9.81 * _wavelength[index]) / (2*Pi) );
   float f = c / _wavelength[index];
   float k = 2*Pi / _wavelength[index];

   y += _amplitude[index] * sin( (dir*k).dot(pos) - f * time
                                 * (c/500.f) + _phase[index]);
}
```

Es folgt das Setzen des in Kapitel 5.1 erläuterten Sicherheitsabstandes, um das Sichtbarwerden des Meshrandes zu verhindern.

```
if (maxAmp < abs(y))
   maxAmp = abs(y);

[...]
mesh->setMaxAmp(maxAmp);
```

5.2.4 GerstnerStrategy

Die GerstnerStrategy implementiert Gerstner-Wellen aus Kapitel 3.3 und ist zu einem großen Teil identisch mit der SumSinusStrategy. Die Wellen werden mit den gleichen Parametern initialisiert und ein-/ausgeblendet. Es bestehen jedoch zwei Unterschiede: dem Gerstner-Modell folgend sind die Mesh-Punkte nicht nur in der Höhe variabel, sondern schwingen auch in horizontaler Richtung. Nach Gleichung 9 gibt es demnach statt einer, drei summierende Gleichungen in der inneren Schleife über alle Wellenzüge:

```
for (int index=0; index < _numberOfWaves; index++){

   [...] /* Berechnung der Parameter - siehe SumSinStrategy */

   sincos_arg = (dir*k).dot(pos) - f * t * (c / 500.f)
                                            + _phase[index];

   x += _amplitude[index] * dir[0] * sin(sincos_arg);
   y += _amplitude[index] * cos(sincos_arg);
   z += _amplitude[index] * dir[1] * sin(sincos_arg);
}
```

Der zweite Unterschied betrifft das Setzen des Sicherheitsabstandes, um die Grenzen des Meshs auch bei großer Oszillation der Mesh-Punkte, außerhalb des Sichtfeldes zu halten. Zusätzlich zum vertikalen Sicherheitsabstand der SumSinusStrategy wird ein horizontaler Offset benötigt:

```
if (maxOffset < abs(x))
   maxOffset = abs(x);
if(maxOffset < abs(z))
   maxOffset = abs(z);

[...]
mesh->setSafetyOffset(maxOffset);
```

5.2.5 GerstnerPerlinStrategy

Die letzte Strategie verbindet Gerstner-Wellen und den Perlin-Noise Algorithmus. Die Manipulation durch Gerstner-Wellen findet dabei auf Vertex-Ebene statt, während der Noise zur Erstellung einer Normal-Map verwendet wird, die das Wasser auf Pixel-Ebene manipuliert. Im Gegensatz zu den bisher vorgestellten Strategien lässt die GerstnerPerlinStrategy die Positionsdaten des Meshs unangetastet, so dass die Bedingung $mesh_y = 0$ für alle Mesh-Punkte zu jeder Zeit erfüllt ist. Die eigentliche Wellenerzeugung findet komplett auf den Shadern der Grafikhardware statt. Die Aufgaben der Update-Methode sind somit zweierlei: Das Erzeugen einer Normal-Map und das Verwalten der Wellenparameter.

Die Erzeugung des Noise entspricht prinzipiell dem Ablauf der PerlinNoiseStrategy. Da der Noise allerdings zur Erzeugung einer Normal-Map und nicht zur direkten Manipulation der Mesh-Punkte verwendet wird, ergeben sich einige Detailunterschiede. So dienen nicht die Koordinaten der Knotenpunkte als Eingabewerte der Noisefunktion, sondern deren ganzzahligen Index-Werte $(0,0)$ bis (Mesh-Auflösung-1, Mesh-Auflösung-1). Dies hat die Unabhängigkeit des erzeugten Noise von der aktuellen Erscheinung des Meshs zur Folge. Außerdem kommt nun Gleichung 4 mit ihren vier Einzelaufrufen der Noise-Funktion zum Einsatz, denn um die Wasseroberfläche mit Hilfe der Normal-Map texturieren zu können, muss diese nahtlos kachelbar sein.

Eine Eigenschaft der Perlin-Noise-Funktion führt bei der Verwendung der Indizes als Parameter zu Problemen. Sind alle drei Eingabewerte der dreidimensionalen Noise-Funktion ganzzahlig, ist das Ergebnis der Funktion immer Null. Da es sich, neben den Indizes, auch bei der Zeit als dritten Parameter um eine ganze Zahl handelt (Millisekunden seit GLUT-Initialisierung), schwingt der Noise in konstanten Zeitintervallen komplett gegen Null und wird so unbrauchbar. Das Problem wird durch die vier Aufrufe der Noise-Funktion zwar vermindert, doch kommt es nach wie vor zu unerwünschten Darstellungseffekten. Zur Lösung des Problems wird der Eingabevektor der Noise-Funktion $input = (i, j, time)$ einer Rotation um einen kleinen Winkel unterzogen, so dass drei gleichzeitige ganze Zahlen als Parameter vermieden werden.

Das Ergebnis ist eine Height-Map mit Werten zwischen -1 und 1, aus der im nächsten Schritt die Normal-Map erzeugt wird. Dazu werden, wie in Abbildung 16 skizziert, die Differenzen zwischen der aktuellen Position des Height-Fields und deren Nachbarn berechnet, um aus ihnen die Normale der Normal-Map, an dieser

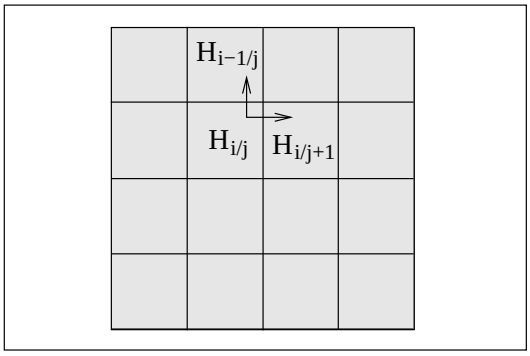

Abbildung 16: Um aus einem Height-Field eine Normal-Map zu erstellen, wird die Steigung zwischen aktuellem Pixel $H_{i/j}$ und dem oberen und rechten Nachbarn, $H_{i-1/j}$ und $H_{i/j+1}$ berechnet.

Position zu bilden:

```
Vec3f normal = Vec3f( _heightMap[i][j] - _heightMap[i-1][j],
                      _heightMap[i][j] - _heightMap[i][j+1],
                      1 );
```

Dieser Vektor muss noch normalisiert und in den Wertebereich [0..1] gebracht werden und kann dann in eine Textur (`osg::TextureChunk`) kodiert werden, wobei die r-/g-/b-Komponenten der Textur den x-/y-/z-Komponenten der Normale entsprechen. Nach Durchlaufen aller Punkte ist die Normal-Map fertig gestellt.

Die Erzeugung der Gerstner-Wellen ist identisch mit den Abläufen der GerstnerStrategy. Initialisierung der Parameter, sowie das Ein- und Ausblenden der Wellenzüge laufen nach dem gleichen Schema ab. Nur der letzte Schritt, das Durchlaufen aller Mesh-Punkte und Wellenzüge zur eigentlichen Berechnung der Gerstner-Wellen, entfällt und wird komplett in ein Shader-Programm ausgelagert.

Um Wellenparameter und Normal-Map an die Shader übergeben zu können, werden sie im Parameter-Container hinterlegt, der zu diesem Zweck entsprechende Methoden zur Verfügung stellt.

```
par->setWavelengthVector(_wavelength);
par->setAmplitudeVector(_amplitude);
par->setDirectionVector(_direction);
par->setWavePhaseVector(_phase);
par->setTexture(_normalMap);
```

Der für die Aktualisierung der Shaderparameter zuständigen Methode der Klasse `WaterScene` wird ein Zeiger auf den Container übergeben, so dass sie abhängig vom aktiven Shader, die benötigten Parameter auslesen kann.

Die für die Wasserbewegung dieser Strategie verantwortlichen Shader sind Gegenstand des folgenden Kapitels.

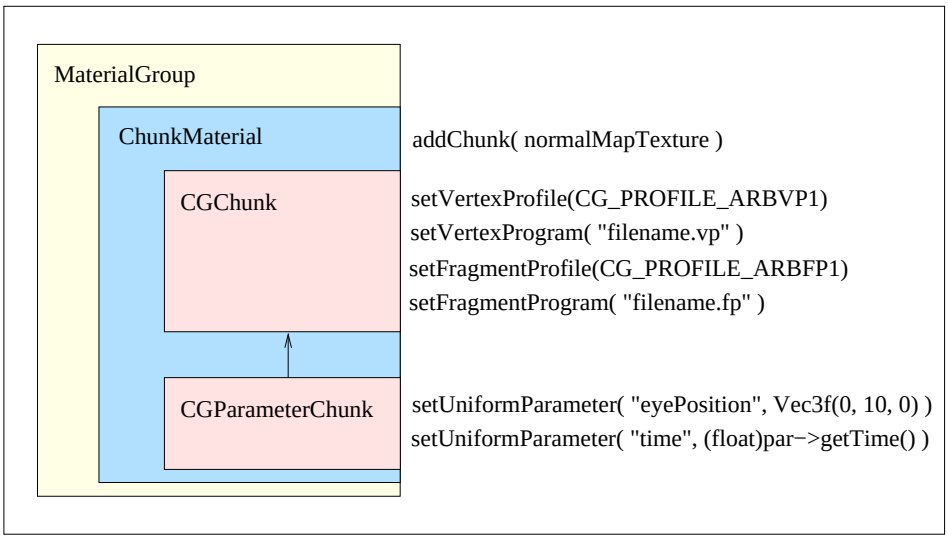

Abbildung 17: Aufbau einer Material-Group zur Realisierung von Cg-Programmen unter OpenSG. Rechts befinden sich einige beispielhafte Aufrufe, um die Aufgaben der einzelnen OpenSG-Datentypen zu verdeutlichen.

5.3 Shader

Das visuelle Erscheinungsbild des Wassers wird maßgeblich von den verwendeten Shadern bestimmt. Shader sind frei programmierbare Einheiten innerhalb der Rendering-Pipeline der Grafikkarte. Man unterscheidet zwischen zwei Shader-Typen. Vertex-Shader führen Berechnungen auf den 3D-Punkten einer Szene durch und können diese manipulieren. Ihnen gegenüber stehen die Fragment-Shader, oft auch als Pixel-Shader bezeichnet, die auf jeden Pixel des darzustellenden Bildes angewendet werden und den Farbwert des Pixels zurück liefern.

Die Schnittstelle zwischen *OceanApp* und den Shader-Programmen liegt, wie bereits erwähnt, in der Klasse `WaterScene`. Anhand eines Parameters aus dem Parameter-Container wird der aktive Shader identifiziert, die entsprechenden Shader-Files werden geladen und die vom Shader benötigten Parameter werden von der Applikation an diesen weitergereicht. Einmal pro Frame werden diese Parameter durch den Aufruf von

```
void WaterScene::updateCGParameter(ParameterContainer* par);
```

aktualisiert. Abbildung 17 illustriert den Aufbau einer Material-Group, die als Core im Szenegraph für die Verbindung zwischen Shader und Applikation sorgt.

Zwei Shader-Paare, bestehend aus je einem Vertex- und einem Fragment-Shader, sind im Zusammenspiel mit den in Kapitel 5.2 beschrieben Strategien notwendig. Ein Paar dient der Darstellung von optischen Effekten der Wasseroberfläche und kann für alle Strategien, mit Ausnahme der GerstnerPerlinStrategy verwendet werden. Die GerstnerPerlinStrategy benötigt ein spezielles Shader-Paar, da zusätzlich

```
void main(  float4 position : POSITION,
            float3 normal   : NORMAL,
            float3 texCoord : TEXCOORD0,

        out float4  oPosition       :   POSITION,
        out float3  I               :   TEXCOORD0,
        out float3  N               :   TEXCOORD1,
        out float3  positionW       :   TEXCOORD2,
        out float   foamFactor      :   TEXCOORD3,
        out float3  oTexCoord       :   TEXCOORD4,
        out float   fogExponent     :   TEXCOORD5,

    uniform float3 eyePositionW
    )
{
    // [...]
}
```

Abbildung 18: Der Kopf des allgemeinen Vertex-Shaders.

zu den Effekten die eigentlichen Wellen auf der Grafikkarte erzeugt werden müssen.

Die folgenden Effekte werden von beiden Shader-Paaren realisiert:

- Environment-Cube-Mapping unter Einbeziehung des Fresnel-Effekts zur Spiegelung der Umgebung auf der Wasseroberfläche,
- Durch direkte Sonnenreflektion entstehende Glanzlichter,
- Bildung von Schaum durch hohe Wellenaktivität,
- Mit der Entfernung zunehmende Nebelbildung.

5.3.1 Die allgemeinen Shader

Der Vertex-Shader dieses Shader-Paares ist relativ einfach aufgebaut. Da die Position der Vertices bereits durch die Applikation endgültig festgelegt wurde und Beleuchtung und Texturierung des Wassers auf Pixel-, statt auf Vertexebene stattfinden, werden lediglich vorbereitende Berechnungen durchgeführt und die Ergebnisse an den Fragment-Shader weitergereicht. Dementsprechend stehen den vier Eingabeparametern (Vertex-Position, Vertex-Normale, Texturkoordinate, Kameraposition in Weltkoordinaten) sieben Ausgabeparameter gegenüber (siehe Abbildung 18), die dem Fragment-Shader wiederum als Eingabeparameter dienen.

Eine Standardoperation eines Vertex-Shaders ist die Transformation der eingehenden Positionsdaten. Diese liegen in Objektkoordinaten vor und müssen zur Weiterverarbeitung durch den Rasterisierer in das so genannte Clipping-Koordinatensystem abgebildet werden. Es werden zwei Matrizen benötigt, die von der Applikation zur Verfügung gestellt werden müssen. Die ModelView-Matrix transformiert die

Raumkoordinaten in das Kamerakoordinatensystem, von wo sie durch die Projektions-Matrix in das Clipping-System transformiert werden. Häufig fasst man diese zwei Matrizen zu einer ModelViewProjection-Matrix zusammen. Durch das von *OceanApp* verwendete Vertex-Profil `arbvp1` erhält man direkten Zugriff auf diese Matrix und kann auf eine Übergabe als zusätzlichen Parameter verzichten.

```
float4x4 modelViewProj = glstate.matrix.mvp;
oPosition = mul(modelViewProj, position);
```

Um Beleuchtungsberechnungen durchführen zu können, müssen die beteiligten Größen in einem Koordinatensystem vorliegen. Da die Position der Sonne dem Fragment-Shader in Welt-Koordinaten übergeben wird, liegt es nah, zusätzlich zu den transformierten Positionsdaten die Vertex-Koordinaten im Welt-Koordinatensystem weiterzugeben.

```
positionW = position.xyz;
```

Es folgt die Berechnung des Sicht-Vektors, der sich von der Kameraposition in Richtung des aktuellen Vertex erstreckt. Die Bezeichnung I steht dabei für den einfallenden (*incident*) Strahl.

```
I = normalize( positionW - eyePositionW );
```

Die Entstehung von Schaum soll von der Stärke des Wellengangs abhängig gemacht werden. Zu diesem Zweck wird die Normale N des aktuellen Vertex mit der Normale (0,1,0) der Wasseroberfläche in Ruhestellung verglichen. Je größer die Abweichung ausfällt, desto mehr Schaum soll an dieser Stelle sichtbar werden. Der Vertex-Shader berechnet diese Abweichung mit Hilfe des Skalarproduktes, das den Cosinus des Winkels zwischen den beiden normalisierten Vektoren darstellt. Ein kleiner `foamFactor` muss vom Fragment-Shader demnach in ein hohes Schaumaufkommen umgesetzt werden.

```
N = normalize(normal);
foamFactor = dot( float(0,1,0), N );
```

Ein Nachteil dieser Methode ist das Ausbleiben von Schaum an den höchsten Punkten einer Welle. Sofern sie nicht sehr spitz zuläuft, flacht die Steigung der Welle nach und nach ab, womit sich auch die Schaumbildung verringert. Trotz hoher und steiler Wellen mit Schaum an den Wellenbergen, kann es so zu schaumlosen Wellenkronen kommen.

Als letztem Effekt widmet sich der Vertex-Shader dem Nebel. Mit wachsender Entfernung soll ein leichter Grauschleier das Wasser bedecken, um es natürlicher wirken zu lassen. Der Einfluss des Nebels hängt vom Abstand des Punktes zur Kameraposition, sowie von der Dichte des Nebels ab.

```
void main(  float3  I           :   TEXCOORD0,
            float3  N           :   TEXCOORD1,
            float4  positionW   :   TEXCOORD2,
            float   foamFactor  :   TEXCOORD3,
            float3  texCoord    :   TEXCOORD4,
            float   fogExponent :   TEXCOORD5,

    uniform float3 sunPositionW,
    uniform float  fresnelPower,

    out float4  color           :   COLOR,

    uniform samplerCUBE environmentMap,
    uniform sampler2D   foamTex
    )
{
    // [...]
}
```

Abbildung 19: Der Kopf des allgemeinen Fragment-Shaders.

```
float fogDensity = 0.0006;
float fogDistance = length( position - eyePositionW );
fogExponent = fogDistance * fogDensity;
```

Eine einfache Weitergabe der Texturkoordinaten der Wassergeometrie komplettiert den allgemeinen Vertex-Shader.

```
oTexCoord = texCoord;
```

Der Fragment-Shader nimmt die Ausgabedaten des Vertex-Shaders entgegen und liefert als Ergebnis die endgültige Farbe der dargestellten Pixel. Dieser Farbwert bildet die einzige Ausgabe eines Fragment-Shaders. Neben den Daten des Vertex-Shaders, erhält auch der Fragment-Shader Parameter direkt aus der Applikation. Besonders hervorzuheben sind hier die Datenstrukturen `samplerCUBE` und `sampler2D`, die eine sechsseitige Cubemap, bzw. eine 2D-Textur repräsentieren (siehe Abbildung 19).

Wie in Kapitel 3.4 erläutert, setzt sich die wahrgenommene Farbe des Wassers zunächst aus einer reflektierten und einer transmittierten Komponente zusammen, wobei letztere als konstant angenommen wird. Für die Berechnung des Reflexions-Vektors stellt Cg ebenso eine Methode bereit, wie für den darauf folgenden Texture-Lookup in der Cubemap.

```
float3 R = reflect(I, N);
float4 reflectedColor = texCube(environmentMap, R);
float4 refractedColor = float4(0.32, 0.39, 0.56, 1.0);
```

Die Gewichtung der beiden Farbwerte wird über den Fresnel-Term bestimmt. Die relativ aufwändige Berechnung des Reflexionsgrades (Gleichung 14) wird dabei durch einen sehr einfachen, approximierenden Term ersetzt, der die Eigenschaften des Fresnel-Effekts ausreichend gut nachbildet.

$$reflectionFactor = (1 - (-I \circ N))^{fresnelPower} \tag{18}$$

Schaut man senkrecht auf die Wasseroberfläche, so dass der Sichtstrahl mit der Normalen des Wassers zusammenfällt, beträgt der Reflexionsgrad Null und das Licht wird vollständig transmittiert. Mit abflachendem Sichtstrahl steigt der Reflexionsgrad und somit auch der Anteil an reflektiertem Licht. Der Exponent regelt die Geschwindigkeit dieses Anstieges. Der Reflexionsgrad dient der linearen Interpolation zwischen Reflexions- und Refraktions-Farbe.

```
float4 reflectionFactor = pow(1.0 + dot(I, N), fresnelPower);

float4 colorFresnel = lerp( refractedColor,
                            reflectedColor,
                            reflectionFactor );
```

Der Schaumeffekt wird durch eine einfache Textur realisiert, die auf die Wasseroberfläche gelegt wird. Wie stark diese sichtbar ist, hängt vom, im Vertex-Shader berechneten, `foamFactor` ab, der zur linearen Interpolation zwischen Schaumtextur und Wasserfarbe verwendet wird.

```
float4 foamColor = tex2D(foamTex, texCoord);

float4 colorFresnelFoam = lerp( foamColor,
                                colorFresnel,
                                foamFactor );
```

Glanzlichter auf der Wasseroberfläche werden nach Gleichung 17 berechnet. Die benötigten Größen `lightColor`, `shininess` und `Ks` werden im Shader deklariert und mit Werten belegt. Eine Auslagerung und Übergabe als Parameter ist für die Zukunft wünschenswert, um den Shader flexibler und allgemeiner anwendbar zu gestalten. Für die Berechnung des spekularen Lichtanteils werden der aktuelle Licht-Vektor, sowie dessen Reflektion benötigt.

```
float3 L = normalize( sunPositionW - position.xyz );
float3 R_light = 2 * dot(N, L) * N - L;

float specularLight = pow( max( dot(-I, R_light), 0), shininess);
float3 specular = Ks * lightColor * specularLight;
```

Durch eine einfache Addition werden Farbe des Wassers und Glanzlichter zusammengeführt.

```
color.xyz = colorFresnelFoam.xyz + specular.xyz;
color.w = 1;
```

Der Fragment-Shader schließt mit der Anwendung des Nebels auf die bisher ermittelte Farbe. Die Herleitung der verwendeten Formel für einheitlichen Nebel findet sich in [14]. Die Abhängigkeit zu Entfernung und Dichte des Nebels wurde bereits im Vertex-Shader hergestellt. Zur Berechnung der Nebelintensität wird eine Exponentialfunktion verwendet, deren Exponent der im Vertex-Shader berechnete `fogExponent` ist.

```
float3 fogColor = float(0.8, 0.8, 0.8);
float fogFactor = exp2( -abs(fogExponent) );
```

Die endgültige Pixelfarbe wird erneut durch lineare Interpolation ermitteln.

```
color.xyz = lerp( fogColor,
                  color.xyz,
                  fogFactor );
```

5.3.2 Die Shader der GerstnerPerlinStratey

Das Shader-Paar der GerstnerPerlinStrategy stellt die selben Effekte wie die allgemeinen Shader des letzten Kapitels dar. Darüber hinaus werden Wellenbewegungen innerhalb der Shader realisiert, die andere Strategien per CPU in der Applikation berechnen müssen. Dadurch ergeben sich im wesentlichen zwei zusätzliche Anforderungen an die Shader.

1. Der Vertex-Shader berechnet die Gerstner-Wellen und manipuliert die Wassergeometrie dementsprechend. Aus den Ableitungen der Gerstner-Funkion berechnet er Normale, Binormale und Tangente und reicht diese an den Fragment-Shader weiter.

2. Der Fragment-Shader erhält die durch Perlin-Noise entstandene Normal-Map als Parameter und führt anhand dieser ein Bump-Mapping durch.

Zur Umsetzung dieser Vorgaben werden eine Reihe neuer Ein- und Ausgabeparameter benötigt. Abbildung 20 zeigt die hinzugekommenen Parameter des Vertex-Shaders. Um Register zur Speicherung der Eingabeparameter zu sparen, wurden mehrere Werte zu Vektoren zusammengefasst. Zum besseren Verständnis der folgenden Code-Ausschnitte werden die erklärenden Bezeichnungen aus Abbildung 20 verwendet.

Die Berechnung der neuen Vertex-Positionen ist eine direkte Umsetzung der Gerstner-Funktion (Gleichung 9). Da die trigonometrische Funktion in allen drei Gleichungen auf den selben Ausdruck angewendet wird, macht es Sinn diesen vorzuberechnen. Dank der Cg-Funktion `sincos` ist dann eine effiziente Berechnung von Sinus und Cosinus mit einem Funktionsaufruf möglich.

```
                                /**
                                 * ============================
out float3  oBinormal            * kapf.x = k
out float3  oTangent             * kapf.y = amplitude
                                 * kapf.z = phase
uniform float4  kapf1            * kapf.w = frequency
uniform float4  dirspeed1        *
uniform float   time             * dirspeed.xyz = direction.xyz
uniform float2  texCoordMulti    * dirspeed.w   = c
                                 * ============================
                                */
```

Abbildung 20: Die zusätzlichen Ein- und Ausgabeparameter des Vertex-Shaders der GerstnerPerlinStrategy.

```
float sinArg = dot(k * direction.xz, position.xz)
                - (frequency * time * c) + phase);

sincos(sinArg, sinResult, cosResult);
```

Durch Abschluss der Gerstner-Berechnungen ergeben sich die endgültigen Vertex-Positionen.

```
resultPosition.x = position.x
                    - (amplitude * direction.x * sinResult);
resultPosition.z = position.z
                    - (amplitude * direction.z * sinResult);
resultPosition.y = amplitude * cosResult;
resultPosition.w = 1;
```

Die Bestimmung von Binormale und Tangente folgt dem gleichen Muster, weshalb auf eine Darstellung verzichtet wird. Wie Gleichungen 10 und 11 zeigen, entsprechen die Sinus- und Cosinus-Funktionen denen der Gerstner-Funktion, so dass die dort berechneten `sinResult` und `cosResult` wiederverwendet werden können. Die Normalen der Vertices erhält man durch Bildung des Kreuzproduktes zwischen Tangente und Binormale.

```
oNormal = cross(tangent, binormal);
```

Der letzte Unterschied zu den allgemeinen Shadern betrifft die Texturkoordinaten. Während sie bisher nur zur Platzierung einer Schaumtextur verwendet wurden, dienen sie im Fragment-Shader der GerstnerPerlinStrategy vor allem dem Bump-Mapping. Durch die Dynamik des Wasser-Meshs verändert sich der Abstand der Mesh-Punkte in Abhängigkeit von Kameraposition und Blickrichtung ständig. Bei statischen, fest definierten Texturkoordinaten würde sich im gleichen Maße die

Größe der durch die Bump-Map erzeugten Strukturen ändern, da die Bump-Map durch Skalierung an die geänderte Größe der Mesh-Felder angepasst wird. Um dies zu verhindern, skaliert der Shader die Texturkoordinaten der Wassergeometrie mit einem Faktor, der das Verhältnis zwischen dem ursprünglichen und dem aktuellen Abstand der Mesh-Vertices angibt.

```
oTexCoord.x = texCoord.x / texCoordMulti.x;
oTexCoord.y = texCoord.y / texCoordMulti.y;
```

Das bisher vorgestellte Konzept realisiert einen einzelnen Gerstner-Wellenzug. Da die Cg-Schnittstelle von OpenSG keine Übergabe von Arrays an die Shader unterstützt, ist eine Schleifenkonstruktion, die mit einer Laufvariable durch die Parameter-Arrays läuft, um die Wellen zu summieren, nicht möglich. Stattdessen gibt es für jede Anzahl von Wellenzügen einen eigenen Shader, der die entsprechende Zahl an Parametern entgegennimmt und die benötigten Werte hintereinander berechnet. Beispielhaft sei hier die Berechnung der x-Komponente für zwei Wellenzüge angeführt.

```
resultPosition = position.x
                 - (  amplitude1 * direction1.x * sinResult1
                    + amplitude2 * direction2.x * sinResult2 );
```

Leider stößt man durch die zusätzlichen Berechnungen relativ schnell an die Grenzen der ausführbaren Instruktionen, die durch das verwendete Vertex-Profil festgelegt werden. Ein Vertex-Shader, der drei Gerstner-Wellen übereinander lagert, stellt zur Zeit das Maximum dieser Technik dar.

Der Fragment-Shader der GerstnerPerlinStrategy weicht einzig bei der Berechnung der Normalen vom allgemeinen Shader ab. Die durch die Gerstner-Wellen definierten Normalen aus dem Vertex-Shader werden anhand einer Normal-Map weiter manipuliert und sorgen so für eine scheinbar detailliertere Wasseroberfläche, ohne dass eine rechenlastige, höher aufgelöste Geometrie notwendig wäre. Neben der Normal-Map und den zusätzlichen Ausgabeparametern des Vertex-Shaders (siehe Abbildung 20), besitzt der Fragment-Shader einen weiteren Eingabeparameter, der die Ausgeprägtheit der Bump-Mapping-Strukturen reguliert.

```
uniform float bumpyness,
uniform sampler2D noiseNormalMap,
```

Die Normal-Map enthält Werte im Bereich [0..1] und kodiert so in ihren drei Farbkanälen die x-/y-/z-Komponenten der Normalen. Diese haben einen Wertebereich von [-1...1], so dass zunächst eine Umrechnung erfolgen muss.

```
float3 noiseNormal = tex2D(noiseNormalMap, texcoord).xyz;
noiseNormal = (noiseNormal - 0.5) * 2;
noiseNormal = normalize(noiseNormal);
```

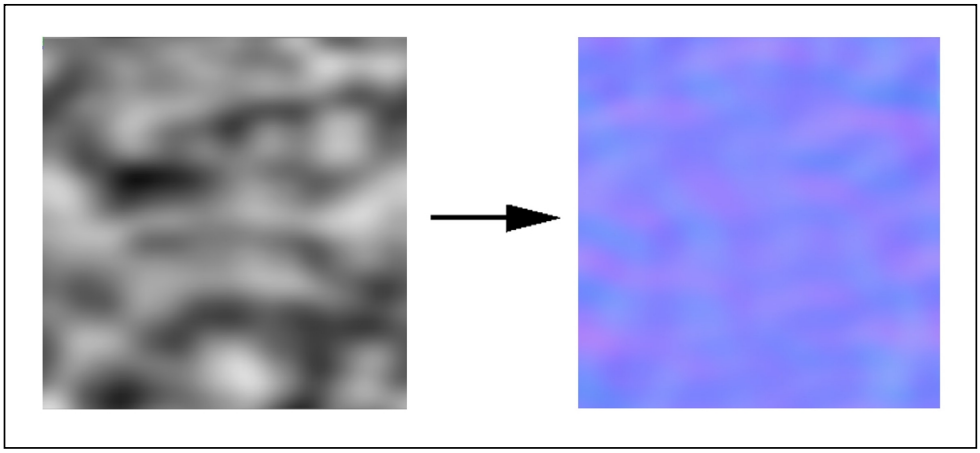

Abbildung 21: Ein aus Perlin-Noise gewonnenes Height-Field und die daraus erzeugte Normal-Map.

Die so gewonnene Normale liegt im so genannten Textur-Koordinatensystem vor. Das Koordinatensystem aufspannende Dreibein besteht aus zwei Achsen, die in der Texturebene liegen und einer dritten Achse, die senkrecht auf der Textur steht.

Um mit der Normalen weiter rechnen zu können, muss sie in das Koordinatensystem der Gerstner-Wellen konvertiert werden. Dieses wird durch das Orthonormalsystem aus Tangente, Binormale und Normale der Wasseroberfläche aufgespannt, die bereits durch den Vertex-Shader berechnet wurden. Die Rotationsmatrix `world2texRotation` mit den drei Vektoren als Spaltenvektoren, transformiert Vektoren aus dem Objekt-Koordinatensystem in das Textur-Koordinatensystem. Um eine Transformation in die entgegengesetzte Richtung zu erreichen, muss die Matrix invertiert werden, was bei einer orthonormalen Matrix der Transponierung entspricht.

```
float3x3 world2texRotation = float3x3(tangent  * bumpyness,
                                      binormal * bumpyness,
                                      normal);
float3x3 tex2worldRotation = transpose(world2texRotation);
```

Der `bumpyness`-Faktor steuert wie erwähnt, wie ausgeprägt das Bump-Mapping zum Vorschein tritt. Durch Anwendung der Transformation auf die Normale der Normal-Map, erhält man die endgültige Normale, mit der alle weiteren, in 5.3.1 beschriebenen, Fragment-Berechnungen durchgeführt werden.

```
float3 noiseNormalWorld = mul(tex2worldRotation, noiseNormal);
float3 resultNormal = normalize(noiseNormalWorld);
```

5.4 Ereignisbehandlung und die Display-Methode

Mit der Möglichkeit, zur Behandlung von Ereignissen wie einer Mausbewegung, oder einem Tastendruck, Methoden zu registrieren, bietet GLUT eine einfach zu verwendende Ereignisbehandlung. So registriert beispielsweise der Aufruf

```
glutKeybordFunc(keyboard);
```

die Methode `keyboard(...)` zur Behandlung von Tastaturereignissen. Da die registrierten Methoden einer von GLUT fest vorgeschriebenen Signatur folgen müssen, registriert *OceanApp* in der Klasse `WaterApplication` Hilfsmethoden, die diesen Signaturen entsprechen und die eigentlichen Methoden abkapseln. Der folgende Codeabschnitt registriert die Methode `glutKeyboard` mit der geforderten Signatur zum Abfangen von Tastaturereignissen. Diese Methode ruft wiederum die `keyboard`-Methode der Klasse `GLUTWindowWrapper`, mit allen notwendigen Parametern, auf. In dieser Klasse sind alle Methoden angesiedelt, die die Ereignisbehandlung letztendlich durchführen.

```
glutKeyboardFunc(glutKeyboard);
[...]
static GLUTWindowWrapper* _window;
[...]
static inline void glutKeyboard(unsigned char key, int x, int y)
{
   _window->keyboard(WaterApplication::the()->getMesh(),
                     WaterApplication::the()->getScene(),
                     WaterApplication::the()->getParameters(),
                     key, x, y);
}
```

Neben den oben genannten Ereignisbeispielen kann man Methoden registrieren, die aufgerufen werden, sobald der Bildschirminhalt aktualisiert werden muss, oder wenn einfach kein Ereignis stattfindet.

```
glutDisplayFunc(...);
glutIdleFunc(...);
```

Für beide „Ereignisse“ registriert *OceanApp* die gleiche Methode – die für die Bildschirmaktualisierung zuständige `display`-Methode.

Wie beim obigen Tastatur-Beispiel wird auch hier zunächst eine Hilfsmethode registriert, die dann die eigentliche `display`-Methode der Klasse `GLUTWindowWrapper` aufruft. Abbildung 22 zeigt den Aufbau dieser Methode, die einmal pro Frame aufgerufen wird und die notwendigen Parameter als Referenz übergeben bekommt. Als ersten Schritt wird die seit Programmstart abgelaufene Zeit ausgelesen und die dafür vorgesehene Variable im Parameter-Container aktualisiert. So ist sichergestellt, dass eine framegenaue Zeit zur Verfügung steht und durch eine einfache

```
void GLUTWindowWrapper::display(WaterScene* scene,
                                MeshData* mesh,
                                ParameterContainer* parameters)
{
    // update time in parameter-container
    Int32 time = glutGet(GLUT_ELAPSED_TIME);
    parameters->setTime(time);

    // update camera-matrix
    scene->nav_updateCameraTransformation();

    // update mesh position
    bool isVisible = mesh->updateMeshPosition(scene->getSceneFrustum());

    // if mesh is visible, update mesh-geometry
    if (isVisible){

        // update texture-coordinate-ratio in parameter-container
        parameters->setMeshStepsizeRatioX( mesh->getStepsizeRatioX() );
        parameters->setMeshStepsizeRatioZ( mesh->getStepsizeRatioZ() );

        // update water-geometry
        _currentWaterStrategy->updateWaterSurface(mesh, scene, parameters);

        // update cg-parameters
        scene->updateCGParameter(parameters);
    }

    // render the scene
    _glutWindow->render(_renderAction);
}
```

Abbildung 22: Die Display-Methode

Übergabe des `ParameterContainer`-Zeigers applikationsweit verwendet werden kann.

```
Int32 time = glutGet(GLUT_ELAPSED_TIME);
parameters->setTime(time);
```

Nach der Aktualisierung der Kamera-Transformationsmatrix erfolgt der in Kapitel 5.1 vorgestellte Aufruf zur Aktualisierung des Wasser-Meshs. Hier erfolgt nur eine Aktualisierung der Positionsdaten – die sichtbare Geometrie ist noch nicht betroffen.

```
bool isVisible = mesh->updateMeshPosition( scene->getSceneFrustum() );
```

Der Rückgabewert `isVisible` zeigt an, ob das Mesh überhaupt innerhalb des aktuellen Kamerasichtfeldes liegt. Ist das nicht der Fall, können alle weiteren Schritte, die die Wassergeometrie betreffen, im aktuellen Frame ausgelassen werden. Die ersten beiden Funktionsaufrufe innerhalb des if-Blocks, schreiben aktualisierte Werte, der in Kapitel 5.3.2 erwähnten Faktoren zur Skalierung der Texturkoordinaten, in den Parameter-Container. Da sich Lage, Ausdehnung und Skalierung der Wassergeometrie in jedem Frame ändern können, müssen auch die Texturkoordinaten entsprechend frameweise angepasst werden.

Es folgt der Aufruf des in Abschnitt 5.2 vorgestellten Interfaces der WaterStrategie-Klassen, der entsprechend der aktiven Strategie die Wassergeometrie manipuliert.

```
_currentWaterStrategy->updateWaterSurface(mesh, scene, parameters);
```

Als letzter Schritt vor dem Renderingprozess werden nun aktuelle Parameter an die aktiven Shader übergeben.

```
scene->updateCGParameter(parameters);
```

Der Parameter-Container enthält sowohl die Information, welche Shader aktiv sind, als auch die zu übergebenen Parameter.

6 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Studienarbeit, anhand von Bildern der Applikation *OceanApp* illustriert.

Abbildung 23: Leichte Wellenaktivität durch einen Perlin-Noise mit kleiner Amplitude (PerlinNoiseStrategy).

Abbildung 24: Perlin-Noise mit erhöhter Amplitude, Blick auf die Sonne und die dadurch entstehenden Glanzlichter auf der Wasseroberfläche.

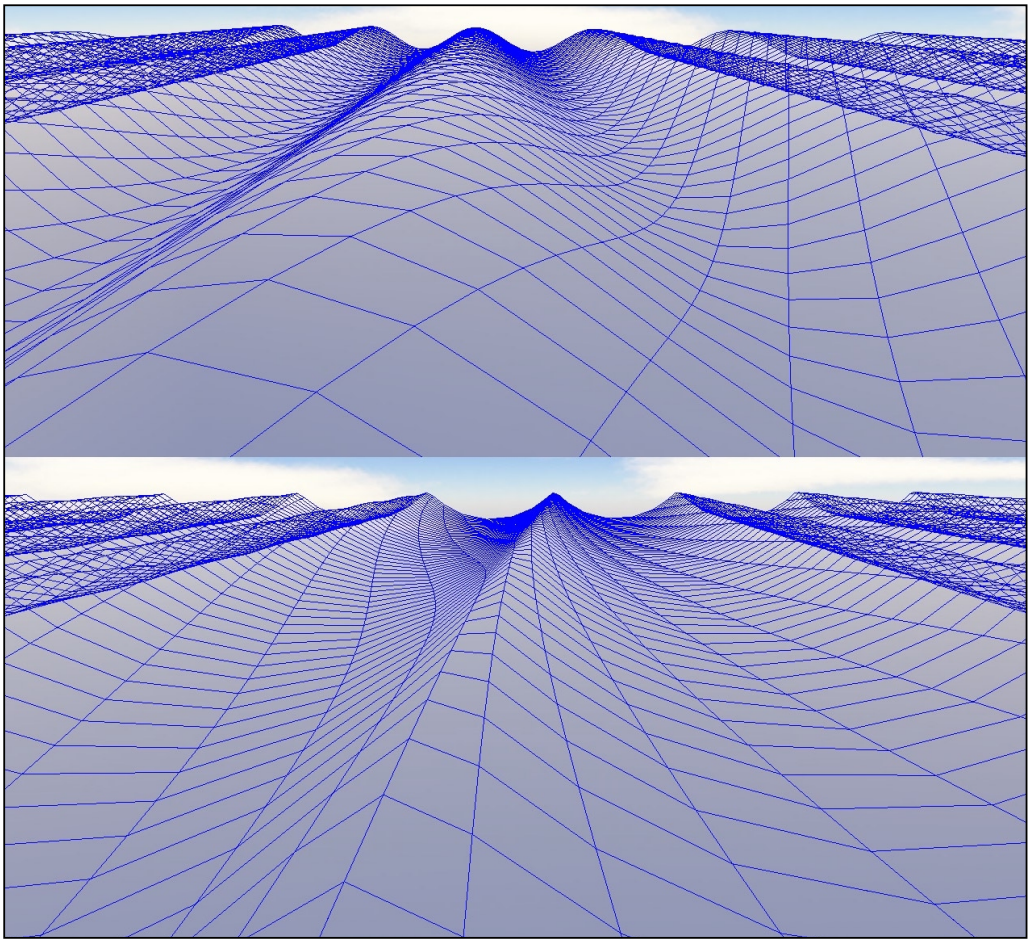

Abbildung 25: Vergleich zwischen runden Sinuswellen (oben) und den spitzer zulaufenden Gerstnerwellen (unten) in einem Wireframe-Modus.

Abbildung 26: Drei überlagerte Gerstner-Wellen, mit...

Abbildung 27: ...und ohne Blick in die Sonne (GerstnerPerlinStrategy).

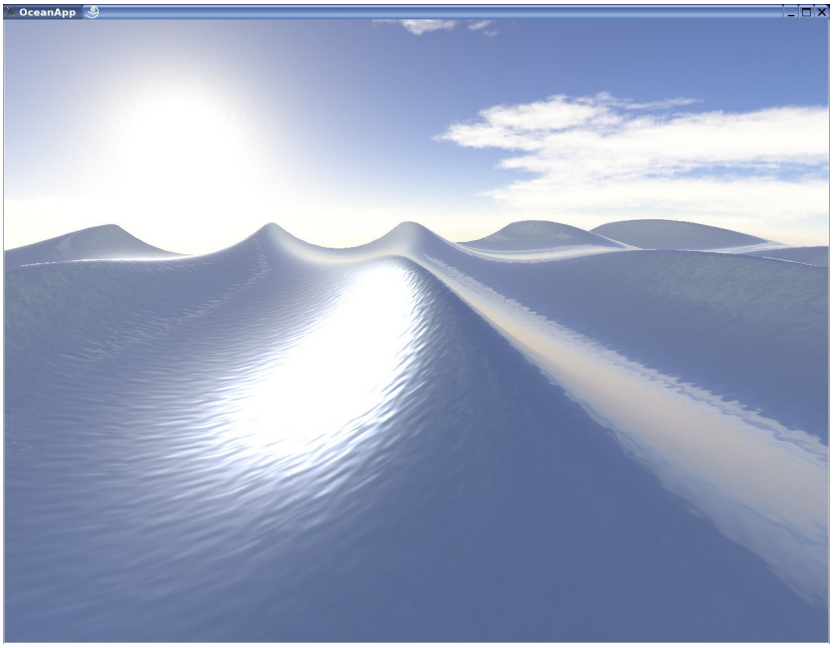

Abbildung 28: Erhöte Amplitude und verminderte Wellenlänge führen zu sich auftürmenden Wellen (GerstnerPerlinStratefy).

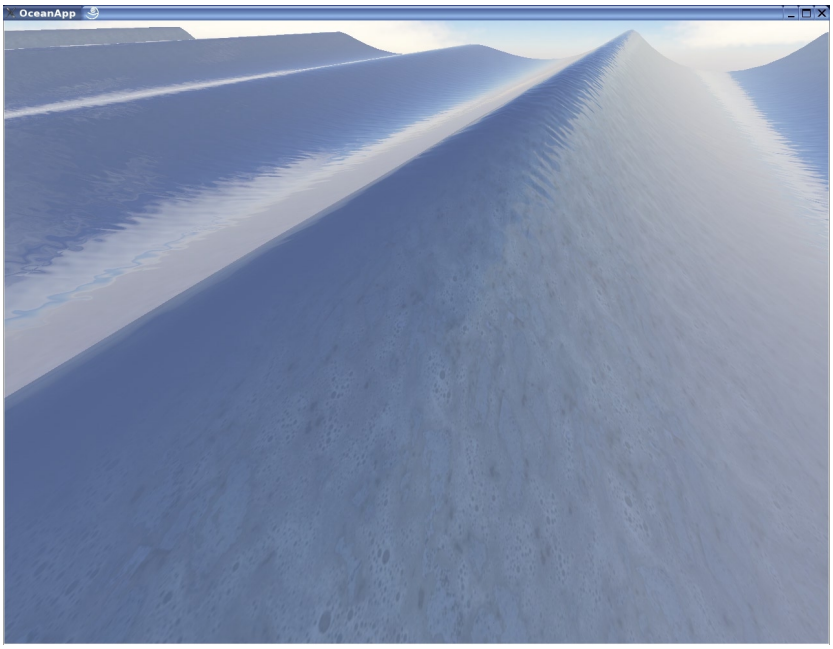

Abbildung 29: Durch den hohen Wellengang wird die Schaumtextur auf der Wellenoberfläche sichtbar.

7 Fazit und Ausblick

7.1 Fazit

Mit der Entwicklung von *OceanApp* wurden die grundlegenden Ziele dieser Studienarbeit erreicht. Es wurde eine Anwendung geschaffen, die verschiedene Algorithmen der Wellendarstellung implementiert und aufgrund ihrer Architektur diesbezüglich auch erweiterbar ist. Neben der Vertiefung von C++-Kenntnissen und der Einarbeitung in die Shadersprache Cg, konnten bei Konzeption und Umsetzung von *OceanApp* Prinzipien der Softwaretechnik erfolgreich in der Praxis eingesetzt werden.

Die implementierten Algorithmen weise verschiedene Stärken und Schwächen auf. Die PerlinNoiseStrategy liefert bei einer leicht bewegten Wasseroberfläche ein überzeugendes optisches Ergebnis, doch mit wachsender Wellenaktivität und somit steigender Noise-Amplitude wirkt die entstehende Bewegung schnell unnatürlich. Bei höherem Wellengang hat sich die Verwendung von Gerstnerwellen bewährt. Im Gegensatz zu reinen noiseerzeugten Wellen, kann außerdem die Richtung von Gerstnerwellen kontrolliert werden, so dass sie sich an einer Windrichtung orientieren können. Um die Wasseroberfläche nicht künstlich glatt wirken zu lassen, müssen entweder sehr viele Gerstnerwellen unterschiedlicher Amplitude verwendet werden, oder die Oberfläche muss mit anderen Verfahren weiter manipuliert werden, wie es die GerstnerPerlinStrategy praktiziert.

Im Nachhinein betrachtet, haben allgemeine Entwurfs- und Programmieraufgaben und die damit einhergehenden Probleme, im Vergleich zu der Beschäftigung mit wasserspezifischen Aufgaben, mehr Zeit in Anspruch genommen, als vor der Arbeit geplant war. Auch musste aus Zeitgründen auf zuvor angedachte Features verzichtet werden, von denen sich einige im folgenden Ausblick wieder finden.

7.2 Ausblick

Verschiedene Erweiterungen von *OceanApp* sind denkbar und nahe liegend. Das Hinzufügen weiterer Verfahren zur Manipulation der Wassergeometrie gestaltet sich dank der in Kapitel 5.2 beschriebenen Architektur relativ problemlos. Das Einbinden neuer Shader ist dagegen mit mehr Aufwand verbunden. Hier wäre eine Verbesserung der Programmstruktur wünschenswert.

Die bisher implementierten Modelle zielen alle auf die Darstellung eines offenen Meeres ab, bei dem die Wassertiefe außer Acht gelassen werden kann. Die Erweiterung der Anwendung um ein Modell zur Simulation geringer Wassertiefen und vor allem die Verbindung der beiden Modelle, ist somit ein weiteres Aufgabengebiet. Dieses bringt, neben dem eigentlichen Wellenmodell, zusätzliche Herausforderungen mit sich, wie das Hinzufügen eines (Unterwasser)-Terrains in den Szenegraph, optische Unterwassereffekte wie Kaustiken, oder das Verhalten des Wassers an einem Strand.

Ein weiteres Betätigungsfeld sind Objekte, die mit dem Wasser interagieren,

wie beispielsweise Schiffe, die einerseits vom Wellengang bewegt werden und diesen andererseits selber beeinflussen. Dazu könnten Partikeleffekte kommen, die die Gischt am Bug eines Schiffes bei schneller Fahrt, oder auch aufeinander treffende Wellen erzeugt.

Eine wesentliche Verbesserung der Applikation wäre eine Optimierung des Wasser-Meshs. Ein interessantes Konzept stellt hierbei ein adaptives, projiziertes Mesh dar, dass seinen Detailgrad an Blickrichtung und -winkel der Kamera anpasst [6][5].

Das Shader-Model 3.0, welches durch NVIDIAs NV40-Grafikchip Einzug in Consumer-PCs erhält, bietet darüber hinaus neue Möglichkeiten in technischer Hinsicht. So können mit dem neuen Modell auch in den Vertex-Shader Texturen geladen und dort verarbeitet werden. Eine Normal-Map könnte also nicht nur im Pixel-Shader zu Beleuchtungszwecken, sondern schon zur Verformung der Geometrie im Vertex-Shader verwendet werden (Displacement Mapping), um die Komplexität der Wasseroberfläche zu steigern.

Literatur

[1] *Fresnelsche Formeln*. http://de.wikipedia.org/wiki/Fresnelsche_Formeln

[2] *Snelliussches Brechungsgesetz*. http://de.wikipedia.org/wiki/Brechungsgesetz_von_Snellius

[3] *Terragen*. http://www.planetside.co.uk/terragen

[4] ALAIN FOURNIER, WILLIAM T. REEVES: A Simple Model of Ocean Waves. In: *ACM SIGGRAPH Computer Graphics* Bd. 20, 1986

[5] CLAES JOHANSON: *Real-time water rendering*. http://graphics.cs.lth.se/theses/projects/projgrid/. – Online Ressource

[6] DAMIEN HINSINGER, FABRICE NEYRET, MARIE-PAULE CANI: Interactive Animation of Ocean Waves. In: *Symposium on Computer Animation*

[7] DOUGLAS ENRIGHT, STEPHEN MARSCHNER, RONALD FEDKIW: Animation and Rendering of Complex Water Surfaces. In: *SIGGRAPH 02*

[8] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995

[9] HUGO ELIAS: *Perlin Noise*. http://freespace.virgin.net/hugo.elias/models/m_perlin.htm. – Online Ressource

[10] JERRY TESSENDORF: Simulating Ocean Water. In: *SIGGRAPH 2001 Course notes*, 2001

[11] KEN PERLIN: *Improved Noise reference implementation*. Version: 2002. http://mrl.nyu.edu/~perlin/noise/. – Online Ressource

[12] KEN PERLIN: Implementing Improved Perlin Noise. In: RANDIMA FERNANDO (Hrsg.): *GPU Gems*. Addison-Wesley Professional, 2004, S. 73–85

[13] MATT ZUCKER: *The Perlin noise math FAQ*. Version: 2001. http://www.robo-murito.net/code/perlin-noise-math-faq.html. – Online Ressource

[14] RANDIMA FERNANDO, MARK J. KILGARD: *The Cg Tutorial*. Addison-Wesley Professional, 1993