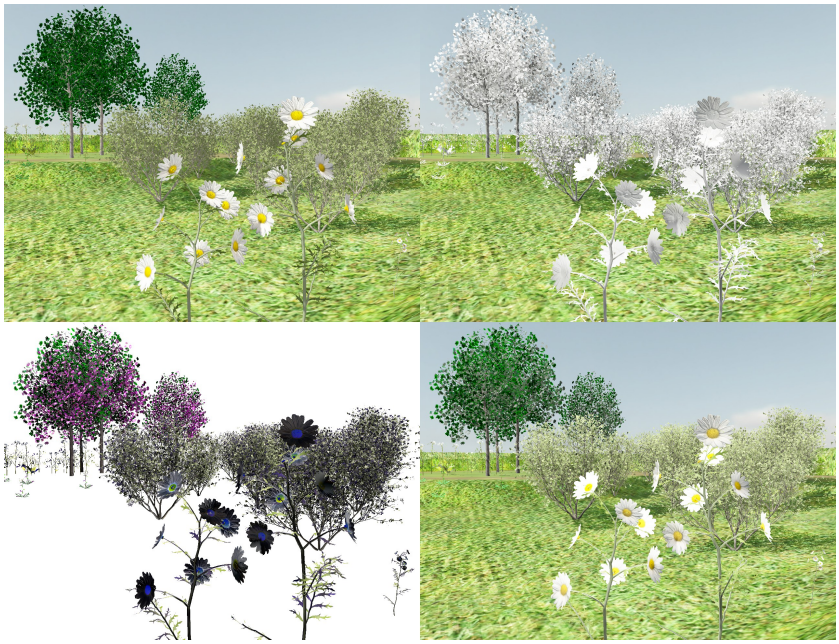
Beleuchtung von Landschaften in interaktiver Darstellung



Malte Clasen

Betreuer:
H. U. Lemke, Prof., Technische Universität Berlin
H.-C. Hege, Hon.-Prof., Zuse-Institut Berlin

28. April 2005

Zusammenfassung

Inhalt dieser Diplomarbeit ist ein Beleuchtungs- und Visualisierungsmodell für Pflanzen im interaktiven Landschaftsrendering. Ziel ist die qualitativ hochwertige Darstellung von einzelnen Individuen nahe des Betrachters, der seine Position in einer ansonsten statischen Szene frei wählen kann.

Um dies zu erreichen wird zunächst vorgestellt, wie ausgehend von einfachen 3D-Modellen und Materialien ein physikalisch basiertes Reflexionsmodell parametrisiert werden kann. Grund hierfür ist der oftmals vorhandene umfangreiche Datenbestand, der auf die OpenGL-Materialien oder ähnliche Shader optimiert ist.

Anschließend wird gezeigt, wie die Echtzeit-Renderingtechniken Shadow Mapping und Precomputed Radiance Transfer kombiniert werden können, um sowohl exakte hochfrequente direkte Beleuchtung als auch niederfrequentes indirektes Streulicht zu berücksichtigen. Diese Kombination wird an Hand der Pfad-Notation des Monte-Carlo-Path-Tracing eingeführt und begründet.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Diplomarbeit beschäftigt sich mit der Darstellung von Landschaften, genauer gesagt mit einer möglichst realistischen Visualisierung der Pflanzen. Zunächst wird ein Überblick über den Kontext gegeben, insbesondere über das Lenné3D-Forschungsprojekt und bisherige Arbeiten. Das zweite Kapitel fasst die wissenschaftlichen Grundlagen, die im folgenden vorausgesetzt werden, zusammen. Das dritte Kapitel geht näher auf die Datensätze ein, die visualisiert werden sollen. Dabei werden insbesondere die Implikationen der einzelnen Dateiformate beleuchtet und geschildert, wie die Inhalte für eine physikalisch basierte Visualisierung genutzt werden können. Diese folgt im vierten Kapitel, das den Kern der Diplomarbeit darstellt. Hier werden bestehende Techniken verglichen, neue vorgestellt und das Zusammenspiel der verschiedenen Algorithmen erläutert. Details zur Implementierung werden im fünften Kapitel gegeben. Das sechste Kapitel stellt die mit der Implementierung erzielten Resultate vor und vergleicht sie mit früheren Ansätzen. Abschließend folgt ein Fazit, das neben einer Zusammenfassung und Bewertung auch einen Ausblick auf weitere Forschungsmöglichkeiten im Bereich Landschafts- und Pflanzenvisualisierung gibt.

1.1 Lenné3D

Das Lenné3D-Forschungsprojekt[1] ist ein von der Deutschen Bundesstiftung Umwelt[2] (DBU) gefördertes institutsübergreifendes Projekt, das interaktive Werkzeuge für virtuell erfahrbare Landschaftsentwicklung entwirft und implementiert. Ziel ist es, Landschaftsplanung für Nicht-Landschaftsplaner (Bürger, Entscheidungsträger) erfassbar zu machen, was über verschiedene

[1] http://www.lenne3d.de
[2] http://www.dbu.de

Abbildung 1.1: Virtuelle Rekonstruktion des verschwundenen Lenné-Gartens (Bildschirmfoto vom interaktiven Lenné3D-Player, GIS- und CAD-datenbasiert, Xfrog-Lenné3D-Pflanzenmodelle, ©Lenné3D 07/2004, Quelle: http://www.lenne3d.de/seiten/proj_kulturstueck.html)

Echtzeit-3D-Darstellungsmodi (fotorealistisch, skizzenhaft; Spaziergänger- und Vogelperspektive) realisiert wird. Unterstützt werden damit nicht nur aktuelle Planungsverfahren sondern auch die Rekonstruktion historischer Landschaften wie z.B. das „italienische Kulturstück“, ein von Peter Joseph Lenné gestalteter Garten im Park des Potsdamer Schlosses Sanssouci (Abb. 1.1)

Das Visualisierungsmodul des Lenné3D-Projektes wird in Zusammenarbeit der Universität Konstanz[3] und des Zuse-Instituts Berlin[4] entwickelt. Die vorliegende Diplomarbeit entstand im Rahmen dieses Projektes am Zuse-Institut in der Abteilung „Visualisierung und Datenanalyse“.

1.2 Problem: Fotorealismus

Neben der skizzenhaften Darstellung ist die fotorealistische Darstellung insbesondere dann gewünscht, wenn man den Interpretationsspielraum minimieren möchte. Sie gibt dem Betrachter ein möglichst exaktes Bild dessen, wie der Entwurf des Planers in der Realität aussehen würde. Dabei stellen bewachsene Landschaften zwei Anforderungen, die sie bei der Visualisierung von den meisten technischen Strukturen unterscheiden:

Zum einen ist sowohl die Anzahl der Pflanzenarten als auch die Anzahl der Individuen meist enorm hoch. Die Szene in Abb. 1.1 zeigt rund 170.000 Individuen aus einer Sammlung von 300 verschiedenen 3D-Modellen. Einige

[3] http://www.cgmi.inf.uni-konstanz.de/
[4] http://www.zib.de

Lösungsansätze zur Bewältigung dieser Datenaufkommen in Echtzeitumgebungen wurden in den letzten Jahren entwickelt. [Deussen et al. 1998] beschreibt Verfahren zur Modellierung der Pflanzen und zur Positionierung der Individuen, die Visualisierung ist allerdings noch nicht interaktiv. [Deussen et al. 2002] und [Coconu und Hege 2002] bilden die Grundlage des heutigen Lenné3D-Players, der in der Lage ist, die genannte Szene auf aktuellen PCs interaktiv zu präsentieren. Der Schwerpunkt liegt hierbei auf effizienten Level-of-Detail-Strukturen, die die Geometriedaten für weiter entfernte Individuen vereinfachen und so dafür sorgen, dass die Pflanzen nur so detailliert gerendert werden, wie sie für den Betrachter noch wahrnehmbar sind.

Die andere Anforderung ist die realistische Darstellung einzelner Pflanzen. In [Franzke und Deussen 2003] wird ausführlich beschrieben, welche Parameter eines Blatts erfasst werden müssen, um eine gute Approximation an das natürliche optische Verhalten zu erzielen. Dabei werden die Blätter in vier Schichten mit jeweils unterschiedlichen Eigenschaften aufgeteilt und ein Algorithmus angegeben, wie man aufbauend auf realen Messwerten mit diesem Modell sehr hochwertige Visualisierungen erzeugt, allerdings nicht in Echtzeit.

Problematisch ist nun, dass beide Aspekte für eine überzeugende Darstellung wichtig sind. Man benötigt viele Pflanzen, die aber auch in sich jeweils wieder eine hohe Komplexität aufweisen. In dieser Diplomarbeit wird nun eine Möglichkeit vorgestellt, beiden Anforderungen gerecht zu werden.

1.3 Lösung: Beleuchtung

Der Lösungsansatz ist die Erweiterung des Level-of-Detail-Prinzips, allerdings in die andere Richtung: Anstatt ausgehend von einem vorgegebenen Modell Vereinfachungen zu errechnen, die gerendert werden, wenn die Pflanze weit entfernt vom Betrachtet steht, wird vorgestellt, wie man ausgehend von denselben Daten ein physikalisch plausibles Verhalten approximieren kann, das dann zum Einsatz kommt, wenn es sich um ein Individuum im Nahbereich handelt. Die Betonung liegt dabei auf der Physik, da übliche Renderer mit Techniken arbeiten, die zwar sehr schnell berechnet werden können, aber leider elementare Gesetze der Physik missachten wie z.B. die Energieerhaltung ([Lewis 1993]).

Kapitel 2

Grundlagen

In den folgenden Kapiteln werden Grundkenntnisse der Computergrafik vorausgesetzt, wie sie beispielsweise in [Watt 1999] vermittelt werden. Auch eine Übersicht über die Fähigkeiten aktueller 3D-Hardware im Rahmen der OpenGL-Schnittstelle ist hilfreich ([Blythe et al. 1999]).

In diesem Kapitel werden die speziellen Teilgebiete der Computergrafik und Optik kurz wiederholt, die für die das vorgestellte Verfahren notwendig sind. Für detaillierte Informationen und Einführungen in die Themen wird auf die jeweilige Literatur verwiesen.

Bei der Notation wird meist auf den Punkt („.“) statt des Kommas („,“) als Dezimaltrennzeichen zurückgegriffen, da sowohl englischsprachige Literatur als auch Programmiersprachen damit arbeiten. Da kein Tausender-Trennzeichen verwendet wird, ergeben sich durch diese Wahl keine Mehrdeutigkeiten.

2.1 Physik

Die physikalischen Eigenschaften des Lichts lassen sich über verschiedene Modelle erfassen. Das älteste und einfachste Modell sind Lichtstrahlen. Hierbei wird eine rein geometrische Abstraktion verwendet: Ein Lichtstrahl hat einen Startpunkt x und eine Richtung $\vec{\omega}$. Dieses Modell ist ausreichend, um einfache Effekte wie Reflexion und Brechung zu beschreiben. Interferenz, Beugung und Polarisation bedürfen einer Erweiterung dieses Modells um Welleneigenschaften, wohingegen Fluoreszenz und Phosphoreszenz nur quantenphysikalisch erklärt werden können.

Obgleich Chlorophyll deutliche Floureszenzeigenschaften aufweist, kann dieser Effekt in den meisten Beleuchtungsszenarien vernachlässigt werden, weshalb wir im folgenden meist vom Strahlenmodell ausgehen. Zur Einführung der Terminologie (analog zu [Jensen 2001]) wird jedoch auch

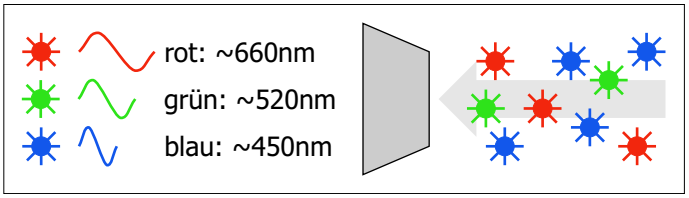


Abbildung 2.1: Die von der Strahlungsmenge bezeichnete Energie ist die gesamte Energie, die auf einem bestimmten Bereich auftrifft. Ein Beispiel wäre die Energie des Sonnenlichts, das innerhalb eines Jahres die Erdoberfläche erreicht.

auf Wellen und Photonen zurückgegriffen. Zu beachten ist, dass neben den deutschen Begriffen auch die englischen Bezeichner eingeführt und hauptsächlich verwendet werden, um den Übergang zu weiterführender Literatur zu erleichtern.

2.1.1 Radiometrie

Radiometrie beschreibt die Messung der Lichts. Grundlegende Messgröße ist dabei die *Energie* e_λ eines einzelnen Photons:

$$e_\lambda = \frac{hc}{\lambda} \tag{2.1}$$

Dabei ist λ die Wellenlänge des Photons, h das Plancksche Wirkungsquantum ($h \approx 6,626 \cdot 10^{-34} Js$) und c die Lichtgeschwindigkeit (im Vakuum: $c = c_0 \approx 3 \cdot 10^8 \frac{m}{s}$). Die Einheit der Energie ist Joule (J). Die *Strahlungsmenge* (engl. radiant energy) ist die auf einem bestimmten Bereich auftreffende Energie (Abb. 2.1):

$$Q = \int_0^\infty n_\lambda e_\lambda d\lambda \tag{2.2}$$

n_λ ist die Anzahl Photonen der Wellenlänge λ. Auch hier ist die Einheit Joule, da nur Energie summiert wird. Der *Strahlungsfluss* (engl. radiant flux) bezeichnet die Strahlungsmenge pro Zeiteinheit:

$$\Phi = \frac{dQ}{dt} \tag{2.3}$$

Damit ist die Einheit Joule pro Sekunde, also Watt (W). Die *Bestrahlungsstärke* bezeichnet nun den Strahlungsfluss, der pro Fläche einfällt (engl. irradiance) bzw. ausgeht (engl. radiosity) (Abb. 2.3):

$$E(x) = \frac{d\Phi}{dA} \tag{2.4}$$

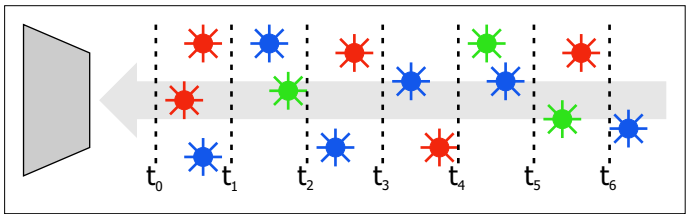


Abbildung 2.2: Der Strahlungsfluss gibt das Verhältnis der Energie zur Zeit an, also z.B. die auf der Erde eingetroffene Sonnenenergie pro Stunde. Integriert man über die Zeit, so ergibt sich die Strahlungsmenge.

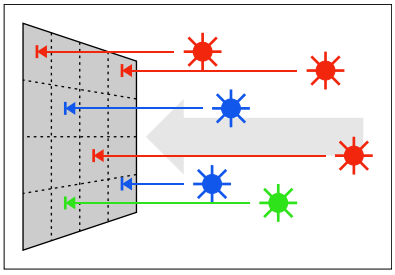


Abbildung 2.3: Die Bestrahlungsstärke gibt an, wieviel Leistung (Energie/Zeit) pro Flächeneinheit anfällt. Für die Sonne wäre dies z.B. die Energie, die pro Stunde und Quadratmeter auf der Erde auftrifft. Integriert man über die gesamte Erdoberfläche, erhält man den Strahlungsfluss.

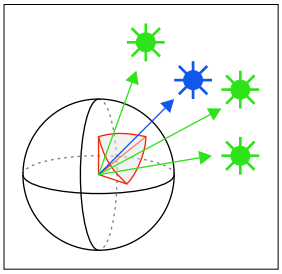


Abbildung 2.4: Die Strahlstärke setzt den Strahlungsfluss ins Verhältnis zum Raumwinkel, also Leistung pro Raumwinkel. Am Beispiel der Sonne wäre dies z.B. die Leistung, die sie in Richtung der Erde abgibt. Die Integration über alle möglichen Richtungen ergibt den Strahlungsfluss.

wobei x der betrachtete Punkt einer Oberfläche A ist. Um die Richtungsabhängigkeit der Strahlung beschreiben zu können benötigen wir zunächst eine Definition aus der Geometrie, den *Raumwinkel*. Der Raumwinkel (Einheit sr) bezeichnet das Verhältnis eines Ausschnitts der Kugeloberfläche zum Quadrat des Radius:

$$\Omega = \frac{S}{r^2} \tag{2.5}$$

Da die Kugeloberfläche vom Quadrat des Radius abhängig ist, ergibt sich für den vollen Raumwinkel eine Konstante von

$$\frac{S}{r^2} = \frac{4\pi r^2}{r^2} = 4\pi \tag{2.6}$$

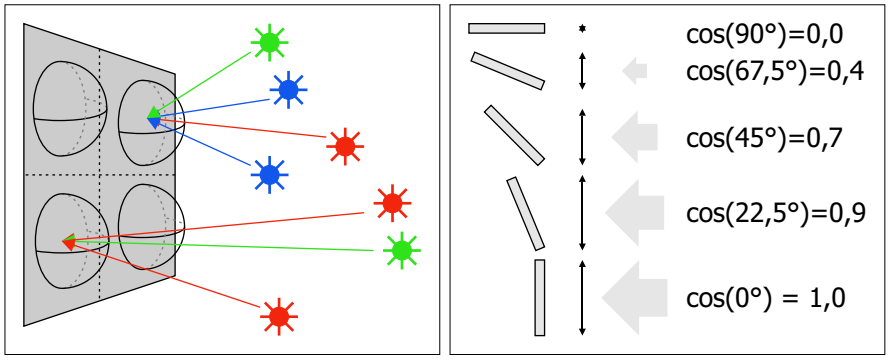
Der Raumwinkel lässt sich analog zum Bogenmaß verwenden. Damit ist es nun möglich, die *Strahlstärke* (engl. radiant intensity) zu definieren (Abb. 2.4):

$$I(\vec{\omega}) = \frac{d\Phi}{d\vec{\omega}} \tag{2.7}$$

Sie beschreibt die Abhängigkeit des Strahlungsflusses von der Raumrichtung. Die für das Rendering wichtigste Größe ist die *Strahldichte* (engl. Radiance), die den Strahlungsfluss pro Fläche und Raumwinkel angibt (Abb. 2.5):

$$L(x, \vec{\omega}) = \frac{d\Phi}{\cos\theta dA d\vec{\omega}} \tag{2.8}$$

Dabei gibt θ den Winkel zwischen der Strahlrichtung $\vec{\omega}$ und der Oberflächennormalen $\vec{n}$ im Punkt x an. Durch diesen Faktor wird berücksichtigt, dass x auf einer Fläche liegt, die nicht senkrecht zur Strahlrichtung steht. Projiziert man die Fläche auf eine Ebene, deren Normalenvektor $\vec{\omega}$ ist, so verkleinert sie sich um den Faktor $\cos\theta$. Für x, die nicht im Kontext



(a) (b)

Abbildung 2.5: Die Strahldichte gibt die Leistung an einem bestimmten Punkt für eine bestimmte Richtung an. Die wahrgenommene Helligkeit hängt direkt mit der Strahldichte zusammen, weshalb sie für Anwendungen in der Computergrafik die wichtigste radiometrische Größe ist. Zu beachten ist, dass x auf einer Fläche liegen kann, deren Normalenvektor ungleich $\vec{\omega}$ ist. Die Projektion der Fläche auf die Ebene, deren Normalenvektor $\vec{\omega}$ ist, wird über $\cos\theta$ berücksichtigt.

deutsch	*englisch*	*Symbol*	*Einheit*
Strahlungsmenge	radiant energy	Q	J
Strahlungsfluss	radiant flux	Φ	W
Bestrahlungsstärke	irradiance	$E(x)$	Wm^{-2}
Bestrahlungsstärke	radiosity	$E(x)$	Wm^{-2}
Strahlstärke	radiant intensity	$I(\vec{\omega})$	Wsr^{-1}
Strahldichte	radiance	$L(x, \vec{\omega})$	$Wm^{-2}sr^{-1}$

Tabelle 2.1: Radiometrische Größen

einer Oberfläche stehen, kann $\cos\theta = 1$ gesetzt werden. Die photometrische Entsprechung der Strahldichte ist die *Leuchtdichte* (engl. luminance), die die wahrgenommene Helligkeit angibt. Neben dieser ist die hier wichtigste Eigenschaft der Strahldichte, dass sie im Vakuum entlang des Strahls konstant ist. Darauf beruht praktisch jeder Raytracing-Algorithmus, da so nur die Schnitte der Strahlen mit Objekt-Oberflächen berechnet werden müssen.

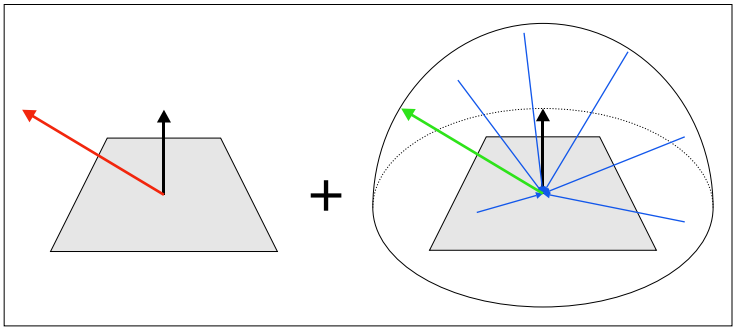


Abbildung 2.6: Die Rendering Equation gibt die ausgehende Radiance als Summe der Emission (links) und der Integration über die reflektierte eingehende Radiance (rechts) an.

2.2 Rendering Equation

Die *Rendering Equation* (eingeführt in [Kajiya 1986]) ist eine Gleichung, mit der man für einen Oberflächenpunkt x die ausgehende Radiance in jeder Richtung beschreiben kann. Sie setzt sich zusammen aus der Summe der Emission L_e in x in Richtung $\vec{\omega}$ und der Lichtreflexion. Letztere ergibt sich aus der eingehenden Radiance in x multipliziert mit der Reflektivität f_r der Oberfläche (Abb. 2.6):

$$L(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_{\Omega} f_r(x, \vec{\omega}', \vec{\omega}) L(x, \vec{\omega}')(\vec{n} \cdot \vec{\omega}') d\vec{\omega}' \tag{2.9}$$

Die Reflektivität f_r wird durch eine *Bidirectional Reflectance Distribution Function* (BRDF) angegeben, die wiedergibt, wie Licht, dass in x aus Richtung $\vec{\omega}'$ in Richtung $\vec{\omega}$ reflektiert wird, abgeschwächt wird. Bei der Wahl einer BRDF[1] muss berücksichtigt werden, dass zum einen nicht mehr Energie reflektiert werden kann als einfällt (Energieerhaltung), zum anderen dass die Licht-Richtung umkehrbar ist (Helmholtz-Reziprozität), d.h. $f_r(x, \vec{\omega}', \vec{\omega}) = f_r(x, \vec{\omega}, \vec{\omega}')$. Letztere wird von den meisten Raytracing-Algorithmen direkt genutzt, da „Sehstrahlen“ so denselben Weg zur Lichtquelle nehmen wie Lichtstrahlen zum Auge hin. Bei der Integration über die Hemisphäre Ω im Punkt x in Normalenrichtung $\vec{n}$ der Oberfläche muss berücksichtigt werden, dass der volle Raumwinkel $4\pi sr$ entspricht. Mit dem Projektionsfaktor $\vec{n} \cdot \vec{\omega}'$ (entspricht dem $\cos\theta$ der Radiance-Definition) ergibt sich:

$$\int_{\Omega} \vec{n} \cdot \vec{\omega}' d\vec{\omega}' = \frac{2\pi}{2} = \pi \tag{2.10}$$

[1] Physikalisch gesehen wird die BRDF eindeutig durch das gewünschte Material festgelegt, eine Wahl im engeren Sinne besteht nur aus Sicht der physikalisch ungenauen Computergraphik.

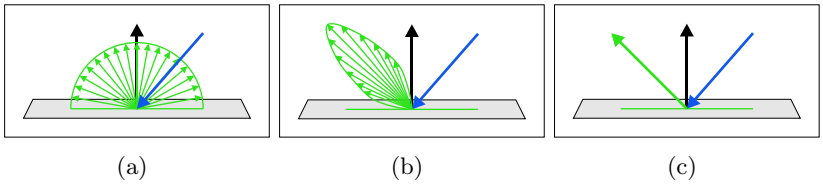


Abbildung 2.7: Drei gängige BRDF: (a) diffus: Eingehendes Licht wird gleichmäßig in alle Richtungen reflektiert. (b) glossy: Die Reflexion erfolgt in eine Richtung, streut aber um diese herum. (c) specular: Licht wird streng nach dem Reflexionsgesetz reflektiert.

Daraus lässt sich die perfekt diffuse BRDF ableiten, d.h. die Funktion, die eingehende Strahlung in alle Richtungen gleichmäßig verteilt (Abb. 2.7):

$$
\begin{aligned}
f_r &\equiv \frac{1}{\pi} && (2.11)\\
\Rightarrow \int_\Omega f_r(x,\vec{\omega}',\vec{\omega})(\vec{n}\cdot\vec{\omega}')d\vec{\omega}' &= \int_\Omega \frac{1}{\pi}(\vec{n}\cdot\vec{\omega}')d\vec{\omega}' && (2.12)\\
&= \frac{1}{\pi}\int_\Omega (\vec{n}\cdot\vec{\omega}')d\vec{\omega}' && (2.13)\\
&= \frac{1}{\pi}\pi && (2.14)\\
&= 1 && (2.15)
\end{aligned}
$$

Mit Hilfe der Beschreibung der lokalen Reflexionseigenschaften der Szene durch die Rendering Equation lässt sich das globale Verhalten ableiten: Die eingehende Radiance $L(x,\vec{\omega}')$ in x ist die ausgehende Radiance $L(x',\vec{\omega}')$ eines anderen Punktes x' der Szene. Dieser Zusammenhang wird deutlich, wenn man nicht über die Hemisphäre Ω über x, sondern über die Oberfläche der Szene A integriert (Abb. 2.8):

$$
L(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_A f_r(x,\omega(x',x),\vec{\omega})L(x,\omega(x',x))G(x',x)V(x',x)dx' \tag{2.16}
$$

$V(x',x)$ berücksichtigt die Sichtbarkeit: Wenn die Strecke zwischen x und x' keinen anderen Punkt der Szene schneidet, ist $V = 1$, ansonsten 0. $G(x',x)$ beschreibt das geometrische Verhältnis der von x zu x' (Abb. 2.9): Bei der Definition der Radiance wurde über $\cos\theta$ die Abweichung des Normalenvektors $\vec{n}$ in x von der Strahlrichtung $\vec{\omega}$ errechnet. Dieser Term wird nun in G berücksichtigt. Dasselbe gilt für x': Auch der Winkel θ' zwischen der Normalen $\vec{n}'$ und $\vec{\omega}$ spielt eine Rolle. Betrachtet man nun ausgehend von x die differentielle Fläche in x', so errechnet sich der Raumwinkel über

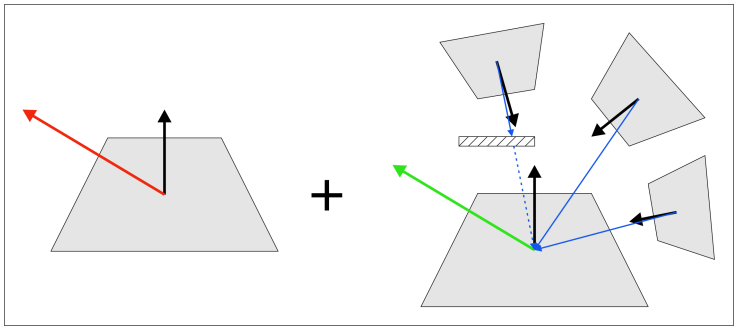


Abbildung 2.8: Bei der Rendering Equation für Oberflächen wird nicht über die Hemisphäre um x integriert, sondern über sämtliche Oberflächen der Szene. Dabei wird das geometrische Verhältnis der Flächen sowie die Sichtbarkeit berücksichtigt.

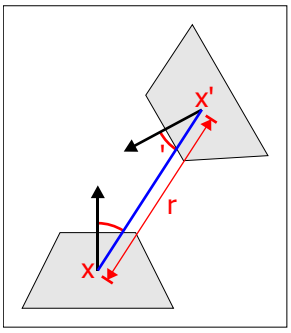


Abbildung 2.9: $G(x', x)$ gibt an, wie zwei Punkte x und x' zueinander stehen. Dabei werden die Abweichungen der Normalenvektoren von der Strahlrichtung und der Abstand berücksichtigt.

die projizierte Fläche geteilt durch das Quadrat des Abstands. Anschaulich bedeutet das, dass die Radiance quadratisch mit dem Abstand abnimmt, da der gleiche Fluß über eine quadratisch wachsende Fläche verteilt wird. Damit ergibt sich für G:

$$G(x', x) = \frac{\cos\theta \cos\theta'}{|x' - x|^2} \tag{2.17}$$

Um nun die Radiance $L(x, \vec{\omega})$ zu errechnen, gibt es verschiedene Wege. Problematisch ist die Abhängigkeitskette der Integrale: Um $L(x, \vec{\omega})$ zu errechnen, benötigt man $L(x', \vec{\omega}')$ für alle x' der Szene, für die $V(x', x) \neq 0$. Für jedes x' werden wiederum alle anderen x'' betrachtet, so dass eine analytische Lösung praktisch ausscheidet. Ausgehend vom Integrationsproblem lassen sich zwei Lösungsstrategien verfolgen: Zum einen die Integration über finite Elemente, zum anderen stochastische Verfahren. Der finite-

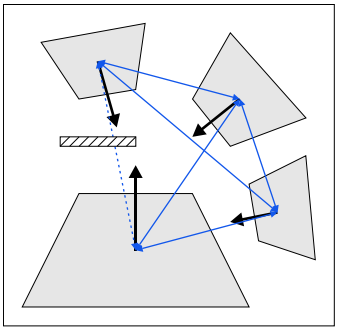


Abbildung 2.10: Beim Radiosity-Algorithmus wird das Energie-Gleichgewicht zwischen allen diskreten Oberflächenelementen p der Szene durch ein lineares Gleichungssystem errechnet. Dabei wird zu jedem p der Energieaustausch mit jedem anderen p' angegeben. Radiosity ist betrachterunabhängig, d.h. die Lösung zu einer Szene gilt für jede Blickrichtung.

Elemente-Ansatz diskretisiert die Szene in sogenannte Patches, für die eine konstante Radiance angenommen wird. Da zur Vereinfachung meist auch die Winkelabhängigkeit vernachlässigt wird, nennt man diesen Ansatz oft entsprechend der somit behandelten radiometischen Größe *Radiosity* (Abb. 2.10). Mit diesen Patches lässt sich ein lineares Gleichungssystem formulieren, da man zu jedem Patch angeben kann, wie der Energieaustausch mit den anderen Patches abläuft. Aufgrund der angenommenen Winkelunabhängigkeit kann man jedoch ausschließlich diffuse Oberflächen berechnen, weshalb diese finite-Elemente-Methode heutzutage nur in speziellen Bereichen zum Einsatz kommt. Ein Vergleich verschiedener Radiosity-Techniken findet sich in [Willmott und Heckbert 1997]. Der derzeit häufigere Ansatz ist die stochastische Approximation des Integrals. Ausgangspunkt ist ein einfacher Raytracing-Algorithmus, der ausgehend vom Auge Schnittpunkte der Sehstrahlen mit der Szene errechnet (Abb. 2.11). Hat man einen Punkt x erreicht, so wird nun üblicherweise ein sogenannter Schattenfühler (Strahl in Richtung der Lichtquelle) ausgesandt und geprüft, ob zwischen x und der Lichtquelle weitere Objekte liegen. Wenn nicht, so kann man über die Radiance der Lichtquelle und der BRDF in x die Radiance für den ursprünglichen Sehstrahl bestimmen. Bei diesem Algorithmus ist es möglich, praktisch beliebige BRDF zu verwenden, eine Einschränkung der Winkelabhängigkeit gibt es im Gegensatz zur Radiosity nicht. Allerdings wird indirekte Beleuchtung komplett vernachlässigt: Schatten sind komplett schwarz, was nur durch zusätzlich eingeführte Korrekturterme (Ambient Light) ausgeglichen werden kann. Dieser auf ersten Blick zu Radiosity gegensätzliche Ansatz lässt sich aber leicht um indirekte Beleuch-

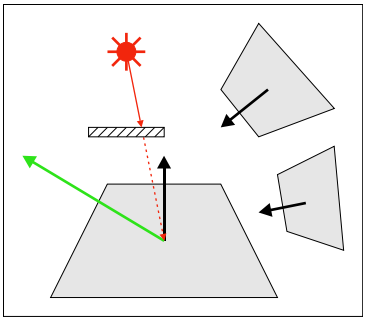


Abbildung 2.11: Raytracing verfolgt die Lichtstrahlen rückwärts ausgehend vom Auge. Dabei wird im einfachsten Fall beim ersten Schnittpunkt mit einem Objekt der Szene nur geprüft, ob von dort aus die Lichtquelle sichtbar ist und die reflektierte Radiance entsprechend berechnet.

tung erweitern: Anstatt nur Schattenfühler auszusenden, kann man alternativ auch weitere „Seh-"Strahlen vom Schnittpunkt x aus in die Szene schicken und die zurückgelieferte Radiance in die Rendering Equation einsetzen. Durch zufällige Verteilung dieser Samples erhält man Distribution Raytracing (Abb. 2.12), das alle Effekte, die durch die Rendering Equation beschrieben werden können, berücksichtigt. Allerdings ist die benötigte Rechenzeit noch sehr ungleich verteilt: Wenn an jedem Schnittpunkt x bis zu einer maximalen Rekursionstiefe t jeweils n Strahlen ausgesandt werden, ergibt sich pro Primärstrahl (also dem Strahl, der von der Kamera ausgeht und dessen Radiance in die gewünschte Pixelfarbe umgerechnet wird) eine Gesamtanzahl von $n^t + 1$ Strahlen. Jede Erhöhung der Rekursionstiefe die benötigte Rechenleistung um das n-fache, trägt aber im Gegenzug meist nur sehr wenig Information zum Endergebnis bei, da nach vielen Reflexionen der Strahlungsfluss durch die Verluste an den Oberflächen (Integral über die BRDF < 1) recht gering ist. *Monte-Carlo-Path-Tracing* (MCPT) fügt eine weitere Zufallskomponente hinzu, um dem entgegen zu wirken: Anstatt in x n weitere Strahlen auszusenden und das Ergebnis mit der BRDF zu verrechnen, wird zufällig entschieden, wie sich ein einzelnes Photon verhalten würde, dass von der Kamera aus in x eintrifft. Möglich sind Absorption und Reflexion, wobei für die Reflexion noch eine Richtung bestimmt werden muss. Eine einfache Möglichkeit ist es, in x eine zufällige Richtung auf der Hemisphäre auszuwählen und zunächst die BRDF zu berechnen. Normalerweise dient sie zur Berechnung der Abschwächung der Radiance. Statt mit ihrem Wert zu multiplizieren, wählt man eine Zufallszahl und prüft, ob sie kleiner als das Ergebnis der BRDF ist. Wenn ja, schickt man genau einen Strahl in die gewählte Richtung und gibt das Ergebnis als Radiance in x zurück (Abb. 2.13). Somit steigt die Anzahl der

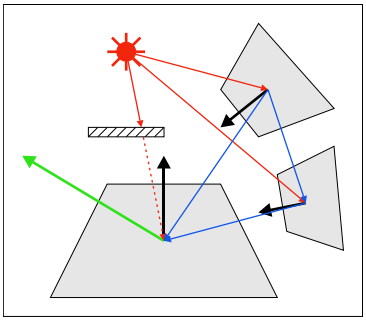


Abbildung 2.12: Distribution Raytracing behandelt im Gegensatz zu einfachem Raytracing auch indirekte Beleuchtung, indem an jedem Schnittpunkt weitere zufällige Strahlen zurückverfolgt werden, deren Radiance auch reflektiert wird.

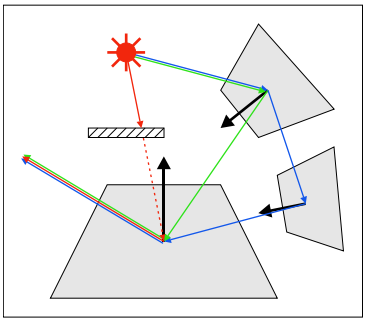


Abbildung 2.13: Im Gegensatz zum Distribution Raytracing werden bei Monte-Carlo-Path-Tracing an den Schnittpunkten nicht mehrere Strahlen verfolgt, sondern nur genau ein zufällig bestimmter. Dies entspricht dem Pfad, der zu einem einzelnen Photon gehören könnte.

Sekundärstrahlen nicht mehr exponentiell an, und nur dort, wo durch geringe Absorption der Strahlungsfluss weitgehend unvermindert reflektiert wird, werden zusätzliche Sekundärstrahlen erzeugt. Die so gewonnene Rechenzeit lässt sich für zusätzliche Primärstrahlen verwenden. Durch das Gesetz der großen Zahlen wird die korrekte BRDF in x dadurch approximiert, dass nicht jeder Strahl abgeschwächt wird und damit nur noch eine Teilmenge der Energie trägt, sondern dass nur noch ein gewisser Prozentsatz der Strahlen mit voller Energie behandelt wird. In der Summe ergibt sich dasselbe Ergebnis, nur dass nun die Gewichtung auf den Primärstrahlen liegt. Neben der Eigenschaft, dass so die Rechenzeit dort verwendet wird, wo wirklich nennenswert Energie übertragen wird, kann man die erhöhte Anzahl Primärstrahlen auch für weitere Zwecke wie Anti-Aliasing einsetzen ([Pharr und Humphreys 2004], Kap. 7). Um die Pfade, die das Licht im

MCPT verfolgt, beschreiben zu können, wird folgende Notation verwendet: Empfänger des Lichts ist das Auge `E`, emittiert wird es von einer Lichtquelle `L`. Auf dem Weg von `L` zu `E` kann es auf verschieden Arten reflektiert werden: *Diffus* (`D`), *glossy* (`G`) und *specular* (`S`). Glossy beschreibt dabei die glänzende Reflexion im Sinne von Hochglanzdrucken, specular die spiegelnde Reflexion wie an Glasscheiben. Diese Unterscheidung ist hilfreich bei der Implementierung, bei der zumindest specular getrennt behandelt werden muss: Da diese Oberflächen streng nach dem Reflexionsgesetz reflektieren, ergibt sich für die BRDF eine Dirac-Stoß-Funktion, d.h. eine Funktion, die nur für eine einzige Richtung ungleich 0 ist. Diese Richtung beim stochastischen Samplen zu erfassen ist praktisch unmöglich, weshalb man bei specular Oberflächen diesen Schritt überspringt und direkt die einzig mögliche Richtung wählt. Die getrennte Notation für diffuse Oberflächen ermöglicht es, auch Radiosity-Pfade zu beschreiben, da hier algorithmisch bedingt nur diffuse Reflexionen auftreten können.

In Anlehnung an reguläre Ausdrücke werden allgemeine Pfade als `E(D|G|S)*L` beschrieben, d.h. zwischen `E` und `L` können beliebig viele (auch 0) Reflexionen beliebigen Typs liegen. Schränkt man auf Pfade mit mindestens einer Reflexion ein ergibt sich `E(D|G|S)+L`. Einfaches Raytracing hat Pfade der Form `E(D|G)L` (genau eine diffuse oder glossy Reflexion), Radiosity entspricht `ED*L` (beliebig viele diffuse Reflexionen).

Weitere Informationen zur Rendering Equation und insbesondere zum Path-Tracing finden sich in [Arvo et al. 2001].

2.3 Sphärische Funktionen

Die Angabe der Radiance erfolgt über einen Punkt x und eine Richtung $\vec{\omega}$. Betrachtet man nur einen bestimmten Punkt der Szene, so lässt sich die Radiance in diesem Punkt auch als $L_x(\vec{\omega})$ schreiben. Der Definitionsbereich von Funktionen dieser Form sind Richtungsvektoren (x, y, z), die in normalisierter Form Punkte auf der Einheitssphäre darstellen, weshalb sie im folgenden *spärische Funktionen* genannt werden (Abb. 2.14). Die Definition einer solchen Funktion kann zum einen algorithmisch erfolgen, so wie die der Radiance $L(x, \vec{\omega})$ in einer Szene über MCPT. Zum anderen kann man aber auch die Sphäre diskretisieren und jedem Element einen Wert zuweisen. Zur Diskretisierung existieren verschiedene Ansätze:

2.3.1 Latitude/Longitude Map

Bei der Latitude/Longitude Map (Lat/Long Map) wird die Sphäre durch zwei Winkel θ und φ parametrisiert, analog zu den Breiten- und Längen-

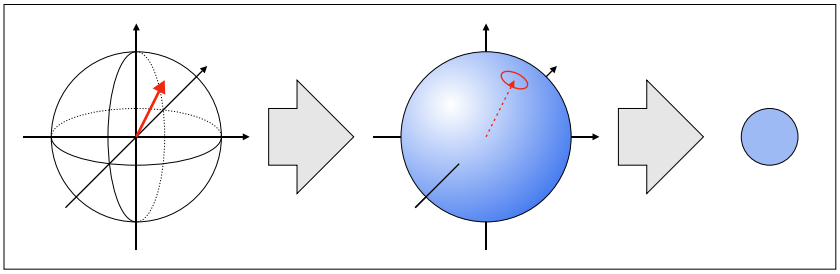


Abbildung 2.14: Sphärische Funktionen bilden Richtungsvektoren auf beliebige Werte ab, in diesem Beispiel auf eine Farbe

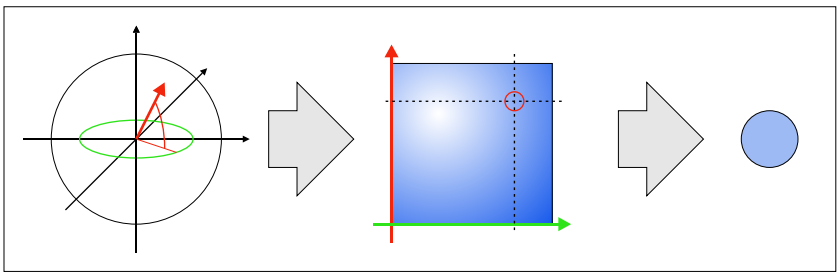


Abbildung 2.15: Eine Latitude/Longitude Map stellt die Sphärenoberfläche als Rechteck dar, dessen (x, y)-Achsen durch die Längen- und Breitengrade (φ, θ) indiziert werden. Das Rechteck wird analog zu einer Textur behandelt, d.h. in Texel aufgeteilt, die die Funktionswerte an ihrer jeweiligen Position angeben.

graden des Globus:

$$(\varphi, \theta) = (arctan\frac{x}{y}, arccosz)$$

Interpretiert man die Winkel als xy-Koordinaten, erhält man ein Rechteck (Abb. 2.15). Die Abbildung Rechteck $\rightarrow$ Sphäre ist surjektiv, jedem Punkt auf der Sphäre wird mindestens ein Punkt des Rechtecks zugewiesen. Das Problem sind die Pole: Für $\theta \in 0, \pi$ wird jeder Punkt des Rechtecks unabhängig von φ auf $(0, 0, 1)$ bzw $(0, 0, -1)$ abgebildet. Da man bei der Auswertung der Funktion an Hand des Punktes auf der Sphäre den Wert im Rechteck sucht, muss man diese Singularitäten speziell behandeln und z.B. $\varphi = 0$ setzen. Das Rechteck wird entsprechend einer Textur diskretisiert. Um in einer Lat/Long Map einen Wert für eine bestimmte Richtung nachschlagen zu können, muss zunächst der Richtungsvektor (x, y, z) in Winkel (φ, θ) umgerechnet werden:

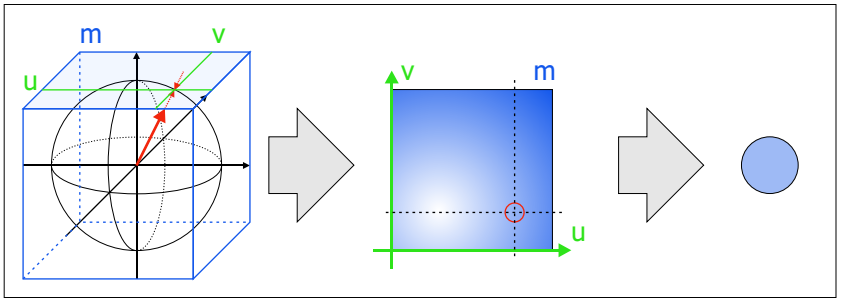


Abbildung 2.16: Cube Maps umgehend das Problem der Singularitäten an den Polen bei Lat/Long Maps durch eine Aufteilung in sechs Würfelseiten statt einem Rechteck. Bei der Auswertung wird der Schnittpunkt des Richtungsvektors mit dem Würfel bestimmt. Die betragsmäßig größte Koordinate gibt die Würfelseite an, die beiden kleineren die Position in der dieser Seite zugeordneten Textur.

2.3.2 Cube Map

Eine Cube Map approximiert die Sphäre (den Definitionsbereich) durch einen Würfel, dessen Seiten jeweils durch eine Textur repräsentiert werden (Abb. 2.16). Dadurch vermeidet man zum einen die Singularitäten der Lat/Long Map, ermöglicht aber zum anderen auch eine einfachere Auswertung. Um einen Punkt von der Sphäre auf den Würfel abzubilden, bestimmt man zunächst das betragsmäßig größte Element m des Richtungsvektors und sein Vorzeichen. Die sechs Möglichkeiten $(x, y, z \times +, -)$ entsprechen den sechs Seiten des Würfels. Die beiden übrigen Elemente u und v werden nun durch m dividiert, wodurch sie im Bereich von -1 bis $+1$ liegen. Nach einer wahlweisen Umrechnung in $[0, 1)$ (z.B. von OpenGL verlangt) lassen sich darüber die Texturen indizieren. Im Gegensatz zur Lat/Long Map hat man also an Stelle zweier trigonometrischer Operationen nur zwei Divisionen, die sich schnell in Hardware realisieren lassen ([Segal und Akeley 2004], 3.8.6).

2.3.3 Cube Wavelets

Analog zur Wavelet-Kodierung in der Bildverarbeitung können auch die Texturen der Cube Map durch Wavelets repräsentiert werden (Abb. 2.17). In [Ng et al. 2003] wird diese Möglichkeit genutzt, um anschließend irrelevante Koeffizienten zu streichen und die Datenmengen für nachfolgende Berechnungen zu reduzieren. Im Vergleich zur einfachen Reduktion der Texturauflösung wird so eine deutlich höhere Qualität erzielt. Zu beachten ist allerdings, dass die Bestimmung einzelner Funktionswerte aus den kodierten Texturen deutlich aufwändiger ist als das Nachschlagen in einer einfachen

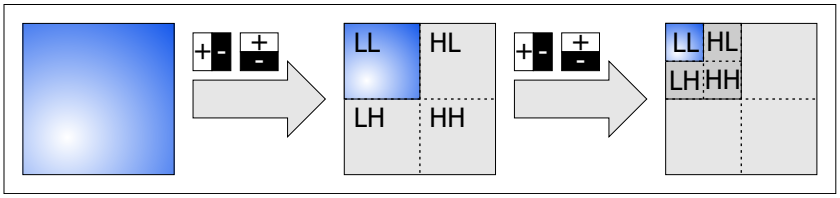


Abbildung 2.17: Cube Wavelets sind Cube Maps, deren Seiten über Wavelets kodiert sind. Die beispielhaft gezeigte Haar-Transformation wendet in jedem Schritt einen vertikalen und horizontalen Hoch- und Tiefpass (H, L) an, wobei das zweimal tiefpassgefilterte Signal aus Ausgang für den nächsten Schritt dient. Diese Umwandlung an sich ist verlustfrei und ändert den Speicherbedarf nicht. Über einen anschließenden Quantisierungsschritt kann aber die Effektivität einer verlustlosen Kompression erhöht werden, wobei die Artefakte deutlich geringer bleiben als bei einer Quantisierung ohne vorherige Transformation.

Cube Map.

2.3.4 Sphärische Wavelets

Während Cube Wavelets die sechs Seiten einer Cube Map getrennt voneinander in Wavelets kodieren, wird in [Schröder und Sweldens 1995] eine Möglichkeit vorgestellt, Wavelets direkt über der Sphärenoberfläche zu definieren. Damit entfällt der Umparametrisierungsschritt, was eine Verbesserung der Approximation ermöglicht. In [Schröder und Sweldens 1995] werden dazu passende Techniken der Bildverarbeitung vorgestellt.

2.3.5 Kugelflächenfunktionen

Kugelflächenfunktionen (engl. Spherical Harmonics, SH) entsprechen der Fourier-Transformation, ähnlich wie sich sphärische Wavelets zu planaren Wavelets verhalten. Es handelt sich um eine Basis zur Zerlegung der spärischen Funktion in verschiedene Frequenzbereiche (Abb. 2.19. Analog zur Fourier-Transformation lassen sich beliebige sphärische Funktionen durch eine Reihe von Koeffizienten darstellen. Durch die Aufteilung in Frequenzbänder reicht es in der praktischen Anwendung oft, nur die Koeffizienten bis zu einer bestimmten Grenzfrequenz zu betrachten, wenn die zu approximierende Funktion bandlimitiert ist oder in den höheren Frequenzen keine relevanten Informationen enthält. [Sillion et al. 1991] verwenden Kugelflächenfunktionen beispielsweise zur Approximation beliebiger BRDF.

Da in Kapitel 4 mit Kugelflächenfunktionen gearbeitet wird, folgt nun eine kurze Übersicht über die mathematischen Grundlagen. Eine detaillier-

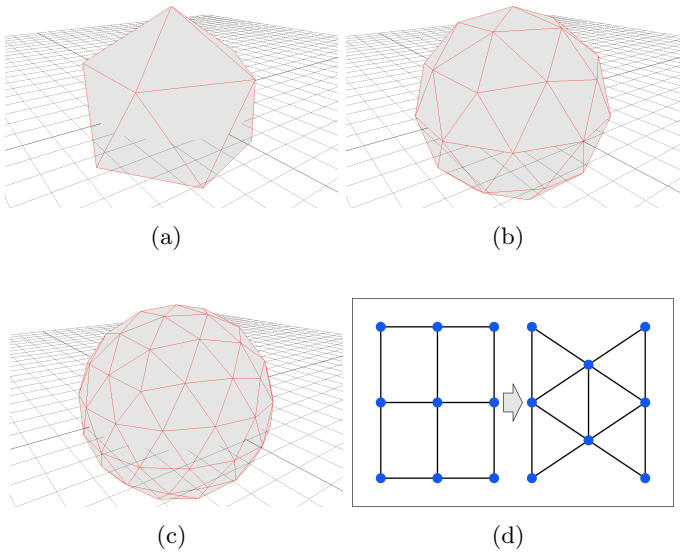


Abbildung 2.18: Während Cube Wavelets für Hoch- und Tiefpass auf die natürliche Nachbarschaft der Texel der Cube-Map-Seiten zurückgreifen können (d, links), wird für sphärische Wavelets eine direkte Diskretisierung der Sphäre verwendet: Durch wiederholte Unterteilung eines Ikosaeders (a,b,c) ergibt sich ein Dreiecksnetz, das die Kugeloberfläche approximiert. Auf diesem Dreiecksnetz kann man nun mit Hilfe der durch die Kanten definierten Vertex-Nachbarschaften (d, rechts) filtern. Da nun die Sphäre nicht durch einen Würfel, sondern beliebig genau approximiert werden kann, werden die Verzerrungen minimiert.

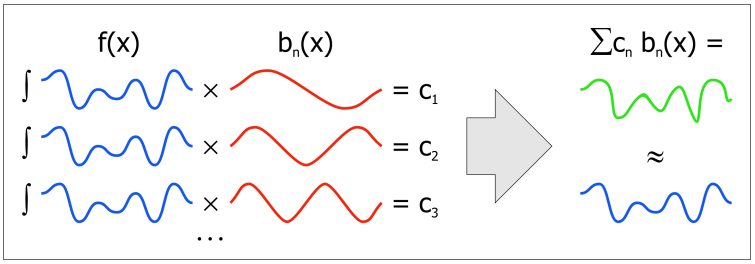


Abbildung 2.19: Um eine Funktion $f(x)$ in Kugelflächenfunktionen darzustellen, integriert man für jede SH-Basisfunktion (hier: $b_n(x)$) über das Produkt aus dieser und f. Summiert man später die so gewonnenen Koeffizienten c_n multipliziert mit den Basisfunktionen, erhält man f zurück. Diese Transformation ist ähnlich wie Wavelets verlustlos, ermöglicht aber eine Kompression durch Quantisierung oder weglassen weniger relevanter Koeffizienten.

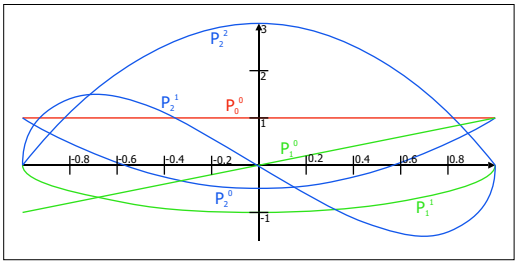


Abbildung 2.20: Legendre Polynome $P(x)$ bilden die Grundlage der SH-Basisfunktionen. Sie sind über dem Intervall $[-1, 1]$ definiert und bilden dort eine orthogonale Basis, d.h. jede Funktion kann durch eine gewichtete Summe von Legende Polynomen dargestellt werden (s.a. 2.19).

te Einführung in das Thema und Implementierungsdetails finden sich in [Green 2003].

Die Parametrisierung der Sphäre entspricht der bei Lat/Long Maps:

$$\begin{aligned}(\varphi, \theta) &= (arctan\frac{x}{y}, arccosz) \\ (x, y, z) &= (\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta)\end{aligned}$$

Analog zu den Basisfunktionen der Fouriertransformation, $\cos(a\varphi)$, werden Kugelflächenfunktionen durch eine orthogonale Basis aus Polynomen definiert. Die sogenannten Legendre-Polynome $P_l^m(x)$ (Abb. 2.20) können induktiv ausgehend von P_0^0 definiert werden:

$$P_0^0(x) = 1 \tag{2.18}$$

$$(l-m)P_l^m(x) = x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m \tag{2.19}$$

$$P_m^m(x) = (-1)^m(2m-1)!!(1-x^2)^{\frac{m}{2}} \tag{2.20}$$

$$P_m^{m+1} = x(2m+1)P_m^m \tag{2.21}$$

Dabei gibt l das Frequenzband an ($l \in \{0, 1, 2, \ldots\}$), m wählt ein Polynom aus diesem Band aus ($m \in \{0, 1, \ldots, l\}$). Die Polynome sind über dem Intervall $[-1, 1]$ definiert.

Mit Hilfe dieser Polynome können nun die SH-Basisfunktionen (Abb. 2.21) für jeden durch (φ, θ) gegebenen Richtungsvektor berechnet werden:

$$y_l^m(\varphi, \theta) = \begin{cases} \sqrt{2}K_l^m\cos(m\varphi)P_l^m(\cos\theta) & m > 0 \\ \sqrt{2}K_l^m\sin(-m\varphi)P_l^{-m}(\cos\theta) & m < 0 \\ K_l^0P_l^0(\cos\theta) & m = 0 \end{cases} \tag{2.22}$$

K ist dabei ein Normalisierungsfaktor:

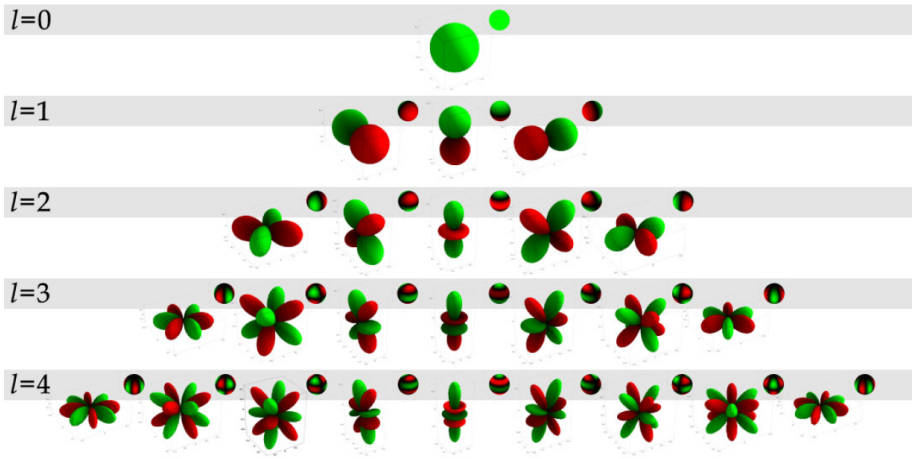


Abbildung 2.21: Die ersten 25 SH-Basisfunktionen nach Frequenzbändern sortiert. (Quelle: [Green 2003])

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}} \tag{2.23}$$

Um nun eine sphärische Funktion in die SH-Basis umzurechnen, kann man die Orthonormalität ausnutzen und über das Produkt der sphärischen Funktion mit der jeweiligen Basisfunktion integrieren:

$$f_l^m = \int_S f(s) y_l^m(s) ds \tag{2.24}$$

Mit Hilfe dieser Koeffizienten f_l^m lässt sich anschließend für jeden Richtungsvektor der Funktionswert ermitteln, indem man die Basisfunktionen entsprechend auswertet und mit den Koeffizienten multipliziert aufsummiert. Für bandlimitierte Funktionen reicht es aus, die Transformation nur bis zur Grenzfrequenz durchzuführen, so dass sich für eine solche Approximation n-ter Ordnung ergibt:

$$\tilde{f}(s) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} f_l^m y_l^m(s) \tag{2.25}$$

bzw.

$$\tilde{f}(s) = \sum_{i=0}^{n^2} f_i y_i(s) \tag{2.26}$$

mit $i = l(l+1)+m+1$. Zwei hilfreiche Eigenschaften im Kontext der Computergrafik sind die Rotationsinvarianz und die Integrationsmöglichkeiten.

Die Rotationsinvarianz besagt, dass eine Approximation n-ter Ordnung beliebig rotiert werden kann, ohne dass sich die Ordnung ändert:

$$\tilde{g}(s) = \tilde{f}(Q(s)) \tag{2.27}$$

Kann man also die gewünschte sphärische Funktion mit n^2 Koeffizienten approximieren, so lässt sich jede beliebige Rotation Q der Funktion durch eine Umrechnung der Koeffizienten darstellen, ohne dass der Approximationsfehler vergrößert wird (abgesehen von numerischen Abweichungen).

Will man über das Produkt zweier SH-Approximationen integrieren, lässt sich dies in eine Summe umschreiben:

$$\int \tilde{a}(s)\tilde{b}(s)ds = \sum_{i=1}^{n^2} a_i b_i \tag{2.28}$$

Diese Anwendung wird dann interessant, wenn man die Rendering Equation erneut betrachtet, Abschnitt 4.2.5 geht näher darauf ein.

Kapitel 3

Daten

Da das Ziel des Lenné3D-Projekts zunächst die Darstellung vieler Pflanzen war und die exakte Darstellung einzelner Individuen im Vordergrund geringere Priorität hatte, waren die Datenformate zu Beginn dieser Diplomarbeit schon dementsprechend festgelegt. In diesem Kapitel werden die vorhandenen Datenstrukturen vorgestellt und beschrieben, wie sie sich so umwandeln lassen, dass eine physikalisch basierte Simulation der Beleuchtung (Kapitel 4) möglich wird.

3.1 Pflanzen

Alle Pflanzenmodelle werden einzeln mit Hilfe von XFrog[1] (basierend auf [Lintermann und Deussen 1998]) modelliert. Zu jeder benötigten Art werden mehrere Modelle erstellt, um den Eindruck der natürlichen Vielfalt zu wahren. Jedes Modell wird zur Darstellung im Player in das an der Uni Konstanz entwickelte TXF-Format umgewandelt.

3.1.1 Eingabeformat TXF

TXF-Dateien sind an die XML-Syntax angelehnte Dateien, die neben in ASCII kodierten Daten auch binäre Elemente enthalten. Sie werden über den Header `<?txf_version='0.91'?>` gekennzeichnet. Die meisten Daten sind zeilenweise abgelegt, wobei die Zeilenumbrüche durch `0x0A` kodiert werden. Die Daten sind in den im folgenden beschriebenen Blöcken organisiert, die durch `<Name>` bzw. `</Name>` eingeschlossen werden, wobei `Name` den Typ des jeweiligen Blocks angibt:

`<Textures>` enthält Angaben zu den verwendeten Texturen. Jede Datenzeile hat die Form

[1] http://www.greenworks.de

```
<tex_number>0</tex_number><tex_name>beispiel.rgb</tex_name>
```

und definiert eine dateiweit gültige Textur-Id mit dem dazugehörigen Dateinamen der Textur.

`<Colors>` enthält dateiweit gültige Materialdefinitionen der Form

```
<col_number> 0 </col_number> <col_name> Default </col_name>
<diff> 0.70 0.70 0.70 1.00 </diff>
<ambi> 0.10 0.10 0.10 1.00 </ambi>
<spec> 0.00 0.00 0.00 1.00 </spec>
<emis> 0.00 0.00 0.00 1.00 </emis>
<shin> 0.00 </shin>
```

Dabei gibt `col_number` die Material-Id an, die für die nachfolgenden Daten gilt, bis eine andere `col_number` gesetzt wird. Im Gegensatz zu XML handelt es sich also nicht um ein kontextfreies Format, da der Parser Zustände wie `col_number` intern speichern muss. Der Materialname `col_name` ist rein informell und für die weitere Verarbeitung ohne Bedeutung. Die Zahlen in den Zeilen `diff` (diffus), `ambi` (ambient), `spec` (specular) und `emis` (emission) stehen für rot, grün, blau und alpha. Der Skalarwert `shin` (shininess) gibt die Streuung von Glanzlichtern (Abb. 2.7(b)) an.

`<Infos>` enthält Metainformationen zum Modell, die für die Interpretation der restlichen Daten irrelevant sind.

`<Set>` enthält Geometriedaten der Form:

```
<set_number> 0 </set_number>

<header>
  <tc_combo> 0 0 </tc_combo>
  <importance> 1.000000 </importance>
  <set_area> 0.003676 </set_area>
  <translucence> 0.000000 </translucence>
  <p_min> -0.018419 -0.008037 0.000000 </p_min>
  <p_max> 0.002775 0.002403 0.596123 </p_max>
  <p_center> -0.007822 -0.002817 0.298061 </p_center>
</header>

<references>
  <header>
    <number_of_references> 0 </number_of_references>
  </header>
</references>
```

```
<vertex_array>
  <header>
    <number_of_vertices> 240 </number_of_vertices>
  </header>
  <data>
...
  </data>
  <index>
    <number_of_index_values> 1404 </number_of_index_values>
...
  </index>
</vertex_array>

<line_array>
  <header>
    <number_of_line_vertices> 41 </number_of_line_vertices>
  </header>
  <data>
...
  </data>
  <index>
    <number_of_index_values> 1404 </number_of_index_values>
...
  </index>
</line_array>

<point_array>
  <header>
    <number_of_subpoints> 0 </number_of_subpoints>
    <number_of_regpoints> 0 </number_of_regpoints>
    <number_of_suppoints> 0 </number_of_suppoints>
  </header>
  <data>
...
  </data>
</point_array>
```

Zu beachten ist, dass die Einrückungen und Leerzeilen der Lesbarkeit halber hinzugefügt wurden und in TXF-Dateien selbst nicht vorhanden sind. Der einzig relevante Eintrag im obersten `header` ist `tc_combo`, hier werden die Textur-Id und Material-Id angegeben, die auf die Geometrie angewandt werden sollen. Der `references`-Block kann komplett übersprungen werden,

auch er enthält keine wesentlichen Informationen.

`vertex_array` enthält eine Liste von Dreiecken, aufgeteilt in Vertices und Indices, wobei `number_of_vertices` und `number_of_index_values` die jeweiligen Anzahlen angibt. An den durch „. . . “ gekennzeichneten Stellen stehen die Binärdaten. Vertices werden entsprechend der C-Struktur

```
struct Vertex {
float texturCoordinate[3];
float normalVector[3];
float position[3];
};
```

abgelegt, wobei `float` für 32-Bit IEEE754 Floats steht, so wie sie von x86-kompatiblen (little endian-) Prozessoren im Speicher gehalten werden. Der dritte Wert der Texturkoordinaten ist mit 0 anzunehmen, da generell nur 2D-Texturen verwendet werden. Die Normalenvektoren werden erst bei der Umwandlung der XFrog-Modelle in TXF automatisch erzeugt, berücksichtigen also nicht den ursprünglichen Oberflächenverlauf bei der Modellierung. Die Indices werden in Dreiergruppen entsprechend der C-Struktur

```
struct Vertex {
unsigned int index[3];
};
```

gespeichert, wobei `unsigned int` für vorzeichenlose 32-Bit little-endian-Zahlen steht. Jede Index-Gruppe definiert ein Dreieck. Obgleich nur Dreiecke verwendet werden gibt `number_of_index_values` die Anzahl Indizes an, also das dreifache der Anzahl der Indexgruppen. Die resultierenden Dreieckslisten sind keine Mannigfaltigkeiten. Bei geschlossenen Zylindern und ähnlichen Strukturen kann es je nach Modell vorkommen, dass die Texturkoordinaten um das Modell herum aufsteigend sind (z.B. $x \in (0.0, 0.2, 0.4, ...)$), das letzte Dreieck aber den Start-Vertex mit der Koordinate 0.0 wiederverwendet (z.B. $x \in (..., 0.6, 0.8, 0.0)$). In diesem Fall ist für das letzte Dreieck die Texturkoordinate auf $1 + x$ zu korrigieren.

`line_array` entspricht dem Aufbau von `vertex_array` mit dem einzigen Unterschied, dass die Index-Gruppen nur zwei Indizes umfassen und Linien statt Dreiecken definieren. `point_array` enthält nur Vertices, da jeder Punkt nur einen Vertex umfasst. Dabei werden alle Vertices ungetrennt nacheinander abgelegt, d.h. insgesamt `number_of_subpoints` + `number_of_regpoints` + `number_of_suppoints`.

Von den Geometriedaten werden hier nur die Dreieckslisten verwendet. Punkte und Linien dienen der alternativen Darstellung im Rahmen einer Level-of-Detail-Unterteilung wie in [Deussen et al. 2002] beschrieben.

3.1.1.1 Material-Interpretation

Die Materialeigenschaften in TXF-Modellen geben kein Referenz-Beleuchtungsmodell vor. In der Praxis erfolgt die Interpretation der Parameter meist durch die Fixed-Function-Pipeline der verwendeten Grafikhardware, d.h. man übergibt die Werte ohne weitere Umwandlung der 3D-API. Dadurch ist das Ergebnis zwar nicht plattformunabhängig, aber durch die oft hoch optimierten Routinen der 3D-Hardware-Hersteller wird eine sehr schnelle Darstellung ermöglicht.

Hier werden die Material-Parameter entsprechend ihrer Bedeutung in OpenGL interpretiert, d.h. sowohl Materialien als auch Lichtquellen werden nach Reflexionsmodellen unterteilt. Dabei stellt `diff` die Absorptionskoeffizienten für diffuse Beleuchtung dar, `ambi` die für die diffuse Reflexion eines virtuellen Umgebungslichts, `spec` die Absorption bei glossy Reflexion[2] und `emis` die Licht-Emission des Objekts selbst. `shin` kann auf den Phong-Exponenten abgebildet werden. Die einzelnen Werte werden jeweils noch mit der Farbe der Textur multipliziert, wobei der Alpha-Wert der Textur direkt die Transparenz angibt.

Problematisch an dieser Interpretation ist, dass sie jeglicher physikalischer Grundlage entbehrt. [Lafortune und Willems 1994] geht auf die Probleme des Phong-Beleuchtungsmodells näher ein und gibt mögliche Korrekturen an, so dass zumindest elementare Eigenschaften wie die Energie-Erhaltung gewährleistet sind.

3.1.2 Internes Format

Die vorhandene Geometrie gibt zu jedem Vertex neben der Position nur einen approximierten Normalenvektor an. Damit fehlt die für anisotrope Materialien (Abb. 3.1 notwendige Orientierkeit, d.h. Tangentenvektor und Binormale. Diese werden an Hand der Texturkoordinaten nach dem Laden des Modells wie von Paul Baker beschrieben [3] approximiert. Bessere Rekonstruktionen des Normalenvektors und damit der Tangenten und Binormalen werden in [Meyer et al. 2002] hergeleitet.

3.1.2.1 Material

Neben der Vervollständigung der Geometriedaten werden die Materialparameter auf ein anderes Reflexionsmodell (Shader) umgerechnet. In [Lewis 1993] werden mehrere physikalisch korrekte Modelle vorgestellt,

[2] OpenGL bezeichnet glossy als specular, da specular geläufiger ist und korrekte specular Reflexion von OpenGL nicht berücksichtigt wird.

[3] `http://www.paulsprojects.net/tutorials/simplebump/simplebump.html`

Parameter	Typ	Bereich	Bedeutung
Roughness	`float[2]`	$[0, 1]$	0=glatt, 1=rau
Isotropy	`float[2]`	$[0, 1]$	0=anisotrop, 1=isotrop
Diffuse	`float[2]`	$[0, 1]$	Anteil diffuser Reflexion
Glossy	`float[2]`	$[0, 1]$	Anteil glossy Reflexion
Specular	`float[2]`	$[0, 1]$	Anteil specular Reflexion
Height	`float[2]`	$\mathbb{R}$	Amplitude der Bump-Map in $[m]$
Translucence	`float`	$[0, 1]$	0=opak, 1=durchsichtig
Color	`Color[2]`	$[0, 1]$	RGB-Reflektivität
Color-Map	`Texture`	$[0, 1]$	RGB-Reflektivität
Normal-Map	`Texture`	$\vec{n}$	Normalen-Verschiebung

Tabelle 3.1: Schlick-Shader-Parameter. Bis auf „Height“ sind die Parameter vom Typ `[2]` jeweils in untere und obere Reflexionsebene unterteilt. Die Color-Map wird nur mit der Reflektivität der unteren Ebene multipliziert, die obere Ebene hat keine Textur. Diffuse, Glossy und Specular müssen sich zu 1 addieren.

die jedoch nur bedingt die Reflexionseigenschaften von Pflanzen erfassen können. [Schlick 1993] stellt ein Modell (Schlick-Shader, Abb. 3.1) vor, dass mehrschichtige Materialien ermöglicht. Dadurch kann die unterschiedliche Reflexion an der Wachsschicht der Blätter und der oberen Zellschicht wiedergegeben werden (Abb. 3.2), was zu deutlich realistischeren Ergebnissen führt. Die Schlick-Shader vernachlässigen zwar auch das Sub-Surface-Scattering (BSSRDF, [Jensen et al. 2001]), was aber nach [Franzke und Deussen 2003] für Blätter durch einfachere Techniken ersetzt werden kann.

Eine Übersicht über die Materialparameter der verwendeten Schlick-Shader ist in Tabelle 3.1 gegeben. Sie orientieren sich an Schlicks Zwei-Ebenen-Modell, enthalten aber zusätzlich Daten zur Transluzenz, Farbtexturen und Normal Maps.

Durch die Wahl einer Zwei-Ebenen-Reflexion müssen die TXF-Materialparameter bei der Umrechnung auf beide Ebenen verteilt werden. Da in der Praxis aber meist für alle Modelle die Standard-Koeffizienten gesetzt sind und keine artspezifischen Reflexionseigenschaften festgehalten wurden, werden bei der Umrechnung einige Annahmen getroffen, die über einen weiten Bereich realistischeres Verhalten zeigen als die konvertierten Standardparameter.

Prinzipiell reflektiert die Wachsschicht stark glossy, weshalb für diese Ebene eine Roughness von 0.1 angenommen wird. Sie weist kein richtungsspezifisches Verhalten auf, ist also vollständig isotrop (1.0). Glossy

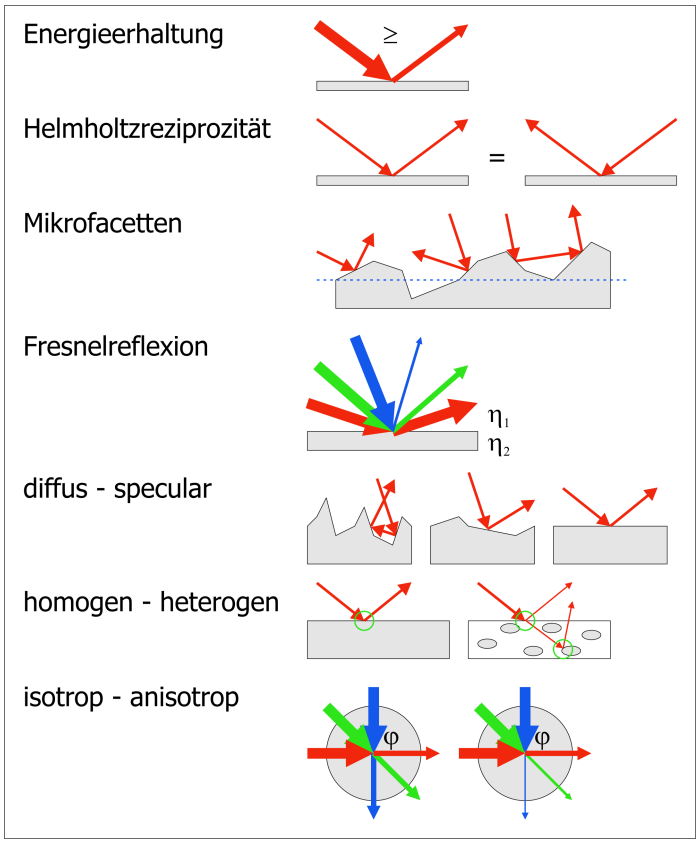


Abbildung 3.1: Dem Schlick-Shader liegen folgende Annahmen zu Grunde: Als *Energieerhaltung* bezeichnet man, dass eine Oberfläche nie mehr Energie reflektiert als auf sie eingestrahlt wird. Die *Helmholtzreziprozität* bezeichnet die Umkehrbarkeit des Lichtwegs. *Microfacetten* erfassen feinste Oberflächenstrukturen, durch die das Licht ungleichmäßig (diffus) reflektiert wird. *Fresnelreflexion* beschreibt die Reflexion an Metallen und Dielektrika in Abhängigkeit der Brechungsindizes. Unter Beachtung dieser Anforderungen ermöglicht Schlicks Reflexionsmodell einen fließenden Übergang zwischen diffusen und specular Materialien, zwischen homogenen (z.B. Metall, Reflexion an nur einer Oberfäche) und heterogenen (z.B. Plastik, Reflexion an transparenter Trägerschicht und opaken Pigmenten) sowie zwischen isotropen und anisotropen (Abhängig vom Winkel zu einem Tangentenvektor, z.B. gebürstetes Metall).

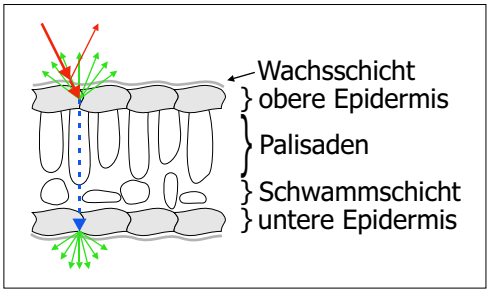


Abbildung 3.2: (Bifaciale) Blätter bestehen aus einer Palisaden- und einer Schwammschicht, die von oberer und unterer Epidermis begrenzt werden. Einfallendes Licht wird zunächst teilweise glossy auf der Wachsschicht über der oberen Epidermis reflektiert. Licht, dass die oberen Zellen erreicht, wird diffus reflektiert. Diese zwei Ebenen entsprechen den heterogenen Materialien des Schlick-Shaders. Das Licht, das das Blatt tiefer durchdringt, wird gestreut und tritt an der Unterseite wieder aus, was durch einen diffusen Term approximiert wird.

wird auf 1.0 gesetzt, dementsprechend Diffuse und Specular auf 0.0. Die Reflektivität wird mit $(0.075, 0.075, 0.075)$ angenommen, d.h. 92.5% des einfallenden Lichts erreichen die zweite Ebene. Diese wird als rein diffus angenommen (Diffuse 1.0, Glossy und Specular 0.0), die Parameter für Roughness und Isotropy werden nicht ausgewertet. Die Reflektivität dieser Schicht wird ausschließlich über die Farbtextur festgelegt, der Farbparameter wird auf $(1.0, 1.0, 1.0)$ gesetzt. Die Amplitude der Bump-Map, d.h. der Höhenverschiebung der Oberfläche, wird auf $\pm 1mm$ gesetzt, die Transluzenz des gesamten Blattes mit 0.3 angenommen. Das Transluzenz-Modell ist ein diffuses Forward-Scattering, d.h. das Licht wird nicht innerhalb des Blattes zur Eingangsseite zurück reflektiert und an der Ausgangsseite gleichmäßig über die Hemisphäre in Normalenrichtung verteilt. Diese an [Franzke und Deussen 2003] angelehnte Vereinfachung ist ein Kompromiss aus exakter Darstellung und Rendergeschwindigkeit, wobei in Ermangelung an Messwerten der Pflanzen auch keine deutlich realistischere Simulation möglich wäre.

Problematisch an dieser Wahl der Parameter ist, dass sie keinerlei biologische Grundlage haben und rein nach einem angenehmen visuellen Erscheinungsbild für wenige Beispielmodelle gewählt wurden. Wären die TXF-Materialparameter pflanzenspezifisch gewählt, so könnte man die Shininess in Roughness bzw. Diffus/Glossy/Specular-Anteile umrechnen und die Farbwerte direkt übernehmen. Wünschenswert wäre es, die Modelle direkt mit einem plausiblen Reflexionsmodell zu entwerfen und so auf einen willkürlichen Konvertierungsschritt verzichten zu können.

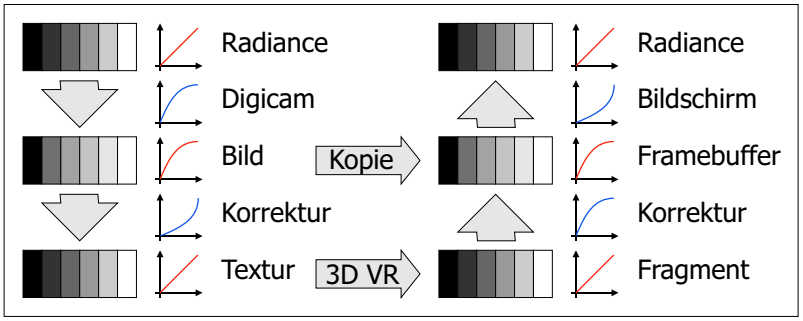


Abbildung 3.3: Die gesamte Kette der Bildverarbeitung von der durch eine Digitalkamera aufgenommenen Radiance bis zur durch den Bildschirm wiedergegebenen Radiance wird von der Nichtlinearität Braunscher Röhren beeinflusst: Die Helligkeit der Bildschirm-Pixel hängt in etwa quadratisch vom Eingabewert ab. Diese Abweichung wird bereits bei der Aufnahme eines Bildes von der Kamera berücksichtigt, so dass dieses ohne weiteres direkt ausgegeben werden kann. Verwendet man das Bild jedoch als Textur in einer 3D-Umgebung, so benötigt man die Reflektivität, nicht das für den Bildschirm angepasste Signal, weshalb ein Linearisierungsschritt notwendig ist. Für das gerenderte Bild hingegen muss die virtuelle Radiance gamma-korrigiert werden, um die Nichtlinearität des Bildschirms auszugleichen.

Um die Farbtextur zur Bestimmung der Reflektivität einsetzen zu können, muss sie zunächst linearisiert werden. Fotos von Digitalkameras und gescannte Analogaufnahmen enthalten meist bereits die für die Darstellung auf Computermonitoren notwendige Gamma-Korrektur von üblicherweise 2.2 (Abb. 3.3). Diese dient ausschließlich zur Kompensation des Ansprechverhaltens von Braunschen Röhren. Da die Texturen aber nicht direkt dargestellt werden sollen sondern nur der Bestimmung der Lichtreflexion dienen, muss die Gamma-Korrektur umgekehrt werden.

In einem weiteren Schritt werden für die Farbtextur die beim Rendering benötigten MipMap-Level errechnet. Dabei wird die Textur stufenweise verkleinert und die einzelnen Stufen zusätzlich der Grafikhardware übergeben. Diese kann nun entsprechend der Texturverkleinerung auf dem Bildschirm die passende Größe wählen. So können das Textur-Aliasing und die benötigte Speicherbandbreite minimiert werden, wobei der Speicherbedarf nur um 33% ansteigt. Für die Farbwerte der Textur können übliche Tiefpassfilter vor der Verkleinerung eingesetzt werden, problematisch sind nur die Alpha-Werte: Scanline-Rasterizer üblicher Grafikhardware kann Transparenzen nur verarbeiten, wenn die Objekte von hinten nach vorn sortiert gerendert werden. Dieser Sortierschritt ist für den gegebenen Anwendungsfall zu zeitaufwendig, weshalb hier nur die Texel gerendert werden, deren Transparenz unter einem gewissen Schwellwert liegt, diese dann aber opak.

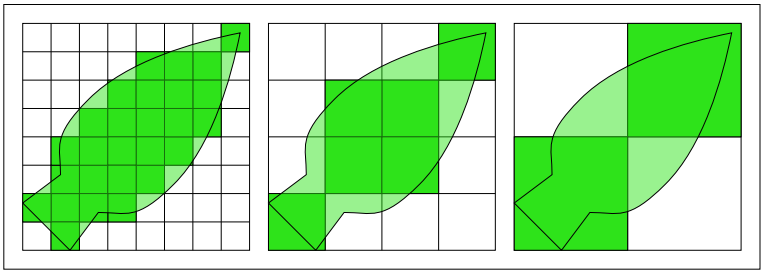
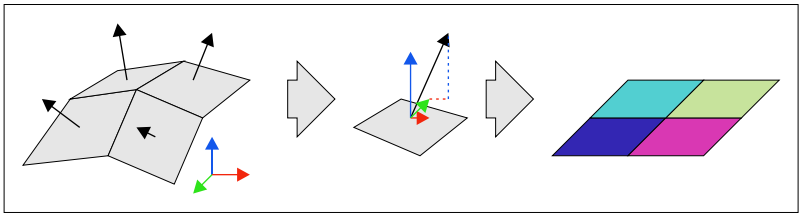


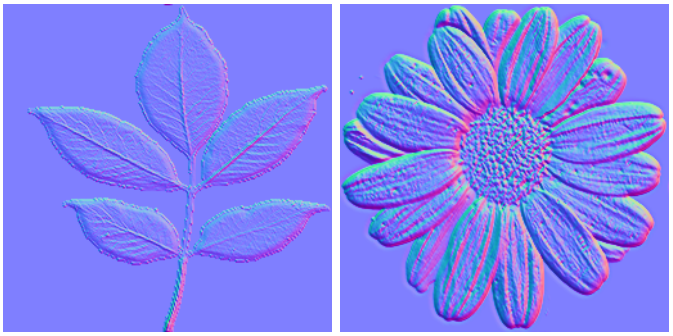
Abbildung 3.4: Bei der MipMap-Erzeugung muss zum einen bei der Filterung der Farbe der Alpha-Wert berücksichtigt werden, damit in diesem Beispiel die Blatt-Pixel ihr grün behalten und nicht mit der transparenten weißen Fläche gemischt werden. Zum anderen muss ein Schwellwert bestimmt werden, ab dem die Texel vollständig opak oder vollständig transparent sind, da Halbtransparenzen eine Polygonsortierung erfordern würden.

Bei der Filterung entstehen nun an den Rändern der in der Textur erfassten Blätter halbtransparente Texel, deren Farbwerte sich zum einen aus den Farben des Blattes, zum anderen aber auch aus den prinzipiell undefinierten Farben der zuvor vollständig transparenten Bereiche zusammensetzen. Bei der Bestimmung der Farbwerte muss also mit dem Alpha-Wert der Texel gewichtet werden. Desweiteren muss sichergestellt werden, dass durch die Wahl des Schwellwerts und der gefilterten Alpha-Werte die Blätter in den kleineren MipMap-Leveln nicht kleiner oder größer werden, als sie es in der ursprünglichen Textur sind. Dies läßt sich durch einen Schwellwert von 0.5 und dem arithmetischen Mittel der Alpha-Werte gewährleisten. Diese Lösung ist zwar nicht optimal, da sie einem Box-Filter entspricht, der hohe Bildfrequenzen unnötig stark absenkt, aber vom visuellen Ergebnis her ist sie ausreichend.

Die Oberflächen der meisten Blätter sind nicht eben. Sie zu modellieren würde aber die Geometrieverarbeitung beim Rendering unnötig beanspruchen, weshalb man Normal Maps einsetzen kann (Abb. 3.5). Normal Maps verändern die Normalenvektoren der Oberfläche ähnlich wie die Farben von Farbtexturen gesteuert werden können. Bei Tangent-Space-Normal-Maps wird angenommen, dass der Normalenvektor der Geometrie immer in Richtung der positiven Z-Achse $(0, 0, 1)$ zeigt. Die Normal Map kodiert dann über die RGB-Kanäle einen neuen Normalenvektor, der statt dessen eingesetzt werden kann: $x, y, z \in [-1, 1]; r, g, b \in [0..1] \Rightarrow (r, g, b) = ((x, y, z) + (1, 1, 1)) \cdot 0.5$. Bedingung an diesen ist, dass er auf dieselbe Seite der Oberfläche zeigt, d.h. das $z > 0$ gilt. Kombiniert mit einem Höhenwert im Alpha-Kanal lassen sich feine Oberflächendetails ohne zusätzliche Geometrie darstellen. Da TXF-Modelle nur Farbtexturen enthalten,



(a)



(b) (c)

Abbildung 3.5: (a) Die Normalenvektoren aus einem hoch aufgelösten 3D-Modell oder einer Bump-Map werden als Farben kodiert und in einer Normal Map gespeichert. (b),(c) Zwei Beispiele mit verschieden stark ausgeprägter Oberflächenstruktur.

müssen diese Oberflächendetails aus der Farbtextur rekonstruiert werden. Die einfachste Möglichkeit dazu ist, den Mittelwert der Farben als Höhe zu interpretieren und den Normalenvektor aus diesen Höhen abzuleiten. Die so erzeugte Normal Map entspricht zwar nicht der wirklichen Oberfläche, orientiert sich aber an ihrer Struktur und ermöglicht so ein visuell befriedigendes Ergebnis. Eine deutliche Verbesserung wäre möglich, wenn die Normal Maps direkt bei Modellierung des Modells korrekt gewählt und mitgegeben würden.

Die bisherigen Annahmen orientierten sich an der Struktur von Blättern. Da im TXF-Format aber nicht gekennzeichnet ist, welche Geometrie und welche Materialien zu Blättern gehören bzw. welche Ästen und Stämmen zugeordnet sind, werden sämtliche Elemente wie Blätter behandelt. Gerade in Bezug auf die Transluzenz ist das offensichtlich falsch, da ein Baumstamm nicht hohl ist und nicht $30\% \cdot 30\%$ des Lichts durchlässt. Im Endergebnis fällt dieser auf dem Papier eklatante Fehler aber nicht deutlich auf, eine wirkliche Bewertung erforderte aber einen Vergleich mit korrekten Materialparametern für alle Elemente einer Pflanze.

3.2 Objekte

Nicht-pflanzliche Objekte werden als reine Dreiecksnetze (ohne Linien oder Punkte) eingebunden. Da sie deutlich seltener in der Szene instanziiert werden, spielt das aber für die Geschwindigkeit keine Rolle.

3.2.1 Eingabeformat 3DS

Das 3DS-Dateiformat wurde von Autodesk[4] für „3D Studio“ entwickelt, dem Modellierungsprogramm, aus dem das heute aktuelle „3DS MAX“[5] entstand. Es gibt keine offene Spezifikation zu 3DS, bedingt durch seine weite Verbreitung Ende der 90er existieren aber einige durch Reverse Engineering entstandene Beschreibungen. Durch die fehlende Spezifikation lässt sich keine abschließende Aussage über die Möglichkeiten von 3DS treffen. Im Lenné3D-Projekt werden ausschließlich die Möglichkeiten genutzt, Dreiecksnetze mit dazugehörigen Texturverweisen zu speichern. Animationen, Kameraeinstellungen etc. werden nicht aus diesem Dateiformat geladen.

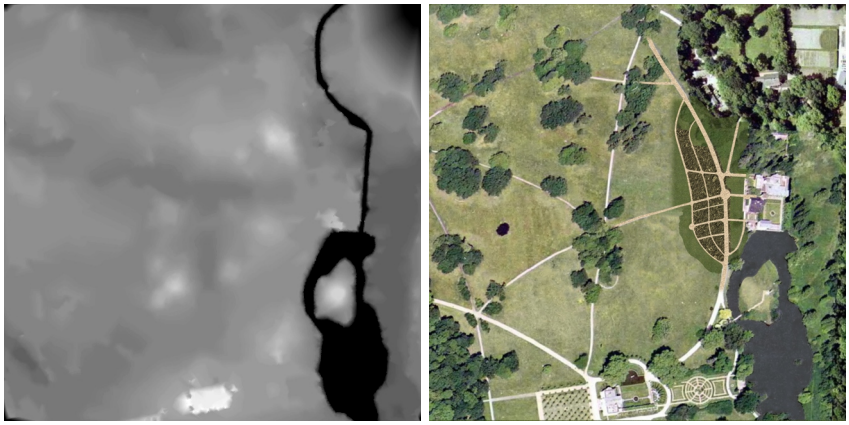
3.2.2 Konvertierung

Der Vollständigkeit halber unterstützen die im Rahmen dieser Diplomarbeit entstandenen Routinen 3DS-Objekte über den Umweg der Konvertierung in „Pflanzen“. Die Dreiecksnetze werden in angepasster Form dem TXF-Interpreter übergeben, weshalb auch alle dort getroffenen Annahmen für die 3DS-Modelle Anwendung finden. Da aber auch 3DS-Materialeigenschaften keine physikalische Basis haben, würden eigens für 3DS-Objekte geschaffene Routinen auch wieder einen Materialkonvertierungsschritt und eigene Annahmen enthalten.

3.3 Terrain

Das Terrain bildet die Grundlage jeder Szene. Pflanzen werden über (x, y)-Koordinaten automatisch auf die jeweilige Geländehöhe gesetzt und der Betrachter bewegt sich in Spaziergängerperspektive in konstanter Höhe relativ zum Boden. Dieses Höhenprofil wird durch eine Height-Map im RAW-Format festgelegt.

[4] http://www.autodesk.com
[5] http://www.discreet.com/



(a) (b)

Abbildung 3.6: Grauwert-kodierte Terrain Height Map und Color Map des italienischen Kulturstücks im Park des Schloss Sanssouci. (Quelle: Lenné3D-Projekt)

3.3.1 Eingabeformat RAW

RAW bezeichnet kein festgelegtes Dateiformat sondern besagt viel mehr, dass kein Header und keine Formatinformationen in der Datei vorhanden sind. Welche Daten letztendlich in einem RAW zu finden sind, bestimmt der Anwendungskontext. Hier handelt es sich um eine Height Map, die in einer in den Szenendaten spezifizierten Auflösung 8-Bit Samples für die Höhenwerte auf einem Rechteck-Gitter enthält (Abb. 3.6(a)). Diese werden zeilenweise abgelegt, so dass eine RAW-Height-Map prinzipiell einem unkomprimierten Graustufen-TGA ohne den 18 Byte großen Header entspricht.

Neben dem Höhenprofil wird auch eine Farbtextur eingelesen (Abb. 3.6(b)). Eine Materialspezifikation liegt nicht vor, weshalb eine rein diffuse Oberfläche angenommen wird.

3.3.2 Internes Format

Das Terrain wird mit Hilfe von Vertex-Texturen gerendert, d.h. die Höhenwerte des Terrains werden von der Grafikhardware zur Laufzeit ausgelesen. Als Vorbereitung für diesen Schritt werden die Samples aus ihrem durch die Szene definierten 8-Bit-Bereich in Meter umgerechnet und in einer Float-Textur gespeichert. Um eine bilineare Filterung mit nur einem Speicherzugriff zu ermöglichen, enthält jeder Texel neben dem eigenen Höhenwert auch die drei rechts bzw. unten liegenden. Damit entspricht ein Texel mit

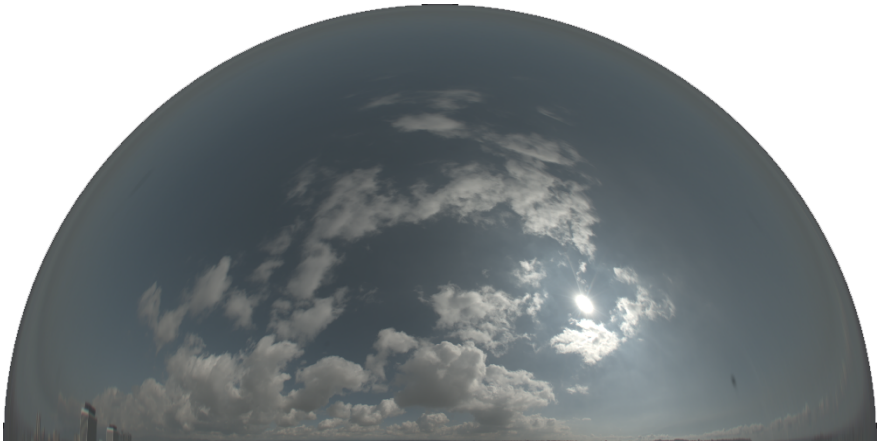


Abbildung 3.7: HDR-Aufnahme eines leicht bewölkten Himmels über dem ICT der University of Southern California am 23.02.2004, 11 Uhr Ortszeit (Quelle: http://www.ict.usc.edu/graphics/skyprobes/)

vier Float-Werten dem RGBA-Format, das von den Vertex-Shadern der NVidia 6x00-Reihe verarbeitet werden kann.

Analog zu den Blatt-Texturen des TXF-Formats wird auch hier eine Normal Map erzeugt, allerdings basierend auf den explizit angegebenen Höhenwerten. Die Farbtextur wird direkt übernommen.

3.4 Himmel

Der Himmel ist bei Lenné3D-Szenen unspezifiziert und besteht meist aus einer Cube Map, die nicht mit der Szene wechselwirkt. Für physikalisch korrekte Beleuchtung ist es jedoch eine Voraussetzung, dass die Sonne nicht die einzige Lichtquelle ist sondern dass auch der Rest des Himmels indirektes Licht beisteuert. Die Idee dazu basiert auf den High Dynamic Range-Aufnahmen (HDR) aus [Debevec und Malik 1997] bzw. den HDR-Environment Maps aus [Debevec 1998]. Hier wird die auf die Szene einwirkende Radiance als Kugelfunktion festgehalten. In [Stumpfel et al. 2004] wird ein Verfahren vorgestellt, um mit einer normalen Digitalkamera und einem Satz von Filtern den gesamten Dynamikumfang des Himmels abzudecken, d.h. etwa ein Kontrastverhältnis von 600000 : 1. Normale 8-Bit-Texturen haben demgegenüber einen Kontrast von 255 : 1. Dadurch ist es möglich, die Szene den Lichtverhältnissen eines realen Himmels auszusetzen. Im folgenden werden die Beispiel-Himmel aus [Stumpfel et al. 2004] verwendet (Abb. 3.7).

Eine Alternative zu aufgenommen Himmeln stellen analytische Model-

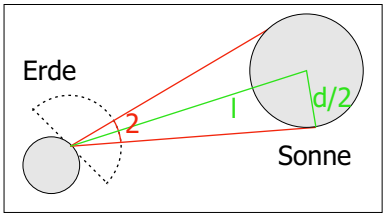


Abbildung 3.8: Der von der Sonne eingeschlossene Winkel kann über Abstand und Radius errechnet werden.

le dar, die realistische Himmel auf Basis einiger Parameter wie z.B. dem Sonnenstand erzeugen können. Verschiedene Möglichkeiten dazu werden in [Preetham et al. 1999] und [Nielsen 2003] beschrieben. Um effektiv Himmel zu generieren müssten die entsprechenden Parameter aber in den Szenendaten festgehalten werden, weshalb vorerst den Aufnahmen der Vorzug gegeben wurde.

3.4.1 Eingabeformat HDR

Die Aufnahmen sind im HDR-Format gegeben, einem von Greg Ward spezifizierten Dateiformat für High Dynamic Range-Bilder. Es ermöglicht eine sehr platzsparende Kodierung, da die einzelnen Pixel durch vier 8-Bit-Koeffizienten, RGBE, repräsentiert werden. Der gemeinsame Exponent E gibt dabei die Skalierung der drei Farb-Mantissen an. So kann ein sehr großer Dynamikumfang bei deutlich geringerem Speicherbedarf im Vergleich zu jeweils drei 32-Bit Floats realisiert werden. Ein Betrachter und Editor[6] für HDR-Bilder wurde von Paul Debevec passend zu [Debevec 1998] entwickelt.

3.4.2 Internes Format

Intern wird der Himmel in Form einer Float-Cube Map gespeichert, um schnell auf die einzelnen Radiance-Werte zugreifen zu können. Da im folgenden die Sonne als stärkste Lichtquelle in Landschaftsszenen getrennt vom restlichen Himmel behandelt werden soll, muss sie in einem Vorverabeitungsschritt identifiziert und heraus getrennt werden. Dazu wird zunächst die erwartete Größe der Sonne berechnet: Bei einem Durchmesser von $d = 1.392.000km$ ist die Sonne $l = 149.600.000km$ von der Erde entfernt (Abb. 3.8). Das bedeutet, dass ihr Radius r von der Erde aus betrachtet folgenden Winkel φ einschließt hat:

[6] http://www.debevec.org/HDRShop/

(a) (b)

Abbildung 3.9: Ein Ausschnitt aus Abb. 3.7, bei dem in (b) die der Sonne zugeordneten Texel rot markiert sind.

$$\varphi = arctan\frac{0.5d}{l} = 0.00465 = 0.2667° \tag{3.1}$$

Aus diesem Radius ergibt sich die Fläche $a = \pi r^2$, die die Sonne auf der Kugel einnimmt. Aufgrund des kleinen r ist die Approximation der Fläche über eine ebene Kreisscheibe akzeptabel. Um daraus die Anzahl der Texel zu bestimmen, die zur Sonne gehören, müssen aber noch zusätzliche Faktoren berücksichtigt werden: Da die Environment Map als Cube Map gespeichert wird, decken nicht alle Texel denselben Raumwinkel ab. Abhängig von der Position der Sonne müsste also die Anzahl Texel variieren. Zusätzlich kommen atmosphärische Streuungen hinzu, die keine scharfe Abgrenzung der Sonnentexel ermöglichen, sowie das sogenannte Blooming des Kamera-CCDs (Lichtsensors), das bei Überbelichtung zu einem Auslaufen der betroffenen Pixel in benachbarte Pixel führt. Experimentell zeigte sich, dass diese Faktoren durch eine Vergrößerung des Sonnenbereichs um 4.0 kompensiert werden konnten, wobei dies bei anderen Aufnahmen neu evaluiert werden müsste. Mit Hilfe dieser Angaben bestimmt man nun die n hellsten Texel $\{T_1, \ldots, T_n\}$ der Cube Map. Der hellste Texel gibt dabei die Richtung der Sonne an. Die Radiance dieser Texel wird zwischengespeichert und durch die des $n+1$-ten Texels (T_{n+1}) ersetzt. Dadurch erhält man eine durchgängige Cube Map des Himmels, in der nur die extrem hellen Sonnensamples fehlen (Abb. 3.9(b)). Anschließend können Irradiance von Sonne und Rest-Himmel berechnet werden. Bei der Sonne ist der Raumwinkel bereits bekannt, aus dem die Radiance-Werte angepasst wurden. Für die Irradiance I_S ergibt sich also:

$$I_S = \frac{\sum_{i=1}^{n} T_i - T_{n+1}}{n} \cdot a \tag{3.2}$$

Die Irradiance des Himmels I_H ergibt sich durch Summation über

die Radiance aller m Texel der Cube Map (einschließlich der ersetzten $\{T_1, \ldots, T_n\}$:

$$I_H = \frac{\sum_{i=1}^{m} T_i}{m} \cdot 2\pi \tag{3.3}$$

Da der Himmel nur für die obere Hemisphäre definiert ist, wird hier auch nur mit dem halben vollen Raumwinkel 2π multipliziert. Abschließend bleibt festzuhalten, dass die Sonnen-Extraktion auch bei bedecktem Himmel keinen Schaden anrichtet: Da nur die Differenz der n hellsten Texel zum nächstdunkleren als Sonne betrachtet wird, erhält man in Abwesenheit dieser nur eine im Vergleich zum Rest-Himmel minimale Irradiance, so dass dieser Fehler vernachlässigt werden kann.

3.5 Szene

Lenné3D-Szenen werden derzeit über das ursprünglich zur Pflanzenverteilung spezifizierte ECO-Format definiert.

3.5.1 Eingabeformat ECO

ECO-Dateien sind zeilenorientierte ASCII-Dateien. Sie lassen sich grob in zwei Bereiche unterteilen: Zuerst wird das Terrain über den Dateinamen der Height Map, dessen Größe in Pixeln und einen Skalierungsfaktor angegeben:

```
beispiel.RAW 1024 1024 1 0.05
```

In diesem Fall wird die Datei „beispiel.RAW“ geladen, für die eine Breite und Höhe von 1024 Pixeln angenommen wird. Die folgende Zahl (1) ist ohne Bedeutung. Die 0.05 gibt den Faktor an, mit die 8-Bit-Werte des Terrains (0..255) multipliziert werden müssen, um die gewünschte Höhe zu erhalten. Da Szenen prinzipiell aus mehreren ECOs bestehen können, folgt als nächstes die Georeferenzierung:

```
<GEOREF> 0 0 200.0 200.0 10.0 110.0 40.0 50.0
```

Hier wird festgelegt, dass das geladene Terrain einen absoluten Ursprung bei $(0.0, 0.0)$ hat und eine räumliche Ausdehnung von 200.0 Metern in x- und y-Richtung. Die nächsten vier Zahlen geben Ursprung und Ausdehnung der sogenannten Area of Interest (AoI) an, also dem Bereich, in dem die nachfolgenden Pflanzen positioniert werden (Abb. 3.10).

Die Zeile `<ECO>` kennzeichnet, dass nun Daten über die Pflanzen der Szene folgen. Über

```
<LAYER> 1
```

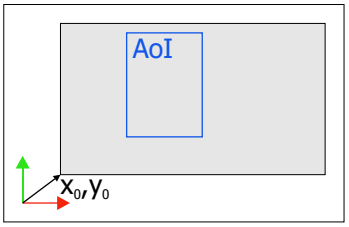


Abbildung 3.10: Über die Georeferenzierung wird die Position des Terrains in Bezug auf ein absolutes Koordinatensystem festgelegt. Innerhalb des Terrains wird eine Area of Interest definiert, in der die nachfolgend aufgeführten Pflanzen platziert werden.

kann die logische Pflanzenebene gewählt werden, um z.B. zwischen Gras und Bäumen unterscheiden zu können. Zeilen der Form

```
<TILING> 0
```

sind derzeit ungenutzt, aber aus Kompatibilitätsgründen noch zulässig. Die einzelnen Pflanzen werden nun über Modellnamen, relative Position innerhalb der Area of Interest, Skalierung und Rotation um die Z-Achse spezifiziert:

```
Anthriscus_sylvestris_001 0.499 0.471 1.3 10.0
```

Die implizite Dateiendung der Pflanzenmodelle ist TXF. Die Rotation wird in ° zwischen 0 und 360 angegeben. Richtlinie ist, dass TXF-Modelle über Koordinaten in Metern definiert werden, weshalb die Skalierung meist im Bereich im 1 liegt, so dass die Modelle in realistischen Größen instanziiert werden.

3.5.2 Internes Format

Die Angabe der Modellnamen der Pflanzen dient der eindeutigen Zuordnung. Intern wird das jeweilige Modell nur einmal geladen und beliebig oft an verschiedenen Stellen in der Szene instanziiert. Dabei können die Instanzen zum einen in Form einer Liste abgelegt werden, in der Position, Skalierung und Rotation explizit aufgeführt werden. Bei größeren Anzahlen ist eine Darstellung über BitTrees (Kap. 5.1.1) sinnvoll, was zukünftig auch als natives Speicherformat der ECO-Dateien dienen soll. Aufgrund der geringen Anzahl der Pflanzen, die ohne Level of Detail gerendert werden können, wird in der Implementierung dieser Diplomarbeit auf einen Szenegraph-Ansatz zurückgegriffen (Abb. 3.11), bei dem die Instanzen durch Transformations-Nodes repräsentiert werden, die jeweils auf den passenden Modell-Node verweisen. Der Speicherbedarf liegt höher als bei den anderen Ansätzen, es vereinfacht aber die Anwendung.

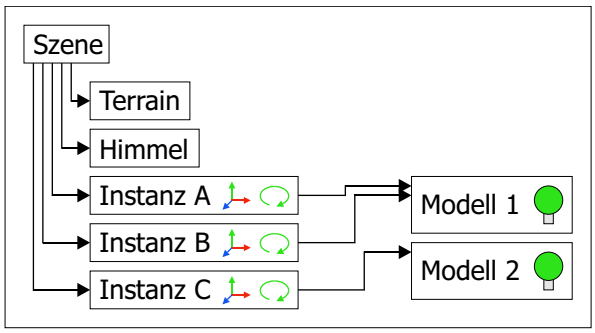


Abbildung 3.11: Der Szenegraph ermöglicht mehreren Instanzen auf dieselben Modelle zuzugreifen. Pro Instanz wird nur die Transformation von Object- in Worldspace (Translation, Rotation, Skalierung) gespeichert sowie ein Verweis auf das zu rendernde Modell, das die eigentlichen Geometrie- und Materialdaten beinhaltet.

Kapitel 4

Rendering

Die Visualisierung der im vorherigen Kapitel beschriebenen Daten ist das Kernthema dieser Diplomarbeit. Dabei wird ausgehend von bestimmten Annahmen erarbeitet, welche bereits existierenden Techniken kombiniert werden können, um eine möglich exakte Darstellung zu ermöglichen.

4.1 Annahmen

Die grundsätzliche Annahme bei der folgenden Betrachtung der verschiedenen Techniken ist, dass nur statische Szenen behandelt werden. Terrain, Pflanzen, Objekte und Himmel ändern weder ihre Position noch ihre Eigenschaften über die Zeit. Der Betrachter soll sich aber frei durch die Szene bewegen können, kann mit dieser jedoch nicht interagieren.

Diese Einschränkung entspricht den Anforderungen des Lenné3D-Projekts: Ziel ist die Visualisierung von realen bzw. realistischen aber statischen Landschaften. Entwurf und Bearbeitung werden über weitere Programme realisiert wie z.B. den LandXplorer[1]. Ein Nachteil der Einschränkung auf statische Szenen ist, dass gerade natürliche Bewegungen (z.B. durch Wind) den gefühlten Realismus deutlich erhöhen. Da die verwendeten Datensätze allerdings keine Spezifikation für Bewegungen bzw. physikalische Interaktion vorsehen, kann dieser Punkt erst in weiteren Entwicklungen berücksichtigt werden.

4.2 Beleuchtungscache

Statische Szenen weisen eine konstante Beleuchtungssituation auf. Da der Betrachter weder mit den Modellen noch mit dem Licht interagiert und so-

[1] http://www.landex.de

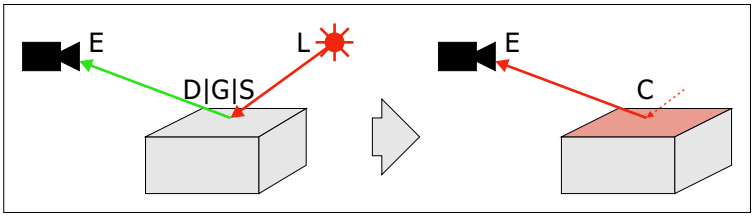


Abbildung 4.1: Der Beleuchtungscache speichert die eingehenden Pfade `C(D|G|S)*L`, so dass beim Rendering die Pfade nur noch bis dort und nicht mehr bis zur Lichtquelle verfolgt werden müssen.

mit auch keine Schatten wirft, ist die Lichtsituation prinzipiell unabhängig von der Ansicht. Damit ist es möglich, Teile der Beleuchtungssituation vor dem eigentlichen Programmstart zu berechnen und bei der Visualisierung auf diesen *Beleuchtungscache* zurückzugreifen.

Dies ist bei Echtzeit-Visualisierungen über die GPU auch erforderlich, sofern man globale Beleuchtungseffekte (`E(D|G|S)+L`) darstellen möchte, da die übliche Rasterisierung ähnlich dem einfachen Raytracing nur den Betrachter, die Lichtquelle und die gerade verarbeitete Oberfläche berücksichtigen kann, mit einigen Tricks auch Schatten. Mehrfachreflexionen oder komplexere Lichtquellen als Punkt-, Spot-, oder Richtungslicht werden derzeit noch nicht direkt unterstützt. Durch die Programmierbarkeit von Vertex- und Fragmentshadern besteht aber die Möglichkeit, diese Effekte durch die im folgenden vorgestellten Beleuchtungscaches zu realisieren.

4.2.1 Prinzip

Das Prinzip des Beleuchtungscache ist an Hand der Pfad-Notation (Kap. 2.2) gut zu verdeutlichen: Allgemeine Pfade entsprechen dem Ausdruck `E(D|G|S)*L`. Von diesen Pfaden ist ausschließlich das `E` variabel, da die Szene bestehend aus Lichtquellen und Oberflächen nicht verändert wird. Im Beleuchtungscache kann also der Zustand des Pfades an einer beliebigen Stelle von `L` ausgehend gespeichert werden.

Beispielsweise könnte generell jede Oberfläche in eine Lichtquelle umgerechnet werden, die entsprechend dem von ihr reflektierten Licht nun Licht emittiert (Abb. 4.1). Für den Betrachter würde diese Änderung keinen Unterschied machen. Dabei ergeben sich folgende Pfade: `C(D|G|S)*L` für den Beleuchtungscache `C` an jedem Oberflächenpunkt der Szene x und für alle ausgehenden Richtungen $\vec{\omega}$ sowie `EC` für den Betrachter, der nun direkt auf den Cache zugreifen kann. Dieses Beispiel würde jedoch einen enormen Speicherbedarf aufweisen, weshalb in der Praxis der Cache an anderen Stellen der Pfade eingeführt wird.

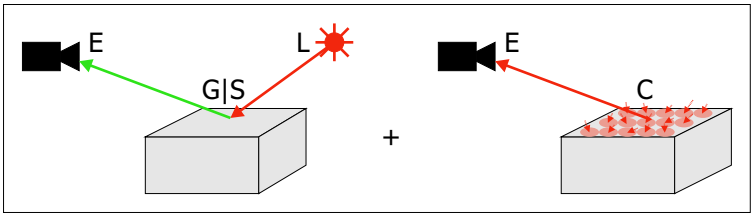


Abbildung 4.2: Die Photon Map speichert auf Oberflächen eintreffende Photonen in einer eigenen Datenstruktur parallel zur Szene. Während glossy und specular Reflexionen beim Rendern mittels MCPT wie zuvor gehandhabt werden, wird im diffusen Fall auf die Photon Map zur Bestimmung der eingehenden Radiance zurückgegriffen, da hier die geringe Auflösung nicht auffällt und im Normalfall viele Samples nötig wären, um die gesamte Hemisphäre korrekt zu erfassen.

4.2.2 Photon Map

Die in [Jensen 2001] beschriebene Photon Map speichert die eingehende Radiance auf allen diffusen Oberflächen. Dazu werden von der Lichtquelle Photonen ausgesandt, deren Pfade über MCPT durch die Szene verfolgt wird. Wann immer sie von einer diffusen Oberfläche reflektiert werden wird ein Eintrag in die Photon Map (üblicherweise ein kd-Tree, s.a. [Wald und Slusallek 2004]) vorgenommen, der die Energie des Photons und die Richtung, aus der es kam, festhält (Abb. 4.2).

Visualisiert man nun die Szene ausgehend vom Betrachter aus über MCPT, so wird der Sehstrahl auf diffusen Oberflächen nicht zufällig reflektiert. Statt dessen wird die an dieser Stelle eingehende Radiance über die nächsten k Photonen approximiert. Dadurch erzielt man verschiedene Vorteile gegenüber reinem MCPT: Die Geschwindigkeit kann durch die abgekürzten Pfade erhöht werden, das hochfrequente Bildrauschen wird durch visuell weniger auffälliges niederfrequentes Rauschen der gefilterten Photon Map ersetzt und Kaustiken (`C(D|G|S)*S+L`) können ohne deutlich erhöhten Zeitaufwand dargestellt werden.

Für die hier vorgestellte Anwendung ist Photon Mapping als solches nicht direkt anwendbar, da es zur Auswertung auf MCPT angewiesen ist, was nicht durch übliche GPUs unterstützt wird. Zur Beschleunigung des Referenzmodells, d.h. bei der Berechnung der anderen Caches, wird es jedoch eingesetzt.

4.2.3 Light Map

Light Maps bezeichnen eine Variante des Beleuchtungscache, die dem Eingangs geschilderten Beispiels recht nahe kommt: Zu jedem Punkt x der

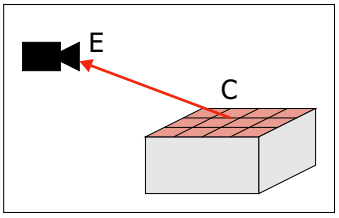


Abbildung 4.3: Light Maps speichern die Radiosity in Form einer Textur. Diese kann zwar von GPUs verarbeitet werden, gibt aber nur diffuse Reflexion korrekt wieder.

Szene wird nicht die ausgehende Radiance sondern die Radiosity gespeichert (Abb. 4.3). Durch die entfallende Winkelabhängigkeit lässt sich die Radiosity in Form einer Textur ablegen. Dabei kann die eventuell vorhandene ursprüngliche Farbtextur ersetzt werden, da diese ja nur die Reflexion von eingehendem zu ausgehendem Licht beeinflusst hat, letzteres aber nun direkt über die Light Map verfügbar ist.

Der größte Vorteil der Light Map ist, dass sie von praktisch jeder GPU unterstützt wird und sehr schnell ausgewertet werden kann. Prinzipbedingter Nachteil ist aber, dass nur noch diffuse Oberflächen visualisiert werden können, d.h. dass man sich auf Pfade der Form `ED(D|G|S)*L` beschränkt. Damit kann die Wachsschicht auf Blättern nur noch unzureichend berücksichtigt werden. Ein weitere Nachteil für den vorgestellten Anwendungsfall ist der enorm hohe Texturspeicherbedarf: Im Gegensatz zu Farbtexturen können die Light-Map-Texturen nicht für mehrere Instanzen eines Modells verwendet werden, da sie die instanzabhängige Beleuchtung wiedergeben sollen. Damit ist zwar das Rendern einer einzelnen Pflanze sehr schnell möglich, aber für Szenen mit mehreren 100000 Pflanzen wäre das Vorhalten einer Light Map für jede einzelne in Echtzeit unmöglich.

4.2.4 Environment Map

Environment Maps geben zu einem beliebigen Punkt x der Szene die eingehende Radiance in Form einer sphärischen Funktion an (Abb. 4.4). Damit können komplexe Beleuchtungsverhältnisse besser wiedergegeben werden als es über die üblichen OpenGL-Lichtquellen möglich ist ([Debevec 1998]). In Pfad-Notation bedeutet das, dass eine imaginäre Grenze zwischen Intra- und Interobjekt-Transfer gezogen und sämtliche eingehende Radiance von anderen Objekten und Lichtquellen durch den Cache erfasst wird. Meist wird pro Objekt eine Environment Map gespeichert, die dann über die Normalenvektoren des Modells indiziert wird: Bei specular Reflexionen wird der Sehstrahl des Betrachters an der Oberflächennormalen reflektiert und mit

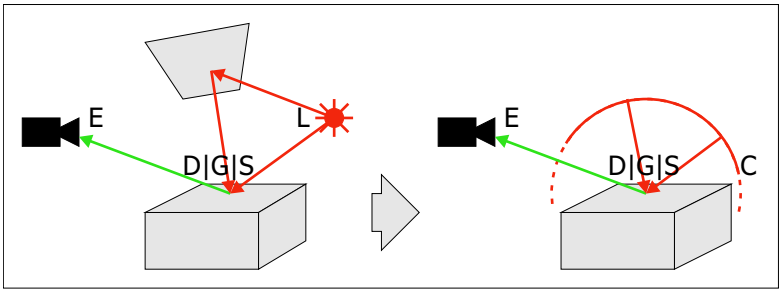


Abbildung 4.4: Die eingehende Radiance in einem Punkt wird in der Environment Map als sphärische Funktion gespeichert. Diese kann für jeden Reflexionstyp als Quelle dienen. Bei der Nutzung von einer Environment Map pro Objekt wird angenommen, dass die in der Environment Map repräsentierten Lichtquellen unendlich weit weg sind, da mangels Geometrieinformation keine Parallaxenverschiebung möglich ist.

dem Ergebnis in der sphärischen Funktion nachgeschlagen, welche Radiance aus dieser Richtung eintrifft. Dieser Wert kann anschließend direkt zurückgegeben werden. Bei glossy und diffusen Oberflächen ist eine Vorfilterung der Environment Map nötig ([Kautz et al. 2000]), die über den den Materialeigenschaften entsprechenden Bereich integriert. Für diffuse Reflexion wird mit $\cos\theta$ gewichtet für jede Richtung über die jeweilige Hemisphäre integriert. Bei der Visualisierung kann dann unabhängig vom Blickwinkel über den Normalenvektor die eingehende Radiance nachgeschlagen werden, ohne dass erneut integriert werden muss.

Environment Mapping ist in verschiedenen Varianten (Sphere Mapping, Dual-Paraboloid-Mapping) seit längerem in OpenGL vorgesehen ([Blythe et al. 1999]). Das bereits vorgestellte Cube Mapping ([Voorhies und Foran 1994]) ist seit 1999 in NVidia Geforce GPUs[2] implementiert und weist eine deutlich geringere Verzerrung auf als die vorherigen Methoden. Ein prinzipbedingtes Problem des Environment Mappings ist allerdings, dass die lokalen Reflexionen und Abschattungen nicht berücksichtigt werden: Jeder Punkt des Objekts indiziert die Environment Map nur über seinen Normalenvektor und prüft nicht, ob die Map in dieser Richtung eventuell vom Objekt selbst verdeckt wird. Auch der Speicherbedarf ist bei einer Realisierung über Cube Maps für Szenen mit vielen Objekten recht hoch, da zu jedem Objekt eine eigene Cube Map vorgehalten werden muss.

Abhilfe schafft hier eine in [Ramamoorthi und Hanrahan 2001] vorgestellte Variante, die diffus vorgefilterte Environment Maps über Ku-

[2] http://www.nvidia.com/page/geforce256.html

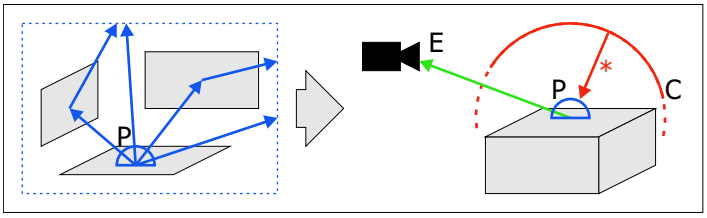


Abbildung 4.5: Der Precomputed Radiance Transfer (PRT) speichert zu einem Punkt auf der Oberfläche eines Objektes, wie sich die Radiance der Environment Map auf diesen Punkt auswirkt. Dadurch können Mehrfachreflexionen und Selbstabschattungen berücksichtigt werden. Da der Radiance Transfer nur lokal betrachtet wird, reicht ein PRT pro Modell. Die Abhängigkeit von der Szene ergibt sich durch die Auswertung mit der instanzbezogenen Environment Map.

gelflächenfunktionen kodiert. Es wird gezeigt, dass durch die Integration über die Hemisphäre die sphärische Funktion so niederfrequent wird, dass man sie mit minimalem Fehler mit nur drei SH-Bändern (9 Koeffizienten) darstellen kann. Als Matrix M geschrieben kann diese Environment Map sehr schnell über $\vec{n}^T M \vec{n}$ ausgewertet werden. Allerdings funktioniert dies nur für diffuse Environment Maps, glossy oder specular ergeben je nach Umgebung deutlich höhere Frequenzen und erfordern mehr Koeffizienten, sofern sie durch Ringing-Probleme nicht gänzlich unbrauchbar werden ([Ng et al. 2003]).

4.2.5 Precomputed Radiance Transfer

Der in [Sloan et al. 2002] vorgestellte Precomputed Radiance Transfer (PRT) erweitert das Konzept der Environment Map, indem nicht mehr auf den Normalenvektor zur Indizierung zurückgegriffen wird. Statt dessen wird zu jedem Punkt der Objektoberfläche angegeben, wie viel von der auf das Objekt eingehenden Radiance (Environment Map) dort ankommt (Abb. 4.5). Ein Blick auf die Rendering Equation (ohne Emission L_e) verdeutlicht diesen Ansatz (P_x: PRT im Punkt x, E: Environment Map):

$$L(x,\vec{\omega}) = \int_\Omega f_r(x,\vec{\omega}',\vec{\omega}) L(x,\vec{\omega}')(\vec{n}\cdot\vec{\omega}')d\vec{\omega}' \tag{4.1}$$

$$= \int_\Omega (f_r(x,\vec{\omega}',\vec{\omega})(\vec{n}\cdot\vec{\omega}'))(L(x,\vec{\omega}'))d\vec{\omega}' \tag{4.2}$$

$$= \int_S P_x(\vec{\omega}',\vec{\omega}) E(\vec{\omega}')d\vec{\omega}' \tag{4.3}$$

Dabei steht S für die komplette Sphäre um x im Gegensatz zur He-

misphäre Ω in Normalenrichtung, da durch die Intraobjekt-Reflexionen (Radiance Transfer) Licht aus jeder Richtung der Environment Map von x reflektiert werden kann. Für diffuse Oberflächen ergibt sich die einfachere Form

$$L(x, \vec{\omega}) = \int_S P_x(\vec{\omega}')E(\vec{\omega}')d\vec{\omega}' \tag{4.4}$$

Beide Varianten lassen sich nicht direkt auf GPUs übertragen, da die Integration über Ω notwendig ist. Eine mögliche Lösung dazu ist die Darstellung von P und E in Kugelflächenfunktionen: Gleichung 2.28 besagt, dass die Integration über das Produkt zweier SH-Approximationen gleich Summe der Produkte der Koeffizienten ist. Kann man also sowohl Environment Map als auch PRT über wenige SH-Koeffizienten approximieren, so lässt sich diese Summe auch effizient auf GPUs berechnen, da diese gerade für Vektoroperationen dieser Art optimiert sind.

Im Gegensatz zum in [Ramamoorthi und Hanrahan 2001] beschriebenen Ansatz kommen beim PRT aber keine vorgefilterten Environment Maps zum Einsatz - die Integration erfolgt ja gerade erst durch die Multiplikation mit dem PRT. Für halbwegs niederfrequente Umgebungen spricht [Sloan et al. 2002] von guten Ergebnissen bei 9 bis 25 Koeffizienten (pro Vertex) im diffusen Fall. Um die Blickwinkelabhängigkeit im allgemeinen Fall zu berücksichtigen wird in [Sloan et al. 2002] mit 625 Koeffizienten gearbeitet und dementsprechend nur eine CPU-Lösung angeboten. Varianten dieser Technik sind in [Sloan et al. 2003b], [Sloan et al. 2003a] und [Annen et al. 2004] beschrieben, eine gutes Tutorial mit Code-Beispielen ist [Green 2003].

Parallel dazu wurde in [Ng et al. 2003] PRT auf Basis von Cube Wavelets vorgestellt, allerdings mit dem Schwerpunkt auf hohe Bildqualität und nicht auf Echtzeit-Darstellungen. Vorteil dieses Ansatzes ist, dass sie nur die jeweils größten Wavelet-Koeffizienten verwenden und so zum einen wenig Speicherplatz benötigen, zum anderen aber dünn besetzte Matrizen (für den allgemeinen Fall) schnell multiplizieren können - allerdings auch nur über die CPU. Auch zeigt die Wavelet-Basis kein Ringing, kann also beliebige sphärische Funktionen problemlos approximieren. Eine denkbare Verbesserung wäre die Nutzung von Spherical Wavelets an Stelle der Cube Wavelets, was die Approximation der Sphäre durch einen Würfel erspart.

Nahe liegend ist es nun, Wavelets mit wenigen Koeffizienten analog zu den Kugelflächenfunktionen zu verwenden, was dann zwar auch auf diffuse Oberflächen beschränkt wäre, aber keine Ringing-Probleme hätte. Gegen diese Übertragung spricht aber, dass für Cube und Spherical Wavelets keine einfache Rotationsmöglichkeit existiert, wie sie für Kugelflächenfunktio-

Technik	diffus	glossy	Frequenz	Schatten	Speicher/Instanz	Speicher/Vertex	GPU-Zeit	GPU-Bandbreite
Light Map	+	-	+	+	- -	++	++	-
Env Map	+	+	+	-	-	++	++	-
Env Map SH	+	o	+	-	+	++	+	+
PRT SH	+	-	-	+	+	o	o	o
PRT WL	+	-	o	+	+	o	o	o

Tabelle 4.1: Vergleich der echtzeitfähigen Beleuchtungs-Cache-Methoden

nen gegeben ist (Gleichung 2.27). Dadurch können die Modell-Instanzen in der Szene zwar verschoben werden, eine Rotation wäre aber nur bedingt möglich, bei Cube Wavelets z.B. durch Permutation der Würfelseiten.

4.2.6 Bewertung

Die Eigenschaften der verschiedenen Caches werden in Tabelle 4.1 zusammengefasst. „Diffus“ und „glossy“ geben die Eignung für den jeweiligen Reflexionstyp an, „Frequenz“ die für höhere Frequenzen und Schatten erfasst, ob ein Objekt sich selbst abschatten kann. Der Speicherbedarf wird aufgeteilt nach „Instanz“ und „Vertex“ erfasst, wobei Speicherbedarf pro Instanz angibt, ob zusätzliche Instanzen wenig zusätzlichen Speicher benötigen. Pro Vertex gibt an, ob der Speicherbedarf unabhängig von der Komplexität des Modells selbst ist oder mit dieser ansteigt. „GPU-Zeit“ gibt die benötigte Rechenzeit in Vertex- bzw. Fragmentshadern an wohingegen „GPU-Bandbreite“ die erforderte Speicherbandbreite z.B. für Texture-Lookups bewertet. Alle „+“ stehen für eine gute Eignung für Echtzeit-Anwendungen, „-“ für problematische Punkte, die es gesondert zu berücksichtigen gilt. „o“ ist der neutrale Mittelwert, also weder Vor- noch Nachteil im Vergleich mit den anderen Techniken.

Besonders relevant bei der Anwendung im Rahmen der Landschaftsvisualisierung ist der niedrige Speicherbedarf pro Instanz, weshalb Light Maps von vornherein ausscheiden, auch einfache Environment Maps sind praktisch nicht nutzbar. Da gerade komplexere Pflanzen wie Büsche und Bäume stark zur Selbstabschattung tendieren, spricht einiges für PRT basiert auf Kugelflächenfunktionen. Das größte Problem hierbei ist, dass die benötigten SH-kodierten ungefilterten Environment Maps hohe Frequenzen

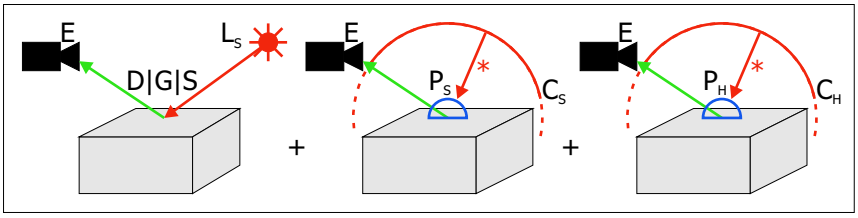


Abbildung 4.6: Die Vorteile der einzelnen Techniken können kombiniert werden, indem man die Pfade kategorisiert: Die Reflexion von direktem Sonnenlicht wird über einen Shader direkt auf der GPU errechnet. Mehrfachreflexionen der Sonne sowie einfallendes Umgebungslicht wird über PRT erfasst und dazu addiert.

sehr schlecht wiedergeben, aber gerade die Sonne am unbedeckten Himmel durch ihren kleinen Radius aber gerade diese erfordert. Zusammenfassend bleibt also zu sagen, dass keine der vorgestellten Methoden allein das Problem löst.

4.3 Idee

Die Idee, die zur Implementierung des Renderings für diese Diplomarbeit geführt hat, ist die Aufteilung der Beleuchtungspfade (Abb. 4.6): Die GPU kann mit Standard-Techniken alle `E(D|G|S)`L_S-Pfade verarbeiten, also das Licht, was von der Sonne als fast ideale Richtungslichtquelle direkt auf Blätter fällt und von dort zum Betrachter reflektiert wird. Übrig bleiben zwei weitere Komponenten: Pfade, bei denen der restliche Himmel ohne die Sonne die Lichtquelle ist (`E(D|G|S)*`L_H), und Pfade, die von der Sonne ausgehend mindestens zwei mal reflektiert werden (`E(D|G|S)+`L_S). Durch diese Aufteilung wird der Hauptnachteil des PRT mit Kugelflächenfunktionen umgangen: Der Himmel ohne die Sonne ist vergleichsweise niederfrequent, wobei die in den Environment Maps der einzelnen Pflanzen berücksichtigten anderen Szenenobjekte sich auch nicht allzu problematisch auswirken. Auch wirkt sich die Annahme rein diffuser Oberflächen nicht sichtbar aus, da nur die Reflexion der Sonne an der Wachsschicht deutlich wahrnehmbar ist. Auf dem Papier problematischer ist die Mehrfachreflexion der Sonne, da hierfür die Sonne selbst in einer Environment Map kodiert werden muss, was die oben angesprochenen Frequenzprobleme mit sich bringt. Allerdings ist dieser Fehler nach mehrfacher Reflexion praktisch nicht mehr wahrnehmbar, weshalb dies kein praktisches Problem darstellt.

Im folgenden werden nun die einzelnen Elemente der Visualisierung vorgestellt. Zunächst muss ein GPU-Schattenalgorithmus ausgewählt werden, da einige Ansätze mit verschiedenen Vor- und Nachteilen existieren. Danach

gilt es den Referenz-Renderer effizient zu gestalten, damit die Vorberechnung der Environment Maps und des PRT handhabbar bleibt, die im darauf folgenden Abschnitt beschrieben wird. Zuletzt erfolgt die Darstellung der Punkte, die zur Laufzeit auf der GPU abgearbeitet werden.

4.4 GPU-Schatten

Bei der Rasterisierung der Geometrie in der GPU stehen zunächst nur Daten über den aktuell bearbeiteten Punkt x zur Verfügung, Informationen über die restliche Szene fehlen. Damit allein ist es nicht möglich, Schatten zu berechnen, da nicht geprüft werden kann, ob zwischen x und der Lichtquelle weitere Geometrie liegt. Schattenberechnungen auf der GPU erfolgen deshalb generell in zwei Schritten: Zuerst werden für eine Lichtquelle die Bereiche bestimmt, die vom Licht erreicht werden, und danach wird diese Information parallel zu den Szenendaten der GPU zur Verfügung gestellt, damit zu jedem x getestet werden kann, ob es im Bereich der Lichtquelle liegt.

4.4.1 Shadow Map

Eine intuitive Variante, die ausgeleuchteten Bereiche zu bestimmen, ist das Shadow Mapping (Abb. 4.7). Dabei wird die Szene zunächst aus Sicht der Lichtquelle L gerendert, wobei nur der Z-Buffer von Bedeutung ist und dem zweiten Durchgang übergeben wird. In diesem kann nun zu jedem x die Entfernung d_L von der Lichtquelle bestimmt und mit dem zuvor berechneten Z-Wert z_x verglichen werden. Ist $d_L <= z_x + \epsilon$, so handelt es sich um den der Lichtquelle nächsten Punkt in dieser Richtung, er wird also direkt beleuchtet. Für $d_L > z_x + \epsilon$ bedeutet das, dass ein anderes Objekt zwischen x und L liegt. Das ϵ dient dem Ausgleich von Rundungsfehlern und ist in Abhängigkeit der Implementierung und der Szene zu wählen. Eine Übersicht über Shadow Mapping wird im Siggraph-Kurs [Akenine-Moeller et al. 2004] gegeben, die Hardware-Implementierung in [Everitt et al. 2001] beschrieben.

Da zur Erstellung der Shadow Map dieselbe Technik zum Einsatz kommt, die auch das Bild aus Sicht des Betrachters rendert, gelten auch dieselben Einschränkungen: Die Auflösung der Shadow Map wird durch die maximale Framebuffergröße der GPU begrenzt. Da Lichtquelle und Betrachter i.A. unterschiedliche Positionen haben, sind die Pixel der Shadow Map nicht 1:1 den Pixel der Betrachteransicht zugeordnet, wodurch es je nach Szene und Positionen zu sehr grob aufgelösten Schatten kommen kann (Abb. 4.8).

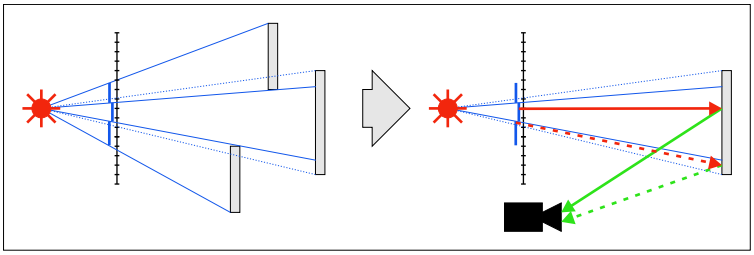


Abbildung 4.7: Beim Shadow Mapping wird die Szene zunächst aus Sicht der Lichtquelle in einen Z-Buffer (Shadow Map) gerendert. Beim anschließenden Rendern aus Sicht der Kamera wird jeder Punkt erneut in die Lichtquellen-Ansicht projiziert und der Z-Wert mit der Shadow Map verglichen. Ist er größer als der Eintrag in der Map, so liegt ein anderes Objekt aus Sicht der Lichtquelle vor dem gerade verarbeiteten und der Punkt befindet sich damit im Schatten.

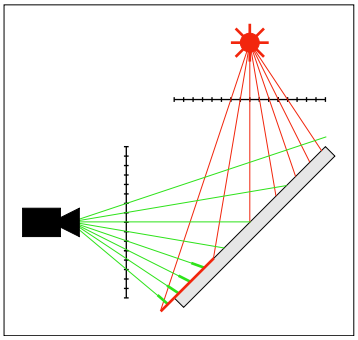


Abbildung 4.8: Da die Shadow Map meist aus einer anderen Perspektive als die der Kamera gerendert wird, können die Pixel der Shadow Map je nach Szene mehrere Pixel der Kamera abdecken, was sich in grob aufgelösten Schatten widerspiegelt. Umgekehrt kann die Shadow Map in anderen Bereichen deutlich höher aufgelöst sein, was von der Kamera nicht erfasst werden kann.

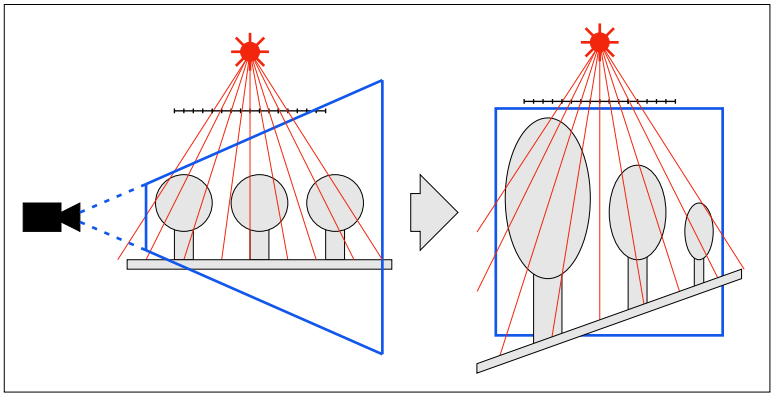


Abbildung 4.9: Beim Perspective Shadow Mapping wird die Szene vor der Rasterisierung aus Sicht der Lichtquelle in den postprojektiven Raum der Kamera transformiert, d.h. mit Modelview- und Projection-Matrix multipliziert. Dadurch werden Objekte nahe der Kamera automatisch größer und in der Shadow Map entsprechend besser aufgelöst.

Um dem zu begegnen, wird in [Stamminger und Drettakis 2002] das Perspective Shadow Mapping (PSM) vorgestellt, das die Shadow Map entsprechend der Sichtpyramide des Betrachters verzerrt (Abb. 4.9). Dies wird erreicht, indem die Lichtquelle die Szene nach der Transformation im postprojektiven Koordinatensystem des Betrachters zu sehen bekommt. Dadurch werden automatisch die Bereiche nahe des Betrachters vergrößert und die weiter entfernten verkleinert, die relative Auflösung der Shadow Map in der Nähe des Betrachters also erhöht. Dieser Ansatz ist nicht trivial zu implementieren, weshalb [Kozlov 2004] einige Verbesserungen bezüglich der Stabilität des Algorithmus einführt. Eine Alternative stellen die Light Space Perspective Shadow Maps (LSM) aus [Wimmer et al. 2004] dar, die eine speziell angepasstes post-projektives Koordinatensystem wählen.

4.4.2 Trapezoidal Shadow Map

Ein weiterer Ansatz zur besseren Gewichtung der Auflösung sind Trapezoidal Shadow Maps (TSM, [Martin und Tan 2004]) (Abb. 4.10). Die Sichtpyramide des Betrachters entspricht in den meisten Ausrichtungen aus Sicht der Lichtquelle einem Trapezoid (Abb. 4.11). Die Shadow Map wird so gedreht, dass der Trapezoid eine bestimmte Ausrichtung erhält und dann perspektivisch in Richtung eines Rechtecks verzerrt. Dadurch wird die schmale Seite nahe des Betrachters vergrößert, die entfernte Seite verkleinert, und somit die gewünschte Auflösungsverschiebung erreicht. Diese Transformation arbeitet auch ohne besondere Vorkehrungen stabiler als PSM, ist ver-

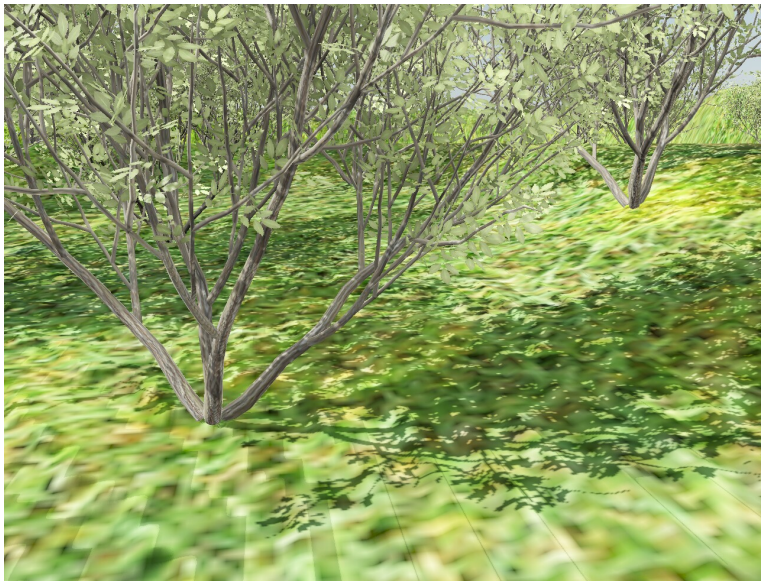


Abbildung 4.10: Die Trapezoidal Shadow Map ermöglicht sehr detaillierte Schatten trotz komplexer Geometrie.

gleichsweise einfach zu implementieren und erzeugt oft bessere Schatten als LSM. Problematisch sind allerdings die Fälle, in denen die Blickrichtung des Betrachters nahezu parallel zur Lichtrichtung ist, da dann die Sichtpyramide keinen Trapezoid darstellt. Dem kann durch Anpassung der Far-Clipplane entgegengewirkt werden, so dass weiter entfernte Objekte zwar keine Schatten mehr werfen, nahe aber besser dargestellt werden.

Bei der Berechnung der TSM muss zunächst die konvexe Hülle der Sichtpyramide auf der Shadow Map bestimmt werden. [Graham 1972] stellt dazu einen Algorithmus in $O(n\,logn)$ vor, der zwar aus heutiger Sicht asymptotisch nicht optimal ist, aber für niedrige n wie hier ($n = 8$) durchaus schnell und vor allem einfach zu implementieren ist. Der aus der konvexen Hülle errechnete Trapezoid wird über in [Heckbert 1989] beschriebene Transformationen in Richtung eines Rechtecks verzerrt. Eine Einschränkung der anschließend zu zeichnenden potentiell schattenwerfenden Objekte wird in [Kozlov 2004] für PSM beschrieben, funktioniert aber genauso bei TSM: Die Bounding Sphere eines Objektes wird entlang der Lichtrichtung extrudiert und mit der Sichtpyramide geschnitten. Damit fallen die Objekte, die aus Sicht der Lichtquelle neben oder hinter der Sichtpyramide liegen, weg.

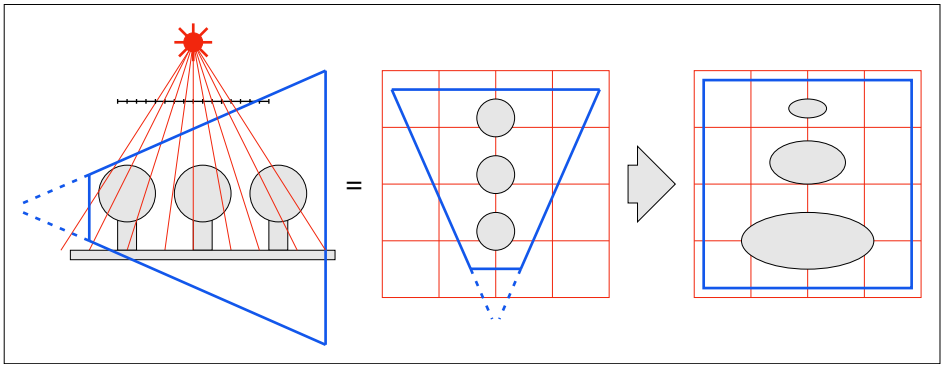


Abbildung 4.11: Beim Trapezoidal Shadow Mapping wird die Sichtpyramide aus Sicht der Lichtquelle perspektivisch von einem Trapez in ein Rechteck transformiert, was die Objekte nahe der Kamera vergrößert und damit in der Shadow Map besser auflöst.

4.4.3 Shadow Volumes

Eine zweite Art, Schatten auf der GPU zu berechnen, bilden neben den Shadow Maps die Shadow Volumes (Abb. 4.12, Hardware-Implementierung in [Everitt und Kilgard 2002]). Hier werden die schattenwerfenden Objekte nicht rasterisiert, sondern in Richtung der Lichtstrahlen rein geometrisch extrudiert. Bei der anschließenden Rasterisierung aus Sicht des Betrachters werden die extrudierten Silhouetten mit dem Stencil-Buffer verrechnet, indem man nun nachschlagen kann, ob sich der aktuelle Punkt in einem Schattenvolumen befindet oder nicht. Dieser Ansatz ermöglicht exakte Schatten ohne die Diskretisierungsartefakte der Shadow Maps.

Ein Nachteil der Shadow Volumes ist aber der vergleichsweise hohe Aufwand, die Silhouetten über die CPU zu bestimmen und die Schattenvolumen zu rasterisieren. Bei komplexeren Pflanzen wäre eine Echtzeit-Realisierung derzeit nicht möglich, zumal die Silhoutten der Blätter im Lenné3D-Projekt nur implizit durch Alpha-Texturen gegeben sind.

4.5 Monte-Carlo-Path-Tracer

Zur Bestimmung der im Beleuchtungscache gespeicherten Werte dient ein Referenz-Renderer auf MCPT-Basis. Obgleich der Cache dank der statischen Szene nicht zur Laufzeit der Visualisierung aktualisiert werden muss, ist sein Rechenzeitbedarf nicht unwichtig. Die Szenenkomplexität reicht von etwa 10^5 bis 10^8 Dreiecken, eine lineare Suche zur Bestimmung der Strahlschnittpunkte ist damit praktisch unmöglich.

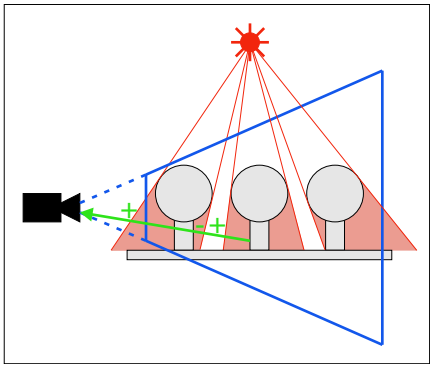


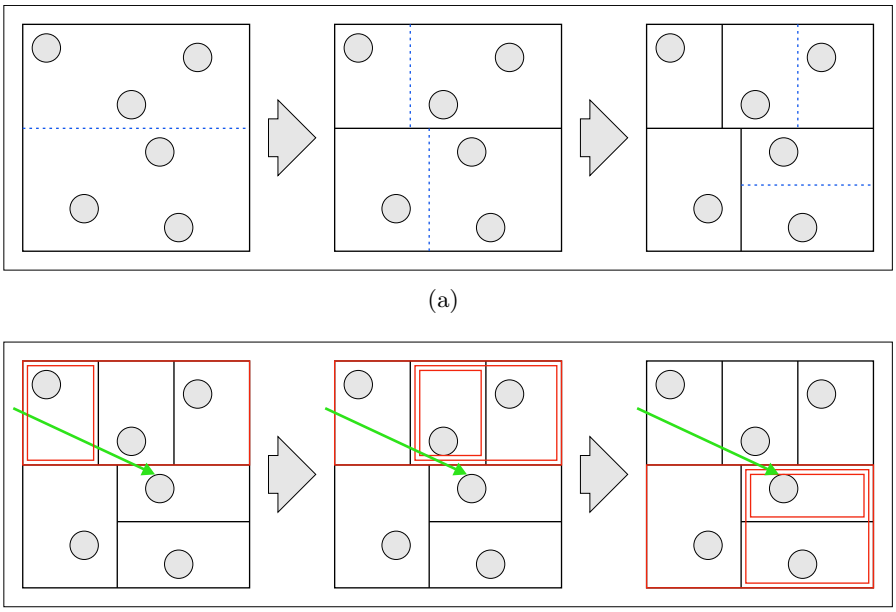
Abbildung 4.12: Shadow Volumes bezeichnen die Volumen, die durch Extrusion der Objektsilhouetten aus Sicht der Lichtquelle entstehen. Beim Rendern (aus Kamerasicht) werden zunächst die die Shadow Volumes begrenzenden Polygone in den Stencil-Buffer gezeichnet, wobei für Front- und Backface jeweils 1 addiert bzw. subtrahiert wird. Beim anschließenden Rendern der Objekte kann im Stencil-Buffer nachgeschlagen werden, ob ein Punkt im Schatten liegt oder nicht. Shadow Volumes werden also passend zum Framebuffer rasterisiert und nicht wie Shadow Maps unabhängig davon, wodurch pixelgenaue Schatten erzeugt werden.

4.5.1 Raumteilung

Das wichtigste Konzept zu effizientem Raytracing ist die Raumteilung. Dabei wird eine hierarchische Struktur aufgebaut, mit der man schnell ganze Teilmengen der Szene vom Strahlschnitt ausschließen kann. Die nach [Havran 2000] wirksamste Datenstruktur für diesen Zweck ist der KD-Tree (Kap. 5.1.2), der in [Fussell und Subramanian 1988] erstmals für diese Anwendung genutzt wird (Abb. 4.13). Die Szene wird iterativ in jeweils zwei Bereiche aufgeteilt, so dass eine Art Binärbaum entsteht. Die Aufteilung erfolgt jeweils entlang einer der drei Koordinatenachsen. [Havran 2000] diskutiert die Wahl der Position der Teilungsebenen und beschreibt die hier verwendete Surface Area Heuristic (SAH), die für Aufteilung an Hand der Strahlschnittwahrscheinlichkeit sorgt. Da die Szene aus Dreiecken besteht, muss in jedem Schritt geprüft werden, ob ein Dreieck links, rechts oder in beiden Teilräumen liegt. Ein effizienter Test hierzu wird in [Akenine-Möller 2001] vorgestellt.

4.5.2 Ray-Triangle-Intersection

Hat man mit Hilfe des KD-Trees bestimmt, welche Dreiecke für einen Schnitt mit einem gegebenen Strahl in Frage kommen, muss dieser letztendlich auch berechnet werden. Dabei gibt es zwei sehr effiziente Ver-



(a)

(b)

Abbildung 4.13: (a) Der KD-Tree wird durch wiederholte Unterteilung der Szene aufgebaut. Dabei wird jeweils parallel zu den Koordinatenachsen eine Trennebene gewählt, die die Objekte so auf verschiedene Teilräume verteilt, dass beim Raytracing möglichst wenige Schnitte mit den Objekten notwendig sind. (b) Beim Raytracing wird zunächst der dem Strahlursprung zugewandte Teilraum jeder Trennebene geprüft. Dabei können die, die vom Strahl nicht geschnitten werden, komplett übersprungen werden (z.B. rechts oben).

fahren, die aber auf unterschiedliche Szenarien optimiert sind: Der Test in [Möller und Trumbore 1997] prüft nacheinander die Schnittkriterien, wobei einfach berechenbare zuerst ausgewertet werden. Dadurch kann sehr schnell abgebrochen werden, wenn klar ist, dass ein Kriterium nicht erfüllt ist. Der Test in [Wald 2004] ist hingegen so strukturiert, dass zuerst die Berechnungen initiiert werden, deren CPU-Latenzzeiten besonders hoch sind, so dass diese auf aktuellen CPU parallel zu den weiteren Rechnungen ausgeführt werden können. Damit ist [Wald 2004] im Vorteil, wenn die Wahrscheinlichkeit eines Schnitts sehr hoch ist, was neben einer guten Raumteilung auch geeignete Szenen voraussetzt. Ein weiterer Unterschied ist, dass [Möller und Trumbore 1997] keinen zusätzlichen Speicherbedarf hat, [Wald 2004] jedoch auf spezielle Vorberechnungen angewiesen ist.

4.5.3 Cache-Optimierung

Die vorberechneten Daten in [Wald 2004] benötigen zwar zum einen mehr zusätzlichen Arbeitsspeicher, ermöglichen aber auch, den Cache aktueller CPUs durch geschickte Strukturierung optimal zu nutzen, da die Daten unabhängig von der eigentlichen Geometrie gespeichert werden können. Das in [Wald 2004] vorgestellte Speicherlayout und die dazugehörigen Algorithmen erhöhten die Geschwindigkeit gegenüber der vorherigen unoptimierten Implementierung um den Faktor 200, was hauptsächlich auf den großen Bandbreiten- und Latenzunterschied zwischen L1- bzw. L2-Cache und dem Hauptspeicher zurückzuführen ist.

4.5.4 Modellprobleme

Die Anforderungen an eine gute Raumteilung stehen der Geometrie der Lenné3D-Pflanzen gegenüber: Die Blätter werden nur grob über Dreiecke approximiert, von denen weite Teile durch die Alpha-Texturen transparent geschaltet sind. Dadurch entspricht die Geometrie nur eingeschränkt dem visuellen Erscheinungsbild und weist bei komplexen Bäumen und Sträuchern eine hohe Anzahl sich gegenseitig im transparenten Bereich überschneidende Dreiecke auf (Abb. 4.14). Diese können über KD-Trees nur unzureichend getrennt werden, wodurch die Zahl der zu prüfenden Schnitte vergleichsweise groß ist.

Auf der anderen Seite sind ausmodellierte Bäume (ohne Alpha-Transparenz) so komplex, dass die schiere Datenmenge zum Problem wird. Die Anforderungen verschieben sich, es lässt sich aber nur schwer sagen, welcher Ansatz am Ende schneller ist. Weitere Test mit verschieden detailliert

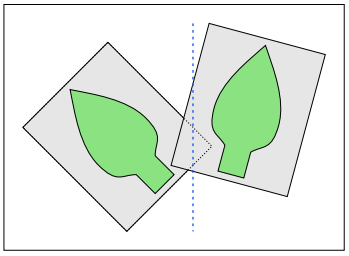


Abbildung 4.14: Obgleich sich die Blätter nicht überschneiden, überlappen sich die Polygone, die die Blatttextur tragen. Dadurch können die Blätter nicht über übliche KD-Tree-Schnitte getrennt werden, die nur die Geometrie, nicht die Materialien (inkl. Textur) berücksichtigen. Dieses Problem lässt sich aber auch nur bedingt über besser angepasste Polygone lösen, da die ansteigende Datenmenge den Vorteil der besseren Aufteilung teils zunichte macht.

modellierten Blättern sind erforderlich, um eine optimale Raytracinggeschwindigkeit zu erreichen.

4.5.5 Terrain

Um das Raytracing des Terrains zu beschleunigen, wird über das Gitter der Height Map ein Quadtree (s.a. [Samet 1990]) gelegt, dessen Zellen minimale und maximale Höhe des überdeckten Terrains angeben. Dadurch ist es im Gegensatz zu [Musgrave 1989] möglich, ganze Terrainbereiche deren Bounding Boxen den Strahl nicht schneiden zu überspringen (Abb. 4.15). Um den Schnitt einer Terrain-Zelle mit dem Strahl zu berechnen, wird die bilineare Approximation aus [Qu et al. 2003] verwendet.

4.6 Vorberechnung

Mit Hilfe des beschriebenen Referenz-Renderers können nun die einzelnen Beleuchtungscaches aufgebaut werden.

4.6.1 Pflanzen

Die Pflanzenmodelle werden mit PRT-Daten pro Vertex ausgestattet, d.h. zu jedem Vertex wird in Form von Kugelflächenfunktionen kodiert, wie das Umgebungslicht aus einer Environment Map `C` reflektiert wird. Dazu werden ausgehend von einem Vertex `V` via MCPT Strahlen in zufällige Richtungen verfolgt und für den Fall, dass ein Strahl keinen Schnittpunkt mit dem Objekt aufweist, seine Richtung $\vec{\omega}$ im Vertex selbst festgehalten (Abb. 4.16). Damit ergeben sich für den PRT Pfade der Form `V(D|G|S)*C`, wobei

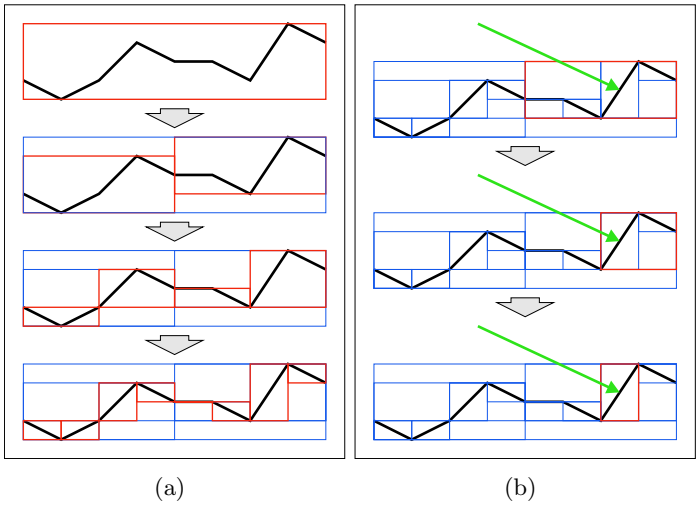


Abbildung 4.15: (a) Über der Terrain-Height-Map wird ein Quadtree aufgebaut. Die Positionsangaben sind dabei implizit durch den Baumaufbau gegeben, so dass nur maximaler und minimaler Höhenwert gespeichert werden müssen. (b) Das Raytracing erfolgt analog zum KD-Tree, d.h. nicht geschnittene Teilbäume werden so früh wie möglich übersprungen.

`C` selbst die Pfade von der Lichtquelle bis zur Position der Environment Map enthält, also `C(D|G|S)*L`. Beim anschließenden Rendern entspricht die PRT-Auswertung in `V` einer diffusen Reflexion, der gesamten Pfad ist also `ED(D|G|S)*L`.

Durch die in 4.3 skizzierte Aufteilung der Pfade müssen prinzipiell zwei verschiedene PRT errechnet werden: Die direkte Reflexion der Sonne wird über eigene Routinen behandelt und ist nicht Teil des PRT. Indirektes Sonnenlicht, d.h. Licht, dass von der Pflanze selbst mehrfach reflektiert wird, soll aber auf diese Weise kodiert werden, weshalb neben dem beschriebenen PRT ein weiterer Datensatz benötigt wird, der nur Pfade der Form `V(D|G|S)+C` enthält. Diese können zur Laufzeit mit einer Environment Map der Sonne verrechnet werden.

Dieser Anforderung steht eine Limitierung aktueller GPUs entgegen: Die PRT-Daten werden pro Vertex errechnet und Vertexshader ausgewertet. Um dort verfügbar zu sein müssen sie entweder als Vertex-Attribut oder über eine Vertex-Textur zur Verfügung gestellt werden. Letztere weisen aber im Vergleich zu den hochoptimierten Fragmentshader-Textureinheiten hohe Latenzzeiten auf, weshalb diese Variante in der Praxis ausscheidet. Die Zahl der Vertex-Attribute ist aber auf NVidia Geforce 6800-GPUs auf 16

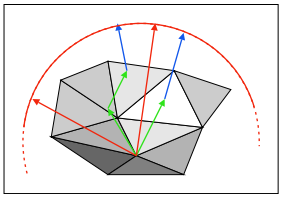


Abbildung 4.16: Ausgehend von einem Vertex werden zufällige Strahlen verfolgt, bis sie die Pflanze selbst nicht mehr schneiden. Da die letzte Strahlrichtung die Richtung bestimmt, für die die Environment Map ausgewertet werden muss, wird diese an Stelle der Richtung, in die der Strahl ursprünglich gesandt wurde, gespeichert. Aufgrund der Unterteilung des Sonnenlichts in einfache und mehrfache Reflexion wird zusätzlich noch zwischen den entsprechenden Pfaden im PRT unterschieden.

`vec4`, auf der ATI Radeon x800 auf 32 begrenzt[3]. Davon werden zunächst zwei für Position und Texturkoordinaten benötigt, weitere drei für die Basis des Tangentialraums. Die 11 `vec4` müssen allerdings noch aufgeteilt werden: Da Pflanzenmodelle beidseitig beleuchtet werden, stehen also pro Vertex-Seite noch maximal 5 `vec4` zur Verfügung, sofern man einen einzelnen Vektor nicht auf beide Seiten verteilen möchte.

Aus dieser Beschränkung ergibt sich folgende Aufteilung der Koeffizienten: Band 1 wird in RGB übergeben und belegt den ersten `vec4`. Die Bänder 2 bis 4 werden nur noch monochrom verarbeitet, was 15 `float`, also 4 `vec4` entspricht. Damit ist allerdings erst ein PRT-Datensatz kodiert. Der zweite für das indirekte Sonnenlicht benötigte wird auf einen einzigen Faktor reduziert: Bei der PRT-Berechnung wird bestimmt, wie viel Prozent der eingehenden Radiance direkt und wie viel indirekt auf den Vertex fällt. Dieser Wert wird im Alpha-Kanal von Band 1 übergeben, so dass man mit den 5 `vec4` pro Seite auskommt. Diese Approximation ist sehr grob, von der Wahrnehmbarkeit her aber vertretbar, da das indirekte Sonnenlicht keinen allzu großen Anteil am Endergebnis hat.

4.6.2 Terrain

Für das Terrain wird eine einfache Light Map erstellt. Dies ist problemlos möglich, da es nur ein einziges Mal instanziiert wird, also kein Unterschied zwischen Speicherbedarf pro Instanz und pro Modell besteht. Die Light Map wird analog zur Color Map angelegt, so dass dieselben Texturkoordinaten zur Indizierung verwendet werden können. Die Annahme diffuser Reflexion ist beim Terrain durchaus gerechtfertigt, da in der Natur nur selten glänzende Böden auftreten.

[3] http://www.delphi3d.net/hardware/

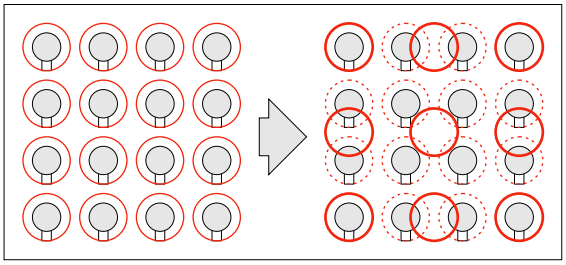


Abbildung 4.17: Ähnliche Environment Maps können zu Clustern zusammengefasst werden.

4.6.3 Szene

Zur Auswertung der PRT der Pflanzen wird für jede Pflanzeninstanz eine Environment Map benötigt, genauer gesagt eine für Umgebungslicht und eine für die Sonne. Entsprechend der Einschränkungen der GPU auf vier SH-Bänder ergibt sich somit ein Speicherbedarf von $2 \cdot 4^2 \cdot 3 = 96$ `float` für zwei RGB Environment Maps. Umgerechnet auf die derzeit größten in Echtzeit darstellbaren Szenen sind das rund 70 Megabyte ($96 \cdot 200000 \cdot 4$ Byte), also an sich noch kein Problem. Im Vergleich zum restlichen Speicherbedarf pro Pflanze, d.h. Position, Skalierung und Z-Rotation ($(3 + 1 + 1) * 4 = 20$ Bytes) handelt es sich aber um eine nicht vernachlässigbare Größe, was bei einer Speicherung in BitTrees (5.1.1) noch deutlicher wird. Damit die Environment Maps nicht die Szenengröße limitieren, wird im folgenden ein Verfahren vorgestellt, über Clustering und Interpolation den Speicherbedarf deutlich zu senken.

4.6.3.1 Cluster

Die Environment Maps zweier benachbarter Individuen eines Weizenfeldes unterscheiden sich meist nur geringfügig. Aber auch außerhalb von Monokulturen sind Sprünge in benachbarten Environment Maps nur selten, nicht zuletzt da sie bedingt durch ihre bandlimitierende Kodierung kaum Details wiedergeben können. Diese Eigenschaft kann man sich durch Clustering zu Nutze machen und ganze Bereiche über wenige Datensätze approximieren (Abb. 4.17).

Eine ähnliche Aufgabenstellung wird in [Pauly et al. 2002] beschrieben, wo Punktwolken dezimiert werden sollen. Die Clustering-Ansätze lassen sich dabei in zwei Gruppen unterteilen: Bottom-Up und Top-Down. Bottom-Up-Clustering geht von den einzelnen Samples aus und bildet iterativ aus diesen gleichförmige Bereiche. Konkret beschrieben wird die Möglichkeit, einem einzelnen Sample so lange den jeweils nächsten Nach-

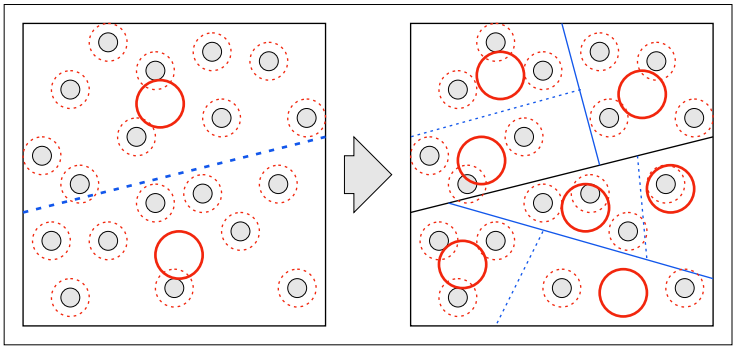
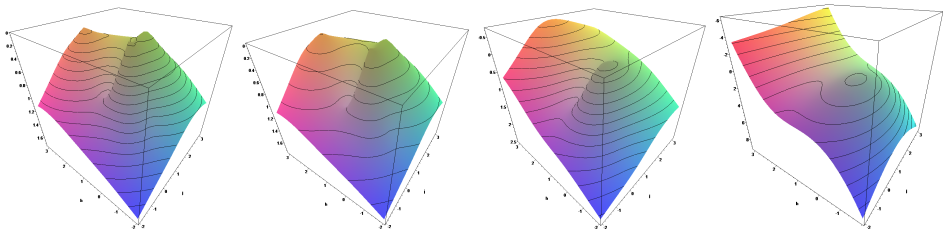


Abbildung 4.18: Ausgehend von einem Cluster, der alle Environment Maps umfasst, wird so lange jeweils der Cluster mit der größten Abweichung entlang einer fehlerminimierenden Trennebene aufgeteilt, bis die Fehlerschranke erreicht ist.

barn zuzuordnen, bis eine Fehlerschranke bei der Approximation der Gruppe durch einen Einzelwert erreicht wird. Danach erstellt man die nächste Gruppe ausgehend von dem Sample, bei dem die Fehlerschranke der vorherigen erreicht bzw. überschritten wurde. Die bei diesem Clustering-Verfahren angenommene Rekonstruktion erfolgt durch Wahl des die Gruppe repräsentierenden Wertes. Durch diese stückweise konstante Approximation entstehen an den Übergängen Unstetigkeiten, die je nach Anwendungsgebiet zu unerwünschten Artefakten führen können.

Eine im hier behandelten Fall sinnvollere Lösung ist ein Top-Down-Verfahren: Bei dieser Art Clustering wird zunächst ein Cluster aus allen Samples erstellt. Danach wird in jedem Schritt jeweils der Cluster mit dem größten Fehler in zwei neue Cluster unterteilt, bis die Fehlerschranke erreicht oder unterschritten wird. Der Vorteil bei diesem Ansatz im Vergleich zum Bottom-Up aus [Pauly et al. 2002] ist, dass die Daten gleichmäßig geclustert werden, d.h. dass sich die Fehler der einzelnen Cluster in ähnlichen Größenordnungen bewegen. Dadurch kann man schon beim Clustern auf Interpolationsmethoden zurückgreifen, die später auch zur Rekonstruktion verwendet werden können. [Heckel et al. 1999] stellt ein solches Verfahren für Vektorfelder vor, was abgesehen von der Fehlerberechnung direkt übernommen werden konnte (Abb. 4.18). Ein Implementierungsunterschied ist außerdem das Verfahren zur Berechnung der Eigenvektoren für die Trennebenen. Da [Hotelling 1933] nur für positive Eigenwerte funktioniert, wird statt dessen die Jacobi-Rotation aus [Press et al. 1992] verwendet.



(a) (b) (c) (d)

Abbildung 4.19: Multiquadric-Rekonstruktion eines Höhenverlaufs aus einzelnen Samples, von (a) bis (d) mit zunehmendem R.

4.6.3.2 Interpolation

Die Rekonstruktion der einzelnen Environment Maps erfolgt durch Interpolation. Aufgrund der beliebigen Position der Cluster kann im Gegensatz zu [Franzke 2004] keine einfache trilineare Interpolation eingesetzt werden. In [Hardy 1971] wurde das Multiquadric-Verfahren zur Rekonstruktion von Höhenverläufen aus ungeordneten Samples vorgestellt (Abb. 4.19), das aber auch bei Vektorfeldern vergleichsweise gute Ergebnisse erzielt.

Zu einem beliebigen Punkt x werden die k nächsten Samples bestimmt. Aus diesen wird zunächst das Gleichungssystem

$$C_i = \sum_{j=1}^{k} \alpha_j \sqrt{\|x_i - x_j\|^2 + R^2} \tag{4.5}$$

aufgestellt, wobei R eine szenenabhängige Konstante ist, die den Frequenzgang der Rekonstruktion limitiert (größere R ergeben einen niedrigeren Tiefpass). C_i ist die Environment Map (Cache) für Sample i, x die Position. Daraus werden nun die α_j berechnet, was allerdings nur über einfache Verfahren wie den Gauss-Algorithmus mit Pivotisierung geschehen kann, da die Matrix zwar symmetrisch aber weder positiv definit noch diagonaldominant ist, numerische Verfahren wie konjugierte Gradienten oder die Gauss-Seidel-Iteration also nicht anwendbar sind. Die Rekonstruktion der Environment Map für x erfolgt nun durch einfaches Einsetzen:

$$C = \sum_{j=1}^{n} \alpha_j \sqrt{\|x - x_j\|^2 + R^2} \tag{4.6}$$

4.6.3.3 Abstandsmaß

Das Abstandsmaß in [Heckel et al. 1999] ist die Abweichung der Stromlinien zwischen dem Ausgangsdatensatz und dem interpolierten Vektorfeld.

Für eine vergleichbar anwendungsbezogene Fehlerberechnung müsste geprüft werden, wie die Unterschiede in den einzelnen SH-Bändern vom Menschen im Kontext des PRT wahrnehmbar sind. Da hierzu keine Veröffentlichungen vorliegen wurde auf die Fehlerquadratsumme zurückgegriffen, die zwar eventuell nicht optimal ist, aber die grundlegenden Anforderungen an das Abstandsmaß erfüllt, d.h. für zwei identische Samples Null ist und ansonsten positive Werte zurück liefert. Konkret wurde die Abweichung der Fehler über alle n SH-Koeffizienten der Umgebung C_H und der Sonne C_S und die 3 Farbbänder summiert:

$$e = \sum_{i=1}^{n} \sum_{j=1}^{3} C_H(i,j)^2 + C_S(i,j)^2 \tag{4.7}$$

Der Fehler eines Clusters ergibt sich aus der Summe der Fehler an jedem Samplepunkt der Ausgangsdaten, die zu diesem Cluster gehören.

4.7 Echtzeit

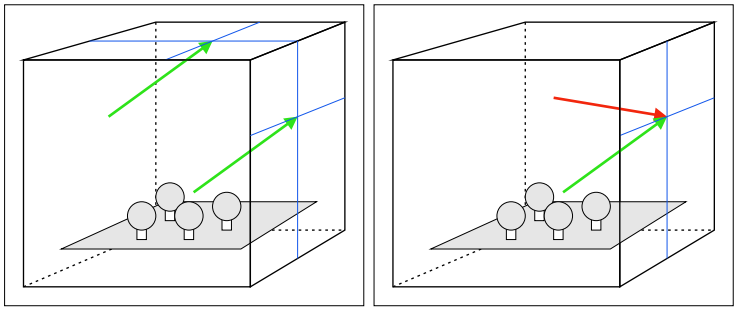
Nachdem die Vorberechnungen abgeschlossen sind kann die Szene interaktiv visualisiert werden. Dabei wird pro Frame zunächst der Himmel gerendert, da dieser den Hintergrund der gesamten Szene darstellt. Darauf folgt das Terrain und anschließend alle Pflanzen.

4.7.1 Himmel

Der Himmel besteht aus einer einfachen Skybox, d.h. einem Würfel, der die gesamte Szene umschließt und über eine Cube Map texturiert wird (Abb. 4.20). Zu beachten ist, dass die Seiten der Cube Map nicht den Seiten der Skybox-Geometrie entsprechen: Wäre dies der Fall, so würde nicht der Eindruck entstehen, dass der Himmel unendlich weit weg ist. Sobald der Betrachter sich bewegt, würde der Eindruck einer großen Halle entstehen, die mit Fototapete ausgekleidet ist. Durch die Cube Map ist jedoch egal, welchen Punkt der Skybox der Betrachter sieht, die Farbe wird ausschließlich durch seine Blickrichtung bestimmt.

4.7.2 Terrain

Für Terrain-Rendering gibt es etliche Methoden, um große Datenbestände in Echtzeit zu visualisieren, z.B. [Losasso und Hoppe 2004], [Duchaineau et al. 1997] und [Lindstrom und Pascucci 2001]. Sie zielen jedoch auf Datenbestände, die an die Speichergrenzen aktueller PCs stoßen ([Losasso und Hoppe 2004]: 40 GB Height Map). In üblichen



(a) (b)

Abbildung 4.20: (a) Für parallele Blickrichtungen gibt die Skybox denselben Wert zurück, unabhängig vom Schnittpunkt mit der geometrischen Repräsentation. (b) Somit kann ein Schnittpunkt unterschiedlich dargestellt werden, je nachdem, wo der Betrachter sich befindet.

Lenné3D-Szenen haben die Height Maps eine Auflösung von 1024^2 bis 4096^2, also nur rund 1 bis 16 MB, aber zu viel, um direkt trianguliert der Grafikkarte übergeben zu werden (2 bis 32 Millionen Dreiecke). Abhilfe schafft die in [Gerasimov et al. 2004] beschriebene Nutzung von Vertex-Texturen. Dabei wird ein Dreiecksnetz am Betrachter ausgerichtet aufgespannt und die Height Map als Textur übergeben. Im Vertexshader kann nun für jeden Vertex die passende Höhe aus dieser Textur ausgelesen werden (Abb. 4.21). Da das Netz relativ zum Betrachter ausgerichtet wird, kann die Auflösung entsprechend des Abstandes gewählt werden, so dass die Größe der Dreiecke aus Sicht des Betrachters annähernd konstant ist. Auf NVidia 6800 GPUs sind so laut [Gerasimov et al. 2004] 33 Millionen Vertices pro Sekunde möglich. Da für die meisten Lenné3D-Szenen etwa 40.000 Dreiecke ausreichen, ist der Anteil des Terrainrenderings an der Gesamtrenderzeit vernachlässigbar gering.

4.7.3 Pflanzen

Zur Visualisierung einer Pflanze wird zunächst eine zur Instanz passende Environment Map benötigt. Da der PRT relativ zum Modell errechnet wird, dieses aber um die Z-Achse rotiert dargestellt werden kann, muss die Environment Map entsprechend rotiert werden. [Green 2003] zeigt eine einfache Möglichkeit für Rotationen um eine Achse des Koordinatensystems.

Im Vertexshader wird der PRT P ausgewertet und das Ergebnis L_P für die Weiterverarbeitung im Fragmentshader in einem Interpolator abgelegt. Dazu müssen zuerst Vorder- und Rückseite des Vertex bestimmt werden (F,

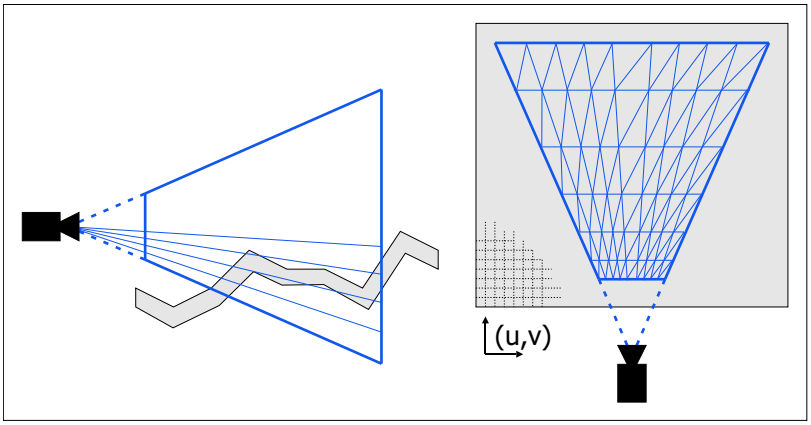


Abbildung 4.21: Das Terrain wird über ein relativ zur Kamera fixiertes Gitternetz dargestellt, dessen Vertices zur Laufzeit über die Height Map auf die richtige Höhe gesetzt werden.

B), da die Modelle beidseitige Beleuchtung erfordern. Anschließend wird auf jede Seite die Environment Map für Umgebungslicht C_H und indirekte Sonne C_S angewandt und das Ergebnis über die Transluzenz t gewichtet addiert (Abb. 4.22):

$$L_P = (P_{FH}C_H + P_{FS}C_S) \cdot (1 - t) + (P_{BH}C_H + P_{BH}C_S) \cdot t; \qquad (4.8)$$

Zu beachten ist, dass bei der Rückseite in beiden Fällen der PRT für Umgebungslicht benutzt wird. Grund hierfür ist, dass die direkte Beleuchtung nur für die Vorderseite errechnet wird, bei der Rückseite werden sämtliche Pfade (nicht nur die mehrfach reflektierten) über den PRT erfasst. Weiterhin werden im Vertexshader die Lichtquellen in den Tangent Space des Vertex transformiert, um das Normal Mapping zu vereinfachen.

Der Fragmentshader berechnet die Radiance der direkten Sonnenreflexion über einen Schlick-Shader, dessen zwei Ebenen dank der Vektoroperationen der GPU gleichzeitig verarbeitet werden können. Bei der Bestimmung der Materialeigenschaften der Oberfläche wird die Normalen-Verkürzung bei der Filterung der Normal Map berücksichtigt: Durch die Filterung einer stark variierenden Normal Map werden die dort gespeicherten Vektoren verkürzt, was in der Roughness der Oberfläche einfließen kann. Dies entspricht einem Transfer der makroskopischen Oberflächenstruktur in die mikroskopische, die durch die Materialparameter bestimmt wird, sobald die Darstellungsauflösung nicht mehr ausreicht, die Normal Map adäquat zu visualisieren. Dieser sogenannte Toksvig-Faktor wurde in [Toksvig 2004] für Phong-Exponenten eingeführt, lässt sich aber problemlos auf Schlick

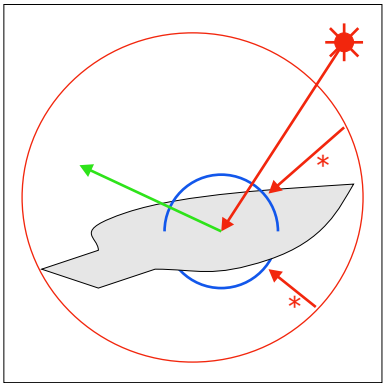


Abbildung 4.22: Die PRT-Auswertung im Vertexshader erfolgt sowohl für die Vorder- als auch für die Rückseite. Beide Terme werden entsprechend der Transluzenz gewichtet und zur direkten Reflexion addiert.

übertragen: Hier wird die Roughness mit der Länge des Normalenvektors multipliziert, was bedeutet, dass Oberflächen, deren Normalen nahe 180° stehen, als rein diffus interpretiert werden.

Eine weitere visuelle Verbesserung ermöglicht das Parallax Mapping aus [Welch 2004]: Die Höheninformationen, die zur Erstellung der Normal Map generiert wurden, können dazu benutzt werden, die Texturkoordinaten der Farbtextur anzupassen. Dies ermöglicht bei seitlichem Blick auf eine Oberfläche den Eindruck einer Höhenstruktur, ohne dass zusätzliche Geometrie benötigt wird.

4.7.4 High Dynamic Range

Da alle Shader mit Floatingpoint-Radiance-Werten L arbeiten, muss in einem letzten Schritt die Anpassung zur Darstellung auf dem Bildschirm erfolgen. Diese besteht aus zwei Schritten: Zunächst wird eine Sättigung entsprechend einem analogen Film simuliert (Abb. 4.25):

$$L_s = 1 - e^{-L*s} \tag{4.9}$$

Über s lässt sich die Empfindlichkeit steuern, die richtige Wahl dieses Wertes hängt von der dargestellten Szene ab. Da es bei High Dynamic Range-Rendering keinen absoluten Weißpunkt, d.h. keine maximale Helligkeit, mehr gibt, wird dieser Parameter benötigt, um per Hand den für den Betrachter relevanten Helligkeitsbereich festzulegen. In einem zweiten Schritt wird die Anpassung an die Gammakurve des Monitors vorgenommen (Abb. 3.3):

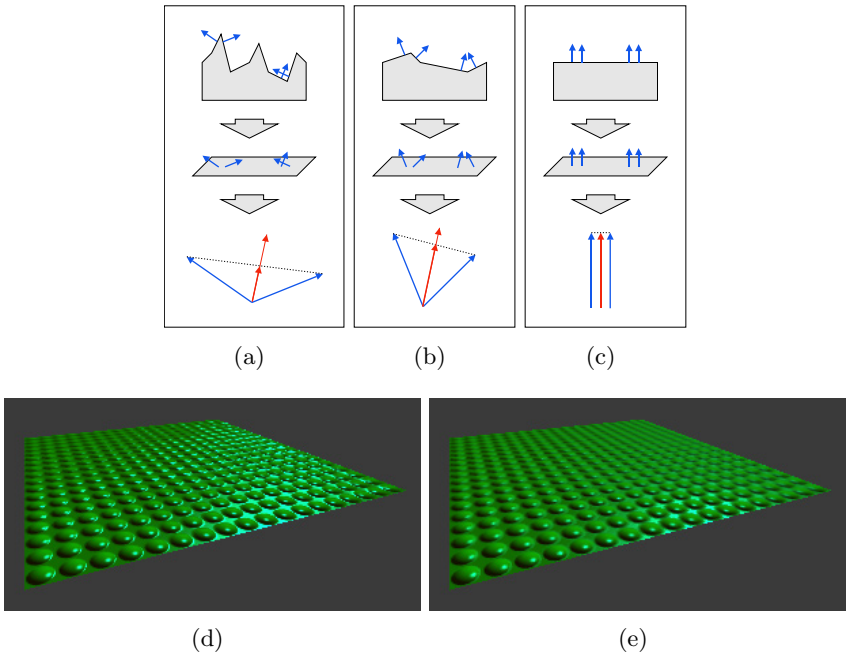


Abbildung 4.23: (a), (b), (c) Die Oberflächenbeschaffenheit kann durch Normal Maps kodiert werden, wenn die geometrische Auflösung nicht ausreicht. Werden diese allerdings beim verkleinerten Rendern gefiltert, verkürzen sich die Normalenvektoren. (d) Durch die Renormalisierung der Normlenvektoren ergibt sich ein Highlight-Aliasing, da die Oberflächenstrukturen zu klein werden. (e) Die nächstkleinere Auflösungsstufe der Oberflächenstruktur wird in den Materialeigenschaften (Schlick-Shader: Roughness) festgehalten. Durch Einbeziehung der Normalenverkürzung kann man das Aliasing vermeiden. (Quelle (d), (e): [Toksvig 2004])

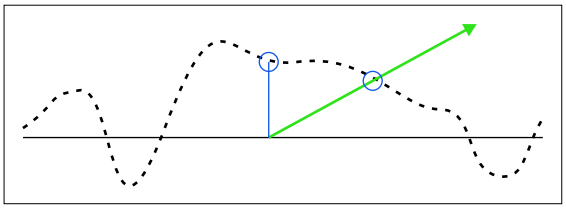


Abbildung 4.24: Die Objektoberfläche wird durch flache Polygone approximiert, wobei zusätzlich über eine Bump Map angegeben werden kann, wie die ursprünglichen Oberfläche vom Polygon in Normalenrichtung abweicht. Bei der Wahl der Farbtextur wird normalerweise nur das Polygon zur Positionsbestimmung herangezogen, obgleich der Schnittpunkt mit der ursprünglichen Oberfläche, je nach Winkel zwischen Polygon-Normale und Blickrichtung, eine deutlich abweichende Position haben kann. Dies lässt sich durch Parallax Mapping ausgleichen, wobei man die Texturkoordinaten entsprechend der Bump Map verschiebt, um den korrekten Schnittpunkt näher zu kommen.

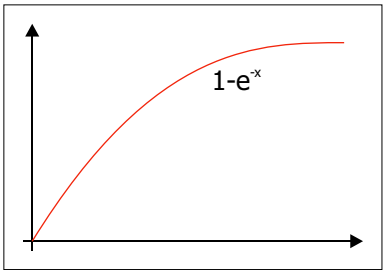


Abbildung 4.25: Um den großen Dynamikumfang auf die Möglichkeiten des Bildschirms zu reduzieren, wird auf die errechnete Radiance eine Sättigungskurve entsprechend einem analogen Film angewandt, da dies der gewohnten Wahrnehmung der Benutzer entspricht.

$$L_g = L_s^\gamma \tag{4.10}$$

Mit Hilfe dieser Einstellung können auch absichtlich dunkle oder helle Bildbereiche hervorgehoben werden, um die Darstellung dem Umgebungslicht anzupassen.

Kapitel 5

Implementierung

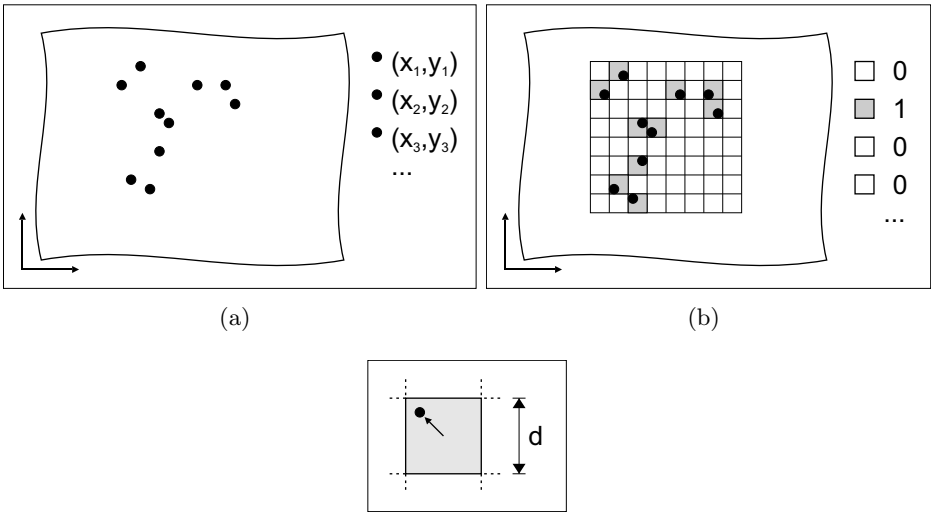
Die Implementierung der vorgestellten Techniken orientiert sich weitgehend an den Vorschlägen der zu Grunde liegenden Literatur und wird nicht näher ausgeführt. In diesem Kapitel vorgestellt werden daher primär die neu entwickelte Datenstruktur BitTree und die vielfältigen Varianten der eingesetzten KD-Trees. Auf eine kurze Diskussion der verwendeten Programmiersprachen folgt eine Übersicht über die eingebundenen Bibliotheken.

5.1 Datenstrukturen

Zwei Datenstrukturen sind von besonderer Bedeutung im Rahmen dieser Diplomarbeit: BitTrees als Möglichkeit, große Mengen Positionsdaten effizient zu speichern, und KD-Trees als Raumstruktur zur Beschleunigung von Suchalgorithmen.

5.1.1 BitTree

Die Idee hinter den BitTrees ist die Umwandlung von expliziten in implizite Positionsangaben (Abb. 5.1). Explizite Positionen werden durch eine Liste der individuellen Positionen charakterisiert. Diese kann eine beliebige Form annehmen, sei es eine einfache lineare Liste, ein KD-Tree oder eine andere dem jeweiligen Anwendungsgebiet abgepasste Struktur. Alle Varianten haben zunächst einen Mindestspeicherbedarf, der durch die Anzahl der gespeicherten Individuen bestimmt wird. Durch Ausnutzung von Korrelationen lässt sich der Bedarf pro Individuum senken, aber die Abhängigkeit von der Anzahl bleibt prinzipiell bestehen. Implizite Speicherung geht von einer Diskretisierung des Bereichs aus, in dem die Individuen angesiedelt sind (Area of Interest). Dabei wird die Elementgröße so gewählt, dass nie mehr als ein Individuum pro Element auftritt. Gespeichert wird nun eine



(a) (b)

(c)

Abbildung 5.1: (a) Explizite Speicherung erfolgt durch Auflistung der Individuen, (b) implizite durch Angabe, welche diskreten Elemente belegt sind. (c) Der Positionsfehler wird durch die Auflösung der Diskretisierung begrenzt.

Liste dieser diskreten Elemente, wobei ein Bit ausreicht, um festzuhalten, ob ein Element ein Individuum enthält oder nicht. Durch die Reihenfolge der Elemente im Datenstrom kann die Position der einzelnen Individuen rekonstruiert werden, wobei die räumliche Auflösung und damit der maximale Fehler durch eingangs gewählte Elementgröße bestimmt wird. Der Speicherbedarf der impliziten Speicherung hängt prinzipiell nur von der Diskretisierung ab, d.h. von der Anzahl der Elemente und nicht mehr von der Anzahl der Individuen. Damit lassen sich nun auch die Anwendungsbereiche expliziter und impliziter Speicherung skizzieren: Das Optimum der expliziten Repräsentation liegt bei Szenen mit sehr wenigen Individuen, da der Speicherbedarf unabhängig von der zu Grunde liegenden Fläche ist, wohingegen die implizite Speicherung ihr Optimum bei der vollständigen Abdeckung der Fläche mit Individuen hat.

Der Speicherbedarf der impliziten Darstellung für realistische Datensätze lässt sich senken, indem man Korrelationen der gespeicherten Bits ausnutzt, so dass man beispielsweise größere freie Bereiche nicht bis zur untersten Detailstufe auflöst. Der hier vorgestellte BitTree-Ansatz verwendet dafür eine Quadtree-Struktur, die ähnlich der impliziten Darstellung von Punkt-Geometrien in [Botsch et al. 2002] funktioniert: In jeder Hierarchie-Ebene wird das übergeordnete Element in n (z.B. $4, 16, 64$) Ele-

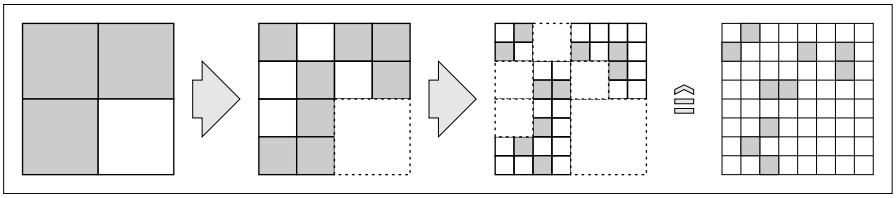


Abbildung 5.2: In der Hierarchie werden nur die Bereiche weiter aufgelöst, die mindestens ein Individuum beinhalten.

mente unterteilt, solange bis die gewünschte Auflösung erreicht ist. Dabei wird in einer Ebene gespeichert, in welchem der n Elemente Individuen vorhanden sind, so dass nur für diese weitere Unterebenen angelegt werden müssen. Auch dies wird über Bits kodiert, wobei 0 für vollständig freie Bereiche steht, 1 für teilweise oder ganz belegte. Dieser Kodierung liegt die Annahme zu Grunde, dass in der Praxis deutlich mehr Bereiche gar keine Individuen enthalten als dass sie vollständig bedeckt sind.

Der Overhead für den worst case, d.h. vollständig belegte Flächen, liegt für $n = 4$ bei 33%, bei größeren n oder nur teilweise bedeckten Flächen weniger. Damit ist die Einführung der Hierarchieebenen in impliziten Darstellungen in jedem Fall unproblematisch.

5.1.2 Kd-Tree

KD-Trees ([Samet 1990]) sind inzwischen eine Standard-Datenstruktur, wenn es um die räumliche Organisation von Objekten zwecks Beschleunigung einer anschließenden Suche geht. Interessant ist aber die Unterscheidung der verschiedenen Faktoren, die die konkrete Implementierung bestimmen:

5.1.2.1 Objekttypen

Der KD-Tree gibt die Dimension der in ihm gespeicherten Objekte nicht vor. Häufige Anwendung finden z.B. die Speicherung von Punkten (0-dimensional, z.B. in Photon Maps) und Flächen bzw. Körpern (2- bzw. 3-dimensional, z.B. beim Raytracing), prinzipiell spricht aber auch nichts gegen Linien (1-dimensional). Der Hauptunterschied zwischen der Speicherung von Punkten und sonstigen Objekten besteht in der Positionierung innerhalb des KD-Trees: Punkte sind immer genau einem Blatt zugeordnet, wohingegen Objekte mit einer gewissen Ausdehnung mehrere Blätter schneiden können.

Aus dieser Unterscheidung folgen verschiedene Möglichkeit zur konkreten Speicherung der Daten: Punkte können in inneren Knoten abgelegt

werden und durch ihre Position die Trennebenen festlegen ([Jensen 2001]), sie können in den Blättern direkt abgelegt werden oder in den Blättern können Referenzen auf die Punkte gespeichert werden. Letzteres ist jedoch unüblich, da ein Punkt nur genau einmal referenziert wird und damit meist unnötiger Overhead entsteht. Die Speicherung über Referenzen ist jedoch bei den anderen Datentypen gebräuchlich, da neben der Speicherersparnis auch die Kohärenz der Daten gewährleistet ist, sofern sich die Attribute der Objekte zur Laufzeit ändern können.

5.1.2.2 Suchtyp

Verschiedene Suchtypen können durch KD-Trees beschleunigt werden. Der einfachste ist die Punkt-Suche, bei der ein Punkt vorgegeben wird und alle Objekte, die diesen Punkt schneiden, zurückgegeben werden. Dieser Suchtyp passt auf alle Objekttypen, wobei das Ergebnis nie zwingend eindeutig ist: Mehrere Punkt-Objekte können an derselben Position gespeichert werden, andere Objekte können sich überlappen.

Die natürliche Erweiterung der Punkt-Suche ist die Nearest-Neighbour-Suche, die zu einem vorgegebenen Punkt die nächsten benachbarten Objekte zurückliefert (z.B. Photon Mapping). Dabei kann die Nachbarschaft durch verschiedene Kriterien limitiert werden, üblich sind die k nächsten Objekte oder die Objekte innerhalb eines Radius r. Beides ist für Punkt-Objekte trivial, für die anderen Typen muss zusätzlich das gewünschte Abstandsmaß definiert werden, also z.B. minimaler Abstand oder Abstand zum Objektmittelpunkt. Im Gegensatz zur Punkt-Suche, deren Ergebnis nach einem Baum-Durchlauf garantiert feststeht ($O(\log n)$), erfordert die Nearest-Neighbour-Suche je nach Datensatz ein Backtracking, um die Nachbarschaft auf weitere Blätter auszudehnen, und weißt somit im worst case eine Komplexität von $O(n)$ auf, falls alle Blätter besucht werden.

Beim Raytracing bildet die Suchbasis nicht ein Punkt sondern ein Strahl, wobei das Objekt gesucht wird, dass den Strahl schneidet und den vom Strahl-Ursprung aus gesehen nächsten Schnittpunkt bildet. Dabei wird zu jeder Trennebene eine Reihenfolge festgelegt, in der beide Seiten durchlaufen werden, um eine Terminierung beim ersten gefundenen und damit nächsten Schnittpunkt zu ermöglichen. Dabei müssen nicht alle Blätter zwangsweise durchlaufen werden, da man über den Schnittpunkt des Strahls mit einer Trennebene bestimmen kann, ob ganze Teilbäume übersprungen werden können.

Die Wahl der Trennebenen wirkt sich deutlich auf die Geschwindigkeit der Suche aus. Median-Trennebenen unterteilen die Objektmenge in jedem Schritt in zwei gleich große Teilmengen, so dass ein balancierter Binärbaum entsteht. Dies ist für die Punkt-Suche vorteilhaft, da hier nur ein Durch-

lauf von der Wurzel zu einem bestimmten Blatt erfolgt. [Havran 2000] zeigt aber, dass es deutlich bessere Möglichkeiten gibt, die Ebenen passen für die Raytracing-Strahl-Suche zu wählen, [Wald und Slusallek 2004] beschreibt ähnliches für Nearest-Neighbour-Suche zwecks Photon Mapping.

5.1.2.3 Speicherlayout

Die Beziehung eines Knotens zu seinen Kindern kann entweder explizit oder implizit festgehalten werden. Die explizite Darstellung erfolgt über einen oder mehrere Pointer, die entweder auf ein Paar oder jedes Kind einzeln zeigen. Je nach Objekttyp ist der Speicherbedarf für die Pointer unverhältnismäßig hoch, weshalb balancierte KD-Trees in Form eines Heaps gespeichert werden können, wobei ein Knoten an Position n Kinder an $2n$ und $2n + 1$ hat.

5.1.2.4 Flexibilität

Die Flexibilität ist ein drittes Kriterium, dass die Datenstruktur deutlich mitbestimmt. Dabei kann zwischen mehreren Fällen unterschieden werden: Statische KD-Trees werden einmal mit einem bestimmten Datensatz aufgebaut und danach nur ausgelesen. Dynamische KD-Trees ermöglichen entweder nur das Einfügen neuer Objekte oder zusätzlich auch das Löschen bereits vorhandener.

Die statische Variante trifft auf die meisten Anwendungsfälle zu. Sie erlaubt eine weitgehende Optimierung des Baums, da die benötigte Leistung leicht durch die gesparte Rechenzeit zur Laufzeit kompensiert werden kann. Dynamische KD-Trees, wie sie z.B. beim Clustering (Kap. 4.6.3.1) auftreten, sind deutlich komplexer und bringen weitere Einschränkungen für die Speicherung der Daten mit sich. Für KD-Trees sind im Gegensatz zu Binärbäumen (AVL, Red-Black) keine einfachen Rebalancierungen durch Rotationen bekannt. Daraus folgt, dass zur Laufzeit komplette Teilbäume neu aufgebaut werden müssen, wenn sie eine Balance-Toleranzgrenze überschreiten. Dynamische KD-Trees schließen ein Heap-Layout praktisch aus, da mit jeder Modifikation eine Rebalancierung notwendig wäre.

5.1.2.5 Abhängigkeiten

Die genannten Anforderungen sind teils abhängig, teils unabhängig voneinander. Die Suchalgorithmen funktionieren zwar prinzipiell mit wenigen Operationen, die jede KD-Tree-Form zur Verfügung stellen kann, so dass sie aus dieser Hinsicht unabhängig von der Datenstruktur selbst implementierbar wäre. Auf der anderen Seite hängt der optimale Aufbau aber deutlich

von der anschließenden Suchstrategie ab, so dass diese schon von vornherein bekannt sein sollte. Diese Abhängigkeiten erschweren eine saubere Aufteilung im Sinne des Policy Based Designs ([Alexandrescu 2001]). Zum Vergleich der verschiedenen Anforderungen und dem Nutzen der unterschiedlichen Optimierungsstrategien wurden im Rahmen der Diplomarbeit zahlreiche KD-Tree-Varianten implementiert, eine elegante Lösung bleibt aber dem Ausblick vorbehalten.

5.2 Sprachen

5.2.1 C++

Als Implementierungssprache wurde C++ ([Stroustrup 2000]) gewählt, wobei verschiedene Gründe eine Rolle spielen: Durch die maschinennahe Spezifikation des Speicherlayouts ist es möglich, Strukturen aus Klassen aufzubauen, die zu Lowlevel-APIs kompatibel bleiben. Ein Beispiel sind Texturdaten:

```
template<typename T, int N> Vector;
template<typename ColorT> class ColorImpl;
class ColorRgb : public ColorImpl<ColorRgb>
    { Vector<unsigned char,3> data; }
template<typename ColorT> class Texture
    { ColorT color[n]; }
```

Die Template-Klasse `Color` gibt einen Farbwert in Form eines Vektors (`Vector<>`) an, wobei die elementaren Operationen von einer davon unabhängigen Klasse `ColorImpl` geerbt werden. Die Textur selbst besteht aus einem Array aus diesen Farb-Instanzen, wobei der Pointer, der das Array repräsentiert, direkt dem OpenGL-(C-)API als Texturspeicher übergeben werden kann. Eine derartige direkte Kompatibilität wäre für Sprachen mit höherem Speichermanagement (z.B. Java) nur schwer realisierbar.

Ein weiterer Grund ist die fortgeschrittene Unterstützung generischer Programmierung, angefangen bei der STL ([Josuttis 1999]) über effiziente Mathematik ([Vandevoorde und Josuttis 2002]) bis hin zu ausformulierten Design Pattern ([Alexandrescu 2001]), die in anderen Lowlevel-Sprachen wie C so nicht gegeben ist.

Bedingt durch die weite Verbreitung der Sprache existiert eine Vielzahl Publikationen über effiziente und vor allem idiomatische Nutzung der Sprache, u.a. [Sutter 1999], [Sutter 2001],
[Sutter und Alexandrescu 2004], [Sutter 2004], [Meyers 1997],

[Meyers 1995], [Meyers 2001]. Für viele alltägliche Programmieraufgaben existieren damit Lösungen, die neben einer guten Kommunizierbarkeit des Codes auch oftmals eine optimale Nutzung der Fähigkeiten der Compiler ermöglichen.

5.2.2 GLSL

Da die Standard-OpenGL-Pipeline für die hier vorgestellten Shader nicht ausreicht, musste auf eine Sprache zur GPU-Programmierung zurückgegriffen werden. Da OpenGL als einziges portables 3D-API praktisch vorbestimmt war, standen NVidias Cg, die Vertex- und Fragment-Program-Extension sowie GLSL[1] zur Auswahl. Da GLSL als einzige Sprache in den OpenGL-Standard aufgenommen wurde ([Segal und Akeley 2004], ab Version 2.0) und von der Funktionalität her den anderen in nichts nachsteht, fiel die Wahl trotz der recht jungen Spezifikation entsprechend. GLSL wurde schnell von den führenden Grafikkartenherstellern unterstützt, zunächst als Extension, inzwischen auch von NVidia als Teil von OpenGL 2.0.

5.3 Bibliotheken

Bedingt durch die weite Verbreitung von C++ existiert auch eine große Anzahl Bibliotheken. Ohne diese wäre die Implementierung der Diplomarbeit nicht in so kurzer Zeit möglich gewesen, weshalb im folgenden die verwendeten Quellen kurz vorgestellt werden:

5.3.1 Boost

Die Boost C++ Bibliotheken[2] sind eine Sammlung verschiedenster portabler Bibliotheken, die über einen Peer-Review-Prozess kontrolliert werden und mit Hinblick auf optimale Zusammenarbeit untereinander und mit der STL entwickelt werden. Ein Ziel dabei ist es, Praxis für zukünftige C++-Standards zu schaffen. Hier verwendet wurden die Smart Pointer, Serialization, Filesystem, Random, Numeric Cast und Noncopyable.

5.3.2 DevIL

Die „Developer's Image Library“[3] stellt ein an OpenGL angelehntes Interface zum Laden von Bildern (hier: jpg, png) zur Verfügung. Zusätzlich existiert Basisfunktionalität zur Bildverarbeitung, z.B. Skalierung.

[1] http://www.opengl.org/documentation/oglsl.html
[2] http://www.boost.org/
[3] http://openil.sourceforge.net/

5.3.3 GLEW

Die verwendete OpenGL-Funktionalität wird prinzipiell vom OpenGL 2.0 Standard abgedeckt. Da unter Microsoft Windows aber nur OpenGL 1.1 nativ unterstützt wird, müssen die weiteren Funktionen (sowohl Extensions als auch Standard-Funktionen) über das Extension-Interface importiert werden. „The OpenGL Extension Wrangler Library“[4] erledigt diese Aufgabe plattformübergreifend auf Windows, Linux, Mac OSX etc.

5.3.4 Glut

Da nur komplett statische Szenen bei bewegtem Betrachter visualisiert werden, reicht ein minimalistisches Interface zu Testzwecken aus. Implementiert wurde mit Hilfe des „The OpenGL Utility Toolkit“[5] eine Anwendung mit einem OpenGL-Render-Fenster, die über Maus und Tastatur gesteuert werden kann.

5.3.5 Image Debugger

Übliche Debugger sind auf die Darstellung skalarer Werte und geschachtelten Datenstrukturen optimiert. Bei der Visualisierung anfallende Daten oft als zwei- oder mehrdimensionale Arrays anfallen, können diese nicht direkt überprüft werden. „The Image Debugger“[6] ermöglicht die Ausgabe von Bildern analog zur String-Ausgabe über `printf()`, so dass das Debuggen von bildverarbeitenden Algorithmen (wie z.B. die Erzeugung von Normal Maps) deutlich vereinfacht wird.

5.3.6 Lib3ds

Die nichtpflanzlichen 3D-Objekte in Lenné3D-Szenen liegen in Form von 3DS-Dateien vor. Trotz des Fehlens einer öffentlichen Spezifikation stellt Lib3ds[7] einen weitgehend vollständigen 3DS-Loader zur Verfügung, der die gewünschten Daten in Form von C-Strukturen zur Verfügung stellt.

5.3.7 Loki

Loki[8] ist die Referenzimplementierung der in [Alexandrescu 2001] vorgestellten Techniken. Design-Pattern wie die hier verwendeten Functor und

[4] http://glew.sourceforge.net/
[5] http://www.opengl.org/resources/libraries/glut.html
[6] http://www.cs.unc.edu/~baxter/projects/imdebug/
[7] http://lib3ds.sourceforge.net/
[8] http://www.moderncppdesign.com/

Singleton stehen in Form von Templates zur Verfügung, darüber hinaus gibt es z.B. mit dem Small Object Allocator aber auch Routinen zur Optimierung des Speichermanagements und ähnliche hilfreiche Funktionen.

5.3.8 OpenEXR

Gleitkommawerte nach IEEE754 stehen üblicherweise in 32 Bit und 64 Bit Breite zur Verfügung. Um im Gegensatz zu 16 Bit Integer oder Festkomma-Darstellungen dieselbe relative Genauigkeit zu erreichen, unterstützen aktuelle Grafikkarten 16 Bit Gleitkommawerte für Texturen und Framebuffer. Um diesen Datentyp in C++ nutzen zu können, wird in OpenEXR[9] eine Klasse zur Verfügung gestellt, die analog zu den eingebauten Datentypen verwendet werden kann.

5.3.9 OpenGL

OpenGL[10] ist das einzige weit verbreitete plattformunabhängige Rendering-API. Da die Software primär auf einfachen Windows-PCs laufen soll, kamen prinzipiell nur OpenGL und Direct3D in Frage. Die Funktionalität beider API vergleichbar ist, Direct3D aber nur auf Windows zur Verfügung steht, wurde zu Gunsten der Flexibilität OpenGL gewählt.

5.3.10 OpenMesh

OpenMesh[11] stellt als Grundlage der Polygonnetz-Datenstrukturen ein optimierte Vektorklasse zur Verfügung, die über Loop-unrolling compilerfreundlichen Code erzeugt. Im Gegensatz zur Template-Spezialisierung in [Vandevoorde und Josuttis 2002] wird mit Macros gearbeitet, die aber für Anwender der Bibliothek unsichtbar bleiben.

[9] http://www.openexr.org/
[10] http://www.opengl.org/
[11] http://www.openmesh.org/

Kapitel 6

Resultate

Die Ergebnisse der Diplomarbeit werden in diesem Kapitel vorgestellt, wobei zunächst die visuelle Qualität an Hand von Bildschirmfotos illustriert wird. Im nachfolgenden Abschnitt wird eine Übersicht über die erzielte Geschwindigkeit gegeben.

6.1 Qualität

Die nachfolgenden Bildschirmfotos wurden auf einer NVidia Geforce 6800 in einer Bildschirmauflösung von 1280×1024 Pixeln erstellt. Die Szene besteht aus rund 100 manuell positionierten Individuen in einer imaginären Landschaft. Da kein Level-of-Detail-Verfahren zum Einsatz kommt musste diese niedrige Anzahl gewählt werden, um interaktive Darstellung zu ermöglichen. Zum Vergleich wird jeweils die Standard-Beleuchtung und die neu entwickelte gezeigt.

6.2 Geschwindigkeit

Die im vorherigen Kapitel gezeigte Szene lässt sich mit rund 4 bis 15 fps darstellen. Da außer View-Frustum-Culling keine weitere Optimierung zum Einsatz kommt (Level-of-Detail o.ä.) werden dabei alle bis zu 100 Individuen in höchster Qualität dargestellt. Eine minimale Szene wie in Abb. 6.8 wird mit durchschnittlich über 150 fps gerendert. Die Darstellung einzelner Individuen in hoher Qualität im Rahmen eines Level-Of-Detail-Schemas sollte also nicht zu einem Flaschenhals werden, womit die ursprüngliche Anforderung erfüllt ist.

Die Vorberechnungszeiten sind stark von der gewünschten visuellen Qualität abhängig. Die Berechnung des PRT für den Baum in Abb. 6.8

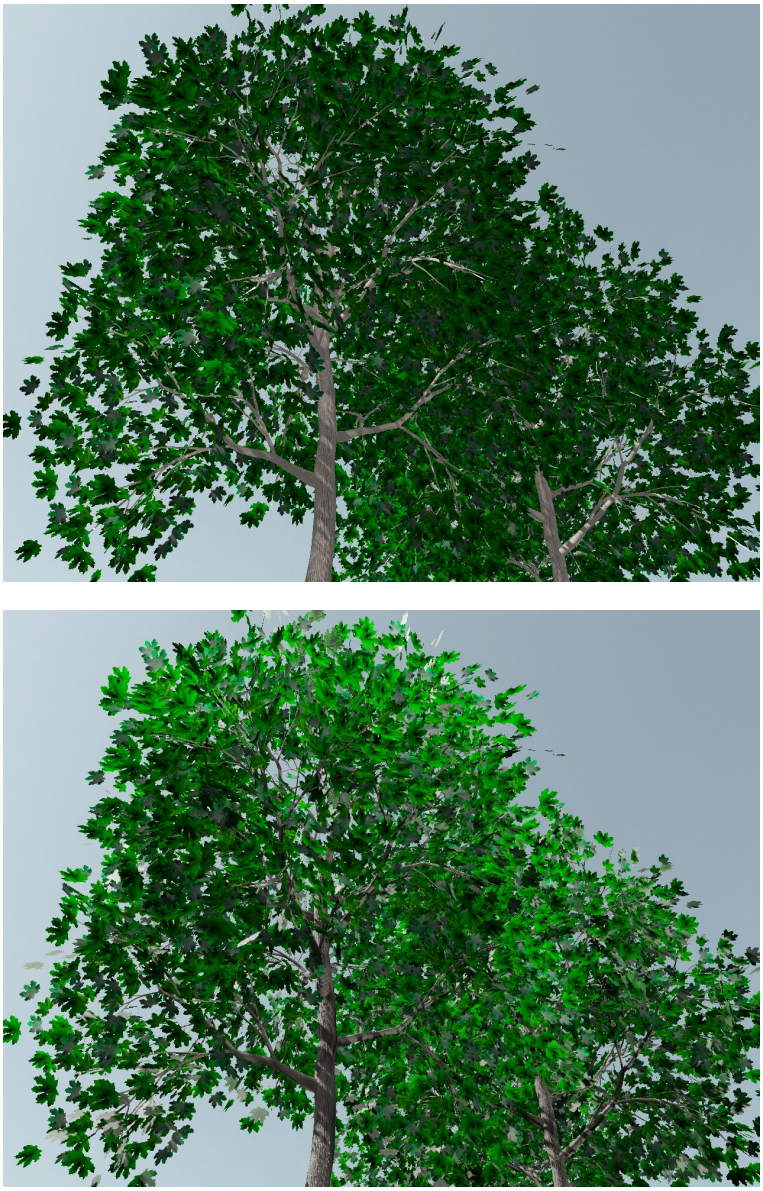


Abbildung 6.1: Der gezeigte Baum wird aus einem Material und zwei Farbtexturen aufgebaut, alle weiteren Effekte sind berechnet. Deutlich sichtbar ist der direkte Schattenwurf durch die Shadow Map auf dem Stamm, subtiler ist die variierende Helligkeit der Blätter je nach reflektiertem Umgebungslicht.

Abbildung 6.2: Blattober- und -unterseite haben dieselben Texturen, werden aber aufgrund unterschiedlicher Beleuchtung und Transluzenz verschieden dargestellt. An der linken Brennnessel sind die niederfrequenten Helligkeitsverläufe bedingt durch die diffuse Transluzenz gut sichtbar, wohingegen die scharfen Schatten der direkten Reflexion auf dem rechten Exemplar vordergründig sind.

Abbildung 6.3: Je nach Blickwinkel und Sonnenstand werden die glossy Reflexionen deutlicher. Im Gegensatz zu Abb. 6.2 zeigt sich wieder ein anderes Farb- und Helligkeitsspektrum, die Oberflächenstrukturen sind stärker betont.

Abbildung 6.4: Die Effekte der Selbstabschattung und der Mehrfachreflexion treten bei komplexen Individuen in den Vordergrund. Auch dieses Modell arbeitet mit nur einer Blatt-Textur und einer Materialdefinition, die Unterschiede in der Helligkeit der Blätter ergibt sich durch den PRT.

Abbildung 6.5: Die Transluzenz der Blätter wird bei einem Blick von unten durch einen Busch gegen die Sonne deutlich: Obgleich auf die Unterseite der Blätter kaum Umgebungslicht fällt, erscheint sie vergleichsweise hell.

Abbildung 6.6: Die Abschattung verschiedener Objekte untereinander wird über die Environment Map berücksichtigt: Die Bäume schatten die kleineren Individuen rechts im Bild stark ab, wohingegen die linken frei stehen und dementsprechend heller erscheinen.

Abbildung 6.7: Bei der direkten Beleuchtung wird die Normal Map angewandt, was sich in der Rindenstruktur widerspiegelt. Die nur diffus von der Umgebung ausgeleuchteten Stellen erscheinen bei vergleichbarer Helligkeit deutlich weniger ausgeprägt.

Abbildung 6.8: Eine minimale Szene mit nur einem Baum wird in 800×1100 Pixeln mit über 150 fps gerendert.

benötigt rund 10 Stunden auf einem Pentium 4 mit 3 GHz, wobei die meisten Modelle aber deutlich einfacher sind und dementsprechend weniger Rechenzeit in Anspruch nehmen. Die Erstellung der Environment Maps erfolgt mehrstufig, wobei zunächst eine Photon Map für die gesamte Szene aufgebaut wird, danach eine Environment Map für jedes Individuum und abschließend das Environment Map Clustering. Der Anteil an der Gesamtzeit hängt stark von der Szene ab, da die Photon Map hauptsächlich von der Anzahl Photonen bestimmt wird und nur bedingt von der Anzahl Individuen. Die Environment Maps werden aber pro Individuum berechnet, eine direkte Abhängigkeit von der Photon Map-Komplexität besteht nicht. Die im vorherigen Kapitel gezeigten Bilder erforderten eine Gesamtzeit für die Berechnung der Szene von rund 30 Minuten mit der Photon Map-Erstellung als wichtigsten Faktor.

Kapitel 7

Fazit

7.1 Zusammenfassung

Ziel der Diplomarbeit war es, die Visualisierung von Pflanzen im Nahbereich bei der Darstellung von Landschaften zu verbessern. Bedingt durch die fortgeschrittene Level-of-Detail-Entwicklung steht dabei für wenige Pflanzen im Vordergrund vergleichsweise viel Rechenzeit zur Verfügung. Dadurch war es möglich, verschiedene aktuelle Forschungsergebnisse der Computergrafik zu kombinieren: Programmierbare Shader und Shadow Maps für die direkte Beleuchtung, Precomputed Radiance Transfer für die indirekte Beleuchtung und das neu entworfene Environment Map Clustering zur Verarbeitung großer Szenen, von denen immer nur ein Ausschnitt über die vorgestellte Technik gerendert wird.

Die Aufteilung in verschiedene Beleuchtungskomponenten wurde formal auf die Pfad-Notation des Monte-Carlo-Path-Tracing gestützt. Dadurch konnte sichergestellt werden, dass die Kombination physikalisch plausibel bleibt und eine fundierte Approximation der Rendering Equation darstellt.

Der pragmatische Teil der Arbeit beschäftigte sich mit der Rekonstruktion physikalisch auswertbarer Objekteigenschaften aus einem Satz gegebener 3D-Modelle, Materialien und Texturen. Diese Daten wurden dabei als Anhaltswerte zur Parametrisierung eines biologisch und physikalisch basierten Materialmodells genommen, um eine visuell ansprechende Darstellung zu ermöglichen.

7.2 Ausblick

Die vorgestellte Lösung kann in vielerlei Hinsicht verbessert werden. Der vom Prozess her naheliegendste Punkt ist die Erweiterung der Ausgangsdaten um physikalisch basierte und vollständige Materialdefinitionen. Da-

durch könnte die gesamte Heuristik, die derzeit OpenGL-Shader in Schlick-Shader umrechnet sowie Normal Maps aus Farbtexturen erstellt, wegfallen. Dadurch wäre nicht nur eine Verbesserung des Realismus sondern insbesondere eine Vergrößerung der Vielfalt möglich, da die hier eingeführten Materialien deutlich flexibler und trotzdem intuitiv parametrisierbar sind. Selbst wenn die Materialeigenschaften nicht auf Messwerte zurückgeführt werden, würde eine direkte Kontrolle durch den Modellierer die Ergebnisse deutlich verbessern können. Hinsichtlich der Modellierung wäre auch eine weitere Untersuchung der Alpha-Texturen sinnvoll, um den optimalen Mittelweg zwischen Geometriemenge und Überschneidungsfreiheit zu finden.

Ausgehend von der Zielsetzung ist der nächste wichtige Schritt, eine stufenlose Integration ein Level-of-Detail-System zu erarbeiten. Solange die niedrigeren Detailstufen auf die alten Beleuchtungsmodelle zurückgreifen wird der Übergang deutlich sichtbar sein, auf der anderen Seite soll ja gerade der Detailgrad reduziert werden.

Die meisten Approximationen bei der Implementierung wurden ausgehend einer groben Abschätzung des visuellen Ergebnisses getroffen, so z.B. die Integration der PRT-Koeffizienten für Sonne und Umgebung in einen Datensatz plus Korrekturfaktor, das Fehlermaß beim Environment Map Clustering, der Tiefpass und die Anzahl Samples bei der Multiquadric-Interpolation der Environment Maps, die Anzahl Photonensamples beim Photon Mapping, die Vergrößerung des Sonnenradius zur Kompensation verschiedener Abbildungsfehler etc.. [Havran 2000] und [Wald 2004] zeigen speziell für Raytracing, wie die Wahl solcher Implementierungsdetails einen signifikanten Einfluss auf die erzielten Ergebnisse haben kann, sowohl von der Geschwindigkeit als auch von der Qualität her. Analog wäre hier eine systematische Überprüfung notwendig, die nicht nur die Theorie sondern auch die Implementierung auf eine saubere Basis stellt.

Seitens der Softwaretechnik wäre einer der wichtigsten Schritte die Entwicklung eines flexiblen KD-Trees, um nicht etliche unterschiedliche Varianten im selben Projekt pflegen zu müssen. Eine Aufteilung in elementare Methoden und Hilfsfunktionen ähnlich wie in [Sutter 2004] für `std::string` beschrieben sowie eine orthogonale Zerlegung der Funktionalität analog zum Policy Based Design in [Alexandrescu 2001] würde die Implementierung deutlich vereinfachen.

7.3 Dank

Ich danke meinem Betreuer Hans-Christian Hege (ZIB[1]) für die mannigfaltige Unterstützung und die fortwährende Motivation, die einen wesent-

[1] Zuse-Institut Berlin, http://www.zib.de

lichen Beitrag zu Realisierung dieser Diplomarbeit in diesem Umfang über die vergangenen sechs Monate hatte.

Prof. Heinz U. Lemke (TU Berlin[2]) danke ich für die Flexibilität und die Möglichkeit, trotz eines Mangels an Lehrkräften Computergrafik als Studienschwerpunkt wählen zu können.

Philip Paar (ZIB) hat aus Anwendersicht dazu beigetragen, dass die Arbeit trotz wissenschaftlicher Basis nicht an der Realität der Landschaftsplaner vorbei entwickelt wurde. Seine Koordination des Lenné3D-Projekts ermöglichte eine reibungslose Zusammenarbeit mit Wieland Röhricht (ZALF[3]), Jan Walter Schliep[4] und Carsten Colditz (Uni Konstanz[5]), die mir u.a. unkompliziert einen ausmodellierten Baum ohne Alpha-Texturen zur Verfügung stellten.

Der BitTree geht auf eine Idee von Hans-Christian Hege und Liviu Coconu (ZIB, Uni Konstanz) zurück.

Prof. Leif Kobbelt (RWTH Aachen[6]) danke ich für die exzellente Einführung in die akademische Computergrafik, die die Wahl meines Studienschwerpunkts maßgeblich beeinflusst hat.

[2] Technische Universität Berlin, http://www.tu-berlin.de

[3] Leibniz-Zentrums für Agrarlandschafts- und Landnutzungsforschung, http://www.zalf.de

[4] http://www.wallis-eck.de/

[5] Universität Konstanz, http://www.uni-konstanz.de/

[6] Rheinisch-Westfälische Technische Hochschule Aachen, http://www.rwth-aachen.de/

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Akenine-Möller 2001] Akenine-Möller, Tomas (2001). *Fast 3D Triangle-Box Overlap Testing*. Journal of graphics tools, 6(1):29–33. 56

[Akenine-Moeller et al. 2004] Akenine-Moeller, Tomas, E. Chan, W. Heidrich, J. Kautz, M. Kilgard und M. Stamminger (2004). *Real-Time Shadowing Techniques*. Siggraph Course Notes. 51

[Alexandrescu 2001] Alexandrescu, Andrei (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in Depth. Addison-Wesley Professional. 77, 79, 92

[Annen et al. 2004] Annen, Thomas, J. Kautz, F. Durand und H.-P. Seidel (2004). *Spherical Harmonic Gradients for Mid-Range Illumination*. In: *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*, S. 331–336. 48

[Arvo et al. 2001] Arvo, James, M. Fajardo, P. Hanrahan, H. W. Jensen, D. Mitchell, M. Pharr und P. Shirley (2001). *State of the Art in Monte Carlo Ray Tracing for Realistic Image Synthesis*. Siggraph 2001 Course Notes. 15

[Blythe et al. 1999] Blythe, David, B. Grantham, T. McReynolds und S. R. Nelson (1999). *Advanced Graphics Programming Techniques Using OpenGL*. Siggraph Course Notes. 4, 46

[Botsch et al. 2002] Botsch, Mario, A. Wiratanaya und L. Kobbelt (2002). *Efficient High Quality Rendering of Point Sampled Geometry*. In: *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, S. 53–64. 73

[Coconu und Hege 2002] Coconu, Liviu und H.-C. Hege (2002). *Hardware-Accelerated Point-Based Rendering of Complex Scenes*. In: *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, S. 43–52. 3

[Debevec 1998] Debevec, Paul (1998). *Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics With Global Illumination and High Dynamic Range Photography*. In: *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, S. 189–198. 36, 37, 45

[Debevec und Malik 1997] Debevec, Paul E. und J. Malik (1997). *Recovering High Dynamic Range Radiance Maps from Photographs*. In: *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, S. 369–378. 36

[Deussen et al. 2002] Deussen, Oliver, C. Colditz, M. Stamminger und G. Drettakis (2002). *Interactive visualization of complex plant ecosystems*. In: *VIS '02: Proceedings of the conference on Visualization '02*, S. 219–226, Washington, DC, USA. IEEE Computer Society. 3, 26

[Deussen et al. 1998] Deussen, Oliver, P. M. Hanrahan, B. Lintermann, R. Mech, M. Pharr und P. Prusinkiewicz (1998). *Realistic Modeling and Rendering of Plant Ecosystems*. In: *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, S. 275–286. 3

[Duchaineau et al. 1997] Duchaineau, Mark A., M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich und M. B. Mineev-Weinstein (1997). *ROAMing Terrain: Real-time Optimally Adapting Meshes*. In: *IEEE Visualization '97*, S. 81–88. 65

[Everitt und Kilgard 2002] Everitt, Cass und M. J. Kilgard (2002). *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*. nVidia Whitepaper. 55

[Everitt et al. 2001] Everitt, Cass, A. Rege und C. Cebenoyan (2001). *Hardware Shadow Mapping*. nVidia Whitepaper. 51

[Franzke 2004] Franzke, Oliver (2004). *Real-Time Global Illumination Of Semi-Static Objects In Static Environments Using Spherical Harmonics*. http://www.oliver-franzke.de/. 64

[Franzke und Deussen 2003] Franzke, Oliver und O. Deussen (2003). *Fast and Accurate Graphical Representation of Plant Leaves*. In: *International Symposium on Plant Growth Model, Simulation, Visualization and Applications*. 3, 28, 30

[Fussell und Subramanian 1988] Fussell, Donald S. und K. R. Subramanian (1988). *Fast ray tracing using k-d trees*. Technischer Bericht CS-TR-88-07, University of Texas, Austin. 56

[Gerasimov et al. 2004] Gerasimov, Philipp, R. Fernando und S. Green (2004). *Using Vertex Textures*. nVidia Whitepaper. 66

[Graham 1972] Graham, RL (1972). *An efficient algorithm for determining the convex hull of a finite planar set*. Information Processing Letters. 54

[Green 2003] Green, Robert (2003). *Spherical Harmonic Lighting: The Gritty Details*. In: *Game Developers Conference 2003*. 20, 21, 48, 66

[Hardy 1971] Hardy, R.L. (1971). *Multiquadric equations of topography and other irregular surfaces*. Journal of Geophysical Research, 76:1906–1915. 64

[Havran 2000] Havran, Vlastimil (2000). *Heuristic Ray Shooting Algorithms*. Ph.D. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague. 56, 76, 92

[Heckbert 1989] Heckbert, Paul S. (1989). *Fundamentals of Texture Mapping and Image Warping*. Diplomarbeit, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley. 54

[Heckel et al. 1999] Heckel, Bjoern, G. H. Weber, B. Hamann und K. I. Joy (1999). *Construction of Vector Field Hierarchies*. In: *IEEE Visualization '99*, S. 19–26. 63, 64

[Hotelling 1933] Hotelling, H. (1933). *Analysis of a complex of statistical variables into principal components*. Journal of Educational Psychology, 24/25:417–441/498–520. 63

[Jensen 2001] Jensen, Henrik Wann (2001). *Realistic Image Synthesis Using Photon Mapping*. AK Peters. 4, 44, 75

[Jensen et al. 2001] Jensen, Henrik Wann, S. R. Marschner, M. Levoy und P. Hanrahan (2001). *A Practical Model for Subsurface Light Transport*. In: *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, S. 511–518. 28

[Josuttis 1999] Josuttis, Nicolai M. (1999). *The C++ Standard Library : A Tutorial and Reference*. Addison-Wesley Professional. 77

[Kajiya 1986] Kajiya, James T. (1986). *The Rendering Equation*. In: *Computer Graphics (Proceedings of SIGGRAPH 86)*, Bd. 20, S. 143–150. 9

[Kautz et al. 2000] Kautz, Jan, P.-P. Vázquez, W. Heidrich, H.-P. Seidel, J. Kautz, P.-P. Vázquez, W. Heidrich und H.-P. Seidel (2000). *A Unified Approach to Prefiltered Environment Maps*. In: *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, S. 185–196. 46

[Kozlov 2004] Kozlov, Simon (2004). *GPU Gems*, Kap. Perspective Shadow Maps: Care and Feeding, S. 217–244. Addison-Wesley Professional. 53, 54

[Lafortune und Willems 1994] Lafortune, Eric P. und Y. D. Willems (1994). *Using the Modified Phong Reflectance Model for Physically Based Rendering*. Technischer Bericht, Department of Computer Science, K. U. Leuven. 27

[Lewis 1993] Lewis, Robert (1993). *Making Shaders More Physically Plausible*. In: *Fourth Eurographics Workshop on Rendering*, S. 47–62. 3, 27

[Lindstrom und Pascucci 2001] Lindstrom, P. und V. Pascucci (2001). *Visualization of large terrains made easy*. In: *IEEE Visualization 2001*, S. 363–370. 65

[Lintermann und Deussen 1998] Lintermann, B. und O. Deussen (1998). *A Modelling Method and User Interface for Creating Plants*. Computer Graphics Forum, 17(1):73–82. 23

[Losasso und Hoppe 2004] Losasso, Frank und H. Hoppe (2004). *Geometry clipmaps: terrain rendering using nested regular grids*. ACM Transactions on Graphics, 23(3):769–776. 65

[Martin und Tan 2004] Martin, Tobias und T.-S. Tan (2004). *Antialiasing and Continuity with Trapezoidal Shadow Maps*. In: *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*, S. 153–160. 53

[Meyer et al. 2002] Meyer, Mark, M. Desbrun, P. Schroeder und A. Barr (2002). *Discrete Differential Geometry Operators for Triangulated 2-Manifolds*. In: *VisMath Proceeding*. 27

[Meyers 1995] Meyers, Scott (1995). *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Professional. 78

[Meyers 1997] Meyers, Scott (1997). *Effective C++: 50 Specific Ways to Improve Your Programs and Design*. Addison-Wesley Professional, 2nd Aufl. 77

[Meyers 2001] Meyers, Scott (2001). *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Professional. 78

[Möller und Trumbore 1997] Möller, Tomas und B. Trumbore (1997). *Fast, Minimum Storage Ray-Triangle Intersection*. Journal of graphics tools, 2(1):21–28. 58

[Musgrave 1989] Musgrave, F. Kenton (1989). *Grid Tracing: Fast Ray Tracing for Height Fields*. Yale Dept. of Computer Science Research Report YALEU/DCS/RR-639, Yale University. 59

[Ng et al. 2003] Ng, Ren, R. Ramamoorthi und P. Hanrahan (2003). *All-Frequency Shadows Using Non-linear Wavelet Lighting Approximation*. ACM Transactions on Graphics, 22(3):376–381. 17, 47, 48

[Nielsen 2003] Nielsen, Ralf Stokholm (2003). *Real Time Rendering of Atmospheric Scattering Effects for Flight Simulators*. Diplomarbeit, Technical University of Denmark. 37

[Pauly et al. 2002] Pauly, Mark, M. Gross und L. P. Kobbelt (2002). *Efficient Simplification of Point-Sampled Surfaces*. In: *VIS '02: Proceedings of the conference on Visualization '02*, Washington, DC, USA. IEEE Computer Society. 62, 63

[Pharr und Humphreys 2004] Pharr, Matt und G. Humphreys (2004). *Physically Based Rendering : From Theory to Implementation*. The Interactive 3d Technology Series. Morgan Kaufmann. 14

[Preetham et al. 1999] Preetham, A. J., P. S. Shirley und B. E. Smits (1999). *A Practical Analytic Model for Daylight*. In: *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, S. 91–100. 37

[Press et al. 1992] Press, William H., S. A. Teukolsky, W. T. Vetterling und B. P. Flannery (1992). *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press. 63

[Qu et al. 2003] Qu, Huamin, F. Qiu, N. Zhang, A. Kaufman und M. Wan (2003). *Ray Tracing Height Fields*. In: *Computer Graphics International*. Computer Graphics Society. 59

[Ramamoorthi und Hanrahan 2001] Ramamoorthi, Ravi und P. Hanrahan (2001). *An Efficient Representation for Irradiance Environment Maps*. In: *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, S. 497–500. 46, 48

[Samet 1990] Samet, Hanan (1990). *The design and analysis of spatial data structures*. Addison Wesley. 59, 74

[Schlick 1993] Schlick, Christophe (1993). *A Customizable Reflectance Model for Everyday Rendering*. In: *Fourth Eurographics Workshop on Rendering*, S. 73–84. 28

[Schröder und Sweldens 1995] Schröder, Peter und W. Sweldens (1995). *Spherical wavelets: Texture processing*. In: Hanrahan, P. und W. Purgathofer, Hrsg.: *Rendering Techniques '95*. Springer Verlag, Wien, New York. 18

[Schröder und Sweldens 1995] Schröder, Peter und W. Sweldens (1995). *Spherical Wavelets: Efficiently Representing Functions on the Sphere*. S. 161–172. 18

[Segal und Akeley 2004] Segal, Mark und K. Akeley (2004). *The OpenGL Graphics System: A Specification, Version 2.0*. http://www.opengl.org. 17, 78

[Sillion et al. 1991] Sillion, François X., J. R. Arvo, S. H. Westin und D. P. Greenberg (1991). *A Global Illumination Solution for General Reflectance Distributions*. In: *Computer Graphics (Proceedings of SIGGRAPH 91)*, Bd. 25, S. 187–196. 18

[Sloan et al. 2003a] Sloan, Peter-Pike, J. Hall, J. Hart und J. Snyder (2003a). *Clustered Principal Components for Precomputed Radiance Transfer*. ACM Transactions on Graphics, 22(3):382–391. 48

[Sloan et al. 2002] Sloan, Peter-Pike, J. Kautz und J. Snyder (2002). *Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments*. ACM Transactions on Graphics, 21(3):527–536. 47, 48

[Sloan et al. 2003b] Sloan, Peter-Pike, X. Liu, H.-Y. Shum und J. Snyder (2003b). *Bi-Scale Radiance Transfer*. ACM Transactions on Graphics, 22(3):370–375. 48

[Stamminger und Drettakis 2002] Stamminger, Marc und G. Drettakis (2002). *Perspective Shadow Maps*. ACM Transactions on Graphics, 21(3):557–562. 53

[Stroustrup 2000] Stroustrup, Bjarne (2000). *The C++ Programming Language*. Addison-Wesley Professional, Special 3rd Aufl. 77

[Stumpfel et al. 2004] Stumpfel, Jessi, C. Tchou, A. Jones, T. Hawkins, A. Wenger und P. Debevec (2004). *Direct HDR capture of the sun and sky*. In: *AFRIGRAPH '04*, S. 145–149, New York, NY, USA. ACM Press. 36

[Sutter 1999] Sutter, Herb (1999). *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. C++ in Depth. Addison-Wesley Professional. 77

[Sutter 2001] Sutter, Herb (2001). *More Exceptional C++*. C++ in Depth. Addison-Wesley Professional. 77

[Sutter 2004] Sutter, Herb (2004). *Exceptional C++ Style : 40 New Engineering Puzzles, Programming Problems, and Solutions*. C++ in Depth. Addison-Wesley Professional. 77, 92

[Sutter und Alexandrescu 2004] Sutter, Herb und A. Alexandrescu (2004). *C++ Coding Standards : 101 Rules, Guidelines, and Best Practices*. C++ in Depth. Addison-Wesley Professional. 77

[Toksvig 2004] Toksvig, Michael (2004). *Mipmapping Normal Maps*. nVidia Whitepaper. 67, 69

[Vandevoorde und Josuttis 2002] Vandevoorde, David und N. M. Josuttis (2002). *C++ Templates: The Complete Guide*. Addison-Wesley Professional. 77, 80

[Voorhies und Foran 1994] Voorhies, Douglas und J. Foran (1994). *Reflection Vector Shading Hardware*. In: *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, S. 163–166. 46

[Wald 2004] Wald, Ingo (2004). *Realtime Ray Tracing and Interactive Global Illumination*. Doktorarbeit, Computer Graphics Group, Saarland University. Available at http://www.mpi-sb.mpg.de/~wald/PhD/. 58, 92

[Wald und Slusallek 2004] Wald, Ingo und J. G. P. Slusallek (2004). *Balancing Considered Harmful - Faster Photon Mapping using the Voxel Volume Heuristic*. Computer Graphics Forum, 23(3):595–603. 44, 76

[Watt 1999] Watt, Alan H. (1999). *3D Computer Graphics*. Addison-Wesley Professional, 3rd Aufl. 4

[Welch 2004] Welch, Terry (2004). *Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces*. Infinispace Research and Development. 68

[Willmott und Heckbert 1997] Willmott, Andrew J. und P. S. Heckbert (1997). *An Empirical Comparison of Radiosity Algorithms*. Technischer Bericht CMU-CS-97-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3890. 12

[Wimmer et al. 2004] Wimmer, Michael, D. Scherzer und W. Purgathofer (2004). *Light Space Perspective Shadow Maps*. In: *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*, S. 143–152. 53