Diminished Reality

Diplomarbeit

Vorgelegt von
Daniel Schilling

Institut für Computervisualistik
Arbeitsgruppe Computergraphik

Betreuer: Dipl.-Inform. Thorsten Grosch
Prüfer: Prof. Dr.-Ing. Stefan Müller

April 2005

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass die vorliegende Arbeit selbständig verfasst wurde und keine anderen, als die angegebenen Quellen und Hilfsmittel verwendet wurden.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

....

Ort, Datum Unterschrift

Danksagung

Eine solch komplexe Arbeit bedarf viel Schweiß und Zeit, und vor allen Dingen helfenden Händen. So soll an dieser Stelle jenen gedankt werden, ohne die diese Arbeit nicht möglich gewesen wäre.
Danken möchte ich vor allen Dingen Dipl.-Inform. Thorsten Grosch, der mich auf dieses spannende Thema erst aufmerksam gemacht hat und als universitärer Betreuer im ständigen Dialog Fragen erläutert und Ideen kritisch diskutiert hat. Eine gute Ausbildung ist ohne die lehrenden Dozenten und Hilfskräfte nicht möglich. Insbesondere Prof. Dr.-Ing. Stefan Müller und Prof. Dr.-Ing. Dietrich Paulus haben mein Interesse an der Computergraphik und Bildverarbeitung geweckt. Großer Dank gilt meiner Lebensgefährtin Sandra, die stets Verständnis für die mangelnde und häufig stressreiche Zeit aufgebracht hat und die ständige Motivation, sowie der kleinen Chantale, die zwischendurch für Entspannung gesorgt hat. Durch die Unterstützung meiner Eltern speziell im finanziellen Bereich war mir dieses Studium erst möglich. Weiterhin hat mir Carina Will, Christian Altenhofen und meine Schwester Manuela sehr geholfen, dieser Ausarbeitung den letzten Schliff durch Korrekturlesen und Anregungen zu geben. Viele kleinere Fragen konnten durch Kommilitonen schnell beantwortet werden.

Daniel Schilling

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung

Die Bedeutung von Augmented Reality wird nicht nur im Bereich von professionellen Speziallösungen zunehmen, sondern auch vermehrt in Massenprodukten. So ist es derzeit der Stand der Technik, dass z.B. Küchenstudios die Zimmer der Kunden auf Grund des Grundrisses vom Computer „modellieren“ lassen. In diese Räume werden anschließend Möbelgruppen integriert und als fotorealistischer Ausdruck dem Kunden zur Untermauerung seiner Vorstellungen zur Verfügung gestellt. Dieser Ansatz wird zunehmend auch in Möbelhäusern verwendet, jedoch nur mit mäßiger Akzeptanz. Im Falle eines Küchenneukaufs wird in der Regel das Zimmer „Küche“ komplett umgestaltet und mit neuen Möbeln ausgestattet. So bietet sich in jedem Fall ein einheitliches und zusammengehöriges Bild. Anders sieht dies im Möbelhandel aus: hier ist der Regelfall so, dass nur bestimmte Möbelgruppen durch neue ersetzt werden sollen und sich die neu anzuschaffenden Möbel aber konsistent in das vorhandene Bild integrieren sollen. Mit den bisherigen Methoden ist dies aber nicht möglich.
Ein möglicher Ausweg aus diesem Dilemma wäre, wenn der Kunde einige vorhandene Fotos seines neu zu gestaltenden Raumes (z.B. des Wohnzimmers) zum Möbelkauf mitbringen würde und mit Hilfe moderner Techniken aus der Augmented Reality ein 3D-Modell rekonstruiert würde. So könnten anschließend interaktiv Veränderungen an diesem virtuellen Raum vorgenommen werden. Einige Veränderungs-Parameter sind das Einfügen von virtuellen Objekten, Einfügen von zusätzlichen Lichtquellen, Verändern von Materialeigenschaften, sowie das Entfernen von realen Objekten. Basierend auf der Arbeit von Thorsten Grosch [Gro05], in der die ersten drei Fälle abgedeckt sind, wird in dieser Diplomarbeit ein Verfahren vorgestellt, mit dem der letzte Punkt, das Entfernen von realen Objekten, realisiert wird. Das Ziel ist ein experimentelles System, in dem alle bisherigen Manipulationsmöglichkeiten vereint sind.
Der Begriff „Diminished Reality“ bedeutet, dass Teile der realen Umgebung entfernt werden und durch Sinnvolles ergänzt wird. Es ist ein Anwendungsgebiet der Augmented Reality. Im Falle dieser Diplomarbeit werden reale Objekte aus der Szene entfernt und durch entsprechende Texturen vervollständigt und photometrisch korrekt beleuchtet. Eine andere Anwendung von Diminished Reality beschreibt z.B. [MF01] in Form von Informationseinblendungen in Head-Mounted-Displays (HMD) und im Speziellen beim „Wearable Computing“.
Dieses Dokument gliedert sich wie folgt: Im anschließenden Kapitel 1 wird die grundlegende Idee skizziert, welche Schritte nötig sind, um reale Objekte aus einer Szene zu entfernen. Der Abschnitt 2 beschäftigt sich mit der Textursynthese und den speziellen Anforderungen an diese Anwendung. Fortgefahren wird mit der konkreten Integration der synthetisierten Texturen in die Szene (Kapitel 3) und einem Versuch das entfernte Objekt an anderer Stelle wieder

einzufügen (Kapitel 5). Abschließend wird in der Sektion 6 die gewonnenen Ergebnisse betrachtet und ein Ausblick aufgezeigt. Im Anhang A befinden sich Quellcodes von Implementierungsdetails.

1. Überblick

Ausgehend von der Idee, einen Panorama-Betrachter (z.B. Apples QuicktimeVR) um AR-Funktionalität zu erweitern, hat die Arbeitsgruppe Computergraphik der Universität Koblenz-Landau, Institut für Computervisualistik eine Software entwickelt, in der verschiedene Ansätze einer solchen Realisierung gezeigt werden.

1.1. Vorhandenes Framework

Basierend auf einer HDR-Lightprobe wurde die Geometrie im 3D-Raum modelliert und anschließend mit einer Radiosity-Simulation verschiedene Veränderungen gezeigt. So können zusätzliche Lichtquellen sowie zusätzliche virtuelle Objekte eingefügt oder die Eigenschaften von Materialien verändert werden. Ferner wurde die starre Projektion auf eine Kugeloberfläche aufgehoben, so dass die Blickrichtung nicht nur aus dem Kugelmittelpunkt heraus, sondern von jeder beliebigen Position im Raum erfolgen kann. In der Lightprobe werden dabei verdeckte Bereiche sinnvoll ergänzt. Im gesamten Projekt wurde größter Wert auf schnelle Verarbeitung im Echtzeitbereich und photometrische Konsistenz gelegt. So schlägt zum Beispiel ein Tisch aus der realen Szene seinen Schatten auf das virtuell eingefügte Objekt.

1.2. HDR-Lightprobe

Fotos wurden bisher unter starren Beleuchtungsbedingungen aufgenommen und können nur einen geringen Kontrast wiedergeben. Auch die bisherige Bildverarbeitung und Computergraphik benutzte die starre Bandbreite von nur maximal 256 Farbabstufungen pro Farbkanal. In der realen Welt jedoch treten besonders bei Sonneneinstrahlung wesentlich höhere Helligkeitsabstufungen auf. Solche Bilder haben einen hohen Dynamikbereich (Dynamic Range). Der Dynamikbereich ist das Verhältnis zwischen dunklen und hellen Regionen. In der Computergraphik treten häufig Probleme und instabilitäten auf Grund des zu geringen Dynamikbereiches auf. Auch aus diesem Grund entstand die Forderung nach entsprechenden „High-Dynamic-Range-Bildern“ (HDR-Bilder). Im Allgemeinen werden die Werte der HDR-Farbkanäle in 16-bit- oder 32-bit-Fließkommazahlen abgelegt. Besonders im Zusammenspiel mit Radiosity, wo Strahlungsaustausch stattfindet und auch die Strahlung an jedem Punkt ermittelt werden kann, ist eine hohe Dynamik von wesentlicher Bedeutung.
Als eine Lightprobe wird eine kugelförmige Aufnahme der Umgebung bezeichnet. Sie wird generiert, in dem eine Aufnahme von einer Spiegelkugel erstellt wird. Nach Paul Debevec

[Deb98] ist eine Lightprobe immer eine HDR-Aufnahme, da sie als Messung von Licht der realen Welt definiert ist. Die in dieser Arbeit verwendete HDR-Lightprobe wurde in einem Büro der Universität mit einer relativ einfachen Digitalkamera aufgenommen. Auf Grund dessen ist die Qualität dieser Aufnahme nicht mit einer professionellen Lightprobe vergleichbar. Die Lightprobe ist als Lat-Long-Textur gespeichert, d.h. die Texturkoordinaten dienen der einfachen Umrechnung in Theta-Phi-Koordinaten (siehe Kapitel 3.3).

1.3. Ideenskizze

Aufbauend auf das Softwaresystem der Arbeitsgruppe soll nun ein reales Objekt komplett aus der Szene entfernt und an anderer Stelle wieder eingefügt werden. Dabei müssen die Schatten des zu entfernenden Objektes aus der Szene entfernt werden und die Stelle, an der sich das Objekt befunden hat, sinvoll ergänzt werden. Die konkrete Implementierung erfolgt experimentell, d.h. einige Parameter müssen manuell ausgewählt werden bzw. das System ist nicht vollständig für die gesamte Szene, sondern nur für bestimmte Teilbereiche validiert (siehe Kapitel 4). Die vorgestellten Algorithmen decken jedoch den gesamten Bereich ab. Das Einfügen des realen Objektes an anderer Stelle wird zum Schluß kurz erläutert. Ausgehend von dieser Prämisse sind die in Abbildung 1.1 dargestellten groben Teilschritte zum Erreichen des Ziels skizziert. Im jeweiligen Kapitel wird detaillierter darauf eingegangen.

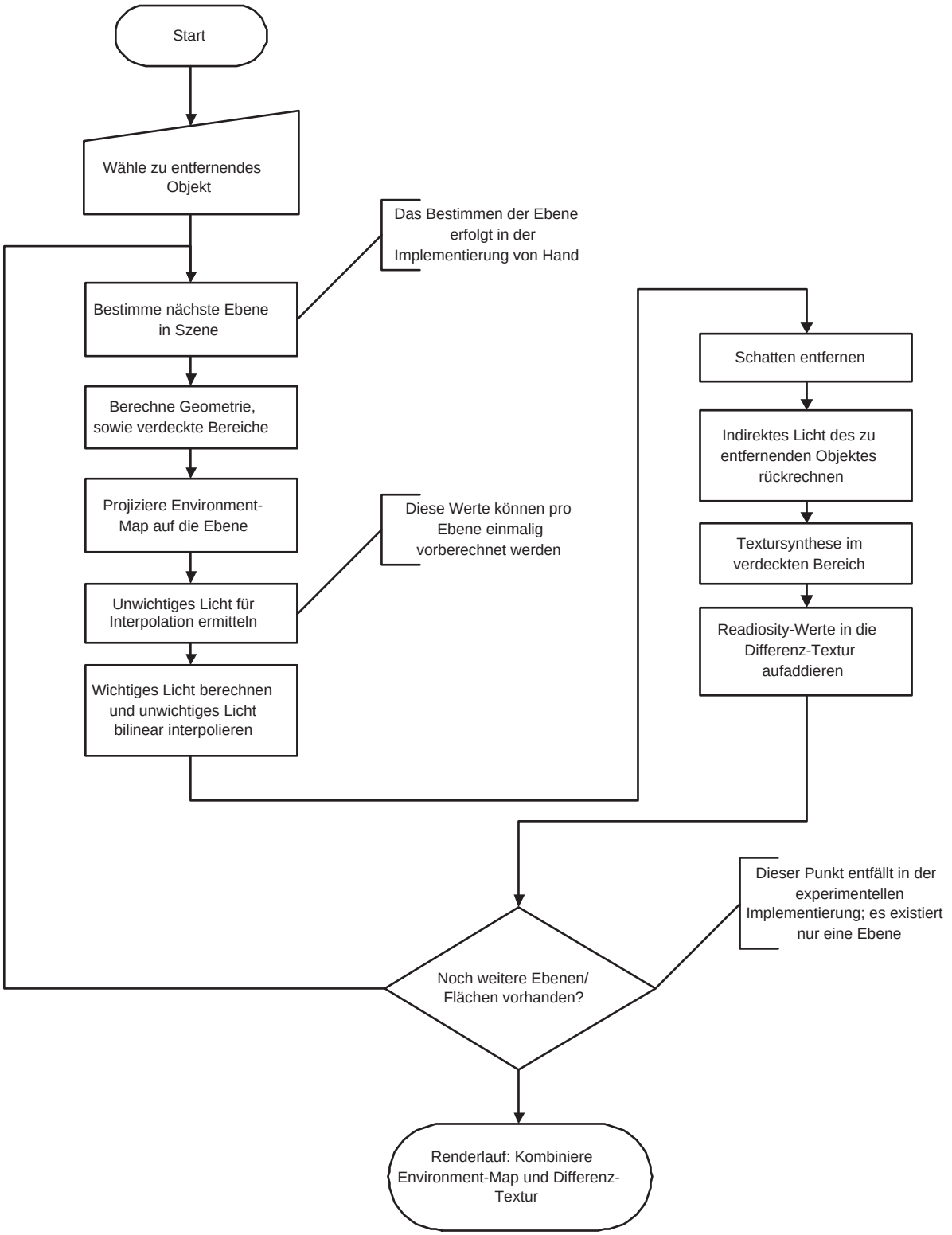

Abbildung 1.1.: Ablaufplan

2. Textursynthese

Die Textursynthese ist eines der zentralen Bestandteile des vorgestellten Verfahrens. Nach dem Entfernen des realen Objektes muss diese Stelle wieder mit sinnvollem Hintergrund ergänzt werden. Soll z.B. eine Kiste, die auf dem Boden steht, entfernt werden, so muss der Teil des Bodens synthetisiert werden, auf dem die Kiste stand. Textursynthese bedeutet, dass aus einer Mustertextur eine neue Textur mit beliebiger vorgegebener Größe erstellt werden soll, die „so ähnlich“ aussieht wie die Mustertextur (siehe Abbildung 2.1). In der Literatur finden sich dazu einige Verfahren mit unterschiedlichen Schwerpunkten.

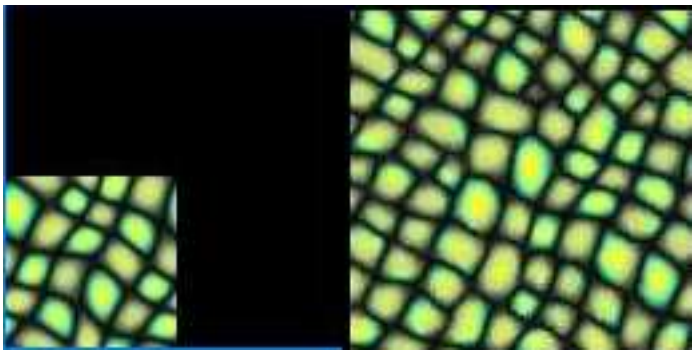

Abbildung 2.1.: links: Muster, rechts: Synthesebild

Generell gliedern sich die Textursynthesealgorithmen in zwei Arten: zum einen statistische Verfahren und zum anderen prozedurale Verfahren. Erstere verwenden statistische Methoden, um Ähnlichkeitskriterien zu berechnen (z.B. Nachbarschaftsvergleiche), während die zweite Gruppe durch fest implementierte Algorithmen **eine** Art von Texturen perfekt synthetisieren kann. Einige Vertreter der ersten Gattung sind [HB95], [OABB85] und [EF01], bei denen der Schwerpunkt auf möglichst guter Musterrekonstruktion liegt. Der Nachteil dabei ist, dass die Verarbeitungsgeschwindigkeit jenseits der Minutengrenze liegt und damit nicht in Frage kommen. Es wird vielmehr ein Algorithmus benötigt, der schnell arbeitet (möglichst in Echtzeit) und trotzdem ausreichend gute Ergebnisse liefert.
Im Nachfolgenden werden Anforderungen an den Textursynthesealgorithmus gestellt, so dass er optimal in die Problemstellung passt. Es wurden existierende Algorithmen auf diese Kriterien hin geprüft. Das Ergebnis ist ein Algorithmus der grundlegend auf [WL02] basiert und viele seiner Eigenschaften ausnutzt.

2.1. Anforderungen

Die Problemstellung für einen Textursynthesealgorithmus lässt sich wie folgt beschreiben: Die zu ergänzende Stelle in der realen Szene soll sich nahtlos einpassen, d.h. es sollen keine Übergangsstellen sichtbar sein. Ideal wäre es, wenn der Benutzer das zu entfernende Objekt mit der Maus auswählt und daraufhin umgehend in Echtzeit auch der Verarbeitungsschritt stattfinden würde. Bereits in Kapitel 1 wurden die groben nötigen Schritte für das gesamte Verfahren aufgezeigt. Der Syntheseschritt ist nur **ein** Teilschritt, wohl aber der aufwändigste. So muss besonders hier das Augenmerk auf Effizienz gerichtet sein, bei ausreichend guten Ergebnissen.
Im Allgemeinen leiten sich deshalb folgende Anforderungskriterien ab:

1. Schnelligkeit
2. gute Syntheseresultate
3. Berücksichtigung der lokalen Übergangs-Schnittstellen
4. Berechnung nur der benötigten Stellen in der Szene (Pro-Pixel-Synthese, ähnlich eines Shaders)
5. Möglichkeit zur Parallel-Verarbeitung
6. gute Skalierbarkeit

Da bei der Nachbarschaftssuche eine Antiproportion zwischen Qualität und Geschwindigkeit besteht [1], müssen Vorkehrungen getroffen werden, dass der Benutzer bei Bedarf das beste Resultat erwarten kann, andererseits aber in seiner Interaktion nicht behindert werden darf. Deshalb ist das Verfahren so angelegt, dass zunächst niedrigere Auflösungen generiert werden. Solange keine weitere Benutzerinteraktion stattfindet, können höhere Auflösungen berechnet werden. Die höchstmögliche Qualität wird also durch mehrere Render-Passes erreicht.t

2.2. Grundlegendes Verfahren

Das Ziel des Algorithmus ist es, eine neue Textur T_o beliebiger Größe zu generieren, die auf lokaler Ebene ähnlich aussieht, wie in der Eingabetextur T_i. Dazu wird für jedes gesuchte Pixel p_o in der Ausgabetextur eine Nachbarschaft $N(p_o)$ gebildet und mit der Nachbarschaft $N(p_i)$ jedes Pixels p_i aus der Eingabetextur ein Ähnlichkeitsmaß berechnet. Der Farbwert des gesuchten Pixels ist nun derjenige Wert aus der Eingabetextur, dessen Nachbarschafts-Ähnlichkeit am höchsten ist. Wird nun die komplette Ausgabetextur durchlaufen, so erhält

[1] Größere Nachbarschaften bedeuten bessere Qualität, da ein größerer lokaler Bereich berücksichtigt wird. Bei größeren Nachbarschaften sind aber auch deutlich mehr Vergleichsoperationen nötig, was zu niedrigeren Geschwindigkeiten führt

man eine Textur, die die lokalen Strukturen im Bereich der Nachbarschaften der Eingabetextur aufweist.
Zum Gelingen dieses Verfahrens müssen jedoch noch zwei Bedingungen erfüllt werden: Zum einen muss die Ausgabetextur mit Rauschfarben aus der Eingabetextur initialisiert werden. Auf der anderen Seite darf die Nachbarschaft nur solche Pixel enthalten, die schon bearbeitet wurden.
Im konkreten Fall wird von einer Verarbeitungsrichtung von links oben nach rechts unten und einer quadratischen Nachbarschaft der Größe n ausgegangen. Die Nachbarschaft enthält demnach alle Pixel links und oberhalb des betrachteten Pixels. Zur Erhaltung lokaler Texturstrukturen müsste n so groß gewählt werden, dass mindestens einmal die größte Struktur in der Nachbarschaft enthalten ist. Dies würde aber selbst bei einfachen Mustern meist eine Nachbarschaft von $n \geq 9$ bedeuten. Im Hinblick auf die Geschwindigkeit unter Berücksichtigung guter lokaler Strukturen führt dies zu einer Gauss'schen Pyramide ähnlich [HB95] und [OABB85] (siehe Abb. 2.2). Eine solche Pyramide $G_T(L)$ wird für beide Texturen T_i und T_o generiert. Sei L die Ebene in der Gauss-Pyramide. Mit abnehmendem L wird eine Textur mit niedriger werdender Auflösung zurückgegeben (Verringerung der Seitenlänge auf die Hälfte je Level).

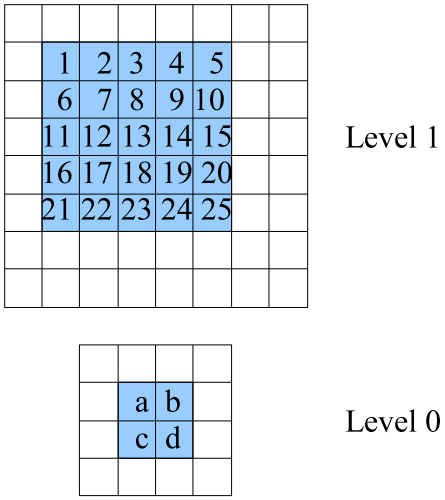

Abbildung 2.2.: Gauss'sche Pyramide

Die Nachbarschaft enthält nun nicht nur Pixel aus einer betrachteten Ebene, sondern auch aus darunterliegenden. Dabei nimmt die Nachbarschaft pro Ebene um die halbe Kantenlänge ab. Die Nachbarschaft $N(p_l)$ eines Pixels p_l enthält also Pixel aus der betrachteten Ebene l und $m < l$. Auf Grund der geringer werdenden Auflösung von einer Ebene zur anderen, kann so mit einer recht kleinen Nachbarschaft eine hohe lokale Struktur erfasst werden [2].

[2] Zum Vergleich: eine Nachbarschaft auf 2 Leveln und einer Größe von $n = 5$ enthält insgesamt $n^2 + (n/2)^2 = 25 + 4 = 29$ Elemente. Um ein ähnliches Ergebnis mit der ebenenlosen Methode zu erzielen, muss die Nachbarschaft nach Versuchen mindestens $n = 7$ sein und damit insgesamt 49 Elemente enthalten.

Nicht zu vergessen ist die Randbehandlung. Der Einfachheithalber wurden die Texturen periodisch fortgesetzt. Eine sinnvollere Behandlung wäre für die Eingangstextur die Spiegelung und für die Ausgabetextur keine Randbehandlung (die Randpixel werden nicht berücksichtigt). Die unterschiedliche Randbehandlung führt jedoch wieder zu einem komplizierteren Ähnlichkeitsmaß, was sich in der Zeit niederschlägt. Deshalb wurde hier dieselbe Randbehandlung für beide Texturen gewählt, um die Anzahl der Elemente in einer Nachbarschaft vorberechnen und konstant halten zu können.
Ein Ähnlichkeitsmaß ist der Kehrwert des Euklidischen Abstandes im RGB-Farbraum. Durch die bereits vorhandene interne Repräsentation in diesem Farbraum und vor allen Dingen dem Fortlaufen der Werte aus OpenGL-Arrays kann jedes dieser Arrays Element für Element durchlaufen werden und das Quadrat der Differenzen gebildet werden. Ein Hin- und Herrechnen entfällt im Gegensatz zu einem Ähnlichkeitsmaß z.B. im HSV-Farbraum. So ist auch an dieser Stelle dem Aspekt der Geschwindigkeit genüge getan. Ein Beispiel mit allen vier Pyramiden-Leveln ist in Abbildung 2.3 dargestellt.

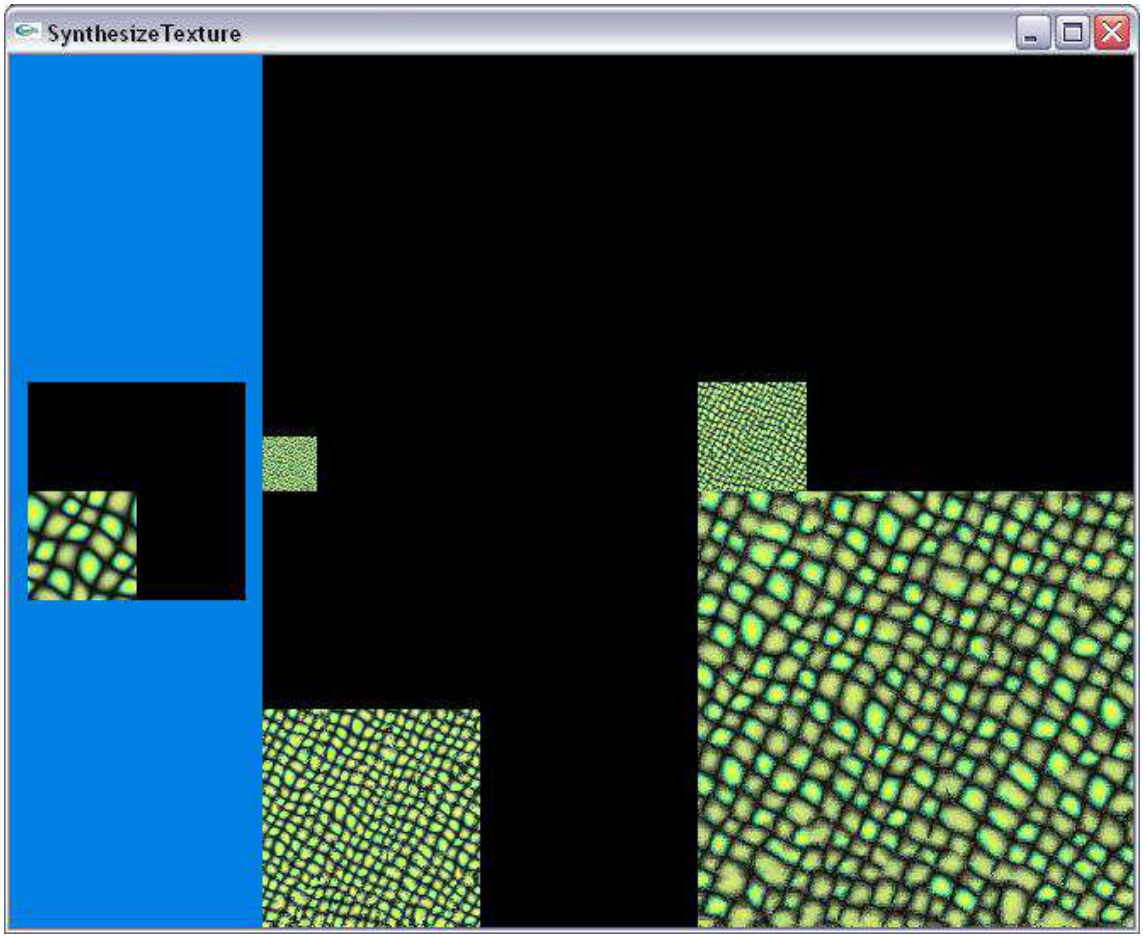

Abbildung 2.3.: Textursynthese: Vier Pyramiden-Level

Somit lässt sich der bisherige Algorithmus wie folgt formalisieren:

```
Image synthesizeTexture(Image Ti, int width, int height, int
   levels)
{
   Image To = new Image(width, height);
   GaussianPyramid Gi = new GaussianPyramid(Ti, levels);
   GaussianPyramid Go = new GaussianPyramid(To, levels);
   Pixel p;
   for (int l = levels-1; l >= 0; l--) {
      Image outLevel = Go->at(level);
      for (int yo = 0; yo < outLevel->getHeight(); yo++) {
         for (int xo = 0; xo < outLevel->getWidth(); xo++) {
            p = findBestMatch(Gi, Go, level, xo, yo);
            curLevel->set(xo, yo, p);
         }
      }
   }
   return Go->at(levels-1);
}

Pixel findBestMatch(GaussianPyramid Gi, GaussianPyramid Go,
   int level, int xo, int yo)
{
   Neighborhood No = new Neighborhood(Go, level, xo, yo);
   Neighborhood Ni;
   Neighborhood Nbest;
   Pixel p;
   Image inLevel = Gi->at(level);
   for (int yi = 0; yi < inLevel->getHeight(); yi++) {
      for (int xi = 0; xi < inLevel->getWidth(); xi++) {
         Ni = new Neighborhood(Gi, level, xin, yin);
         if (matchNeighborhood(Ni, No) > matchNeighborhood(
            Nbest, No)) {
            Nbest = Ni;
            p = inLevel->at(xin, yin);
         }
      }
   }
   return p;
}
```

2.3. Unabhängigkeit von der Verarbeitungsrichtung

Wird nun einmal der Abhängigkeitsgraph für ein Pixel betrachtet, so ergibt sich, dass jedes Pixel zyklisch von sich selbst abhängt. Betrachtet wird zur Veranschaulichung folgendes simple Beispiel, was in Abbildung 2.4 dargestellt ist: Die vorhandene Textur hat eine Größe von 8*8 Pixeln. Es wird eine einfache Nachbarschaft auf einer Ebene von $n = 3$ angenommen. Farbig hinterlegt sind die schon bearbeiteten Pixel und damit die Abhängigkeiten. Wird beispielhaft das Pixel 4 untersucht, so ist das Pixel 3 in seiner Nachbarschaft vorhanden. Pixel 4 hängt von Pixel 3 ab. Pixel 2 wiederum gehört zur Nachbarschaft von 3 und so fort, bis Pixel 5 wieder von 4 abhängig ist. Abgebildet sind beispielhaft 3 Fälle: Links dargestellt ist die Nachbarschaft zu Beginn der Betrachtung des Pixel 4. Das mittlere Bild zeigt die Nachbarschaft des Pixels 1, die durch die Randbehandlung auf der rechten Seite fortgesetzt wird. Rechts wird verdeutlicht, dass Pixel 4 zur Nachbarschaft von Pixel 5 gehört. Abbildung 2.5 zeigt den Abhängigkeitsgraphen, beschränkt auf das jeweils links daneben liegende Pixel. Daraus ist die zyklische Abhängigkeit ersichtlich.

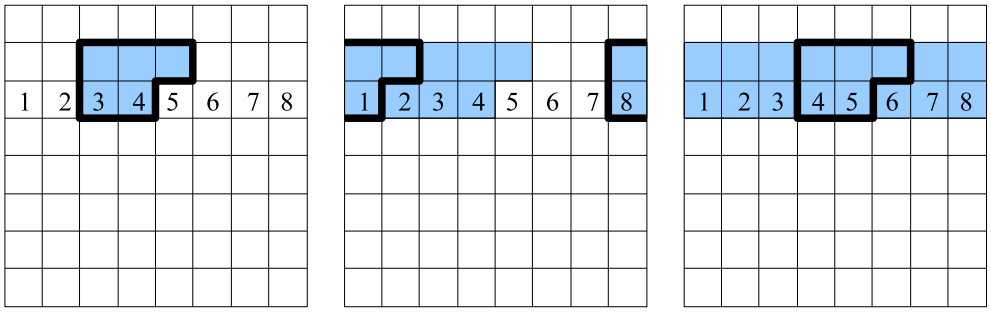

Abbildung 2.4.: Beispiel der zyklischen Abhängigkeit

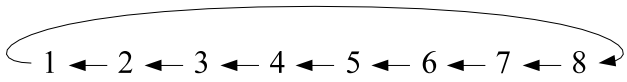

Abbildung 2.5.: Abhängigkeitsgraph

Durch diese zyklische Abhängigkeit wird verhindert, dass ein einzelnes Pixel auf Anforderung synthetisiert werden kann. Es müssen quasi immer alle Pixel synthetisiert werden. Da aber die Pro-Pixel-Synthese eines der Anforderungen ist, muss der Algorithmus so modifiziert werden, dass ein Pixel nicht mehr von sich selbst abhängt. Dazu wird für jedes angeforderte Pixel eine Generation G eingeführt. In der Nachbarschaft dürfen demnach nur noch Pixel auf derselben Ebene aus früheren Generationen enthalten sein, sowie solche aus niedrigeren Ebenen und allen Generationen.

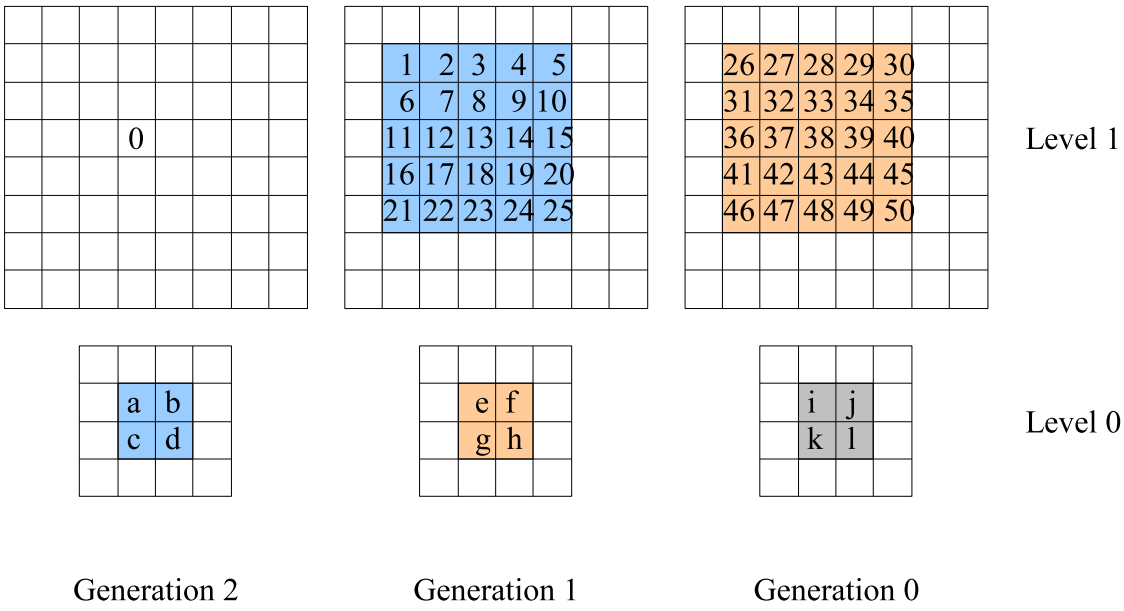

Abbildung 2.6.: Gauss'sche Pyramide erweitert um Generationen

Sei $N(p, L, G)$ die Nachbarschaft des Pixels p auf dem Level L und der Generation G. Dann enthält die Nachbarschaft nur diejenigen Pixel p_{G_i,L_i}, die das folgende Kriterium erfüllen:

$$(L_i < L)\ oder\ (L_i = L\ und\ G_i < G) \tag{2.1}$$

Die gesuchte Ausgabetextur ist demnach die der höchsten Generation und des höchsten Levels. Die unterste Generation des untersten Levels wird wiederum analog zum vorherigen Abschnitt mit einem Rauschen aus der Eingabetextur initialisiert. Dadurch, dass die Pixel nicht mehr zyklisch voneinander abhängen, ist auch der Anforderung, ein Pixel bei Bedarf zu synthetisieren, genüge getan. Viel mehr noch können nun Pixel auch auf jeder Ebene und damit in jeder Auflösung generiert werden, womit auch der Punkt der Skalierbarkeit durch mehrere Render-Passes erfüllt ist. Zunächst werden benötigte Pixel auf unteren Leveln (und damit niedrigen Auflösungen) berechnet und dem Benutzer angezeigt (durch OpenGL-Texture-Mapping). Solange keine Benutzerinteraktion erfolgt können höhere Auflösungen Stufe für Stufe synthetisiert werden (in OpenGL z.B. in der Idle-Funktion). Auch die Möglichkeit der Parallelverarbeitung ist durch die Unabhängigkeit der Pixel gegeben. Es wäre möglich, in mehreren Threads verschiedene Teile der Ausgabetextur parallel rechnen zu lassen. Dadurch könnte die HyperThreading-Struktur modernen Intel Pentium4-Prozessoren oder die Verfügbarkeit von günstigen Dualprozessorsystemen, wie dem AMD Opteron 2xx, ausgenutzt werden.

2.4. Datenstruktur für die Nachbarschaftssuche

Die Nachbarschaftssuche erfolgt so, dass für jedes Pixel der Ausgabetextur die Nachbarschaft gebildet wird und mit der Nachbarschaft eines jeden Pixels der Eingabetextur verglichen wird. Im naiven Ansatz ergibt sich daraus eine Doppelschleife, in der jedesmal die Positionen der Nachbarschaftspixel neu berechnet werden. Kommen nun noch pro Level mehrere Generationen hinzu, so werden selbst bei kleineren Texturen die Nachbarschaftspositionen schnell tausendfach neu kalkuliert.

```
main {
   /* Initialisiierungen */
   for (alle ausgabepixel po in To) {
      Npo = berechne_Nachbarschaft(po);
      for (alle eingabepixel pi in Ti) {
         Npi = berechne_Nachbarschaft(pi);
         vergleiche(Npi, Npo);
     }
   }
   /* weitere Verarbeitung */
}

Nachbarschaft berechne_Nachbarschaft(pi) {
   Nachbarschaft Ni;
   for (alle Pixel pn in Nachbarschaft von pi) {
      Ni->add(pn);
   }
   return Ni;
}
```

Mit jedem Schleifendurchlauf des Pixels für die Ausgabetextur werden **immer alle** Nachbarschaften der Eingabetextur berechnet. In der Praxis spiegelt sich das darin wieder, dass die Textursynthese hauptsächlich mit dem Berechnen der Nachbarschaftspixel beschäftigt ist und dadurch Laufzeiten von einigen Minuten der Standard sind.
Dies führt zu der Forderung nach einer effizienten Datenstruktur zumindest für die Eingabetextur. Hier bietet sich eine Baumstruktur mit der Nachbarschaft pro Pixel an. Jeder Pixel der Nachbarschaft ist wiederum ein Pointer in die originale Ausgabetextur. Zusätzlich wird auf oberster Baumebene der Level berücksichtigt. Diese Baumstruktur wird einmalig vorberechnet und wie eine Lookup-Table benutzt.

```
main {
   /* Initialisiierungen */
   berechneAlleNachbarschaften(Ti);

   for (alle ausgabepixel po in To) {
```

```
      Npo = berechne_Nachbarschaft(po);
      for (alle eingabepixel pi in Ti) {
         Npi = gibNachbarschaft(pi);
         vergleiche(Npi, Npo);
      }
   }
   /* weitere Verarbeitung */
}

Nachbarschaft gibNachbarschaft(pi) {
   return mNi[pi];
}
```

Der Geschwindigkeitsgewinn durch diese Modifikation ist enorm. Bei sonst gleichen Parametern wie Größe der Eingabetextur, Größe der Synthesetextur, Level und Generationen ist bei dem naiven Verfahren die Laufzeit 272 Sekunden und bei der verbesserten Methode nur noch 4 Sekunden. Dieser Wert ist zwar noch immer von der Echtzeitverarbeitung entfernt, jedoch ist dieser Wert akzeptabel, da er nur einmalig pro zu entfernendem Objekt berechnet werden muss. Ferner beinhaltet diese Laufzeit die komplette Synthese auf allen Leveln. Im Abschnitt 2.6 wird noch beschrieben, wie die Ebenenstruktur ausgenutzt werden kann, um die Latenz für den Benutzer gering zu halten.

2.5. Berücksichtigung lokaler Strukturen aus der Ausgangsszene

Bis zu diesem Punkt ist noch keinerlei Information aus der Umgebungsstruktur der Darstellungsszene eingeflossen. Die Textursynthese beschränkt sich als einzigen Parameter auf das Eingabe-Texturmuster. Das Ergebnis der Synthese ist eine Textur, die sich nicht nahtlos in die Szene einpassen lässt. Es sind deutlich sichtbare Schnittstellen zu erkennen, die den visuellen Gesamteindruck trüben. Es ist deshalb ein Mechanismus zu kreieren, der als weiteren Parameter die lokalen Strukturen im Bereich des entfernten Objektes während der Synthese berücksichtigt.
Untersucht wird zunächst noch einmal der Ansatz der Textursynthese aus dem ersten Teil des Abschnittes 2.2. Dort wird die Ausgabetextur mit einem Farbrauschen aus der Eingabetextur initialisiert. Dies stellt sicher, dass in der Nachbarschaft der Ausgabe immer Farbwerte enthalten sind, die auch in der Nachbarschaft der Mustertextur enthalten sind. Wird nun die Nachbarschaft eines Ausgabepixels zu Beginn der Synthese betrachtet, so sind in ihr nur zusammenhangslose Einzelpixel enthalten. Mit dem weiteren Verlauf der Synthese wird das Rauschbild mit jedem Synthesepixel verfeinert. Diese Verfeinerung geschieht wiederum durch die Nachbarschaftssuche.
Initialisiert man nun die Ausgabetextur mit schon vorhandenen Strukturen aus dem Randbe-

reich des entfernten Objektes, so werden diese Strukturen auch bei der Nachbarschaftssuche berücksichtigt und damit die lokale Struktur der Schnittstelle. Adaptiert auf das Gesamtverfahren am Ende des Abschnittes 2.3 ergibt sich, dass Pixel aus der Szene in die höchste Generation des obersten Levels kopiert werden müssen. Der Algorithmus startet somit mit sinvollen Farbwerten im Bereich der Schnittstelle.

2.6. Verfeinern der Textursynthese

Zur Vermeidung von Latenzzeiten bietet sich an, dass Zwischenergebnisse umgehend nach erfolgreicher Berechnung am Bildschirm erscheinen sollen. Der Benutzer soll nicht auf das eine perfekte Bild warten müssen, sondern unterdessen weiterarbeiten können. Pausen in der Benutzerinteraktion werden ausgenutzt (z.B. die OpenGL-Idle-Funktion), um die Berechnung fortzuführen. Die richtungsunabhängige Verarbeitungsrichtung gewährt eine Pro-Pixel-Synthese. Dies kann ausgenutzt werden, um z.B. vorerst nur jedes vierte Pixel zu berechnen und die fehlenden Pixel zu interpolieren. Auch die Pyramidenstruktur bietet eine Art Detail-Stufen („Level-of-Details“). So ist es möglich, zunächst eine *niedrigere* Ebene und damit eine niedrigere Texturauflösung schnell berechnen und anzeigen zu lassen. Die Kombination aus beiden Optionen lassen genügend Aktionsraum zur Skalierung zu.

3. Entfernen von Objekten

Nachdem bereits der Ablauf zur Objektentfernung im Kapitel 1 erläutert und im vorhergehenden Kapitel die Textursynthese vorgestellt wurde, widmet sich dieses Kapitel mit dem eigentlichen Kern dieser Arbeit. Hierbei werden die notwendigen Schritte tiefgreifend vorgestellt. Da das gesamte Softwaresystem auf einer Radiosity-Simulation basiert, werden Begrifflichkeiten und Funktionsweisen aus diesem Bereich stillschweigend vorausgesetzt und decken sich weitestgehend mit [Mül03b], [SP94] und [CW93]. Dennoch werden im folgenden Abschnitt einige Termini näher definiert.

3.1. Die Radiosity-Simulation

Lokale Beleuchtungsmodelle basieren auf der Annahme, dass eine feste Anzahl von Lichtquellen auf eine Szene einwirken. Ein Patch in einer solchen Szene erhält seine Farbe nur aus der Eigenemission und den definierten Lichtquellen. Durch das Fehlen von indirekter Beleuchtung durch andere Flächen oder über Spiegel hinweg, werden solche Szenen als nicht realistisch wahrgenommen, der „natürliche Eindruck“ fehlt. Ferner können keine Halbschatten dargestellt werden.
Ein globales Beleuchtungsmodell erfüllt alle oben genannten Eigenschaften. Aus der globalen Beleuchtungsformel („Rendering-Equation“) wird durch die Annahme einiger Spezialfälle die Radiosity-Gleichung abgeleitet. Erst numerische Verfahren zur Lösung des Radiosity-Gleichungssystems führen zu einer (für heutige Computer) verhältnismäßig schnellen angenäherten Berechenbarkeit.

3.1.1. Spektrale Größen

Photorealistische Computergraphik bedeutet, dass Strahlungsaustausch (z.B. von Licht) simuliert wird. Dazu sind einige (spektrale) Größen von Bedeutung. Diese werden in diesem Abschnitt kurz vorgestellt.
Der *Raumwinkel* ω ist die 3-dimensionale Entsprechung zum Bogenmaß. Es ist definiert durch das Verhältnis der bedeckten Fläche auf einer Kugel A_k zum Quadrat des Kugelradius r:

$$\omega = \frac{A_k}{r^2} \tag{3.1}$$

Häufig werden Polarkoordinaten zur Beschreibung eines Punktes auf der Kugeloberfläche verwendet (z.B. für Texturkoordinaten der Lat-Long-Textur in 3.3). So lässt sich ein Flächenstück

auf der Kugeloberfläche auch durch die Differenz-Beträge $d\theta$ und $d\varphi$ darstellen, so dass der Differenz-Raumwinkel $d\omega$ durch Infinitesimalrechnung wie folgt gegeben ist (siehe Abbildung 3.1):

$$d\omega = \frac{dA}{r^2} = \sin\theta \; d\theta \; d\varphi \tag{3.2}$$

Der *Strahlungsfluss* Φ ist die Strahlungsmenge pro Zeit. Eine Lichtquelle gibt einen sol-

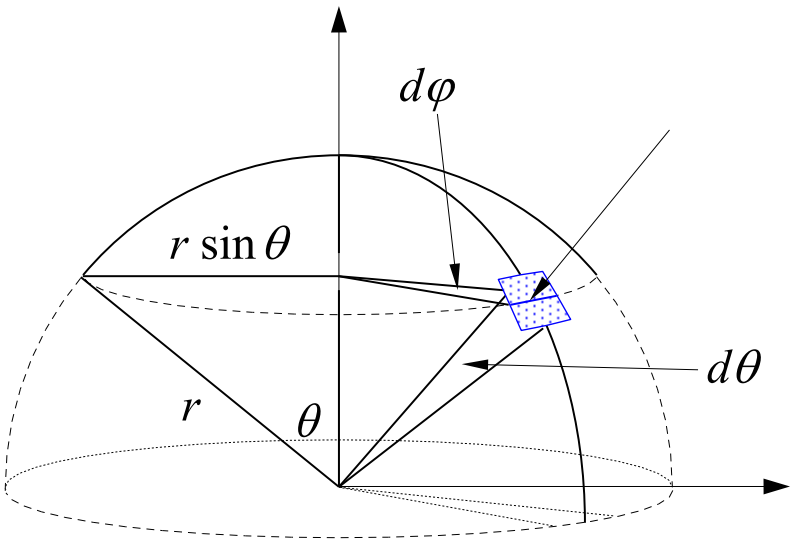

Abbildung 3.1.: Raumwinkel

chen Strahlungsfluss in den Raum ab und ist ein Maß für dessen Leistung. Der infinitesimale Strahlungsfluss ist gegeben als

$$d\Phi = dA \int_{\Omega} L \; \cos\theta \; d\omega \tag{3.3}$$

Die wohl wichtigste Größe ist die *Strahldichte (Radiance)* L. Sie wird vom menschlichen Auge als Helligkeit wahrgenommen und ist definiert als die Menge an Energie, die zu einem spezifischen Punkt, in einer spezifischen Richtung pro Zeiteinheit und Fläche abgestrahlt wird.

$$L = \frac{d^2\Phi}{dA \; \cos\theta \; d\omega} \tag{3.4}$$

In seinem Kern ist das Radiosity-Verfahren nichts anderes, als das Berechnen von Strahlungsaustausch (siehe Abbildung 3.2). Deshalb sind die historisch betrachteten Parameter die *Bestrahlungsstärke (Irradiance)* E und die *spezifische Lichtausstrahlung (Radiosity)* B. Die Bestrahlungsstärke ist der **auftreffende** Strahlungsfluss $d\Phi_{empfangen}$ pro Fläche dA_e und ist

somit eine Empfängergröße. Die spezifische Lichtausstrahlung hingegen ist eine Sendergröße und ist definiert durch den **abgegebenen** Strahlungsfluss $d\Phi_{abgegeben}$ pro Flächenelement dA_S. Bestrahlungsstärke (Irradiance):

$$E = \frac{d\Phi_{empfangen}}{dA_e} \tag{3.5}$$

Spezifische Lichtausstrahlung (Radiosity):

$$B = \frac{d\Phi_{abgegeben}}{dA_s} \tag{3.6}$$

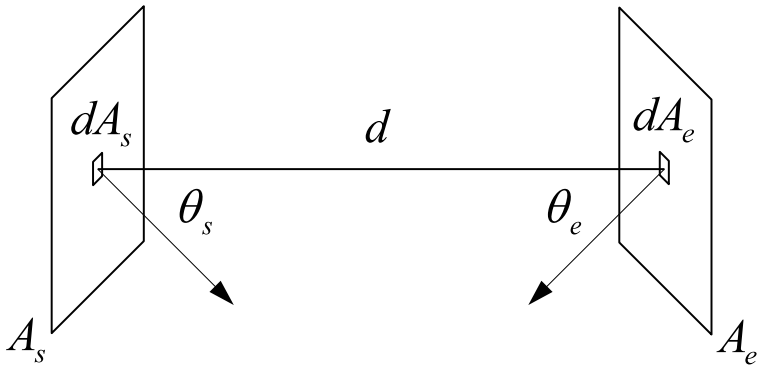

Abbildung 3.2.: Strahlungsaustausch

3.1.2. Die globale Beleuchtungsformel

Um den kompletten Strahlungsaustausch zwischen Flächen zu simulieren, wird das globale Beleuchtungsmodell in einer allgemeingültigen Formel repräsentiert. Diese basiert auf spektralen Parametern:

$$L_o(dA_e, d\vec{\omega_o}) = L_e(dA_e, d\vec{\omega_o}) + \int_{2\pi} \rho(dA_e, d\vec{\omega_i}, d\vec{\omega_o})\ L_i(dA_e, d\vec{\omega_i})\ \cos\theta_i\ d\omega_i \tag{3.7}$$

Eine detaillierte Untersuchung dieser Gleichung würde den Rahmen dieser Arbeit sprengen. Aus diesem Grund folgt eine kurze Erklärung einiger Bestandteile dieser Formel. Die Abgegebene Strahlung L_o („Licht“) einer Fläche dA_e in eine Richtung $d\vec{\omega_o}$ ergibt sich aus

- der Eigenemission L_e der Fläche in Richtung ω_o,
- der von allen Richtungen ω_i einfallenden Strahlung L_i, die in Richtung ω_o reflektiert wird, sowie
- dem Parameter ρ, der den Reflexionsgrad der Fläche repräsentiert und durch die Bidirektionale Verteilungsfunktion (BRDF) berechnet wird.

3.1.3. Das Radiosity-Gleichungssystem

Die Radiosity-Simulation basiert auf der Annahme verschiedener Faktoren, die es ermöglichen, die globale Beleuchtungsformel zu vereinfachen. Die wichtigsten Vereinfachungen werden an dieser Stelle kurz erläutert und die finale Radiosity-Gleichung vorgestellt. Diese dient als Basis für die Radiosity-Simulation und verschiedener in weiter unten vorgestellten Verfahren.
Die Vereinfachung der globalen Beleuchtungsformel basiert im Wesentlichen auf der Annahme, dass alle Flächen ideale diffuse (Lambert-)Strahler sind. Daraus resultiert, dass die Strahldichte $L(dA_e, d\vec{\omega_o})$ für ausgehende Strahlung einer Fläche richtungsunabhängig ist:

$$L(dA_e, d\vec{\omega_o}) \equiv L \tag{3.8}$$

Durch Formel 3.3 und 3.6 ergibt sich

$$B = \int_{\Omega} L \; cos\theta \; d\omega \tag{3.9}$$

Durch die Richtungsunabhängigkeit von L kann es vor das Integral gezogen werden:

$$\begin{aligned} B &= L \int_{\Omega} \cos\theta \; d\omega && (3.10)\\ &= L \int_0^{\pi} \int_0^{2\pi} \cos\theta \; \sin\theta \; d\theta \; d\varphi && (3.11)\\ &= L \; \pi && (3.12) \end{aligned}$$

Ähnlich verhält sich das mit der BRDF: auch sie wird unabhängig von der Richtung und kann vor das Integral aus der globalen Beleuchtungsformel 3.7 gezogen werden. Übrig bleibt dennoch eine Formel, die nicht analytisch gelöst werden kann (wegen des verbleibenden Integrals).

$$\begin{aligned} L_o(dA_e, d\vec{\omega_o}) &= L_e(dA_e, d\vec{\omega_o}) + \int_{2\pi} \rho(dA_e, d\vec{\omega_i}, d\vec{\omega_o}) \; L_i(dA_e, d\vec{\omega_i}) \; \cos\theta_i \; d\omega_i && (3.13)\\ L_o(dA_e) &= L_e(dA_e) + \frac{\rho(dA_e)}{\pi} \int_{2\pi} L_i(dA_e, d\vec{\omega_i}) \; \cos\theta_i \; d\omega_i && (3.14) \end{aligned}$$

Die Lösung besteht nun darin, die Flächen einer Szene zu diskretisieren. Durch eine finite Menge an Flächen (Patches) löst sich das Integral-Problem weitestgehend auf; übrig bleibt eine Summe über alle Sender-Flächen. Die finale Radiosity-Formel lautet demnach wie folgt:

$$B_e = E_e + \rho_e \sum_{s=1}^{N} B_s \; F_{es} \tag{3.15}$$

Zur Deutung: Die Radiosity, die eine Fläche e abstrahlt, ergibt sich aus der Eigenemission E_e und der um den Reflexionsgrad ρ_e verminderten empfangenen Radiosity aus allen Sender-Flächen der Szene. B_s ist die Radiosity eines spezifischen Sender-Patches s. Der Parameter F_{es} wird als Formfaktor bezeichnet und ist der Anteil der Strahlung, der von einem Sender-Patch in die Szene abgegeben und von einem anderen Patch empfangen wird. Der Formfaktor kann näherungsweise durch die 1. oder 2. Formfaktorvereinfachung effizient berechnet werden. Der Geschwindigkeit halber findet die 2. Formfaktorvereinfachung Anwendung, da es sich im Allgemeinen sowohl bei der Sender- als auch der Empfängerfläche um kleine Flächen handelt:

$$F_{es} = V_{se} \frac{\cos\theta_s \cos\theta_e}{\pi d^2} A_s \tag{3.16}$$

Der Formfaktor enthält demnach leicht zu ermittelnde Werte (θ_s, θ_e, die Senderfläche A_s, sowie den Abstand zwischen beiden Flächen d), sowie einen binären Visibilitäts-Operator.

3.2. Betrachtung der Ausgangslage und Parametrierung

Ein Radiosity-System basiert auf dem Prinzip, dass Strahlung ausgetauscht wird. Wird nun ein einzelnes Objekt einer Szene betrachtet, so ergeben sich verschiedene Effekte für die Gesamtszene, wie Schattenwurf und indirekte Beleuchtung. Soll nun dieses Objekt entfernt werden, müssen diese Effekte zurückgerechnet werden. Dies bedeutet, dass der Strahlungsaustausch dieses Objektes mit der gesamten Szene invertiert werden muss. Betrachtet wird nun folgender simpler Fall, der aber Allgemeingültigkeit besitzt: Gegeben sei ein geometrisches Objekt, z.B. ein Quader („Kiste“), der auf einer Ebenen steht („Boden“). Die Hauptbeleuchtung kommt von Patches von der Decke („Lampen“), aber auch aus indirekter Beleuchtung von allen anderen Patches („Wände und andere Objekte im Raum“). Angenommen, der Schatten fällt nur auf den Boden, so muss der Strahlungsaustausch nur für die Patches des Bodens berechnet werden. Daraus ergeben sich folgende drei Parameter:

1. eine Menge von Patches, die das zu entfernende Objekt bilden,
2. eine Menge von Patches, die eine Ebene bilden, aus der der Schatten des Objektes entfernt werden soll [1], sowie
3. eine Menge von Patches, in denen Bereiche durch das zu entfernende Objekt verdeckt sind und sich in der Ebene von Punkt 2 befindet; diese müssen rekonstruiert werden.

Gegeben sei ferner eine HDR-Lightprobe im Lat-Long-Format, die auf eine Kugeloberfläche projiziert wird (ähnlich Apples Quicktime VR) und die gesamte Geometrie der Szene. Die

[1] der reale Strahlungsaustausch betrifft immer alle Patches; dies stellt jedoch keine Einschränkung dar, da diese Menge beliebig erweitert werden kann. Aus Performancegründen und der Einfachheit halber wird diese Annahme getroffen

Ausgabe soll eine HDR-Textur der Differenzwerte der Lightprobe sein, die im Renderlauf auf die Lightprobe aufaddiert wird.

3.3. Die Theta-Phi-Projektion

Bei der Theta-Phi-Projektion oder Mercator-Projektion wird eine 2D-Textur auf eine 3D-Kugel projiziert. Die HDR-Lightprobe stellt eine Environment-Map dar, die auf diese Art und Weise auf eine (Einheits-)Kugel projiziert wird. Nun müssen an verschiedenen Stellen Umrechnungen vom Theta-Phi-Raum in den 3D-Raum und umgekehrt durchgeführt werden. Die Lightprobe enthält Radiance-Werte (basierend auf einem Foto), die zu weiteren Berechnungen im 3D-Raum benötigt werden (z.B. die Formfaktor-Berechnung der 2. Vereinfachung basiert auf dem Abstand von zwei Flächen im Raum, deren Größe und dem Radiance-Wert des Sender-Patches). Betrachtet werden die Umrechnung von Polarkoordinaten (θ, φ) in 3D-Koordinaten (x, y, z), ausgehend von der Einheitskugel und dem Mittelpunkt in $(0, 0, 0)$:

$$x = \sin(\theta)\cos(\varphi) \tag{3.17}$$
$$y = \cos(\theta) \tag{3.18}$$
$$z = \sin(\theta)\sin(\varphi) \tag{3.19}$$

Umrechnung ergibt:

$$\theta = \arccos(y) \tag{3.20}$$
$$\varphi = \arctan 2(z, x) \tag{3.21}$$

Zu jedem 3D-Punkt existiert eine Polarkoordinate, die wiederum in eine Texturkoordinate $(u, v), u \in [0, breite] \wedge v \in [0, höhe]$ umgerechnet werden muss. Ist φ negativ (da $\arctan 2 \in [-\pi, \pi]$), wird 2π aufaddiert. Die Umrechnung in die eigentliche Texturkoordinate (u, v) berechnet sich wie folgt:

$$u = breite\ \frac{2\pi - \varphi}{2\pi} \tag{3.22}$$
$$v = höhe\ \frac{\pi - \theta}{\pi} \tag{3.23}$$

3.4. Berechnung der Reflexionsgrade der Ebene

Um in späteren Abschnitten Schatten zu entfernen und die Textursynthese durchzuführen, muss zunächst Vorarbeit auf die aktuelle Bearbeitungs-Ebene geleistet werden. Diese Ebene repräsentiert eine möglichst pixelgenaue Projektion der Umgebung. Diese Ebene wird in einem späteren Schritt wiederum auf die Differenz-Lat-Long-Textur aufaddiert. Um Radiosity-Berechnungen durchführen zu können, muss für jedes „Pixel“ dieser Ebene der Reflexionsgrad bestimmt werden. Da durch die Projektion die Radiosity-Werte bekannt sind, lässt sich

der Reflexionsgrad aus Formel 3.15 und vernachlässigter Eigenemission („Alle Patches sind reine diffuse Lambert-Strahler“) wie folgt berechnen:

$$B_e = \rho_e \sum_{s=1}^{N} B_s \, F_{es} \tag{3.24}$$

$$\rho_e = \frac{B_e}{\sum_{s=1}^{N} B_s \, F_{es}} \tag{3.25}$$

Der Reflexionsgrad eines Empfängerpatches e ergibt sich demnach aus dem Quotienten aus der Radiosity des betrachteten Pixels (sie ist bekannt) und der Summe aller auf diesem Pixel ankommenden Radiosity-Strahlung aller anderen Patches. Da die Geometrie mit den anderen Radiosity-Werten vorhanden ist, so ist es ein leichtes, den Reflexionsgrad pro Pixel zu berechnen. In Pseudocode bedeutet das:

```
für alle Empfänger-Pixel p der Ebene {
   r = 0;
   für alle Sender-Patches s der Umgebung {
      Berechne Formfaktor FF zwischen s und p;
      r = r + Radiosity(s)*FF;
   }
   Reflexionsgrad(p) = Radiosity(p) / r;
}
```

3.5. Schatten entfernen

Schatten entfernen bedeutet nichts anderes, als fehlendes Licht aufzuaddieren. Dazu muss für jedes Pixel der Ebene ein Schnittest mit jedem Patch der Szene durchgeführt werden. Ein zu entfernender Schatten ist dann vorhanden, wenn das zu entfernende Objekt beim Schnitttest getroffen wird und sich zwischen Objekt und Ebene sonst kein weiteres Patch mehr befindet. In diesem Fall wird der Formfaktor zwischen Ebenen-Pixel und Szenen-Patch berechnet, die Radiosity des Patches ermittelt und daraus wiederum der Reflexionsgrad berechnet. Aus Formel 3.24 ist zu erkennen, dass sich die Radiosity aus der Summe des „Senderlichtes“ multipliziert mit dem Reflexionsgrad ergibt. Dadurch kann der Schatten „entfernt“ werden, in dem das fehlende Licht auf die schon vorhandene Radiosity aufaddiert wird.

$$B_e = B_e + \rho_e \, B_s \, F_{es} \tag{3.26}$$

Die Formfaktorberechnung enthält einen binären Visibilitäts-Operator. In diesen kann gleichzeitig auch der oben genannte Schnitttest integriert werden, so dass auch in diesem Fall doppelte Berechnungen entfallen.

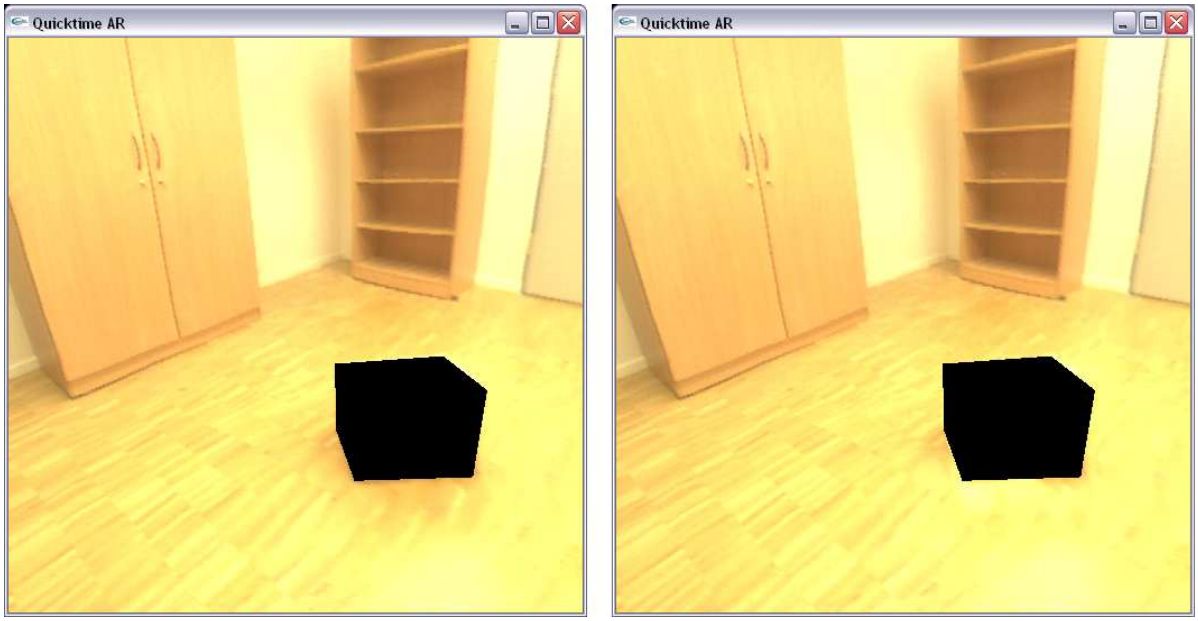

Abbildung 3.3.: Entfernung des Schattens

3.6. Indirektes Licht vom entfernten Objekt abziehen

Im vorangegangenen Abschnitt wurde das Licht auf die Ebene hinzuaddiert, welches durch die Verdeckung durch das zu entfernende Objekt entstanden ist („Schatten entfernen“). Dieser Abschnitt beschäftigt sich nun mit dem Licht, welches von dem zu entfernenden Objekt in die gesamte Szene ausgestrahlt wird. Nach dem Wegfall dieses Objektes muss dieses Licht wiederum aus der Szene entfernt werden. Dies basiert im Prinzip auf dem schon im vorangegangenen Abschnitt vorgestellten Verfahren, jedoch mit umgekehrtem Vorzeichen.
Wird das entfallende Objekt betrachtet, so strahlt dieses Licht in die Szene aus, nämlich für jedes Empfängerpatch e und dem Senderpatch s, welches Bestandteil des Objektes ist:

$$B_e = B_{restlicheSzene} + \rho_e \, B_s \, F_{es} \tag{3.27}$$

$$B_{restlicheSzene} = B_e - \rho_e \, B_s \, F_{es} \tag{3.28}$$

Es wird folglich das Licht von der Radiosity eines Pixel der Ebene abgezogen, das von einem Patch des zu entfernenden Objektes ausgestrahlt wird, und das Ebenen-Pixel trifft. Der Visibilitäts-Operator der Formfaktorberechnung berücksichtigt demnach solche Fälle, in denen das Ebenen-Pixel und das Patch des Objektes zueinander gewendet sind und zwischen denen keine weiteren Objekte stehen.

3.7. Textursynthese im verdeckten Bereich

Nachdem in der Ebene die Schatten entfernt wurden und das Licht, das vom zu entfernenden Objekt in die Szene gestrahlt wird, weggerechnet wurde, kann die Textursynthese im Bereich des Objektes erfolgen. Betrachtet wird nochmals das Einführungsbeispiel: eine Kiste steht auf dem Boden. Nun verdeckt die Kiste einen Teil des Bodens - dieser muss rekonstruiert werden. In Abbildung 3.4 ist die weiß markierte Fläche diejenige, die synthetisiert werden muss.

Abbildung 3.4.: Ausmaskiertes zu entfernendes Objekt

Aus Abschnitt 3.2 stammt die Anforderung, dass eine Bitmaske für die Ebene mit dem ausmaskierten Bereich existiert. Diese kann nun zusammen mit den Reflexionsgraden an die Textursynthese übergeben werden. Der Synthesealgorithmus vervollständigt den Bereich anhand der Bitmaske. Im Anschluss an dieses Verfahren müssen aus den Reflexionsgraden in diesem Bereich die Radiosity-Werte berechnet werden. Dies wiederum erfolgt analog den schon bekannten Algorithmen und Berechnungen aus den vorherigen Abschnitten.

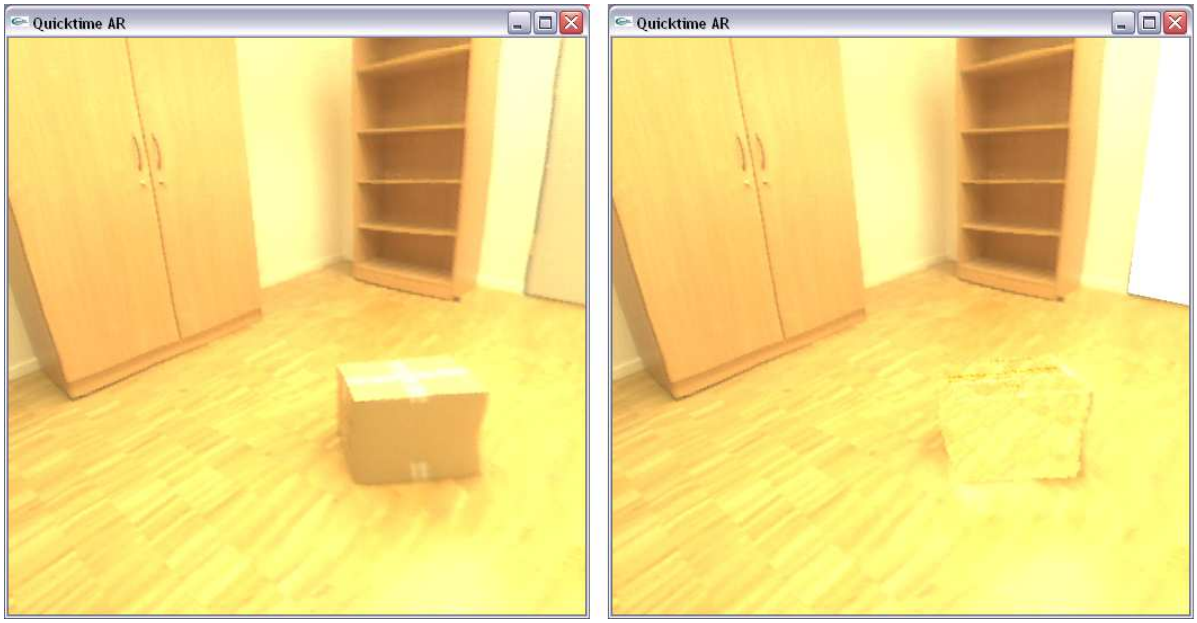

Abbildung 3.5.: Textursynthese mit Schattenentfernung

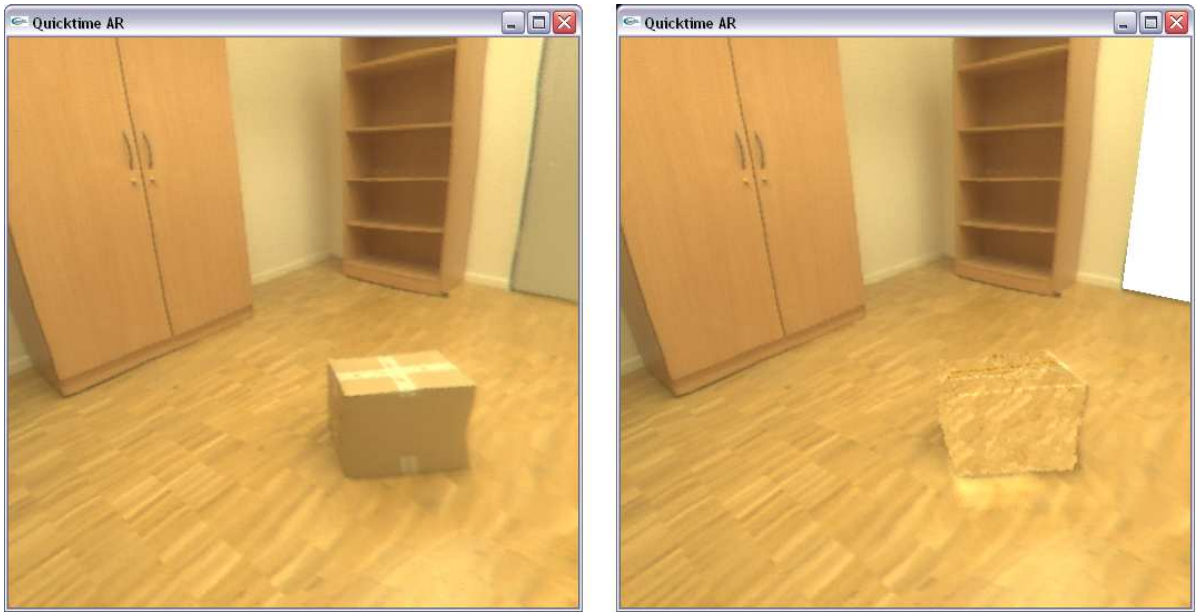

Abbildung 3.6.: Textursynthese mit Schattenentfernung, dunkler

Anmerkung zu den Abbildungen: Die noch sichtbaren Ränder der Kiste resultieren primär aus drei Faktoren:

1. der nicht genauen Positionierung der Geometrie in Bezug zur Environment-Map (die Geometrie der Kiste ist minimal zu klein)
2. der manuellen approximierten Berechnung der Ebenen-Textur (die Ebene liegt etwas schräger als der Originalboden)
3. sowie dem Rundungsfehler durch Umrechnung von Theta-Phi- in 3D-Koordinaten und umgekehrt

Dadurch überdecken sich die Environment-Map und die Differenz-Textur nicht exakt.

4. Implementierungsdetails

Während im Kapitel 3 ein Algorithmus zur Entfernung eines Objektes vorgestellt wurde, so dient dieser Abschnitt der Betrachtung der praktischen Implementation. Es werden vor allen Dingen Designentscheidungen und Eigenheiten erläutert.

4.1. Betrachtung des vorhandenen Frameworks

Das vorliegende Framework basiert auf der Arbeit von [Gro05]. In diesem sind im Wesentlichen die folgenden für diese Arbeit relevanten Strukturen integriert:

- eine HDR-LatLong-Textur mit der Umgebung („Kugelfoto“)
- ein Feld von Dreiecken für die Szenen-Geometrie
- eine Dreiecksklasse für geometrische sowie photometrische Parameter
- ein Feld mit Materialeigenschaften
- ein Feld, das Parameter der wichtigsten Lichtquellen enthält
- eine Renderingfunktion

Die HDR-Lat-Long-Textur wird wie bereits an früherer Stelle erwähnt per Theta-Phi-Projektion auf eine Kugeloberfläche projiziert. Jedem 3D-Punkt der Geometrie kann ein 2D-Texturpunkt zugeordnet werden, so dass die Textur auf die Geometrie gemappt werden kann. Die Zuordnung erfolgt in Shadern.
Das Feld von Dreiecken hält die Szenengeometrie vor. Neben den üblichen geometrischen Parametern wie Dreieckspunkt im Raum, Normale, Mittelpunkt, Fläche und Abstand zum Ursprung wird in der Dreiecksklasse auch ein Material-Index verwaltet. Der Material-Index verweist auf ein konkretes Material im Material-Feld. Ferner ist jedes Dreieck mit einer bestimmten Auflösung in Patches unterteilt. Diese Patches dienen der Radiosity-Simulation; so sind hier die wesentlichen fotometrischen Parameter wie Radiance und Reflectance hinterlegt.
Das Material-Array dient der Repräsentation einer Gruppe von geometrisch zusammengehörigen Dreiecken zu einem Objekt (z.B. Schrank, Regal, Kiste). Hierin sind Reflexionswerte festgehalten.
Um die Radiosity-Simulation möglichst schnell (in Echtzeit) durchzuführen, wird nicht über alle Patches der Geometrie iteriert, sondern nur über eine Anzahl der wichtigsten (leuchtstärksten) Patches. Standardmäßig werden 32 wichtige Lichtquellen bestimmt. Da diese Patches

den stärksten Effekt hervorrufen (Schatten), ist kaum ein subjektiver Unterschied zu erkennen.
In der Rendering-Funktion werden alle Parameter verwendet, um daraus ein Bild zu generieren. Dies geschieht durch die OpenGL-Display-Funktion. Je nach Anzahl der Lichtquellen und der Methode des Schattenverfahrens liegt die Framerate auf dem Entwicklungssystem [1] zwischen 2 und 25 Frames pro Sekunde.

4.2. Eigenschaften der Ebenen-Klasse

Die vorhandene Geometrie-Struktur basiert auf Dreiecken, die in eine bestimmte Anzahl Patches aufgelöst ist. Wird nun beispielhaft die Kiste auf dem Boden betrachtet (siehe Abbildung 3.3), so ist ersichtlich, dass der Schatten auf eine Ebene – dem Boden – fällt. Selbst für mehrere Wände ist intuitiv ersichtlich, dass diese zunächst auch nur Ebenen sind. Zur Texturesynthese und dem Herausrechnen der Schatten ist die Auflösung in Patches zu grobgranular. Eine Auflösung im Pixelbereich ist nötig! Aus diesen Beweggründen enstand eine rechteckige Ebenenklasse mit einer vorgegebenen aber variablen Auflösung (z.B. 512x512). Dies ist zudem sinnvoll, da alle Bodendreiecke den selben Materialindex besitzen und sich dadurch eine Gesamtebene berechnen lässt (in der Implementierung ist diese Ebene händisch vorgegeben worden, deren Berechnung nicht Bestandteil dieser Arbeit ist). Ferner kann iterativ jede Fläche durch diese Ebene speichersparend verwendet werden, so dass die Szene nach und nach bearbeitet wird und in die neue Differenz-Textur geschrieben werden kann.

Zu den Eigenschaften der Ebene gehören:

- Aufpunkt im 3D-Raum
- Spannvektoren s und t
- Normale
- Abstand zum Ursprung
- Koordinaten eines jeden 2D-Ebenen-Punktes (Texturkoordinate) im 3D-Raum und im Theta-Phi-System
- Reflectance und Radiance eines jeden Punktes
- ob ein Ebenen-Punkt durch ein anderes Objekt verdeckt ist
- ob ein Punkt durch das zu entfernende Objekt verdeckt ist, dieser Punkt also synthetisiert werden muss

[1] NVidia GeForce 6800 mit 128 MB Texturspeicher, Intel Pentium 4 mit 2.8 GHz, 1024 MB Arbeitsspeicher

- eine Methode „isInside“, die die Ebenen-Koordinate zurückgibt, falls der übergebene 3D-Punkt in der Ebene liegt

Jede Eigenschaften der Texturkoordinate ist separat in einem eigenen Feld (Array) abgelegt, so dass diese mit den OpenGL-Texturkommandos effizient verarbeitet werden können. So dient z.B. die Synthese-Eigenschaft als Textur-Bit-Maske für die Textursynthese.

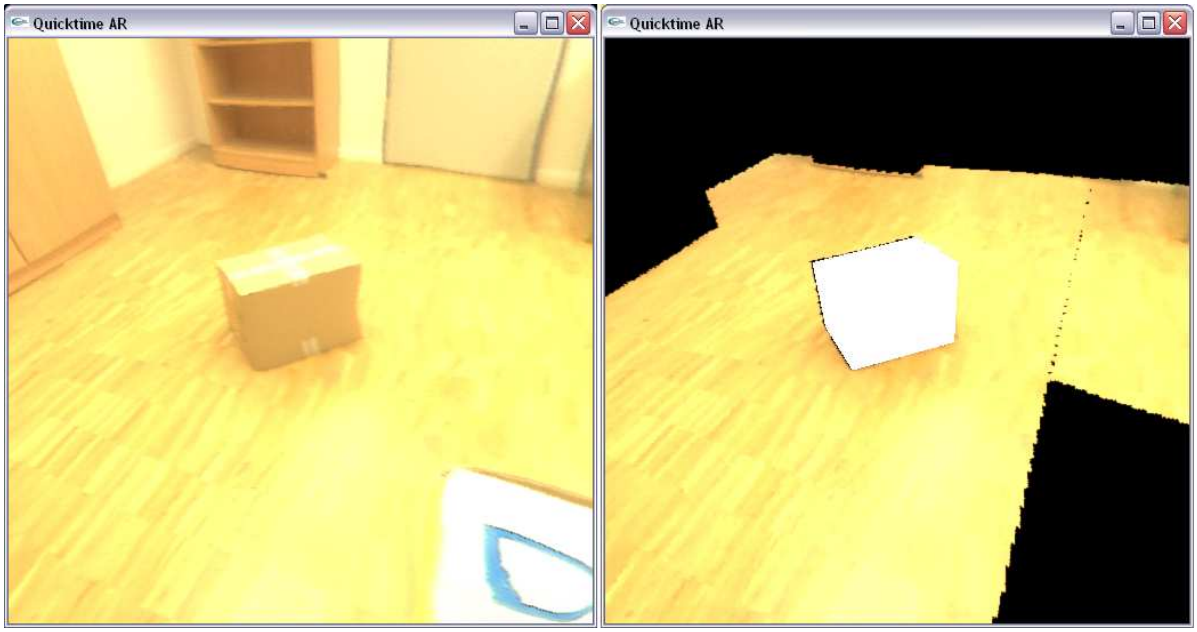

Abbildung 4.1.: Projektion der Umgebung auf eine Ebene

4.3. Unterteilung der Reflexionsgrade in wichtige und unwichtige Strahlung

Betrachtet wird eine Radiosity-Szene: jedes Patch beleuchtet jedes andere. Dieses andere wiederum beleuchtet alle anderen Patches, solange bis die Radiosity-Simulation konvergiert. Somit ist diese Simulation von einem Aufwand höherer Ordnung und in Echtzeit kaum zu berechnen. Untersucht man aber nun, woher das meiste Licht kommt, so stellt sich heraus, dass diese meist von einigen wenigen Lichtquellen stammen. Zur Beschleunigung der Simulation wird nun die Strahlung in wichtige und unwichtige eingeteilt. Die unwichtige wird als Konstante für größere Patches angesehen, während die wichtigen (für eine bestimmte Anzahl an leuchtstärksten Patches) pixelgenau verrechnet wird. Die stärksten Schatteneffekte treten durch diese „Lichtquellen“ auf. Die unwichtige Strahlung kann vorberechnet und später in die Ebenen-Pixel einfach eingerechnet werden, während für die wichtige Strahlung pixelgenaue Schnitttests und genaue Berechnungen der Reflexionsgrade durchgeführt werden. So ist der

Aufwand im Vergleich zur konvergierten Simulation hauptsächlich auf das Kalkulieren der wichtigen Strahlung beschränkt (die unwichtige Strahlung wird vorberechnet).

4.4. Bilineare Interpolation der unwichtigen Reflexionsgrade

Im vorigen Abschnitt wurde darauf eingegangen, dass sich der Gesamt-Reflexionsgrad aus der unwichtigen und der wichtigen Strahlung zusammensetzt. Die unwichtige Strahlung wird dabei für größere Patches berechnet (also einer niedrigeren Auflösung). Dazu wird auf die betrachtete Ebene ein grobes Raster gelegt. Beispielsweise wird eine Ebene mit einer Auflösung von 512x512 Pixeln in nxn (z.B. n = 16) unwichtige Strahlungs-Patches gerastert. Für jeden Eckpunkt dieses Rasters wird der Reflexionsgrad ohne die wichtigen Lichtpatches in einer (n+1)x(n+1)-Matrix gesammelt. Dies ist die vorberechnete unwichtige Strahlung.
Bei der Pro-Pixel-Strahlung der Ebene wird auf den ermittelten Reflexionsgrad aus den wichtigen Lichtpatches ein bilinear interpolierter Wert aus den unwichtigen Reflexionsgraden aufaddiert. Zunächst wird die Position des Ebenen-Pixels in der Rastermatrix bestimmt. Hier angegeben ist eine Eck-Koordinate des Rasters; die drei anderen sind dadurch unmittelbar gegeben:

$$R_{x,y} = \lfloor P/Auflösung \rfloor \tag{4.1}$$

Für die bilineare Interpolation wird die lineare Interpolationsformel für zwei Wert A und B

$$lerp(A, B, t) = (1 - t)A + tB \tag{4.2}$$

für die jeweils eine Achsen-Richtung (in diesem Fall die x-Achse) angewendet und die daraus resultierenden Werte in die andere Achsen-Richtung (y-Achse) interpoliert. Der Parameter t_x und t_y für die jeweilige Achsenrichtung berechnet sich wie folgt:

$$t_x = \frac{x \ mod \ Auflösung}{Auflösung} \tag{4.3}$$

$$t_y = \frac{y \ mod \ Auflösung}{Auflösung} \tag{4.4}$$

Es folgt der Pseudocode der bilinearen Interpolation der unwichtigen Strahlung. Das Array mit den gerasterten Werten ist als *mUIIrrad*, sowie die Auflösung als *uiResPixels* bezeichnet.

```
int xm = x / uiResPixels;
int ym = y / uiResPixels;
float tx = ((float)(x \% uiResPixels)) / ((float)uiResPixels);
float ty = ((float)(y \% uiResPixels)) / ((float)uiResPixels);
irradUI_X1 = (1 - tx) * mUIIrrad[ym][xm] + tx * mUIIrrad[ym][
   xm+1];
```

```
irradUI_X2 = (1 - tx) * mUIIrrad[ym+1][xm] + tx * mUIIrrad[ym
   +1][xm+1];
irradUI_Y = ((1 - ty) * irradUI1 + ty * irradUI2);
```

4.5. Abbilden der Geometrie zurück in eine Theta-Phi-Textur

Im naiven Ansatz würde die 3D-Position eines jeden Pixels der Ebene in Theta-Phi-Koordinaten umgerechnet werden. Da die Lat-Long-Textur jedoch diskrete Positionswerte besitzt, bleiben einige Pixel leer, da bei der Umrechnung auf diskrete Werte gerundet werden muss.
Um sicherzustellen, dass jedem Wert in der Lat-Long-Textur ein Wert aus der Ebene zugeordnet wird, ergibt sich das folgende Verfahren: Für jeden Pixel der Lat-Long-Textur wird dessen korrespondierende Theta-Phi-Koordinate berechnet und diese wiederum in eine 3D-Position umgewandelt (siehe 3.3). Da die Lat-Long-Textur zur Darstellung auf eine Kugeloberfläche im 3D-Raum projiziert wird, kann ein Schnittpunkttest von der 3D-Position der Textur-Koordinate in Richtung der Ebene erfolgen. Das Ergebnis ist ein Schnittpunkt mit der Ebene im Raum, sowie dessen Abstand. Es werden nur solche Werte berücksichtigt, die einen Schnittpunkt mit der Ebene besitzen und der zu untersuchenden Textur-Koordinate im Raum zugewendet sind (der Abstand also einen positiven Wert besitzt). Sind diese Kriterien erfüllt, kann durch den Schnittpunkt die konkrete Koordinate in der Ebenen-Textur bestimmt werden.

```
// PICWIDTH und PICHEIGHT sind die Ausmaße der Lat-Long-Textur
    in Pixeln
pos[0] = doPlane->getPos()[0];
pos[1] = doPlane->getPos()[1];
pos[2] = doPlane->getPos()[2];
normal = doPlane->getNormal();
dist = doPlane->getDistance();
for (int y = 0 ; y < PICHEIGHT ; y++) {
   for (int x = 0 ; x < PICWIDTH ; x++) {
      float t;
      float theta = (PICHEIGHT - y) * PI / (float)(PICHEIGHT);
      float phi = (PICWIDTH - x) * 2*PI / (float)(PICWIDTH);

      dir[0] = sin(theta)*cos(phi);
      dir[1] = cos(theta);
      dir[2] = sin(theta)*sin(phi);

      //find proper Pixel;
      bool hit = calcRayPlaneIsect(&t, isectPoint, dir, normal
         , dist);
```

```
      if (hit && t > 0) {
         int px;
         int py;
         if (doPlane->isInside(&px, &py, isectPoint[0],
            isectPoint[1], isectPoint[2])) {
            Vector c;
            c[0] = doPlane->getRadiance(px, py)[0];
            c[1] = doPlane->getRadiance(px, py)[1];
            c[2] = doPlane->getRadiance(px, py)[2];

            latLongImage2->setHDRPixelXY(x, y, c[0], c[1], c
               [2]);
         }
      }
   }
}
```

5. Einfügen des realen Objektes

Dieses Kapitel skizziert einen *Ansatz*, wie das entfernte Objekt an einer anderen Stelle eingefügt werden kann. Dabei kommen die in den vorangegangenen Kapiteln vorgestellten Verfahren leicht verändert zur Anwendung.

5.1. Einfügeposition bestimmen

Das Einfügen des Objektes an anderer Stelle lässt sich auf die Translation und Rotation der Objektgeometrie an einen anderen Punkt im Raum zurückführen. Diese Rotation kann jedoch nicht beliebig sein, da der „Blickwinkel“ auf das ursprüngliche Objekt durch das Umgebungsbild starr vorgegeben ist. Eine Fläche des Objektes, die vor der Rotation nicht zu sehen war, darf auch nach der Rotation nicht sichtbar sein. Daraus resultiert, dass die Rotation starr vorgegeben ist, nämlich so, dass immer derselbe Blickwinkel zum Objekt bestehen muss. Die Theta-Phi-Projektion basiert auf der Projektion des Umgebungsbildes auf eine imaginäre Kugel. Zur Berechnung des Rotationswinkels wird eine imaginäre Gerade vom Kugelmittelpunkt ausgehend zum Objektmittelpunkt gezogen. Der Winkel zwischen dem Objekt und dieser imaginären Gerade muss immer identisch sein. Daraus ergibt sich die Bedingung für die Rotation. Zur Visualisierung: „Das Objekt rotiert um den Kugelmittelpunkt der Projektion. Der Abstand zwischen Objekt und Mittelpunkt kann beliebig im positiven Bereich verändert werden.

5.2. Beleuchtung in der Szene anpassen

Die photometrische Information ist mit der Geometrie verknüpft, d.h. diese Information steht auch nach einer Translation zur Verfügung. Das Verfahren sieht nun vor, zunächst die Schatten auf dem Objekt selbst ähnlich dem in Kapitel 3.5 beschriebenen Algorithmus zu entfernen. Das Objekt setzt sich aus einzelnen Patches zusammen, die wiederum auf eine Ebenen-Textur im Raum projiziert werden können. So findet die Schattenentfernung analog dem vorgenannten Verfahren statt. Im Anschluss daran wird die neue Position bestimmt und die Szenenbeleuchtung nach dem Verfahren zur Beleuchtung eines virtuellen Objektes in [Gro05] adaptiert.

5.3. Ausblick: Freie Transformation

Das Auflösen der Restriktionen der Transformation aus Abschnitt 5.1 bedingt, dass Flächenteile nach der Transformation sichtbar werden können, die vorher nicht sichbar waren. Diese Teil-

flächen beinhalten zunächst keine vorhandenen photometrischen Informationen und müssen z.B. anhand des Verfahrens aus Sektion 3.4 ermittelt und anschließend mit der Textursynthese aus Kapitel 2 vervollständigt werden.

6. Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde ein Verfahren vorgestellt, mit dem sich ein beliebiges reales Objekt einer Szene in einem radiosity-basierten Panorama-Bildbetrachter entfernen lässt. Dabei wurde größter Wert auf eine schnelle Verarbeitung im Echtzeitbereich gelegt. Auch der Photorealismus sollte beibehalten werden, weshalb Schatten von dem zu entfernenden Objekt herausgerechnet werden und das Licht, das vom Objekt selbst in die Szene abgestrahlt wird, entfernt. Die Lücke, die das Objekt in der Environment-Map hinterlässt, wird durch einen effizienten Synthesealgorithmus geschlossen.
Dieser Synthesealgorithmus sollte den Anforderungen an ein Foto gerecht werden, dabei jedoch auch schnell arbeiten. Um dies zu erreichen wurden simpel steuerbare Parameter — z.B. die Nachbarschaftsgröße und die Pyramidenlevel und -generation — implementiert, um eine intuitive Feinjustierung zu ermöglichen. Neben der Effizienz sollten auch schon vorhandene Strukturen einer Textur berücksichtigt werden, die das zu füllende Loch umgeben. Ferner wurde an der Möglichkeit gearbeitet, die Textursynthese pro Pyramiden-Level durchzuführen. Ein niedrigeren Level kann schnell berechnet werden und demnach schnell angezeigt werden. In weiteren Rendering-Läufen kann die Textur schrittweise verbessert werden. So ergeben sich für den Benutzer geringe Latenzzeiten, da zunächst die gröbere Texturauflösung (Level of Detail) angezeigt wird und z.B. in einer Idle-Funktion die entsprechend höheren Synthese-Level weitergerechnet werden können. Dies wird unter anderem durch die Unabhängigkeit von der Verarbeitungsrichtung erreicht.
Der Vollständigkeit halber wurde ein Ansatz skizziert, der die schon vorgestellten Algorithmen ausnutzt, um das entfernte Objekt an einer anderen Stelle im Raum wieder auftauchen zu lassen. Dabei wurden Restriktionen in der Bewegungsfreiheit angenommen, um das Verfahren zunächst einfach zu halten. Es wurde ferner kurz erläutert, welche Probleme durch das wegfallen dieser Einschränkungen auftreten.

Die vorliegende experimentelle Implementierung führt fast alle Berechnungen ausschließlich auf der CPU aus. Ein Ansatz der Geschwindigkeitssteigerung ist, die Textursynthese auf die Graphikhardware auszulagern. Gerade große Texturspeicher und die Parallelverarbeitung machen eine moderne GPU prädestiniert für diesen Zweck. Bei der Objektentfernung könnte eine verbesserte Randbehandlung im Synthese-Bereich wahrscheinlich einen nahtlosen Übergang produzieren. Auch die momentan noch manuell generierte Ebenen-Textur könnte automatisch ermittelt werden. Dies würde eine genauere Untersuchung der kompletten Szenengeometrie voraussetzen. Weiterhin kann die Ebenen-Textur selbst als Mustertextur dienen. Um jedoch eine Vorauswahl an Mustertexturen zu erhalten, muss der Suchalgorithmus effizienter werden und darf nicht nach der Top-Down-Methode arbeiten. Eine Betrachtung im Frequenzraum

würde sicherlich robustere Möglichkeiten bieten. Da sich der Synthesealgorithmus selbst nicht auf einen Farbraum beschränkt, wäre eine Untersuchung in anderen Farbräumen sinnvoll, um ein besseres und effizienter berechenbares Ähnlichkeitsmaß zu finden.
Im Bereich des Verschiebens von Objekten sollten die Restriktionen aufhebbar sein. Erst ein frei dreh- und schwenkbares reales Objekt lässt eine komplette Modifikation der Szene zu. Erst dann kann den Anforderungen z.B. von Möbelhäusern gerecht werden, die Anhand von Kundenfotos ganze Räume am Computer umgestalten können. Die Vereinigung von einem realen und einem virtuellen Objekt zu einem Gesamtobjekt würde neue Möglichkeiten der Umgestaltung bringen. So könnten z.B. vorhandene reale Schrankelemente durch neue virtuelle Elemente ergänzt und/oder geändert werden. Solche Verarbeitungsmöglichkeiten lassen ganz neue Raumgestaltungsaspekte zu.

A. Quellcodes

A.1. Interface zur Textursynthese-Klasse

```
#ifndef _SYNTHESIZETEXTURE_H
#define _SYNTHESIZETEXTURE_H

#include <time.h>
#include <vector>
#include "Image.h"
#include "GaussianPyramid.h"
#include "Neighborhood.h"


class SynthesizeTexture
{
public:
   SynthesizeTexture(void);
   ~SynthesizeTexture(void);

   // Initialisiert eine neue Textur mit gegebener Größe x*y
   SynthesizeTexture(unsigned int width, unsigned int height);

   // Berechnet eine synthetisierten Textur anhand eines
      Eingangssamples
   Image* compute(Image* inImg);

   // Berechnet eine synthetisierten Textur anhand eines
      Eingangssamples
   void compute(float* synTex, bool* bitMask, int sX, int sY,
      int sWidth);

   // Gibt die synthetisierte neue Textur zurück
   Image* getTexture();
   //Image* getTexture(int level);
```

```
private:

protected:
   // Synthetisiert einen gegebenen Pixel
   float* synthesizePixel(int level, int generation, int x,
      int y);

   // Gibt die Ausgabe-Nachbarschaft zurück
   void buildOutputNeighborhood(int level, int generation, int
       x, int y, vector<float*>* nhd);

   // Gibt die Eingabe-Nachbarschaft zurück
   void buildInputNeighborhood(int level, int x, int y, vector
      <float*>* nhd);

   // Gibt ein Ähnlichkeitskriterium von zwei übergebenen
      Nachbarschaften zurück
   float matchNeighborhood(vector<float*>* a, vector<float*>*
      b);

   // Existiert ein Ausgabepixel p für einen bestimmten Level
      L und eine bestimmte Generation m im Cache?
   float* CacheValue(int level, int generation, int x, int y);

   bool CacheHit(float* color);

   // Einen neuen Eintrag im Cache vornehmen
   void AddCacheEntry(int level, int generation, int x, int y,
       float* color);

   void findBestMatch(int level, int x, int y, Image* outImg);

   void makeCache(int width, int height, int level, Image*
      inImg);
   int calculateNeighborhoodSize(int size);
   void initCache();

   unsigned int mwidth;
   unsigned int mheight;
   Image* mimg;
   GaussianPyramid* pyrIn;
   vector<GaussianPyramid*> pcache;
```

```
   float* mCacheFail;

   Neighborhood* xnhdIn;
};

#endif
```

A.2. Zentraler Bestandteil der Nachbarschafts-Generierung

```
vector< vector<float*> > Neighborhood::compute(GaussianPyramid
   * gp, int level, int nhdSize)
{
   vector< vector<float*> > nhdImg;
   vector<float*>* nhdPixel;
   int size = nhdSize;
   int sizeh = size/2;
   nhdPixel = new vector<float*>;
   Image* img;
   int mx, my;

   for (unsigned int y = 0; y < gp->at(level)->width(); y++) {
      for (unsigned int x = 0; x < gp->at(level)->height(); x
         ++) {
         nhdPixel = new vector<float*>;
         size = nhdSize;
         sizeh = size/2;
         mx = x;
         my = y;
         for (int l = 0; l < NEIGHBORHOOD_LEVELS; l++) {
            //if (level >= 2)
            // printf("h");
            if (level-l >= 0) {
               img = gp->at(level-l);
               for (int i = 0; i < size; i++) {
                  for (int j = 0; j < size; j++) {
                     for (int chan = 0; chan < 3; chan++) {
                        nhdPixel->push_back((img->at(mx-sizeh
                           + j, my-sizeh + i)+chan));
                     }
                  }
```

```
                }
            }
            else {
                for (int i = 0; i < size*size*3; i++) {
                    nhdPixel->push_back(&minus1);
                }
            }
            size /= 2;
            sizeh = size/2;
            mx /= 2;
            my /= 2;
        }
        nhdImg.push_back(*nhdPixel);
        printf("*");
      }
   }
   printf("\n");

   return nhdImg;
}
```

A.3. Interface der Ebenen-Klasse

```
#ifndef _DORECTANGLE_H
#define _DORECTANGLE_H

#include <vector>
#include "round.h"
#include "globals.h"
#include "Vector.h"


using namespace std;

class doRectangle
{
public:
   doRectangle(void);
   doRectangle(const Vector& pos, const Vector& s, const
      Vector& t);
   ~doRectangle(void);
```

```
int getResolution() { return _resolution; };
void makePoints();

Vector getNormal() { return _normal; };
Vector getPos() { return _pos; };
Vector getS() { return _s; };
Vector getT() { return _t; };

float getDistance() { return _distance; };
float* get3DPos(int px, int py) { return &_pixelpos[py][px
   ][0]; };
void set3DPos(int px, int py, float x, float y, float z) {
   _pixelpos[py][px][0] = x; _pixelpos[py][px][1] = y;
   _pixelpos[py][px][2] = z; };

int* getEnvMapPos(int px, int py) { return &_envmappos[py][
   px][0]; };
void setEnvMapPos(int px, int py, int theta, int phi) {
   _envmappos[py][px][0] = theta; _envmappos[py][px][1] =
   phi; };

float* getReflectance() { return &_reflectance[0][0][0]; };
float* getReflectance(int px, int py) { return &
   _reflectance[py][px][0]; };
void setReflectance(int px, int py, float x, float y, float
    z) { _reflectance[py][px][0] = x; _reflectance[py][px
   ][1] = y; _reflectance[py][px][2] = z; };
void addReflectance(int px, int py, float x, float y, float
    z) { _reflectance[py][px][0] += x; _reflectance[py][px
   ][1] += y; _reflectance[py][px][2] += z; };

float* getRadiance() { return &_radiance[0][0][0]; };
float* getRadiance(int px, int py) { return &_radiance[py][
   px][0]; };
void setRadiance(int px, int py, float x, float y, float z)
    { _radiance[py][px][0] = x; _radiance[py][px][1] = y;
   _radiance[py][px][2] = z; };
void addRadiance(int px, int py, float x, float y, float z)
    { _radiance[py][px][0] += x; _radiance[py][px][1] += y;
    _radiance[py][px][2] += z; };
```

```
   bool getSubsurface(int px, int py) { return _subsurface[py
      ][px]; };
   void setSubsurface(int px, int py, bool subsurface) {
      _subsurface[py][px] = subsurface; };

   bool* getSynthesize() { return &_synthesize[0][0]; };
   bool getSynthesize(int px, int py) { return _synthesize[py
      ][px]; };
   void setSynthesize(int px, int py, bool synthesize) {
      _synthesize[py][px] = synthesize; };

   bool isInside(int *px, int *py, float x, float y, float z);
   //bool isInside(float x, float y, float z);

private:
   Vector _pos;
   Vector _s;
   Vector _t;
   Vector _normal;
   int _resolution;
   float _distance;

   float _pixelpos[512][512][3];
   int _envmappos[512][512][2];
   float _reflectance[512][512][3];
   float _radiance[512][512][3];
   bool _subsurface[512][512];
   bool _synthesize[512][512];

};

#endif
```

A.4. Interface der Textur-Synthese-Klasse

```
#ifndef _SYNTHESIZETEXTURE_H
#define _SYNTHESIZETEXTURE_H

#include <time.h>
#include <vector>
#include "Image.h"
#include "GaussianPyramid.h"
```

```
#include "Neighborhood.h"


class SynthesizeTexture
{
public:
   SynthesizeTexture(void);
   ~SynthesizeTexture(void);

   // Initialisiert eine neue Textur mit gegebener Größe x*y
   SynthesizeTexture(unsigned int width, unsigned int height);

   // Berechnet eine synthetisierten Textur anhand eines
      Eingangssamples
   Image* compute(Image* inImg);

   // Berechnet eine synthetisierten Textur anhand eines
      Eingangssamples
   void compute(float* synTex, bool* bitMask, int sX, int sY,
      int sWidth);

   // Gibt die synthetisierte neue Textur zurück
   Image* getTexture();
   //Image* getTexture(int level);

private:

protected:
   // Synthetisiert einen gegebenen Pixel
   float* synthesizePixel(int level, int generation, int x,
      int y);

   // Gibt die Ausgabe-Nachbarschaft zurück
   void buildOutputNeighborhood(int level, int generation, int
       x, int y, vector<float*>* nhd);

   // Gibt die Eingabe-Nachbarschaft zurück
   void buildInputNeighborhood(int level, int x, int y, vector
      <float*>* nhd);

   // Gibt ein Ähnlichkeitskriterium von zwei übergebenen
      Nachbarschaften zurück
```

```
    float matchNeighborhood(vector<float*>* a, vector<float*>*
       b);

    // Existiert ein Ausgabepixel p für einen bestimmten Level
       L und eine bestimmte Generation m im Cache?
    float* CacheValue(int level, int generation, int x, int y);

    bool CacheHit(float* color);

    // Einen neuen Eintrag im Cache vornehmen
    void AddCacheEntry(int level, int generation, int x, int y,
        float* color);

    //// Gibt einen Cache-Wert zurück für ein Pixel p, dem
       Level L und der Generation m
    //void GetCacheValue(void);

    void findBestMatch(int level, int x, int y, Image* outImg);

    void makeCache(int width, int height, int level, Image*
       inImg);
    int calculateNeighborhoodSize(int size);
    void initCache();

    unsigned int mwidth;
    unsigned int mheight;
    Image* mimg;
    GaussianPyramid* pyrIn;
    vector<GaussianPyramid*> pcache;
    float* mCacheFail;

    Neighborhood* xnhdIn;

    float minus1;
};

#endif
```

A.5. Auszug aus der Textur-Synthese

```
void SynthesizeTexture::compute(float* synTex, bool* bitMask,
   int sX, int sY, int sWidth)
```

```
{
   Image* img = new Image(sWidth, sWidth, 0.5f);
   int xPos, Pos;
   int cx, cy;
   float* color = NULL;
   pyrIn = new GaussianPyramid(LEVELS);
   xnhdIn = new Neighborhood();
   for (int y = 0; y < sWidth; y++) {
      for (int x = 0; x < sWidth; x++) {
         Pos = (y*sWidth+x) * 3;
         xPos = ((sY+y)*mwidth+(sX+x)) * 3;
         *(img->at(Pos+0)) = synTex[xPos+0];
         *(img->at(Pos+1)) = synTex[xPos+1];
         *(img->at(Pos+2)) = synTex[xPos+2];
      }
   }
   pyrIn->compute(img, GAUSS_SIZE);
   xnhdIn->compute(pyrIn);

   cx = mwidth;
   cy = mheight;
   for (int i = 0; i < LEVELS-1; i++) {
      cx /= 2;
      cy /= 2;
   }

   initCache();
   makeCache(cx, cy, 0, img);

   printf("Generating␣initial␣Cache:␣%ix%i\n", mwidth, mheight
      );
   float* mc = new float[3];
   for (int y = 0; y < mheight; y++) {
      for (int x = 0; x < mwidth; x++) {
         Pos = (y*mwidth + x);
         xPos = Pos * 3;
         if (!bitMask[Pos]) {
            mc[0] = synTex[xPos+0];
            mc[1] = synTex[xPos+1];
            mc[2] = synTex[xPos+2];
            AddCacheEntry(LEVELS-1, GENERATIONS-1, x, y, mc);
            //printf(".");
```

```
         }
      }
   }
   GaussianPyramid* p = pcache[GENERATIONS-1];
   p->compute(GAUSS_SIZE);


   printf("Output-Size: %ux%u\n", mwidth, mheight);
   for (unsigned int y = 0; y < mheight; y++) {
      for (unsigned int x = 0; x < mwidth; x++) {

         Pos = (y*mwidth+x);
         xPos = Pos * 3;
         if (bitMask[Pos]) {
            color = this->synthesizePixel(LEVELS-1,
               GENERATIONS-1, x, y);
            synTex[xPos+0] = synTex[xPos+0] + color[0];
            synTex[xPos+1] = synTex[xPos+1] + color[1];
            synTex[xPos+2] = synTex[xPos+2] + color[2];
         }
      }
      printf(".");
   }
   printf("\n");
}
```

A.6. Projektion der Umgebung auf die Ebene

```
for (int y = 0; y < doPlane->getResolution(); y++) {
   for (int x = 0; x < doPlane->getResolution(); x++) {
      Vector dir(doPlane->get3DPos(x, y)[0], doPlane->get3DPos
         (x, y)[1], doPlane->get3DPos(x, y)[2]);
      tri = calc3DIsectPoint(pos, dir, isectPoint);

      if (tri != NULL) { //Schnittpunkt mit einem Objekt
         if (tri->getIsDeletedObject()) { // Schnittpunkt mit
            entferntem Objekt gefunden => Bereich muss ergänzt
             werden
            doPlane->setSynthesize(x, y, true);

            int EnvMapX = doPlane->getEnvMapPos(x, y)[0];
            int EnvMapY = doPlane->getEnvMapPos(x, y)[1];
```

```
            float* color = latLongImage->getHDRPixelXY(EnvMapX
               , EnvMapY);
            doPlane->setRadiance(x, y, -1 * color[0], -1 *
               color[1], -1 * color[2]);
         }
         else if (tri->getIsMarked()) { // Schnittpunkt mit zu
             ergänzender Ebene gefunden => Werte aus EnvMap
            einfügen
            int EnvMapX = doPlane->getEnvMapPos(x, y)[0];
            int EnvMapY = doPlane->getEnvMapPos(x, y)[1];
            //Wert aus EnvMap holen
            float* color = latLongImage->getHDRPixelXY(EnvMapX
               , EnvMapY);

            doPlane->setRadiance(x, y, color[0], color[1],
               color[2]);
         }
         else { // Schnittpunkt weder mit entferntem Objekt,
            noch mit zu ergänzender Ebene => Pixel verdeckt,
            nicht verwenden
            doPlane->setSubsurface(x, y, true);
         }
      }
      else { //Kein Schnittpunkt => ungültiges Pixel
         doPlane->setSubsurface(x, y, true);
      }
   } // x
} // y
```

A.7. Reflexionsgrade der Ebenen-Pixel berechnen

```
normal[0] = doPlane->getNormal()[0];
normal[1] = doPlane->getNormal()[1];
normal[2] = doPlane->getNormal()[2];
for (int y = 0; y < doPlane->getResolution(); y++) {
   for (int x = 0; x < doPlane->getResolution(); x++) {
      if (!doPlane->getSubsurface(x, y) && !doPlane->
         getSynthesize(x, y)) { // sichtbarer Bereich des zu
         Ergänzenden Polygons (Boden) ohne entferntes Objekt
         irrad = Vector(0.0, 0.0, 0.0);

         if (calc) { //Radiance-Values berechnen!
```

```
            pos[0] = doPlane->get3DPos(x, y)[0];
            pos[1] = doPlane->get3DPos(x, y)[1];
            pos[2] = doPlane->get3DPos(x, y)[2];

            // Calculate only important Irradiance
            for (int k = 0; k < numSenders; k++) {
               irrad = irrad + calcIrradianceV2DO(pos, normal,
                   senderPositions[k], senderNormals[k],
                  senderRadiances[k], senderAreas[k], false,
                  false);
            }
         }

         //Addding unimportant Radiance (bilinear
            interpolation)
         int xm = x / uiResPixels;
         int ym = y / uiResPixels;
         float tx = ((float)(x % uiResPixels)) / ((float)
            uiResPixels);
         float ty = ((float)(y % uiResPixels)) / ((float)
            uiResPixels);
         irradUI1 = (1 - tx) * mUIIrradDO[ym][xm] + tx *
            mUIIrradDO[ym][xm+1];
         irradUI2 = (1 - tx) * mUIIrradDO[ym+1][xm] + tx *
            mUIIrradDO[ym+1][xm+1];
         irrad = irrad + ((1 - ty) * irradUI1 + ty * irradUI2)
            ;

         // Writing values back to plane
         Vector B(doPlane->getRadiance(x, y)[0], doPlane->
            getRadiance(x, y)[1], doPlane->getRadiance(x, y)
            [2]);
         if (irrad[0] == 0.0 || irrad[1] == 0.0 || irrad[2] ==
             0.0) { //Verhindert Division durch Null!
            doPlane->setReflectance(x, y, 0.0, 0.0, 0.0);
         }
         else {
            Vector rho = B / irrad;
            doPlane->setReflectance(x, y, rho[0], rho[1], rho
               [2]);
         }
      }
```

```
   }
}
```

A.8. Rückprojektion der Ebene in eine Lat-Long-Differenz-Textur

```
pos[0] = doPlane->getPos()[0];
pos[1] = doPlane->getPos()[1];
pos[2] = doPlane->getPos()[2];
normal = doPlane->getNormal();
dist = doPlane->getDistance();
for (int y = 0 ; y < PICHEIGHT ; y++) {
   for (int x = 0 ; x < PICWIDTH ; x++) {
      float t;
      float theta = (PICHEIGHT - y) * PI / (float)(PICHEIGHT);
      float phi = (PICWIDTH - x) * 2*PI / (float)(PICWIDTH);

      dir[0] = sin(theta)*cos(phi);
      dir[1] = cos(theta);
      dir[2] = sin(theta)*sin(phi);

      //find proper Pixel;
      bool hit = calcRayPlaneIsect(&t, isectPoint, dir, normal
         , dist);
      if (hit && t > 0) {
            int px;
            int py;
            if (doPlane->isInside(&px, &py, isectPoint[0],
               isectPoint[1], isectPoint[2])) {
               Vector c;
               c[0] = doPlane->getRadiance(px, py)[0];
               c[1] = doPlane->getRadiance(px, py)[1];
               c[2] = doPlane->getRadiance(px, py)[2];
               latLongImage2->setHDRPixelXY(x, y, c[0], c[1],
                  c[2]);
            }
      }
   }
}
```

Literaturverzeichnis

[Azu95] R. Azuma. A survey of augmented reality, 1995.

[BA83] Peter J. Burt and Edward H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, COM-31,4:532–540, 1983.

[CW93] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Boston, MA, 1993.

[Deb98] Paul E. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *SIGGRAPH98*, 1998.

[EF01] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 341–346. ACM Press / ACM SIGGRAPH, 2001.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[Gro05] Thorsten Grosch. Panoar: Interactive augmentation of omni-directional images with consistent lighting. In *Proceedings of Mirage 2005 (Computer Vision / Computer Graphics Collaboration Techniques and Applications)*, INRIA Rocquencourt, France, March, 1-2 2005.

[Har] Mark Harris. Render-to-texture. http://www.markmark.net/misc/rendertexture.html.

[HB95] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *SIGGRAPH*, pages 229–238, 1995.

[LB01] Vincent Lepetit and Marie-Odile Berger. A semi-interactive and intuitive tool for outlining objects in video sequences with application to augmented and diminished reality. In *International Symposium on Mixed Reality (ISMR2001)*, 2001.

[LFD+99] C. Loscos, M.-C. Frasson, G. Drettakis, B. Walter, X. Granier, and P. Poulin. Interactive virtual relighting and remodeling of real scenes. In *10th Eurographics Workshop on Rendering*, Granada, June 1999.

[MF01] Steve Mann and James Fung. Videoorbits on eye tap devices for deliberately diminished reality or altering the visual perception of rigid planar patches of a real world scene. In *International Symposium on Mixed Reality (ISMR2001)*, March 14-15 2001.

[Mül03a] Stefan Müller. *Vorlesungsskript zur Computergrafik 1+2*, 2003.

[Mül03b] Stefan Müller. *Vorlesungsskript zur Photorealistischen Computergrafik*, 2003.

[NVI04a] NVIDIA Corporation. *Cg Toolkit 1.2 User's Manual*, 2004.

[NVI04b] NVIDIA Corporation. *NVIDIA GPU Programming Guide Version 2.1.0*, 2004.

[OABB85] J. Ogden, E. Adelson, J. Bergen, and P. Burt. Pyramid-based computer graphics. *RCA Engineer*, 30:4–15, 1985.

[Pau02] Dietrich Paulus. *Vorlesungsskript zur Bildverarbeitung*, 2002.

[Shi02] Peter Shirley. *Fundamentals of Computer Graphics*. A K Peters, 2002.

[SP94] Francois Sillion and Claude Puech. *Radiosity and Global Illumination*. Morgan Kaufmann, San Francisco, CA, 1994.

[SWND03] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley, 4. edition, 2003.

[Wat00] Alan H. Watt. *3D computer graphics*. Addison-Wesley, 3. edition, 2000.

[WL00] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 479–488. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[WL02] Li-Yi Wei and Marc Levoy. Order-independent texture synthesis. In *Technical Report TR-2002-01*. Computer Science Department, Stanford University, 2002.

[Wyn02] Chris Wynn. *OpenGL Render-to-Texture*. NVIDIA Corporation, 2002.

[YDMH99] Yizhou Yu, Paul Debevec, Jitendra Malik, and Tim Hawkins. Inverse global illumination: Recovering reflectance models of real scenes from photographs. In Alyn Rockwood, editor, *SIGGRAPH99, Annual Conference Series*, pages 215–224, Los Angeles, 1999. Addison Wesley Longman.