Dynamische Environment Maps

Studienarbeit im Fach Informatik

vorgelegt von

Christoph Dietze

geb. am 3. Juni 1980 in Erlangen

angefertigt am

Institut für Informatik
Lehrstuhl für Graphische Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr. Marc Stamminger

Betreuender Hochschullehrer: Prof. Dr. Marc Stamminger

Beginn der Arbeit: 15. Juli 2004

Abgabe der Arbeit: 15. April 2005

Inhaltsverzeichnis

Kapitel 1

Einleitung

In lokalen Beleuchtungsmodellen wird die Farbe eines Punktes ausschließlich aus den Eigenschaften der Oberfläche an dieser Stelle und der von Lichtquellen direkt eintreffenden Lichtstrahlen berechnet. In diesen Modellen wird also nicht einbezogen, dass durch Abdeckung Schatten entstehen können, oder dass sich in einem Objekt die Umgebung spiegeln kann. Im Unterschied hierzu werden bei globalen Beleuchtungsmodellen auch Reflexion und Transmission von Licht an Oberflächen berücksichtigt, die Berechnung ist jedoch wesentlich aufwändiger. Durch Verfahren wie Raytracing können diese Modelle in nicht-interaktiven Anwendungen realisiert werden. Für interaktive Anwendungen ist der Aufwand jedoch meist zu groß und man verwendet v.a. lokale Beleuchtungsmodelle. Da jedoch Effekte wie Schatten oder Reflexionen wesentlich zum Realismus beitragen, wurden verschiedene Verfahren entwickelt, die zwar physikalisch nur annähernd korrekt sind, jedoch wesentlich effizienter zu berechnen sind und trotzdem überzeugend wirken.

Environment Mapping (EM) ist ein solches Verfahren, welches unter bestimmten Voraussetzungen realistisch wirkende Reflexionen erzeugen kann. Es ist sowohl geeignet, künstliche Umgebungen, als auch von Kameras aus der Realität aufgenommene Umgebungen, in Spiegelungen darzustellen. Siehe z.B. Abbildung 1.1. In interaktiven Anwendungen finden Environment Maps bisher meist nur in statischen Szenen Anwendung. In dieser Arbeit werden Verfahren vorgestellt, welche mithilfe von Erweiterungen moderner Graphikkarten, EM in dynamischen Szenen ermöglicht.

Mit der dramatischen Weiterentwicklung der Graphikhardware ist es jetzt jedoch möglich, EM auch in dynamischen Szenen zu verwenden.

In Kapitel 2 werden die benötigten Voraussetzungen der Graphikhardware beschrieben, Kapitel 3 beschreibt verschiedene Verfahren zum EM und in Kapitel 4 wird die Implementierung der Demo-Anwendung zu diesen Verfahren vorgestellt und die interessanten Teile ausführlicher beschrieben.

Abbildung 1.1: Mittels Environment Mapping wurden die Spiegelungen auf dem T-1000 Roboter in dem 1991 erschienenen Film „Terminator 2: Judgment Day“ erzeugt.

Kapitel 2

Leistungsumfang moderner Graphikkarten

Die Leistungsfähigkeit von Graphikkarten hat sich in den letzten Jahren sehr rasant entwickelt. Bei Prozessoren gibt es die Faustregel, dass sich ihre Leistung etwa alle 18 Monate (Moores Gesetz) verdoppelt. Bei Graphikkarten sagt man, dass sich ihre Leistung jedes Jahr mehr als verdoppelt.

Durch so genannte Erweiterungen ist es möglich, bestimmte Aufgaben, die sonst die CPU verrichten müsste durch die Graphikkarte erledigen zu lassen. Dies bringt oftmals große Geschwindigkeitsvorteile, da die CPU entlastet wird, die Graphikchips für diese Aufgaben besser spezialisiert sind und der teure Datentransfer zwischen CPU und GPU geringer ist, bzw. oft nicht mehr nötig ist.

Gerade für interaktive Graphikanwendungen haben sich durch höhere Taktraten und neue Erweiterungen der Graphikkarten viele neue Möglichkeiten ergeben. Zum Beispiel wäre das Verfahren, Environment Maps durch Verzerren der Szene zu generieren, wie es in dieser Arbeit vorgestellt wird, ohne die Erweiterung Vertex Shader nicht in interaktiven Anwendungen einsetzbar.

2.1 Vertex Shader

Bis vor wenigen Jahren hatte jede Graphikkarte ausschließlich eine *feste* Transformations- und Beleuchtungseinheit. Aktuelle Karten besitzen jedoch so genannte *Vertex Shader* (auch Vertex Programme genannt), in denen die Anwendung die Transformations- und Beleuchtungsberechnung definieren kann. Vergleiche Abbildung 2.1. Ist ein Vertex Shader aktiv, so wird dieser für jeden zu zeichnenden Eckpunkt (vertex) ausgeführt. Zu den wichtigsten Ein- und Ausgabe Eigenschaften vergleiche Abbildung 2.2.

Es gibt eine ganze Fülle von Einsatzgebieten für Vertex Shader. Oft werden Vertex Shader zusammen mit Pixel Shadern eingesetzt, um erweiterte Beleuchtungsmodelle zu realisieren. Animationen lassen sich oft effizienter durch Vertex Shader implementieren, z.B. lassen sich die Wellenbewegungen einer Wasseroberfläche durch entsprechende Veränderung der Position der Eckpunkte in einem Vertex Shader realisieren, statt jede Eckpunktverschiebung in der Anwendung definieren zu müssen.

Es gibt verschiedene Sprachen, in denen Vertex Shader programmiert werden können. Vom ARB (Architecture Review Board) wurde z.B. die assemblerähnliche Sprache „ARB vertex program“ [2, 4, 13] definiert. Es gibt jedoch mittlerweile eine Reihe von Hochsprachen wie z.B. Cg, HLSL oder GLslang. In dieser Arbeit werden ausschließlich ARB vertex programs verwendet.

Als Beispiel für ein sehr einfaches, aber vollständiges ARB vertex program siehe Listing 2.1. Die Endposition (result.position) jedes Eckpunkts wird durch Multiplikation der kombinierten Modelview-

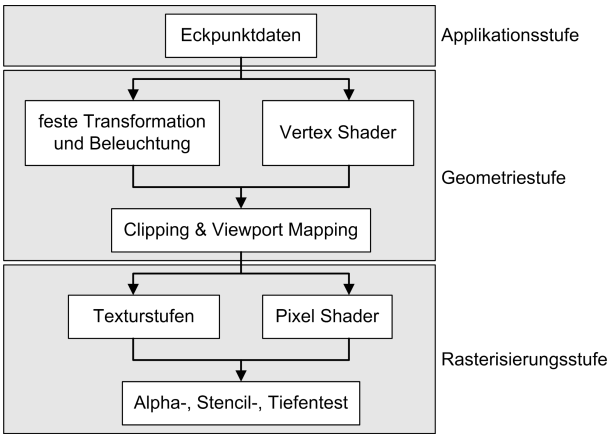

Abbildung 2.1: Rendering Pipeline

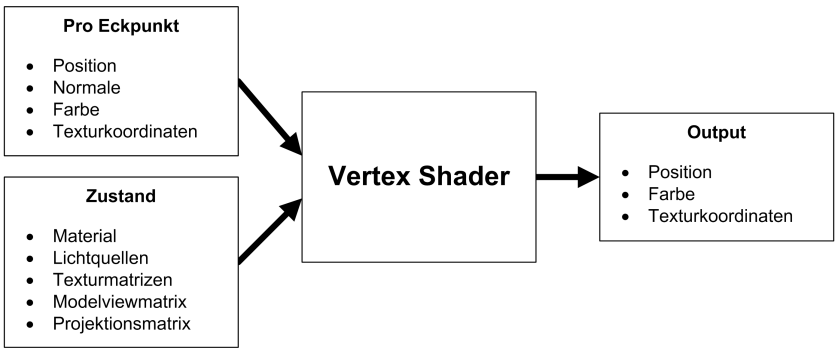

Abbildung 2.2: Vertex Shader Ein-/Ausgabe

```
1  !!ARBvp1.0
2
3  PARAM mvp[4] = { state.matrix.mvp };
4
5  DP4 result.position.x, mvp[0], vertex.position;
6  DP4 result.position.y, mvp[1], vertex.position;
7  DP4 result.position.z, mvp[2], vertex.position;
8  DP4 result.position.w, mvp[3], vertex.position;
9
10 MOV result.color, vertex.color;
11 MOV result.texcoord, vertex.texcoord;
12
13 END
```

Listing 2.1: Einfaches ARB vertex program

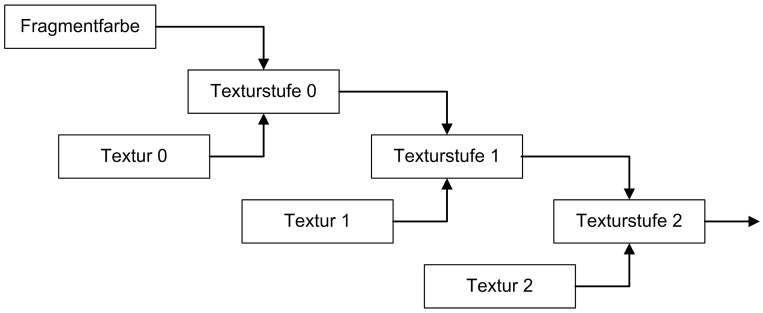

Abbildung 2.3: Texturstufen

und Projektionsmatrix (state.matrix.mvp) mit der Eckpunktposition (vertex.position) berechnet. Die Multiplikation einer Matrix mit einem Vektor wird hier durch viermalige Berechnung des Skalarprodukts erreicht. Die eingehende Farbe (vertex.color) und Texturkoordinaten (vertex.texcoord) wird unverändert in die endgültige Farbe (vertex.color) und Texturkoordinaten (vertex.texcoord) geschrieben.

2.2 Multitexturing

Heutzutage können die meisten Graphikkarten zwei oder mehr Texturen in einem einzigen Zeichendurchlauf auf dasselbe Polygon anwenden. Diesen Prozess bezeichnet man als *Multitexturing*. Hierbei durchläuft jedes Bildfragment eine Reihe von so genannten Texturstufen, die auch als Textureinheiten bezeichnet werden. In jeder Texturstufe ist eine Textur und eine Kombinierungsfunktion definiert. In der ersten Texturstufe wird die Textur dieser Stufe entsprechend der Kombinierungsfunktion mit der Farbe des Fragments kombiniert. In den folgenden Stufen wird die jeweilige Textur mit dem Ergebnis der vorhergehenden Stufe kombiniert. Man kann sich die hintereinander geschalteten Texturstufen als eine Art Pipeline vorstellen, siehe Abbildung 2.3.

Zum Beispiel wird Multitexturing beim Lightmapping eingesetzt. Hier werden vorher die Helligkeiten einer Szene (z.B. mittels Radiosity) berechnet und in Texturen gespeichert. Diese Texturen werden dann mit den Texturen, die die eigentliche Farbe der Objekte enthalten, multipliziert. Siehe Abbildung 2.4. Somit können die Texturen mit der Farbinformation mehrfach in einer Szene ver-

Abbildung 2.4: Lightmapping

wendet werden. Außerdem lassen sich die Texturen, die die Beleuchtungsinformationen enthalten, effizient speichern, da nur ein Farbkanal nötig ist und die Auflösung dieser Texturen geringer gewählt werden kann.

Weitere, typische Einsatzgebiete von Multitexturing sind Detailmapping oder die Implementierung anderer Beleuchtungsmodelle. Oft werden in Texturen nicht Farbwerte gespeichert, sondern man verwendet die drei oder vier Fließkommazahlen um Vektoren darzustellen (z.B. bei Normal Maps).

2.3 PBuffer

PBuffer (pixel buffer) ist eine OpenGL Erweiterung, die es ermöglicht, einen unsichtbaren Renderkontext zu erzeugen, in den wie in ein übliches OpenGL-Fenster gezeichnet werden kann (dieser Vorgang wird auch als *off-screen rendering* bezeichnet). Dieser PBuffer kann dann in eine Textur kopiert werden. Das Kopieren geschieht auf der Graphikkarte, ist also relativ effizient. Darüber hinaus gibt es aktuelle Hardware, die das direkte Zeichnen in Texturen erlaubt [12].

Die Möglichkeit in Texturen zeichnen zu können, findet eine ganze Reihe von Anwendungen, z.B. Shadow Mapping, Impostors oder, wie in dieser Arbeit, zur Erzeugung von Environment Maps.

2.4 Automatische Mipmap Generierung

Aktuelle Graphikhardware unterstützt das automatische Generieren von Mipmaps. Ist diese Erweiterung aktiviert, so werden automatisch Mipmaps erzeugt und aktualisiert, wenn sich die betreffende Textur ändert. Die Aktualisierungen werden von der Graphikkarte übernommen und sind für die Anwendung transparent.

2.5 Automatische Generierung von Texturkoordinaten

In OpenGL ist es möglich, automatisch Texturkoordinaten generieren zu lassen, statt diese für jeden Eckpunkt bei jedem Zeichnen von der Anwendung zu setzen. Es lässt sich für jede der vier Texturkoordinatenkomponenten (s, t, r, q) getrennt eine Generierungsfunktion spezifizieren. Die hier vorgestellten Funktionen sind GL_OBJECT_LINEAR, GL_EYE_LINEAR, GL_SPHERE_MAP und GL_REFLECTION_MAP, wobei vor allem die letzteren beiden für diese Arbeit interessant sind.

Im Modus GL_OBJECT_LINEAR spezifiziert man die Parameter a, b, c, d. Der Wert der Komponente berechnet sich durch $av_x + bv_y + cv_z + dv_w$, wobei v_x, v_y, v_z, v_w die Koordinaten des Eckpunkts im Objektraum sind. Interpretiert man die Parameter als Faktoren in der Ebenengleichung

$ax + by + cz + d = 0$, so ist der Wert der Texturkoordinate der Abstand des Eckpunkts zu dieser Ebene. Die Texturkoordinaten verändern sich nicht, wenn das Objekt transformiert wird und so scheint die Textur fest an das Objekt gebunden.

Der Modus GL_EYE_LINEAR funktioniert analog zu GL_OBJECT_LINEAR, nur dass hier die Parameter a, b, c, d mit der inversen Modelviewmatrix multipliziert werden, die bei der Spezifizierung der Parameter gültig ist. Dies hat den Effekt, dass sich die Ebene nicht mit dem Objekt bewegt und so scheint das Objekt durch die Textur zu schwimmen.

GL_SPHERE_MAP wird vorrangig dazu benutzt, um die Texturkoordinaten zu generieren, die eine sphärische Environment Map auf ein Objekt auftragen. Hierbei wird automatisch der Reflexionsvektor wie in Gleichung 3.1 berechnet und die Texturkoordinaten dann nach Gleichung 3.6 und 3.7 gesetzt (siehe Kapitel 3.3).

Im Modus GL_REFLECTION_MAP wird lediglich der Reflexionsvektor wie in Gleichung 3.1 berechnet und seine Komponenten in den Texturkoordinaten gespeichert ($(s, t, r) = (r_x, r_y, r_z)$). In dieser Arbeit wird dieser Modus benutzt, um die Texturkoordinaten für parabolische Environment Maps zu generieren. Hierzu muss zusätzlich noch die Texturmatrix entsprechend gesetzt werden (siehe Kapitel 3.4).

All diese Funktionen lassen sich alternativ auch durch Vertex Programme implementieren. Vertex Programme sind weitaus flexibler, so dass sich auch weitere Funktionen zur Generierung von Texturkoordinaten mit ihnen realisieren lassen. Vertex Programme und das automatische Generieren sind nicht kombinierbar, d.h. wenn ein Vertex Programm aktiv ist, so ist es nicht möglich, die Texturkoordinaten automatisch generieren zu lassen, sondern man muss dies in das Vertex Programm integrieren.

Kapitel 3

Environment Mapping

3.1 Allgemeines zu Environment Mapping

Environment Mapping (EM) ist eine einfache Möglichkeit, annähernd realistische Reflexionen der Umgebung auf spiegelnden Objekten zu erzeugen. Hierzu wird die gesamte Umgebung vom Mittelpunkt des spiegelnden Objekts in eine oder mehrere Texturen gezeichnet. Beim Zeichnen des spiegelnden Objekts wird für jeden Punkt, abhängig von der Blickrichtung des Betrachters und der Normalen des Punktes, die Richtung des Reflexionsstrahls berechnet. Dieser wird verwendet, um die richtige Textur und Texturkoordinaten zu bestimmen.

Beim EM macht man die Annahme, dass, relativ zur Größe des spiegelnden Objekts, sowohl Betrachter, als auch reflektierte Objekte, weit von diesem entfernt sind. Außerdem darf sich das Objekt nicht in sich selbst spiegeln. Durch diese Annahmen ist die Umgebung von allen Punkten des Objekts ähnlich und es genügt, die Umgebung vom Mittelpunkt des Objekts zu speichern, um annähernd korrekte Spiegelungen zu erzeugen. Als *dynamisches* Environment Mapping bezeichnet man die Verfahren, welche die Texturen in jedem Frame neu erzeugen.

Für Environment Mapping sind folgende Schritte notwendig:

1. Lade oder erzeuge die Textur/-en, welche die Umgebung des spiegelnden Objekts enthält/enthalten.

2. Berechne für jeden Punkt auf dem spiegelnden Objekt den Reflexionsvektor aus der Blickrichtung des Betrachters und der Normalen.

3. Wähle anhand des Reflexionsvektors die entsprechende Textur und Texturparameter.

4. Zeichne das texturierte Objekt.

Der Reflexionsvektor r eines Punkts berechnet sich aus der Oberflächennormalen n an diesem Punkt und e, dem Vektor vom Punkt in Richtung des Betrachters. Siehe Abbildung 3.1 und Abbildung 3.2. Sind n und e normalisiert, so gilt:

$$r = 2(n \cdot e)n - e \tag{3.1}$$

Da der Einfallswinkel gleich dem Ausfallswinkel ist, ist n der Vektor im Halbwinkel von e und r. Deshalb ist n die normalisierte Summe von e und r:

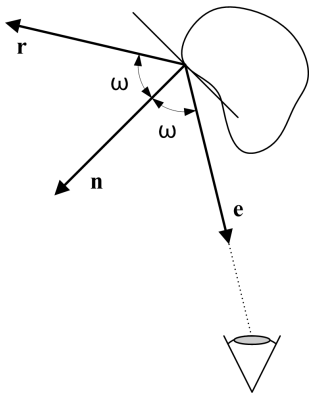

Abbildung 3.1: Reflexion eines Lichtstrahls

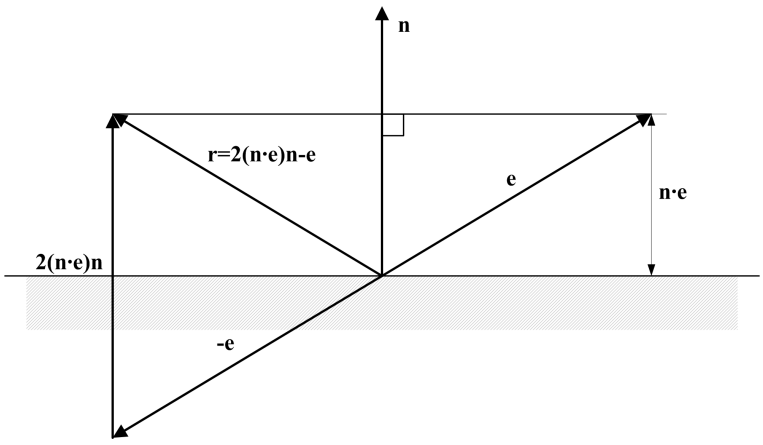

Abbildung 3.2: Berechnung des Reflexionsvektors r

$$n = \frac{e + r}{\|e + r\|} \tag{3.2}$$

Man bezeichnet eine Environment Map als *blickrichtungsunabhängig* (view independent), wenn sie sämtliche Reflexionsrichtungen mit ausreichender Genauigkeit erfasst. Beziehungsweise bezeichnet man EMs, die man nur aus bestimmten Richtungen betrachten kann, als *blickrichtungsabhängig* (view dependent). Objekte, die für ihre Spiegelung blickrichtungsunabhängige EMs verwenden, können also von allen Richtungen aus betrachtet werden und die Umgebung entsprechend widerspiegeln, ohne das die EMs neu erzeugt werden müssen (sofern sich die Umgebung nicht verändert).

Rekursive Reflexionen sind dadurch möglich, dass beim Generieren der Texturen für ein spiegelndes Objekt, alle anderen spiegelnden Objekte der Szene mit ihrer aktuellsten Textur gezeichnet werden. Dies hat zur Folge, dass diese Spiegelung in der Spiegelung bereits einen Frame alt ist, bzw. bei jeder weiteren Rekursion die Spiegelungen um jeweils einen weiteren Frame älter sind. Dieser Effekt ist jedoch unauffällig, da die Größe in der diese alten Spiegelungen in der endgültigen Szene gezeichnet werden mit der Rekursionstiefe abnimmt. Bei drastischen Änderungen der Umgebung kann dieser Effekt jedoch kurz wahrgenommen werden.

Environment Mapping funktioniert nicht gut für planare, spiegelnde Objekte. Auf planaren Flächen sind die Normalen konstant und da sich der Blickwinkel nur gering ändert, unterscheiden sich auch die Reflexionsrichtungen nur gering. Dies resultiert in einer unrealistisch wirkenden Spiegelung, die einen Vergrößerungseffekt zu haben scheint. Zeichnet man die Szene nicht mit einer perspektivischen, sondern mit einer orthographischen Projektion, so ist das Problem noch gravierender, da nun der Blickwinkel konstant ist und somit alle Reflexionsvektoren für die gesamte Fläche gleich sind und diese folglich einfarbig gezeichnet wird. Für Spiegelungen auf planaren Flächen sollten deshalb andere Verfahren als EM verwendet werden.

Environment Maps lassen sich nicht nur für Reflexionen einsetzen, sondern auch für Refraktionen [7]. Beim Zeichnen des texturierten Objekts muss lediglich nicht der Reflexionsvektor, sondern der Refraktionsvektor verwendet werden um die Texturkoordinaten zu berechnen. Dies funktioniert jedoch nur für infinitesimal dünne Oberflächen, bei denen der Eintrittspunkt des Lichtstrahls gleich dem Austrittspunkt ist. Z.B. sind Fensterscheiben oder Brillengläser dünn genug, um ihre Brechungen so zu simulieren. An dickeren Objekten, wie etwa einer Glaskugel, müssten zwei Lichtbrechungen (beim Eintritt und beim Austritt) berechnet werden. Die Richtung des Lichtstrahls hängt dann von der Position und den Normalen von Eintritts- und Austrittspunkt ab. Die hier vorgestellten EMs reichen deshalb nicht aus, um Refraktionen an dicken Objekten zu realisieren.

Im Folgenden werden verschiedene Verfahren untersucht, welche sich hinsichtlich der Parametrisierung der Umgebung in eine oder mehrere Texturen unterscheiden.

3.2 Kubisches Environment Mapping

Dieses Verfahren wurde 1986 von Ned Greene vorgestellt. Kubische Environment Maps speichern die Umgebung eines Punktes in sechs Texturen. Man erzeugt diese Texturen, indem eine Kamera auf dem Bezugspunkt positioniert wird und die Umgebung auf jede der sechs Seiten eines Würfels (daher *kubisches* EM) um den Bezugspunkt projiziert wird. Vergleiche Abbildung 3.3. Kubische EMs tasten die gesamte Umgebung ausreichend genau ab und sind somit blickrichtungsunabhängig.

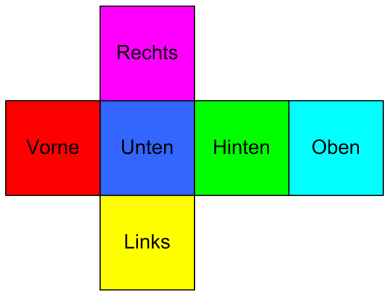

Abbildung 3.3: Kubische Environment Map, die in der Mitte eines Würfels erzeugt wurde

3.2.1 Erzeugen der Texturen

Mit diesem Verfahren lassen sich Texturen aus künstlichen als auch aus realen Umgebungen erzeugen. Ein großer Vorteil der kubischen Environment Maps ist, dass das Erzeugen aus künstlichen Umgebungen relativ einfach ist und im Prinzip mit jeder Graphikkarte durchgeführt werden kann, da die Projektion auf eine der Würfelseiten einer üblichen perspektivischen Projektion mit einem Blickwinkel von 90 Grad entspricht.

3.2.2 Zeichnen des texturierten Objekts

Welche der Texturen mit welchen Texturkoordinaten an einem Eckpunkt verwendet werden soll, kann durch folgende, einfache Rechenschritte bestimmt werden [1].

Durch Gleichung 3.1 wird pro Eckpunkt die Reflexionsrichtung berechnet (der Reflexionsvektor bräuchte für die folgenden Schritte nicht normalisiert sein). Die Komponente des Reflexionsvektors mit dem größten Betrag bestimmt, welche der Texturen verwendet wird, z.B. würde bei dem Reflexionsvektor $(-5.4, 3.2, -2.0)$ die „negative x-Seite“, also die linke Textur gewählt werden. Die Texturkoordinaten berechnen sich dann, indem die beiden anderen Komponenten durch den Betrag der größten Komponente geteilt werden. Bei $(-5.4, 3.2, -2.0)$ würden sich die Koordinaten $(s, t) = (3.2/5.4, -2.0/5.4) \approx (0.59, -0.37)$ ergeben. Der Gültigkeitsbereich dieser Koordinaten ist $-1 \leq s, t \leq 1$.

Liegen die Eckpunkte einer Kante auf verschiedenen Texturen, muss auf korrekte Interpolation geachtet werden. Wird kubisches Environment Mapping von der Graphikkarte unterstützt, so kann diese pro Pixel die richtige Textur auswählen und somit nahtlose Übergänge erreichen.

3.3 Sphärisches Environment Mapping

Dieses Verfahren wurde erstmals 1983 von Lance Williams vorgeschlagen und unabhängig von Gene S. Miller und C. Robert Hoffmann 1984 [10] entwickelt. Beim sphärischen Environment Mapping wird die Umgebung eines Punkts in einer einzelnen Textur gespeichert. Die Textur erhält man durch die orthographische Abbildung einer perfekt spiegelnden Kugel (daher *sphärisches* EM). Abbildung 3.4 zeigt eine sphärische EM.

Sphärische Environment Maps sind blickrichtungsabhängig, da der Bereich hinter der Kugel stark verzerrt auf den äußeren Kreis der Textur abgebildet wird und der Punkt genau hinter der Kugel eine Singularität darstellt und nicht in der Textur enthalten ist. Dieses Verfahren ist also nur einsetzbar, wenn die Blickrichtung beim Zeichnen gleich der Blickrichtung beim Erzeugen der Textur ist.

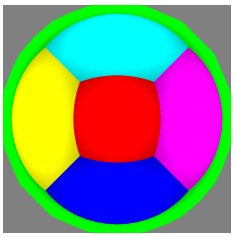

Abbildung 3.4: Sphärische Environment Map, die in der Mitte eines Würfels erzeugt wurde

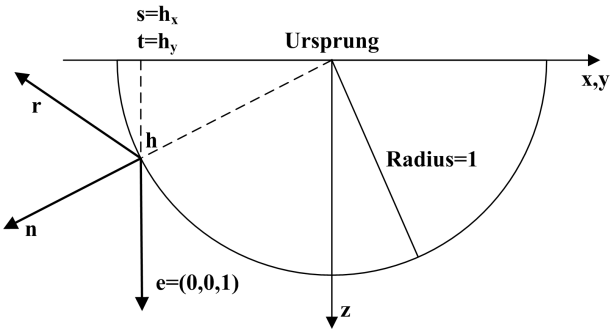

Abbildung 3.5: Berechnung der Texturkoordinaten s und t

Aus Gleichung 3.1, bzw. Gleichung 3.2 kann man berechnen, auf welche Texturkoordinaten eine bestimmte Reflexionsrichtung r abgebildet wird. Wählt man wie in Abbildung 3.5 als reflektierende Kugel eine Einheitskugel um den Ursprung und blickt entlang der negativen z-Achse, also $e = (0, 0, 1)$, so gleichen die Koordinaten des Schnittpunkts h denen der Normalen n. Die gesuchten Texturkoordinaten s und t lassen sich dann durch

$$m = \sqrt{r_x^2 + r_y^2 + (1 + r_z)^2} \tag{3.3}$$

$$s = \frac{r_x}{m} \tag{3.4}$$

$$t = \frac{r_y}{m} \tag{3.5}$$

berechnen. Wobei hier der gültige Bereich der Textur $-1 \leq s, t \leq 1$ ist. Soll für die Textur $0 \leq s, t \leq 1$ gelten, so müssen Gleichungen 3.4 und 3.5 durch

$$s = \frac{r_x}{2m} + 0.5 \tag{3.6}$$

$$t = \frac{r_y}{2m} + 0.5 \tag{3.7}$$

ersetzt werden.

3.3.1 Erzeugen der Textur

Eine sphärische Environment Map lässt sich mit relativ geringem Aufwand aus realen Umgebungen erzeugen. Hierzu wird eine perfekt spiegelnde Kugel an die gewünschte Position gebracht und ein

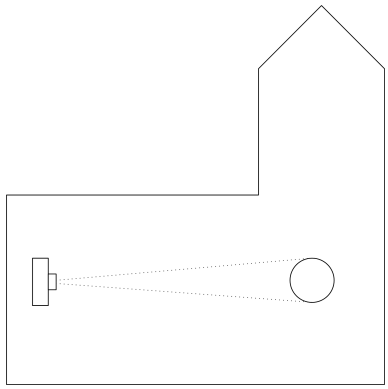

Abbildung 3.6: Aufbau, um photographisch eine sphärische Environment Map zu erzeugen

Photo mit einer Kamera mit starkem Teleobjektiv aus großer Entfernung gemacht. Vergleiche Abbildung 3.6. Die große Entfernung und das Teleobjektiv sind wichtig, um einer orthographischen Projektion möglichst nahe zu kommen.

Die Erzeugung der Textur aus einer künstlichen Umgebung lässt sich entweder durch Raytracing, durch die Transformation von kubischen Environment Maps oder durch ein Verzerren der Szene und Zeichnen in eine Textur erreichen. Durch Raytracing kann man die beste Qualität erreichen und für die Erzeugung statischer EMs ist es gut geeignet. Durch den hohen Rechenaufwand ist es jedoch nicht für dynamische EMs geeignet. Durch die Transformation kubischer EMs lassen sich qualitativ hochwertige sphärische EMs erzeugen, für dynamische Zwecke ist jedoch auch hier der Rechenaufwand zu hoch. Besonders interessant für die dynamische Erzeugung ist das Verzerren der Szene, da sich diese durch einen entsprechenden Vertex Shader bewerkstelligen lässt und nur einen zusätzlichen Zeichendurchlauf benötigt.

Erzeugung durch Verzerren der Szene

Verzerren der Szene bedeutet hier, alle Eckpunkte der Szene so zu transformieren, dass man durch folgendes Zeichnen eine sphärische Environment Map erhält. Hierzu muss die Szene zunächst so transformiert werden, dass sich der Betrachter auf der Position des Bezugspunktes befindet. Die Blickrichtung darf nicht verändert werden, damit diese beim Erzeugen und Anwenden der Textur gleich ist. Dann kann man die x-Komponente der Position eines Eckpunkts durch Gleichung 3.4 und die y-Komponente durch Gleichung 3.5 berechnen. Die Reflexionsrichtung r in den Gleichungen ist hier der normalisierte Vektor vom Bezugspunkt zum Eckpunkt.

Die Qualität von sphärischen Environment Maps die durch Verzerren generiert wurden hängt stark von der Feinheit der Tesselierung der Szene ab. Problematisch ist nämlich, dass Kanten, die nicht parallel zur z-Achse verlaufen, ursprünglich korrekt als Liniensegmente gezeichnet werden, nach der Verzerrung jedoch gekrümmt sein müssten. Da man jedoch lediglich die Eckpunkte transformiert, wird in der verzerrten Szene eine solche Kante auch als Liniensegment gezeichnet. Vergleiche Abbildung 3.7.

Der Tiefenwert, der bei der üblichen perspektivischen und orthographischen Projektion verwendet wird, kann hier nicht benutzt werden, da auch Objekte, die sich hinter dem Bezugspunkt befinden, gezeichnet werden sollen. In OpenGL wird der Tiefenwert bei der üblichen perspektivischen und orthographischen Projektion durch

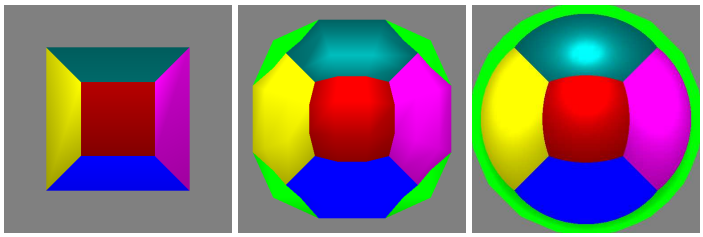

Abbildung 3.7: Sphärische Environment Maps, erzeugt im Inneren eines Würfels mit verschieden feiner Tesselierung; von links nach rechts: 2, 4, 21 Eckpunkte pro Würfelkante

$$depth = \frac{f+n}{f-n} - \frac{2fn}{f-n} \cdot \frac{1}{z} \tag{3.8}$$

berechnet. Wobei n die Entfernung des Betrachters zur near clipping plane ist und f die zur far clipping plane. Dadurch erhalten Punkte auf der near clipping plane den Tiefenwert -1 und Punkte auf der far clipping plane den Tiefenwert 1.

Ersetzt man in Gleichung 3.8 den Abstand entlang der z-Achse z durch den Abstand zum Bezugspunkt $\sqrt{x^2+y^2+z^2}$, so erhält man eine sinnvolle Tiefenwertfunktion:

$$depth = \frac{f+n}{f-n} - \frac{2fn}{f-n} \cdot \frac{1}{\sqrt{x^2+y^2+z^2}} \tag{3.9}$$

Diese Gleichung erhält die positive Eigenschaft, Abstände, die näher am Betrachter sind mit höherer Genauigkeit aufzulösen, als solche, die weiter entfernt sind. Anschaulich gesehen, wird das near und far clipping jetzt nicht mehr an Ebenen, sondern an Kugeloberflächen durchgeführt. Hierbei beschreibt n den Radius der „near clipping sphere“ und f den der „far clipping sphere“.

Kanten, die genau hinter dem Bezugspunkt liegen, also für die $r_z \approx -1$ gilt, dürfen nicht gezeichnet werden, da diese sich über die gesamte Textur erstrecken können, siehe obere Bilder in Abbildung 3.8. Dies kann erreicht werden, indem Eckpunkte, die im Bereich eines Kegels, dessen Spitze auf dem Bezugspunkt liegt und in Richtung des Betrachters zeigt, liegen, entfernt werden. Der Kegelbereich lässt sich durch $-1+\epsilon \geq r_z$ beschreiben. Wie ϵ gewählt werden sollte, hängt davon ab, wie fein die Szene tesselliert ist. Bei der Implementierung der Demo-Anwendung hat sich $\epsilon = 0.05$ als sinnvoller Wert herausgestellt. Das Entfernen lässt sich elegant lösen, indem die homogene Komponente gleich 0 gesetzt wird, also diese Punkte unendlich weit weg verschoben werden. Nach dem Clipping werden diese Kanten zwar immer noch gezeichnet, jedoch beeinflussen sie nicht mehr den relevanten Bereich der Textur. Siehe untere Bilder in Abbildung 3.8.

Ein Problem stellt die Interpolation der Tiefenwerte dar, da lineare Interpolation nicht korrekt ist. In Abbildung 3.9 sind AB und CD zwei Kanten in der Szene, wobei CD von AB verdeckt wird. Interpoliert man die aus Gleichung 3.9 berechneten Tiefenwerte linear, so hat CD kleinere Tiefenwerte als AB, CD wird also nicht verdeckt. Je länger eine Kante ist und je näher sie beim Betrachter liegt, desto größer ist der Abstandsfehler. Der Abstandsfehler einer Kante der Länge l, die parallel zur xy-Ebene verläuft und deren Eckpunkte den Abstand d zum Bezugspunkt haben, ist in der Mitte der Kante am größten, somit lässt sich durch folgende Gleichung die obere Grenze des Fehlers berechnen.

$$Abstandsfehler_{max} = d - \sqrt{d^2 - 0.25l^2} \tag{3.10}$$

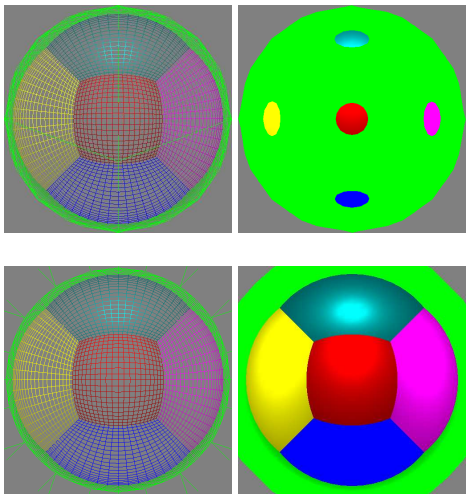

Abbildung 3.8: Kanten, die sich im Bereich der Singularität befinden können sich über die gesamte Textur erstrecken (oben). Man kann dies verhindern, indem man sie unendlich weit weg verschiebt (unten).

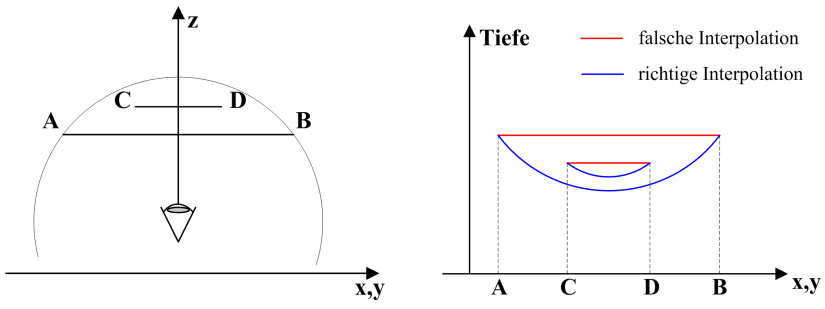

Abbildung 3.9: Interpolation der Tiefenwerte

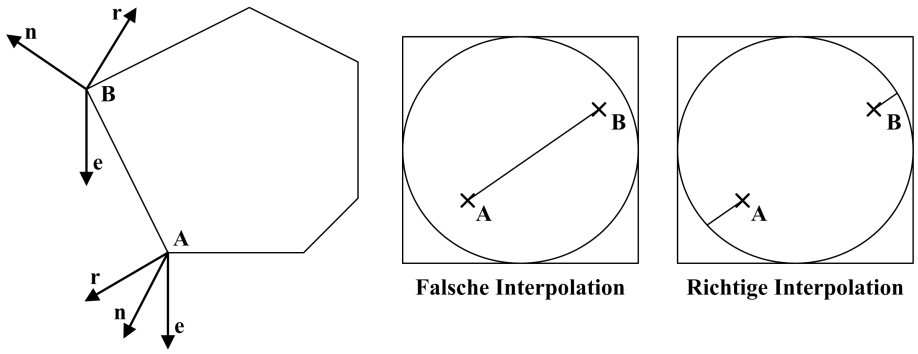

Abbildung 3.10: An der Silhouette von spiegelnden Objekten können Artefakte wegen falscher Interpolation entstehen.

3.3.2 Zeichnen des texturierten Objekts

Für die Texturierung des spiegelnden Objekts wird pro Eckpunkt der Reflexionsvektor r durch Gleichung 3.1 berechnet. Die Texturkoordinaten werden durch die Gleichungen 3.6 und 3.7 berechnet.

Moderne Graphikkarten unterstützen die automatische Generierung von Texturkoordinaten für sphärische Environment Maps. Alternativ kann die Berechnung auch in einem einfachen Vertex Shader implementiert werden. Die Berechnung in der Anwendung durchzuführen und das Objekt mit den aktualisierten Texturkoordinaten zu zeichnen ist auch möglich, die beiden anderen Möglichkeiten sind jedoch wesentlich effizienter und deshalb vorzuziehen.

Auch hier ist es problematisch, dass die lineare Interpolation auf der Textur nicht korrekt ist. Der Fehler wird am Rand der Textur größer. Je feiner das spiegelnde Objekt tesseliert ist, desto geringer ist dieser Fehler. Durch Berechnung des Reflexionsvektors und der korrekten Texturkoordinaten pro Pixel in einem Pixel Shader kann eine korrekte Interpolation implementiert werden.

Die Singularität stellt beim Texturieren meist kein Problem dar, da sie auf dem äußeren Kreis der Textur liegt und alle Texturkoordinaten innerhalb dieses Kreises liegen. Daher gibt es keine Kanten, die den Kreis schneiden. Probleme können an der Silhouette des Objekts auftreten [9], nämlich bei Kanten, die einen Eckpunkt haben dessen Normale vom Betrachter weg zeigt (also $n_z < 0$). Artefakte entstehen dadurch, dass die Texturkoordinaten einer solchen Kante quer über die Textur interpoliert werden. Korrekt ist es, über den Rand der Textur hinweg zu interpolieren. Siehe Abbildung 3.10.

Enthält eine Szene mehrere reflektierende Objekte, so kann die Singularität des einen Objekts in der Spiegelung eines anderen sichtbar werden. Siehe Abbildung 3.11. Dieses Problem lässt sich nicht vermeiden, jedoch sind die sichtbaren Artefakte im Allgemeinen klein.

3.4 Parabolisches Environment Mapping

Parabolisches Environment Mapping wurde 1998 von Wolfgang Heidrich und Hans-Peter Seidel [6] vorgestellt. Bei diesem Verfahren wird die Umgebung eines Punkts in zwei Texturen gespeichert. Die Idee ist ähnlich wie beim sphärischen EM, nur das hier nicht eine Kugel verwendet wird um den gesamten Raum abzudecken, sondern zwei Paraboloide die jeweils einen Halbraum abdecken. Die Oberfläche eines solchen parabolischen Spiegels wird durch die Funktion

$$f(x, y) = 0.5 - 0.5(x^2 + y^2) \qquad x^2 + y^2 \leq 1 \tag{3.11}$$

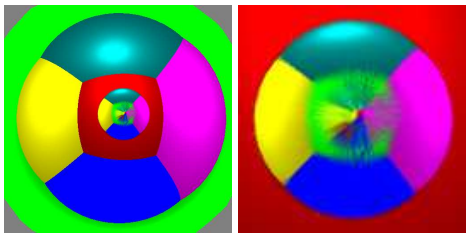

Abbildung 3.11: Sphärische Environment Map, in der die Singularität eines anderen spiegelnden Objekts sichtbar wird; rechts ist der interessante Ausschnitt vergrößert

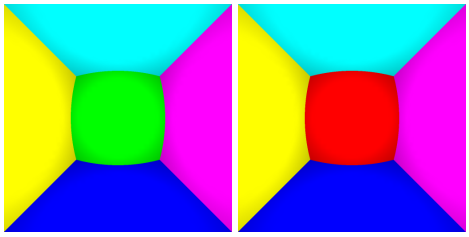

Abbildung 3.12: Parabolische Environment Maps, die in der Mitte eines Würfels erzeugt wurden

beschrieben. Parabolische EMs tasten die gesamte Umgebung ausreichend genau ab und sind somit blickrichtungsunabhängig.

Es ist jedoch im Allgemeinen sinnvoll, den Gültigkeitsbereich von $x^2+y^2 \leq 1$ auf $-1 \leq x, y \leq 1$ zu vergrößern, um Artefakte beim Überblenden zwischen der beiden Texturen zu vermeiden. Dies bedeutet, dass auch manche Objekte, die hinter dem jeweiligen Paraboloiden liegen, gezeichnet werden. Abbildung 3.12 zeigt zwei solche Texturen mit vergrößertem Gültigkeitsbereich. Der kleinere Gültigkeitsbereich entspricht jeweils dem kreisförmigen (Kreis mit Durchmesser 1) Ausschnitt der Texturen.

Aus Gleichung 3.1 folgt, auf welche Texturkoordinaten eine bestimmte Reflexionsrichtung abgebildet wird. Ist der Gültigkeitsbereich der Textur $-1 \leq s, t \leq 1$ ist, so gilt für die vordere Textur:

$$s = \frac{r_x}{1+r_z} \tag{3.12}$$

$$t = \frac{r_y}{1+r_z} \tag{3.13}$$

Soll für die Textur $0 \leq s, t \leq 1$ gelten, so gilt:

$$s = \frac{r_x}{2+2r_z} + 0.5 \tag{3.14}$$

$$t = \frac{r_y}{2+2r_z} + 0.5 \tag{3.15}$$

Die Koordinaten für die hintere Textur berechnen sich genauso, nur mit gedrehtem Vorzeichen von r_z.

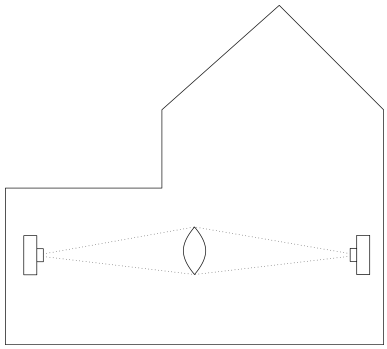

Abbildung 3.13: Aufbau, um photographisch parabolische Environment Maps zu erzeugen

3.4.1 Erzeugen der Texturen

Die Texturen können ähnlich wie beim sphärischen Environment Mapping mithilfe der Photographie aus realen Umgebungen erstellt werden. Hierzu verwendet man statt einer spiegelnden Kugel zwei aneinander liegende parabolische Spiegel, deren Oberflächen durch die Funktion 3.11 beschrieben sind. Zur Erstellung der Texturen muss von jedem Paraboloiden ein Photo geschossen werden, das zweite genau aus der Gegenrichtung des ersten. Siehe Abbildung 3.13. Jedes dieser Bilder enthält dann genau einen Halbraum der Umgebung, bzw. etwas darüber hinaus, wenn der quadratische Gültigkeitsbereich verwendet wird. Dieses Vorgehen ist aufwändiger als beim sphärischen EM, da das erforderliche zweite Photo aus der Gegenrichtung viel zusätzlichen Platz voraussetzt. So ist es wohl in vielen Gebäuden vom Platz her möglich, sphärische EMs zu erstellen, jedoch keine parabolischen. Auch die Herstellung von brauchbaren parabolischen Spiegeln ist schwieriger als die von kugelförmigen Spiegeln.

Eine andere Möglichkeit zur Erstellung von parabolischen Environment Maps von realen Umgebungen wurde von Nayar [11] vorgestellt. Nayar hat hierbei eine kompakte Konstruktion aus Kameras und Linsen vorgestellt, welche es ermöglicht diese Art von Bildern zu erstellen.

Die Erzeugung der Texturen aus künstlichen Umgebungen lässt sich (analog zur Erzeugung der sphärischen EMs) entweder durch Raytracing, durch die Transformation von kubischen EMs oder durch ein Verzerren der Szene und Zeichnen in Texturen erreichen.

Erzeugen durch Verzerren der Szene

Die Idee ist hier die gleiche, wie beim Erzeugen der sphärischen Environment Maps, siehe also auch Kapitel 3.3.1. Verzerren der Szene bedeutet hier also, alle Eckpunkte der Szene so zu transformieren, dass man durch folgendes Zeichnen eine parabolische EM erhält. Auch hier lässt sich die Verzerrung durch einen entsprechenden Vertex Shader bewerkstelligen. Die x-Komponente der Position eines Eckpunkts berechnet sich durch Gleichung 3.12 und die y-Komponente durch Gleichung 3.13. Die Szene muss für die beiden Texturen unterschiedlich verzerrt werden, weshalb zwei zusätzliche Zeichendurchläufe benötigt werden.

Eine feine Tesselierung ist auch hier notwendig. Da die Verzerrung jedoch nicht so stark wie bei sphärischen EMs ist, reicht eine weniger feine Tesselierung aus, siehe Abbildung 3.14.

Möchte man nur den Bereich innerhalb des Kreises zeichnen, so kann die Tiefenwertfunktion der üblichen orthographischen und perspektivischen Projektion (Gleichung 3.8) verwendet werden. Um den Bereich $-1 \leq x, y \leq 1$ zeichnen zu können, welcher für einen nahtlosen Übergang beim Zeich-

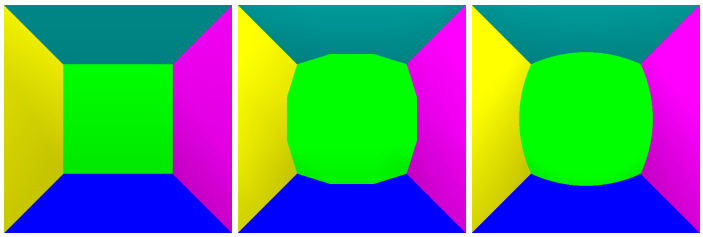

Abbildung 3.14: Parabolische Environment Maps, erzeugt im Inneren eines Würfels mit verschieden feiner Tesselierung; von links nach rechts: 2, 4, 21 Eckpunkte pro Würfelkante

nen des texturierten Objekts nötig ist, reicht diese Tiefenwertfunktion nicht mehr aus. Die Funktion, die bei der Erzeugung der sphärischen EM verwendet wurde, ist hierfür gut geeignet, siehe Gleichung 3.9. Das Problem bezüglich der Interpolation der Tiefenwerte ist hier das gleiche.

Wie bei sphärischen EMs, dürfen auch hier Kanten, die genau hinter dem Bezugspunkt liegen, nicht gezeichnet werden. Der Grund ist auch hier, dass für die vordere Textur Kanten mit Eckpunkten, für die $r_z \approx -1$ ($r_z \approx 1$ bei der hinteren Textur) gilt, sich über die gesamte Textur erstrecken können. Dies kann verhindert werden, indem bei der vorderen Textur Eckpunkte für die $-1 + \epsilon \geq r_z$ ($1 - \epsilon \leq r_z$ bei der hinteren Textur) gilt, unendlich weit weg verschoben werden, wobei ϵ hier größer als bei sphärischen EMs gewählt werden darf, da nur ein geringer Teil der hinter dem Bezugspunkt liegenden Objekte innerhalb der Textur landet. Bei der Implementierung der Demo-Anwendung hat sich $\epsilon = 0.5$ als sinnvoller Wert herausgestellt.

3.4.2 Zeichnen des texturierten Objekts

Für die Texturierung des spiegelnden Objekts wird pro Eckpunkt oder Pixel der Reflexionsvektor r durch Gleichung 3.1 berechnet. Gilt $r_z \geq 0$, so wird die vordere Textur verwendet, andernfalls die hintere. Die Texturkoordinaten für die vordere Textur werden durch Gleichungen 3.14 und 3.15 berechnet. Die Koordinaten für die hintere Textur berechnen sich genauso, nur mit gedrehtem Vorzeichen von r_z.

Ein angenehmer Aspekt dieser Gleichungen ist, dass sie lediglich Addition, Multiplikation und Divisionen benutzen. Deshalb ist es möglich, sie auf Graphikhardware berechnen zu lassen, welche das automatische Generieren von Reflexionsvektoren als Texturkoordinaten unterstützt. Durch setzen der entsprechenden Transformationsmatrizen kann man dann die gewünschten Texturkoordinaten erhalten. Die Transformationsmatrix muss für die vordere Textur wie folgt gesetzt werden:

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} r_x \\ r_y \\ r_z \\ 1 \end{pmatrix} \tag{3.16}$$

Und für die hintere Textur:

$$\begin{pmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & -2 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} r_x \\ r_y \\ r_z \\ 1 \end{pmatrix} \tag{3.17}$$

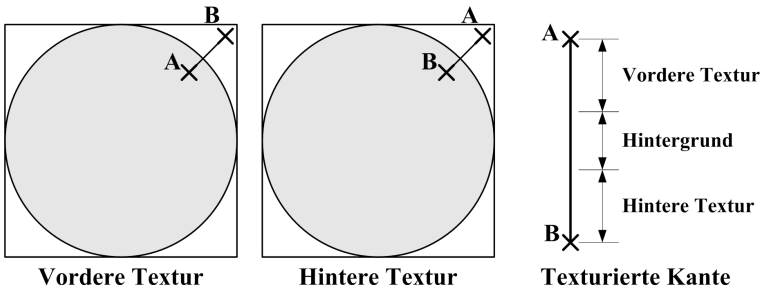

Abbildung 3.15: Überblenden der beiden Texturen

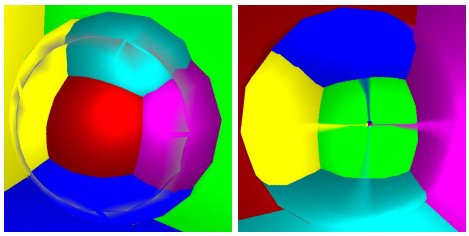

Abbildung 3.16: Artefakte wegen linearer Interpolation und Verwendung des kreisförmigen Gültigkeitsbereichs (links); Artefakte wegen falscher $alpha$-Werte (rechts)

Besonderes Augenmerk verdient der Übergangsbereich zwischen der vorderen und der hinteren Textur. Verwendet man für Gleichung 3.11 den Gültigkeitsbereich $x^2 + y^2 \leq 1$, so wird in die vordere Textur der Bereich $r_z \geq 0$ und in die hintere der Bereich $r_z \leq 0$ gezeichnet. Die Ebene $r_z = 0$ wird auf den äußeren Kreis beider Texturen abgebildet. Seien nun A und B zwei benachbarte Eckpunkte eines Polygons der Szene, wobei $r_z(A) > 0$ und $r_z(B) < 0$. A liegt also auf der vorderen Textur und B auf der hinteren. Siehe Abbildung 3.15. Bei linearer Interpolation der Texturkoordinaten gibt es einen Bereich, der außerhalb des äußeren Kreises beider Texturen liegt und typischerweise Flecken in der Hintergrundfarbe im Übergangsbereich erzeugt. Siehe linkes Bild in Abbildung 3.16.

Da das Problem daher rührt, dass die lineare Interpolation der Texturkoordinaten nicht korrekt ist, kann man dieses Problem beseitigen, indem man in einem Pixel Shader die Texturkoordinaten pro Pixel berechnet.

Eine andere Möglichkeit besteht darin, nicht nur den kreisförmigen Bereich der Textur zu füllen, sondern das gesamte Quadrat. Dann gibt es keinen Bereich mehr, der von keiner Textur abgedeckt wird. Es gibt dann zwar immer noch Artefakte an der Nahtstelle zwischen den beiden Texturen, jedoch sind diese kaum zu erkennen - besonders dann nicht, wenn das spiegelnde Objekt fein tesseliert ist.

Zum Mischen der beiden Texturen kann unter OpenGL der Modus GL_INTERPOLATE verwendet werden, um die Funktion $alpha \times VordereTextur + (1 - alpha) \times HintereTextur$ zu implementieren. $alpha$ muss hierbei gleich 1 sein, wenn die vordere Textur verwendet werden soll (also wenn $r_z \geq 0$), ansonsten gleich 0.

Verwendet man für das Setzen des $alpha$-Wertes die $alpha$-Komponente der vorderen Textur, so muss man beachten, dass wirklich für alle Texturkoordinaten, die außerhalb des Kreises liegen $alpha = 0$ gilt. Beschränkt man die Texturkoordinaten auf den Bereich $0 \leq s, t \leq 1$ (z.B. in OpenGL durch den Modus GL_CLAMP), so werden Koordinaten die größer 1 sind auf 1, und alle die kleiner

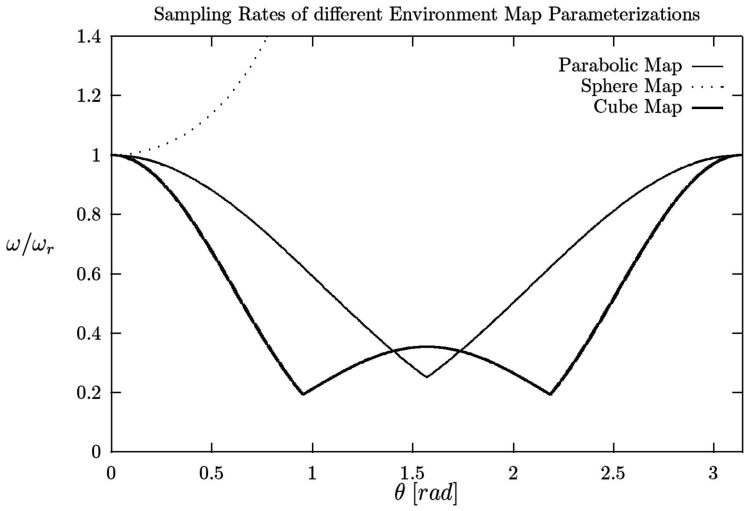

Abbildung 3.17: Abtastraten von parabolischen, sphärischen und kubischen Environment Maps [7]

0 sind, auf 0 abgebildet. Problematisch sind nun Texturkoordinaten, die außerhalb der Textur liegen und auf Schnittpunkte des Kreises mit dem Rand der Textur abgebildet werden. Z.B. werden die Texturkoordinaten $(2.4, 0.0)$ auf $(1.0, 0.0)$ abgebildet, was genau auf dem Kreis liegt, also gilt hier $alpha = 1$, obwohl die Koordinaten weit außerhalb des Kreises liegen. Dies führt dazu, dass alle Punkte, für deren Reflexionsvektoren $r_x = 0$ oder $r_y = 0$ gilt, $alpha = 1$ ist, und somit die vordere Textur verwendet wird, siehe rechtes Bild in Abbildung 3.16. Löst man das Problem, indem man sicherstellt, dass für den gesamten Rand der Textur $alpha = 0$ gilt, so muss man die Textur um einen Rand erweitern auf den außerhalb liegende Koordinaten abgebildet werden, oder der Kreis muss entsprechend verkleinert werden, d.h. alle Texturkoordinaten entsprechend skaliert werden. Es ist dann außerdem zu beachten, dass Texturfilter diesen Rand nicht beeinflussen dürfen.

Dieses Problem kann umgangen werden, wenn man einen Vertex Shader verwendet und den $alpha$-Wert der Farbe jedes Eckpunkts abhängig vom Reflexionsvektor setzt, also $alpha = 1$ bei $r_z \geq 0$ und $alpha = 0$ bei $r_z < 0$. Das bedeutet, dass man den $alpha$-Wert nicht beim Erzeugen der Texturen berechnet, sondern erst beim Zeichnen des spiegelnden Objekts.

3.5 Vergleich der Abtastraten

Abbildung 3.17 vergleicht die Abtastraten von parabolischen, sphärischen und kubischen Environment Maps. θ ist hier der Winkel zwischen der negativen z-Achse und dem Reflexionsstrahl. ω_r ist der Raumwinkel, den der Pixel genau hinter dem Betrachter einnimmt, welcher als Referenz dient. ω ist der Raumwinkel, den ein Pixel bei dem Winkel θ einnimmt. ω/ω_r beschreibt also, wie sich die Abtastrate bei einem bestimmten Winkel zur Abtastrate beim Referenzpixel verhält. Die Abbildung zeigt die äußerst ungleichmäßige Abtastung beim sphärischen EM, bei den beiden anderen Verfahren ist die Abtastung recht gleichmäßig, wobei die des parabolischen EM etwas stabiler als die des kubischen EM ist.

3.6 Mipmap Generierung

Um Aliasingeffekte zu vermeiden, können die üblichen Filteralgorithmen, wie z.B. Mipmapping, eingesetzt werden. Bei allen drei Verfahren (kubisches, sphärisches und parabolisches EM) wird ein Texel einer Mipmapstufe aus der *gewichteten* Summe des entsprechenden 2x2 Texelblocks der darunter liegenden Stufe berechnet. Die Gewichtung erfolgt anhand der Größe des jeweils zugehörigen Raumwinkels. Die Summe der vier Raumwinkel ist der Raumwinkel, den der berechnete Texel abdeckt. Auf diese Weise können alle Mipmapstufen korrekt berechnet werden. Anisotrop gefilterte Texturen können analog erzeugt werden. Bei parabolischen EMs ist es unbedingt notwendig, dass der gesamte Bereich der Textur definiert ist [7].

Problematisch bei der korrekten Mipmapberechnung ist die fehlende Hardwareunterstützung. Die unter OpenGL mittels automatischer Mipmap Generierung erzeugten Mipmaps sind nicht korrekt, da hier jedem Texel einer Stufe das gleiche Gewicht zugeteilt wird. Bei dynamischen EMs ist das Erzeugen korrekter Mipmaps jedoch im Allgemeinen zu teuer und man greift deshalb trotzdem zum automatischen Mipmapping [8].

Kapitel 4

Implementierung einer Demo-Anwendung

Um den Einsatz von dynamisch erzeugten sphärischen und parabolischen Environment Maps zu testen und zu demonstrieren, wurde die Demo-Anwendung dynenvmaps implementiert. Das Programm zeigt eine Szene, in die über eine GUI Objekte eingefügt werden können. Diese Objekte können beliebig interaktiv transformiert werden und es kann für jedes eingestellt werden, ob es spiegelnd sein soll, oder nicht. Es lassen sich verschiedene Umgebungen für diese Objekte auswählen. Beim Programmstart kann über einen Parameter festgelegt werden, ob parabolische (Standardeinstellung) oder sphärische EMs verwendet werden sollen. Gleichzeitige Verwendung von sphärischen und parabolischen Maps ist nicht möglich.

4.1 Bedienung

4.1.1 Bedienelemente

Abbildung 4.1 zeigt einen Schnappschuss des Programms. An der rechten Seite des Fensters befinden sich die Bedienelemente. Hier eine kurze Beschreibung der Elemente von oben nach unten:

- Select Object: Hiermit lässt sich das aktive Objekt auswählen. Zur Auswahl stehen sämtliche Objekte der Szene, die nicht zur Umgebung gehören.

- Object Transformation: Hiermit lässt sich das aktive Objekt transformieren. Mögliche Transformationen sind Rotation, Translation und Skalierung.

- Object Options: Mit der Checkbox „reflecting“ lässt sich einstellen, ob das aktive Objekt spiegelnd sein soll oder nicht. Die Checkbox „dummy textures“ ist nur verfügbar, wenn parabolische EMs verwendet werden, ist diese aktiv, wird statt der erzeugten Texturen eine einfarbig rote Textur als vordere und eine einfarbig türkise als hintere Textur verwendet. Durch den Knopf „save textures“ werden die erzeugten Texturen eines spiegelnden Objekts im Arbeitsverzeichnis als TGA-Bilder gespeichert.

- Add Object: Durch Auswählen eines Objekts aus dieser Liste, wird dieses der Szene hinzugefügt. Dieses ist fortan im “Select Object” Menü verfügbar.

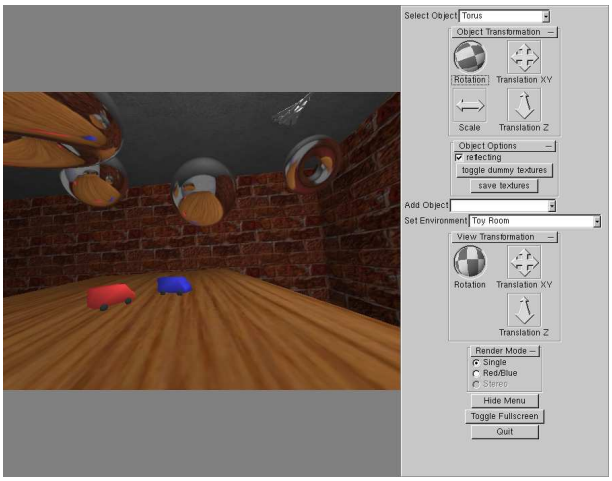

Abbildung 4.1: Schnappschuss von dynenvmaps

- Set Environment: Hier können verschiedene, vorgefertigte Umgebungen ausgewählt werden. Wählt man eine andere Umgebung, bleiben die hinzugefügten Objekte unverändert erhalten. Zur Auswahl stehen verschieden fein tesselierte, farbige Würfel, ein Drahtgitterwürfel und ein Kellerraum mit einigen animierten Objekten.

- View Transformation: Hiermit lässt sich die Kamera transformieren. Mögliche Transformationen sind Rotation und Translation.

- Render Mode: Standardeinstellung ist der „Single“-Modus, in dem die Szene wie üblich einmal gezeichnet wird. Im „Red/Blue“-Modus wird die Szene zweimal gezeichnet. Einmal rot für das linke Auge und einmal türkis für das rechte Auge. Trägt man nun eine passende Stereobrille, so erhält man einen 3-dimensionalen Eindruck der Szene. Verfügt man über eine Stereoprojektionswand, so kann man das Programm mit dem Parameter „–stereo“ starten und somit den „Stereo“-Modus auswählen. Hier wird die Szene einmal für das linke Auge in den linken Bildspeicher und einmal für das rechte Auge in den rechten Bildspeicher gezeichnet. Die Farben werden hier nicht verändert.

- Hide Menu (Shortcut: H): Hiermit kann das Menü ausgeblendet werden und die Szene wird in voller Fenstergröße gezeichnet.

- Toggle Fullscreen (Shortcut F): Hiermit wird die Fenstergröße maximiert, so dass das Fenster den gesamten Desktop einnimmt, bzw. wird es auf die ursprüngliche Größe reduziert, wenn es bereits maximiert war.

- Quit (Shortcut: Q): Schließt das Fenster und beendet die Anwendung.

4.1.2 Parameter

Es gibt eine Reihe von Parametern, durch die sich das Programm beim Start konfigurieren lässt. Startet man das Programm mit dem Parameter `--help`, so erhält man eine Liste aller möglichen Parameter und eine englische Kurzbeschreibung (siehe Listing 4.1).

```
Usage: ./dynenvmaps [OPTIONS]
Options:
  --help            display this help and exit
  --stereo          enable output suitable for stereo projectors
  --empty           shows an empty cube as initial scene
  --demo            non-interactive camera (camera flies
                    through the scene automatically)
  --fps             print the current fps every few seconds
  --sphere          use spherical maps instead of parabolic ones
  --tex64           set parabolic/spherical map size to 64x64
  --tex128          set parabolic/spherical map size to 128x128
  --tex256          set parabolic/spherical map size to 256x256
                    (this is the default setting)
  --tex512          set parabolic/spherical map size to 512x512
  --tex1024         set parabolic/spherical map size to 1024x1024
  --tracker         enable tracking devices (glasses and pen)
  --tracker-pen     enable just the pen tracking device
  --tracker-glasses enable just the glasses tracking device
```

Listing 4.1: Beschreibung der Parameter von dynenvmaps

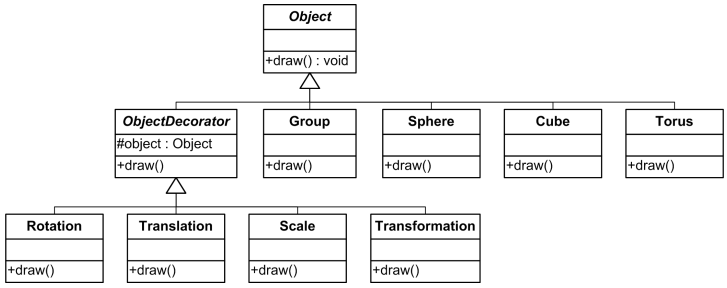

Abbildung 4.2: Ausschnitt der Klassenhierarchie für den Szenengraphen

4.2 Implementierung

Das Programm ist in C++ geschrieben und verwendet, neben den Standardbibliotheken, die Bibliotheken OpenGL, glu, glut und glui (Bibliothek für die Bedienelemente). Es wurde auf aktuellen Graphikkarten von Nvidia und von ATI getestet. Entwickelt wurde es ausschließlich für die Linux-Plattform.

4.2.1 Szenengraph

Es wurde eine Objekthierarchie für einen Szenengraphen implementiert. Hierbei wurden die Entwurfsmuster Composite und Decorator [5] verwendet. Siehe Abbildung 4.2. Die Klasse Object ist abstrakt und definiert lediglich die rein virtuelle Methode draw. Alle Objekte im Szenengraph sind von dieser Klasse abgeleitet. Die Klasse Group ist ein Container und enthält Zeiger auf beliebig viele Objekte, bei einem Aufruf von draw ruft Group die draw Funktion aller enthaltenen Objekte auf. Dies ist eine Anwendung des Composite Entwurfsmusters. Das Decorator Entwurfsmuster wurde verwendet, um die Transformationen zu implementieren. D.h. diese Klassen besitzen einen Zeiger auf das zu transformierende Objekt. Bei einem Aufruf von draw wird die entsprechende Transformation ausgeführt und das betroffene Objekt anschließend gezeichnet.

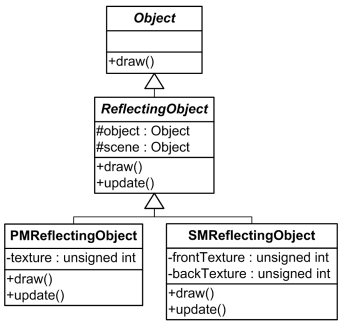

Abbildung 4.3: Integration der reflektierenden Objekte in den Szenengraphen

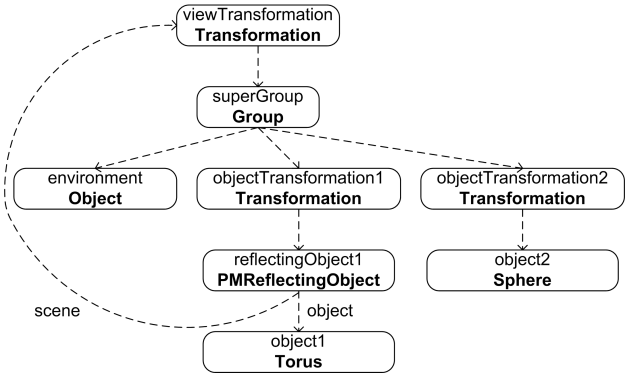

Abbildung 4.4: Zusammenhang der Instanzen, um die Szene zu repräsentieren

Parabolisch spiegelnde Objekte werden durch die Klasse PMReflectingObject (PM steht für paraboloid map) und sphärisch spiegelnde Objekte durch die Klasse SMReflectingObject (SM steht für sphere map) realisiert. Beide sind von der abstrakten Klasse ReflectingObject abgeleitet. Siehe Abbildung 4.3. Das Attribut object zeigt auf das Objekt, das spiegeln soll und scene zeigt auf die Umgebung, welche in der Spiegelung enthalten sein soll. Die Funktion update aktualisiert die EMs, indem scene entsprechend verzerrt in die privaten Texturen (texture bei SMReflectingObject, frontTexture und backTexture bei PMReflectingObject) gezeichnet wird.

In Abbildung 4.4 ist der Szenengraph dargestellt, wie er zur Laufzeit in der Anwendung verwendet wird. Die Wurzel ist die Transformation viewTransformation, welche die Group superGroup betrifft, in der alle restlichen Objekte der Szene enthalten sind. Fügt man ein Objekt der Szene hinzu, so wird dieses durch eine Transformation dekoriert und der superGroup hinzugefügt. Soll ein Objekt spiegelnd sein, so wird eine Instanz von SMReflectingObject, bzw. PMReflectingObject erzeugt und zwischen der Objekttransformation und dem Objekt eingehängt. Das Attribut scene zeigt immer auf die Wurzel des Szenengraph, da stets sämtliche Objekte der Szene in den Reflexionen enthalten sein sollen.

4.2.2 Funktionen

In diesem Kapitel werden die wichtigsten Funktionen, die mit der Erzeugung und Anwendung der Environment Maps zu tun haben, vorgestellt. Die folgenden Listings enthalten nur die wichtigsten Schritte, sind also keineswegs vollständig. Die größten Teile wurden 1:1 aus dem Quelltext der An-

wendung übernommen, teilweise wurden jedoch Variablen- oder Funktionsnamen geändert und weniger wichtige Teile weggelassen um den Rahmen dieser Arbeit nicht zu sprengen.

Vor dem finalen Zeichnen der Szene werden die EMs aller reflektierender Objekte der Szene aktualisiert. Der Vektor reflectingObjects enthält Zeiger auf alle SMReflectingObjects, bzw. PMReflectingObjects der Szene. Von all diesen Objekten wird die update Funktion aufgerufen. Anschließend wird die Szene gezeichnet und die reflektierenden Objekte werden beim folgenden Aufruf ihrer draw Funktionen die aktualisierten Texturen verwenden. Siehe Listing 4.2.

```
1 void display() {
2     for (int i=0; i<reflectingObjects.size(); ++i) {
3         reflectingObjects[i]->update();
4     }
5     viewTransformation->draw();
6 }
```

Listing 4.2: display()

Neben der von ReflectingObject geerbten Attribute scene und object haben SMReflectingObject und analog PMReflectingObject noch folgende Attribute:

```
1 class SMReflectingObject : public ReflectingObject {
2     static bool isUpdating;
3     bool enabled;
4     static PBuffer* pbuffer;
5     static int genProgram;
6     static int applyProgram;
7     static int genAndApplyProgram;
8 ...
```

Die Klassenvariable isUpdating dient zum erkennen, ob gerade in eine Textur gezeichnet wird. Die Instanzvariable enabled wird benutzt, um zu verhindern, dass sich ein spiegelndes Objekt in seine eigene EM zeichnet. Die Variable pbuffer stellt die Funktionen zum off-screen rendering bereit, es genügt ein PBuffer für alle EMs. Die drei Attribute genProgram, applyProgram und genAndApplyProgram sind Handles für die drei nötigen Vertex Programme. Das Programm genProgram zeichnet die Szene verzerrt und dient zur Generierung der Texturen, applyProgram dient zur Anwendung der sphärischen, bzw. parabolischen EMs. Das Programm genAndApplyProgram muss dann aktiv sein, wenn ein spiegelndes Objekt gezeichnet wird, während die EM für ein anderes Objekt gezeichnet wird. Siehe hierzu auch Kapitel 4.2.3. Listings 4.3, 4.4, 4.5, 4.6, 4.7 und 4.8 zeigen für die Klassen SMReflectingObject und PMReflectingObject die Funktionen draw, update und die von PMReflectingObject::draw benötigten Hilfsfunktionen enableBlendFunction und disableBlendFunction.

```
1  void SMReflectingObject::update() {
2      enabled = false; // avoid drawing oneself
3      isUpdating = true;
4      pbuffer->enable();
5      glEnable(GL_VERTEX_PROGRAM_ARB);
6      glBindProgramARB(GL_VERTEX_PROGRAM_ARB, genProgram);
7
8      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
9      scene->draw();
10     pbuffer->saveToTexture(texture);
11
12     glDisable(GL_VERTEX_PROGRAM_ARB);
13     pbuffer->disable();
14     isUpdating = false;
```

```
15      enabled = true;
16  }
```

Listing 4.3: SMReflectingObject::update()

```
1   void PMReflectingObject::update() {
2       enabled = false; // avoid drawing oneself
3       isUpdating = true;
4       pbuffer->enable();
5       glEnable(GL_VERTEX_PROGRAM_ARB);
6       glBindProgramARB(GL_VERTEX_PROGRAM_ARB, genProgram);
7
8       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
9       // signal the vertex program to draw the front
10      glProgramEnvParameter4fARB(GL_VERTEX_PROGRAM_ARB, 0,
11          1.0, 0.0, 0.0, 0.0);
12      scene->draw();
13      pbuffer->saveToTexture(frontTexture);
14
15      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
16      // signal the vertex program to draw the back
17      glProgramEnvParameter4fARB(GL_VERTEX_PROGRAM_ARB, 0,
18          -1.0, 0.0, 0.0, 0.0);
19      scene->draw();
20      pbuffer->saveToTexture(backTexture);
21
22      glDisable(GL_VERTEX_PROGRAM_ARB);
23      pbuffer->disable();
24      isUpdating = false;
25      enabled = true;
26  }
```

Listing 4.4: PMReflectingObject::update()

```
1   void SMReflectingObject::draw() const {
2       if (!enabled) {
3           // we're generating our own EM, so don't draw oneself
4           return;
5       }
6       glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
7       glEnable(GL_TEXTURE_2D);
8       glBindTexture(GL_TEXTURE_2D, texture);
9
10      if (isUpdating) {
11          glBindProgramARB(GL_VERTEX_PROGRAM_ARB, genAndApplyProgram);
12          object->draw();
13          glBindProgramARB(GL_VERTEX_PROGRAM_ARB, genProgram);
14      } else {
15          glEnable(GL_VERTEX_PROGRAM_ARB);
16          glBindProgramARB(GL_VERTEX_PROGRAM_ARB, applyProgram);
17          object->draw();
18          glDisable(GL_VERTEX_PROGRAM_ARB);
19      }
20      glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
21      glDisable(GL_TEXTURE_2D);
22  }
```

Listing 4.5: SMReflectingObject::draw()

```
1  void PMReflectingObject::enableBlendFunction() const {
2      glActiveTextureARB(GL_TEXTURE0_ARB);
3      glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
4      glEnable(GL_TEXTURE_2D);
5      glBindTexture(GL_TEXTURE_2D, frontTexture);
6
7      glActiveTextureARB(GL_TEXTURE1_ARB);
8      glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
9      glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_INTERPOLATE);
10     glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_PREVIOUS);
11     glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
12     glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB, GL_TEXTURE1);
13     glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_RGB, GL_SRC_COLOR);
14     glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE2_RGB, GL_PRIMARY_COLOR);
15     glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND2_RGB, GL_SRC_ALPHA);
16     glEnable(GL_TEXTURE_2D);
17     glBindTexture(GL_TEXTURE_2D, backTexture);
18 }
```

Listing 4.6: PMReflectingObject::enableBlendFunction()

```
1  void PMReflectingObject::disableBlendFunction() const {
2      glActiveTextureARB(GL_TEXTURE1_ARB);
3      glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
4      glDisable(GL_TEXTURE_2D);
5      glActiveTextureARB(GL_TEXTURE0_ARB);
6      glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
7      glDisable(GL_TEXTURE_2D);
8  }
```

Listing 4.7: PMReflectingObject::disableBlendFunction()

```
1  void PMReflectingObject::draw() const {
2      if (!enabled) {
3          // we're generating our own EM, so don't draw oneself
4          return;
5      }
6      enableBlendFunction();
7
8      if (isUpdating) {
9          glBindProgramARB(GL_VERTEX_PROGRAM_ARB, genAndApplyProgram);
10         object->draw();
11         glBindProgramARB(GL_VERTEX_PROGRAM_ARB, genProgram);
12     } else {
13         glEnable(GL_VERTEX_PROGRAM_ARB);
14         glBindProgramARB(GL_VERTEX_PROGRAM_ARB, applyProgram);
15         object->draw();
16         glDisable(GL_VERTEX_PROGRAM_ARB);
17     }
18     disableBlendFunction();
19 }
```

Listing 4.8: PMReflectingObject::draw()

4.2.3 Vertex Programme

In diesem Kapitel werden die wichtigsten Teile der Vertex Programme zur Erzeugung und Anwendung der sphärischen und parabolischen Texturen beschrieben. Vollständige Listings der Programme finden sich in Anhang A. Die Programme sind ARB vertex programs und verwenden Version 1.0. Da große Teile der Programme für das sphärische und parabolische EM gleich sind, werden diese in diesem Kapitel gemeinsam diskutiert.

Allen Vertex Programmen gemein ist, dass zunächst die Position des Eckpunkts (vertex.position) in den Modelviewraum transformiert wird und in der Variablen vPos gespeichert wird. In vPosUnit wird der Richtungsvektor zum Eckpunkt gespeichert, also vPos normalisiert. Die Normale des Eckpunkts (vertex.normal) wird ebenfalls in den Modelviewraum transformiert und in vNormal gespeichert.

```
1  PARAM mv[4] = { state.matrix.modelview };
2  PARAM mvinvtrans[4] = { state.matrix.modelview.invtrans };
3
4  DP4 vPos.x, mv[0], vertex.position;
5  DP4 vPos.y, mv[1], vertex.position;
6  DP4 vPos.z, mv[2], vertex.position;
7  DP4 vPos.w, mv[3], vertex.position;
8
9  DP3 temp.x, vPos, vPos;
10 RSQ temp.x, temp.x;
11 MUL vPosUnit.xyz, vPos, temp.x;
12
13 DP3 vNormal.x, mvinvtrans[0], vertex.normal;
14 DP3 vNormal.y, mvinvtrans[1], vertex.normal;
15 DP3 vNormal.z, mvinvtrans[2], vertex.normal;
```

Listing 4.9: Berechnung von vPos, vPosUnit und vNormal

Programme zum Generieren der EMs

Für das Generieren von sphärischen und parabolischen Environment Maps durch Verzerren der Szene ist je ein Vertex Programm notwendig. Die Modelviewmatrix muss beim Aufruf der Programme so konfiguriert sein, dass sich die Kamera auf dem Bezugspunkt (Mittelpunkt des spiegelnden Objekts) befindet. Die Blickrichtung muss beim sphärischen EM in Richtung des Betrachters zeigen. Beim parabolischen EM ist die Blickrichtung wegen der Blickrichtungsunabhängigkeit beliebig, es muss jedoch beim Anwenden die gleiche wie beim Erzeugen der Texturen verwendet werden.

Beide Programme führen eine übliche Beleuchtungsberechnung durch und leiten die Texturkoordinaten unverändert weiter. Die Verzerrung äußert sich durch die Zuweisung der x und y Komponenten der endgültigen Position (result.position.xy). Die sphärische Verzerrung erfolgt wie in Gleichungen 3.4 und 3.5 beschrieben.

```
1  ADD temp, vPosUnit, {0, 0, 1, 0};
2  DP3 temp.x, temp, temp; # = rx^2+ry^2+(rz+1)^2
3  RSQ temp.x, temp.x; # = 1/m = 1/(rx^2+ry^2+(rz+1)^2)^0.5
4  MUL result.position.xy, vPosUnit, temp.x; # = rx|ry / m
```

Listing 4.10: Berechnung der sphärischen Verzerrung

Das Programm für die parabolischen Texturen bekommt über einen Parameter (sideflag.x) von der Anwendung mitgeteilt, ob die vordere oder die hintere Textur gezeichnet werden soll. Die parabolische Verzerrung erfolgt wie in Gleichungen 3.12 und 3.13 beschrieben.

```
1 # sideflag.x is 1.0 for the front and -1.0 for the back side
2 PARAM sideflag = program.env[0];
3 ...
4 MAD temp.z, vPosUnit.z, sideflag.x, 1.0; # = 1 +|- rz
5 RCP temp.z, temp.z; # = 1 / (1 +|- rz)
6 MUL result.position.xy, vPosUnit, temp.z; # = rx|ry / (1 +|- rz)
```

Listing 4.11: Berechnung der parabolischen Verzerrung

Die Berechnung des Tiefenwertes (result.position.z) wird in beiden Programmen nach Gleichung 3.9 durchgeführt. Wählt man für den Abstand der near clipping sphere 1 und für den Abstand der far clipping sphere 150, so errechnen sich die Koeffizienten $\frac{151}{149} \approx 1.0134$ und $-\frac{300}{149} \approx -2.0134$.

```
1 # put z between 1.0 (far plane) and -1.0 (near plane)
2 DP3 temp.x, vPos, vPos; # = x^2+y^2+z^2
3 RSQ temp.x, temp.x; # = 1/(x^2+y^2+z^2)^0.5
4 MAD result.position.z, temp.x, -2.0134, 1.0134;
```

Listing 4.12: Berechnung des Tiefenwerts

Die homogene Komponente der Endposition (result.position.w) ist normalerweise 1, wird aber gleich 0 gesetzt, um Eckpunkte, die nicht gezeichnet werden sollen, unendlich weit weg zu verschieben. Wie in Kapitel 3.3.1 beschrieben, können so beim sphärischen EM die Punkte, für die $r_z \approx -1$ gilt, gemieden werden.

```
1 SGE result.position.w, vPosUnit.z, -0.95;
```

Listing 4.13: Vermeidung unerwünschter Punkte beim sphärischen EM

Das gleiche Prinzip wird beim Programm für die parabolischen EMs verwendet, um die deutlich hinter dem Paraboloiden liegenden Kanten nicht zu zeichnen (wie in Kapitel 3.4.1 beschrieben).

```
1 MUL temp.x, vPosUnit.z, sideflag.x;
2 SGE result.position.w, temp.x, -0.5;
```

Listing 4.14: Vermeidung unerwünschter Punkte beim parabolisches EM

Anwenden der EMs ohne Vertex Programme

Sowohl beim sphärischen, als auch beim parabolischen EM lässt sich das Zeichnen der spiegelnden Objekte auch ohne Vertex Shader implementieren. Das automatische Generieren der sphärischen Texturkoordinaten muss wie folgt aktiviert werden.

```
1 glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
2 glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
3 glEnable(GL_TEXTURE_GEN_S);
4 glEnable(GL_TEXTURE_GEN_T);
```

Listing 4.15: Automatische Generierung der sphärischen Texturkoordinaten

Folgendes Listing aktiviert das Generieren der parabolischen Texturkoordinaten für die vordere Textur in Textureinheit 0. Das Aktivieren für die hintere Textur, welche in Textureinheit 1 liegt, ist analog und wurde in folgendem Listing weggelassen. Das Array matFront enthält Matrix 3.16 und matBack enthält Matrix 3.17 (OpenGL stellt Matrizen in transponierter Form dar).

```
GLfloat matFront[16] = { 1.0,  0.0,  0.0,  0.0,
                         0.0,  1.0,  0.0,  0.0,
                         1.0,  1.0,  0.0,  2.0,
                         1.0,  1.0,  0.0,  2.0 };

GLfloat matBack[16] = { 1.0,  0.0,  0.0,  0.0,
                        0.0,  1.0,  0.0,  0.0,
                       -1.0, -1.0,  0.0, -2.0,
                        1.0,  1.0,  0.0,  2.0 };

glActiveTextureARB(GL_TEXTURE0_ARB);
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glMatrixMode(GL_TEXTURE);
glLoadMatrixf(matFront);
```

Listing 4.16: Automatische Generierung der parabolischen Texturkoordinaten

Programme zum Anwenden der EMs

In der Anwendung werden Vertex Shader benutzt, um die Texturkoordinaten zu erzeugen. Beim parabolischen EM bringt das Vertex Programm den Vorteil, dass das Problem, den korrekten alpha-Wert zur Interpolation der beiden Texturen zu finden, einfach gelöst werden kann, indem man den alpha-Wert der Farbe entsprechend setzt.

Wie in den Programmen zur Generierung der Texturen werden zunächst die Variablen vPos, vPosUnit und vNormal berechnet. Nun wird der Reflexionsvektor rUnit wie in Gleichung 3.1 berechnet.

```
DP3 temp.x, vNormal, vPosUnit; # = -n*e (vPosUnit = -e)
MUL temp.x, temp.x, 2.0; # = -2(n*e)
MAD rUnit, vNormal, -temp.x, vPosUnit; # = 2(n*e)n - e
```

Listing 4.17: Berechnung des Reflexionsvektors

Beim Programm für die sphärische EM berechnen sich die Texturkoordinaten wie in Gleichungen 3.6 und 3.7 beschrieben.

```
ADD temp, rUnit, {0, 0, 1, 0};
DP3 temp.x, temp, temp; # = rx^2+ry^2+(rz+1)^2
RSQ temp.x, temp.x; # = 1/m = 1/(rx^2+ry^2+(rz+1)^2)^0.5
MUL temp.xy, rUnit, temp.x; # = rx|ry / m
MAD result.texcoord[0].xy, temp, 0.5, 0.5; # = rx|ry / 2m + 0.5
MOV result.texcoord[0].zw, 1.0;
```

Listing 4.18: Berechnung der sphärischen Texturkoordinaten

Die Texturkoordinaten für die vordere und hintere Textur beim parabolischen EM berechnen sich nach Gleichungen 3.14 und 3.15.

```
ADD temp.z, rUnit.z, 1.0; # = 1+rz
RCP temp.z, temp.z; # = 1 / (1+rz)
MUL temp.xy, rUnit, temp.z; # = rx|ry / (1+rz)
MAD result.texcoord[0].xy, temp, 0.5, 0.5; # = rx|ry / (2+2rz) + 0.5
```

```
5  MOV result.texcoord[0].zw, 1.0;
6
7  ADD temp.z, -rUnit.z, 1.0; # = 1-rz
8  RCP temp.z, temp.z; # = 1 / (1-rz)
9  MUL temp.xy, rUnit, temp.z; # = rx|ry / (1-rz)
10 MAD result.texcoord[1].xy, temp, 0.5, 0.5; # = rx|ry / (2-2rz) + 0.5
11 MOV result.texcoord[1].zw, 1.0;
```

Listing 4.19: Berechnung der parabolischen Texturkoordinaten

Um beim parabolischen EM den alpha-Wert in der Farbe zu speichern setzt man result.color.w gleich 1, wenn die vordere Textur für den aktuellen Eckpunkt gültig ist ($r_z \geq 0$).

```
1  SGE result.color.w, rUnit.z, 0.0;
```

Listing 4.20: Setzen des alpha-Werts der Farbe

Die endgültige Position der Eckpunkte berechnet sich in beiden Vertex Programmen durch die normale perspektivische Projektion. Beleuchtungsberechnung ist nicht notwendig, da von perfekt spiegelnden Objekten ausgegangen wird.

Programme zum Generieren und Anwenden der EMs

Können in einer Szene mehr als nur ein spiegelndes Objekt vorkommen, so ist bei beiden Verfahren ein drittes Vertex Programm notwendig, welches dann aktiv ist, wenn das eine spiegelnde Objekt gezeichnet wird, während die Textur für das andere erzeugt wird. Diese Programme berechnen die verzerrte Position und den Tiefenwert wie die Programme zum Erzeugen der Texturen. Die Texturkoordinaten (und der alpha-Wert der Farbe bei parabolischen EMs) werden wie in den Programmen zur Anwendung der Texturen berechnet. Beleuchtungsberechnung ist auch hier nicht nötig.

Sphärische und Parabolisch spiegelnde Objekte in derselben Szene

Möchte man gleichzeitig sphärische und parabolische Spiegelungen in einer Szene unterstützen, so sind noch zwei weitere Vertex Programme notwendig. Das eine erzeugt parabolische Texturkoordinaten und verzerrt die Eckpunkte sphärisch, das andere erzeugt sphärische Texturkoordinaten und verzerrt die Eckpunkte parabolisch.

Anhang A

Vollständige Vertex Programme

A.1 Programm zur Generierung der sphärischen EM

```
1  !!ARBvp1.0
2
3  ################################################################################
4  # smgen.arb
5  #
6  # This vertex program generates a spherical environment map.
7  ################################################################################
8
9
10 ################################################################################
11 # Declaration of parameters, aliases, temps
12 ################################################################################
13
14 PARAM mv[4] = { state.matrix.modelview };
15 PARAM mvinvtrans[4] = { state.matrix.modelview.invtrans };
16
17 TEMP vPos; # Position of vertex in modelview-space
18 TEMP vPosUnit; # Unified vPos
19 TEMP vNormal; # Normal in modelview-space
20 TEMP temp;
21
22 # Stuff needed for lighting computation
23 PARAM lPos = state.light[0].position;
24 PARAM lGlobalAmbient = state.lightmodel.ambient;
25 PARAM lAmbient = state.lightprod[0].ambient;
26 PARAM lDiffuse = state.lightprod[0].diffuse;
27 PARAM lSpecular = state.lightprod[0].specular;
28 PARAM lShininess = state.material.shininess;
29 PARAM lEmission = state.material.emission;
30 TEMP vertexToLight;
31 TEMP lCoeff;
32 TEMP lHalf;
33
34
35 ################################################################################
36 # Calculate the position (both absolute and normalized) and
37 # normal of the vertex. This is done in modelview space.
38 ################################################################################
39
40 DP4 vPos.x, mv[0], vertex.position;
41 DP4 vPos.y, mv[1], vertex.position;
42 DP4 vPos.z, mv[2], vertex.position;
43 DP4 vPos.w, mv[3], vertex.position;
44
```

```
45  # Calculate normalized position vector
46  DP3 temp.x, vPos, vPos;
47  RSQ temp.x, temp.x;
48  MUL vPosUnit.xyz, vPos, temp.x;
49
50  DP3 vNormal.x, mvinvtrans[0], vertex.normal;
51  DP3 vNormal.y, mvinvtrans[1], vertex.normal;
52  DP3 vNormal.z, mvinvtrans[2], vertex.normal;
53
54
55  ###############################################################################
56  # Apply the spherical distortion
57  ###############################################################################
58
59  ADD temp, vPosUnit, {0, 0, 1, 0};
60  DP3 temp.x, temp, temp; # = rx^2+ry^2+(rz+1)^2
61  RSQ temp.x, temp.x; # = 1/m = 1/(rx^2+ry^2+(rz+1)^2)^0.5
62  MUL result.position.xy, vPosUnit, temp.x; # = rx|ry / m
63
64
65  ###############################################################################
66  # Transform z value into normalized coordinates
67  ###############################################################################
68
69  # Put z between 1.0 (far plane) and -1.0 (near plane) (for clipping)
70  DP3 temp.x, vPos, vPos; # = x^2+y^2+z^2
71  RSQ temp.x, temp.x; # = 1/(x^2+y^2+z^2)^0.5
72  MAD result.position.z, temp.x, -2.0134, 1.0134;
73
74  # Throw everything that is close to the singularity far, far away
75  SGE result.position.w, vPosUnit.z, -0.95;
76
77
78  ###############################################################################
79  # Pass on the texture coordinates
80  ###############################################################################
81
82  MOV result.texcoord[0], vertex.texcoord[0];
83  MOV result.texcoord[1], vertex.texcoord[1];
84
85
86  ###############################################################################
87  # Lighting calculation
88  # not considering:
89  # - attenuation
90  ###############################################################################
91
92  # Calculate the normalized vertex to lightsource vector
93  ADD vertexToLight, lPos, -vPos;
94  DP3 vertexToLight.w, vertexToLight, vertexToLight;
95  RSQ vertexToLight.w, vertexToLight.w;
96  MUL vertexToLight.xyz, vertexToLight.w, vertexToLight;
97
98  # Calculate half-way vector
99  ADD lHalf, vertexToLight, -vPosUnit;
100
101 # Normalize half-way vector
102 DP3 lHalf.w, lHalf, lHalf;
103 RSQ lHalf.w, lHalf.w;
104 MUL lHalf.xyz, lHalf.w, lHalf;
105
106 # Prepare the coefficients for the lighting computation
107 DP3 lCoeff.x, vNormal, vertexToLight; # diffuse
108 DP3 lCoeff.y, vNormal, lHalf; # specular
109 MOV lCoeff.w, lShininess.x; # specular exponent
110
```

```
111 # Calculate diffuse & specular components
112 LIT lCoeff, lCoeff;
113
114 # Sum up ambient, diffuse, specular components and emission
115 MAD temp, lAmbient, lCoeff.x, lEmission;
116 MAD temp, lDiffuse, lCoeff.y, temp;
117 MAD temp, lSpecular, lCoeff.z, temp;
118 MAD result.color, lGlobalAmbient, state.material.ambient, temp;
119
120 # Fill color alpha with diffuse alpha
121 MOV result.color.w, lDiffuse.w;
122
123
124 END
```

Listing A.1: smgen

A.2 Programm zur Anwendung der sphärischen EM

```
1  !!ARBvp1.0
2
3  ##############################################################################
4  # smapply.arb
5  #
6  # This vertex shader applies a spherical environment map.
7  ##############################################################################
8
9
10 ##############################################################################
11 # Declaration of parameters, aliases, temps
12 ##############################################################################
13
14 PARAM mv[4] = { state.matrix.modelview };
15 PARAM mvp[4] = { state.matrix.mvp };
16 PARAM mvinvtrans[4] = { state.matrix.modelview.invtrans };
17
18 TEMP vPos; # Position of vertex in modelview-space
19 TEMP vPosUnit; # Unified vPos
20 TEMP vNormal; # Normal in modelview-space
21 TEMP rUnit; # Unified reflection vector in eye-space
22 TEMP temp;
23
24
25 ##############################################################################
26 # Calculate the position (both absolute and normalized) and
27 # normal of the vertex. This is done in modelview-space.
28 ##############################################################################
29
30 DP4 vPos.x, mv[0], vertex.position;
31 DP4 vPos.y, mv[1], vertex.position;
32 DP4 vPos.z, mv[2], vertex.position;
33 DP4 vPos.w, mv[3], vertex.position;
34
35 # Calculate normalized position vector
36 DP3 temp.x, vPos, vPos;
37 RSQ temp.x, temp.x;
38 MUL vPosUnit.xyz, vPos, temp.x;
39
40 DP3 vNormal.x, mvinvtrans[0], vertex.normal;
41 DP3 vNormal.y, mvinvtrans[1], vertex.normal;
42 DP3 vNormal.z, mvinvtrans[2], vertex.normal;
43
44 # Normalize normal
```

```
45 DP3 temp.x, vNormal, vNormal;
46 RSQ temp.x, temp.x;
47 MUL vNormal.xyz, vNormal, temp.x;
48
49
50 ###############################################################################
51 # Set the position using by concatenation of the
52 # Modelview and Projection matrices (usual perspective transformation)
53 ###############################################################################
54
55 DP4 result.position.x, mvp[0], vertex.position;
56 DP4 result.position.y, mvp[1], vertex.position;
57 DP4 result.position.z, mvp[2], vertex.position;
58 DP4 result.position.w, mvp[3], vertex.position;
59
60
61 ###############################################################################
62 # Compute the reflection vector in eye-space
63 ###############################################################################
64
65 DP3 temp.x, vNormal, vPosUnit; # = -n*e (vPosUnit = -e)
66 MUL temp.x, temp.x, 2.0; # = -2(n*e)
67 MAD rUnit, vNormal, -temp.x, vPosUnit; # = 2(n*e)n - e
68
69
70 ###############################################################################
71 # Calculate the texture coordinates
72 ###############################################################################
73
74 ADD temp, rUnit, {0, 0, 1, 0};
75 DP3 temp.x, temp, temp; # = rx^2+ry^2+(rz+1)^2
76 RSQ temp.x, temp.x; # = 1/m = 1/(rx^2+ry^2+(rz+1)^2)^0.5
77 MUL temp.xy, rUnit, temp.x; # = rx|ry / m
78 MAD result.texcoord[0].xy, temp, 0.5, 0.5; # = rx|ry / 2m + 0.5
79 MOV result.texcoord[0].zw, 1.0;
80
81
82 END
```

Listing A.2: smapply

A.3 Programm zur Generierung und Anwendung der sphärischen EM

```
1  !!ARBvp1.0
2
3  ###############################################################################
4  # smgenandapply.arb
5  #
6  # This vertex program applies a spherical environment map, while applying
7  # spherical distortion. It is enabled for reflecting objects while drawing
8  # the texture for another reflecting object.
9  ###############################################################################
10
11
12 ###############################################################################
13 # Declaration of parameters, aliases, temps
14 ###############################################################################
15
16 PARAM mv[4] = { state.matrix.modelview };
17 PARAM mvinvtrans[4] = { state.matrix.modelview.invtrans };
18
19 TEMP vPos; # Position of vertex in modelview-space
20 TEMP vPosUnit; # Unified vPos
```

```
21 TEMP vNormal; # Normal in modelview-space
22 TEMP rUnit; # Unified reflection vector in eye-space
23 TEMP temp;
24
25
26 ##############################################################################
27 # Calculate the position (both absolute and normalized) and
28 # normal of the vertex. This is done in modelview space.
29 ##############################################################################
30
31 DP4 vPos.x, mv[0], vertex.position;
32 DP4 vPos.y, mv[1], vertex.position;
33 DP4 vPos.z, mv[2], vertex.position;
34 DP4 vPos.w, mv[3], vertex.position;
35
36 # Calculate normalized position vector
37 DP3 temp.x, vPos, vPos;
38 RSQ temp.x, temp.x;
39 MUL vPosUnit.xyz, vPos, temp.x;
40
41 DP3 vNormal.x, mvinvtrans[0], vertex.normal;
42 DP3 vNormal.y, mvinvtrans[1], vertex.normal;
43 DP3 vNormal.z, mvinvtrans[2], vertex.normal;
44
45 # Normalize normal
46 DP3 vNormal.w, vNormal, vNormal;
47 RSQ vNormal.w, vNormal.w;
48 MUL vNormal.xyz, vNormal.w, vNormal;
49
50
51 ##############################################################################
52 # Apply the spherical distortion
53 ##############################################################################
54
55 ADD temp, vPosUnit, {0, 0, 1, 0};
56 DP3 temp.x, temp, temp; # = rx^2+ry^2+(rz+1)^2
57 RSQ temp.x, temp.x; # = 1/m = 1/(rx^2+ry^2+(rz+1)^2)^0.5
58 MUL result.position.xy, vPosUnit, temp.x; # = rx|ry / m
59
60
61 ##############################################################################
62 # Transform z value into normalized coordinates
63 ##############################################################################
64
65 # Put z between 1.0 (far plane) and -1.0 (near plane) (for clipping)
66 DP3 temp.x, vPos, vPos; # = x^2+y^2+z^2
67 RSQ temp.x, temp.x; # = 1/(x^2+y^2+z^2)^0.5
68 MAD result.position.z, temp.x, -2.0134, 1.0134;
69
70 # Throw everything that is close to the singularity far, far away
71 SGE result.position.w, vPosUnit.z, -0.95;
72
73
74 ##############################################################################
75 # Compute the reflection vector in eye-space
76 ##############################################################################
77
78 DP3 temp.x, vNormal, vPosUnit; # = -n*e (vPosUnit = -e)
79 MUL temp.x, temp.x, 2.0; # = -2(n*e)
80 MAD rUnit, vNormal, -temp.x, vPosUnit; # = 2(n*e)n - e
81
82
83 ##############################################################################
84 # Calculate the texture coordinates
85 ##############################################################################
86
```

```
87 ADD temp, rUnit, {0, 0, 1, 0};
88 DP3 temp.x, temp, temp; # = rx^2+ry^2+(rz+1)^2
89 RSQ temp.x, temp.x; # = 1/m = 1/(rx^2+ry^2+(rz+1)^2)^0.5
90 MUL temp.xy, rUnit, temp.x; # = rx|ry / m
91 MAD result.texcoord[0].xy, temp, 0.5, 0.5; # = rx|ry / 2m + 0.5
92 MOV result.texcoord[0].zw, 1.0;
93
94
95 END
```

Listing A.3: smgenandapply

A.4 Programm zur Generierung der parabolischen EM

```
1  !!ARBvp1.0
2
3  ##############################################################################
4  # pmgen.arb
5  #
6  # This vertex program generates parabolic environment maps.
7  # It should be called twice - once for the front and once for the back side.
8  ##############################################################################
9
10
11 ##############################################################################
12 # Declaration of parameters from the application
13 ##############################################################################
14
15 # sideflag.x is 1.0 for the front and -1.0 for the back side
16 PARAM sideflag =  program.env[0];
17
18
19 ##############################################################################
20 # Declaration of parameters, aliases, temps
21 ##############################################################################
22
23 PARAM mv[4] = { state.matrix.modelview };
24 PARAM mvinvtrans[4] = { state.matrix.modelview.invtrans };
25
26 TEMP vPos; # Position of vertex in modelview-space
27 TEMP vPosUnit; # Unified vPos
28 TEMP vNormal; # Normal in modelview-space
29 TEMP temp;
30
31 # Stuff needed for lighting computation
32 PARAM lPos = state.light[0].position;
33 PARAM lGlobalAmbient = state.lightmodel.ambient;
34 PARAM lAmbient = state.lightprod[0].ambient;
35 PARAM lDiffuse = state.lightprod[0].diffuse;
36 PARAM lSpecular = state.lightprod[0].specular;
37 PARAM lShininess = state.material.shininess;
38 PARAM lEmission = state.material.emission;
39 TEMP vertexToLight;
40 TEMP lCoeff;
41 TEMP lHalf;
42
43
44 ##############################################################################
45 # Calculate the position (both absolute and normalized) and
46 # normal of the vertex. This is done in modelview-space.
47 ##############################################################################
48
49 DP4 vPos.x, mv[0], vertex.position;
```

```
50  DP4 vPos.y, mv[1], vertex.position;
51  DP4 vPos.z, mv[2], vertex.position;
52  DP4 vPos.w, mv[3], vertex.position;
53
54  # Calculate normalized position vector
55  DP3 temp.x, vPos, vPos;
56  RSQ temp.x, temp.x;
57  MUL vPosUnit.xyz, vPos, temp.x;
58
59  DP3 vNormal.x, mvinvtrans[0], vertex.normal;
60  DP3 vNormal.y, mvinvtrans[1], vertex.normal;
61  DP3 vNormal.z, mvinvtrans[2], vertex.normal;
62
63
64  ###############################################################################
65  # Apply the parabolic distortion
66  ###############################################################################
67
68  MAD temp.z, vPosUnit.z, sideflag.x, 1.0; # = 1 +|- rz
69  RCP temp.z, temp.z; # = 1 / (1 +|- rz)
70  MUL result.position.xy, vPosUnit, temp.z; # = rx|ry / (1 +/- rz)
71
72  SGE result.color.w, vPosUnit.z, 0.0;
73
74
75  ###############################################################################
76  # Transform z value into normalized coordinates
77  ###############################################################################
78
79  # Put z between 1.0 (far plane) and -1.0 (near plane) (for clipping)
80  DP3 temp.x, vPos, vPos; # = x^2+y^2+z^2
81  RSQ temp.x, temp.x; # = 1/(x^2+y^2+z^2)^0.5
82  MAD result.position.z, temp.x, -2.0134, 1.0134;
83
84  # Throw everything that is way behind the paraboloid far, far away
85  MUL temp.x, vPosUnit.z, sideflag.x;
86  SGE result.position.w, temp.x, -0.5;
87
88
89  ###############################################################################
90  # Pass on the texture coordinates
91  ###############################################################################
92
93  MOV result.texcoord[0], vertex.texcoord[0];
94  MOV result.texcoord[1], vertex.texcoord[1];
95
96
97  ###############################################################################
98  # Lighting calculation
99  # not considering:
100 # - attenuation
101 ###############################################################################
102
103 # Calculate the normalized vertex to lightsource vector
104 ADD vertexToLight, lPos, -vPos;
105 DP3 vertexToLight.w, vertexToLight, vertexToLight;
106 RSQ vertexToLight.w, vertexToLight.w;
107 MUL vertexToLight.xyz, vertexToLight.w, vertexToLight;
108
109 # Calculate half-way vector
110 ADD lHalf, vertexToLight, -vPosUnit;
111
112 # Normalize half-way vector
113 DP3 lHalf.w, lHalf, lHalf;
114 RSQ lHalf.w, lHalf.w;
115 MUL lHalf.xyz, lHalf.w, lHalf;
```

```
116
117 # Prepare the coefficients for the lighting computation
118 DP3 lCoeff.x, vNormal, vertexToLight; # diffuse
119 DP3 lCoeff.y, vNormal, lHalf; # specular
120 MOV lCoeff.w, lShininess.x; # specular exponent
121
122 # Calculate diffuse & specular components
123 LIT lCoeff, lCoeff;
124
125 # Sum up ambient, diffuse, specular components and emission
126 MAD temp, lAmbient, lCoeff.x, lEmission;
127 MAD temp, lDiffuse, lCoeff.y, temp;
128 MAD temp, lSpecular, lCoeff.z, temp;
129 MAD result.color, lGlobalAmbient, state.material.ambient, temp;
130
131 # Fill color alpha with diffuse alpha
132 MOV result.color.w, lDiffuse.w;
133
134
135 END
```

Listing A.4: pmgen

A.5 Programm zur Anwendung der parabolischen EM

```
1  !!ARBvp1.0
2
3  ##############################################################################
4  # pmapply.arb
5  #
6  # This vertex shader applies parabolic environment maps.
7  # It generates the texture coordinates for the front a back texture.
8  ##############################################################################
9
10
11 ##############################################################################
12 # Declaration of parameters, aliases, temps
13 ##############################################################################
14
15 PARAM mv[4] = { state.matrix.modelview };
16 PARAM mvp[4] = { state.matrix.mvp };
17 PARAM mvinvtrans[4] = { state.matrix.modelview.invtrans };
18
19 TEMP vPos; # Position of vertex in modelview-space
20 TEMP vPosUnit; # Unified vPos
21 TEMP vNormal; # Normal in modelview-space
22 TEMP rUnit; # Unified reflection vector in object-space
23 TEMP temp;
24
25
26 ##############################################################################
27 # Calculate the position (both absolute and normalized) and
28 # normal of the vertex. This is done in modelview-space.
29 ##############################################################################
30
31 DP4 vPos.x, mv[0], vertex.position;
32 DP4 vPos.y, mv[1], vertex.position;
33 DP4 vPos.z, mv[2], vertex.position;
34 DP4 vPos.w, mv[3], vertex.position;
35
36 # Calculate normalized position vector
37 DP3 temp.x, vPos, vPos;
38 RSQ temp.x, temp.x;
```

```
39  MUL vPosUnit.xyz, vPos, temp.x;
40
41  DP3 vNormal.x, mvinvtrans[0], vertex.normal;
42  DP3 vNormal.y, mvinvtrans[1], vertex.normal;
43  DP3 vNormal.z, mvinvtrans[2], vertex.normal;
44
45  # Normalize normal
46  DP3 temp.x, vNormal, vNormal;
47  RSQ temp.x, temp.x;
48  MUL vNormal.xyz, vNormal, temp.x;
49
50
51  ###############################################################################
52  # Set the position using by concatenation of the
53  # Modelview and Projection matrices (usual perspective transformation)
54  ###############################################################################
55
56  DP4 result.position.x, mvp[0], vertex.position;
57  DP4 result.position.y, mvp[1], vertex.position;
58  DP4 result.position.z, mvp[2], vertex.position;
59  DP4 result.position.w, mvp[3], vertex.position;
60
61
62  ###############################################################################
63  # Compute the reflection vector in eye-space
64  ###############################################################################
65
66  DP3 temp.x, vNormal, vPosUnit; # = -n*e (vPosUnit = -e)
67  MUL temp.x, temp.x, 2.0; # = -2(n*e)
68  MAD rUnit, vNormal, -temp.x, vPosUnit; # = 2(n*e)n - e
69
70
71  ###############################################################################
72  # Calculate the front side's texture coordinates
73  ###############################################################################
74
75  ADD temp.z, rUnit.z, 1.0; # = 1+rz
76  RCP temp.z, temp.z; # = 1 / (1+rz)
77  MUL temp.xy, rUnit, temp.z; # = rx|ry / (1+rz)
78  MAD result.texcoord[0].xy, temp, 0.5, 0.5; # = rx|ry / (2+2rz) + 0.5
79  MOV result.texcoord[0].zw, 1.0;
80
81
82  ###############################################################################
83  # Calculate the back side's texture coordinates
84  ###############################################################################
85
86  ADD temp.z, -rUnit.z, 1.0; # = 1-rz
87  RCP temp.z, temp.z; # = 1 / (1-rz)
88  MUL temp.xy, rUnit, temp.z; # = rx|ry / (1-rz)
89  MAD result.texcoord[1].xy, temp, 0.5, 0.5; # = rx|ry / (2-2rz) + 0.5
90  MOV result.texcoord[1].zw, 1.0;
91
92
93  ###############################################################################
94  # Set the primary color's alpha channel to 1.0 if the front is active, to
95  # 0.0 if the back side is active.
96  ###############################################################################
97
98  SGE result.color.w, rUnit.z, 0.0;
99
100
101 END
```

Listing A.5: pmapply

A.6 Programm zur Generierung und Anwendung der parabolischen EM

```
1  !!ARBvp1.0
2
3  ################################################################################
4  # pmgenandapply.arb
5  #
6  # This vertex shader applies parabolic environment maps while distorting.
7  # It is enabled for reflecting objects while drawing the texture for another
8  # reflecting object.
9  ################################################################################
10
11
12 ################################################################################
13 # Declaration of parameters from the application
14 ################################################################################
15
16 # sideflag.x is 1.0 for the front and -1.0 for the back side
17 PARAM sideflag =  program.env[0];
18
19
20 ################################################################################
21 # Declaration of parameters, aliases, temps
22 ################################################################################
23
24 PARAM mv[4] = { state.matrix.modelview };
25 PARAM mvinvtrans[4] = { state.matrix.modelview.invtrans };
26
27 TEMP vPos; # Position of vertex in modelview-space
28 TEMP vPosUnit; # Unified vPos
29 TEMP vNormal; # Normal in modelview-space
30 TEMP rUnit; # Unified reflection vector in object-space
31 TEMP temp;
32
33
34 ################################################################################
35 # Calculate the position (both absolute and normalized) and
36 # normal of the vertex. This is done in modelview-space.
37 ################################################################################
38
39 DP4 vPos.x, mv[0], vertex.position;
40 DP4 vPos.y, mv[1], vertex.position;
41 DP4 vPos.z, mv[2], vertex.position;
42 DP4 vPos.w, mv[3], vertex.position;
43
44 # Calculate normalized position vector
45 DP3 temp.x, vPos, vPos;
46 RSQ temp.x, temp.x;
47 MUL vPosUnit.xyz, vPos, temp.x;
48
49 DP3 vNormal.x, mvinvtrans[0], vertex.normal;
50 DP3 vNormal.y, mvinvtrans[1], vertex.normal;
51 DP3 vNormal.z, mvinvtrans[2], vertex.normal;
52
53 # Normalize normal
54 DP3 temp.x, vNormal, vNormal;
55 RSQ temp.x, temp.x;
56 MUL vNormal.xyz, vNormal, temp.x;
57
58
59 ################################################################################
60 # Apply the parabolic distortion
61 ################################################################################
62
```

```
63  MAD temp.z, vPosUnit.z, sideflag.x, 1.0; # = 1 +|- rz
64  RCP temp.z, temp.z; # = 1 / (1 +|- rz)
65  MUL result.position.xy, vPosUnit, temp.z; # = rx|ry / (1 +/- rz)
66
67
68  ###############################################################################
69  # Transform z value into normalized coordinates
70  ###############################################################################
71
72  # Put z between 1.0 (far plane) and -1.0 (near plane) (for clipping)
73  DP3 temp.x, vPos, vPos; # = x^2+y^2+z^2
74  RSQ temp.x, temp.x; # = 1/(x^2+y^2+z^2)^0.5
75  MAD result.position.z, temp.x, -2.0134, 1.0134;
76
77  # Throw everything that is way behind the paraboloid far, far away
78  MUL temp.x, vPosUnit.z, sideflag.x;
79  SGE result.position.w, temp.x, -0.5;
80
81
82  ###############################################################################
83  # Compute the reflection vector in eye-space
84  ###############################################################################
85
86  DP3 temp.x, vNormal, vPosUnit; # = -n*e (vPosUnit = -e)
87  MUL temp.x, temp.x, 2.0; # = -2(n*e)
88  MAD rUnit, vNormal, -temp.x, vPosUnit; # = 2(n*e)n - e
89
90
91  ###############################################################################
92  # Calculate the front side's texture coordinates
93  ###############################################################################
94
95  ADD temp.z, rUnit.z, 1.0; # = 1+rz
96  RCP temp.z, temp.z; # = 1 / (1+rz)
97  MUL temp.xy, rUnit, temp.z; # = rx|ry / (1+rz)
98  MAD result.texcoord[0].xy, temp, 0.5, 0.5; # = rx|ry / (2+2rz) + 0.5
99  MOV result.texcoord[0].zw, 1.0;
100
101
102 ###############################################################################
103 # Calculate the back side's texture coordinates
104 ###############################################################################
105
106 ADD temp.z, -rUnit.z, 1.0; # = 1-rz
107 RCP temp.z, temp.z; # = 1 / (1-rz)
108 MUL temp.xy, rUnit, temp.z; # = rx|ry / (1-rz)
109 MAD result.texcoord[1].xy, temp, 0.5, 0.5; # = rx|ry / (2-2rz) + 0.5
110 MOV result.texcoord[1].zw, 1.0;
111
112
113 ###############################################################################
114 # Set the primary color's alpha channel to 1.0 if the front is active, to
115 # 0.0 if the back side is active.
116 ###############################################################################
117
118 SGE result.color.w, rUnit.z, 0.0;
119
120
121 END
```

Listing A.6: pmgenandapply

Literaturverzeichnis

[1] Thomas Akenine-Möller, *Real-Time Rendering*, A K Peters Ltd., 2002

[2] ARB (Architectural Review Board), *ARB_vertex_program*, `http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt`, 2004

[3] ARB (Architectural Review Board), *OpenGL Specification, version 2.0*, 2004

[4] Cass Everitt, *OpenGL ARB Vertex Program*, Game Developers Conference, 2003

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison Wesley, 1995

[6] Wolfgang Heidrich, Hans-Peter Seidel, *View-independent Environment Maps*, ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1998

[7] Wolfgang Heidrich, *High-quality Shading and Lighting for Hardware-accelerated Rendering*, Dissertation, Technische Fakultät der Universität Erlangen-Nürnberg, `http://www.cs.ubc.ca/~heidrich/Papers/phd.pdf`, 1999

[8] Mark J. Kilgard, *Real-time Environment Reflections with OpenGL*, Nvidia Corporation, `http://developer.nvidia.com`, 1999

[9] Mark J. Kilgard, *Computations for Hardware Lighting and Shading*, Game Developers Conference, `http://developer.nvidia.com`, 2000

[10] Gene S. Miller, C. Robert Hoffmann, *Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments*, SIGGRAPH, 1984

[11] Shree K. Nayar, *Catadioptric Omnidirectional Camera*, IEEE Conference on Computer Vision and Pattern Recognition, 1997

[12] Kim Pallister, *Rendering to Texture Surfaces using DirectX7*, Gamasutra, `http://www.gamasutra.com/features/19991112/pallister_01.htm`, 1999

[13] Chriss Wynn, *OpenGL Vertex Programming on Future-Generation GPUs*, Game Developers Conference, `http://developer.nvidia.com/attach/6361`, 2001

Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, daß die Arbeit veröffentlicht wird und daß in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 15. April 2005

(Christoph Dietze)