Humboldt- Universität zu Berlin

Mathematisch-Naturwissenschaftliche Fakultät II

Institut für Informatik

Lehrstuhl für Künstliche Intelligenz

3DView2: eine dreidimensionale Visualisierungs- und Steuerungskomponente für die MicroPsi-Multiagentenplattform.

Wissenschaftliche Arbeit zur Erlangung des

Grades eines Diplom-Informatikers

vorgelegt von

David Salz (me@davidsalz.de)

Betreuer: **Dipl. Inf. Joscha Bach**

Gutachter: **Prof. Dr. Hans-Dieter Burkhard**

Facts mean nothing when they are preempted by appearence.

Do not underestimate the power of impression over reality.

From *Prelude to Dune: House Harkonnen*
by Brian Herbert and Kevin J. Anderson

Inhaltsverzeichnis

1.Einleitung

Thema dieser Diplomarbeit ist die Konzeption und Implementation einer dreidimensionalen Visualisierungs- und Steuerungskomponente für die MicroPsi-Multiagentenplattform. Diese Arbeit ist die Fortsetzung einer Studienarbeit zum gleichen Thema [SAL04].

3DView2 ist eine Komponente, die eine existierende virtuelle Weltsimulation dreidimensional darstellt und dem Benutzer darüber hinaus erlaubt, in Echtzeit mit dieser Welt zu interagieren.

Da diese Arbeit in ein größeres Projekt eingebunden ist, wird im ersten Teil das MicroPsi-Projekt (Humboldt-Universität zu Berlin, Universität Osnabrück) [BAC+03] sowie dessen unmittelbares Vorbild, das Psi-Projekt von Prof. Dietrich Dörner (Universität Bamberg) [DOE99], vorgestellt. Erst danach wird genauer darauf eingegangen werden, welche Ziele mit 3DView2 verfolgt werden und wie es in das Gesamtprojekt eingebunden ist.

Der zweite Teil beschäftigt sich mit dem für das Vorhaben relevanten Teil der MicroPsi-Architektur, insbesondere wird dabei auf die Kommunikationsschnittstellen zwischen den Komponenten und auf die im Rahmen dieser Arbeit entwickelten Erweiterungen eingegangen.

Der dritte und größte Teil dieser Arbeit ist Graphikalgorithmen gewidmet. Dieser Abschnitt ist wiederum in unterschiedliche Problemfelder – wie zum Beispiel Terraindarstellung oder Wasserdarstellung – gegliedert. Zu jedem Problemfeld wird eine Übersicht über die aktuellen Standardverfahren gegeben und die letztendliche Implementation vorgestellt. Es geht ausdrücklich nicht darum, diese Themengebiete erschöpfend zu behandeln; das würde den Rahmen dieser Arbeit sprengen und wäre dem eigentlichen Ziel – der Implementation einer Software – auch nicht dienlich. Vielmehr soll dargestellt werden, welche Methoden aktuell zur Verfügung stehen, warum in diesem Projekt Entscheidungen zugunsten oder zuungunsten bestimmter Technologien gefallen sind und wie verschiedene Techniken letztendlich in einer komplexen Anwendung kombiniert worden sind.

Am Schluss gibt es einen Ausblick auf mögliche Weiterentwicklungen des 3DViewers und des MicroPsi-Projektes.

Die meisten der für diese Arbeit und im MicroPsi-Toolkit verwendeten graphischen Assets – also Bitmaps, Texturen und 3D-Modelle für Agenten, Weltobjekte und das 3D-Emotionsgesicht – stammen von Florian Busse, Joscha Bach, Henning Zahn und Jens Meisner. Vielen Dank!

2.Das Vorbild: PSI

Prof. Dr. Dietrich Dörner forscht im Bereich Theoretische Psychologie an der Otto-Friedrich-Universität in Bamberg. Ein Schwerpunkt seiner Arbeit ist die Frage nach der Erklärbarkeit und Vorhersagbarkeit menschlichen Handelns in komplexen Situationen. Dörner weist darauf hin, dass Menschen nur in den seltensten Situation von einer einzigen Absicht oder einem einzigen Motiv getrieben werden; statt dessen ist *Multiple Motivation* die Regel.

Die von Dörner einwickelte PSI-Theorie [DOE99] ist ein einfaches Modell, das menschliche Entscheidungsprozesse abbilden soll. Dabei vereinigt und vereinheitlicht Dörner Theorien aus Sozial-, Emotions- und allgemeiner Psychologie. Das heißt, das PSI modelliert sowohl kognitive Aspekte als auch Motive und Emotionen.

Um ihr Modell auf Vollständigkeit und Widerspruchsfreiheit zu testen, haben Dörner und seine Forschungsgruppe eine Reihe von Computersimulationen entwickelt, in denen sich künstliche Agenten auf Basis der PSI-Theorie in komplexen virtuellen Umgebungen zurechtfinden müssen. In einigen dieser Simulationsumgebungen wurden Tests mit menschlichen Versuchspersonen durchgeführt, so dass das Verhalten der künstlichen Agenten direkt mit menschlichem Verhalten in einer identischen Situation vergleichbar wurde und das theoretische Modell entsprechend verbessert und verfeinert werden konnte.

Die PSI-Theorie ist bewusst einfach gehalten und kommt mit einem einzigen Grundbaustein aus, so genannten (theoretischen) Neuronen. Es handelt sich dabei um kleine, informationsverarbeitende Einheiten, die über eine Reihe von Ein- und Ausgängen verfügen, über die sie sich miteinander vernetzen lassen. Sie sollten nicht mit biologischen Neuronen verglichen oder verwechselt werden; vielmehr ähnelt die Gesamtstruktur *Influence Networks* oder *Belief Networks* [BAC03]. Sämtliche Prozesse und Speichervorgänge können mit Hilfe dieser Neuronen beschrieben werden. Über Sensoren und Aktoren ist das Netzwerk mit der (virtuellen) Außenwelt verbunden. Sensoren sind spezielle Neuronen, die Signale feuern, sobald bestimmte äußere Bedingungen erfüllt sind. Sie können beispielsweise auf Feuchtigkeit, Wärme, Kälte oder ähnliches reagieren. Aktoren sind das Gegenstück zu Sensoren, sie manipulieren die Umwelt, sobald sie ein entsprechendes Signal erhalten. Über die Ansteuerung von Aktoren kann ein Agent sich beispielsweise fortbewegen oder Nahrung aufnehmen. Die PSI-Theorie modelliert auch Emotion. Unter Emotion versteht Dörner eine Rückkopplung im System oder anders gesagt die Wahrnehmung bestimmter fest verdrahteter Prozesse im eigenen Körper. Beispielsweise führt der erfolglose Versuch, sich an die Lösung eines Problems zu erinnern, zu einem Gefühl von Hilflosigkeit und eine fehlgeschlagenen Aktion verursacht ein Gefühl des Ärgers. Diese Begriffe – „Hilflosigkeit“ und „Ärger“ – sind natürlich reine Interpretation von außen; prinzipiell sind es einfach Signale, die unter bestimmten Bedingungen am Netz anliegen und das weitere Verhalten beeinflussen können. Dörner konnte in seinen Experimenten zeigen, dass diese Emotionen einen sinnvollen Einfluss auf das Verhalten haben und kognitive Prozesse ergänzen können.

Diese Arbeit wird bewusst nicht näher auf die eigentliche PSI-Theorie eingehen, sondern sich nur Aspekten widmen, die für das Verständnis der Visualisierungsproblematik relevant sind.

2.1.PSI Reality 2D

Der Standard-Agent nach der PSI-Theory wird einfach „Ψ“ (Psi) genannt. Psi ist eine Dampfmaschine, die mit Sensoren ihre Umgebung wahrnehmen kann, sich auf Rädern fortbewegt und über einen Multifunktions-Greifarm zur Manipulation ihrer Umwelt verfügt. Psi hat drei primäre Bedürfnisse:

1. das Bedürfnis nach Wasser für den Dampfkessel, welches als „Durst“ bezeichnet wird,

2. das Bedürfnis nach Brennstoff, also „Hunger“, und

3. das Bedürfnis nach körperlicher Unversehrtheit, denn Psi kann durch äußere Einflüsse beschädigt werden.

Über die primären Bedürfnisse hinaus gibt es auch sekundäre Bedürfnisse, beispielsweise den Lerntrieb. Sekundäre Bedürfnisse machen das Verhalten der Agenten variantenreicher und erlauben auch dann eine weitere Optimierung des Verhaltens, wenn die primären Bedürfnisse in einer bestimmten Umgebung leicht erfüllbar sind.

Dörner hat Psi nach eigener Aussage als Dampfmaschine konzipiert, um zu zeigen, dass auch unbelebte Gegenstände als lebendig betrachtet werden können, sobald sie eine Reihe von Grundprinzipien realisieren [DOE98].

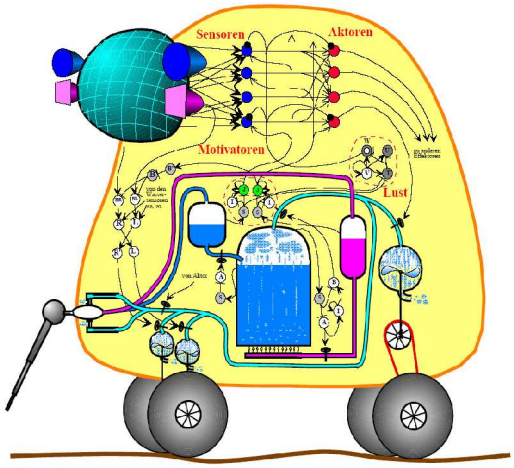


Abbildung 1: Schematische Darstellung eines einfachen PSI-Agenten (Quelle: [DOE98])

Es wurde eine Software – *Psi Reality 2D* – entwickelt, die künstliche Agenten auf Basis der Psi-Theorie in einer virtuellen Umgebung simulieren kann [DET00]. Als Szenario wurde eine Insel gewählt, da sich die Simulationswelt auf diese Weise schlüssig räumlich begrenzen lässt. Die virtuelle Welt enthält eine Vielzahl verschiedener Objekte, mit denen ein Agent interagieren kann; beispielsweise Bäume, verschiedene Pflanzen, Sanddünen und Wasserlöcher. Als Interaktionsmöglichkeiten stehen „Greifen“, „Saugen“, „Schütteln“, „Hämmern“ und „Sieben“ zur Verfügung. Da im nächsten Kapitel das Nachfolgeprogramm „Psi Reality 3D“ ausführlich vorgestellt wird, wird an dieser Stelle auf eine detailierte Beschreibung der Simulationswelt verzichtet.

Ein künstlicher Agent beginnt die Simulation ohne Wissen über die Welt. Er muss durch Versuch und Irrtum lernen, welche Aktionen seine primären und sekundären Bedürfnisse befriedigen und welche Aktionen möglicherweise Schaden anrichten. Psis primäres Ziel ist, in der Simulationswelt zu überleben, also seine primären Bedürfnisse zu befriedigen. Sein sekundäres Ziel ist, so genannte *Nukleotide* zu sammeln. Nukleotide sind ein fiktives Mineral, das an verschiedenen Stellen der Insel gefunden werden kann. Das Finden von Nukleotiden vermittelt dem Agenten ein „Erfolgserlebnis“ und befriedigt damit ein sekundäres Bedürfnis.

Psi Reality 2D kann auch von menschlichen Versuchspersonen bedient werden. Dabei stehen dem Menschen die gleichen Interaktionsmöglichkeiten zur Verfügung, über die auch ein virtueller Agent verfügt. Ziel war, das Verhalten des künstlichen Agenten direkt mit dem eines Menschen in einer identischer Situation vergleichen zu können. Die Rückschlüsse aus diesen Experimenten sollten zur Verbesserung der Psi-Theorie genutzt werden. Letztendlich sollte gezeigt werden, dass die Psi-Theorie zur Modellierung menschlichen Verhaltens geeignet ist.

Die Simulationswelt in Psi Reality 2D besteht aus einer Reihe von *Orten,* die über *Wege* miteinander verbunden sind. Der Agent kann direkt von Ort zu Ort wechseln, dies ist eine atomare Aktion. Mit jedem Weg sind bestimmte Kosten verbunden, d.h. jeder Weg erhöht Hunger und Durst des Agenten bzw. fügt ihm sogar Schaden zu. Jeder Ort enthält eine Reihe von *Objekten.* Um mit einem Objekt zu interagieren, muss das Objekt zunächst fokussiert werden. Danach können die genannten Aktionen auf das Objekt angewendet werden. Das Nachfolgeprojekt *Psi Reality 3D*, das im nächsten Kapitel vorgestellt wird, ermöglicht wesentlich freiere Bewegung.

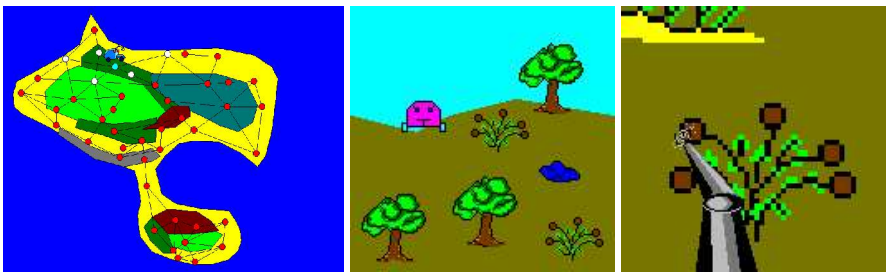


Abbildung 2: Die 2D-Insel (links) besteht aus Situationen ("Orten"), die über "Wege" miteinander verbunden sind. Jeder Ort enthält eine Reihe von Objekten (Mitte). Um mit einem Objekt zu interagieren muss es fokussiert werden (rechts). (Screenshots von Psi Reality 2D)

2.2.PSI Reality 3D

2.2.1.Idee und Anspruch

In Versuchen mit Psi-2D konnte nachgewiesen werden, dass der Psi-Agent sich ähnlich verhält wie menschliche Versuchspersonen in der selben Situation [DET00]. Im nächsten Schritt sollte das Szenario für soziologische Experimente erweitert werden. Es ging dabei um die Entstehung und Änderung sozialer Strukturen in Gruppen intelligenter, multimotivierter, emotionaler Agenten [GER+01]. Hierzu war es notwendig, eine neue Simulationsumgebung zu schaffen, in der mehrere autonome Agenten gleichzeitig agieren können. Genaugenommen war das Forschungsvorhaben sogar breiter angelegt; drei Punkte waren essentiell:

1. Eine homogene Menge künstlicher Agenten sollte in der Simulationsumgebung gleichzeitig agieren können. Dabei sollten Gruppenbildung, Gruppenstabilität, Gruppenstrukturen und Gruppenzerfall unter verschiedenen Bedingungen untersucht werden. Umweltparameter, Fähigkeitsprofile und die Persönlichkeiten der Agenten sollten variierbar sein.

2. Mehrere menschliche Versuchspersonen sollten in der Simulationsumgebung gleichzeitig agieren können. Dabei sollte untersucht werden, welche Gruppenprozesse bei Menschen in der Versuchssituation tatsächlich auftreten und ob diese Prozesse durch die Psi-Theorie abgebildet und erklärt werden können. Außerdem sollte die Frage beantwortet werden, ob Psi-Agenten zur gleichen Spannbreite an unterschiedlichen Verhaltensweisen fähig sind wie menschliche Versuchspersonen.

3. Letztendlich sollte die Simulation auch Hybridbetrieb erlauben, also das gleichzeitige Agieren von menschlichen Versuchspersonen und künstlichen Agenten ermöglichen. In einem solchen Rahmen kann sehr gut untersucht werden, inwiefern die künstlichen Agenten ähnliche Verhaltensweisen zeigen wie menschliche. Wie beim Turing-Test sollten die menschlichen Versuchsteilnehmer anschließend befragt werden, welche Agenten künstlich und welche menschlich waren. Die Ergebnisse der Befragung könnten Anhaltspunkte zur Verbesserung des Psi-Programmes liefern.

Um diesen Zielstellungen gerecht zu werden, musste die simulierte Umwelt komplex genug sein, um menschlichen Agenten genügend Herausforderungen zu bieten, andererseits müssten dieses Herausforderungen auch von den künstlichen Agenten zu bewältigen sein. Das Szenario musste hinsichtlich der räumlichen Ausdehnung groß genug sein, um sich nicht in zu kurzer Zeit vollständig erkunden zu lassen. Es musste auch größeren Agentenpopulationen ausreichend Platz bieten, zudem musste es möglichst viele verschiedene Verhaltensweisen ermöglichen und viele verschiedene Probleme unterschiedlicher Komplexität bereithalten. Nicht alle Eigenschaften und Wechselwirkungen dieser Welt dürfen sofort erkennbar sein. Außerdem sollte die Umwelt eigendynamische Komponenten enthalten, welche die Agenten vor überraschende Herausforderungen und Bedrohungen stellen können.

Psi 3D greift das Inselszenario des Vorgängerprojektes auf, erweitert jedoch massiv die räumliche Größe der Welt, die Anzahl möglicher Aktionen usw.

2.2.2.Welt und Interaktion

Da Psi 3D ein direktes Vorbild für die MicroPsi-Weltsimulation und die im Rahmen dieser Arbeit zu entwickelnde 3DViewer-Komponente ist, sollen in diesem Abschnitt ausführlich Dörners Inselszenario und die vielfältigen Interaktionsmöglichkeiten in dieser virtuellen Welt beschrieben werden. Die MicroPsi-Weltsimulation ist momentan leider noch weit von Komplexität und Variantenreichtum dieses Szenarios entfernt.

Für eine menschliche Versuchsperson stellt sich Psi Reality 3D als eine Art Computerspiel dar. Der Spieler wird auf die fiktive, unbewohnte Südseeinsel „Nukleotidios“ versetzt. Er übernimmt per Fernsteuerung die Kontrolle über den Roboter „Robbi“ und bekommt die Aufgabe, so genannte „Nukleotide“ zu sammeln. Nukleotide, so klärt die Spielanleitung [HÄM03] auf, sind kleine, gelbe Steine die, da es sich bei ihnen um nahezu ideale Energieträger handelt, sehr wertvoll sind. Ein aktiver Vulkan auf der Insel befördert Nukleotide aus dem Erdinneren an die Oberfläche. Robbi wird von von einem Schiff an der Nordseite der Insel abgesetzt. Von hier aus muss sich der Spieler auf die Suche nach Nukleotiden machen und diese wieder beim Schiff abliefern. Der Erfolg des Spielers wird an den am Schiff abgelieferten Nukleotiden gemessen.

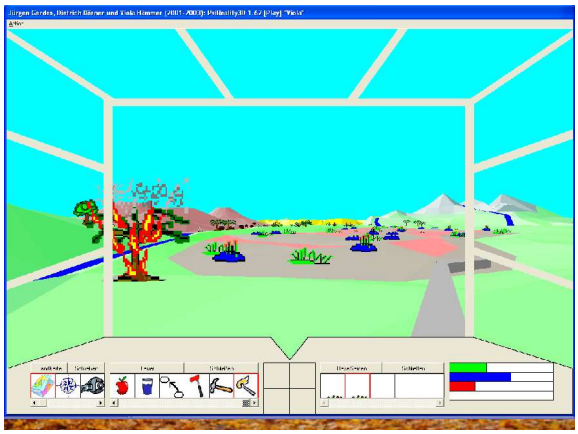


Abbildung 3: Psi Reality 3D aus Sicht eines menschlichen Spielers.
Die Icons unten links befinden sich Buttons für die verfügbaren Aktionen, die Icons unten rechts stellen das aktuelle Gepäck des Roboters dar. Das Balkendiagramm zeigt Hunger (grün), Durst (blau) und Schaden (rot).
Der momentan leere Balken steigt mit den gesammelte Nukleotiden an. Unten in der Mitte befindet sich das Steuerfeld. (Screenshot von Psi Reality 3D)

Diese Aufgabe ist selbstverständlich nicht so einfach, wie sie zunächst klingt. Auf der Insel lauern allerlei Gefahren wie z.B. Schwefeldämpfe, durch die Robbi Schaden nehmen kann. Außerdem ist Robbis Antrieb eine Dampfmaschine, die zum Funktionieren Wasser und Öl benötigt. Entsprechend hat der Spieler auf dem Bildschirm Anzeigebalken für Hunger, Durst und den erlittenen Schaden. Seine Aufgabe ist es, diese Balken unten zu halten – gelingt ihm das nicht und einer der Balken erreicht 100%, wird Robbi zerstört und der Spieler muss mit einem neuen Roboter vom Schiff aus starten. Dies ist aus Sicht des Spielers sehr unerfreulich, da jeder zerstörte Roboter dem Spieler mit sechs Nukleotiden in Rechnung gestellt wird. Zudem geht mit dem Roboter der gesamte Inhalt seines Laderaums verloren. Bei einigen Experimenten wurde die Bezahlung der Testpersonen von der Anzahl gesammelter Nukleotide abhängig gemacht [DET00].

Der Spieler muss also bei der Suche nach Nukleotiden darauf achten, dass er den Roboter nicht beschädigt und zudem dessen Bedürfnisse, nämlich Hunger und Durst, befriedigt. Wasser findet sich an verschiedenen Stellen auf der Insel, Öl kann aus ölhaltigen Pflanzen gewonnen werden. Es gibt sogar Pflanzen, die am Roboter entstandenen Schaden heilen. Welche Pflanzen das sind und welche Möglichkeiten und Gefahren die Insel bereithält, muss der Spieler selbst herausfinden.

Der Spieler sieht vor sich ein Kamerabild aus Robbis Perspektive, d.h. eine dreidimensionale Ansicht der Insel. Robbi wird komplett mit der Maus gesteuert. Er kann vorwärts und rückwärts fahren und sich nach links oder rechts drehen. Robbi kann allerdings nur Gefälle bis zu einem gewissen Winkel passieren, sowohl abwärts als auch aufwärts. Die Insel besteht aus vielen verschiedenen Bodentypen, die auch Einfluss auf Robbis Bewegungsmöglichkeiten haben. Auf einer grünen Wiese kommt Robbi z.B. erheblich schneller voran als auf lockerem Sand. Wasserflächen kann er ohne Hilfsmittel nicht überqueren. Bodentypen haben ebenfalls Einfluss auf Robbis Bedürfnisse, so steigt sein Durst im Wüstensand deutlich schneller an als z.B. im Grasland. Beim Überqueren von Lava oder schlammigen Sumpfgebieten nimmt er sogar Schaden.

Der Spieler kann Objekte in Robbis unmittelbarer Umgebung anwählen und damit Aktionen durchführen. Auch der Boden direkt vor Robbi kann angewählt werden. Das Terrain der Insel ist aus Dreiecken aufgebaut, konzeptionell sind diese Bodendreiecke ebenfalls Objekte. Die möglichen Aktionen sind:

- *Nehmen*: Robbi hat einen kleinen Laderaum (*Gepäck*), in dem er Gegenstände verstauen kann. Jeder Gegenstand hat ein Gewicht und ein Volumen; der Laderaum ist dahingehend begrenzt. Einige Aktionen lassen sich auch mit Gegenständen im Gepäck durchführen.
- *Ablegen*: Gegenstände aus dem Gepäck lassen sich wieder auf den Boden legen.
- *Fressen*: Einige Gegenstände können gegessen werden.
- *Trinken*: Robbi kann z.B. aus Bächen oder Pfützen trinken.
- *Anwenden auf*: Robbi kann Gegenstände miteinander benutzen. Was dabei genau geschieht ist von den Gegenständen abhängig. So können beispielsweise Nukleotide beim Schiff abgegeben oder Kühe mit dem Stock vertrieben werden.
- *Axt*: Mit der Axt kann Robbi Gegenstände bearbeiten, insbesondere Holz.
- *Hammer*: Mit dem Hammer kann auf Gegenstände eingeschlagen werden
- *Feuer*: Gegenstände können in Brand gesetzt werden.

Nicht jede Aktion ist auf jeden Gegenstand anwendbar. Was genau bei einer Aktion geschieht, hängt vom Gegenstand ab. Ein Baum wird mit der Akt gefällt, d.h. er verwandelt sich ein einen „gefällten Baum“. Eine weitere Anwendung der Axt macht daraus einen Baumstumpf, einen Baumstamm und einige Äste. Teilweise führen Aktionen bei (scheinbar) gleichen Objekten zu unterschiedlichen Ergebnissen; zum Beispiel können Steine mit dem Hammer zertrümmert werden, einige enthalten ein Nukleotid, andere nicht.

Abbildung 4: Die Psi-3D-Insel in der Draufsicht.(Screenshot von Psi Reality 3D)

Die Insel ist in verschiedene Regionen aufgeteilt. In der unmittelbaren Umgebung des Startpunktes gibt es reichlich Nahrung in Form von Nüssen sowie Wasser in den Bächen bzw. in einem kleinen See. Im Norden der Insel befindet sich ein Sumpfgebiet, in dem Agenten leicht versinken (d.h. Schaden nehmen) können. Der Süden wird von Vulkanen dominiert, im Osten gibt es ein Wüstengebiet, im Westen einen dichten Wald. Die Gebiete erfordern unterschiedliche Strategien was das Überleben und die Nukleotidsuche betrifft:

- Im Sumpf muss der Spieler vor allem lernen, festen und unsicheren Boden zu unterscheiden.
- Der Wald ist sehr dicht; daher muss sich Robbi eventuell einen Weg mit der Axt bahnen. Alternativ lässt sich der Wald in Brand setzen, in der Regel brennt er dabei komplett nieder. Im Wald befindet sich eine Wombathöhle, in der es viele Nukleotide zu finden gibt. Durch einen Waldbrand werden diese jedoch vernichtet.
- Die Wüste ist sehr unübersichtlich, da die Sanddünen zu hoch sind, um sie zu überblicken oder gar zu überwinden. In der Wüste befinden sich aber Oasen, in denen es viele Nukleotide gibt. Es ist sinnvoll, sich von einem nahe gelegenen Hügel aus zunächst einen Überblick zu verschaffen, dann einen Weg zur Oase zu planen und den Weg mit ausreichend Wasser im Gepäck anzutreten.
- In der Nähe der Vulkane gibt es Schwefeldämpfe, die Robbi Schaden zufügen; mit Pfefferminzpflanzen im Gepäck kann er sich jedoch dagegen schützen. Das Vulkangebiet ist reich an Nukleotiden und Steinen, die beim Zertrümmern Nukleotide freigeben. Sie sind jedoch viel zu heiß zum Aufnehmen – Robbi nimmt bei dem Versuch Schaden (heiße Objekte sind optisch an kleinen Rauchfahnen zu erkennen). Mit Wasser aus seinem Gepäck kann der Spieler sie jedoch ausreichend kühlen.

Die Vegetation der Insel verändert sich je nach Region. In den kälteren Regionen werden Hunger und Durst mit Nüssen und dem Wasser aus Bächen gestillt, im wärmeren Süden dienen dazu Sonnenblumen und Kokosnüsse.

Es gibt noch viele weitere Beispiele für komplexe Aktionen, die in der Psi-Welt möglich sind; an dieser Stelle seien nur einige genannt, um einen groben Eindruck zu vermitteln:

- Kühe setzen sich gern auf Nukleotide, da diese so angenehm warm sind. Kühe lassen sich mit Steinen bewerfen oder mit Holzstöcken vertreiben (Objekte *aufeinander anwenden*), das provoziert aber meist einen schmerzhaften Gegenangriff. Der weniger riskante Lösungsweg wäre, ein Grasbüschel einzusammeln und es in Sichtweite der Kuh auf den Boden zu legen – das lockt sie an und gibt das Nukleotid frei.
- Palmen können gefällt werden, um nahrhafte Kokusnüsse zu gewinnen. Bevor sie gegessen werden können, müssen die Nüsse jedoch mit dem Hammer zertrümmert werden.
- Pflanzen, z.B. Sonnenblumen und Haselnussbüsche, verdorren auf kargem Boden. Der Spieler kann sie gießen, um diese wichtigen Nahrungsquellen zu erhalten. Es ist sogar möglich, neue Sträucher und Sonnenblumen auszusähen (Samen auf den Boden legen, Wasser aufnehmen und auf den Samen anwenden).
- Bäche sind unter normalen Umständen unüberwindlich. Ein gefällter Baum liefert jedoch nach weiterer Bearbeitung mit der Axt einen Baumstamm, der als Brücke auf einem Wasserdreieck abgelegt werden kann.
- Einige Nukleotide liegen auf Felsvorsprüngen, die für Robbis Greifarm nicht erreichbar sind. Mit einem langen Stock (der durch das Fällen von Bäumen gewonnen werden kann) können sie jedoch herunter gestoßen und anschließend eingesammelt werden.
- Baume, Sträucher oder trockene Äste können in Brand gesetzt werden. Nach kurzer Zeit fangen in der Nähe befindliche brennbare Objekte ebenfalls Feuer, so dass sich Brände über die ganze Insel ausbreiten können. Mit Wasser können brennende Objekte wieder gelöscht werden, was in der Praxis jedoch kaum gelingt, da Feuer sich viel zu schnell ausbreitet.

Als zusätzliche Motivation erhält der Spieler für jeweils zehn am Schiff abgelieferte Nukleotide eine frei wählbare Belohnung; er kann sich beispielsweise für textuelle Informationen über Gefahren und Nukleotidvorkommen auf der Insel entscheiden oder seinen Roboter aufrüsten lassen. Als Extras stehen Solarkollektoren (halbieren das Hungerbedürfnis), Sonnenschutz (halbiert den Durst) und eine Verdopplung des Laderaums zur Auswahl.

2.2.3.Technische Konzepte

Architektur

Psi 3D [GER+03] besteht aus einer Reihe von separaten Programmen. Für den Mulitagentenbetrieb ist eine Client/Server-Lösung erforderlich. Die Client-Programme verfügen jeweils über eine komplette, eigene Repräsentation der Welt. Änderungen an der Welt werden an einen Server kommuniziert, der über eine eigene Weltdarstellung verfügt und die Änderungen an die übrigen Clients weitergibt. Der Server verarbeitet die ankommenden Nachrichten sequentiell und löst dadurch Konflikte, die durch gleichzeitige oder zeitnahe widersprüchliche Aktionen auf verschiedenen Clients auftreten können. Entsprechend kann er Clients mitteilen, dass die von ihnen ausgeführte Aktion gescheitert ist. Bewegungen der Agenten sind ebenfalls Aktionen; der Server verhindert dabei, dass zwei Agenten den gleichen Ort einnehmen.

Die Welt mit ihren Regeln ist nicht in der Software kodiert, sondern kann komplett über Textdateien definiert werden. Diese beschreiben Objekttypen und Nachrichten (siehe unten); auf diese Weise kann – mit Einschränkungen – ein komplett anderes Szenario realisiert werden, ohne den Programmcode zu verändern.

Die Welt lässt sich mit einem Editor bearbeiten. Der Editor erlaubt sowohl das dialoggesteuerte Bearbeiten der oben genannten Textdateien – also das Erstellen neuer Objekttypen, Nachrichtentypen u.ä. – als auch das Platzieren von Objekten in der 3D-Welt und das Bearbeiten der dreidimensionalen Geländeform.

Die Clients legen ein vollständiges Protokoll aller ausgeführten Aktionen an. Mit einem separaten Programm lassen sich diese Protokolle auswerten und Graphiken erstellen, beispielsweise ein Bewegungsdiagramm des Agenten.

Ein weiterer Programmmodus erlaubt einen Flug über die dreidimensionale Landschaft.

Terrain

Das Terrain wird durch ein zusammenhängendes Netz aus Dreiecken definiert. Jedes Dreieck gehört einem bestimmten Terraintyp an, z.B. Wüste oder Vulkan. Das Terrain kann mit einem separaten Editor bearbeitet werden. Es ist möglich, Formen mit einander überschneidenden oder durchdringenden Dreiecken zu konstruieren. Obwohl es mit dieser freien Gestaltungstechnik theoretisch möglich wäre, Brücken oder Überhänge zu konstruieren, ist das ausdrücklich verboten und wird von der Software als Fehler gewertet. Es werden statt dessen nur Ebenen mit unterschiedlichen Höhenverläufen unterstützt.

Objekte

Als *Objekte* werden in der Psi-Welt Gegenstände wie Pflanzen, Bäume, Steine usw. bezeichnet, aber auch Bodendreiecke zählen als Objekte. Jedes Objekt hat einen Typ, für jeden Objekttypen werden eine Reihe von Parametern definiert:

- ein Name
- eine Reihe von Bitmaps, die die Ansichten definieren
- ein Radius; für die Kollisionserkennung werden Objekte als rund angenommen
- Gewicht und Volumen; sind entscheidend, wenn Objekte ins Gepäck eines anderen Objektes genommen werden sollen (Agenten sind auch Objekte)
- eine Farbe; nur relevant für Bodendreiecke
- Schrittweite und Steigung; geben an, wie weit sich ein Objekt mit einem Schritt bewegen kann (dieser Wert ist bei unbeweglichen Objekten Null) und welche Steigung maximal überwunden werden kann
- Geschwindigkeitsfaktor; gibt bei Bodendreiecken an, mit welchem Faktor die Geschwindigkeit eines Objektes multipliziert wird, wenn es sich über dieses Bodenmaterial bewegt (meist < 1)
- ein Maximalvolumen und Maximalgewicht, das Objekte dieses Types als Gepäck tragen können

Die konkreten Objekttypen werden komplett über Textdateien, also ohne Änderungen im Quelltext, definiert.

Normale Objekte, also alle, die keine Bodendreiecke sind, werden in der 3D-Welt mittels *Billboards* dargestellt; das sind Bitmaps, die sich immer automatisch dem Betrachter zuwenden. Es können pro Objekttyp mehrere Bitmaps angegeben werden, die dann als Ansichten von verschiedenen Seiten dienen. Abhängig von Blickrichtung des Betrachters und räumlicher Orientierung des Objektes wird dann automatisch die passende Bitmap zur Darstellung gewählt.

Dass Bodendreicke ebenfalls als Objekte gewertet werden, mag zunächst überraschen, sind doch die Unterschiede zu den normalen Objekten größer als die Gemeinsamkeiten. Das wird schon dadurch deutlich, dass die meisten Objekttypparameter entweder für normale Objekte oder für Bodendreicke gelten, jedoch nicht für beide. Durch diesen konzeptionellen Trick fügen sich Bodendreiecke jedoch sehr elegant in das Nachrichtensystem ein und ermöglichen Interaktion zwischen Bodendreiecken und normalen Objekten.

Nachrichtensystem

Objekte kommunizieren über Nachrichten miteinander, d.h. ein Objekt kann Nachrichten versenden und auf Nachrichten reagieren. Genau wie die Objekttypen können auch die Nachrichten und die Reaktionen der Objekte über ein festes Schema in Textdateien definiert werden, ohne den Quelltext der Applikation zu verändern. Eine Nachricht besteht aus einem Nachrichtennamen und einer Reihe von Parametern und wird an ein konkretes Objekt geschickt. Als Reaktion darauf kann das Objekt seinen Objekttyp ändern und/oder weitere Nachrichten an sich selbst oder andere Objekte verschicken. Die für ein bestimmtes Objekt relevanten Nachrichten werden pro Objekttyp nach folgendem Schema definiert:

- *Objekttyp* – der Objekttyp, der auf die Nachricht reagieren soll
- *Ziel* – der Objekttyp, in den sich das Objekt aufgrund der Nachricht verwandeln soll; es kann „Weg“ angegeben werden, um das Objekt stattdessen zu löschen
- *Name* – der Name der Nachricht
- *Wahrscheinlichkeit* – die Wahrscheinlichkeit, dass die Nachricht überhaupt bearbeitet wird
- *Dauer* – eine Zeitverzögerung in Sekunden, bevor die Nachricht bearbeitet wird
- *MsgSelf* – der Name einer Nachricht, die das Objekt an sich selbst schicken soll
- *MsgOther* – der Name einer Nachricht, die das Objekt an andere Objekte schicken soll. Je nach Nachricht können alle Objekte in der Welt oder nur Objekte in einem bestimmten Umkreis adressiert werden.
- *Autostart* – boolescher Wert, gibt an, ob die Nachricht *MsgSelf* nach Ablauf der in *Dauer* festgelegten Zeit vom Programmstart an automatisch gesendet werden soll
- *AutoStartRandom* – boolescher Wert, wenn Autostart den Wert true hat, gibt diese Variable an, ob statt der exakten Zeit in *Dauer* ein Zufallswert zwischen 0 und *Dauer* verwendet werden soll
- *Autorepeat* – boolescher Wert, gibt an, ob die in MsgSelf angegebene Nachricht automatisch wiederholt wird

Dieses Schema erscheint im ersten Moment rigide und kompliziert, erweist sich jedoch in der Praxis als stabil und sehr vielseitig. Tatsächlich sind alle bisher beschriebenen Interaktionen mit und in der Welt mittels dieses Schemas realisiert. Selbst die Steuerung des Agenten durch den Menschen geschieht intern über solche Nachrichten, im Quelltext selbst stecken keine Informationen über die Inselwelt. Ein konkretes Beispiel:

- *Objekttyp:* Baum
- *Ziel*: BaumBrennt1
- *Name*: Anzünden
- *Wahrscheinlichkeit*: 1.00000000
- *Dauer*: 4 s
- *MsgSelf*: BaumBrennt1
- *MsgOther*: Anzünden 150
- *Autostart, AutoStartRandom, AutoRepeat*: 0 (alle false)

Wenn ein Objekt vom Typ „Baum“ die Nachricht „Anzünden“ erhält, reagiert es nach 4 Sekunden mit einer Wahrscheinlichkeit von 1, also garantiert, indem es sich in ein Objekt vom Typ „BaumBrennt1“ verwandelt. Im Moment der Verwandlung sendet es eine Nachricht „BaumBrennt1“ an sich selbst und eine Nachricht „Anzünden“ an alle Objekte im Umkreis von 150 Metern. BaumBrennt1 ist ein anderer Objekttyp, der als Bitmap einen brennenden Baum hat; dadurch wird die Veränderung in der Welt sichtbar. BaumBrennt1 definiert wiederum eigene Reaktionen auf bestimmte Nachrichten. Die Nachricht BaumBrennt1, die das Objekt an sich selbst geschickt hat, führt dazu, dass BaumBrennt1 sich nach einer gewissen Zeit in BaumBrennt2 verwandelt – das ist ein fortgeschrittenes Stadium des Verbrennens. Dieses Verhalten ist für den Objekttypen Baumbrennt1 definiert. Auf diese Weise lassen sich Ketten von Ereignissen programmieren, beispielsweise das schrittweise Verbrennen eines Baumes, das Wachsen, Früchte-Tragen und Verdorren eines Strauches oder ähnliches. Durch das Versenden der „Anzünden“-Nachricht im Umkreis von 150 Metern fangen Bäume und andere Objekte in der Umgebung ebenfalls Feuer. Jedes Objekt, das auf eine solche Nachricht reagieren möchte, muss explizit eine Reaktion definieren. Erhält ein Objekt eine Nachricht, die für diesen Objekttyp nicht definiert ist, erfolgt keine Reaktion. Steine in der Umgebung erhalten die „Anzünden“-Nachricht beispielsweise auch, reagieren aber nicht darauf.

An dieser Stelle kommen die Bodendreiecke ins Spiel, denn da sie auch Objekte sind, können sie ebenfalls Nachrichten versenden und erhalten. Viele Bodendreiecke versenden ständig Nachrichten an ihre unmittelbare Umgebung. Der Bodentyp Sumpf beispielsweise sendet in kurzen Abständen „Versinken“-Nachrichten; die meisten kleineren Objekttypen reagieren auf diese Nachricht, indem sie sich selbst löschen. Wiesendreiecke versenden analog „Wachsen“-Nachrichten; Pflanzen reagieren auf diese Nachricht, indem sie größer werden. Durch diesen Mechanismus wachsen Pflanzen nur auf bestimmten Bodentypen. Analog erhalten Agentenobjekte von bestimmten Bodendreiecken Nachrichten, die ihre Bedürfnisse verändern, beispielsweise steigern Wüstendreiecke den Durst.

Das ständige Wiederholen von Nachrichten passiert über eine Indirektion. Der Bodentyp Sumpf definiert beispielsweise eine Nachricht „SendVersinken“, die er mittels *AutoStart* und *AutoRepeat* immer wieder an sich selbst (*MsgSelf)* schickt. Als Nebeneffekt dieser Nachricht wird eine Nachricht „Versinken“ an alle Objekte in der Umgebung verschickt (*MsgOther*). Da „SendVersinken“ eine Dauer von 10 Sekunden definiert hat, kann es bis zu 10 Sekunden dauern, bis ein Objekt im Sumpf die erste „Versinken“-Nachricht erhält.

2.3.Kritikpunkte

Es gibt eine ganze Reihe von Kritikpunkten an Psi 3D, die vor allem die technische Umsetzung betreffen. Der MicroPsi-3DViewer wird versuchen, diese Probleme zu adressieren.

Die Steuerung beschränkt sich auf Vorwärts- und Rückwärtslaufen mit unterschiedlichen Geschwindigkeiten sowie Links- und Rechtsdrehungen. Die Steuerung erfolgt über einen virtuellen Joystick auf dem Bildschirm, der mit der Maus bewegt werden kann. Das Konzept ist unkonventionell; eine (zumindest optionale) Steuerung mittels Tastatur oder echtem Joystick wäre weniger erklärungsbedürftig und präziser gewesen. Es wäre wichtig, auch nach oben und unten blicken zu können. Kleine Gegenstände unmittelbar vor dem Robot verschwinden schnell aus dem Sichtfeld. Oftmals muss nach dem Ablegen eines Gegenstandes ein Schritt rückwärts gemacht werden, um den Gegenstand sehen zu können. Bei kommerziellen Computerspielen hat sich hier ein bewährter Standard etabliert; die Tastatur dient der Bewegung, die Maus kontrolliert Drehung und Blickrichtung der Spielfigur.

Die Bedienung ist teilweise unlogisch; beispielsweise kann ein Baumstamm als Brücke über einen Bach verwendet werden, indem er in unmittelbarer Nähe des Baches abgelegt wird. Die ebenso logische Aktion, den Baumstamm auf das Bach-Objekt anzuwenden, schlägt fehlt. Um eine Sonnenblume zu säen, müssen Sonnenblumenkerne auf den Boden gelegt und Wasser aus dem Gepäck darauf anwendet werden. Die Vermutung, die Samen müssten in die Erde eingebracht werden, indem sie auf den Boden angewandt werden oder ähnliches erweist sich als falsch; auf diese Weise hat der Spieler oft richtige Ideen, findet aber unter Umständen nicht die richtige Methode, sie im Programm umzusetzen.

Das Interface auf dem Bildschirm ist unpraktisch; die Leiste mit den möglichen Aktionen ist mit einem Rollbalken versehen, der den Spieler zum ständigen Scrollen zwingt. Für mit Computern unerfahrene Benutzer könnte dies sogar ein ernstes Hindernis darstellen und dazu führen, dass sie bestimmte Aktionsmöglichkeiten gar nicht wahrnehmen. Ein so wichtiger Bestandteil der Benutzeroberfläche sollte möglichst immer vollständig sichtbar sein.

Einer der Gegenstände im Gepäck ist ständig angewählt; zusätzlich dazu kann ein Gegenstand in der Welt angewählt sein. Beim Ausführen der Aktion „Fressen“ wird jedoch immer der im Gepäck gewählte Gegenstand gefressen – nur, wenn das nicht möglich ist, wird die Aktion auf den Gegenstand in der Welt angewendet. Die Anleitung rät, vor dem Fressen einen nicht essbaren Gegenstand im Gepäck anzuwählen oder das Gepäck mit dem dafür vorgesehenen Knopf zu schließen. Um Gegenstände einsammeln zu können muss das Gepäck jedoch geöffnet sein. Dieses Bedienelement ist sehr unpraktisch, zumal es keine anderen Gründe gibt, das Gepäck zu öffnen oder zu schließen. Mögliche Lösungsvorschläge:

- Statt des Öffnen/Schließen-Knopfes für das Gepäck wäre ein Essen/Trinken-Knopf sinnvoller gewesen; tatsächlich sind das die einzigen Aktionen, die mit Gegenständen im Gepäck funktionieren.

- Es könnte auch komplett darauf verzichtet werden, Gegenstände direkt aus dem Gepäck zu essen. Statt dessen müssten sie vorher – wie bei allen anderen Aktionen – auf dem Boden abgelegt werden.

- Eventuell könnte auch darauf verzichtet werden, sowohl einen Gegenstand in der Welt als auch im Inventar angewählt zu haben. Dazu müsste die „Benutzen mit“-Aktion anders gestaltet werden, da sie zwei Objekte benötigt.

Der größte Kritikpunkt ist die Graphik, die dem aktuellen Stand der Technik nicht gerecht wird. Das ist nicht nur ein ästhetisches Problem, sondern beeinträchtigt teilweise auch die Funktion. Viele Gegenstände sind nur schwer als das zu erkennen, was sie darstellen sollen. Hier wären zumindest Beschreibungstexte, die beim Anwählen eines Gegenstandes eingeblendet werden, hilfreich gewesen. Auch stimmt aufgrund der sehr einfachen graphischen Darstellung die Objektkollision nicht mit dem optischen Eindruck überein. Das ist insbesondere im Waldgebiet problematisch, wo der Spieler permanent mit scheinbar unsichtbaren Hindernissen kollidiert. Baumstämme können als Brücken über Bäche genutzt werden, graphisch wird das jedoch nicht deutlich gemacht. Optisch scheint der Baumstamm zu kurz zu sein, um die Entfernung zu überbrücken, und er kann eine gewisse Strecke vom Bach entfernt liegen; trotzdem kann der Spieler den Bach überqueren.

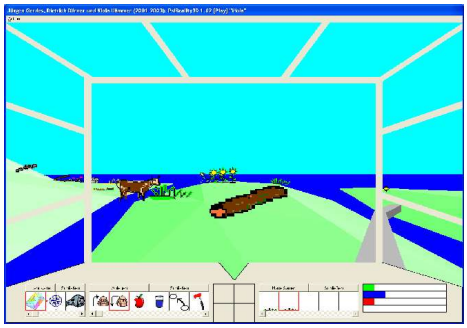


Abbildung 5: Der optische Eindruck trügt - dieser Baumstamm überbrückt sowohl den Bach auf der linken als auch den auf der rechten Seite (Screenshot von Psi Reality 3D)

Das Anwählen von Objekten wird dadurch erschwert, dass das *Picking,* also die Zuordnung von Mausklicks zu Objekten auf dem Bildschirm, auf der rechteckigen Form der Bitmaps basiert, obwohl diese zu großen Teilen transparent sind. So kann oftmals ein deutlich sichtbares Objekt trotz präzisen Anklickens mit der Maus nicht angewählt werden, stattdessen wird ein in der Szene weiter vorn befindliches Objekt angewählt. Insbesondere in den engen Oasen in der Wüste und im Wald tritt dieses Problem auf.

Es gibt leider keine Publikationen über die technische Umsetzung der 3D-Graphik, jedoch lässt sich feststellen, dass sie trotz des geringen Detailgrades und der niedrigen Polygonzahl selbst auf sehr leistungsfähigen Rechnern langsam ist. Dies lässt vermuten, dass auf Standardverfahren wie hardwarefreundliche Sortierung und Gruppierung der Geometrie und *Culling* verzichtet worden ist. Optimierungen, wie z.B. *Level-of-Detail*-Algorithmen (LOD), sind nicht erkennbar. Die Bedienung des Welteditors ist äußerst umständlich; statt die Definition der groben Geländeform durch Bitmaps (siehe 9.1) oder den Import aus 3D-Modellierungsprogrammen zu erlauben, wurde ein eigener 3D-Editor für Geländeformen implementiert, dessen Fähigkeiten im Vergleich zu existierender Standardsoftware sehr beschränkt sind.

3.Das MicroPsi-Projekt

Dörners PSI-Theorie ist aus einer psychologischen Fragestellung heraus entwickelt worden. Angesichts der Computersimulationen, die er und seine Arbeitsgruppe durchgeführt haben, drängt sich die Frage auf, inwiefern sich die PSI-Theorie auch im Bereich der klassischen KI-Forschung verwerten lässt, also beispielsweise zur Schaffung künstlicher Agenten oder zur Robotersteuerung [BAC03]. Das MicroPsi-Team beschäftigt sich mit genau diesen Fragestellungen. Ziel von MicroPsi ist, eine abstraktere und formalere Version von Dörners Modell zu entwickeln, die auch für Anwendungen in der Informatik geeignet ist. Darüber hinaus soll die Theorie um neue Konzepte erweitert werden und es sollen schließlich neue Agenten auf Basis des MicroPsi-Modells entwickelt werden.

Das MicroPsi-Projektteam hat sich an der Humboldt-Universität zu Berlin formiert, ursprünglich unter dem Namen *„Artificial Emotion Project“ (AEP)*. Da sich der Projektfokus jedoch mittlerweile von künstlichen Emotionen auf einen viel allgemeineren Anspruch verschoben hat, wurde dieser Name abgelegt; statt dessen wird jetzt von *„Cognitive Agents“* oder einfach vom MicroPsi-Projekt gesprochen. Inzwischen ist auch die Universität Osnabrück am Projekt beteiligt und es existieren Kooperationen mit weiteren Hochschulen.

Ein zentraler Teil des Projektes ist die Entwicklung des MicroPsi-Toolkits, einer komplexen Softwareumgebung, die die komfortable Konstruktion von Agenten und die Durchführung von Simulationen mit diesen Agenten erlaubt. Das Toolkit ist nicht als Entwicklungswerkzeug gedacht, sondern soll vielmehr als Experimentierplattform dienen. Die offene Architektur ermöglicht die Entwicklung einer Vielzahl verschiedener Agententypen.

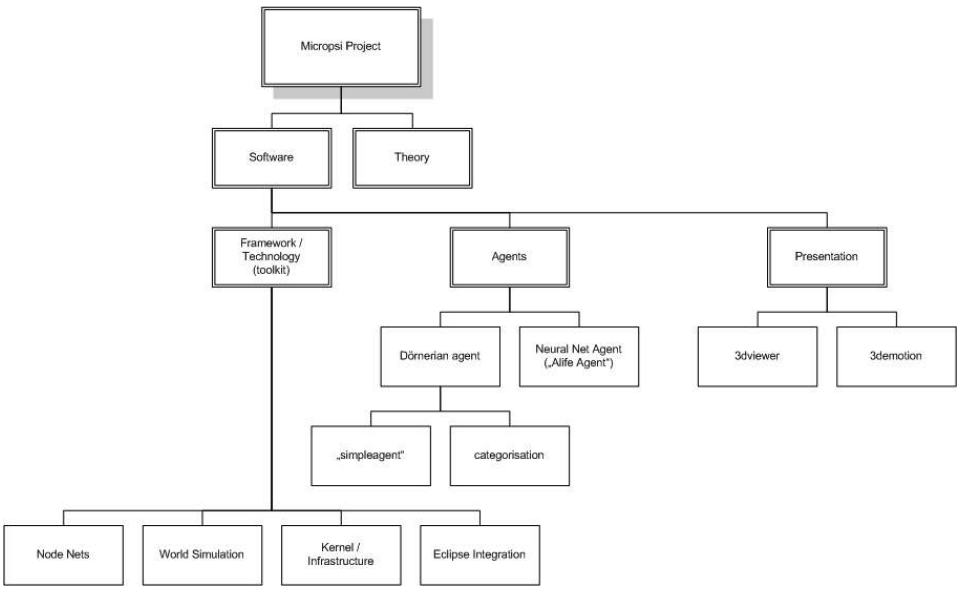


Abbildung 6: MicroPsi-Projektstruktur (Quelle: [BAC+05])

3.1.Node Nets

Kern des MicroPsi-Konzeptes sind so genannte *Node Nets* [BAC+03], [VUI+03]. Nodes sind eine formalere und weiter gefasste Version dessen, was Dörner als Neuronen bezeichnet. In dieser Arbeit wird nur sehr oberflächlich auf Node Nets eingegangen, da sie für das Verständnis der Visualisierungsproblematik nicht notwendig sind; statt dessen soll nur ein Eindruck vermittelt werden, wie das MicroPsi Toolkit funktioniert.

Ein Nodenetz besteht aus Knoten *U* (*NetEntities*), Kanten *V* (*Links*), Datenquellen (*Data Sources*) und Datensenken (*Data Targets*). Eine Übergangsfunktion f_{net} berechnet aus dem aktuellen Zustand des Netzes den Folgezustand.

$NN = \{ U, V, \text{Data Sources, Data Targets, } f_{net} \}$

Jede NetEntity wird durch eine *id* eineindeutig bezeichnet. Eine NetEntity besitzt einen Vektor *I* von Eingängen (*Slots*), einen Vektor *O* von Ausgängen (*Gates*) sowie eine Aktivierung α. Eine Aktivierungsfunktion $f_{act} : I \rightarrow \alpha$ berechnet diese Aktivierung aus den Gates. Ein Link verbindet jeweils ein Gate mit einem Slot; diese Links propagieren Aktivierung durch das Netz. Eine aktivierte Node führt ihre *Node Function* $f_{node} : NN \rightarrow NN$ aus. Diese Notation soll aussagen, dass eine Node Function das Netz beliebig verändern kann.

Links besitzen jeweils eine Gewichtung (*weight*) im Bereich (-1, 1), eine Sicherheit (*certainty*) im Bereich [0, 1]. Sowohl Gewichtung als auch Sicherheit werden bei der Propagierung von Aktivierung mit der Aktivierungsstärke multipliziert; der Unterschied besteht darin, dass eine Gewichtung von 0 dazu führt, dass ein Link aus dem Netz entfernt wird, was bei einer Sicherheit von 0 nicht der Fall ist. Darüber hinaus besitzen Links einen vierdimensionalen Vektor, über den sie räumliche und zeitliche Koordinaten zwischen NetEntities propagieren können.

Gates besitzen einen Schwellenwert (*threshold*). Ein Gate propagiert seine Aktivierung nur dann über die mit ihm verbundenen Links weiter, wenn die Aktivierung den Schwellenwert überschreitet. Der Schwellenwert wird zuvor von der Aktivierung subtrahiert. Darüber hinaus lassen sich für jedes Gate Minimal- und Maximalwerte sowie ein Verstärkungsfaktor (*amplification factor*) für die propagierte Aktivierung festlegen.

Diese Architektur ist ausreichend, um neuronale Netze abzubilden. Die MicroPsi-Architektur definiert eine Reihe spezieller Typen von NetEntities. Als wichtigster Typ seien hier die *Concept Nodes* genannt, welche den Grundbaustein der MicroPsi-Node-Netze bilden. Concept Nodes haben einen einzigen Slot mit dem Namen *gen* (generic). Ihre Aktivierung ist identisch mit der an diesem Slot anliegenden Aktivierung. Die von Dörner verwendeten Linktypen *por, ret, sub* und *sur* bilden Concept Nodes als Gates ab. Por und ret stehen dabei für ursächliche Verkettung vor- bzw. rückwärts, sub und sur bezeichnen eine „Teil-von"- bzw. „Enthält"-Beziehung. Darüber hinaus besitzen Concept Nodes die Gates *cat* (*category*) und exp (*exemplar*) zur Kategorienbildung, die Gates *sym* (*symbol*) und *ref* (*reference*) zur Benennung von Konzepten sowie das Gate *gen,* welches die am gleichnamigen Slot anliegende Aktivierung bereitstellt.

Nodes können zu *NodeSpaces* gruppiert werden. Ein NodeSpace ist eine NetEntity, die ein komplettes Node Net kapselt und ihrerseits in ein NodeNet eingebunden ist. Dabei werden die Datenquellen des inneren Netzes auf die Slots des NodeSpaces abgebildet und die Datensenken werden auf die Gates abgebildet:

$S = \{ U^S, Data\ Sources^S, Data\ Targets^S, f_{net}{}^S \}$

Zwei weitere spezielle Nodetypen sind Sensoren (*sensors*) und Aktoren (*actors*). Sensoren besitzen ein einzelnes Gate vom Typ *gen*, welches den Wert einer externen Datenquelle (DataSource) bereitstellt. Aktoren besitzen einen Slot vom Typ *gen* und geben die dort anliegende Aktivierung an eine Datensenke weiter. Datenquellen und Datensenken können eine Verbindung in einen anderen NodeSpace herstellen oder mit externen Parametern der Simulation verbunden sein, d.h. über Sensoren kann ein Nodenetz Informationen aus einer realen oder simulierten Umwelt erhalten und über Aktoren kann diese Umwelt manipuliert werden.

Weiteren Nodetypen sollen hier nur erwähnt werden, für eine umfassende Diskussion sei auf das MicroPsi-Entwicklerhandbuch [VUI+03] verwiesen; eine aktualisierte Fassung des Handbuches ist zur Zeit in Vorbereitung.

- *Register Nodes* besitzen jeweils einen Slot und ein Gate vom Typ gen, die Aktivierung der Nodes entspricht der am Slot anliegenden Aktivierung.

- *General Activation Nodes* sind analog zu Register Nodes aufgebaut und erhöhen die Aktivierung aller Nodes im gleichen NodeSpace.

- *General Deactivator Nodes* sind die Gegenstücke zu General Activator Nodes; sie senken die Aktivierung alles Nodes im gleichen NodeSpace.

- *Associator Nodes* und *Disassociator Nodes* werden verwendet um Nodes innerhalb eines NodesSpaces miteinander zu verbinden bzw. Verbindungen zu entfernen.

- *Activators* sind Register, die mit den unterschiedlichen Gatetypen in einem NodeSpace korrespondieren. Ihre Gates werden von der Aktivierungsfunktion des NodeSpaces gelesen und zur Berechnung der Gates dieses NodeSpaces verwendet.

Unter Verwendung dieser Nodetypen können Agenten nach dem Vorbild von Dörners Psi-Theorie konstruiert werden.

Es gibt die Möglichkeit, so genannte *Native Modules* anzulegen. Dabei handelt es sich um NetEntities, deren Aktivierungs- und Nodefunktion direkt in Java implementiert sind. Native Modules können eine beliebige Anzahl von Slots und Gates besitzen. Native Modules können verwendet werden, um komplexe Funktionen auf übersichtlichere und effizientere Weise zu realisieren, als es mit herkömmlichen Nodenetzen möglich wäre.

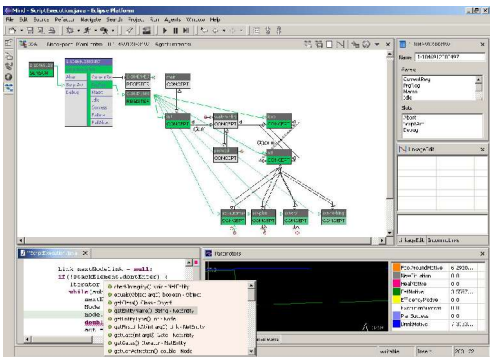


Abbildung 7: Ein graphischer Editor, die Mind Console, erlaubt das Erstellen und Editieren von Node Nets (Screenshot vom MicroPsi-Toolkit).

3.2.Das MicroPsi-Toolkit

Wie schon angedeutet ist das MicroPsi-Toolkit eine Experimentierplattform, die es erlauben soll, mit geringem Aufwand eine große Zahl verschiedenster Agenten zu konstruieren und sie in einer virtuellen oder auch realen Umgebung zu testen [VUI+03]. Das Toolkit ist – mit Ausnahme des 3DViewers und des 3DEmotion-Viewers, die Thema dieser Arbeit sind – komplett in Java entwickelt worden. Alle Komponenten können über ein Netzwerk miteinander kommunizieren, so dass es möglich ist, ein MicroPsi-System über beliebig viele Rechner verteilt aufzusetzen. Dadurch können beispielsweise sehr aufwändige Weltsimulationen oder rechenintensive Agenten auf dedizierten Rechnern untergebracht werden.

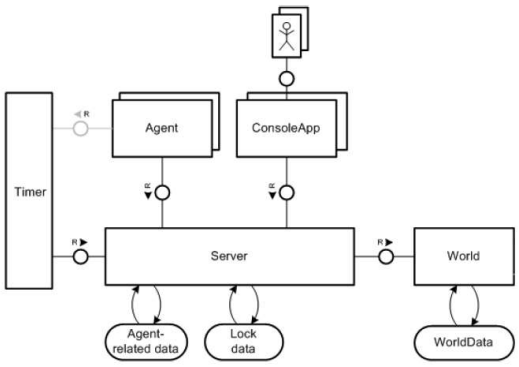


Abbildung 8: Die Architekur des MicroPsi-Toolkits (modifizierte Abbildung aus [VUI+03])

Zentraler Bestandteil des Toolkits ist ein *Server*, der Nachrichten zwischen den anderen Komponenten weiterleitet. Allein der Server weiß, wie die Komponenten im Netzwerk verteilt sind. Kapitel 8 beschreibt das Kommunikationssystem genauer.

Der *Timer* ist eine zentrale Zeitgeberkomponente. Er wird verwendet, wenn Agenten oder die Welt getaktet simuliert werden sollen (was bei den aktuellen Implementationen der Fall ist).

Die Weltkomponente *(World)* ist die Schnittstelle zu der Welt, in der die Agenten agieren können. Sie liefert den Agenten Wahrnehmungsdaten und ermöglicht ihnen die Ausführung verschiedener Aktionen. Momentan ist diese Komponente als Simulation einer einfachen virtuellen Welt implementiert; in Kapitel 7 wird näher auf diese Weltsimulation eingegangen. Die vorliegende Arbeit beschäftigt sich mit der Visualisierung dieser virtuellen Welt. Theoretisch wäre es jedoch möglich, die Weltkomponente als Schnittstelle zur realen Welt zu implementieren, um beispielsweise Roboter zu steuern. In diesem Fall würden die Wahrnehmungsdaten von echten Sensoren erzeugt werden und die vom Agenten gewünschten Aktionen würden durch einen Roboter in der realen Welt ausgeführt werden.

Jeder *Agent* ist ebenfalls eine Komponente. Eine unbegrenzte Anzahl von Agenten kann gleichzeitig im System aktiv sein. Agenten können auf beliebige Art implementiert sein; so lange sie die notwendigen Kommunikationsprotokolle beherrschen, können sie in das MicroPsi-Toolkit integriert werden. „Klassische“ MicroPsi-Agenten werden mit Hilfe der beschriebenen Nodenetze und gegebenenfalls Native Modules konstruiert. Das MicroPsi-Toolkit stellt Funktionen zum Erstellen, Editieren, Simulieren und Debuggen dieser Netze bereit.

Konsolen (*Consoles*) sind Komponenten, die Menschen die Interaktion mit den anderen Systemkomponenten ermöglichen. Alle MicroPsi-Konsolenapplikationen sind in die plattformübergreifende Open-Source-Entwicklungsumgebung Eclipse[1] eingebunden. Eclipse wurde ursprünglich von IBM entwickelt und 2001 als Open Source freigegeben. Ein Industriekonsortium, dem unter anderem Rational Software, Red Hat und Borland angehörten, wurde ins Leben gerufen, um die Weiterentwicklung von Eclipse zu überwachen. Viele weitere namhafte Firmen wie Oracle und Fujitsu traten dem Konsortium in den folgenden Jahren bei. 2004 wurde die Eclipse Foundation als not-for-profit corporation gegründet, die das das Projekt seither betreut.

Eclipse ist keine zwingende Voraussetzung für MicroPsi-Konsolenapplikationen. Der in dieser Arbeit entwickelte 3DViewer ist die erste MicroPsi-Konsolenapplikation, die, obwohl sie sich wie alle anderen Konsolen in Eclipse integrieren lässt, auch ohne Eclipse lauffähig ist.

Zur Zeit existieren neben dem hier vorgestellten 3DViewer drei weitere Konsolenapplikationen:

- Die *Admin Console* ermöglicht über ein Textinterface das Absetzen von Fragen und Kommandos an beliebige andere Komponenten. Desweiteren zeigt sie Logmeldungen aller Komponenten an.

- Die *World Console* visualisiert die von der Weltkomponente simulierte virtuelle Welt. Sie erlaubt das Editieren dieser Welt und die Steuerung der Weltkomponente. Der 3DViewer ist in erster Linie eine neue Implementierung dieser Konsole mit erweiterten Möglichkeiten; Näheres dazu beinhaltet Kapitel 7.

- Die *Mind Console* erlaubt das graphische Editieren, Simulieren und Debuggen von Node Nets (siehe Abb. 7, Seite 22). Diese Konsole unterliegt als einzige einer räumlichen Einschränkung; sie muss auf dem gleichen Rechner (genauer: in der gleichen Java Virtual Machine) laufen wie der zu bearbeitende Agent.

1 http://www.eclipse.org

4.Zielstellung

Die vorliegende Diplomarbeit beschäftigt sich mit der Entwicklung einer Visualisierungs- und interaktiven Steuerungskomponente für die MicroPsi-Multiagentenplattform. Das 3DView2-Projekt besteht mittlerweile aus zwei Komponenten, dem eigentlichen 3DViewer und dem 3D-Emotion-Viewer.

3DViewer

Die Hauptaufgabe der 3DViewer-Komponente ist, die existierende MicroPsi-Weltsimulation dreidimensional darzustellen. Der erste 3DViewer wurde als Studienarbeit auf Basis einer Open-Source-3D-Engine entwickelt [SAL04]. Die im Rahmen dieser Diplomarbeit entstandene Weiterentwicklung erhielt den Namen 3DView2, da es sich um eine komplette Neuentwicklung auf einer anderen technischen Grundlage handelt. 3DView2 basiert komplett auf Microsoft DirectX und ist in C++ geschrieben.

Es soll – ebenfalls im Gegensatz zur ersten Version von 3DView – die bestehende Kommunikationsinfrastruktur des MicroPsi-Toolkits genutzt werden, d.h. 3DView2 soll als MicroPsi-Konsolenanwendung implementiert werden, so dass seine Funktionsweise für das System völlig transparent bleibt. Die folgenden Aufgaben stehen bei der Entwicklung von 3DView2 im Vordergrund:

1. **Dreidimensionale Visualisierung der Weltsimulation**

   Der 3DViewer soll einem Experimentator ermöglichen, die Weltsimulation so zu sehen, wie es für einen Menschen am natürlichsten und intuitivsten ist – nämlich wie durch seine eigenen Augen. Er soll sich als Beobachter völlig frei in der virtuellen Welt bewegen, allerdings selbst keinen Einfluss darauf nehmen können.

2. **Interaktive Agentensteuerung**

   Es soll einem Menschen ermöglicht werden, die Weltsimulation aus der Perspektive eines Agenten zu erleben, d.h. er übernimmt die Kontrolle über den „Körper“ eines Agenten und hat exakt die gleichen Möglichkeiten wie ein künstlicher Agent und unterliegt auch den gleichen physischen Notwendigkeiten. Dieser Modus erlaubt Experimente in der Art, wie Dörner sie durchgeführt hat – also den direkten Vergleich des menschlichen Verhaltens mit dem eines künstlichen Agenten. Darüber hinaus ist eine interaktive Agentensteuerung eine wertvolle Hilfe beim Testen der Weltsimulation.

3. **Dreidimensionaler Welteditor**

   Der 3DViewer soll eine ähnliche Funktionalität bieten wie der zweidimensionale Welteditor; es soll möglich sein, neue Objekte in der Welt zu plazieren, sie zu manipulieren oder wieder zu entfernen.

4. **Graphisch anspruchsvolle Weltdarstellung**

   Der 3DViewer soll sich technisch und optisch an aktuellen Computerspielen orientieren. Dazu ist es notwendig, die Möglichkeiten moderner 3D-Graphikhardware auszunutzen. Die Hauptanforderungen an die graphische Darstellung sind:

   - Darstellung von dreidimensionalen Terrains mit hunderten Metern Durchmesser; Darstellung verschiedener Bodenmaterialien wie Gras, Stein, Sumpf oder Sand
   - Darstellung von ausgedehnten Wasserflächen, da sich die Weltsimulation an Dörners Insel-Szenario orientiert
   - gleichzeitige Darstellung von mehreren hundert Objekten in der Welt, insbesondere Vegetation
   - Animation von Objekten, insbesondere der Agenten
   - Darstellung von Echtzeitschatten bei Objekten, da Schatten eine wesentliche Rolle in der dreidimensionalen Wahrnehmung des Menschen spielen

3D-Emotion

3D-Emotion war ursprünglich ein eigenständiges Subprojekt innerhalb des MicroPsi-Projektes. Ziel ist, analog zu Dörners „Face“-Komponente in Psi Reality 2D [GER+99], Muskeln in einem virtuellen Gesicht getrennt voneinander ansteuern zu können, um bestimmte Emotionsausdrücke zu erreichen.

Ein erster Prototyp von Wiech und Naphtali basierte auf Java3D [WIE+04]. Im Rahmen dieser Arbeit sollte mit Hilfe eines zweiten Prototypen ermittelt werden, inwiefern die 3DViewer-Technologie auch für diese Komponente eingesetzt werden kann.

5.Basistechnologien

Die erste Version des 3DViewers basierte auf der Open-Source Graphikengine „The Nebula Device“ des Berliner Spieleentwicklungsstudios Radon Labs GmbH [SAL04]. Der ursprüngliche Plan war, für die Weiterentwicklung des 3DViewers die „Nebula2“-Engine [RAD05] zu verwenden. Nebula2 ist die direkte Weiterentwicklung der Nebula-Engine und ebenfalls unter einer Open-Source-Lizenz erhältlich. Die wesentlichste Neuerung in Nebula2 ist Ausrichtung der Engine auf die konsequente Nutzung von Pixel- und Vertexshadern. Obwohl die ersten Versuche sehr vielversprechend waren, wurde die Nutzung von Nebula2 nach einigen Wochen zugunsten einer anderen Technologie aufgegeben. Die wesentlichen Probleme mit Nebula2 waren:

- Bugs – zu dem Zeitpunkt, als die Entwicklung von 3DView2 begonnen wurde, war Nebula2 noch in einem relativ frühen Entwicklungsstadium. Viele Features waren noch nicht implementiert bzw. funktionierten nicht wie erwartet. Verlässliche Angaben, wann bestimmte Probleme beseitigt würden, konnten die Entwickler verständlicherweise nicht geben. Radon Labs entwickelt in erster Linie selbst Software auf Basis von Nebula2, entsprechend haben Probleme Vorrang, die hausinterne Projekte betreffen.

- Instabilität der Interfaces – jeder neue Release von Nebula2 brachte Änderungen im Interface und im Verhalten bestimmter Funktionen, so dass die Applikation aufwändig angepasst werden musste.

- Mangelnde Dokumentation – die einzige existierende Dokumentation wird automatisch aus dem Code und speziellen Kommentaren im Code erzeugt und ist allenfalls lückenhaft. Beispielprojekte gibt es nur aus der Open-Source-Community und auch nur sehr wenige.

- Zwei Codeversionen – Nebula2 ist Open Source, allerdings gibt es eine hausinterne Version, die sich zum Teil erheblich von der im Internet erhältlichen Open-Source-Version unterscheidet. Radon Labs arbeitet ausschließlich mit der hausinternen Version, nur in großen zeitlichen Abständen werden Änderungen propagiert.

- Kommerzielle Komponenten – Wesentlicher Bestandteil von Nebula2 ist das „Nebula2 Maya-Toolkit“, eine Sammlung von Tools und Plugins, die überhaupt erst den Export von Graphiken und Animationen in die Nebual2-Engine erlauben. Das Nebula2-Toolkit ist nicht Open Source, sondern ein kostenpflichtiges Programm – der aktuelle Preis beträgt 464 Euro pro Arbeitsplatz (Stand: Juni 2005). Es existieren kostenlose Exporter aus der Open-Source-Community, diese bieten jedoch nicht annähernd den gleichen Funktionsumfang. Radon Labs sagte mündlich eine kostenlose Beta-Tester-Lizenz für das Toolkit zu, ein offizieller Vertrag kam jedoch leider nicht zu Stande. Das Nebula2 Maya-Toolkit funktioniert nur mit der hausinternen Version von Nebula2, die zum Lieferumfang gehört.

- Architekturelle Zwänge – Nebula2 lässt sich nicht wie eine externe Graphikbibliothek benutzen sondern zwingt die Nutzer in eine rigide Softwarearchitektur. Beispielsweise müssen viele eigene Komponenten von Nebula-Klassen abgeleitet werden und das gesamte Projekt muss auf einem proprietären Buildsystem aufbauen.

Als Alternative bot sich die Nutzung einer eigenen Graphikengine („e42“) und einiger weiterer Bibliotheken, die im Rahmen eines anderen Projektes in Zusammenarbeit mit Daniel Matzke entstanden waren. Obwohl diese Engine noch nicht annähernd den Funktionsumfang von Nebula2 bot, schien sie eine verlässlichere und flexiblere Grundlage

für die Weiterentwicklung des 3DViewers zu sein. Neben e42 kommen in 3DView2 eine Reihe weiterer selbstentwickelter Bibliotheken zum Einsatz, beispielsweise eine einfache Soundengine (*SoundLib*) und eine Bibliothek für graphische Benutzerschnittstellen (*UILib*). Da diese Bibliotheken zum größten Teil schon vor Entwicklungsbeginn von 3DView2 existiert haben wird in dieser Arbeit nicht näher auf sie eingegangen werden. Statt dessen werden sich die folgenden Kapitel auf die Beschreibung von Technologien beschränken, die speziell für den 3DViewer entwickelt worden sind. Der komplette Quelltext von 3DView2 und aller beteiligter Bibliotheken steht über das CVS des MicroPsi-Projektes[2] bzw. auf der Begleit-CD dieser Arbeit zur Verfügung. Für nähere Informationen zum Quelltext und zur Projektstruktur siehe auch Anhang A.

e42 basiert, genau wie Nebula2, auf Microsoft DirectX [MS05] und ist somit nur unter Microsoft Windows lauffähig. (Für Nebula2 war eine OpenGL-Portierung geplant, die auch auf UNIX-Derivaten lauffähig gewesen wäre, diese ist bisher jedoch nicht zu Stande gekommen). Microsoft DirectX wurde mit Windows 95 eingeführt. Windows 95 war das erste Betriebssystem von Microsoft, das aufgrund seiner Struktur Anwendungsentwicklern keinen direkten Zugriff auf die Hardware gestattete. Dieser Zugriff ist für Echtzeitgraphikanwendungen wie z.B. Computerspiele jedoch zwingend notwendig; einerseits aus Geschwindigkeitsgründen, andererseits weil bestimmte Funktionen sich nur so realisieren lassen. Die meisten Computerspiele wurden somit weiterhin für MS-DOS entwickelt, was den Durchbruch von Windows auf einem entscheidenden Markt verhinderte. Abhilfe schaffte Microsoft DirectX, eine Middleware, die Windows-Anwendungen eine relativ direkte Kommunikation mit der Hardware erlaubt, ohne sich von konkreter Hardware abhängig zu machen.

Kernstück von DirectX ist ein *Hardware Abstraction Layer* (*HAL*), welches dem Anwender erlaubt, die Fähigkeiten der vorliegenden Hardware zu erfragen und diese Hardware über ein einheitliches Interface anzusprechen. Die konkrete Funktionalität wird in speziellen DirectX-kompatiblen Gerätetreibern realisiert, die in der Regel direkt vom Hardwareanbieter bereitgestellt werden. Bestimmte Funktionen können, wenn sie nicht direkt von der Hardware unterstützt werden, auf Wunsch auch von DirectX in Software emuliert werden, dazu steht ein *Hardware Emulation Layer* (*HEL*) bereit. DirectX besteht aus verschiedenen Komponenten:

- *DirectX Graphics* (früher unterteilt in *Direct3D* und *DirectDraw*) ermöglicht direkten Zugriff auf Graphikkarten.
- *DirectSound* erlaubt direkten Zugriff auf Soundkarten.
- *DirectInput* erlaubt die Abfrage von Eingabegeräten wie Mäusen, Keyboards, Joysticks, Gamepads, Lenkrädern u.ä. DirectInput unterstützt auch spezielle Funktionen von Eingabegeräten, beispielsweise *Force Feedback.* Force Feedback erlaubt die Kraftübertragung auf ein Eingabegerät, um beispielsweise Widerstände oder Vibrationen an Steuerknüppeln zu simulieren.
- *DirectMusic* ist ein Modul zur dynamischen Erzeugung situationsabhängiger Musik, findet jedoch in der Praxis kaum Verwendung.
- *DirectPlay* bietet Basisfunktionalität für Netzwerkkommunikation, wurde in der Praxis aber kaum verwendet und ist mittlerweile von Microsoft zugunsten der Windows-Netzwerkfunktionen zum Auslaufmodell erklärt worden.
- *DirectShow*: ermöglicht das Abspielen von Videos mit Hilfe im System installierter Codecs.

2 http://www.cognitive-agents.org/webdoc/space/CVS

Microsoft DirectX wird ständig weiterentwickelt, die aktuelle Versionsnummer ist 9.0c. DirectX kommt auch auf der Microsoft Spielkonsole X-Box zum Einsatz. Mit der bevorstehenden Einführung von Windows Longhorn wird DirectX durch die *Windows Graphics Foundation* (*WGF*) abgelöst werden.

Ein verhältnismäßig neuer Bestandteil von DirectX sind die *DirectX Extensions*. Hierbei handelt es sich um eine Sammlung von Hilfsfunktionen und Tools, die auf DirectX aufbauen und im Leistungsumfang zunehmend mit einer einfachen 3D-Engine vergleichbar sind. Die DirectX Extensions beinhalten z.B. ein eigenes Dateiformat für 3D-Modelle und Animationen (*X-Files*) und bieten sowohl passende Exportprogramme für verschiedene kommerzielle Graphikprogramme als auch eine Vielzahl von Funktion zum Laden, Konvertieren und Anzeigen dieser Dateien. Darüber hinaus enthalten sie einen Compiler, der Pixel- und Vertexshader aus einer Hochsprache, der so genannten *High Level Shading Language* (*HLSL*) just-in-time für die konkret vorliegende Graphikhardware kompiliert. Letztere Technik ist so überzeugend, dass die meisten Graphikengines für Windows, sowohl kommerzielle als auch nichtkommerzielle, diesen Compiler inzwischen integriert haben.

e42 bzw. 3DView2 nutzt sehr viele von den DirectX Extensions bereitgestellte Funktionen, beispielsweise das X-File-Format und den HLSL-Compiler. Da e42 im Gegensatz zu anderen Graphikengines wie z.B. Nebula2 sehr direkt auf DirectX aufbaut, profitieren Anwendungsentwickler auch direkt von der hervorragenden DirectX-Dokumentation, den Beispielapplikationen, die Microsoft selbst bereitstellt und der großen Anzahl von Drittpublikationen.

Darüber hinaus kommen in 3DView2 eine Reihe von Open-Source-Bibliotheken zum Einsatz, die hier kurz erwähnt werden sollen:

- **DevIL**[3] ist eine Open-Source-Bildbearbeitungsbibliothek. DevIL kann die meisten gängigen Bitmapformate lesen und schreiben, darunter TGA, BMP, PNG, JPG und GIF. Darüber hinaus kann DevIL Bilder skalieren, zwischen verschieden Farbformaten konvertieren und einfache Filter auf Bilder anwenden.

- **TinyXML**[4] ist eine simple Open-Source-XML-Bibliothek. TinyXML kann XML-Dateien und Streams lesen und schreiben. TinyXML will ausdrücklich eine minimale Bibliothek und kein kompletter Parser sein, es werden keine komplexen Features wie DTDs oder XSL unterstützt.

- **Lib Vorbis** and **Lib Ogg** [5]: OGG ist ein Containerformat für Audio- und Videodaten, Vorbis ist ein Open-Source-Audioformat mit psychoakustischer Kompression, vergleichbar mit MP3. Diese Bibliotheken ermöglichen das Dekomprimieren von OGG-Dateien zu Audiostreams, die anschließend von der Applikation abgemischt und abgespielt werden können.

- **Opcode**[6]: Opcode steht für *Optimized Collision Detection* und ist eine Open-Source-Bibliothek, die effizient Schnittpunkte zwischen geometrischen Primitiven und/oder dreidimensionalen polygonalen Objekten berechnen kann.

3 http://openil.sourceforge.net/
4 http://www.grinninglizard.com/tinyxml/
5 http://www.vorbis.com/
6 http://www.codercorner.com/Opcode.htm

6.Animation

6.1.Animationstechniken

Die Illusion einer kontinuierlichen Bewegung entsteht im menschlichen Gehirn durch eine schnelle Folge von Einzelbildern. Spielfilme arbeiten traditionell mit 24 Bildern pro Sekunde, bei der unter anderem in Deutschland gebräuchlichen Fernsehnorm PAL sind es 25. Damit eine Figur in einem Zeichentrickfilm sich bewegt, müssen Künstler eine entsprechende Anzahl von Einzelbildern zeichnen. Dabei wird in der Regel so vorgegangen, dass zunächst nur die wichtigsten Posen innerhalb der Animation gezeichnet werden; es wird dabei von *Keyframes* gesprochen. Später, wenn der Ablauf der gesamten Bewegung fest steht, werden die zwischen den Keyframes fehlenden Bilder von den Zeichnern ergänzt.

Keyframe Animation

Die ersten graphischen Anwendungen auf Computern waren genau wie Trickfilme zweidimensional und übernahmen somit die klassische Animationstechnik aus der Filmindustrie. Während die dritte Dimension in einer Zeichnung allerdings nur vorgetäuscht ist, ermöglichte der Einsatz von Computern ein Objekt oder eine Szene tatsächlich dreidimensional zu modellieren und daraus automatisch eine zweidimensionale Darstellung aus einem beliebigen Winkel errechnen zu lassen. Diese Arbeitsweise ist flexibler und ökonomischer und setzte sich daher sowohl in der Film- als auch in der Softwareindustrie durch.

Der einfachste Ansatz zur Animation dreidimensionaler Modelle ist vollkommen analog zur klassischen Trickfilmanimation. Die Animation wird aus vielen Einzelbildern zusammengesetzt, wobei jedes Einzelbild ein komplettes dreidimensionales Modell in der entsprechenden Pose ist. Der Speicherbedarf dieser Technik ist enorm hoch; pro Sekunde Animation fällt eine Datenmenge an, die der 24fachen Größe des Originalmodells entspricht. Die Datenmenge lässt sich durch Kompression verringern, ist jedoch für Echtzeitanwendungen in der Regel trotzdem nicht akzeptabel. Eine Alternative zur Speicherung aller Einzelbilder der Animation ist die Speicherung einer weitaus geringeren Zahl von Keyframes, zwischen denen dann nach Bedarf interpoliert wird. Um diese Interpolation effizient durchführen zu können, sollte zwischen den Einzelpunkten der Keyframe-3D-Modelle eine eineindeutige Abbildung existieren. Die Position jedes einzelnen Punktes lässt sich aus seinen jeweiligen Positionen in den beiden unmittelbar benachbarten Keyframes t_0 und t_1 berechnen. Im einfachsten Fall, der linearen Interpolation, lautet die Berechnungsvorschrift dazu

$$v = v_{t0} \cdot (1 - \beta) + v_{t1} \cdot \beta \quad ,$$

wobei β im Intervall [0, 1] liegt und dem linearen Zeitverlauf zwischen den Zeitpunkten t_0 und t_1 entspricht. Je nach Bedarf sind auch andere Arten der Interpolation möglich, beispielsweise quadratische Interpolation oder Interpolation auf einer S-Kurve. Die Reduzierung auf Keyframes senkt den Speicherbedarf deutlich, er ist aber noch immer sehr hoch. Bei komplexen Animationen ist eine große Anzahl von Keyframes notwendig, um durch die Interpolation bedingte visuelle Artefakte zu vermeiden.

Vertex Skinning

Beim Vertex Skinning wird in das 3D-Modell ein so genanntes Skelett eingefügt. Dieses Skelett ist eine Analogie zum Skelett beim Menschen; es besteht aus Knochen (*Bones*) und Gelenken (*Joints*). Das 3D-Modell (genauer: ein *Mesh* aus Polygonen) wird an das Skelett angebunden. Die Bewegung des Skelettes wird dann auf das 3D-Modell übertragen. Das Modell überzieht das Skelett quasi wie eine „Haut“, die allen Bewegungen folgt; daher der Begriff *Skinning*. Animiert werden muss nur noch das Skelett; bereits mit einer geringen Anzahl von Bones lässt sich ein fast beliebig komplexes 3D-Modell überzeugend animieren. Das Skelett selbst ist in der graphischen Darstellung später nicht sichtbar.

Grundbestandteil des Skelettes sind die Gelenke oder *Joints*. Die Joints bilden eine Hierarchie mit einem eindeutigen Wurzelelement. Jedes Joint besteht aus einem Positionsoffset und einer Rotation gegenüber seinem Vorgänger in der Hierarchie; diese Informationen werden meist als 4x4-Matrix abgelegt. Der Abstand zwischen einem Joint und seinem Vorgänger wird als *Bone* bezeichnet. Die gebräuchlichen 3D-Modellierungsprogramme modellieren Skelette jeweils etwas unterschiedlich. Maya von Alias Wavefront beispielsweise modelliert Skelette als eine Menge von Joints; Bones entstehen implizit zwischen zwei benachbarten Joints. Die Konkurrenzprodukte 3D Studio Max von discreet und Lightwave von NewTek hingegen modellieren Skelette als Menge von Bones. Jeder Bone hat eine Richtung und eine Länge, die Joints entstehen dabei implizit. Prinzipiell sind beide Ansätze äquivalent. In der Literatur werden beide Begriffe oft synonym gebraucht; wenn in der vorliegenden Arbeit in Zukunft von Bones die Rede ist, sind damit Joints gemeint, die den Anfang eines Bones markieren, d.h. das dem Wurzelelement näher gelegene Joint des Bones.

Der Begriff *Skinning* bezeichnet genau genommen nur den Prozess des Anbindens des Skelettes an das 3D-Modell. Die Verformung des Modells wird *Skinning Deformation* genannt. Die Anbindung des Skelettes geschieht in einer speziellen Position des Modells, der so genannten *Bind Pose* oder *Reference Pose.* Das eigentliche 3D-Modell und das Skelett werden in dieser Pose abgespeichert. Beim Anbinden des Skelettes werden jedem Punkt (*Vertex*) des Modelles ein oder mehrere Bones bzw. Joints zugeordnet. Diese Bones werden *Influences* genannt. Beim so genannten *Rigid Skinning* wird jedem Vertex genau ein Joint zugeordnet. Beim *Smooth Skinning* können jedem Vertex mehrere Bones zugeordnet werden, dabei wird mit jedem Bone ein Gewicht assoziiert, wobei diese Gewichte in der Summe den Wert eins ergeben müssen.

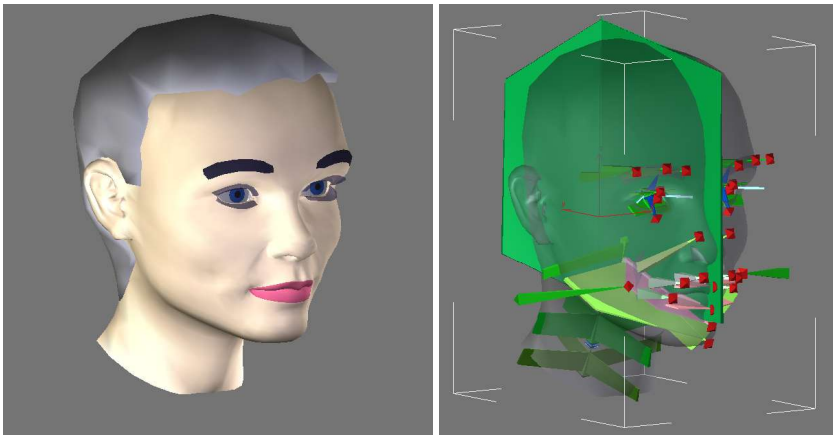


Abbildung 9: Das 3DEmotion-Gesicht in der Bind Pose (links) und das dazugehörige Skelett (rechts) in discreet 3D Studio MAX

Die Grundidee beim Skinning ist, dass die Position eines Vertex sich relativ zu den beeinflussenden Bones definieren lässt; das geschieht in der Bind Pose. Werden die Bones in eine andere Pose bewegt, bleibt der Vertex relativ zu seinen Bones jedoch in der gleichen Position. Das bedeutet, dass sich die neue Position des Vertex aus seiner Position in der Bind Pose, der Position der beeinflussenden Bones in der Bind Pose und der neuen Position der Bones berechnen lässt. Dazu wird der Vertex aus dem Koordinatensystem des 3D-Modells (*Model Space*) in das lokale Koordinatensystem eines beeinflussenden Bones in der Bind Pose (*Bone Space)* transformiert. Die Position des Vertex im Bone Space ist invariant; nach der Transformation des Bones kann der Vertex zurück in den Model Space transformiert werden.

Es sei X_i die Transformation von Joint J_i und Q_i die Rotation von Joint J_i gegenüber dem Vorgänger J_{i-1}. Die *Bind Pose Matrix* B_j eines Joints J_j errechnet sich folgendermaßen [GRE02]:

$$B_j = \prod_{i \equiv 0}^{j} X_i \cdot Q_i$$

Bei einer Animation wird das Skelett in eine andere Pose versetzt. Es sei T_i die Translation und R_i die Rotation eines Joints J_i in dieser neuen Pose. Die neue Matrix P_j eines Joints J_j errechnet sich analog zur Bind Pose:

$$P_j = \prod_{i \equiv 0}^{j} T_i \cdot R_i$$

Ein Vertex v wird in den Bone Space eines Bone J_j transformiert, indem er mit der inversen Bind Pose Matrix B_j^{-1} transformiert wird. B^{-1} wird daher auch als *Bone Offset Matrix* bezeichnet. Die Multiplikation mit P_j transformiert den Vertex zurück in den Model Space und gleichzeitig auf die korrekte neue Position.

$$v' = P_i \cdot B_i^{-1} \cdot v$$

Wird v von mehr als einem Bone beeinflusst, wird für jeden Bone eine neue Position des Vertex errechnet und die endgültige Position v' ist das gewichtete Mittel dieser Positionen. Dabei kommen die Gewichtungen der Bones bezüglich v zum Einsatz.

$$v' = \sum_{i \equiv 1}^{n} w_i \cdot v_i' \quad , \quad \sum_{i \equiv 1}^{n} w_i = 1$$

wobei v von n Bones beeinflusst wird und w_i die Gewichtung von Bone i bezüglich v ist. Eine Animation ist eine Sequenz von Posen für das Skelett, also eine Menge von Matrizen $P_j(t)$ für alle j und einen Zeitpunkt t. Im einfachsten Fall sind nur diskrete t zulässig und für jeden Zeitpunkt t und jeden Bone j wird eine Matrix abgespeichert. Speichereffizienter ist es jedoch, mit Keyframes für das Skelett zu arbeiten. Dabei wird für t ein kontinuierlicher Wertebereich zugelassen und für bestimmte Werte von t wird eine Pose des Skeletts abgespeichert. Um die Skelettpose für beliebige t zu ermitteln wird zwischen den beiden unmittelbar angrenzenden Keyframes interpoliert.

Translationen sind Vektoren im R^3 können direkt linear interpoliert werden; diese Operation wird auch als *LERP* (*Linear Interpolation*) bezeichnet:

$$t = (1-\beta)\cdot t_1 + \beta \cdot t_2 \quad ,$$

wobei β im Intervall [0, 1] liegt. Rotationen lassen sich als Matrizen darstellen, die Interpolation von Matrizen ist jedoch schwierig und führt zu einer Reihe von mathematischen Problemen. Wesentlich einfacher und unproblematischer ist die Darstellung von Rotationen als *Quaternions*. Quaternions sind prinzipiell vierdimensionale Vektoren, entsprechend können sie wie oben beschrieben linear interpoliert werden. Bessere Ergebnisse liefert jedoch die *Spherical Linear Interpolation* (*SLERP*), bei der die Gewichtungen folgendermaßen berechnet werden [SHO85]:

$$\cos\theta = r_1 \cdot r_2 \quad \text{, also} \quad \theta = \cos^{-1}(r_1 \cdot r_2) \quad ;$$

$$w_1 = \frac{\sin((1-\beta)\cdot\theta)}{\sin\theta} \quad \text{und} \quad w_2 = \frac{\sin(\beta\cdot\theta)}{\sin\theta} \quad .$$

$$r = w_1 \cdot r_1 + w_2 \cdot r_2$$

Auf die gleiche Weise, wie zwischen zwei Keyframes interpoliert werden kann, kann auch eine Interpolation zwischen zwei verschiedenen Animation berechnet werden. So ist es z.B. möglich, zwei oder mehr Animationen gleichzeitig auf dem gleichen Objekt abzuspielen oder einen weichen Übergang zwischen zwei Animationen zu realisieren.

Hardware Skinning

Wie bereits erläutert muss zur Darstellung des Meshes in einer gegebenen Pose jeder Vertex für jeden beeinflussenden Bone mit der kombinierten Matrix $P_iB_i^{-1}$ multipliziert werden. Moderne Graphikkarten können diese Matrixmultiplikationen und auch das anschließende gewichtete Summieren in Hardware ausführen. Die Anzahl Matrizen und Blendgewichte, die sich auf der Hardware laden bzw. gleichzeitig multiplizieren lassen, ist jedoch begrenzt. Viele Echtzeitanwendungen beschränken daher die Anzahl Bones, die einen einzelnen Vertex beeinflussen dürfen; üblich ist zur Zeit ein Limit von vier Influences pro Vertex.

Grundsätzlich wird zwischen *Indexed Skinning* und *Non-Indexed Skinning* unterschieden. Beim Non-Indexed Skinning wird das Mesh in Gruppen von Vertices unterteilt, die von den gleichen Bones abhängig sind. Diese Gruppen werden nacheinander und getrennt voneinander gerendert. Bevor eine Gruppe gezeichnet wird, werden die entsprechenden Bonematrizen auf die Graphikkarte geladen. Jeder Vertex enthält ein Blendgewicht für jede der geladenen Matrizen. Non-Indexed Skinning wird typischerweise verwendet, wenn die Graphikkarte nur vier Matrizen gleichzeitig laden kann. Vertices, die von weniger als vier Bones abhängig sind, können mit künstlichen Null-Gewichtungen beliebigen weiteren Bones zugeordnet werden.

Indexed Skinning ist eine Erweiterung dieser Technik für Graphikkarten, die eine größere Anzahl Matrizen laden können. Es werden *Matrix Palettes* (auch: *Bone Palettes*) von Bonematrizen gebildet, die gleichzeitig auf die Graphikkarte geladen werden. Jeder Vertex erhält zusätzlich zu den Blendgewichten Indizes, die angeben, von welchen Bones der Palette er abhängig ist. Somit lassen sich weniger und größere Gruppen von Vertices bilden, die dann gleichzeitig von der Graphikhardware bearbeitet werden. Indexed Skinning ist somit effizienter als Non-Indexed Skinning. Beide Skinningtechniken können als *Vertex Shader* realisiert werden.

6.2.Animation in 3DView2 und 3DEmotion

3DView2 und 3DEmotion verwenden Matrix Palette Skinning. Animationen werden mit Standard-3D-Modellierungsprogrammen wie Maya oder 3D Studio MAX erstellt und in das Microsoft X-File-Format exportiert. X-Files sind ein einfaches Dateiformat für 3D-Modelle und Animationen, das von Microsoft entwickelt wurde und über das DirectX-SDK Entwicklern zur Verfügung steht. Es gibt X-File-Exporter für praktisch alle gängigen 3D-Modellierungs- und Animationsprogramme; teilweise direkt von Microsoft, teilweise von Drittanbietern. Das DirectX-SDK stellt darüber hinaus Funktionen zum Laden und Bearbeiten von X-Files bereit. Dazu gehören unter anderem Funktionen zum automatischen Berechnen geeigneter Matrixpaletten für Indexed Skinning. Das eigentliche Skinning muss vom Anwendungsentwickler realisiert werden; 3DView2 und 3DEmotion realisieren Indexed Skinning mit Vertex Shadern. Vertex Shader können effizient in Software emuliert werden, wenn keine Hardwareunterstützung vorhanden ist.

3DEmotion

Die "Face"-Komponente in PSI Reality 2D von Dietrich Dörner [GER+99] visualisiert die simulierten „Gefühle“ eines künstlichen Agenten anhand eines schematisch dargestellten menschlichen Gesichts. Für das MicroPsi-Projekt sollte eine ähnliche Komponente geschaffen werden. Ein erster Prototyp von Wiech und Naphtali [WIE+04] verwendete *Facial Animation Parameters (FAPs)* nach dem MPEG4-Standard. Im Rahmen dieser Diplomarbeit entstand ein zweiter Prototyp, der auf Skinning Animation basiert. Dazu wurde von Künstlern ein dreidimensionales Modell eines menschlichen Kopfes erstellt und mit einem entsprechenden Skelett versehen. Für jeden Bone wurden zwei Keyframes definiert, welche die Extremstellungen dieses Bones definieren; d.h. alle gültigen Positionen dieses Bones lassen sich als (sphärisch-lineare) Interpolation dieser beiden Keyframes darstellen. Der Prototyp des 3DEmotion-Viewers stellt eine graphische Benutzeroberfläche bereit, über die sich jeder Bone einzeln in Echtzeit manipulieren lässt; so lassen sich verschiedene Gesichtsausdrücke erzeugen. (Weitere Bilder des Programms befinden sich im Anhang dieser Arbeit). Zu einem späteren Zeitpunkt ist die Integration der Komponente in das MicroPsi-Toolkit geplant. Die notwendige Infrastruktur zur Kommunikation mit dem Toolkit ist in 3DView2 bereits vorhanden, momentan fehlt aber noch eine Komponente im Toolkit, die „Emotionen“ eines Agenten auf das Gesicht übertragen könnte.

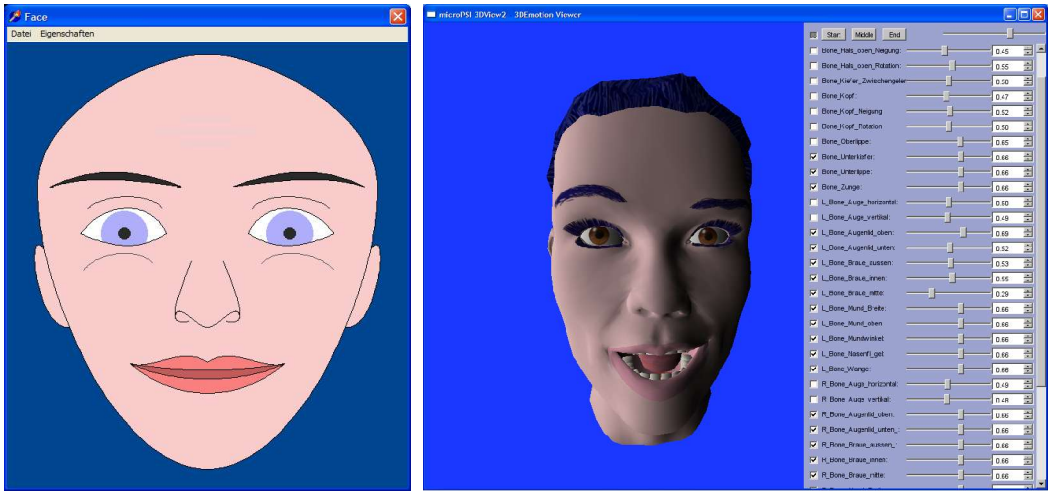


Abbildung 10: links: die "Face"-Komponente in PSI Reality 2D von Dietrich Dörner (Screenshot); rechts: Prototyp von MicroPsi 3DEmotion

7.Objektmanagement

Die Repräsentation der Welt wird im MicroPsi-Weltserver gehalten, 3DView2 soll diese Welt graphisch darstellen. Für den Server ist die kleinste Einheit ein Objektteil (*object part*). Objektteile bilden Bäume, d.h. jedes Objektteil kann Bestandteile (*sub parts*) enthalten. Ein Objektteil besteht aus:

- einer Objektklasse (String)
- einer eindeutigen Identifikationsnummer, auch ID genannt
- einer Position im dreidimensionalen Raum (Vektor)
- einer Ausdehnung im dreidimensionalen Raum (Vektor)
- einem Rotationswinkel in der X/Z-Ebene
- einer Menge von obligatorischen Eigenschaften (*properties*) (Schlüssel/Wert-Paare)
- einer Menge von optionalen Eigenschaften (Schlüssel/Wert-Paare)
- einer Menge von Bestandteilen (Liste von Objektteilen; leer, wenn Teil atomar ist)

Das Wurzelelement einer Hierarchie von Objektteilen wird als Objekt bezeichnet. Zusätzlich zu den genannten Eigenschaften besitzt ein Objekt einen Namen, ein Gewicht und und einen Zustand (*state*).

Die Unterteilung von Objekten in Bestandteile wurde eingeführt, um den MicroPsi-Agenten eine differenziertere Wahrnehmung zu ermöglichen; sie sollen in Zukunft in der Lage sein, Objekte anhand ihrer Bestandteile zu erkennen. Für die Visualisierungskomponenten, also den 2D-Welteditor und 3DView2, ist diese Einteilung zunächst nicht relevant, sie operieren auf Objekten als kleinster Einheit.

MicroPsi verwendet ein Koordinatensystem, bei dem die positive Z-Achse nach oben zeigt. Momentan betrachtet der Server die Welt allerdings als flach, d.h. alle Objekte, die unmittelbar auf dem Boden stehen, haben eine Z-Koordinate von Null. Die räumliche Ausdehnung und das Gewicht von Objekten werden momentan nicht konsequent von der Simulation genutzt, daher haben diese Attribute bei vielen Objekten keinen sinnvollen Inhalt.

Die Objektklasse bezeichnet den Typen des Objektes, d.h. spezifiziert das Verhalten des Objektes und die Interaktionsmöglichkeiten mit dem Objekt. Jede Objektklasse korrespondiert mit einer Javaklasse, die serverintern Objekte dieses Types simuliert.

Graphische Darstellungen sind für den Server transparent, d.h. der Server verfügt lediglich über Informationen zur Struktur der Welt, nicht aber über ihre Darstellung. Der 2D-Welteditor verwendet daher eine eigene XML-Datei, die Objektklassen Bitmaps zuordnet. Für jede Bitmap werden die Koordinaten des Mittelpunktes definiert; die Bitmap wird so dargestellt, dass ihr definierter Mittelpunkt mit den Koordinaten des Objektes zusammenfällt. Um die Rotation eines Objektes auch in der zweidimensionalen Darstellung deutlich zu machen, können verschiedene Bitmaps angegeben werden, die das Objekt aus verschiedenen Perspektiven zeigen. Da das Erzeugen solcher Bitmaps mit höherem Aufwand verbunden ist, existieren solche Ansichten nur für Objekttypen, deren Rotation relevant ist, insbesondere Agenten. Auf die Visualisierung des Terrains wird in Kapitel 9.6 näher eingegangen.

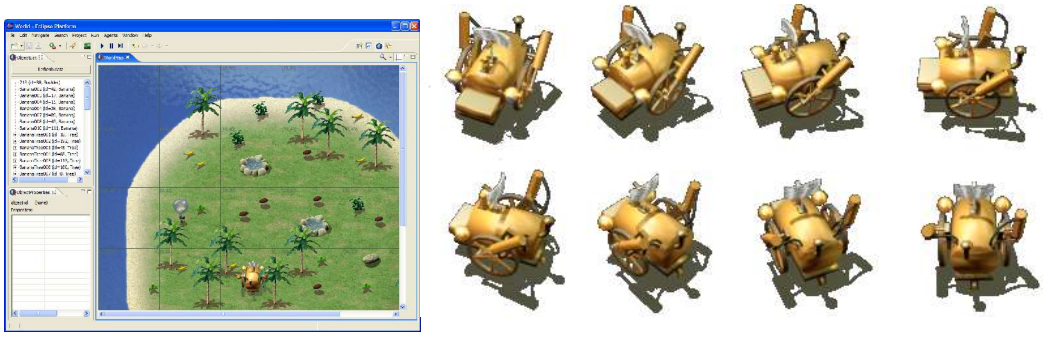


Abbildung 11: Der 2D-Viewer (links) verwendet Listen von Bitmaps (rechts), um Objekte aus verschiedenen Ansichten zu zeigen (Screenshots vom MicroPsi-Toolkit)

Serverseitig gibt es desweiteren das Konzept der *Welttypen*; der Welttyp legt fest, welche Objekte in einer Welt vorhanden sein können. Aktuell existiert nur der Welttyp „Doerner“, welcher an das von Dörner entworfene Inselszenario angelehnt ist.

Visualisierungen

3DView2 wählt den gleichen Ansatz wie der 2D-Welteditor und definiert alle zur graphischen Darstellung notwendigen Informationen, die der Server nicht direkt liefern kann, in eigenen XML-Dateien. Um zusätzliche Flexibilität zu erreichen, wurde das Konzept der *Visualisierungen* (v*isualizations*) eingeführt. Eine Visualisierung definiert das komplette Aussehen der Welt. Die gleiche Welt kann durch Austauschen der Visualisierung ein komplett anderes Aussehen erhalten. So ist es z.B. möglich, Visualisierungen für verschiedene Jahreszeiten zu definieren oder eine Welt wahlweise realistisch oder im Comicstil darzustellen. Für jeden existierenden Welttypen muss mindestens eine Visualisierung erstellt werden. Eine ausführliche Anleitung zum Erstellen von Visualisierungen befindet sich in Anhang A dieser Arbeit.

Als Minimum benötigt 3DView2 pro Objektklasse ein dreidimensionales Modell, das diesen Objekttypen in der 3D-Darstellung repräsentieren soll. Solche 3D-Modelle können – wie in Kapitel 6 erläutert – mit Standard-Graphikmodellierungssoftware erstellt und auf Wunsch auch animiert werden. Um eine größere optische Vielfalt zu ermöglichen, können pro Objektklasse mehrere solcher dreidimensionaler Modelle definiert werden; dabei wird von *Variationen* (*variations)* gesprochen. Mit welcher Variation ein konkretes Objekt dargestellt wird, kann jedoch nicht zufällig entschieden werden. Die Entscheidung muss deterministisch sein, da die Welt bei jedem Neustart des 3D-Viewers bzw. auch auf verschiedenen parallel laufenden Instanzen des 3D-Viewers exakt gleich aussehen soll. Da der Server keine Informationen über die graphische Darstellung besitzen soll, werden Variationen aufgrund einer bereits existierenden Information, nämlich der eindeutigen ID-Nummer eines Objektes, gewählt.

MicroPsi-Welten können tausende von Objekten beinhalten. Bei der dreidimensionalen Visualisierung solcher Welten können somit hunderte von Objekten gleichzeitig sichtbar sein; um zu gewährleisten, dass die Darstellung trotzdem mit angemessener Geschwindigkeit läuft, muss eine Vielzahl von Maßnahmen getroffen werden. Diese Maßnahmen beginnen bereits bei der Definition einer Visualisierung. Die Darstellung eines 3D-Modelles erfordert abhängig von dessen Komplexität unterschiedlich viel Zeit. Die genauen Zusammenhänge sind komplex und in hohem Maße abhängig von der im konkreten Fall verwendeten Hardware [RIG02]. Grundsätzlich ist es jedoch so, dass die Darstellung langsamer wird, je mehr Polygone (Dreiecke) ein Objekt enthält und je mehr verschiedene

Texturen oder Shader verwendet werden. Es muss ein Kompromiss zwischen Qualität der graphischen Darstellung und verfügbarer Leistung gefunden werden. Ein wesentlicher Umstand, der dabei ausgenutzt werden kann, ist die Tatsache, dass weiter vom Betrachter entfernte Objekte in der dreidimensionalen Projektion kleiner erscheinen und daher mit weniger Details dargestellt werden können, ohne dass der graphische Gesamteindruck beeinträchtigt wird. Daher ist es ratsam, die im Nahbereich notwendigen, sehr detailierten 3D-Modelle mit zunehmender Entfernung vom Betrachter durch weniger detailierte und entsprechend ressourcenschonende Versionen auszutauschen. Bei sehr kleinen Objekten ist es sogar möglich, ab einer bestimmten Entfernung komplett auf ihre Darstellung zu verzichten. Dieses Vorgehen bezeichnet man als *Level of Detail* oder kurz *LOD*-Algorithmus.

In einer *Visualisierung* für 3DView2 kann pro *Variation* eine beliebige Anzahl von *LOD-Stufen* angegeben werden. Eine LOD-Stufe besteht aus einem 3D-Modell und der Angabe, bis zu welcher Maximalentfernung vom Betrachter dieses Modell verwendet werden darf. Wenn für eine LOD-Stufe keine explizite Maximalentfernung angegeben wird, wird diese als unendlich angenommen. Jedem konkreten Objekt in der Welt wird wie beschrieben eine Variation zugeordnet. Abhängig von seiner aktuellen Entfernung zum Betrachter wählt das Objekt dann jeweils die LOD-Stufe mit der geringsten noch zulässigen Maximalentfernung (in der Annahme, dass dies das detailierteste aktuell zulässige Modell ist). Ein Beispiel: Objekte vom Typ „Baum“ besitzen zwei LOD-Stufen. Das detailierte 3D-Modell wird bis zu einer Maximalentfernung von 100 Metern vom Betrachter verwendet, danach kommt ein weniger detailiertes Modell zum Einsatz, für das keine Maximalentfernung angegeben wurde. Objekte vom Typ „Pilz“ hingegen besitzen nur eine LOD-Stufe, die bis zu einer Maximalentfernung von 30 Metern verwendet werden darf. Pilze, die weiter als 30 Meter vom Betrachter entfernt sind, werden nicht mehr dargestellt.

Die Erstellung von unterschiedlich detailierten 3D-Modellen für die verschiedenen Objekttypen sowie die Festlegung der Maximalentfernungen ist dem Benutzer überlassen.

Die Positionierung einiger Objekte ist vom Terrain abhängig. Eine Tanne steht auch an einem steilen Hang immer senkrecht. Der klassische MicroPsi-Agent bewegt sich jedoch auf drei Rädern fort und steht in Hanglage entsprechend schief. Da der MicroPsi-Weltserver momentan kein dreidimensionales Terrain modelliert, sondern die Welt als Ebene annimmt, bleibt die Modellierung dieser Zusammenhänge dem 3DViewer überlassen. Für jede Objektklasse (genauer: für jede *Variation*) lassen sich deshalb so genannte *Ground Contact Points* definieren. Dies sind drei Punkte im lokalen Koordinatensystem des 3D-Modelles, die in der 3D-Visualisierung Kontakt mit dem Boden haben sollen. Die Koordinaten dieser Punkte können direkt im 3D-Modellierungsprogramm abgelesen werden, unabhängig davon, welches Programm konkret verwendet wird. Im Fall des klassischen MicroPsi-Agenten müssen beispielsweise die drei Räder den Boden berühren. Zur Laufzeit des 3D-Viewers werden diese Punkte abhängig von Position und Rotation des Objektes in das Weltkoordinatensystem transformiert und die Höhe des Terrains an diesen Punkten wird bestimmt. Das Modell wird anschließend so rotiert, dass die angegebenen Punkte im Modell den Boden exakt berühren.

Für weitere Beispiele und die genaue Syntax der Visualisierungs-XML-Dateien sei noch einmal auf Anhang A verwiesen. Kapitel 9 beschäftigt sich mit der Terraindarstellung und wird ebenfalls noch einmal auf *Visualisierungen* zurückkommen.

Abbildung 12: Agenten müssen den Boden mit allen Rädern berühren (links); für viele Objekte gibt es neben den detailierten Modellen in verschiedenen Variationen (Mitte) auch niedriger aufgelöste LOD-Stufen (rechts)

Online- und Offline-Modus

Die wichtigste Aufgabe von 3DView2 ist die graphische Darstellung einer Welt, die von einem zentralen Weltserver verwaltet und simuliert wird. 3DView2 soll über Netzwerk mit diesem Server kommunizieren. Da die Bandbreite einer Netzwerkverbindung begrenzt ist und der 3DViewer eine Vielzahl von Informationen benötigt, über die der Server gar nicht verfügt, ist es unumgänglich, dass 3DView2 eine eigene Repräsentation der Welt aufbaut und diese zum Server synchron hält.

3DView2 kennt zwei grundlegende Betriebsmodi, den Offline- und den Onlinemodus. Im Offlinemodus besteht keinerlei Verbindung zu einem Weltserver, der 3DViewer läuft völlig autark. Dieser Modus ist für Entwicklungszwecke wie z.B. Testing und Debugging äußerst nützlich, außerdem kann er für Demonstrationen genutzt werden. Im Offlinemodus wird die gesamte Repräsentation der Welt aus einer XML-Datei geladen. Eine Weltrepräsentation besteht aus

- einem Namen,
- einem Beschreibungstext,
- der gewählten *Visualisierung* (Name der XML-Datei)
- einer Liste aller in der Welt vorhandenen Objekte mit ihren Attributen (Klasse, Position, Rotation usw.) sowie
- Informationen über das Terrain (siehe Kapitel 9.6).

Der in 3DView2 integrierte Welteditor manipuliert im Offlinemodus direkt die interne Weltrepräsentation. Die veränderte Welt kann wieder als XML-Datei gespeichert werden.

Im Onlinemodus verbindet sich der 3DViewer mit einem Weltserver und empfängt von ihm alle Informationen über die darzustellende Welt. Einige Informationen kann der Server jedoch nicht liefern, an erster Stelle ist hier das zu verwendende dreidimensionale Terrain zu nennen. Daher muss auch im Onlinemodus eine Weltrepräsentation aus einer XML-Datei geladen werden. Dabei wird das gleiche Dateiformat wie im Offlinemodus verwendet, allerdings wird die Objektliste aus der Datei ignoriert und bestimmte Informationen über Positionierung und Skalierung des Terrains, die eigentlich in der XML-Datei vorhanden wären, werden statt dessen vom Server übernommen.

Serverseitig wird zwischen *Welten* (*Worlds*) und *Karten* (*Maps*) unterschieden. Während Welten – genau wie im 3DView2-Offlinemodus – in XML-Dateien gespeichert werden, sind Maps lediglich Bitmapdateien, die das Terrain vorgeben. Jeder Welt ist eine Map

zugeordnet. Im Offlinemodus entsprechen die von 3DView2 verwendeten XML-Weltdateien den Weltdateien des Servers. Im Onlinemodus hingegen sind sie eher mit Maps vergleichbar, da die Objekte nicht aus der Datei, sondern vom Server kommen. Deshalb wurde – vor allem auf Wunsch der Benutzer – jeder Map auf dem Server auf Seiten des 3D-Viewers eine XML-Weltdatei zugeordnet.

Wenn 3DView2 eine Verbindung zu einem MicroPsi-Weltserver aufbaut, erfragt er zunächst grundlegende Informationen, insbesondere den Namen der aktuell vom Server verwendeten Map. Anschließend versucht 3DView2 eine gleichnamige XML-Weltrepräsentation zu laden. Die Datei enthält Informationen über die zu verwendende Visualisierung und das darzustellende Terrain; in der Datei eventuell vorhandene Objektlisten werden dabei ignoriert. Statt dessen wird vom Server eine komplette Liste aller in der Welt befindlichen Objekte angefordert. Der Server wird desweiteren aufgefordert, die 3DViewer-Komponente über alle Veränderungen an dieser Liste in Zukunft zu informieren. Im Onlinemodus manipuliert der in 3DView2 eingebaute Welteditor nicht die eigene Weltrepräsentation, sondern setzt Befehle wie das Erzeugen und Löschen von Objekten in entsprechende Anfragen an den Server um. Insofern der Server die Anfragen umsetzt informiert er den 3DViewer und alle anderen angeschlossenen Visualisierungskomponenten über die Veränderung; erst dadurch wird das Ergebnis der Aktion für den Benutzer auf dem Bildschirm sichtbar. Der Speichern-Befehl schreibt nicht wie im Offline-Modus die lokale Repräsentation der Welt in eine XML-Datei, sondern fordert vielmehr den Server auf, seine Repräsentation der Welt als Datei abzuspeichern. Kapitel 8 beschäftigt sich ausführlicher mit der Kommunikation zwischen verschiedenen MicroPsi-Komponenten.

Effiziente Objektverwaltung

Bei der Kommunikation zwischen 3DViewer und Server werden Objekte immer durch ihre eindeutige ID-Nummer bezeichnet. 3DView2 benutzt daher eine Hashfunktion, um zu einer gegebenen ID-Nummer schnell die eigene Objektrepräsentation auffinden zu können.

Beim Rendering bzw. bei Kollisionstests – beispielsweise dem Strahlentest, wenn ein Objekt im Editor angeklickt wird – ist die schnelle räumlich Lokalisierung von Objekten notwendig. 3DView2 verwendet für diesen Zweck einen *Quadtree [EBE01]*. Ein Quadtree ist eine Datenstruktur, die eine Fläche hierarchisch in immer kleinere Segmente unterteilt. Das Pendant zu Quadtrees im dreidimensionalen Raum sind *Octrees*, da die Psi-Welt jedoch keine signifikante Ausdehnung in der Höhe hat, ist ein Quadtree ausreichend.

Abbildung 13 zeigt als Beispiel die ersten drei Ebenen eines Quadtrees, wie er von 3DView2 zum schnellen Auffinden von Objekten aufgrund räumlicher Kriterien verwendet wird, sowie zwei Weltobjekte.

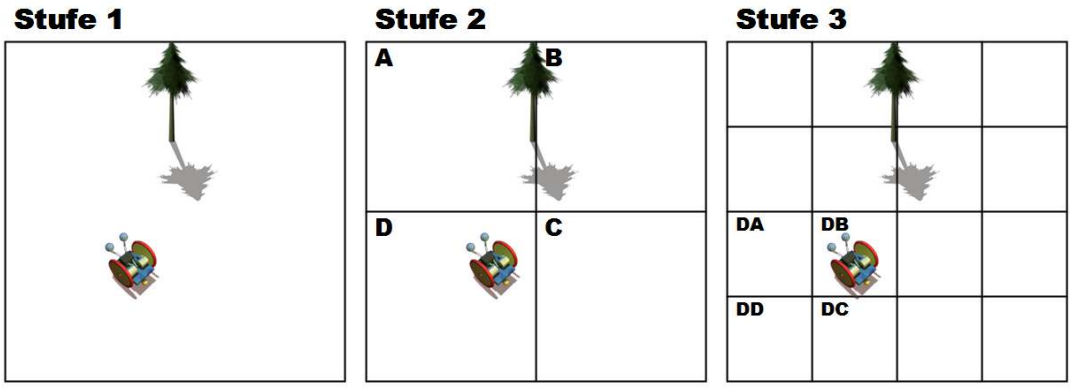


Abbildung 13: Einordnung von Weltobjekten in einen Quadtree

Im Beispiel sollen zunächst ein Agent und eine Fichte in den Quadtree einsortiert werden. Von beiden Objekten ist die Position in Weltkoordinaten sowie die räumliche Ausdehnung bekannt. Der 3DViewer verwendet *Bounding Spheres* zur Modellierung der räumlichen Größe eines Objektes; das sind Kugeln, definiert durch Mittelpunkt und Radius, die das Objekt komplett einschließen. Für die Arbeit mit dem Quadtree ist nur die zweidimensionale Ausdehnung des Objektes in der xz-Ebene relevant.

Das Wurzelelement des Baumes repräsentiert das gesamte Terrain. In einem Quadtree hat jeder Knoten genau vier Kinder. Die Kinder eines jeden Knoten partitionieren die von ihrem Elternknoten abgedeckte Fläche in vier gleich große Teile. Jeder Knoten enthält eine Liste von Objekten, die ihm zugeordnet sind. Ein Objekt wird einem Knoten zugeordnet, wenn seine Fläche vollständig in der von diesem Knoten repräsentierten Fläche enthalten ist. Dabei soll jedes Objekt möglichst tief in der Baumstruktur platziert werden.

Das Einordnen eines Objektes in den Baum geschieht rekursiv. Das Agentenobjekt in der Abbildung kann in jedem Fall dem Wurzelknoten zugeordnet werden. Eine Überprüfung seiner Kinder zeigt, dass der Agent auch dem Knoten D zugewiesen werden könnte. Eine Überprüfung dessen Kinder führt zur Zuordnung des Agenten zum Knoten DB. Im Beispiel hat der Quadtree nur drei Ebenen, daher ist diese Einordnung endgültig. Die im Beispiel dargestellte Fichte kann auf der zweiten Ebene nicht eindeutig zugeordnet werden, da sie teilweise in A und teilweise in B liegt; daher wird sie dem Wurzelelement zugewiesen.

Abbildung 14 zeigt beispielhaft *View Frustum Culling [EBE01]* mittels eines Quadtrees. Ziel ist, effizient zu ermitteln, welche Objekte sich aktuell im Sichtkegel (*View Frustum,* genauer: Sichtkegelstumpf) des Betrachters befinden. Der Sichtkegel wird rekursiv gegen den Quadtree getestet. Alle Knoten, deren Flächen vom Sichtkegel geschnitten werden oder darin enthalten ist, sind teilweise oder ganz sichtbar. Alle diesen sichtbaren Knoten zugeordneten Objekte gelten als potentiell sichtbar.

Der Sichtkegel ist immer im Wurzelelement des Quadtrees enthalten. Im Beispiel werden auf der zweiten Ebene nur die Kinder A und B geschnitten, die Kinder von C und D werden hingegen nicht weiter betrachtet. Somit steht bereits zu diesem Zeitpunkt fest, dass das Agentenobjekt aktuell nicht sichtbar ist. Alle potentiell sichtbaren Objekte werden anschließend noch einmal einzeln gegen das View Frustum geprüft. 3DView2 verwendet bei diesem Test die erwähnte Bounding Sphere des Objektes und überprüft, ob sie im View Frustum enthalten ist oder dieses schneidet. Dies ist eine grobe, konservative Prüfung; ein genauer Test der tatsächlichen Objektgeometrie gegen den Sichtkegel wäre angesichts des geringen Zusatznutzens zu aufwändig.

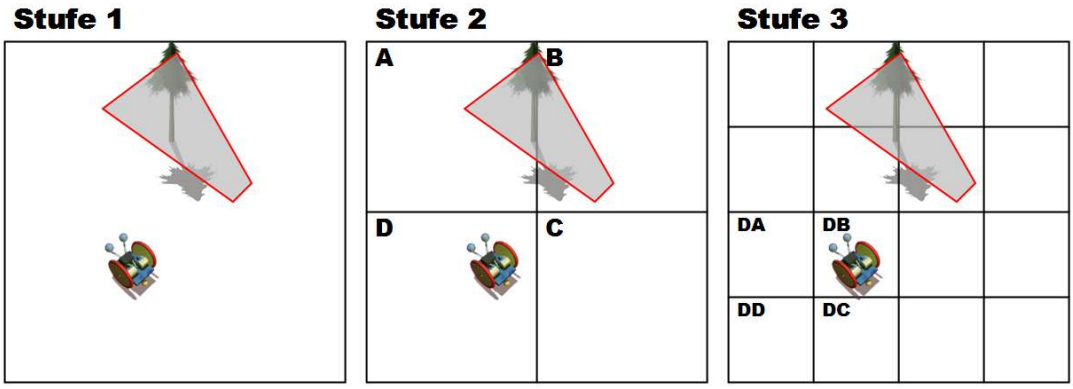


Abbildung 14: View Frustum Culling mittels Quadtree

8.Kommunikation mit dem MicroPSI Framework

Die erste Version des 3DViewers kommunizierte über ein proprietäres binäres Protokoll über TCP/IP mit dem MicroPsi-Weltserver [SAL04]. Obwohl dieser Kommunikationsmodus auch von 3DView2 noch immer unterstützt wird, ist ein wesentliches Ziel dieser Arbeit die Umstellung von 3DView2 auf das XML-Kommunikationssystem, über das alle existierenden MicroPsi-Komponenten miteinander kommunizieren. Da es bisher keine schriftliche Dokumentation dieses Systems gibt, soll hier näher darauf eingegangen werden. Die Vorstellung wird sich auf die für diese Arbeit relevanten Teile beschränken.

8.1.Das XML-Kommunikationsschema

Das MicroPsi-Framework besteht aus *Komponenten*, die miteinander kommunizieren können. Jede Komponente kann unabhängig von den anderen auf einem eigenen Rechner laufen. Grundlage der Kommunikation sind XML-Strings, die über das HTTP-Protokoll zwischen den Komponenten ausgetauscht werden. Prinzipiell wären auch andere Kommunikationsprotokolle, wie z.B. der direkte Datenaustausch über TCP/IP, denkbar, HTTP hat jedoch den Vorteil, dass es selbst in sehr restriktiv konfigurierten Netzwerken in der Regel problemlos funktioniert.

Die Kommunikation zwischen den Komponenten läuft über eine zentrale *Serverkomponente;* sie kennt alle aktuell aktiven Komponenten und weiß, wie sie zu erreichen sind. Wenn eine Komponente mit einer anderen kommunizieren möchte, richtet sie einen *Request* an den Server. Ein Request wird mittels einer HTTP-Post-Operation übertragen. Requests haben die Form:

```
<req id="Requesthandler" sender="Quellkomponente"> Payload </req>
```

Requesthandler bezeichnet einen Dienst innerhalb des MircoPsi-Netzwerkes. Der *Consoleservice* (id="console") ermöglicht es Komponenten, Fragen und Antworten auszutauschen. Über diesen Dienst werden Anzeigekomponenten und Editoren realisiert, beispielsweise der 2D-Welteditor und der 3DViewer. Der *Agentservice* (id="agent") hingegen ermöglicht Agenten die Verbindung zum System. Agenten können über diese Schnittstelle Aktionen ausführen und Wahrnehmungsreize empfangen. *Quellkomponente* bezeichnet die Komponente, die den Request gesendet hat. Jede Komponente im System muss eine eindeutige Bezeichnung haben. *Payload* ist ein XML-String; abhängig vom gewählten Dienst sind verschiedene Inhalte zulässig.

Jeder Request wird mit einer *Response* beantwortet:

```
<resp id="Service" type="Typ"> Payload </resp>
```

Im Payload der Response befinden sich Nachrichten an die sendende Komponente, beispielsweise Anworten auf gestellte Fragen. Das Payload der Response ist nicht notwendigerweise die Reaktion auf den zuvor gestellten Request, denn die Weiterleitung des Requests an die zuständige Komponente und die Bearbeitung des Requests können eine gewisse Zeit in Anspruch nehmen – insbesondere, wenn die Komponenten im Netzwerk verteilt sind. Responses sind die einzige Möglichkeit für eine Komponente, Daten vom System zu empfangen. Es ist somit notwendig, in regelmäßigen Abständen Requests an das System zu richten – notfalls leere Requests – um Responses zu erhalten. Tatsächlich werden Komponenten, die eine bestimmte Zeit keine Requests gesendet haben, automatisch vom System abgemeldet (*Timeout*).

8.2.Kommunikation mit dem Console Service

Das Payload eines *Requests* an den Console Service ist ein einzelner *Console Request.* Requests können zwar auch leer sein, sie können jedoch nicht mehrere Console Requests enthalten. Console Requests haben die Form:

```
<m300 type="Requesttyp"> Console Questions </m300>
```

Der *Requesttyp* gibt an, ob es sich um den ersten (Wert = 1) oder den letzten (Wert = 2) Request dieser Komponente handelt. Andernfalls sollte der Requesttyp den Wert 0 aufweisen. Dieser Mechanismus wird genutzt, um Komponenten im System an- und abzumelden. Als Antwort auf einen *Console Request* erhält die Komponente eine *Console Response* (aber nicht notwendigerweise sofort):

```
<m301 time="Time" text="Komponentenname"> Console Answers </m301>
```

Für den ersten Request kann eine Komponente einen beliebigen Namen für sich selbst wählen. Wird dieser Name bereits von einer anderen am System angemeldeten Komponente genutzt, wählt der Server einen anderen Namen für die neue Komponente und teilt ihr diesen im „Text"-Attribut der nächsten Console Response mit. Ist dies nicht der Fall ist das „Text"-Attribut leer. Die neuen Namen werden aus dem „Wunschnamen" der Komponente und einer fortlaufenden Nummer gebildet. Wenn einer Komponente ein neuer Name zugewiesen wird, muss sie diesen bei allen zukünftigen Requests benutzen – andernfalls wird sie keine Antworten erhalten. *Time* ist ein Integerwert und gibt an, in welchem Zeitschritt die Antwort erstellt wurde.

Console Requests können eine beliebige Anzahl von *Console Questions* enthalten. Questions können Fragen im Sinne von Bitten um Informationen sein, sie können jedoch auch Anfragen sein, also die Bitte, etwas bestimmtes zu tun. Questions haben die Form:

```
<m302 dest="Zielkomponente" step="Zeitschritt" am="Antwortmodus"
origin="Quellkomponente" qname="Frage" param0="Parameter 0"
param1="Parameter 1" ... />
```

Zielkomponente bezeichnet eine andere Komponente im System, an die die Frage gerichtet ist. Eine typische Zielkomponente ist „world", die Weltsimulation. 3DView2 richtet praktisch alle Fragen an diese Komponente. *Quellkomponente* bezeichnet wiederum diejenige Komponente, welche die Antwort auf die Frage erhalten möchte. *Zeitschritt* ist der Zeitpunkt, zu dem die Frage gestellt wird. Die aktuelle Zeit lässt sich bei der Timerkomponente erfragen. *Frage* ist der Name der eigentlichen Frage; welche Fragen verstanden werden, hängt von der befragten Komponente ab. Fragen können eine beliebige Anzahl Parameter haben. *Antwortmodus* ist einer der folgenden Werte:

- 0: *Don't Answer:* auf die Frage folgt keine Antwort
- 1: *Stop Answering*: das System hört auf, die Frage wiederholt zu beantworten
- 2: *Answer Once:* die Frage wird genau einmal beantwortet
- 3: *Answer Continously:* die Frage wird in jedem Simulationsschritt erneut beantwortet
- 4: *Answer every 5 Steps:* die Frage wird alle 5 Simulationsschritte erneut beantwortet
- 5: *Answer every 10 Steps:* alle 10 Simulationsschritte
- 6: *Answer every 50 Steps:* alle 50 Simulationsschritte
- 7: *Answer every 100 Steps:* alle 100 Simulationsschritte

Mit Ausnahme der Antwortmodi 0 und 1 erfolgen auf jede Frage ein oder mehrere Antworten in Form von *Console Answers*. Diese haben die Form:

```
<m303 type="Antworttyp" step="Zeitschritt" dest="Zielkomponente"
origin="Quellkomponente"> Payload </m303>
```

Im Payload einer Answer befindet sich immer eine Kopie der dazugehörigen Question. Was darüber hinaus im Payload enthalten ist, kann am *Antworttyp* abgelesen werden. Zulässige Werte sind:

- 0: *String:* die Antwort ist ein ein einfacher Textstring
- 1: *OK*: die Frage wurde erfolgreich bearbeitet, die Anwort ist leer
- 2: *Error:* es ist ein Fehler aufgetreten, Payload ist evtl. eine textuelle Fehlermeldung
- 3: *Complex Answer:* komplexe Antwort, d.h. die Antwort ist ein XML-String

Komplexe Antworten können bisher nur einen einzigen Typen von XML-Tags enthalten, die so genannten *MTreeNodes*. Eine MTreeNode hat die Form

```
<m304 name="Name" value="Wert"> ... </m304>
```

und kann beliebig viele weitere MTreeNodes enthalten. MTreeNodes bilden also einen Baum von Schlüssel-Wert-Paaren. Mit Hilfe dieser Struktur werden komplexe Daten zwischen Komponenten ausgetauscht.

8.3.Kommunikation mit der Weltsimulation

Die Weltsimulation ist über den *Console Service* erreichbar und kann zu jedem Zeitpunkt über den Zustand der Simulationswelt Auskunft geben. Der 2D-Welteditor und 3DView2 visualisieren die Welt auf Basis der so erfragten Daten. Die wichtigsten *Questions*, die von der Weltkomponente beantwortet werden können, sind:

getobjectlist (keine Paramter) – liefert die vollständige Liste aller momentan in der Welt befindlichen Objekte und alle Attribute dieser Objekte. Die Antwort auf eine (erfolgreiche) Anfrage ist z.B.

```
<m304 name="success" value="">
  <m304 name="object list" value="">
    <m304 name="object" value="">
      <m304 name="id" value="139"></m304>
      <m304 name="position" value="(618.09, 420.86, 0.0)"></m304>
      <m304 name="orientation" value="0.0"></m304>
      <m304 name="objectclass" value="PineTree"></m304>
      <m304 name="objectname" value="PineTree-158"></m304>
    </m304>
    <m304 name="object" value="">
      <m304 name="id" value="963"></m304>
      <m304 name="position" value="(304.97, 522.01, 0.0)"></m304>
      <m304 name="orientation" value="0.0"></m304>
      <m304 name="objectclass" value="TreeStump"></m304>
      <m304 name="objectname" value="TreeStump-92"></m304>
    </m304>
    ...
  </m304>
</m304>
```

getobjectchangelist (keine Parameter) – liefert eine Liste aller Objekte, die sich seit der letzten gleichartigen Anfrage verändert haben. Diese Frage wird typischerweise mit Antwortmodus 3 (*Answer Continously*) gestellt, so dass der Weltserver nach jedem Simulationstakt automatisch eine Liste aller geänderten Objekte an den Client schickt. Es

können jedoch nur Änderungen erfragt werden, die nicht mehr als 20 Simulationstakte zurückliegen (diese Zahl ist konfigurierbar). Bei Überschreiten dieser Grenze muss die komplette Objektliste erfragt werden, andernfalls läuft der Client Gefahr, asynchron zum Server zu werden. Die Antwort auf diese Frage sieht wie folgt aus:

```
<m304 name="success" value="">
  <m304 name="object change list" value="">
    <m304 name="object" value="">
      <m304 name="id" value="1528"></m304>
      <m304 name="position" value="(146.12, 277.48, 0.0)"></m304>
      <m304 name="orientation" value="0.0"></m304>
      <m304 name="objectclass" value="PineTree"></m304>
      <m304 name="objectname" value="PineTree-1"></m304>
    </m304>
  </m304>
  <m304 name="remove" value="1529"></m304>
  <m304 name="globals version" value="2"></m304>
</m304>
```

In diesem Beispiel enthält die Liste der Änderungen zwei Objekte. Das Objekt mit der ID 1528 ist entweder neu hinzugekommen oder es hat sich seit dem letzen Simulationstakt mindestens eines seiner Attribute verändert – in jedem Fall enthält die Änderungsliste das komplette Objekt mit allen seinen Attributen. Das Objekt mit der ID 1529 hingegen ist aus der Welt entfernt worden.

Das letzte Tag der Liste mit dem Namen *Globals Version* ist in jeder Änderungsliste enthalten und teilt dem Client die Versionsnummer der globalen Daten der Simulationswelt mit. Globale Daten sind beispielsweise Informationen über den Typ der Welt oder das Terrain. Jede Veränderung an den globalen Daten inkrementiert die Versionsnummer; wenn ein Client eine Veränderung der Versionsnummer registriert, kann er die globalen Daten neu anfordern.

getglobaldata (keine Parameter) – Diese Frage liefert alle Informationen über die Welt, die nicht mit Objekten zusammenhängen. Ein Client stellt diese Frage typischerweise einmalig nach dem Öffnen einer Verbindung zum Server und danach jedes Mal, wenn sich die *Globals Version* ändert. Die Antwort hat das Format:

```
<m304 name="success" value="">
  <m304 name="filename" value="world.xml"></m304>
  <m304 name="groundmap" value="">
    <m304 name="area" value="">
      <m304 name="lower bound" value="(0.0, 0.0, 0.0)"></m304>
      <m304 name="upper bound" value="(800.0, 800.0, 0.0)"></m304>
    </m304>
    <m304 name="image filename" value="doernerisland.png"></m304>
  </m304>
  <m304 name="visible area" value="">
    <m304 name="lower bound" value="(0.0, 0.0, 0.0)"></m304>
    <m304 name="upper bound" value="(800.0, 800.0, 0.0)"></m304>
  </m304>
  <m304 name="version" value="2"></m304>
</m304>
```

Übertragen werden der Name der Datei, aus der die Welt geladen wurde (*filename*), Größe und Position des Terrains in Weltkoordinaten (*lower bound*, *upper bound*) sowie der Name der *Material Map* (*image filename,* siehe dazu Kapitel 9).

saveworld (Parameter 0: *Dateiname* (optional)) – fordert die Weltkomponente auf, den aktuellen Zustand der Welt lokal auf ihrem Rechner in einer Datei zu speichern. Wird kein Dateiname übergeben, so wird die Datei verwendet, aus der die Welt ursprünglich geladen wurde.

loadworld (Parameter 0: *Dateiname*) – fordert die Weltkomponente auf, eine Welt aus der angegebenen XML-Datei zu lesen. Die Datei muss lokal bei der Weltkomponente liegen.

getexistingworldfiles (keine Parameter) – liefert eine Liste der auf dem Rechner der Weltkomponente lokal vorhandenen Weltdateien.

createobject (Parameter 0: *Klassenname*, Parameter 1: *Name*, Parameter 2: *Position*) – fordert die Weltkomponente auf, ein neues Objekt zu erzeugen. *Klassenname* muss eine bekannte Objektklasse bezeichnen. *Name* ist der Name des zu erstellenden Objektes und kann ein beliebiger String sein; ist er leer, wird der Name des neuen Objektes automatisch aus der Objektklasse und einer fortlaufenden Nummer generiert. *Position* bezeichnet die Weltkoordinaten, an denen das Objekt erstellt werden soll, im Format *(x,y,z)*.

deleteobject (Parameter 0: *ID*) – fordert die Weltkomponente auf, das Objekt mit der übergebenen ID zu löschen. Agentenobjekte können auf diese Weise nicht gelöscht werden, da Agenten eigene Komponenten im System sind und durch Objekte in der Welt nur repräsentiert werden.

getagentobjectid (Parameter 0: *Agentenname*) – liefert die ID des Objektes, das den Agenten mit dem übergebenen Namen repräsentiert.

Die hier vorgestellten *Questions* sind nur eine Auswahl; die vollständige Liste kann am leichtesten über die *Admin Console* im MicroPsi Toolkit recherchiert werden. Die Admin Console zeigt Fragen und Antworten leider nicht als XML, sondern nur in einer Baumansicht.

8.4.Kommunikation mit dem Agent-Service

Agenten sind unabhängige MicroPsi-Komponenten. Über den *Agent Service* können sich Agenten am System anmelden, bekommen einen *Avatar* zugewiesen – das ist eine Repräsentation als Objekt in der Weltsimulation – können Aktionen in der Welt durchführen und erhalten Wahrnehmungsdaten.

Analog zum Console Service werden an den Agent Service *Agent Requests* gerichtet:

```
<m200 type="Requesttyp" agent="Agentenname" atype="Agententyp">
Questions / Actions </m200>
```

Der *Requesttyp* gibt an, ob es sich um die Anmeldung eines neuen Agenten (Wert = 0), eine normale Operation eines bereits angemeldeten Agenten (Wert = 1) oder eine Abmeldung (Wert = 2) handelt. Jeder Agent muss sich als erste Aktion anmelden und kann nach der Abmeldung keine weiteren Aktionen ausführen. *Agentenname* ist ein selbst gewählter Name des Agenten. *Agententyp* muss einen dem System bekannten Typ von Agenten bezeichnen, beispielsweise *SteamVehicleAgent* (ein Dampfmaschinen-Agent nach dem Vorbild von Dörners Psi) oder *BraitenbergVehicleAgent* (ein Agent nach dem Vorbild der Braitenbergvehikel, die auf Licht reagieren). Die Angabe des Agententyps ist nur bei einer Anmeldung erforderlich; der Typ kann später nicht mehr verändert werden.

Ein Agent Request kann neben beliebig vielen Questions, die analog zum Console

Service beantwortet werden, auch *Agent Actions* enthalten. Der Agent ist auf eine Action pro Simulationstakt beschränkt. Agent Actions haben die Form

```
<m202 type="Aktionstyp" agent="Agentenname" oid="ObjektID"
ticket="Ticketnummer" param0="Parameter0" param1="Paramter1" .../>
```

Aktuell werden fünf verschiedene *Aktionstypen* unterstützt:

- *NOOP*: No Operation – der Agent führt in diesem Simulationstakt keine Aktion aus.
- *MOVE*: Bewegung – der Agent bewegt sich in der Welt; dabei geben *Parameter0* und *Parameter1* die zurückgelegte Strecke in x- bzw. y-Richtung in Weltkoordinaten an.
- *EAT*: Essen – der Agent versucht, das Weltobjekt mit der ID *ObjektID* zu essen.
- *DRINK:* Trinken – der Agent versucht, das Weltobjekt mit der ID *ObjektID* zu trinken bzw. aus diesem Objekt zu trinken
- *FOCUS:* Fokussieren – der Agent richtet seine Aufmerksamkeit auf das Objekt mit der ID *ObjektID*. Diese Aktion ist für die Wahrnehmung bedeutsam und wird an dieser Stelle nicht weiter erläutert, da für die Fernsteuerung des Agenten durch einen Menschen keine speziellen Wahrnehmungsdaten erforderlich sind. Menschen können auf Basis des vom 3DViewer generierten Bildes der Welt agieren.

Auf jeden Agent Request folgt eine Agent Response:

```
<m201 time="Time" type="201" ctext="Agentenname"> Agent Action
Responses / Questions </m201>
```

Die Agent Response kann einen neuen Agentennamen enthalten, wenn der Server gezwungen war, den Agenten aus Gründen der Eindeutigkeit umzubenennen. Jede Agent Response enthält im Payload Antworten auf gestellte Fragen sowie *Agent Action Responses.* Auf jede Aktion ausser NOOP folgt eine Agent Action Response*:*

```
<m203 agent="Agentenname" succ="Erfolgsmeldung" ticket=
"Ticketnummer"/>
```

Aktionen können erfolgreich verlaufen oder fehlschlagen, entsprechend bekommt der Agent über *Agent Action Responses* eine Rückmeldung. Jeder *Agent Action* kann eine beliebige Ticketnummer mitgegeben werden. Diese Ticketnummer wird in der *Agent Action Response* wiederholt, damit der Agent die Rückmeldungen, die wie alle anderen Meldungen auch zeitlich verzögert sind, seinen Aktionen zuordnen kann.

8.5.Kommunikationsinfrastruktur in 3DView2

Zur Abwicklung der HTTP-Kommunikation verwendet 3DView2 die Microsoft Foundation Classes (MFC). Das Einbinden der MFC verlangt eine spezielle Konfiguration des Projektes, die das Einbinden anderer Bibliotheken verkompliziert. Daher wurde mit der *HTTPLib* eine Bibliothek geschaffen, die in transparenter Weise die von 3DView2 benötigte Funktionalität mit Hilfe der MFC realisiert und als *DLL* (*Dynamic Link Library*) eingebunden werden kann. Hierdurch vermeidet 3DView2 das direkte Einbinden der MFC.

Der zentrale Bestandteil des Kommunikationssystemes ist die Klasse *CCommunicationModule*. Sie bietet Zugang zu den Schnittstellen *CWorldController* und *CRemoteAgentController*. *CWorldController* dient der Manipulation der Welt; über dieses Interface können z.B. Objekte erzeugt oder gelöscht werden. *CRemoteAgentController* ermöglicht die Erzeugung und Steuerung von Agenten in der Welt. Beide Schnittstellen sind je nach aktuellem Kommunikationsmodus unterschiedlich realisiert. Zur Zeit existieren drei Kommunikationsmodi:

- XML over HTTP: Standard-Kommunikationsmethode aller MicroPsi-Komponenten. Dieser Modus ist der einzige, der alle Interaktionsmöglichkeiten bietet.

- Proprietäres TCP/IP: Dieser Kommunikationsmodus wurde für die erste Version des 3DViewers verwendet [SAL04]. In diesem Modus ist 3DView2 lediglich in der Lage, Änderungen von der Welt zu empfangen, kann die Welt selbst aber nicht manipulieren oder Agenten steuern. Obwohl dieser Modus veraltet ist, wurde er in das neue System integriert; möglicherweise wird es in Zukunft einen *XML over IP*-Modus geben, der Teile der Technologie wiederverwendet.

- Offline: im Offline-Modus (siehe auch Kapitel 7) besteht keine Verbindung zu einem MicroPsi-Server. Die Weltrepräsentation des 3DViewers kann in diesem Modus direkt manipuliert werden, das Steuern von Agenten ist jedoch nicht möglich.

Die Klasse *CHTTPCommunicationService* übernimmt die Kommunikation im *XML over HTTP*-Modus. Manipulation der Welt bzw. Steuerung von Agenten wird durch die Klassen *CXMLWorldController* und *CXMLRemoteAgentController* realisiert. Diese Klassen sind nicht auf die Kommunikation über HTTP, sondern lediglich auf das XML-Nachrichtenformat festgelegt. Das Kommunikationssystem von 3DView2 repliziert lose die Nachrichten-Klassenstruktur des MicroPsi-Toolkits mit der Basisklasse *CMessage* (Abbildung 15, unten; das MicroPsi-Toolkit verwendet eine analoge Klassenstruktur in Java). Nachrichten werden als Instanzen dieser Klassen erzeugt und erst beim Versenden in XML-Strings konvertiert. Analog erzeugt das Nachrichtensystem aus eingehenden XML-Strings die korrespondierenden Nachrichtenklassen, bevor die Nachrichten an ihre endgültigen Empfänger innerhalb der Applikation weitergeleitet werden.

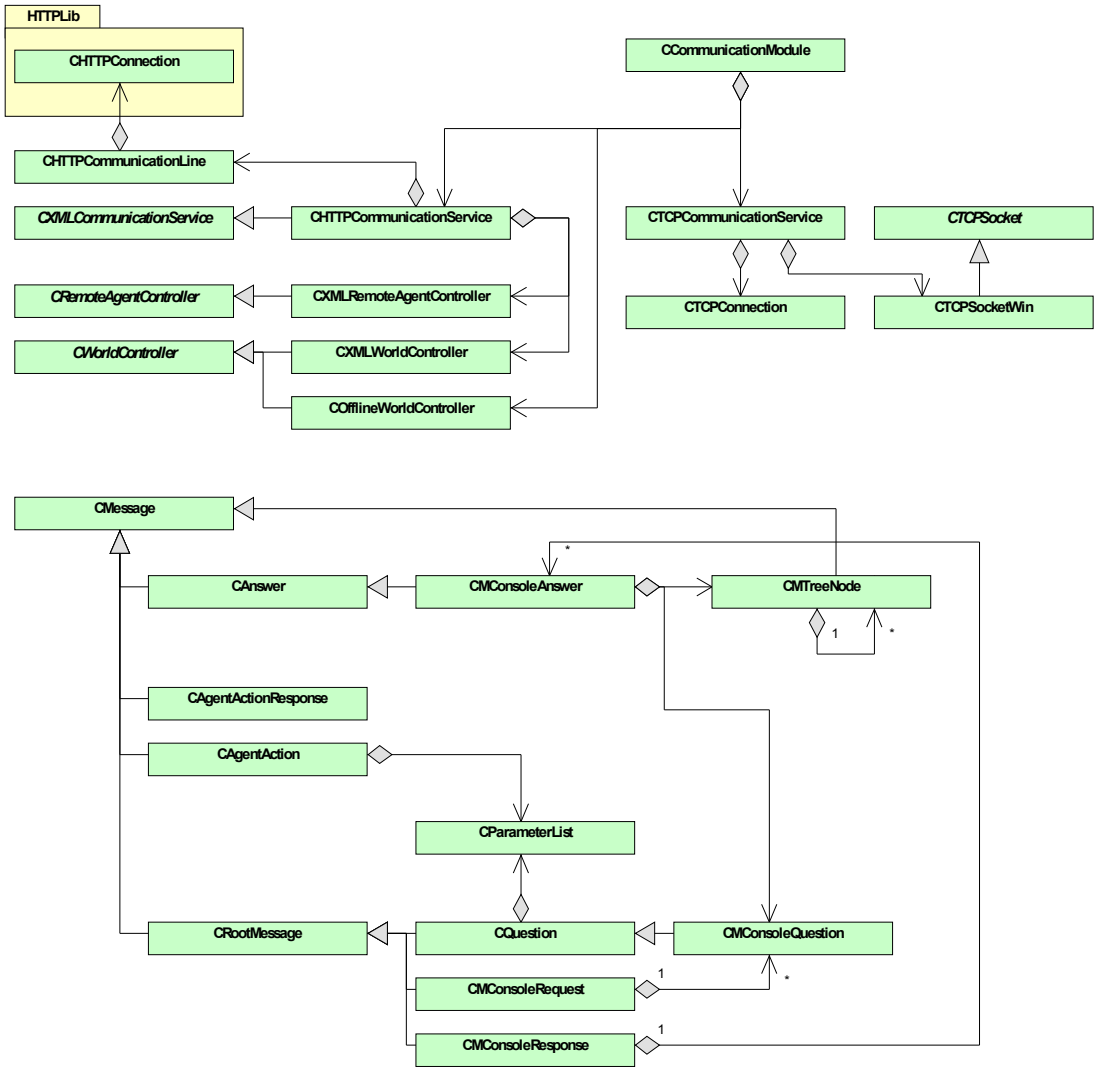


Abbildung 15: Kommunikationsinfrastruktur in 3DView2 (UML-Klassendiagramm)

Der MicroPsi-Server meldet Komponenten, die keine Requests senden, nach einer gewissen Zeitspanne automatisch ab. Diese Zeitspanne ist kurz, bereits das Laden eines neuen Terrains seitens des 3DViewes kann ausreichen, um einen solchen *Timeout* zu verursachen. Um dieses zu vermeiden, laufen mehrere Komponenten des 3DView2-Kommunikationssystems in eigenen Threads. Eine dieser Komponenten ist die Klasse *CHTTPCommunicationLine.* Sie nimmt abzusendende Nachrichten entgegen und speichert sie in einer Warteschlange. Die Nachrichten werden so schnell wie möglich asynchron zur restlichen Applikation zugestellt. Von außen eingehende Nachrichten werden ebenfalls in einer Warteschlange gespeichert und können von der Applikation zu einem geeigneten Zeitpunkt abgeholt werden. WorldController und RemoteAgentController generieren in eigenen Threads ständig Nachrichten an den Server, unabhängig davon, in welchem Zustand sich die restliche Applikation befindet.

9.Terraindarstellung

Für die glaubhafte Darstellung einer virtuellen Welt ist eine gute Terraindarstellung essentiell. Mit Terraindarstellung sei an dieser Stelle zunächst nur die Darstellung der reinen Geländeform und deren Oberflächenstruktur gemeint. Der Begriff kann weiter gefasst und auf die Darstellung von Wasserflächen und Vegetation ausgedehnt werden – auf diese Punkte wird in späteren Kapiteln näher eingegangen.

9.1.Datendarstellung von Terrain

Height Fields / Elevation Maps

Die Begriffe Height Field, Height Map und Elevation Map können synonym verwendet werden. Sie bezeichnen die einfachste und auch gebräuchlichste Möglichkeit, Geländeformen zu repräsentieren. Das Gelände wird als rechteckig angenommen und mit einem zweidimensionalen, quadratischen Raster überzogen. An jedem Rasterpunkt wird die Bodenhöhe über Normalnull oder einem beliebigen anderen Referenzwert ermittelt. Die Auflösung des Rasters und der Höhenwerte kann nach Bedarf beliebig gewählt werden.

Eine wesentliche Einschränkung von Height Fields ist die Tatsache, dass die Geländeform tatsächlich nur zweidimensional erfasst wird. Höhlen, Brücken oder Tunnel können mit Height Fields nicht dargestellt werden. Die Geländeform zwischen zwei Rasterpunkten wird in der Regel interpoliert. Bei natürlichen Terrainformen mit sanften Höhenverläufen werden damit zumeist sehr gute Ergebnisse erzielt, sehr steile oder gar komplett senkrechte Gefälle gehen dabei jedoch verloren. Eine Erhöhung der Auflösung des Height Fields reduziert das Problem, kann es jedoch nicht grundsätzlich lösen. Height Fields eignen sich somit sehr gut für die Darstellung natürlicher Terrains, jedoch weniger für Stadtszenarien oder sehr schroffe Geländeformen.

Eine sehr beliebte Darstellungsform von Height Fields sind Graustufenbitmaps. Dabei werden dunkle Bildpunkte als niedrig und helle Bildpunkte als hoch gelegen interpretiert. Die ursprünglichen Höhenwerte müssen, um etablierte Bildformate nutzen zu können, auf 8 bzw. 16 Bit Genauigkeit diskretisiert werden, was für die meisten Anwendungen völlig ausreichend ist. Für diese Darstellungsform spricht vor allem, dass eine große Anzahl von Bildbearbeitungsprogrammen und -bibliotheken existiert, die zur Erstellung und Bearbeitung von Height Fields verwendet werden können.

Voxel

Voxel ist ein Kunstwort und steht für *Volume Pixel*, wobei Pixel wiederum ein Kunstwort für *Picture Element* ist. Voxelbilder sind dreidimensionale Bitmaps, wobei jede Zelle entweder gefüllt oder leer ist. Im Gegensatz zu Height Fields erlauben Voxelbilder eine echte dreidimensionale Repräsentation einer Szene, erfordern jedoch wesentlich mehr Speicherplatz und sind komplexer in der Handhabung, da zu ihrer Bearbeitung spezielle Software erforderlich ist.

Voxelbilder werden z.B. bevorzugt in der Medizintechnik für Aufnahmen innerer Organe eingesetzt. Für Geländedarstellungen wird meist den einfacheren Height Fields der Vorzug gegeben, insofern nicht eine sehr komplexe Geländeform den Einsatz von Voxeln notwendig macht.

Es gab und gibt Echtzeitanwendungen im Standard-PC-Bereich, die von Voxeln Gebrauch machen, jedoch sind 3D-Graphikkarten zur Zeit komplett auf die Darstellung von Polygonen ausgelegt; daher muss eine Voxeldarstellung entweder direkt von der CPU übernommen oder von der CPU in eine polygonale Darstellung konvertiert werden. Anwendungen, die direkt auf polygonbasiertes Rendering setzen, haben daher meist einen Leistungsvorteil.

Meshes

Meshes sind Netze bzw. einfache Listen von Dreiecken. Da 3D-Graphikkarten auf Dreiecken operieren, ist es bei jeder Datendarstellung notwendig, die Terraindaten vor dem Rendering in eine Meshdarstellung zu überführen. Selbstverständlich ist es auch möglich, das Terrain von vornherein als Mesh zu entwerfen. Es gibt eine große Auswahl von Standardsoftware, mit der sich Meshes direkt erstellen und bearbeiten bzw. auch aus anderen Datenformaten generieren lassen. Meshes erlauben im Gegensatz zu Height Fields beliebige dreidimensionale Formen, also auch Überhänge, Brücken u.ä.

Meshes sind ein populäres Mittel zur Terraindarstellung in Computerspielen oder ähnlichen interaktiven Anwendungen, bei denen ein hoher graphischer Anspruch besteht und Landschaften in der Regel von Künstlern entworfen werden. Im Gegensatz zu anderen Datendarstellungen erlauben Meshes ein sehr hohes Maß an Kontrolle über die endgültige visuelle Darstellung und ermöglichen das Ausarbeiten feinster Details im Terrain. Sie sind das genaueste und flexibelste der hier vorgestellten Datenmodelle, jedoch gleichzeitig das komplexeste bzw. arbeitsintensivste.

Die Tatsache, dass Meshes ohne weitere Verarbeitung für das Rendering eingesetzt werden können, kann ein Vorteil sein. Anderseits muss bedacht werden, dass bei hohen Sichtweiten aus Geschwindigkeitsgründen meist eine dynamische Vereinfachung der Terraingeometrie erforderlich wird. Es entfällt also nicht unbedingt ein Verarbeitungsschritt – im Gegenteil, das Bearbeiten von Meshdaten ist deutlich komlexer als beispielsweise das Bearbeiten eines Height Fields oder einer Voxeldarstellung. Der nächste Abschnitt beschäftigt sich ausführlicher mit den Kernproblemen der Terraindarstellung.

9.2.Probleme bei der Terraindarstellung

Bevor auf konkrete Algorithmen eingegangen wird, sollen zunächst kurz die Kernprobleme bei der Geländedarstellung erläutert werden:

1. Die Rohdaten sind sehr groß.

   Die Größe der Rohdaten ist abhängig von der Größe des Geländes und von der gewünschten Auflösung. Generell werden die Daten selbst bei kompakter Darstellung schnell sehr groß. Rechenbeispiel: Ein Height Field von 10 mal 10 Kilometern mit einer Auflösung von einem Meter und 32 Bit Präzision bei den Höhenwerten belegt unkomprimiert über 380 MB Speicher. Bei 50 mal 50 Kilometern sind es bereits über 9 Gigabyte. Angesichts dieser Zahlen wird klar, dass es nicht sinnvoll ist, diese Datenmengen im Arbeitsspeicher halten zu wollen. Für die meisten Anwendungen wird ohnehin nur ein Bruchteil der Gesamtdaten gleichzeitig benötigt; es sind daher Strategien gefragt, die es erlauben, die erforderlichen Daten dynamisch zu laden und nicht mehr benötigte wieder zu verwerfen.

2. Die Terraindaten werden in sehr unterschiedlichen Detailstufen genutzt.

   Bei einem kilometergroßen Gelände werden nicht alle Daten gleichzeitig zur Darstellung benötigt, da der Betrachter gar nicht das gesamte Gelände mit allen Details wahrnehmen kann. Aus mehreren hundert Metern Höhe können zwar viele Quadratkilometer überblickt werden, dafür ist das kleinste Detail, das am Boden aufgelöst werden kann, vielleicht 30 Meter groß. Verringert der Betrachter seine Flughöhe, werden mehr Details sichtbar und das Sichtfeld schrumpft. Steht der Betrachter auf dem Boden, kann er direkt zu seinen Füßen Details im Zentimeterbereich erkennen, Berge am Horizont jedoch nur sehr grob.

   Hierbei wird von *Level of Detail* oder kurz *LOD-Algorithmen* gesprochen. Dabei liegen die Daten in unterschiedlichen Auflösungen oder Detailgraden vor; durch die Wahl einer geeigneten LOD-Stufe werden Speicherplatz und Rechenzeit eingespart.

   Die Herausforderung besteht einerseits darin, die Datenstrukturen so zu gestalten, dass ein effizienter Zugriff auf die Geländedaten in unterschiedlichen Auflösungen gewährleistet ist, andererseits muss bei der Darstellung darauf geachtet werden, dass die Wechsel des Detailgrades nicht zu abrupt vonstatten geht. Meist soll dem Betrachter eine flüssige Bewegung durch die Szene ermöglicht werden, daher müssen auch die Übergänge zwischen Detailstufen fließend gestaltet werden. Andernfalls „springen“ Details plötzlich ins Bild, was vom Betrachter als störend empfunden werden kann.

3. Der benötigte Detailgrad ist blickrichtungsabhängig.

   Der für ein Terrainstück benötigte Detailgrad ist abhängig von der Entfernung des Betrachters zu und seinem Blickwinkel auf das Terrain. Terraindarstellungsalgorithmen müssen daher sehr schnell und dynamisch auf Änderungen der Blickrichtung reagieren können.

4. Die Graphikleistung ist limitiert.

   Unabhängig davon, ob Raytracing oder polygonbasiertes Rendering, Echtzeitdarstellung oder vorberechnete Animation – die Graphikleistung ist limitiert und abhängig von der Komplexität der darzustellenden Geometrie. Dies ist ein weiterer Grund, für unterschiedlich weit entfernte Objekte unterschiedliche Auflösungen der Rohdaten zu verwenden. Nichtsdestotrotz bleibt die Optimierung des eigentlichen Renderingprozesses sehr wichtig. Sie kann die Wahl der Datenstrukturen und der LOD-Techniken ganz wesentlich beeinflussen, wie in Kapitel 9.4 erläutert werden wird.

5. Terrain ist nicht natürlich zerlegbar.

   Alle bisher angesprochenen Punkte haben impliziert, dass das Terrain in irgend einer Weise zerlegt werden muss – die komplette Datenmenge passt nicht in den Speicher, es sind immer nur Teile im Bild, weiter entfernte Geländeformationen brauchen weniger Details usw. Eine solche Unterteilung ist jedoch immer komplett künstlich, denn Terrain ist eine einzige, geschlossene Oberfläche und so werden die Rohdaten in der Regel auch repräsentiert. Die verschiedenen Algorithmen, die hier vorgestellt werden, benutzen unterschiedliche Methoden der Terrainzerlegung. Teilweise führen diese Zerlegungen zu neuen visuellen Problemen, die dann ebenfalls gelöst werden müssen.

9.3.CPU-orientierte Techniken

Algorithmen zur Terraindarstellung werden mittlerweile in zwei Klassen eingeteilt: die CPU-orientierten und die neueren, GPU-orientierten Techniken. Es ist sinnvoll, die Verfahren in chronologischer Reihenfolge ihrer Entwicklung zu betrachten, daher werden zunächst die CPU-orientierten Techniken erläutert und erst später darauf eingegangen, warum die Einführung von 3D-Graphikkarten zu einem Umdenken geführt hat.

Die in diesem Abschnitt vorgestellten Verfahren konzentrieren sich in erster Linie auf das Problem der limitierten Graphikleistung und versuchen, die darzustellende Geometrie zu reduzieren, ohne die Qualität der Darstellung übermäßig zu beeinträchtigen. Die Möglichkeiten moderner 3D-Graphikhardware werden dabei noch nicht berücksichtigt, somit wird implizit davon ausgegangen, dass die Arbeit komplett von der CPU erledigt wird.

9.3.1.Progressive Meshes

Hoppe stellt in [HOP96] mit den *Progressive Meshes* ein Verfahren zur sukzessiven Simplifizierung eines beliebigen Dreiecks-Meshes vor. Obwohl das Verfahren nicht ausschließlich zur Terraindarstellung gedacht ist, ist es dennoch hervorragend dafür geeignet und hat darüber hinaus die Entwicklung vieler spezialisierterer Varianten inspiriert.

Progressive Meshes adressieren eine ganze Reihe praktischer Probleme bei der Meshdarstellung:

Vereinfachung von Meshes: Meshes sind in der Form, wie sie von Künstlern erstellt oder bei Scans produziert werden, nicht unbedingt für Echtzeitdarstellungen geeignet. Aus Geschwindigkeitsgründen müssen sie in ihrer Komplexität reduziert werden; es wäre wünschenswert, dies durch ein automatisches Verfahren bei minimalem Verlust an visueller Qualität zu erreichen.

LOD-Darstellung: Die gleiche Technologie kann zur Erstellung von reduzierten Meshes für LOD-Algorithmen genutzt werden.

Progressive Übertragung: Bei der Übertragung über ein langsames Medium, z.B. ein Netzwerk, wäre es wünschenswert, schon während des Übertragungsvorganges sukzessiv genauer werdende Versionen des Meshes anzeigen zu können.

Kompression: Es wird eine möglichst platzsparende Darstellungsform des Meshes angestrebt.

Selektive Verbesserung: Unter Umständen wird – abhängig von dynamischen Parametern – in einigen Bereichen des Meshes mehr Detail benötigt als in anderen. Terraindarstellung ist ein klassisches Beispiel für einen solchen Fall; nahe am Betrachter gelegene Teile des Terrains sollten mit einem höheren Detailgrad dargestellt werden als weiter entfernte.

Ein Mesh sei definiert durch:

- eine Menge von Punkten (*Vertices*) $V = \{v_1, \ldots, v_n\}$
- eine Menge von Dreiecken *(Faces*), die über den Punkten definiert ist; eine Verbindung zwischen je zwei Punkten wird als *Kante (Edge)* bezeichnet

- eine Menge von *diskreten Attributen*. Diskrete Attribute sind pro Dreieck definiert; dazu gehören beispielsweise Materialinformationen wie Texturen, Shader und Shaderparameter.
- eine Menge von *skalaren Attributen.* Skalare Attribute sind auf einem Tupel aus Punkt und zugehörigem Dreieck definiert, dazu gehören beispielsweise Farben, Texturkoordinaten und Normalenvektoren. Die Endpunkte von Kanten, an denen zwei Materialien aneinander angrenzen, haben beispielsweise unterschiedliche Texturkoordinaten für die beiden adjazenten Dreiecke.

Das Mesh wird durch das Löschen von Kanten (*Edge Collapsing*) reduziert. Durch das Zusammenlegen zweier Punkt v_s, v_t zu einem neuen Punkt v_s' verschwinden die Kante $\{v_s, v_t\}$ sowie die Dreiecke $\{v_s, v_t, v_l\}$ und $\{v_t, v_s, v_r\}$. Die Koordinaten des neuen Punktes v_s' dürfen sich dabei von v_s und v_t unterscheiden. Diese Operation wird mit *ecol* (v_s, v_t) bezeichnet (Abbildung 16).

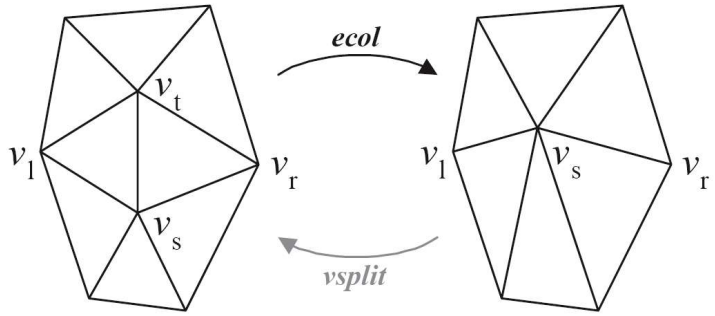


Abbildung 16: Egde Collapse und Vertex Split (Quelle: [HOP96])

Durch eine Folge von *n* Kantenlöschungen wird das Originalmesh M in ein vereinfachtes Mesh M^0 überführt. Die Zahl *n* wird dabei nach Bedarf festgelegt. Jede Löschoperation wird protokolliert, d.h. die ursprünglichen Positionen der beiden betroffenen Punkte sowie ihre Attribute werden gespeichert. Mit Hilfe dieser Attributinformationen *A* lässt sich zu *ecol (vs, vt)* eine inverse Operation *vsplit (s, l, r, t, A)* definieren, die v_s in die ursprünglichen Punkte v_s und v_t aufsplittet und die Dreiecke $\{v_s, v_t, v_l\}$ und $\{v_t, v_s, v_r\}$ wiederherstellt. Ausgehend von M^0 erzeugt jede Splitoperation ein neues Mesh M^1, M^2, M^3 bis schließlich $M^n = M$ erreicht ist. Es ist folglich ausreichend, M^0 und die Attributinformationen aller durchgeführten Kantenlöschungen abzuspeichern, um später das ursprüngliche Mesh M und alle Zwischenschritte M^1 bis M^{n-1} rekonstruieren zu können. Diese Zwischenschritte sind beispielsweise als LOD-Stufen geeignet, zudem kann bei einer progressiven Datenübertragung mit der Anzeige des Meshes begonnen werden, sobald alle Daten für M^0 empfangen worden sind. Hoppe konnte in seiner Arbeit darüber hinaus zeigen, dass die Progressive-Mesh-Darstellung eine effiziente Kompression des Meshes sein kann.

Die Qualität der reduzierten Meshes hängt in erster Linie davon ab, welche Kanten in welcher Reihenfolge gelöscht werden; hierin besteht die eigentliche Herausforderung. Der einfachste Ansatz liegt darin, die zu löschenden Kanten und die Reihenfolge zufällig zu wählen. Obwohl dieses Vorgehen bei sehr regelmäßigen Meshes zufriedenstellende Ergebnisse liefern mag, führt es in den meisten Fällen jedoch zu inakzeptablen Deformationen. Diese Deformationen treten auf, weil für die Form des Meshes charakteristische Kanten gelöscht werden. Ein adäquates Mittel, dies zu vermeiden, ist es, bei der Löschung von Kanten den Winkel zwischen den angrenzenden Flächen zu

berücksichtigen (*Smallest Angle Approach)*. Zu diesem Zweck wird für jede Kante des Meshes der Winkel zwischen den Normalenvektoren der adjazenten Dreiecke berechnet. Kanten mit kleinen Winkeln werden bevorzugt gelöscht. Dieser Ansatz liefert bereits sehr gute, aber noch keine optimalen Ergebnisse.

Das von Hoppe vorgeschlagene Verfahren geht über das eben beschriebene hinaus und berücksichtigt neben dem Winkel der Flächen auch skalare und diskrete Attribute. Innerhalb eines Meshes verlaufen so genannte *Discontinuity Curves (Diskontinuitätskurven)*. Diese Kurven bestehen aus Kanten, die für das Aussehen des Meshes charakteristisch sind. Kanten des Meshes können gelöscht werden, ohne das Aussehen des Meshes negativ zu beeinflussen, wenn dabei die Topologie der Discontinuity Curves nicht verändert wird. Selbst Kanten, die selbst direkt zu einer Discontinuity Curve gehören, können unter dieser Bedingung gelöscht werden.

Eine Kante wird als *scharfe Kante* bezeichnet, wenn der Winkel zwischen den adjazenten Flächen einen festgelegten Grenzwert überschreitet, die diskreten Attribute der adjazenten Flächen sich unterscheiden oder die skalaren Attribute der Endpunkte der Kante sich stärker unterscheiden als ein Schwellenwert erlaubt. Gegeben sei eine Funktion *sharp* (v_s, v_t), die testet, ob eine Kante $\{v_s, v_t\}$ scharf ist. Es sei $\#sharp(v_s)$ die Anzahl der zum Punkt v_s adjazenten scharfen Kanten. Das Löschen einer Kante verändert die Topologie einer Discontinuity Curve (bzgl. Abbildung 16), wenn:

- $sharp(v_s, v_l)$ und $sharp(v_t, v_l)$ oder
- $sharp(v_s, v_r)$ und $sharp(v_t, v_r)$ oder
- $\#sharp(v_s) \geq 1$ und $\#sharp(v_t) \geq 1$ und nicht $sharp(v_s, v_t)$ oder
- $\#sharp(v_s) \geq 3$ und $\#sharp(v_t) \geq 3$ und $sharp(v_s, v_t)$ oder
- $sharp(v_s, v_t)$ und $\#sharp(v_s) = 1$ und $\#sharp(v_t) \neq 2$ oder
- $sharp(v_s, v_t)$ und $\#sharp(v_t) = 1$ und $\#sharp(v_s) \neq 2$

Das Verfahren läuft ab wie der Smallest Angle Approach – alle Kanten des Meshes werden so priorisiert, dass Kanten, deren adjazente Flächen einen kleinen Winkel aufweisen und deren Attribute ähnlich sind, zuerst gelöscht werden. Kanten, deren Löschung die Topologie eine Discontinuity Curve verändern würde, werden entweder nie gelöscht oder – wenn es sich aufgrund ihrer Anzahl nicht vermeiden lässt – mit entsprechend höheren Kosten versehen.

9.3.2.Continuous LOD nach Lindstrom et al

Eine Gruppe um den in Schweden geborenen Wissenschaftler Peter Lindstrom stellt in [LIN+96] ein stufenloses LOD-Verfahren speziell für Terrains in Height-Field-Darstellung vor. Das Verfahren reduziert blickwinkelabhängig die Anzahl der Polygone, mit denen ein bestimmtes Terrainstück dargestellt wird, wobei der Verlust an Bildqualität minimiert wird. Der Qualitätsverlust wird mit einer austauschbaren Metrik gemessen, der Schwellenwert für akzeptable Verluste kann vom Anwender bestimmt werden. Im Gegensatz zu progressive Meshes müssen die Ausgangsdaten nicht in ein spezielles Format überführt werden; der Algorithmus arbeitet direkt auf der Height-Field-Darstellung.

Terrainblöcke

Gegeben sei ein Height Field mit einem festen Sampling-Intervall von x_{res}, y_{res} in x- respektive y-Richtung. Das kleinste Terrain, das sich darstellen lässt, besteht aus 3x3 Stützpunkten und ist tesseliert wie in Abbildung 17a zu sehen. Höhere Auflösungsstufen sind immer eine 2x2-Anordnung der vorherigen Stufe, d.h. das nächstgrößere mögliche Terrain besteht aus 5x5 Stützpunkten respektive vier 3x3-Blöcken (Abbildung 17b) und das wiederum nächstgrößere Terrain aus 9x9 Punkten (17c). Allgemein gesprochen hat ein Terrainstück der Stufe *l* eine Kantenlänge von 2^l+1 Stützpunkten.

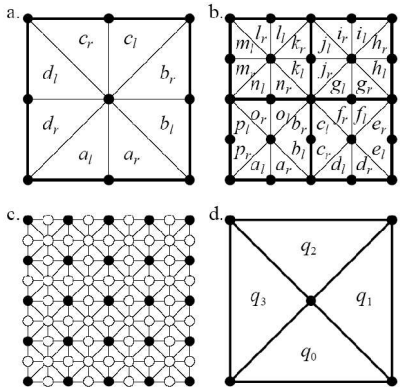


Abbildung 17: Terrainblöcke in verschiedenen Auflösungen (Quelle: [LIN+96])

Für einen bestimmten Wert *l=n* wird die Struktur als *Terrainblock* bezeichnet. Das darzustellende Terrain wird in solche Terrainblöcke zerlegt, wobei aneinander angrenzende Terrainblöcke alle Punkte entlang ihrer Berührungskante gemeinsam haben. Terrainblöcke werden als diskrete LOD-Stufen verwendet. Ihre Auflösungsstufe *n* kann frei gewählt werden, allerdings hat die Wahl Einfluss auf die Ausführungsgeschwindigkeit, so dass für eine konkrete Anwendung experimentell ein Optimum bestimmt werden sollte. Die Fläche des Originalterrains, die von einem Terrainblock dargestellt wird, kann wiederum in Stufen variieren; sie beträgt $2^{m+n}x_{res}$ x $2^{m+n}y_{res}$, wobei *m* eine natürliche Zahl und $m\geq 0$ ist. Für $m=0$ repräsentiert der Block das Terrain in der maximal zur Verfügung stehenden Auflösung. Ein Block der Stufe $m=1$ besteht aus vier Blöcken der Stufe $m=0$, wobei jeder zweite Stützpunkt in vertikaler und horizontaler Richtung weggelassen wurde; dieses Muster setzt sich rekursiv fort, bis das gesamte Terrain von einem einzigen Terrainblock dargestellt wird. Die Hierarchie der Terrainblöcke lässt sich am besten in Form eines *Quadtrees* darstellen; das ist ein Baum, in dem jeder Knoten entweder genau vier oder keine Kinder besitzt.

Vereinfachung auf Vertex-Ebene

Jeder Terrainblock besteht aus der gleichen Anzahl von Dreiecken; diese Anzahl lässt sich jedoch abhängig von den Z-Koordinaten der Dreiecke reduzieren, ohne dass ein inakzeptabler Qualitätsverlust auftritt.

Wenn ein gleichschenklig-rechtwinkliges Dreieck entlang der Mittelsenkrechten der Hypotenuse geteilt wird, entstehen wiederum zwei gleichschenklig-rechtwinklige Dreiecke, die sich rekursiv auf gleiche Weise teilen lassen. Ausgehend von der Konfiguration in Abbildung 17d entsteht so die Triangulierung eines Terrainblocks. Zwei durch eine solche Teilung entstandenen Dreiecke werden *Ko-Dreiecke* genannt. Ein Terrainblock lässt sich vereinfachen, indem dieser Teilungsprozess rückwärts vollzogen wird. D.h. ausgehend von der maximalen Triangulierung werden ausgewählte Paare von Ko-Dreiecken vereinigt und insofern zum neu enstandenen Dreieck ein ebenfalls vereinigtes Ko-Dreieck existiert, wird dieses Paar rekursiv wiederum für eine Vereinigung in Betracht gezogen.

Zwei Dreiecke sollten nur dann vereinigt werden, wenn die Auswirkung auf die Darstellungsgenauigkeit des Terrains möglichst gering ist. Durch die Vereinigung zweier Dreiecke entfällt jeweils ein Punkt, beispielsweise eliminiert die Vereinigung der Dreiecke ABE und BCE zu ACE in Abbildung 18 den Punkt B. Zwischen dem ursprünglichen Punkt B und seiner Projektion auf das Dreieck ACE besteht ein Höhenunterschied δ_B; dieser Unterschied führt zu einem Darstellungsfehler. Diese Strecke wird auch als *Delta-Segment* bezeichnet.

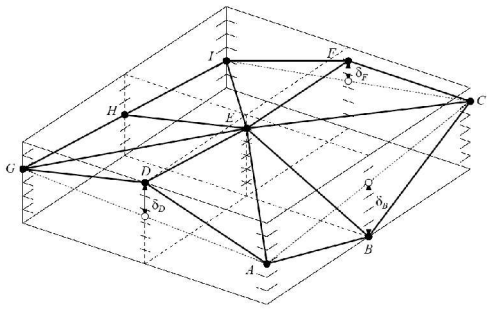


Abbildung 18: Höhenänderung durch Vereinigung von Dreiecken (Quelle: [LIN+96])

Es gilt: $\delta_B = B_z - \frac{A_z + C_z}{2}$.

Die Größe des wahrgenommenen Fehlers kann berechnet werden, indem die Strecke δ_B auf die Projection Plane, also auf den Bildschirm, projiziert wird. Je weiter die betreffenden Dreiecke in der 3D-Szene vom Betrachter entfernt sind, desto geringfügiger wirkt sich der Fehler auf das endgültige Bild aus. Vom Benutzer wird ein Schwellenwert τ für den maximal akzeptablen wahrgenommenen Bildfehler festgelegt.

Lindstrom et al leiten in ihrem Paper die Berechnung des wahrgenommenen Fehlers her, er beträgt

$$\delta_{screen} = \frac{d\lambda\delta\sqrt{(e_x - V_x)^2 + (e_y - V_y)^2}}{(e_x - V_x)^2 + (e_y - V_y)^2 + (e_z - V_z)^2}$$, wobei

- e der Eye Point, also der Standpunkt des Betrachters ist,
- d die Entfernung zwischen e und der Projection Plane ist,
- λ die Anzahl Pixel pro Weltkoordinateneinheit im Bildschirmkoordinatensystem ist, wobei Pixel als quadratisch angenommen werden, und
- V der Mittelpunkt des Delta-Segments ist für die beiden betroffenen Dreiecke ist.

Optimierungen

Die Triangulierung eines Terrainblocks muss nach jeder Bewegung der Kamera ausgehend von seiner maximalen Triangulierung komplett neu berechnet werden; dabei kostet insbesondere die Berechnung von δ_{screen} für jedes Dreieck sehr viel Zeit. Mit Hilfe des im letzten Abschnitt besprochenen Quadtrees lässt sich das Verfahren wesentlich beschleunigen. Vier benachbarte Terrainblöcke können direkt durch ihren gemeinsamen Elternblock im Quadtree ersetzt werden, wenn bekannt ist, dass die projizierten Delta-Segmente aller dadurch eliminierten Punkte kleiner als der Schwellenwert τ sind. Für jeden Terrainblock im Quadtree wird dazu ein δ_{max} berechnet, welches dem größten in diesem Block auftretenden Delta-Segment entspricht. Für jeden Block muss weiterhin die *Bounding Box* berechnet werden, also der kleinste Quader, der das Volumen des Terrainblockes vollständig umschließt. Mit diesen Informationen kann für eine gegebene Betrachterposition entschieden werden, ob die Blöcke direkt ersetzt werden können. Zu diesem Zweck wird angenommen, dass δ_{max} an der dem Betrachter nächstgelegenen Ecke der Bounding Box auftritt; dies ist eine konservative Approximation. Die Ersetzungen werden so lange wie möglich durchgeführt; danach werden die Terrainblöcke wie eingangs beschrieben getrennt voneinander weiter vereinfacht.

Auf Grundlage dieser Idee lässt sich auch die Vereinfachung innerhalb eines Terrainblockes weiter optimieren. Lindstrom stellte fest, dass es möglich ist, für jeden konkreten Terrainblock abhängig von der Betrachterposition den kleinsten Wert für δ zu ermitteln, der projiziert τ überschreiten kann. Dieser Wert sei mit δ_l bezeichnet; Dreiecke, deren Delta-Segment kleiner als δ_l ist, können ohne Berechnung von δ_{screen} direkt vereinigt werden. Analog lässt sich der größte Wert δ_h berechnen, der projiziert τ überschreiten kann. Dreiecke, deren Delta-Segment größer δ_h ist, können nicht vereinigt werden. Aus Geschwindigkeitsgründen können δ_l und δ_h nicht exakt berechnet, sondern nur konservativ angenähert werden. Daher gibt es zwischen den beiden Werten ein Unsicherheitsintervall, in dem alle Dreiecke einzeln getestet werden müssen.

Beziehungen zwischen Dreiecken

Damit ein Dreieck mit seinem Ko-Dreieck vereinigt werden kann, muss sichergestellt sein, dass dieses Ko-Dreieck seinerseits komplett vereinigt ist. Diese Beziehung zwischen den Dreiecken kann als binärer Baum von Dreiecken aufgefasst und entsprechend implementiert werden. Eine andere Herangehensweise besteht darin, die Beziehung als einen Baum von Punkten darzustellen. Damit ein Punkt eliminiert werden kann, müssen zunächst vier andere Punkt eliminiert werden; für Punkt E in Abbildung 18 sind das D, B, F und H. Diese Punkte sind Kinder von E im Baum. Die Eliminierung von Punkt E ist jeweils eine notwendige, aber nicht hinreichende Bedingung für die Eliminierung zweier weiterer Punkte (G, C bzw. I, A). Diese Punkte sind Elternknoten im Baum. Punkte am Rand des Terrains haben entsprechend weniger Eltern- bzw. Kindknoten. Ein Punkt kann nur eliminiert werden, wenn zuvor alle seine Kinder eliminiert worden sind. Es ist außerdem zu bedenken, dass die Abhängigkeiten zwischen Punkten bzw. Dreiecken über Blockgrenzen hinweg bestehen.

9.3.3.ROAM

ROAM steht für *Real-time Optimally Adapting Meshes*, (frei übersetzt: optimal angepasste Drahtgitter in Echtzeit) und wurde am Los Alamos National Laboratory bzw. am Lawrence Livermore National Laboratory entwickelt [DUCH+97]. Die ursprünglichen Autoren haben das Verfahren bei Flugzeugtests am Boden verwendet, um künstlichen Sensorinput zu erzeugen.

Wie der Name schon vermuten lässt, approximiert ROAM ein gegebenes Terrain, definiert durch ein Height Field, mit möglichst wenig Polygonen. Die Triangulierung ist dabei einerseits abhängig vom Terrain, das heißt schroffes Gelände wird mit mehr Polygonen dargestellt als flaches; andererseits hängt die Triangulierung von Position und Blickwinkel des Betrachters ab, das heißt, sie wird mit steigender Entfernung vom Betrachter grober. Ziel ist dabei eine möglichst hohe Darstellungsqualität bei maximaler Ausführungsgeschwindigkeit.

Die besondere Stärke von ROAM ist seine Flexibilität im Hinblick auf die Echtzeitdarstellung. Die Darstellungsqualität ist – wie bei LOD-Algorithmen üblich – über eine Fehlermetrik regelbar. ROAM unterstützt eine große Bandbreite solcher Metriken, darüber hinaus ist es aber auch möglich, für die Szene ein Polygon- oder sogar ein Zeitlimit festzulegen. In diesen Fällen modifiziert der Algorithmus das Mesh so lange, bis die gewünschte Polygonzahl erreicht oder das Zeitlimit überschritten ist. Bei Zeitlimits kann die Optimierung des Meshes über mehrere Frames hinweg fortgesetzt werden, wodurch die graphische Qualität von Bild zu Bild steigt. ROAM garantiert ein unter den jeweils gegebenen Beschränkungen optimales Ergebnis im Sinne der verwendeten Metrik.

Das Verfahren erfordert nur eine minimale Vorverarbeitung der Rohdaten und ist daher selbst für dynamisch veränderbare Terrains geeignet.

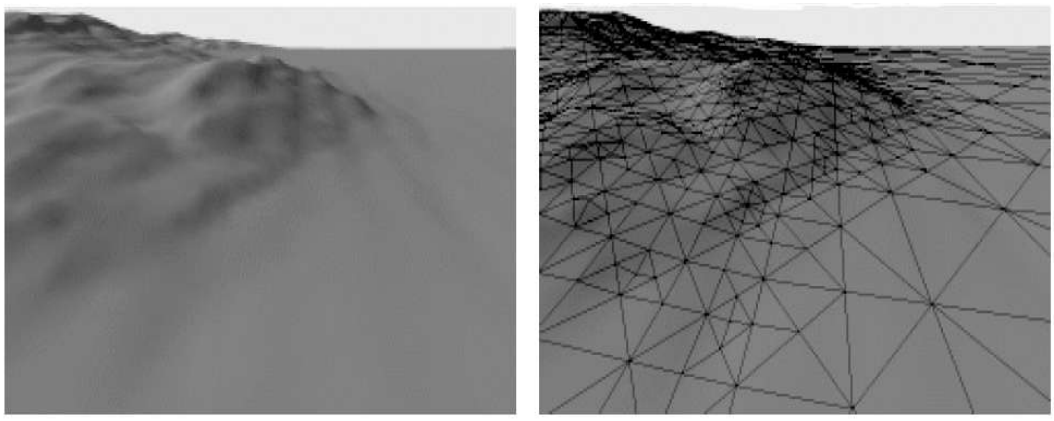


Abbildung 19: Terraindarstellung mit ROAM; links herkömmliches Rendering, rechts mit Wireframe (Quelle: [DUCH+97])

Binary Triangle Trees

Grundlage für ROAM sind so genannte *binary triangle trees* oder *triangle bintrees*. Ausgangspunkt der Überlegung sind gleichschenklige, rechtwinklige Dreiecke. Ein solches Dreieck lässt sich entlang der Mittelsenkrechte der Hypotenuse in zwei kongruente Dreiecke mit wiederum den gleichen Eigenschaften splitten. Dieses Splitting lässt sich über beliebig viele Stufen fortsetzen (siehe Abb. 20). Dabei entsteht ein binärer Baum; Dreieck T ist das Wurzelelement des Baumes, T_0 und T_1 sind seine Kindelemente und gleichzeitig Wurzelelemente ihrer eigenen Bäume. Jedes Dreieck gehört eindeutig zu einer bestimmten Stufe im Baum, die Wurzel ist Stufe L=0, ihre Kinder L=1 usw.

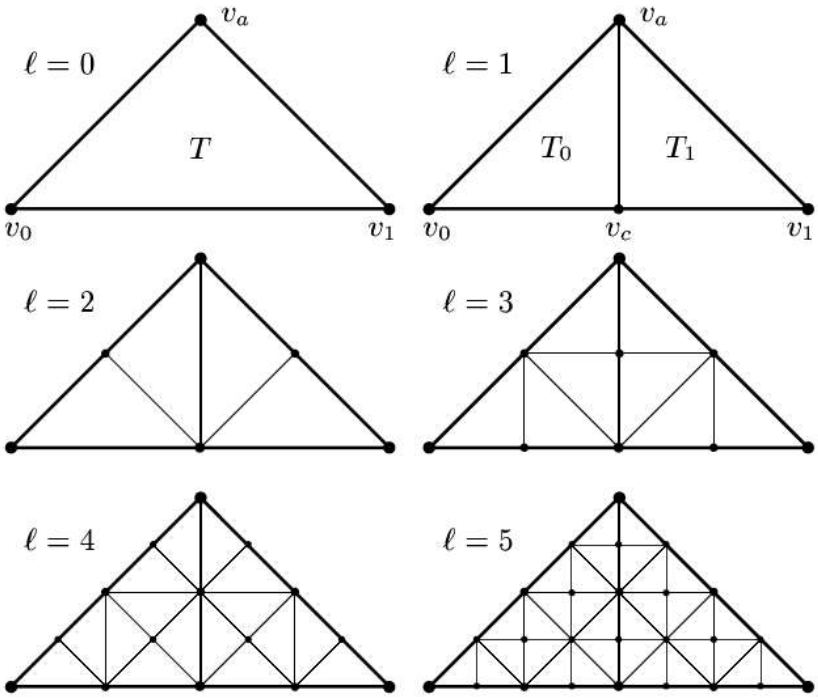


Abbildung 20: Splitting eines Dreiecks über mehrere Stufen (Quelle: [DUCH+97])

Triangulierung des Terrains

Die Grundidee von ROAM ist, das Terrain aus solchen gleichschenkligen, rechtwinkligen Dreiecken zusammenzusetzen. Wo mehr Detail benötigt wird, können Dreiecke gesplittet werden, bis genügend Stützpunkte vorhanden sind, um das Gelände ausreichend genau zu approximieren. Üblicherweise wird das Terrain durch ein Heightfield definiert, das ist aber keine zwingende Voraussetzung für ROAM.

Startpunkt für die Triangulierung ist ein Quadrat, das aus zwei gleichschenkligen, rechtwinkligen Dreiecken besteht und das gesamte Gelände repräsentiert (das ursprüngliche Paper bezeichnet dies auch als *Diamant* bzw. *diamond*).

Die Hauptschwierigkeit besteht nun darin, dass das Terrain durch ein geschlossenes Mesh repräsentiert werden muss, andernfalls entstehen deutlich sichtbare Löcher. Für ein geschlossenes Mesh gilt: Zwei Dreiecke haben entweder keine gemeinsamen Punkte oder genau einen gemeinsamen Eckpunkt oder genau eine gemeinsame Seite.

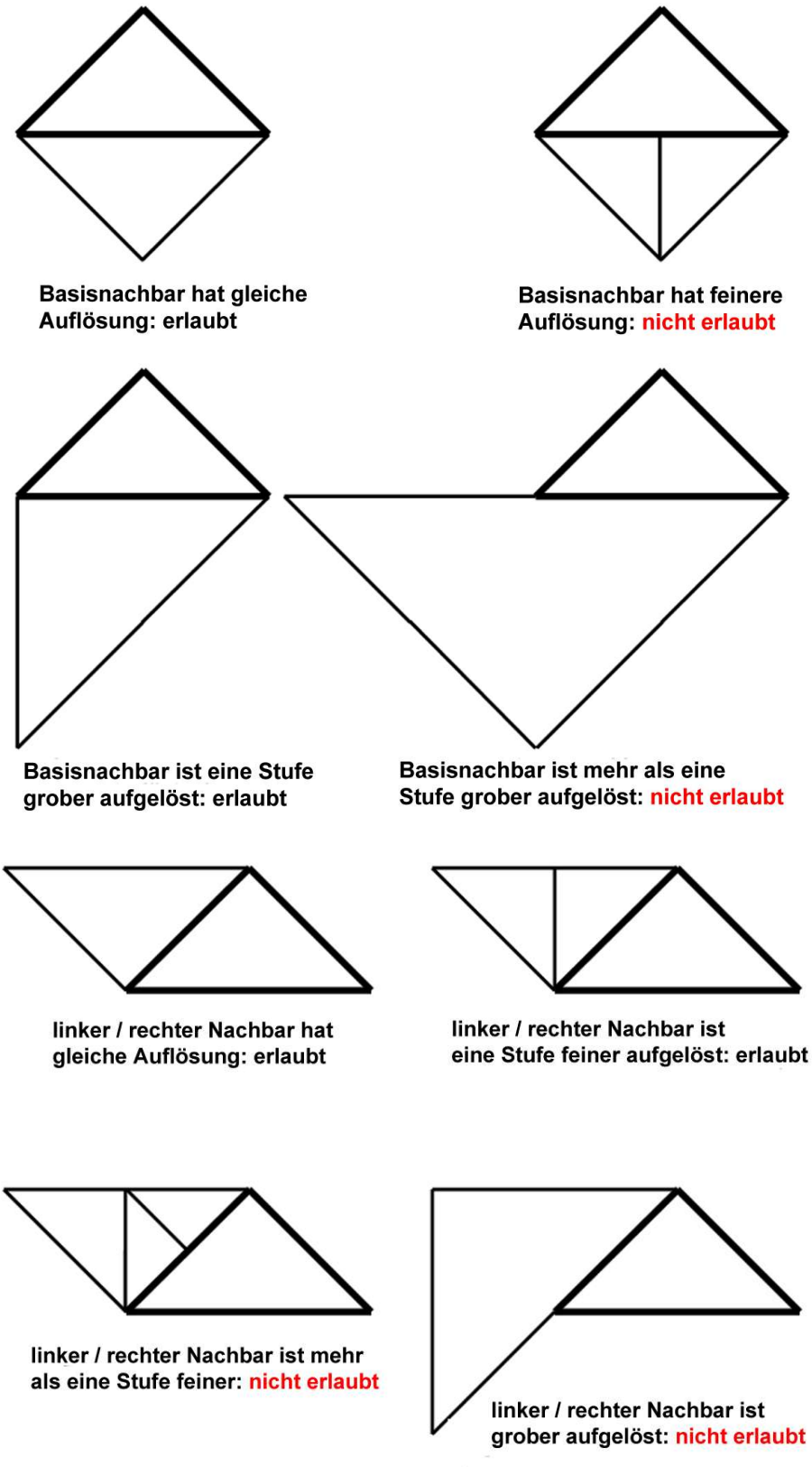


Abbildung 21: erlaubte und unerlaubte Dreieckskonstellationen

Um diese Eigenschaft sicherzustellen, müssen beim Splitten eines Dreiecks auch dessen Nachbarn im Mesh betrachtet werden. Das an die Hypotenuse angrenzende Dreieck wird als Basisnachbar (*base neighbor*) bezeichnet und die an die Katheten angrenzenden Dreiecke als linker bzw. rechter Nachbar (*left / right neighbor*). Abbildung 21 zeigt alle möglichen erlaubten und unerlaubten Konstellationen. Daraus kann eine einfache Regel abgeleitet werden: In einer gültigen Triangulation gilt für jedes Dreieck der Stufe L: Der Basisnachbar hat Stufe L oder Stufe L-1, der linke und rechte Nachbar haben Stufe L oder L+1.

Der Algorithmus beginnt mit einem Diamanten, der das gesamte Terrain repräsentiert und es werden so lange Dreiecke gesplittet, bis die notwendige Auflösung erreicht ist. Die Split-Operation muss so gestaltet werden, dass die Geschlossenheit des Meshes erhalten bleibt. Dabei hilft folgende Überlegung: Wenn genau ein Dreieck eines Diamanten gesplittet wird, so wird die Geschlossenheit des Meshes verletzt (siehe. Abb. 21). Werden allerdings beide Dreiecke gesplittet, so bleibt die Geschlossenheit erhalten und es entsteht nur ein neuer Punkt. Analog muss bei der inversen Operation, dem Vereinigen (*merge*) eines Dreiecks, auch dessen Basisnachbar vereinigt werden (siehe Abb. 22).

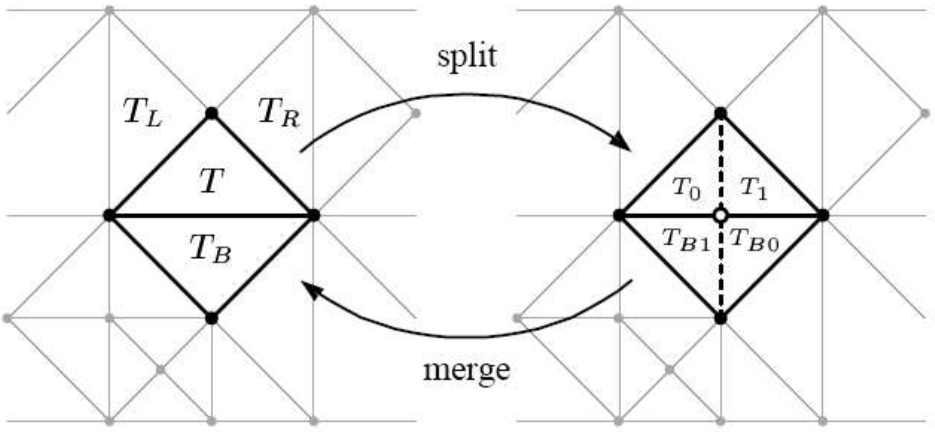


Abbildung 22: Die Dreiecke eines Diamanten müssen gemeinsam gesplittet und vereinigt werden (Quelle: [DUCH+97])

Mit diesem Wissen kann eine Split-Funktion konstruiert werden, die für beliebige Dreiecke funktioniert. Wenn ein Dreieck der Stufe L gesplittet werden soll, können dabei drei Fälle auftreten:

1. Das Dreieck ist Teil eines Diamanten, d.h. sein Basisnachbar hat ebenfalls die Auflösungsstufe L. In diesen Fall wird der Basisnachbar ebenfalls gesplittet.

2. Das Dreieck hat keinen Basisnachbarn, da seine Hypotenuse den Rand des Terrains bildet. In diesem Fall kann es ohne Folgen gesplittet werden.

3. Der Basisnachbar des Dreiecks hat die grobere Auflösungsstufe L+1. In diesem Fall muss zunächst das Splitten des Basisnachbarn erzwungen werden (*forced split*); dadurch wird das aktuelle Dreieck zum Teil eines Diamanten und es kann wie in Fall 1 verfahren werden. Das erzwungene Splitten des Basisnachbarn geschieht durch einen rekursiven Aufruf der Split-Funktion, d.h. für den Basisnachbarn des aktuellen Dreieckes tritt wiederum einer der drei Fälle auf und eventuell muss auch sein Basisnachbar zum Splitten gezwungen werden. Die Rekursion endet, sobald ein Diamant oder der Rand des Terrains getroffen wird (Fall 1 und 2). Abbildung 23 verdeutlicht den Splitalgorithmus.

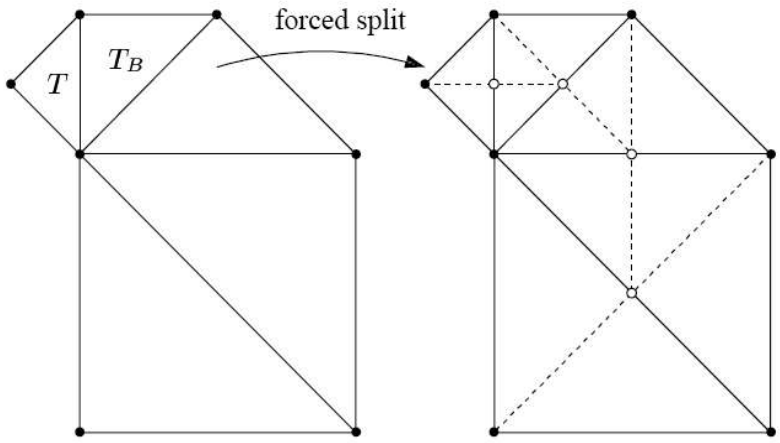


Abbildung 23: Erzwungenes Splitten passiert rekursiv, bis ein Diamant oder der Rand des Terrains erreicht wird (Quelle: [DUCH+97]).

Algorithmus

Gegeben sei ein Prioritätswert p für jedes Dreieck, der ausdrückt, wie dringend das Dreieck gesplittet werden muss. Diese Prioritätswerte basieren meist auf einer blickrichtungsabhängigen Fehlermetrik; dies wird im nächsten Abschnitt näher erläutert werden. Es wird verlangt, dass diese Prioritäten monoton sind, das heißt, kein Dreieck hat eine höhere Priorität als sein Elterndreieck im Binärbaum. Das Problem kann nun mit einer Bestensuche (*greedy search*) gelöst werden. Pseudocode:

- Es sei *T* die Menge der Dreiecke; initial enthält T einen Diamanten (2 Dreiecke).
- Alle Dreiecke *t* aus *T* werden in eine Prioritätswarteschlange *Q* eingefügt.
- Wiederhole, so lange *T* zu klein oder zu ungenau ist:
  - Dreieck *t* mit höchster Priorität ermitteln
  - *t* splitten (das ist evtl. eine rekursive Operation)
  - *t* und alle anderen gesplitteten Dreiecke aus *Q* entfernen
  - alle neu entstandenen Dreiecke in *Q* einfügen

Die Abbruchbedingung der Schleife kann beliebig gestaltet werden; beispielsweise kann die Verfeinerung abgebrochen werden, sobald alle Dreiecke einem Schwellenwert für die Fehlermetrik genügen oder eine bestimmte Zahl von Dreiecken erreicht ist oder eine bestimmte Zeit verstrichen ist.

In einer praktischen Implementation ist es ungünstig, tatsächlich das gesamte Terrain mit zwei binären Bäumen repräsentieren zu wollen; abhängig von der Terraingröße werden diese Baume sehr tief. Effizienter ist es, das Terrain in quadratische Bereiche (*Chunks*) aufzuteilen und für jeden Chunk eigene Bäume zu halten. Die Chunks können gleichzeitig für Sichtbarkeitsbestimmung (*View Frustum Culling*) oder dynamisches Nachladen von Terrain genutzt werden. Erzwungene Splits von Dreiecken können jedoch über Chunkgrenzen hinweg passieren, so dass jeder Chunk seine Nachbarn kennen muss.

Fehlermetriken

Die Autoren von ROAM verwenden eine Fehlermetrik, die sie *Geometric Screen Distortion* nennen. Für jedes Dreieck kann bestimmt werden, wie stark es von der exakten Geländeform abweicht. Diese Abweichungen bestehen aufgrund des Verfahrens nur in der Terrainhöhe. Es sei ohne Beschränkung der Allgemeinheit z die Achse, durch die die z-Koordinate ausgedrückt wird. Ein Keil (*Wedgie*) ist ein Volumen (x,y,z) mit dreieckiger Grundfläche, wobei x und y Koordinaten innerhalb dieses Dreiecks sind und e_T mit

$$\left|z - z_T(x, y)\right| \leq e_T$$

die Dicke des Keils bezeichnet; z_T ist dabei die Höhe des Dreieckes an einer Position (x,y). Mit der Dicke dieser Keile lässt sich die Abweichung eines Dreiecks von der korrekten Terrainhöhe ausdrücken. Auf der feinsten Auflösungsstufe beträgt die Dicke der Keile 0. Für alle anderen Dreiecke ergibt sich e_T rekursiv aus e_{T0} und e_{T1}, den Abweichungen ihrer Kinder,

$$e_T = max\{e_{T0}, e_{T1}\} + \left|z(v_c) - z_T(v_c)\right| \quad ,$$

wobei die neue, approximierte z-Koordinate des bei der Vereinigung der Kinder eliminierten Punktes v_c sich aus

$$z_T(v_c) = \frac{z(v_0) + z(v_1)}{2}$$

ergibt. Für jeden approximierten Punkt kann die Abweichung von seiner korrekten Position in Bildschirmkoordinaten ermittelt werden. Die in ROAM verwendete Fehlermetrik basiert darauf, diese Berechnung nicht pro Punkt durchzuführen, sondern vielmehr für jedes Dreieck eine obere Schranke für den Fehler zu ermitteln. Dazu wird der Keil jedes approximierten Dreiecks in den Screen Space projiziert, die Höhe des Keils in Bildschirmkoordinaten kann nun als Prioritätswert für den Algorithmus genutzt werden. Keile, welche die Near Clipping Plane durchdringen, werden gesondert behandelt; ihnen wird ein künstlicher Maximalwert zugewiesen. Das höchste Element der Prioritätswarteschlange stellt gleichzeitig den maximalen Fehler im gesamten Bild dar.

Darüber hinaus unterbreiten die Autoren eine ganze Reihe von Ideen für verbesserte Metriken:

- Dreiecke, deren Unterbäume komplett von der Kamera abgewandt sind, könnten minimale Priorität erhalten.

- Dreiecke außerhalb des Sichtkegels könnten minimale Priorität erhalten.

- die Abweichung der Flächennormalen könnte berücksichtigt werden. Das kann für eine korrekte Beleuchtung von Bedeutung sein.

- Verzerrung von Texturkoordinaten könnte berücksichtigt werden.

- Objekte sollten korrekt auf dem Boden positioniert sein, d.h. die Geländeform unter einem Objekt sollte exakt dargestellt werden; um dies zu erreichen könnte z.B. den Dreiecken direkt unter Objekten eine höhere Priorität gegeben werden.

- Sichtlinien könnten bedeutsam sein, d.h. korrekte Verdeckung oder Sichtbarkeit von Objekten; zu diesem Zweck könnten Dreiecken entlang solcher Sichtlinien eine höhere Priorität erhalten.

- Berücksichtigung atmosphärischer Effekte; Dreiecke, die durch Nebel o.ä. schlecht sichtbar sind, könnten eine geringere Priorität erhalten.

- korrekte Silhouette: Dreiecke, die am Übergang zwischen der Kamera zugewandten und abgewandten Dreiecken liegen, also mit einer Kante die Silhouette der Landschaft bilden, könnten höhere Priorität erhalten.

9.4.GPU-orientierte Techniken

Die Einführung von 3D-Graphikkarten für Standard-PCs Ende der 1990er Jahre brachte viele Veränderungen. Ursprünglich hat die CPU die gesamte Last der Graphikdarstellung getragen und musste direkt Pixel in einen speziellen Bildspeicherbereich schreiben. Die Graphikkarte leistete kaum mehr, als den Inhalt dieses Speicherbereichs in ein Monitorsignal umzuwandeln. 3D-Graphikkarten hingegen realisieren eigenständig komplexe Graphikalgorithmen. Die CPU liefert nur noch die Ausgangsdaten – 3D-Koordinaten, Dreieckslisten, Texturen, usw. Die eigentliche Arbeit wird von hochspezialisierten Graphikprozessoren übernommen. Diese Prozessoren haben teilweise mehr Transistoren als die CPU, sind heutzutage frei programmierbar (wenn auch nicht annährend so flexibel wie die CPU), verwalten hunderte Megabyte eigenen Speichers und sind mehrfach parallel ausgelegt, d.h. sie verarbeiten mehrere 3D-Koordinaten oder mehrere Pixel gleichzeitig. Diese Graphikprozessoren (GPUs) sind der CPU auf ihrem Aufgabenfeld weitaus überlegen und haben einen weiteren unschätzbaren Vorteil – sie arbeiten parallel zur CPU.

Angesichts dieser Umwälzung müssen die bisher betrachteten Algorithmen zur Terraindarstellung neu bewertet werden. Sie gehen implizit von der Annahme aus, dass das eigentliche Rendering von der CPU durchgeführt wird. Da das Zeichnen eines texturierten Dreiecks auf der CPU eine sehr teure Operation ist, konzentrieren sich die Verfahren auf die Reduzierung der Dreieckszahl pro Bild.

Auf modernen Graphikprozessoren stellt sich die Situation anders dar. GPUs sind darauf optimiert, in kürzester Zeit hunderttausende von Dreiecken zu zeichnen. Natürlich braucht auch die GPU für eine große Zahl von Dreiecken länger als für eine kleine, allerdings ist das erstens bei weitem nicht der einzige Faktor – andere Einflussgrößen wirken sich unter Umständen weitaus stärker auf die Geschwindigkeit aus als die Anzahl der zu zeichnenden Dreiecke – und zweitens haben sich die Dimensionen verschoben; einige tausend eingesparte Dreiecke haben kaum Auswirkungen auf die GPU. Bei den CPU-orientierten Algorithmen besteht somit die Gefahr, dass mehr Zeit darauf verwendet wird, Dreiecke zu eliminieren, als die GPU überhaupt zum Zeichnen dieser Dreiecke benötigt hätte.

Heutige GPUs sind leider nicht in der Lage, Geometrie (d.h. Dreiecke) zu erzeugen oder zu verwerfen. Sie können nur Geometrie verformen, d.h. Koordinaten transformieren. Daher ist es nicht möglich, einen Algorithmus wie ROAM komplett auf der GPU zu implementieren. Es ist aber abzusehen, dass GPUs in Zukunft eine viel freiere Programmierung erlauben werden und die heute geltenden Beschränkungen damit hinfällig machen. Momentan ist jedoch eine andere Art von Algorithmen gefragt – eine, die die CPU entlastet und die GPU optimal belastet. Um die Parallelität beider Prozessoren auszunutzen, sollte die CPU zudem möglichst wenig Zeit für das Vorbereiten von Daten für die GPU benötigen. Eine weitere wichtige Überlegung betrifft die Kommunikation zwischen CPU und GPU. Die GPU verwaltet große eigene Speicherbereiche und kann nur Daten aus diesem Graphikspeicher verarbeiten. Die GPU hat keinen Zugriff auf den regulären Hauptspeicher und die CPU hat keinen Zugriff auf den Graphikspeicher – alle Daten, welche die CPU der GPU zur Verfügung stellen soll, müssen vom Hauptspeicher durch den BUS in den Graphikspeicher kopiert werden. Das Kopieren kostet signifikant Zeit, zudem konvertiert die GPU die Daten anschließend eventuell in ein proprietäres Format, um den internen Zugriff zu beschleunigen.

Zusammenfassend kann gesagt werden: Es ist nach wie vor wichtig, die Komplexität der Geometrie zu reduzieren. Im Hinblick auf aktuelle Graphikprozessoren sollte die CPU dabei nur minimal belastet werden und es sollten möglichst wenige Daten pro Bild zur Graphikkarte kopiert werden müssen. Die GPU sollte die Hauptlast tragen.

9.4.1.GeoMipMaps

Geometrical MipMapping, oder kurz GeoMipMapping, wurde von Willem de Boer in [BOE00] vorgestellt. Das Verfahren reduziert vom Betrachter weiter entfernte Geometrie in ihrer Komplexität, um eine schnellere Darstellung zu ermöglichen. Im Gegensatz zu CPU-orientierten Techniken konzentriert sie sich dabei nicht auf einzelne Dreiecke, sondern auf größere Terrainblöcke. Der Begriff GeoMipMapping ist eine Analogie zu MipMap-Stufen bei Texturen.

Datenrepräsentation

Der GeoMipMapping-Algorithmus arbeitet auf einem Heightfield, von dem verlangt wird, dass es quadratisch ist und eine Kantenlänge von $2^n + 1$ Punkten hat, wobei n eine natürliche Zahl und $n \geq 1$ ist.

Das Terrain wird als quadratisches Raster von Punkten konstruiert. Jeweils vier benachbarte Punkte bilden als Eckpunkte ein *Quad*. Ein Quad besteht aus zwei gleichschenkligen, rechtwinkligen Dreiecken. Auf der höchsten Auflösungsstufe korrespondieren die Terrainpunkte genau mit den Punkten des Heightfields, d.h. das gesamte Terrain besteht aus 2^n Quads. Die x/z-Koordinaten der Terrainpunkte sind während des gesamten Verfahrens fix, die y-Koordinaten entsprechen der Höhe des Heightfields an der jeweiligen Position (siehe Abb. 24).

Das Terrain wird darüber hinaus in so genannte *Terrainblöcke* unterteilt. Terrainblöcke sind wiederum quadratische Ausschnitte des Terrains mit einer Kantenlänge von 2^m+1 Punkten, wobei m eine natürliche Zahl mit $1 \leq m \leq n$ ist. Die Terrainblöcke überlappen einander nicht, lediglich die Randpunkte werden von zwei benachbarten Blöcken geteilt (bzw. die Eckpunkte von bis zu vier Blöcken). Das Gesamtterrain setzt sich somit aus $2^{(n - m +1)}$ Blöcken zusammen. Der konkrete Wert für m beeinflusst die Darstellungsgeschwindigkeit des Terrains nachhaltig; sein Optimum ist aber stark abhängig von der Applikation und der verwendeten Hardware und sollte daher experimentell bestimmt werden.

Die Terrainblöcke werden für das *view frustum culling* verwendet; Blöcke außerhalb des Sichtbereiches werden beim Rendering nicht berücksichtigt.

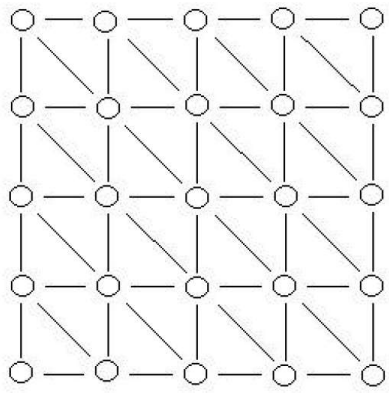


Abbildung 24: Tessellierung des Terrains für den GeoMipMap-Algorithmus (Quelle: [BOE00])

GeoMipMaps

Mipmapping ist ein Begriff, der im Zusammenhang mit Texturen geprägt wurde. Eine Textur hat die beste optische Wirkung, wenn jeder Pixel in der Textur (*Texel*) mehr oder weniger einem Pixel auf dem Bildschirm entspricht. Dabei wird von von *Sampling* gesprochen – für jeden Pixel auf dem Bildschirm wird ein zugehöriger Texel in der Textur gesucht. Bewegt sich der Betrachter auf das texturierte Objekt zu, muss die Textur vergrößert werden; d.h. jeder Texel korrespondiert mit mehreren Pixeln auf dem Bildschirm. Entfernt sich der Betrachter vom texturierten Objekt, so haben nur noch wenige Texel eine Entsprechung auf dem Bildschirm. Dabei entsteht ein Problem: bereits kleinste Veränderungen der Blickrichtung oder Entfernung zum Objekt führen dazu, dass immer wieder andere Texel für die Darstellung auf dem Bildschirm ausgewählt werden. Wenn die Textur starke Kontraste oder Kanten enthält, macht sich das durch ein unangenehmes Flackern des Objektes bemerkbar. Es gibt die Möglichkeit, die Textur in Echtzeit zu filtern (*bilineare Filterung*), d.h. beim Sampling wird der Mittelwert aus dem jeweiligen Texel und seinen vier unmittelbaren Nachbartexeln gebildet. Dies reicht jedoch nicht aus, um das Flackern für weit entfernte Objekte zu eliminieren.

Mipmaps sind eine sehr effektive und effiziente Lösung für das Problem. Für die Textur wird eine Reihe von *Mipmapstufen* erstellt. Mipmapstufe 0 ist die Textur selbst, für Stufe 1 wird die Textur auf die Hälfte ihrer Kantenlänge heruntergerechnet, für Stufe 2 auf ein Viertel und so weiter. Da das Herunterrechnen nur ein einziges Mal geschehen muss, können dabei etwas langsamere Verfahren eingesetzt werden, die im Gegenzug eine höhere Bildqualität liefern. Wenn sich der Betrachter nun von der Textur entfernt, wird ab einer bestimmten Entfernung auf eine höhere (d.h. niedriger aufgelöste) Mipmapstufe umgeschaltet. Das eliminiert das Flackern, führt aber zu einem neuen Problem: Das Umschalten der Textur ist unter Umständen für den Betrachter deutlich zu sehen. Um diesem Problem zu begegnen, wurde eine weitere Filtertechnik eingeführt, die *trilineare Filterung*. Sie berücksichtigt beim Sampling nicht nur die benachbarten Texel in der Textur, sondern interpoliert darüber hinaus zwischen den beiden nächstgelegenen Mipmapstufen. So wird statt eines plötzlichen Umschaltens ein weiches Überblenden zwischen zwei Mipmapstufen realisiert.

GeoMipMapping überträgt diese Ideen auf Terrainvisualisierungen. Die im vorherigen Abschnitt beschriebenen Terrainblöcke dienen als GeoMipMaps der Stufe 0, also als die feinste mögliche Auflösung. Höhere (d.h. grobere) Geomipmapstufen werden jeweils durch Halbierung der Kantenlänge der vorherigen Stufe erzeugt. Abhängig von der Entfernung eines Terrainblocks zum Betrachter wird eine Mipmapstufe für ihn gewählt, weiter entfernte Blöcke werden so mit deutlich weniger Dreiecken dargestellt.

Dieses Verfahren kann auf aktuellen 3D-Graphikkarten sehr effizient implementiert werden, da jede Mipmapstufe einem Vertexbuffer entspricht, der nur ein einziges Mal erzeugt und auf die Graphikkarte übertragen werden muss; danach wird er nicht mehr verändert. Die meisten Implementationen verwenden sogar nur einen einzigen Vertexbuffer pro Terrainblock und realisieren die Mipmapstufen als Indexbuffer. Die Indexbuffer bleiben nach ihrer Erzeugung ebenfalls unverändert.

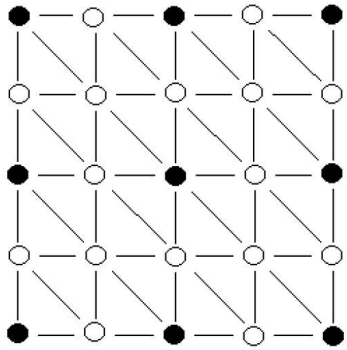


Abbildung 25: Geomipmapstufen. Weiße Punkte sind in Stufe 0, schwarze Punkte in Stufe 0 und Stufe 1 (Quelle: [BOE00])

Der bisher beschriebene Algorithmus führt zu drei Problemen: Erstens sorgt das Umschalten auf eine grobere Mipmapstufe für eine optisch sichtbare Deformation des Geländes, d.h. es kommt zu einem Verlust an Genauigkeit. Es muss ein Verfahren gefunden werden, das nur dann umschaltet, wenn der Verlust innerhalb akzeptabler Grenzen liegt. Zweitens können Löcher in der Geländeoberfläche auftreten, wenn benachbarte Terrainblöcke unterschiedliche Mipmapstufen verwenden. Drittens ist das Umschalten eines ganzen Terrainblocks in eine andere Mipmapstufe für den Betrachter meist deutlich sichtbar. Algorithmen wie ROAM verändern ständig einzelne Dreiecke, das ist bei weitem nicht so auffällig wie das Umschalten eines ganze Blockes.

Die folgenden Abschnitte befassen sich genauer mit den genannten drei Problemen.

Wahl der Geomipmapstufe

Im einfachsten Fall wird die darzustellende Mipmapstufe in Abhängigkeit von der Entfernung eines Blockes zur Kamera festgelegt. Je nach Gelände kann dieses Vorgehen aber zu deutlich sichtbaren Anomalien führen. Grobere Mipmapstufen sind mit einem wachsenden Genauigkeitsfehler behaftet. Ist dieser Fehler sehr groß, fällt das Umschalten der Mipmaps dem Betrachter entsprechend stark auf. Wichtige Geländedetails verschwinden plötzlich aus dem Bild, zudem können sich Sichtlinien verändern oder Objekte stehen nicht mehr mit ihrer gesamten Grundfläche auf dem Boden. Es ist daher ratsam, nur dann grobere Mipmapstufen zu verwenden, wenn der Genauigkeitsfehler akzeptabel ist.

Der Genauigkeitsverlust entsteht, weil grobere Mipmapstufen weniger Höhenpunkte verwenden. Es sei δ der Höhenunterschied, der durch das Weglassen eines Punktes entsteht (siehe Abb. 26).

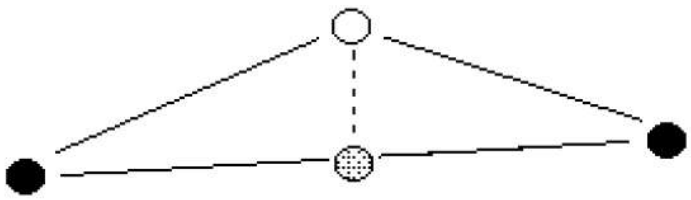


Abbildung 26: Der weiße Punkt gehört zu Stufe 0, die schwarzen Punkte zu Stufe 1. Der graue Punkt ist die imaginäre Position des weißen Punktes in Stufe 1, die gestrichelte Linie entspricht daher der Höhenänderung für diesen Punkt zwischen den Geomipmapstufen (Quelle: [BOE00]).

Innerhalb einer Mipmapstufe > 0 treten mehrere dieser Höhenunterschiede auf, weil gegenüber Mipmapstufe 0 mehrere Punkte weggelassen werden. Mit δ sei der maximale Höhenunterschied bezeichnet, der dabei für eine bestimmte Mipmapstufe auftritt, also $\delta = max\{\delta_0,... \delta_{n-1}\}$.

Wie stark diese Höhenänderung für den Betrachter bemerkbar ist, ist abhängig von der Entfernung d des Terrainblocks zum Betrachter und dem Blickwinkel zwischen Betrachter und dem Block. Mit diesen Informationen ist der von δ verursachten Fehler in Bildschirmpixeln (*screen space error*) ε berechenbar. Es wird ein Schwellenwert (*threshhold*) τ für ε festgelegt und es kann nur dann auf eine Mipmapstufe gewechselt werden, wenn ihr $\varepsilon < \tau$ ist. Eine genaue Formel für die Berechnung von ε gibt [LIN+96]. Unglücklicherweise ist die Berechnung jedoch zu aufwändig, um sie einmal pro Bild für jeden Terrainblock durchzuführen. δ lässt sich für jede Mipmapstufe vorberechnen, ε wegen der Abhängigkeit von d und dem Blickwinkel jedoch nicht. Da ε von zwei Parametern abhängig ist, ist auch eine Tabelle mit vorberechneten Werten hier keine Lösung. Es ist allerdings eine Vereinfachung möglich: Der Fehler ε ist dann am größten, wenn die Blickrichtung der Kamera genau parallel zum Boden verläuft. Bei der Berechnung wird daher davon ausgegangen, dass dies immer der Fall ist. Dieser Ansatz ist sehr konservativ, bei steilen Kamerawinkeln werden so feinere Mipmapstufen gewählt als eigentlich notwendig wären. Im Gegenzug wird die Berechnung von ε aber dramatisch einfacher, sie ist nur noch von d abhängig. Genau genommen ist es jetzt sogar möglich, für jede Mipmapstufe eines bestimmten Terrainblockes vorzuberechnen, ab welcher Entfernung ihr $\varepsilon < \tau$ wird. Diese Entfernung wird mit D_n bezeichnet – die Entfernung, ab der die Mipmapstufe n dieses Blockes verwendet werden kann. Die Berechnungsvorschrift für D_n lautet:

$$D_n = |\delta| \cdot C \quad ,$$

wobei C eine Konstante ist, die sich folgendermaßen berechnet:

$$C = \frac{A}{T} \quad \text{, wobei}$$

$T = 2\frac{\tau}{v_{res}}$ und v_{res} die vertikale Bildschirmauflösung ist und

$$A = \frac{n}{|t|} \quad .$$

n und t sind dabei Parameter des View Frustums; der Autor geht von OpenGL aus, wo das View Frustum mittels der Funktion *glfrustum(l, r, b, t, n, f)* definiert werden kann. n ist dabei der Abstand der *Near Clipping Plane* zum Betrachter und t ist die oberste Koordinate auf der *Near Clipping Plane* und definiert somit die *Top Clipping Plane*. Alternativ kann t auch anders berechnet werden:

$t = n \cdot \tan(\frac{fov \cdot \pi}{360})$, wobei *fov* der Öffnungswinkel des Sichtkegels in Grad ist.

Eine letzte Optimierung kann noch angebracht werden: Es ist günstiger, statt D_n jeweils D_n^2 zu berechnen; dann muss nämlich statt des Abstandes d zwischen jedem Terrainblock und Betrachter nur d^2 berechnet werden, wodurch pro Terrainblock und Bild je eine Wurzeloperation entfällt.

Vermeidung von Löchern

Wenn unterschiedlich hoch aufgelöste Terrainblöcke aneinander angrenzen, können die schon von anderen Verfahren her bekannten Löcher in der Landschaft auftreten. Das liegt daran, dass der jeweils höher aufgelöste Block mehr Randpunkte und daher mehr Detail hat als sein Nachbar.

Es gibt verschiedene Möglichkeiten dies zu vermeiden – beispielsweise könnte der niedriger aufgelöste Block entsprechend mehr Randpunkte erhalten, um perfekt an seinen Nachbarn zu passen. Dies würde jedoch eine dynamische Veränderung der Geometrie oder zumindest viele verschiedene Varianten einer Mipmap bedeuten, was aus den eingangs genannten Gründen vermieden werden sollte. Andererseits könnten die „überzähligen" Randpunkte des höher aufgelösten Blockes auf eine passende Höhe gebracht werden, also auf den Mittelwert ihrer Nachbarn. Das führt leider zu so genannten T-Kreuzungen (*t-junctions*), also Stellen, wo ein Punkt (des höher aufgelösten Blockes) auf eine Kante (des niedriger aufgelösten Blockes) trifft. An solchen Stellen kommt es regelmäßig zu Genauigkeitsproblemen beim Rasterisieren, was sich am Ende in fehlenden Pixeln bemerkbar macht. Daher ist auch von diesem Vorgehen abzuraten.

De Boer schlägt statt dessen vor, die überzähligen Punkte einfach nicht zu benutzen und statt dessen die Indizierung der Dreiecke zu verändern (siehe Abb. 27). Dazu ist keine Änderung an den Punkten nötig, es werden lediglich verschiedene Indexbuffer für die verschiedenen Kombinationen von Nachbarn benötigt, was noch akzeptabel erscheint. Bei dieser Methode muss jeder Terrainblock seine vier unmittelbaren Nachbarn kennen und vor dem Rendern wissen, welche Mipmapstufe sie benutzen werden. Ein Nachteil ist, dass durch das Verändern der Triangulierung an den Rändern auch die Beleuchtung der betroffenen Dreiecke minimal verändert wird. Dieser Effekt ist klein, aber beim Umschalten der Mipmapstufen theoretisch sichtbar.

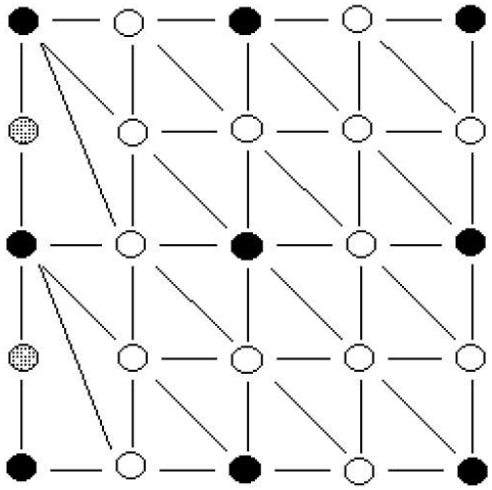


Abbildung 27: Diese Geomipmapstufe grenzt an eine niedriger aufgelöste an. Daher werden die Dreiecke am linken Rand anders angeordnet, die grauen Punkte werden nicht mehr benutzt (Quelle: [BOE00]).

Trilineares Geomipmapping

Ein letztes Problem bleibt das deutlich sichtbare Umschalten der Mipmapstufen. Hier kann die Analogie zu Texturen fortgesetzt werden – genau wie für Texturen trilineare Filterung benutzt wird, um zwischen Mipmaps zu interpolieren und so das Umschalten zu verbergen, kann auch zwischen Geomipmapstufen interpoliert werden. Der Interpolationsfaktor t kann aus der Entfernung d des Blockes zum Betrachter, der Minimalentfernung der aktuellen Mipmapstufe, D_n, und der Minimalentfernung der nächsthöheren Mipmapstuft D_{n+1} berechnet werden:

$$t = \frac{(d - D_n)}{(D_{n+1} - D_n)} \quad .$$

Dieses t kann nun benutzt werden, um die Position eines Punktes zwischen seinen Positionen in den beiden Mipmapstufen zu interpolieren:

$$v' = v - t \cdot \delta_v \quad ,$$

wobei v der Punkt in Mipmapstufe n ist, δ_v die Höhenänderung gegenüber Stufe $n+1$ und v' die interpolierte Position des Punktes.

Dieses Verfahren kann mit *Vertex Shadern* implementiert werden und läuft somit komplett auf der GPU.

9.4.2.Chunked LOD

Wie auch GeoMipmapping arbeitet Chunked Level Of Detail von Thatcher Ulrich [ULR02] auf größeren Terraineinheiten (hier *Chunks* genannt) statt auf einzelnen Punkten oder Dreiecken und ermöglicht dadurch eine effiziente Nutzung der GPU. Besonderes Augenmerk wird auf das Problem der Größe der Terraindaten gelegt. Chunked LOD eignet sich hervorragend für Fälle, in denen der größte Teil der Daten nicht im RAM gehalten werden kann, sondern auf einem langsameren Medium – wie z.B. einer Festplatte – liegt. Dadurch werden praktisch unbegrenzt große Terrains möglich. Ein wesentlicher Nachteil ist jedoch, dass ein nicht-trivialer Vorverarbeitungsschritt für die gesamten Daten notwendig ist.

Basisalgorithmus

Ausgangspunkt ist ein hochdetailiertes Datenmodell des Terrains, das beispielsweise als Mesh oder als Heightfield vorliegen kann. Im Vorverarbeitungsschritt wird aus diesem Referenzmesh ein Baum von unabhängigen Meshes (*Chunks*) berechnet. Das Wurzelelement stellt das gesamte Terrain (oder wahlweise einen Teil davon) in einer niedrigen Auflösung dar. Die Kindelemente spalten ihr Elternelement in mehrere Teile, jedes Kindelement stellt eines dieser Teile in einer höheren Auflösung dar. Der Baum kann bis zur gewünschten Tiefe rekursiv fortgesetzt werden; typischerweise bis auf der untersten Ebene die Auflösung des Referenzmeshes erreicht ist.

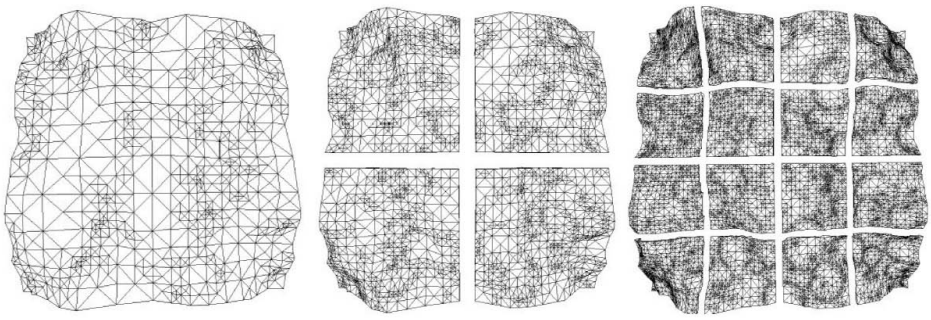


Abbildung 28: Die ersten drei Stufen eines Chunked-LOD-Baumes (Quelle: [ULR02])

Jedem Chunk wird ein maximaler geometischer Fehler, δ, zugewiesen. δ gibt an, wie stark ein Chunk vom zugrunde liegenden Referenzmesh abweicht. Die Chunks werden so berechnet, dass für jeden Level L des Baumes gilt:

$$\delta(L+1)=\delta\frac{(L)}{2} \quad .$$

Der Einfachheit halber haben also alle Chunks einer Stufe im Baum den gleichen maximalen geometrischen Fehler.

Der einfachste Fall wäre, dieses Schema auf einem Height Field anzuwenden; der Baum der Chunks kann dann als *Quadtree* organisiert werden. Der geometrische Fehler δ eines Chunk-Punktes ist die vertikale Differenz zum Referenzmesh an der entsprechenden Stelle. Der geometrische Fehler eines Chunks ist der größte Fehler eines seiner Punkte.

Beim Rendern der Szene wird abhängig von der Entfernung zum Betrachter für ein bestimmtes Stück Terrain ein passender Chunk gewählt. Dazu wird der maximale Bildfehler ρ eines Chunks verwendet:

$$\rho = \frac{\delta}{D} K \quad ,$$

wobei D die Entfernung des Betrachters zum nächstgelegenen Punkt des Chunks und K ein perspektivischer Skalierungsfaktor ist. K kann folgendermaßen berechnet werden:

$$K = \frac{viewportwidth}{2 \tan\left(\frac{horizontalfov}{2}\right)}$$

Dabei ist *viewportwidth* die Breite des Bildes in Pixeln und *horizontalfov* der horizontale Öffnungswinkel der virtuellen Kamera. Diese Metrik ist sehr einfach und ein wenig willkürlich, weil der Darstellungsfehler immer in der Mitte des Bildes gemessen wird. Vorteilhaft ist jedoch, dass sie unabhängig von der Blickrichtung ist, wodurch die Darstellung stabil bleibt, wenn sich der Betrachter auf der Stelle dreht.

Die Darstellung des Terrains funktioniert folgendermaßen: Es wird ein maximal tolerierbarer Bildfehler τ gewählt. Die Traversierung des Baumes wird beim Wurzelelement begonnen und der Bildfehler ρ des Chunks entsprechend der aktuellen Betrachterposition berechnet. Ist der Fehler akzeptabel, also $\rho < \tau$, wird der Chunk dargestellt. Andernfalls werden rekursiv die Kindelemente überprüft, bis geeignete Chunks für die Darstellung gefunden sind.

Vermeidung von Löchern

Wie schon bei den bisher betrachteten Verfahren können auch beim Chunked LOD Löcher zwischen benachbarten Chunks auftreten, wenn diese unterschiedlich aufgelöst sind. Diesem Problem ließe sich begegnen, indem die Geometrie an den Rändern der Chunks auf ein gemeinsames Minimum reduziert wird; das hätte jedoch negative Auswirkungen auf die Darstellungsqualität. Eine individuelle Anpassung der Chunks aneinander ist hingegen nicht trivial. Ulrich schlägt statt dessen eine Reihe sehr pragmatischer Lösungen vor: Schräge Kanten (*flanges*) um die Meshes herum könnten die Löcher kaschieren, allerdings entstehen dadurch leicht Artefakte, da die Kanten sich gegenseitig durchdringen. Alternativ könnten die Löcher durch spezielle, genau passende Meshes (*ribbons*) gefüllt werden, diese sind jedoch in der Erzeugung kompliziert, da ihre Form von den aktuellen LOD-Stufen der beiden benachbarten Chunks abhängig ist. Die laut Ullrich beste Lösung ist ein Mittelweg – vertikale Schürzen (*skirts*) um jeden Chunk herum verdecken mögliche Lücken in der Geometrie. Die Oberkante der Schürzen stimmt exakt mit der Form des Chunks überein, während die Form der Unterkante relativ beliebig ist. Die Schürzen können mit der existierenden Textur des Chunks versehen werden. Abhängig vom Schwellenwert für den maximalen Bildfehler sind von den Schürzen nur wenige Pixel zu sehen, so dass der subjektive Eindruck nicht gestört wird.

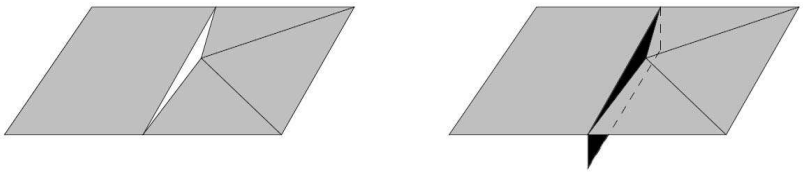


Abbildung 29: Vertikale Schürzen um das Mesh verhindern Löcher an den Kanten (Quelle: [ULR02]).

9.4.3.ROAM revisited

Das ursprüngliche Paper über ROAM wurde geschrieben, bevor 3D-Graphikkarten für private Computeranwender verfügbar wurden. Entsprechend gehen die Autoren implizit davon aus, dass die Liste der zu rendernden Dreiecke in jedem Frame neu erstellt wird, was für heutige Graphikkarten aus den zu Beginn von Kapitel 9.4 diskutierten Gründen suboptimal ist.

Snook [SNO03] stellt eine Implementation von ROAM vor, die dem aktuellen Stand der Technik Rechnung trägt. Dabei wird für das Terrain bzw. jeden einzelnen Chunk ein Vertex Buffer konstruiert, welcher der höchstmöglichen Terrainauflösung entspricht. In jedem Frame muss für die Szene ein neuer Index Buffer erzeugt und an die Graphikkarte übertragen werden. Diese Lösung ist akzeptabel, andererseits sind bei GeoMipMapping und Cunked LOD auch die Index Buffer statisch.

Ein weiterer kritischer Punkt ist, dass ROAM auf großen binären Bäumen operiert. Snook baut die Bäume sogar in jedem Frame komplett neu auf. Dabei entfällt ein beachtlicher Teil des Zeitbedarfs auf dynamische Speicherallokationen. Snook verwendet daher einen statischen Pool von Nodes; statt sie dynamisch zu allozieren, werden Nodes aus dem Pool verwendet, bis dieser erschöpft ist. Der Pool hat eine feste Größe, am Ende eines Frames werden die Bäume verworfen und alle Nodes im Pool sind wieder frei. In der festen Größe des Pools liegt die Schwäche dieses Ansatzes; die sukzessive Verfeinerung des Terrains muss beendet werden, sobald der Pool erschöpft ist.

9.4.4. Vergleich der GPU-orientierten Verfahren

Zusammenfassend noch einmal die wichtigsten Eckdaten der GPU-orientierten Verfahren im Überblick:

	GeoMipMapping	**Chunked LOD**	**ROAM nach Snook**
Präprozessing	nein	ja	nein
Vertex Buffer statisch	ja	ja	ja
Index Buffer statisch	ja	ja	nein
CPU-Last pro Frame	sehr gering (nur Wahl der Block-LOD-Stufe)	sehr gering (nur Wahl der Block-LOD-Stufe)	hoch (Erzeugung der Triangulierung mittels binärerer Bäume)
Strategie gegen sichtbares Umschalten des LOD	„Trilineares GeoMipMapping“ (Morphing)	Morphing	keine; Effekt nicht auffällig, da ständig neu trianguliert wird
Strategie gegen Löcher in der Geometrie	veränderte Triangulierung an den Blockrändern	„Schürzen“ (*skirts*) um die Blöcke herum	Algorithmus erlaubt nur geschlossene Triangulierung

9.5.Texturierung von Terrain

Die Geometrie bestimmt nur die Oberflächenform des Terrains, für das Aussehen und die Authentizität ist aber auch die Texturierung ausschlaggebend. Erst Texturen geben dem Terrain eine sichtbare Oberfläche und lassen Art und Beschaffenheit des Bodens erkennen. Der Boden kann aus den unterschiedlichsten Typen von Sand und Gestein bestehen oder er kann bewachsen sein – mit Moos oder Gras beispielsweise. Verschiedene Arten von Bewuchs seien hier der Einfachheit halber ebenfalls als Bodenmaterialien bezeichnet, da sie sich, in gewissen Grenzen, ebenfalls mit Texturen darstellen lassen.

Wie schon bei der Geometriedarstellung ist auch für die Texturierung erheblich, aus welcher Entfernung der Betrachter das Gelände sieht. Die einfachste Methode ist sicherlich, das gesamte Terrain mit einer einzigen Textur zu belegen, beispielsweise mit einem Satellitenbild. Dies mag für einige Anwendungen akzeptabel sein, diese Technik stößt jedoch an ihre Grenzen, sobald sich der Betrachter direkt im Gelände befindet und sehr kleine Details unmittelbar vor seinen Füßen erkennen kann. Wie beispielsweise Peasly [PEA01] feststellt, muss bei Echtzeitanwendungen, wie z.B. Computerspielen, mit engen Speicherbudgets gearbeitet werden. Eine akzeptable Ausführungsgeschwindigkeit ist nur zu erreichen, wenn alle Texturen im Speicher der Graphikkarte Platz finden; zudem haben Graphikkarten Limitierungen hinsichtlich der maximalen Auflösung von Texturen.

9.5.1.Unique Texturing

Die einfachste Möglichkeit der Terraintexturierung besteht darin, das gesamte Terrain mit einer einzigen, entsprechend großen Textur zu überziehen. Dieses Verfahren kommt insbesondere dann zum Einsatz, wenn eine real existierende Landschaft abgebildet werden soll. Üblicherweise liegen in solchen Fällen Satellitenfotos der Region vor, die direkt als Texturen verwendet werden können. Viele Programme zur Erzeugung fiktiver Terrains sind ebenfalls in der Lage, entsprechende Texturen zu erzeugen. Die dabei auftretenden Datenmengen sind nicht selten im Gigabytebereich angesiedelt; die für ein einziges Bild nötige Texturmenge kann in der Regel nicht gleichzeitig im Speicher eines Rechners untergebrachte werden, was für eine Echtzeitdarstellung allerdings notwendig wäre. Das Problem kann durch ein LOD-Verfahren umgangen werden, indem für weiter vom Betrachter entfernte Teile des Terrains grober aufgelöste Texturen verwendet werden. Dieses Vorgehen führt nicht zwangsläufig zu einer Verschlechterung der Bildqualität, da weit entfernte Teile des Terrains anteilig auch weniger Punkte im Bild einnehmen. Die Bildqualität kann im Gegenteil sogar steigen, denn diese Technik ist prinzipiell eine From von *MipMapping*.

Eine sehr elegante Form der Verwaltung der LOD-Texturen ist ein *Quadtree*. Die Terraintextur wird in $2^n \times 2^n$ gleich große rechteckige *Texturblöcke* der Auflösung x_{res} mal y_{res} Texel unterteilt. Diese Texturblöcke bilden die unterste Ebene des Quadtrees. Für jeweils vier im Quadrat benachbarte Terrainblöcke wird ein gemeinsamer Elternblock berechnet, der ebenfalls die Auflösung $x_{res,}\ y_{res}$ besitzt; dabei kommt je nach Texturtyp ein geeigneter Filteralgorithmus zum Einsatz. Dieses wird rekursiv fortgesetzt, bis schließlich als Wurzelelement ein einziger Block der Auflösung $x_{res,}\ y_{res}$ das gesamte Terrain repräsentiert. Welche LOD-Textur für einen bestimmten Terrainblock verwendet wird, wird anhand der Entfernung zum Betrachter entschieden. Plötzliches Umschalten sollte allerdings vermieden werden, da dieser Wechsel einem Betrachter unangenehm auffallen würde. Statt dessen bietet sich ein sanftes Überblenden über eine gewisse Entfernung an. Problematisch sind darüber hinaus die unter Umständen deutlich sichbaren Kanten zwischen Texturblöcken unterschiedlicher Auflösung.

Für Anwendungen wie z.B. Flugsimulationen sind Satellitenbilder als Texturierungsgrundlage meist ausreichend. Wenn der Betrachter das Terrain hingegen aus unmittelbarer Nähe, d.h. aus einem Abstand von einigen Metern oder weniger sehen kann, bietet diese Form der Texturierung nicht mehr genügend Details; die Geländetextur müsste hunderte Pixel pro Meter im virtuellen Terrain enthalten, was zu unvertretbar großen Texturen führen würde. Abhilfe bietet hier die Verwendung von *Detailtexturen* (*Detail Textures*). Dabei wird bei geringer Entferung zwischen Betrachter und Terrain über die Geländetextur multiplikativ eine zweite Textur gelegt, die ein unspezifisches Rauschen bzw. eine geeignete Struktur enthält. Diese Detailtextur wird so aufgebracht, dass sie pro Einheit im Weltkoordinatensystem wesentlich mehr Texel enthält als die Terraintextur, im Gegenzug ist sie verhältnismäßig klein und wird so oft gekachelt wie notwendig. Die Detailtextur täuscht auf diese Weise feine Oberflächendetails vor, die in der Terraintextur nicht vorhanden sind.

Unique Texturing stellt keinerlei Anforderungen an die Triangulierung des Terrains.

9.5.2.Kachelung

Peasley betrachtet in [PEA01] eine Technik, die schon seit den Anfängen der Computergraphik in unzähligen Variationen verwendet wird – den Aufbau des Terrains aus verschiedenen Kacheln.

Es besteht die Möglichkeit, Texturen über eine Fläche zu „kacheln“, d.h. die Textur ständig zu wiederholen. Die Anzahl der möglichen Wiederholungen wird wiederum von der Graphikhardware begrenzt. Aktuelle Graphikkarten unterstützen zwei Arten der Texturkachelung. Im Wrap-Modus wird die Textur einfach wiederholt, so dass an den rechten Rand wiederum der linke und an den unteren wiederum der obere anschließt. Im Mirror-Modus wird die Textur so wiederholt, als wäre sie an allen ihren Rändern gespiegelt, d.h. der rechte Rand trifft wieder auf den rechten Rand, der untere auf den unteren usw. (siehe Abb. 30).

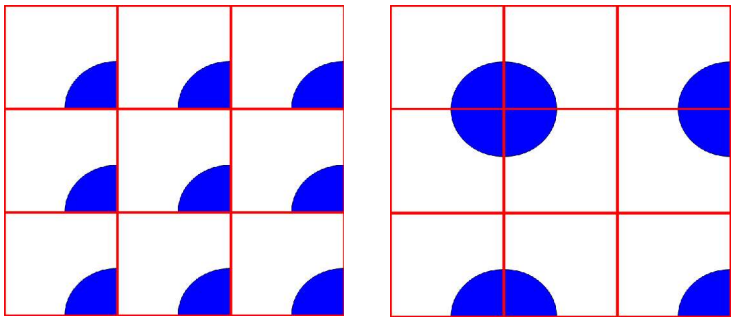


Abbildung 30: Texturierungsmodi Wrap (links) und Mirror (rechts), jeweils mit einem 9x9-Raster

Im Wrap-Modus muss sichergestellt werden, dass die jeweils entgegengesetzten Ränder der Textur so perfekt aneinander passen, dass die Verbindungskante für den Betrachter idealerweise nicht zu erkennen ist. Aktuelle Bildverarbeitungsprogramme bieten Werkzeuge, mit denen sich kachelbare Texturen automatisch aus beliebigen Bitmaps generieren lassen, wirklich gute Ergebnisse sind jedoch nur durch manuelle

Nachbearbeitung erreichbar. Der Vorteil des Mirror-Modus ist, dass jede Textur ohne Nachbearbeitung sofort kachelbar ist. Die Wiederholung der Textur, egal in welchem Modus, kann dem Betrachter leicht auffallen, insbesondere dann, wenn die Textur auffällige Merkmale enthält – dies muss vermieden werden.

Die von Peasley favorisierte Technik verwendet ein Schachbrettmuster, in dem jede Kachel mit einer anderen Textur versehen werden kann. Um maximale Variation zu erreichen, passen nicht nur die Ränder einer Texur im Wrap-Modus nahtlos aneinander, sondern die Textur kann zusätzlich in 90°-Schritten gedreht werden. Voraussetzung dafür ist, dass alle vier Kanten der Textur identisch sind; für einen talentierten Graphiker ist das kein Problem. Nicht jede Textur ist geeignet, um rotierbare Kacheln zu erstellen, sie sollte beispielsweise keine Lichteinfallsrichtung erkennen lassen. Um einen noch natürlicheren Eindruck zu erreichen, werden pro Bodenmaterial nicht nur eine, sondern mehrere beliebig rotierbare Kacheln erstellt. Um aneinander zu passen, müssen alle Kacheln in allen Kanten übereinstimmen; auch das ist mit herkömmlicher Bildbearbeitungssoftware mit vernünftigem Aufwand zu erreichen.

Für Übergänge zwischen verschiedenen Bodenmaterialien müssen spezielle Kacheln erstellt werden. Das Minimum sind drei beliebig drehbare Übergangskacheln pro Materialpaar.

Abbildung 31: zwei Kacheln, drei Übergangskacheln (Quelle: [PEA01])

Sollen an einer Stelle drei oder mehr Bodenmaterialien aneinander angrenzen dürfen, müssen für jede solche Kombination wiederum spezielle Übergangskacheln erstellt werden. Dies führt sehr schnell zu einer kombinatorischen Explosion, so dass in der Praxis nur eine begrenzte Anzahl von verschiedenen Materialübergängen zugelassen werden kann.

Die hier beschriebene Technik wird in einer Reihe von Computerspielen erfolgreich eingesetzt. Für das Verfahren spricht, dass sichtbare Wiederholungen in den Bodentexturen sich weitgehend eliminieren lassen und ein sehr natürlicher Eindruck entsteht. Dabei zielt das Verfahren darauf ab, ein enges Texturspeicherbudget möglichst effektiv zu nutzen. Der Entwickler behält ein hohes Maß an Kontrolle über das Endergebnis, da alle Materialübergänge individuell gestaltet werden können (und müssen). Die wesentlichen Nachteile sind, dass jede einzelne Textur von einem Graphiker aufwändig bearbeitet werden muss und dass nur bestimmte Materialien aneinander angrenzen dürfen. Letzteres Problem kann gelöst werden, indem die Kacheltechnik mit einem Verfahren kombiniert wird, das entweder Übergangskacheln automatisch vorberechnet oder im Bildspeicher Übergänge zwischen den Texturen erzeugt – beispielsweise Splatting (siehe 9.5.3). Des weiteren erfordert das Verfahren zwingend, dass die Triangulierung der Terraingeometrie dem Schachbrettmuster folgt.

9.5.3.Splatting

Charles Bloom stellt in [BLO00] eine Technik vor, die er als *Splatting* bezeichnet. Dabei werden Übergänge zwischen verschiedenen Bodenmaterialien dynamisch im Bildspeicher der Graphikkarte erzeugt. Es handelt sich hierbei um ein Multipassverfahren, d.h. es basiert darauf, das Terrain mehrfach mit unterschiedlichen Transparenzen übereinander zu zeichnen, um den endgültigen Effekt zu erzielen.

Jedem Bodenmaterial (z.B. Gras, Stein, Sand) wird eine kachelbare Textur zugeordnet. Die verschiedenen Bodentexturen werden schichtweise übereinandergezeichnet. Mittels Alphablending werden an den Materialrändern – also dort, wo verschiedene Bodenmaterialien aufeinander stoßen – weiche Übergänge geschaffen. Der Vorteil dieses Verfahrens ist, dass das manuelle Erstellen von Texturübergängen vollkommen entfällt. Der Anwender erstellt also z.B. nur eine reine Grastextur und eine reine Steintextur. Wenn im Gelände Gras und Stein aneinander grenzen, schafft der Splatting-Algorithmus automatisch einen weichen optischen Übergang. Die Anzahl von verschiedenen Bodenmaterialien, die an einer Stelle aufeinander treffen darf, ist im Gegensatz zu anderen Verfahren nicht begrenzt.

Splatting ist für Graphikhardware vom Niveau der nVidia GForce2 oder der Sony Playstation 2 konzipiert. Der Algorithmus läuft, abgesehen vom Präprozessing der Daten, vollständig auf der GPU und belastet die CPU fast gar nicht.

Basisalgorithmus

Es wird im Folgenden davon ausgegangen, dass die Geländegeometrie durch ein regelmäßiges, quadratisches Raster definiert wird. Ein Rasterelement wird als ein *Quad* bezeichnet; es besteht aus zwei Dreiecken und vier Eckpunkten. Die Höhenkoordinaten der Rasterpunkte bestimmen die Geländeform. Sie können direkt aus einem Height Field erzeugt werden.

Darüber hinaus sei eine Material Map gegeben, die in einem ebenfalls quadratischen Raster Bodenmaterialien definiert, z.B. Sand, Stein oder Gras (siehe Abb. 32). Zur Vereinfachung wird hier davon ausgegangen, dass die Rastergrößen für Geometrie und Bodenmaterialien übereinstimmen, d.h. dass jedem Quad genau ein Bodenmaterial zugeordnet ist. Der Algorithmus kann problemlos für gröber oder feiner aufgelöste Material Maps modifiziert werden. Die Auflösung der Material Map hat kaum Auswirkungen auf die Geschwindigkeit des Verfahrens, allerdings ist der Speicherbedarf linear davon abhängig.

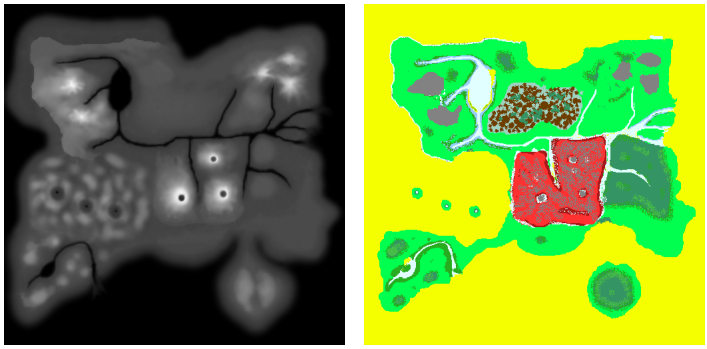


Abbildung 32: Heightmap (links) und Materialmap (rechts) für eine dem Dörner-Szenario nachempfundene Insel.

Jedem Bodenmaterial ist eine kachelbare Textur zugeordnet (siehe Abb. 33). Die weichen Übergänge zwischen den verschiedenen Materialtexturen werden mittels Alphablending von der Graphikhardware erzeugt. Beim Alphablending wird die Farbe eines zu zeichnenden Pixels mit dem an dieser Stelle bereits im Bildspeicher vorhanden Pixel vermischt, um die endgültige Pixelfarbe zu erhalten. Auf diese Weise können halbtransparente Objekte dargestellt werden – zunächst wird der Hintergrund gezeichnet, danach das halbtransparente Objekt im Alphablendingmodus. Die Farbe jedes Pixels wird nach der Formel

$$Farbe = Farbe_{Texel} * Alpha_{Source} + Farbe_{Bildspeicherpixel} * Alpha_{Destination}$$

berechnet. Dabei sind SourceAlpha und DestinationAlpha Werte zwischen 0 und 1. Da zwei Materialtexturen gemischt werden sollen, ohne den Gesamthelligkeitswert zu verändern, werden SourceAlpha und DestinationAlpha so gewählt, dass sie in der Summe 1 ergeben. Üblicherweise wird DestinationAlpha zu diesem Zweck als 1 - Sourcealpha definiert. Daher wird pro zu zeichnendem Pixel nur ein Alphawert zwischen 0 und 1 benötigt, der die Transparenz des Pixels angibt.

Abbildung 33: Jedem Bodenmaterial wird eine kachelbare Textur zugeordnet.

Die verschiedenen Materialtexturen werden nacheinander überlappend gezeichnet. Mittels Alphablending wird dabei dafür gesorgt, dass jedes Material am Rand nach außen hin zunehmend transparenter wird und somit die benachbarten, zuvor gezeichneten Materialien durchscheinen.

Bevor auf das Blending und insbesondere auf das Ermitteln der Alphawerte eingegangen wird, soll zunächst das überlappende Zeichnen der Materialtexturen erläutert werden.

Das Gelände wird in so genannte *Chunks* unterteilt. Ein Chunk ist ein rechteckiger Bereich aus Quads; eine adäquate Größe ist laut Bloom etwa 32x32. Die Unterteilung in Chunks erlaubt später dringend notwendige Optimierungen wie Culling oder LOD. Kleinere Chunks führen zu effizienterem Culling, allerdings auch zu wesentlich mehr Funktionsaufrufen an die Graphikhardware. Für eine konkrete Anwendung können optimale Werte experimentell bestimmt werden.

Für jeden Chunk werden zunächst die Bodenmaterialien ermittelt, die ihn beeinflussen; das sind alle Materialien, die im Chunk selbst vorkommen oder unmittelbar an den Chunk angrenzen. D.h. für einen Chunk von 32x32 Quads müssen im Beispiel 34x34 Felder in der Material Map betrachtet werden. Ergebnis dieses Schrittes ist eine Liste von Texturen, mit denen dieser Chunk gerendert werden muss.

Für jede dieser Texturen wird eine Liste mit allen Quads im aktuellen Chunk erstellt, die diese Textur verwenden oder an ein Quad angrenzen, das diese Textur verwendet – auch über die Chunkgrenzen hinaus. An Materialrändern sollen die Texturen ineinander geblendet werden. Wenn ein Quad mit Steintextur an ein Quad mit Grastextur grenzt, soll auch etwas Stein im Grasquad und etwas Gras im Steinquad zu sehen sein. Es müssen daher beide Quads mit beiden Texturen gerendert werden. Würde das Steinquad zusätzlich an ein Quad mit Sandtextur grenzen, müsste es dreimal gerendert werden – einmal mit Sand-, einmal mit Gras- und einmal mit Steintextur.

Die Liste von Quads, die mit der gleichen Textur gerendert werden, bezeichnet Bloom als *Splat*. Die Splats überlappen sich an den Rändern, wo verschiedene Materialien aufeinander treffen. Das Rendern des Chunks besteht letztendlich aus dem Rendern der einzelnen Splats in einer bestimmten Reihenfolge und mit Alphablending.

Für jeden Chunk wird ein Vertexbuffer erstellt. Für jeden Splat wird aus der Quadliste einen Indexbuffer erstellt, der Vertexbuffer wird nach dem Erstellen nicht mehr verändert. Das ist ein großer Vorteil, da er so nur einmal auf die Graphikkarte übertragen werden muss und dort verbleiben kann. Auch muss der Vertexbuffer während des Renderns der einzelnen Splats nicht umgeschaltet werden. Das mehrfache Rendern des gleichen Vertexbuffers ist auf aktueller Hardware wesentlich effizienter als das Rendern verschiedener Vertexbuffer.

Es bleibt die Frage, wie die Alphawerte für das Zeichnen der Splats ermittelt und gespeichert werden. Die klassische Lösung wäre, die Alphawerte in den Vertices des Terrains zu speichern. Da jedoch die Alphawerte für jeden Splat anders aussehen, würde das entweder bedeuten, dass jeder Splat einen eigenen Vertexbuffer benötigt oder der Vertexbuffer vor dem Rendern jedes Splats verändert werden müsste. Beide Lösungen sind aus Effizienzgründen nicht empfehlenswert. An dieser Stelle könnte argumentiert werden, dass sich durchaus mehrere Alphawerte in einem einzelnen Vertex unterbringen ließen. Dies würde allerdings einen etwas komplizierteren Vertexshader voraussetzen und bei älteren Graphikkarten zu Kompatibilitätsproblemen führen. Das Verfahren, das statt dessen vorgestellt werden soll, ist einfacher und stellt geringere Anforderungen an die Hardware.

Um die Alphawerte zu speichern, werden Blendtexturen verwendet. Pro Splat wird eine Blendtextur benötigt. Die Blendtextur überdeckt den gesamten Chunk, im Gegensatz dazu dürfen die Materialtexturen in einem Chunk zweimal oder öfter gekachelt werden. Beim Rendern wird die Texelfarbe aus der Materialtextur verwendet, aber der Alphawert aus der dazugehörigen Blendtextur. Die Auflösung der Blendtextur sollte etwa doppelt so groß sein wie die Auflösung der Material Map; im Beispiel mit 32x32 Quads pro Chunk und gleicher Auflösung der Material Map würden also Blendtexturen von 64x64 Pixeln verwendet werden. Dank der bilinearen Texturfilterung genügt diese Auflösung vollkommen, höhere Auflösungen bringen kaum eine optische Verbesserung und kosten Texturspeicher. Vier statt acht Bit Auflösung für die Alphawerte sind ebenfalls ausreichend.

Es wird nur der Alphakanal der Blendtexturen verwendet. Falls die Hardware es unterstützt, kann daher ein reines Alpha-Texturformat benutzt werden; andernfalls muss eine speicherintensivere RGBA-Textur verwendet werden.

Die Blendtexturen müssen einmalig aus der Material Map berechnet werden. Für jeden Pixel der Blendtextur wird der Einfluss des dazugehörigen Bodenmaterials an dieser Position im Chunk ermittelt. Die Bodenmaterialen sind pro Quad definiert. Es wird davon ausgegangen, dass der Einfluss eines Bodenmaterials im Zentrum des Quads am größten ist und mit zunehmendem Abstand davon kleiner wird.

Der Einfluss eines Punktes p1 auf einen anderen Punkt p2 sei definiert als:

$$Weight(\overline{p1}, \overline{p2}) = max(0; 1 - \frac{|\overline{p1} - \overline{p2}|^2}{1.75^2}) \quad .$$

Dazu werden die Texelkoordinaten der Blendmap in das Koordinatensystem der Materialmap umgerechnet. Im Beispiel (32x32 Quads/Chunk und 64x64 Pixel/Blendmap) hat das linke obere Quad den Mittelpunkt (0.5; 0.5), der linke obere Texel (0.25; 0.25). Für jeden Pixel der Blendmap ergibt sich der Einfluss der zugehörigen Materialtextur als Summe des Einflusses der umliegenden Quads mit diesem Material. Bei der obigen Formel genügt es, pro Pixel die 3x3 umliegenden Quads zu betrachten. Weiter entfernte Quads haben auf Grund der Entfernung immer einen Einfluss von 0.

Nachdem diese Berechnung für alle Splats des Chunks erfolgt ist, müssen die Gewichte pro Pixel normalisiert werden. Die Summe der Einflüsse der unterschiedlichen Materialien auf einen Pixel muss 1 betragen.

Die endgültigen Einflusswerte pro Pixel sind dann die Alphawerte für die Blendmaps.

Verfeinerung des Algorithmus

Bei dem bisher beschriebenen Verfahren ergeben sich einen Reihe von Problemen.

Erstens muss sichergestellt werden, dass die Splats in verschiedenen Chunks immer in der gleichen Materialreihenfolge gezeichnet werden. Der Grund dafür ist, dass beim Ineinanderblenden von mehr als zwei Materialien kein echter Mittelwert zustande kommt, sondern durch das Auftragen in mehreren Pässen die zuletzt gezeichneten Materialien dominieren. Angenommen die Materialien A, B und C treffen an einer Stelle aufeinander. Zuerst werden A und B im Verhältnis 50:50 (beispielsweise) gemischt. Danach wird das Ergebnis 50:50 mit C gemischt. Am endgültigen Pixel auf dem Bildschirm haben A und B je 25% und C 50% Anteil. Es könnte argumentiert werden, dass das korrekte Mischungsverhältnis 33% für jedes Material betragen müsste. Meistens ist die Dominanz der zuletzt gezeichneten Materialien optisch jedoch gar kein Problem, vorausgesetzt, die Zeichenreihenfolge und damit die Dominanz der Materialien ist in allen Chunks gleich. Andernfalls können an den Chunkgrenzen optische Anomalien entstehen – die Chunkgrenzen sind dann deutlich zu sehen, weil das gleiche Material in benachbarten Chunks unterschiedlich hell erscheint (siehe Abb. 34).

Abbildung 34: Ist die Reihenfolge, in der Splats gezeichnet werden, nicht für alle Chunks gleich, entstehen optische Anomalien an den Kanten der Chunks.

Ein zweites Problem ist, dass das Zeichnen der Splats den Bildspeicher nicht notwendigerweise völlig deckend überschreibt. An den Grenzen zwischen zwei Splats können transparente Bereiche entstehen, in denen der bisherige Inhalt des Bildspeichers durchscheint. Beispiel: Zwei Materialien A und B stoßen aneinander. Zuerst wird Splat A gezeichnet und trifft einen bestimmten Pixel mit 50% Transparenz. Danach wird Splat B gezeichnet und trifft den Pixel ebenfalls mit 50% Transparenz. Die Farbe des resultierenden Pixels wird somit zu 25% vom Inhalt des Bildspeichers vor dem Zeichnen des Terrains bestimmt. Eine ganz einfache Lösung für dieses Problem wäre, das gesamte Terrain in einem ersten Durchgang mit einer Basistextur ohne jede Transparenz zu rendern. Als Basistextur bietet sich aus Effizienzgründen diejenige Textur an, die insgesamt am häufigsten vorkommt. Für diese Textur müssen keine Splats gezeichnet werden; sie ist überall dort zu sehen, wo die anderen Splats sie nicht völlig überschreiben. Dadurch wird dieses Material automatisch zum am wenigsten dominanten. Leider ist diese Lösung nicht optimal; die Basistextur scheint teilweise sehr deutlich an Stellen durch, an denen das entsprechende Bodenmaterial gar nicht vorkommt (siehe Abb. 35).

Abbildung 35: Wenn zwei Splats aneinander angrenzen, kann eine Lücke entstehen, durch die ein drittes, zuvor gezeichnetes Material durchscheint (linkes Bild). Das Aufbringen einer speziell berechneten Basistextur (mittleres Bild) behebt das Problem (rechtes Bild).

Eine bessere Lösung für das Problem ist das Vorberechnen einer individuellen Basistextur für jeden Chunk. Eine angemessene Größe für die Basistextur ist laut Bloom etwa die doppelte Auflösung der Blendtextur, also im genannten Beispiel 128x128 Pixel.

Ein weiterer Vorteil der vorberechneten Basistextur ist, dass ab einer bestimmten Entfernung zwischen Kamera und Chunk komplett auf das aufwändige Rendern der Splats verzichtet und statt dessen nur noch die Basistextur verwenden werden kann.

Es gibt eine offensichtliche Optimierung des Verfahrens: Chunks, die nur von einem einzigen Bodenmaterial beeinflusst werden, können direkt, d.h. ohne Verwendung von Splats, gerendert werden. (Zur Erinnerung: beeinflussende Materialien sind im Chunk vorkommende und direkt angrenzende Materialien.) Entsprechend kann für diese Chunks auf die Generierung von Blendmaps und der Basistextur komplett verzichtet werden. Selbst wenn die Basistextur als LOD-Stufe eingesetzt wird, kann bei Chunks mit nur einem Material darauf verzichtet werden – das Mipmapping der Graphikkarte erzeugt ein beinahe identisches Ergebnis.

9.5.4.Multi-Layer Terrain Shader

Die Grundidee des Splatting ist das Ineinanderblenden verschiedener kachelnder Materialtexturen. Es gibt eine Vielzahl von Variationen dieses Ansatzes; an dieser Stelle sollen nur zwei wesentliche erwähnt werden.

Das Blending muss nicht wie beim Splatting nach Bloom im Framebuffer durch mehrere Renderpässe geschehen. Die Texturen können statt dessen auch mittels *Multitexturing* in einem einzigen Pass kombiniert werden, insofern die Graphikhardware dieses unterstützt; weniger Renderpässe bedeuten einen Performancevorteil. Die Anzahl Texturen, die eine GPU pro Renderpass unterstützt, ist in jedem Fall begrenzt. Splatting nach Bloom setzt bereits eine multitexturingfähige Hardware voraus, es genügen jedoch zwei Texturen pro Pass. Die momentan besten Graphikkarten im PC-Consumerbereich unterstützen maximal 16 Texturen pro Pass, die am weitesten verbreiteten Modelle jedoch nur zwischen 4 und 8. Die Anzahl von Bodenmaterialien, die in einem einzigen Pass gerendert werden können, ist somit begrenzt. Neben den Materialtexturen müssen auch die Blendgewichte der GPU in Form einer Textur zur Verfügung gestellt werden. Die Nebula2-Engine [RAD05] beispielsweise verwendet hierzu eine RGBA-Textur, wobei jeder der vier Kanäle der Textur Blendgewichte für eine Materialtextur enthält. Nebula2 unterstützt nur bis zu vier verschiedene Bodenmaterialien, somit muss die Hardware fünf Texturen pro Renderpass unterstützen. Ist das nicht der Fall, wird wie beim Splatting auf vier getrennte Renderpässe zurückgegriffen. Die Kombination eines Multi-Layer Shaders, wie Nebula2 ihn benutzt, mit einem Mulitpassverfahren ist nicht trivial; das Verfahren ist somit nur geeignet, wenn die Anzahl verschiedener Bodenmaterialien entsprechend begrenzt ist.

Beim Splatting nach Bloom werden die Blendtexturen automatisch aus der Material Map berechnet. Es ist alternativ möglich, auf eine Material Map komplett zu verzichten und statt dessen direkt individuelle Blendtexturen für jedes Material zu erstellen. Diese Technik kommt im Computerspielbereich bevorzugt zum Einsatz, da Terrains hier von Künstlern gestaltet werden und ein hohes Maß an Kontrolle über das Endergebnis erwünscht ist. Im Gegensatz zum ursprünglichen Splatting können Übergänge zwischen Bodenmaterialien individuell gestaltet und über beliebige Distanzen ausgedehnt werden, auch harte Kanten sind möglich. Der menschliche Arbeitsaufwand ist hierbei allerdings ungleich höher als beim Splatting.

9.6.Terraindarstellung in 3DView2

Ausgangsituation

Zum Zeitpunkt dieser Arbeit ist die MicroPsi-Weltsimulation komplett zweidimensional, d.h. das Gelände wird als perfekte Ebene angenommen. Prinzipiell haben alle Objekte dreidimensionale Koordinaten, jedoch ist die dritte Komponente von Positionen, Geschwindigkeiten etc. in der aktuellen Implementation meist null, da sich fast alle Objekte auf dem Boden befinden. Es ist angedacht, in Zukunft echte dreidimensionale Terrains in der Simulation zu unterstützen, jedoch wird dieses Feature bisher nicht benötigt und hat daher bei der Entwicklung keine Priorität. Die bisherige Weltdarstellung war ebenfalls rein zweidimensional; die Entwicklung einer dreidimensionalen Weltsimulation erfordert zunächst die Möglichkeit der dreidimensionalen Visualisierung und Werkzeuge zum Bearbeiten einer dreidimensionalen Welt – beides ist Gegenstand dieser Arbeit. Es wurde daher entschieden, die Weltsimulation vorerst nicht anzupassen und die dritte Dimension allein in der Visualisierung existieren zu lassen. Das bedeutet, 3DView2 zeigt ein dreidimensionales Terrain mit Hügeln und Tälern an, die Weltsimulation betrachtet das Gelände aber weiterhin als Ebene. Die Höhen-Koordinaten der Objekte (positive Z-Achse in der Weltsimulation) werden von 3DView2 als Höhe über dem Boden interpretiert.

Die Bodenbeschaffenheit, also die Verteilung verschiedener Bodenmaterialien, wird der MicroPsi-Weltsimulation in Form einer Material Map zur Verfügung gestellt. Wasserflächen werden dabei ebenfalls als Bodenmaterial modelliert. Bislang wird die Bodenbeschaffenheit von der Weltsimulation noch nicht konsequent genutzt, es wird lediglich zwischen begehbarem Boden (Gras) und nicht begehbarem Boden (Wasser) unterschieden; Wasserflächen dienen dabei als natürliche Begrenzung der Simulationsumgebung. Geplant ist, dass Bodenmaterialien – ähnlich wie in Dörners Inselsimulation – Einfluss auf Bewegungsgeschwindigkeit und Energieverbrauch haben sollen und dass beispielsweise Pflanzen auf unterschiedlichen Böden unterschiedlich gut gedeihen.

Wahl und Kombination der Renderingverfahren

Während für die Darstellung der Geländeform eine ganze Reihe von Standardverfahren zur Verfügung stehen, erweist sich eine funktionale und gleichzeitig optisch ansprechende Texturierung des Terrains als das schwierigere Problem. Die Wahl eines Texturierungsverfahrens hat zudem Einfluss auf die Geometriedarstellung, da die verschiedenen Texturierungsverfahren auch bestimmte Anforderungen an die Beschaffenheit der Geländegeometrie stellen. Die Wahl fiel auf eine Variante des von Charles Bloom beschriebenen „Splatting“. Ausschlaggebend dafür war vor allem die Tatsache, dass dieses Verfahren Material Maps einsetzt. Material Maps sind erstens eine intuitive Darstellungsform, sie sind sehr leicht – d.h. mit geringem Zeitaufwand und Standardbildbearbeitungsprogammen – zu erstellen und zu pflegen und zweitens setzt die existierende Weltsimulation bereits Material Maps ein, wodurch die Möglichkeit besteht, die 3D-Visualisierung ohne zusätzliche Arbeitsschritte auf den gleichen Terraindaten arbeiten zu lassen wie die Weltsimulation.

Splatting ist ein Kompromiss zugunsten einfacher Handhabbarkeit und geringem Aufwand zur Erstellung neuer Simulationswelten. Flug- oder Fahrzeugsimulationen verwenden meist eine Texturierung mit Hilfe von Quadtrees und Satellitenbildern, da die Datenmenge keine Rolle spielt und eine individuelle Texturierung eine hervorragende Qualität liefert, insbesondere, wenn real existierende Landschaften abgebildet werden

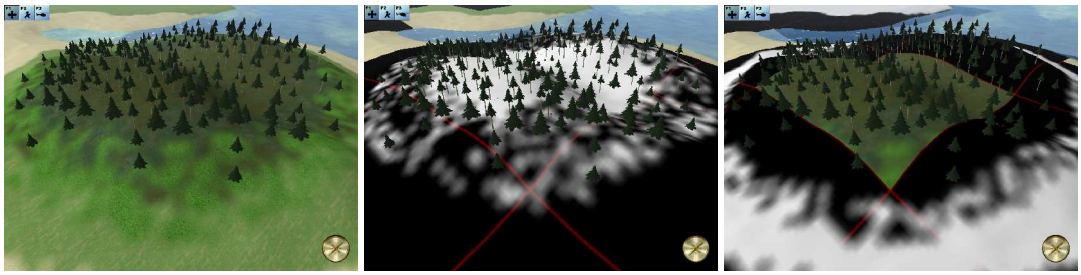


Abbildung 36: 3DView2-Terrain mit Blendmaps für zwei verschiedene Bodenmaterialien. Rote Kanten kennzeichnen die Grenzen von Terrainblöcken.

sollen. 3DView2 ähnelt bezüglich der Anforderungen mehr einem Computerspiel, jedoch setzen Computerspiele in der Regel auf die manuelle Erstellung von Blendmaps, weil dieses Vorgehen ein hohes Maß an Kontrolle über das Endergebnis erlaubt.

Im Hinblick auf die Wahl eines Verfahrens für die Geometrie-Darstellung kam für 3DView2 nur ein GPU-freundliches Verfahren in Frage. Alle hier vorgestellten Verfahren können mit Eingabedaten in Form von Height Maps arbeiten, was wiederum der Forderung nach einfacher Handhabbarkeit genügt. Es ist vorstellbar, dass in Zukunft auch die MicroPsi-Weltsimulation Height Maps zur Repräsentation der Terraintopologie nutzen könnte.

Der Chunked-LOD-Algorithmus erscheint für 3DView2 ungeeignet, weil er einen aufwändigen Vorverarbeitungsschritt erfordert; es sollte die Option offen gehalten werden, Terraindaten in kurzer Zeit über ein Netzwerk zu übertragen und anzuzeigen. Gegen ROAM sprach einerseits die Tatsache, dass die CPU stärker als bei den anderen Verfahren beansprucht wird und dass ROAM andererseits nur schwer mit Splatting kombinierbar ist.

Wie schon erläutert geht der Splatting-Algorithmus von einer festen Triangulierung des Terrains aus, nämlich von einem regelmäßigen Raster von Quads. Für jeden Splat wird dabei ein individueller Indexbuffer erstellt. Das bedeutet, dass zumindest innerhalb des Bereiches, in dem Splats gerendert werden, keine Reduktion der Geländegeometrie vorgenommen werden kann. Weiter entfernte Geländeteile, die lediglich mit der Basistextur gerendert werden, können hingegen beliebig vereinfacht werden.

Sowohl der ROAM- als auch Chunked-LOD-Algorithmus könnten mit künstlichen Mitteln gezwungen werden, auf Terrainvereinfachungen innerhalb eines bestimmten Bereiches zu verzichten. Beim GeoMipMapping-Verfahren ist diese Forderung jedoch am leichtesten umzusetzen, denn der GeoMipMapping- und der Splatting-Algorithmus arbeiten auf den gleichen elementaren Terraineinheiten; die Terrainblöcke, die bei der Beschreibung des Splatting-Algorithmus als Chunks bezeichnet wurden, entsprechen genau GeoMipMaps der Stufe 0. Somit können die Verfahren leicht kombiniert werden. Terrainblöcke, die mit Splats gerendert werden, müssen immer Geomipmapstufe 0 verwenden, alle anderen Terrainblöcke können ihre Mipmapstufe entsprechend ihrer Geometrie und Entfernung zum Betrachter frei wählen.

Das von de Boer vorgeschlagene Verfahren zur Vermeidung von Löchern zwischen Terrainblöcken – eine veränderte Triangulierung an den Rändern der Blöcke – ist ebenfalls nicht mit der vom Splatting verlangten Triangulierung vereinbar. Einen einfachen Ausweg bieten jedoch die von Ullrich für Chunked LOD vorgeschlagenen Schürzen; sie kaschieren Lücken zwischen Terrainblöcken, ohne die Triangulierung der Blöcke selbst zu

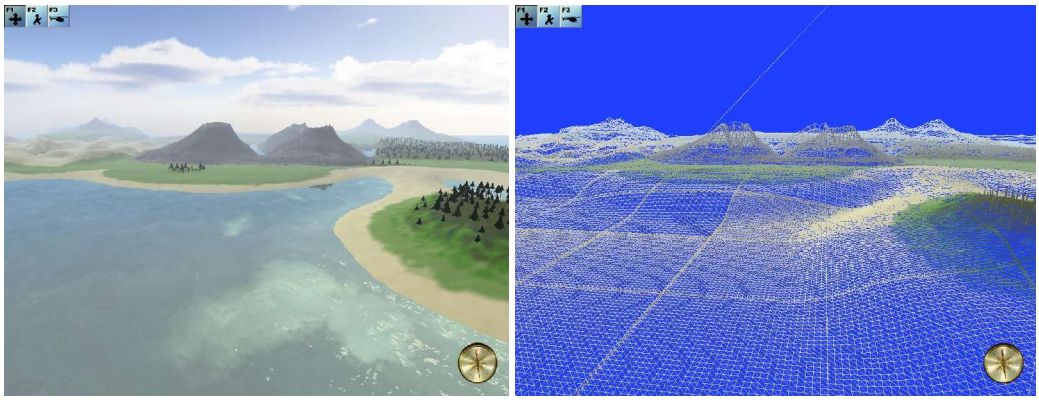


Abbildung 37: 3DView2 verwendet GeoMipMaps für die Terraingeometrie. Um die Kombination mit Splatting zu ermöglichen, werden Schürzen zur Kaschierung von Lücken zwischen den Terrianblöcken verwendet.

beeinflussen. Abbildung 37 zeigt ein 3DView2-Terrain als Wireframe, deutlich zu erkennen sind das abnehmende geometrische Detail mit steigender Entfernung und die Schürzen an den Blockrändern.

3DView2 arbeitet mit Height Maps der Größe 2^n+1 und Material Maps der Größe 2^n, wobei n ganzzahlig und für Height Map und Material Map identisch sein muss. Die Maps werden vom Anwender in Form von Bitmapdateien in gängigen Formaten bereitgestellt. Sind die Anforderungen an die Bitmapgrößen nicht erfüllt, werden die Bitmaps vom Programm automatisch auf gültige Werte skaliert. Eine ausführliche Anleitung zur Erstellung von Terrains findet sich in Anhang A.

Das Terrain wird dynamisch von einer direktionalen Lichtquelle beleuchtet. Lichtwerte werden pro Vertex berechnet und über die Fläche eines Dreiecks interpoliert (*Gouraud Shading*). Die Normalenvektoren der Terrainvertices werden als Mittelwert der Normalenvektoren der adjazenten Flächen berechnet.

Alle Terrainblöcke werden gemäß ihrere räumlichen Position in einen Quadtree eingeordnet, dadurch kann effizient ermittelt werden, welche Terrainblöcke zu einem gegeben Zeitpunkt im Blickfeld der Kamera liegen und gezeichnet werden müssen (*View Frustum Culling)*.

Umgang mit den Terraingrenzen

Das durch Height Map und Material Map definierte Terrain hat eine endliche Größe und somit ein „Ende“. Ein solches Ende zerstört für einen Betrachter jedoch die Illusion einer virtuellen Welt; selbst wenn ihm nur die Bewegung innerhalb des definierten Terrains erlaubt ist, kann er dennoch weit über die Grenzen dieses Terrains hinaus blicken. Daher ist es notwendig, das Terrain an den Rändern in geeigneter Form fortzusetzen.

3DView2 unterstützt zwei Modelle, mit Terraingrenzen umzugehen. Im einfachsten Fall wird die Bewegung des Betrachters auf das definierte Terrain und einen rechteckigen Bereich um dieses Terrain beschränkt, d.h. der Betrachter hat die Möglichkeit, sich entlang der x- und y-Achse jeweils um eine konstante Strecke vom Terrain zu entfernen. Dadurch wird ihm beispielsweise ermöglicht, eine Insel aus größerer Entfernung zu betrachten. Das Terrain selbst wird an den Rändern mit „flachen“ Terrainblöcken so weit fortgesetzt, wie ein Betrachter vom Rand seines Bewegungsbereiches aus maximal blicken kann. „Flache“

Terrainblöcke sind Terrainblöcke, die durchgängig Höhenwerte von Null aufweisen und alle mit dem gleichen Material bedeckt sind; daher können sie ohne Splatting gerendert und mit nur jeweils zwei Dreiecken dargestellt werden. Das 3DView2-Terrainsystem erlaubt es, einen einzelnen Terrainblock an verschiedenen Positionen innerhalb der Welt anzuzeigen und mit verschiedenen Positionen im Terrain-Quadtree einzuordnen. Daher genügt ein einziger physischer flacher Terrainblock um den sichtbaren Bereich ausserhalb des definierten Terrains aufzufüllen. Damit zwischen dem durch die Height Map definierten und dem sie umgebenden flachen Terrain keine Lücken entstehen, wird die Height Map vor der Generierung aller Terrainblöcke entsprechend angepasst – ihre Ränder werden auf Höhe null gesetzt.

Alternativ zur Fortsetzung des Terrains mit flachen Terrainblöcken gibt es den so genannten „Wrap-Around"-Modus. Dabei wird das Terrain selbst gekachelt, d.h. an den linken Rand des Terrains schließt sich der rechte an und an den oberen Rand der untere. Dies entspricht der Projektion des Terrains auf einen Torus. Ein Betrachter kann sich in diesem Modus beliebig lange in die gleiche Richtung bewegen, ohne jemals an eine Grenze zu stoßen. Tatsächlich durchschreitet er wieder und wieder das gleiche Terrain; seine Bewegungsfreiheit ist auf das durch die Height Map definierte Terrain eingeschränkt. Anders als im Standardmodus wird er aber nicht am Überschreiten der Terraingrenzen gehindert, sondern an das jeweils gegenüberliegende Ende des Terrains versetzt. Das Terrain wird an den Rändern entsprechend der maximalen Sichtweite mit den gleichen Terrainblöcken fortgesetzt, dadurch entsteht die Illusion einer sich endlos wiederholenden Landschaft. Damit an den Rändern keine Lücken entstehen, muss die Height Map selbst kachelbar sein. Dieses wird vor der Generierung der Terrainblöcke erzwungen, indem jeweils der linke und rechte bzw. obere und untere Rand der Height Map gleichgesetzt werden.

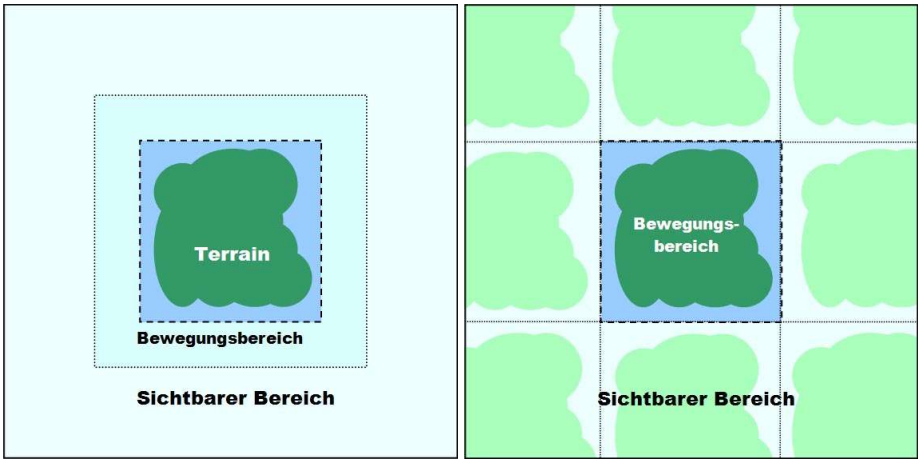


Abbildung 38: Das Terrain kann entweder mit flachen Blöcken fortgesetzt (links) oder im Wrap-Around-Modus gekachelt werden (rechts).

10.Wasserdarstellung

Wasser ist der realen Welt allgegenwärtig, daher beschäftigt sich auch die Computergraphik von ihrem Beginn an mit der Simulation und Darstellung von Wasser.

Die *Navier-Stokes-Gleichungen* [FOS96] ermöglichen eine akkurate Simulation von Wasser bzw. beliebigen, nicht-kompressiblen Flüssigkeiten. Es handelt sich dabei um ein System nicht-linearer, partieller Differentialgleichungen, die auf Newtons Impulserhaltungssatz basieren. Sie stellen Druck und Geschwindigkeit an jedem Punkt der Flüssigkeit in Abhängigkeit der Zeit dar, dabei werden Gravitation und Viskosität des Mediums berücksichtigt. Für kompressible Fluide, wie z.B. Luft, gibt es abgewandelte Formen der Gleichungen, die auch die Dichte berücksichtigen. Die Navier-Stokes-Gleichungen gehen davon aus, dass die Flüssigkeit beliebig teilbar ist. Analytisch sind die Gleichungen bislang jedoch nicht lösbar; üblicherweise werden statt dessen zusätzliche vereinfachende Annahmen gemacht oder es wird ein geeignetes Näherungsverfahren eingesetzt.

Eine physikalisch korrekte Simulation von Wasser ist also möglich, in der Praxis jedoch extrem aufwändig. Insbesondere für Echtzeitanwendungen ist es deshalb erforderlich, stark vereinfachende Annahmen zu treffen. Dazu ist es notwendig, verschiedene Szenarien zu unterscheiden. Im schwierigsten Fall wird eine Menge Wasser in einer beliebigen dreidimensionalen Umgebung simuliert, wobei das Wasser frei fließen kann. Die Simulation wird erheblich einfacher, wenn sie auf zwei Dimensionen eingegrenzt wird – d.h. das Wasser hat nur eine Oberfläche, kein echtes Volumen. Die Oberfläche kann allerdings unterschiedliche Höhen aufweisen. Um einen See oder das offene Meer akkurat darzustellen, genügt es unter Umständen, nur das Verhalten dieser Oberfläche zu simulieren und Druck und Massetransport komplett außer acht zu lassen.

Auf der anderen Seite ist für viele Anwendungen eine korrekte Simulation gar nicht erforderlich, vielmehr zählt nur ein subjektiv korrekter Eindruck. Um diesen zu erreichen sind Kenntnisse der zugrunde liegenden physikalischen Gesetzmäßigkeiten notwendig, ohne dass diese jedoch exakt abgebildet werden müssen. Wieder sind starke Vereinfachungen möglich, wenn das Szenario entsprechend eingegrenzt wird. Im offenen Meer kommt es vor allem auf einen realistischen Wellengang an. Geeignete Verfahren zur Erzeugung von realistisch anmutenden Wellen sind z.B. Fourier-Synthese und Gerstner-Wellen [TES01], [JEN01]. In flachen Gewässern hingegen sind nur sehr kleine Wellen zu beobachten, so genannte Kapillarwellen mit Wellenlängen bis 2 cm. Bei jedem Gewässer treten an der Wasseroberfläche Reflexionen, Refraktionen und Lichtstreuungen auf; das Wasser erhält dadurch in jeder Umgebung ein anderes, charakteristisches Aussehen.

Eine umfassende Diskussion dieser Materie würde den Rahmen dieser Arbeit sprengen, daher sei für einen Überblick über Simulationsverfahren für und Darstellung von Wasser auf [MAT05] verwiesen. An dieser Stelle sollen lediglich die für 3DView2 zutreffenden Einschränkungen festgelegt und im Anschluss die für diesen Fall relevanten Verfahren behandelt werden.

3DView2 soll ein Inselszenario ähnlich dem von Dörner visualisieren. Es können folgende Einschränkungen vorgenommen werden:

1. Eine physikalische Wassersimulation ist nicht notwendig, es genügt der subjektiv korrekte Eindruck.

2. Es gibt nur einen schwachen Wellengang.

3. Es gibt einen einheitlichen Meeresspiegel und keine höher gelegenen Seen, Wasserfälle oder ähnliches.

4. Das Wasser ist klar genug, dass der Betrachter bis zu einer bestimmten Tiefe den Grund und Objekte auf dem Grund erkennen kann.

5. Umgebung und Objekte sollen sich bei entsprechendem Blickwinkel korrekt im Wasser spiegeln.

10.1.Reflexion

Lokale Reflexion

An Wasseroberflächen werden Anteile des einfallenden Lichtes reflektiert. Dabei gilt, dass der Einfallswinkel des Lichtes, d.h. der Winkel zwischen Oberflächennormale und Lichtvektor, gleich seinem Ausfallswinkel ist, d.h. dem Winkel zwischen Oberflächennormale und Vektor des reflektierten Lichtstrahls. Gegeben sei ein Lichtstrahl, der auf einen Punkt *x* eine reflektierenden Oberfläche trifft:

$$f_l(t)=\vec{x}-t\,\vec{L} \qquad t\geq 0 \quad .$$

Dabei ist *L* die Richtung des einfallenden Lichtes. Gesucht ist die Richtung des ausfallenden Lichtes *R*, so dass

$$f_r(t)=\vec{x}+t\,\vec{R} \qquad t\geq 0$$

der ausfallende Lichtstrahl ist. *R* lässt sich mit Hilfe der Oberflächennormale *N* in *x* folgendermaßen berechnen [FOL97]:

$$\vec{R}=2\,\vec{N}\,(\vec{N}\cdot\vec{L})-\vec{L}$$

In der Computergraphik wird diese Formel genau entgegengesetzt angewendet. Relevant sind nur diejenigen Lichtstrahlen, welche senkrecht auf die virtuelle Kamera treffen. Daher werden nur genau diese Strahlen konstruiert und zu jedem wird der Ursprungspunkt in der Szene ermittelt. Die Farbe des sichtbaren Pixels ergibt sich aus der Farbe dieses Ursprungspunktes, also des reflektierten Objektes, und den Eigenschaften der reflektierenden Oberfläche, die nicht notwendigerweise zu einhundert Prozent reflektierend sein muss, sondern darüber hinaus auch selbst eine Farbe haben kann. Da einfallendes und reflektiertes Licht sich symmetrisch zueinander verhalten, können *L* und *R* in der Berechnung vertauscht werden, somit ergibt sich:

$$\vec{L}=2\,\vec{N}\,(\vec{N}\cdot\vec{R})-\vec{R} \quad .$$

Das Verfolgen von Lichtstrahlen von der Kamera zu ihren Ursprungspunkten in der Szene ist das Grundprinzip von Raytracing-Verfahren. Solche Verfahren werden jedoch von der handelsüblichen Hardware noch nicht unterstützt und sind wegen ihrer Komplexität bislang nicht für Echtzeitanwendungen geeignet. Es gibt jedoch vereinfachte Algorithmen, die zumindest in speziellen Anwendungsfällen sehr gute Ergebnisse liefern können.

Globale Reflexion

Bei der so genannten globalen Reflexion (*global reflection*) [LOM04] wird davon ausgegangen, dass das reflektierte Objekt, also der Ursprungspunkt des Strahles f_l, unendlich weit entfernt ist. Der bisher betrachtete allgemeinere Fall wird analog auch als lokale Reflexion (*local reflection)* bezeichnet. Durch die Annahme, das reflektierte Objekt sei unendlich weit entfernt, ist der in der Reflexion sichtbare Punkt nur noch vom Lichtvektor L und nicht mehr vom Reflexionspunkt x abhängig. Das bedeutet, dass ein Betrachter auf der reflektierenden Oberfläche immer das gleiche Spiegelbild sieht; selbst, wenn er sich relativ zu ihr bewegt. Das Spiegelbild ändert sich nur in Abhängigkeit vom Blickwinkel auf die reflektierende Oberfläche.

Auf 3D-Graphikkarten lassen sich globale Reflexionen sehr leicht mittels so genannter *Cube Maps* implementieren. Eine Cube Map ist eine Textur, die in Wahrheit aus sechs quadratischen Texturen gleicher Größe besteht. Diese Texturen entsprechen den Seitenflächen eines Würfels. Auf eine Cube Map wird nicht über zweidimensionale Texturkoordinaten, sondern mittels eines (normalisierten) Vektors zugegriffen; jedem Vektor ist eindeutig ein Punkt der Cube Map zugeordnet.

Eine typische Anwendung von globalen Reflexionen ist die Spiegelung des Himmels bzw. von Objekten am Horizont auf diversen Oberflächen. Mit anderen Worten: Als Cube Map für globale Reflexionen wird häufig direkt die Skybox verwendet. Zwar sind weder Himmel noch Horizont wirklich unendlich weit entfernt, jedoch kann sich die Kamera in dem meisten Anwendungen nicht weit oder nicht schnell genug bewegen, als dass diese Vereinfachung subjektiv ins Gewicht fallen würde.

Globale Reflexionen können auch dann gut eingesetzt werden, wenn das Spiegelbild aufgrund der Beschaffenheit der reflektierenden Oberfläche nicht klar zu erkennen sein wird. Beispielsweise spiegelt die Oberfläche einer gläsernen Vase nur schwach, ein großer Teil das Lichtes wird hindurch gelassen. Zudem ist die Oberfläche der Vase gekrümmt oder in Facetten geschliffen. Für einen Betrachter ist somit nur schwer zu erkennen, ob das Spiegelbild in der Vase tatsächlich physikalisch korrekt ist. Hier kann eine globale Reflexion eingesetzt werden, indem aus der Umgebung der Vase eine Cube Map berechnet wird; dies kann in einem einmaligen Vorverarbeitungsschritt geschehen. Diese Implementation ist sehr effizient und hält einer oberflächlichen Betrachtung stand.

Lokale Reflexion an einer Ebene

Auch lokale Reflexionen lassen sich auf 3D-Graphikkarten effizient implementieren, wenn die Einschränkung gemacht wird, dass die Reflexionsfläche vollkommen eben sein muss. Abbildung 39 verdeutlicht die Grundidee.

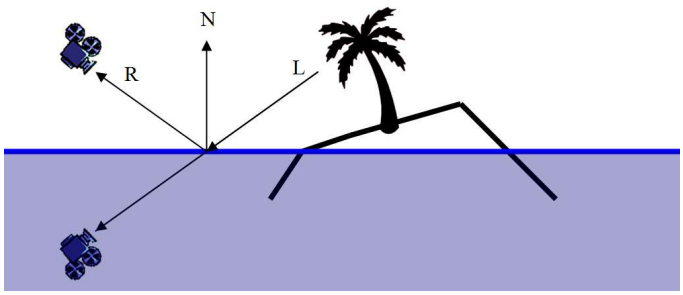


Abbildung 39: Reflexionen an einer planaren Wasseroberfläche können gerendert werden, indem die Position der Kamera an der Oberfläche gespiegelt wird.

Zu jeden Lichtstrahl *R*, der senkrecht auf die Kamera trifft, existiert ein Lichtstrahl *L,* der seinen Ursprung irgendwo in der Szene hat und bestimmt, welches Objekt die Kamera am Reflexionspunkt sieht. Gegeben sei eine zweite Kamera, die in ihrem Öffnungswinkel der ersten entspricht, deren Position und Rotation aber an der reflektierenden Oberfläche gespiegelt ist. Im Falle einer Wasseroberfläche befindet sich diese zweite Kamera dadurch unter Wasser. Da die Reflexionsfläche planar ist, trifft für jeden Strahl *R*, der senkrecht auf die erste Kamera trifft, *L* senkrecht auf die zweite Kamera. Dieser Umstand kann mit Hilfe von *Render Target Textures* ausgenutzt werden.

Das Rendern von texturierter Geometrie auf den Bildschirm ist nichts anderes als Beschreiben eines Speicherbereiches auf der Graphikkarte, der anschließend als Bildschirmspeicher genutzt wird. Analog ist es möglich, in einen Speicherbereich zu rendern und diesen anschließend als Textur für ein anderes Rendering zu nutzen; diese Textur bezeichnet man daher als Render Target Texture.

Eine Möglichkeit, eine spiegelnde Wasseroberfläche zu simulieren, besteht darin, zunächst das gewünschte Spiegelbild in eine Textur zu rendern und diese Textur anschließend zum Zeichnen der Wasserfläche zu verwenden [VLA02]. Diese Textur wird entsprechend ihrer Verwendung auch als *Reflection Map* bezeichnet. Zum Rendern des Spiegelbildes wird die Position der Kamera wie beschrieben an der Wasseroberfläche gespiegelt. In die Reflection Map wird alles gerendert, was sich anschließend im Wasser spiegeln soll; im Falle von 3DView2 sind das der Himmel, die Objekte und das Terrain. Entscheidend ist, dass nur Pixel gezeichnet werden, die sich tatsächlich über der Wasseroberfläche befinden – Objekte oder Objektteile unterhalb der Wasseroberfläche würden falsche Spiegelbilder verursachen. Für solche Zwecke ist es auf den meisten Graphikkarten möglich, so genannte *User Clip Planes* zu definieren, das sind frei definierbare Schnittebenen, die von der Hardware beim Rendering automatisch berücksichtigt werden. Es wird nun eine solche Schnittebene definiert, die exakt mit der Wasseroberfläche übereinstimmt und somit Geometrie unterhalb der Oberfläche wegschneidet.

Beim späteren Zeichnen der Wasseroberfläche wird die Reflection Map auf die Wasseroberfläche projiziert. Das so genannte *Projective Texture Mapping* wurde erstmals in [SEG+92] vorgestellt. Dabei wird eine Textur wie von einem Filmprojektor aus in die Szene hineinprojiziert. Der Vorgang ist analog zur Funktionsweise einer virtuellen Kamera beim Rendering (siehe auch Kapitel 11): Die Kamera wird durch eine Matrix repräsentiert, die einen Punkt aus dem Weltkoordinatensystem (*World Space*) in den zweidimensionalen Raum des Bildschirms (*Screen Space*) transformiert. Die Kameramatrix wird aus Position, Rotation und Öffnungswinkel der virtuellen Kamera konstruiert. Analog lässt sich eine Matrix für einen „Projektor“ konstruieren, die Punkte aus dem Weltkoordinatensystem in seine Projektionsfläche transformiert. Die resultierenden Koordinaten werden benutzt, um die projizierte Textur zu indizieren. Beim Projizieren der Reflection Map auf die Wasseroberfläche dient die Kamera gleichzeitig als Projektor, d.h. die Kameramatrix wird auch als Projektionsmatrix für die Textur verwendet.

Heutige Graphikkarten bieten spezielle Funktionen für projektive Texturen. [LOM04] zeigt beispielsweise eine Implementation mittels Vertex- und Pixel-Shadern, [EVE01b] erläutert, wie OpenGL Projective Texture Mapping unterstützt.

10.2. Der 3DView2 Wasser-Shader

3DView2 betrachtet die Wasseroberfläche als perfekte Ebene und realisiert Spiegelungen wie im vorherigen Abschnitt beschrieben. Um die Geschwindigkeit zu steigern wird das Terrain im Spiegelbild nicht mit Splatting, sondern nur mit der Basistextur gezeichnet (vgl. Kapitel 9.5.3). Darüber hinaus ist für den Benutzer konfigurierbar, ob und bis zu welcher Entfernung sich Objekte im Wasser spiegeln und ob sich Objektschatten im Wasser spiegeln oder nicht. Der Verzicht auf derartige Details bringt einen Geschwindigkeitsgewinn (zum Thema Konfiguration siehe Kapitel 13).

Ein realistischer Eindruck einer Wasseroberfläche entsteht erst durch die Simulation von Wellengang. Wie zu Anfang dieses Kapitels festgelegt ist für 3DView2 ein leichter Wellengang ausreichend. [VLA02] beispielsweise berechnet im Vertex Shader Cosinus- und Sinusfunktionen, um gleichermaßen die Wassergeometrie und die Reflection Map zu verzerren. 3DView2 verwendet statt dessen *Normal Maps*. Normal Maps sind Texturen, deren Texel nicht Farbinformationen, sondern Normalenvektoren sind. Diese Normalenvektoren können statt der geometrischen Flächennormalen zur Lichtberechnung eingesetzt werden; so werden Details auf einer Oberfläche vorgetäuscht, die in der Geometrie nicht vorhanden sind. In 3DView2 ist die Geometrie der Wasseroberfläche eine perfekte Ebene; Normal Maps täuschen darauf kleine Wellen vor.

Normal Maps lassen sich mit normalen Bildbearbeitungsprogrammen oder auch mit speziell für diesen Zweck entworfener Software generieren. Die Wasseroberfläche im 3DViewer wird von zwei kachelnden Normal Maps überdeckt, die jeweils ein Wellenmuster enthalten. Der Normalenvektor eines Punktes auf der Wasseroberfläche ergibt sich als Mittelwert der aus den beiden Normal Maps ausgelesenen Vektoren an der entsprechenden Stelle. Durch zeitabhängiges Verschieben der Normal Maps in entgegengesetzte Richtungen erscheint das Wellenmuster in Bewegung.

Auf der Wasseroberfläche wird eine spekulare Beleuchtungsberechnung (*Specular Lighting,* siehe z.B. [EBE01]) durchgeführt, diese sorgt für das Glitzern der Wellen. Desweiteren werden die Texturkoordinaten der Reflection Map durch Aufaddieren der Normalenvektoren verzerrt, dadurch verschwimmt das Spiegelbild. Zuletzt wird für jeden Pixel der Wasseroberfläche ein Transparenzwert errechnet, der abnimmt, je flacher der Betrachter auf die Wasseroberfläche schaut. All diese Berechnungen werden im Pixel Shader durchgeführt. Für ältere Graphikkarten, die noch keine Pixel Shader unterstützen, enthält 3DView2 einen vereinfachten Shader, der die Wasseroberfläche ebenfalls spiegelnd, aber ohne Wellengang darstellt.

Abbildung 40: Objekte unter Wasser sind in einem steilen Winkel gut zu erkennen (links), die Wasseroberfläche wird mit flacherem Winkel zunehmend weniger transparent (rechts)

11.Vermeidung von Z-Fighting

Z-Fighting - Definition und Ursachen

Der Begriff *Z-Fighting* bezeichnet einen Darstellungsfehler beim 3D-Rendering, der durch die begrenzte Auflösung bzw. Ungenauigkeit des *Z-Buffers* zustande kommt. Z-Fighting wird gelegentlich auch als *Tearing, Bleeding* oder *Stitching* bezeichnet.

Beim Z-Fighting wird eine eigentlich verdeckte Fläche plötzlich teilweise sichtbar und scheint die sie verdeckende Fläche an einigen Stellen zu durchdringen. Durch das gleiche Phänomen kann eine Schnittkante zweier Flächen, die eigentlich gerade sein sollte, zu einer scharf gezackten Linie werden. In einer bewegten Szene fällt das Problem besonders unangenehm auf, weil die fehlerhaft dargestellten Bildbereiche stark zu flimmern beginnen.

Abbildung 41: Eine Szene mit Z-Fighting (links) und in korrekter Darstellung (rechts). An den Kanten der Insel ist der Effekt besonders gut zu erkennen, aber auch das kleine Wasserloch ganz unten im Bild ist betroffen.

Zunächst sollen die Ursachen dieses Darstellungsfehlers geklärt werden. Die darzustellende Szene besteht aus Dreiecken im dreidimensionalen Raum, die auf den zweidimensionalen Bildschirm projiziert werden. Dabei muss sichergestellt werden, dass die Dreiecke sich in korrekter Weise gegenseitig verdecken. Der naive Lösungsansatz wäre, die Dreiecke von der Kamera aus gesehen von hinten nach vorn zu sortieren und dann in dieser Reihenfolge zu zeichnen. Dieses Verfahren ist als *Painters Algorithm* bekannt; ähnlich wie von einem Maler auf der Leinwand wird zunächst der Hintergrund gezeichnet und dann Stück für Stück mit den Objekten im Vordergrund verdeckt. Leider ist das Verfahren in dieser Form fehlerhaft, denn wenn zwei Dreiecke sich gegenseitig durchdringen, kann keine korrekte Zeichenreihenfolge gefunden werden; beide Dreiecke sind teilweise sichtbar und werden gleichzeitig teilweise vom jeweils anderen Dreieck verdeckt. Lösen lässt sich dieses Problem nur, indem die betroffenen Dreiecke entlang ihrer Schnittkante in mehrere Dreiecke unterteilt werden. Diese Lösung ist extrem aufwändig, da alle Dreiecke der Szene gegeneinander getestet und bei Bedarf unterteilt werden müssen.

Eine bessere und wesentlich elegantere Lösung für das Verdeckungsproblem ist der so genannte Z-Buffer. Die Idee wurde schon 1974 vom Computergraphikpionier Edwin Catmull entwickelt [CAT74]. Das Verfahren ist zwar konzeptionell sehr einfach, aber speicher- und rechenintensiv und war somit für Echtzeitanwendungen erst geeignet, als entsprechende Hardwareunterstützung verfügbar wurde – im Consumer-Bereich war das erst in den 1990er Jahren der Fall. Bis dahin mussten 3D-Engines auf den erwähnten Painters Algorithm zurückgreifen; oftmals wurde aus Geschwindigkeitsgründen sogar auf das Splitten von Dreiecken verzichtet und die auftretenden Artefakte wurden in Kauf genommen.

Ein Z-Buffer speichert pro Bildpixel einen Tiefenwert ab. Beim Zeichnen eines Dreieckes wird für jeden zu zeichnenden Pixel ein Tiefenwert berechnet, dieser ergibt sich durch Interpolation der Eckpunkte des Dreiecks. Dieser Tiefenwert wird mit dem entsprechenden Wert im Z-Buffer verglichen; nur, wenn der neue Pixel einen kleineren Wert aufweist, also näher an der Kamera liegt, wird er gezeichnet und sein Tiefenwert in den Z-Buffer geschrieben. Auf diese Weise realisiert der Z-Buffer eine individuelle Sortierung der Dreiecke pro Pixel.

Vor dem Rendering werden alle Koordinaten aus dem Weltkoordinatensystem (*World Space*) in ein Kamerakoordinatensystem (*View Space*) transformiert. Im View Space liegt die Kamera im Koordinatenursprung und blickt in positive Z-Richtung. Anschließend wird bei der Perspektiventransformation vom View Space in den so genannten *Clipping Space* transformiert. Dabei wird der Sichtkegel der Kamera in den Einheitswürfel $[-1, 1]^3$ transformiert (Abbildung 42). Der Sichtkegel ist genau genommen ein Pyramidenstumpf (*View Frustum*), er wird durch einen Öffnungswinkel und zwei Ebenen definiert – die *Near Clipping Plane* und die *Far Clipping Plane* (oft auch einfach nur *Near Plane* und *Far Plane* genannte). Diese Ebenen sind parallel zur xy-Ebene im *View Space* und können daher einfach einfach durch ihren Abstand vom Koordinatenursprung, *n* und *f*, definiert werden.

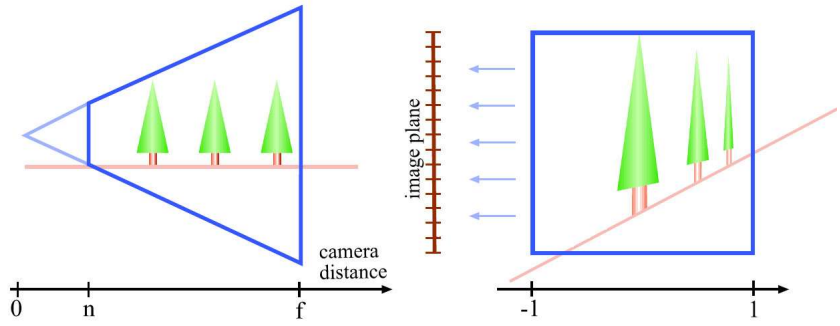


Abbildung 42: Durch die Perspektiventransformation wird der Sichtkegel (links) in einen Einheitswürfel (rechts) projiziert. n und f bezeichnen den Abstand der Near- bzw. Far Clipping Plane von der Kamera. Deutlich zu erkennen: Weiter von der Kamera entfernte Objekte werden verkleinert. (modifizierte Abbildung aus [STA+02])

Alle Punkte, die nach der Perspektiventransformation im Würfel $[-1, 1]^3$ liegen, sind für die Kamera sichtbar, alle Punkte außerhalb sind nicht sichtbar. (Daraus leitet sich der Begriff Clipping Space ab; engl. to clip = etwas abschneiden). Die Near Clipping Plane im View Space wird dabei zur Vorderseite des Würfels im Clipping Space (z=-1) und die Far Clipping Plane zur Rückseite (z=1). Nach der Perspektiventransformation können die x- und y-Komponente der Koordinaten direkt als Bildschirmkoordinaten verwendet werden; es

muss lediglich vom Wertebereich -1..1 auf 0.. Bildbreite bzw. 0 .. Bildhöhe umgerechnet werden. Die resultierenden z-Koordinaten werden nur noch für das Z-Buffering benötigt. Der Z-Buffer muss also Werte im Bereich von -1..1 speichern können, vor dem Rendern der Szene wird er komplett mit -1 initialisiert, damit an jeder Stelle des Bildschirms garantiert ein Pixel gezeichnet wird.

Der Zweck der Perspektiventransformation liegt darin, weiter von der Kamera entfernte Objekte für die Darstellung auf dem Bildschirm zu verkleinern. Dabei werden die z-Koordinaten ungleichmäßig über die Entfernung verteilt, die z-Koordinate im Clipping Space z' wird berechnet als

$$z' = \frac{f+n}{f-n} + \frac{1}{z}\left(\frac{-2 \cdot f \cdot n}{f-n}\right) \quad ,$$

wobei z die z-Koordinate im View Space ist und n und f wie beschrieben der Abstand der Near- bzw. Far Clipping Plane vom Koordinatenursprung im View Space sind. Die Verteilung der z-Werte im Clipping Space ist ungleichmäßig; je weiter eine Strecke von der Kamera entfernt ist, desto stärker wird sie in z-Richtung gestaucht. Abbildung 42 veranschaulicht dieses; die weiter von der Kamera entfernten Tannen werden durch die Perspektiventransformation nicht nur in der Höhe kleiner, sondern auch in Z-Richtung „flacher“.

Wie zu Beginn des Kapitels angedeutet entsteht das Z-Fighting durch die begrenzte Auflösung des Z-Buffers. Je nach Graphikkarte werden die z-Werte als Fließkommazahlen oder Fixkommazahlen mit 8, 16, 24 oder 32 bit gespeichert. Wenn zwei Punkte zweier Dreiecke relativ nah beieinander liegen, kann es aufgrund dieser begrenzten Auflösung vorkommen, dass beide Punkte im Clipping Space die gleiche z-Koordinate haben, die Graphikkarte kann somit nicht mehr entscheiden, welches der beiden Dreiecke an dieser Stelle zu sehen sein sollte. Aufgrund der eben erläuterten Stauchung der z-Koordinaten tritt dieses Problem besonders häufig bei weiter von der Kamera entfernten Objekten, also nahe der Far Clipping Plane, auf. Der Z-Buffer hat somit nahe der Kamera eine wesentlich bessere Präzision als weiter entfernt, was so beabsichtigt ist, aber je nach Situation auch zu Problemen führen kann.

Einfache Vermeidungsstrategien

Es gibt eine Reihe von einfachen Möglichkeiten, den Z-Fighting-Effekt abzumildern oder völlig zu vermeiden:

- **Ändern der Z-Funktion –** Normalerweise wird ein Pixel gezeichnet, wenn sein z-Wert kleiner ist als der bereits im Z-Buffer vorhandene Wert. Gängige Graphikkarten bieten aber die Möglichkeit, auch andere Vergleichsfunktion zu verwenden. So kann der neue Pixel beispielsweise auch genau dann gezeichnet werden, wenn sein z-Wert kleiner oder gleich dem Z-Buffer-Wert ist. Das Ergebnis ist, dass bei scheinbar gleichen z-Werten das zuletzt gezeichnete Dreieck statt des zuerst gezeichneten Dreiecks zu sehen ist. Das kann den optischen Eindruck unter Umständen verbessern, ist aber keine korrekte Lösung.

- **Höhere Z-Buffer Auflösung wählen –** Viele Geräte unterstützen verschiedene Auflösungen des Z-Buffers und lassen dem Programmierer beispielsweise die Wahl zwischen 16, 24 oder 32 bit Präzision. Eine höhere Auflösung verhindert Z-Fighting sehr effektiv, kostet aber auch wertvollen Speicher.

- **Wahl der Near- und Far Plane** – Um die Auflösung des Z-Buffers optimal auszunutzen, sollte die Near Clipping Plane möglichst weit von der Kamera entfernt und die Far Clipping Plane möglichst nah an der Kamera sein. Hier sind aber oft aus praktischen Gründen Grenzen gesetzt, denn die Far Clipping Plane wird durch die gewünschte Sichtweite diktiert und eine zu weit von der Kamera entfernte Near Clipping Plane sorgt dafür, dass Objeke plötzlich aus dem Sichtfeld verschwinden oder buchstäblich „aufgeschnitten“ werden (siehe Abb. 43).

- **Die Geometrie optimieren** – Situationen, in denen zwei Polygone koplanar oder fast koplanar sind, sollten vermieden werden, ebenso sich gegenseitig durchdringende Polygone. Bei teilweise automatisch generierter Geometrie wie z.B. in Terrainrenderern ist das allerdings nur schwer durchzusetzen.

Abbildung 43: Punkte, die vor der Near Clipping Plane liegen, werden nicht dargestellt. Ist sie zu weit von der Kamera entfernt, werden Objekte regelrecht "aufgeschnitten".

Z-Fighting in 3DView2

Die Situation in 3DView2 ist schwierig, weil die klassischen Vermeidungsstrategien nicht greifen. Die Entfernung Far Clipping Plane ist durch die gewünschte Sichtweite festgelegt, sie soll mindestens 1000m betragen. Ein Herausschieben der Near Clipping Plane führt zu inakzeptablen Effekten, da der Betrachter sehr nah an Objekte herangehen kann. Die 3D-Modelle lassen sich bezüglich Z-Fighting optimieren, jedoch wird das Terrain wie beschrieben aus einem Height Field erzeugt, weil das eine einfache Benutzung garantiert. Z-Fighting trat anfangs vor allem zwischen dem Terrain und der Wasseroberfläche auf, meist schon in einer Entfernung von 150-200m auf Graphikkarten mit 16bit Z-Buffer (Abbildung 41). Dies hätte nur vermieden werden können, indem der Strand der Inseln wesentlich steiler gestaltet worden wäre, was aber sowohl inhaltlich als auch vom Arbeitsaufwand her für den Benutzer völlig inakzeptabel ist.

Statt dessen wurde ein anderes Verfahren verwendet, das auch in anderen 3D-Engines, z.B. Nebula2, zum Einsatz kommt. Das Rendering der gesamten Szene wird in zwei Pässe mit unterschiedlichen Einstellungen für die Near- und Far Clipping Plane unterteilt. Es seien n und f die gewünschten Einstellungen für die Near- und Far Plane. Bei 3DView2 waren das 0,1m und 1000m. Eine Zwischenentfernung s mit $n < s < f$ wird nun so gewählt, dass

$$\frac{s}{n} = \frac{f}{s} \quad \text{, also}$$

$$s = \sqrt{f \cdot n} \quad .$$

Dieses Verhältnis sichert eine gleichmäßige Ausnutzung des Z-Buffers. Die gesamte Szene wird nun zwei Mal gerendert. Im ersten Pass wird der hintere Teil der Szene gezeichnet, d.h. die Near Plane liegt bei $n_1 = s$ und die Far Plane bei $f_1 = f$. Anschließend muss der Z-Buffer gelöscht, d.h. erneut komplett mit -1 initialisiert werden. In einem zweiten Pass wird nun der vordere Teil der Szene gezeichnet; dieses Mal ist die Entfernung der Near Plane $n_2 = n$ und die Entfernung der Far Plane $f_2 = s$. Das Löschen des Z-Buffers zwischen den Pässen ist notwendig, da die dort gespeicherten Z-Werte wie erläutert von den Einstellungen für die Clipping Planes abhängen und nach einer Neueinstellung selbiger nicht sinnvoll weiterverwendet werden können. Durch das Zurücksetzen des Z-Buffers überdeckt der zweite Pass das vom ersten gezeichnete Bild an vielen Stellen, wodurch letztendlich ein korrekter Gesamteindruck entsteht (Abbildung 44).

Abbildung 44: Vermeidung von Z-Fighting durch mehrere Renderpässe. Von links nach rechts: nur erster Pass, nur zweiter Pass, zusammengesetztes Gesamtbild.

Der Himmel bildet im ganzen Prozess eine Ausnahme, er wird grundsätzlich nur ein einziges Mal pro Frame vor allen anderen Graphikobjekten gezeichnet. Da er nichts verdecken kann, muss er auch keine Werte in den Z-Buffer schreiben. Zur Himmelsdarstellung wird eine so genannte *Skybox* verwendet, das ist ein innen mit einer passenden Textur versehener Würfel, der sich immer automatisch so positioniert, dass die Kamera sich exakt in seinem Zentrum befindet. Die Skybox beschreibt den Z-Buffer nicht und wird daher immer von allen anderen Objekten verdeckt. Dadurch ist egal, wie groß die Skybox ist; 3DView2 verwendet einen Würfel mit – umgerechnet auf die Weltdarstellung – 10 Metern Kantenlänge. Da er sich mit der Kamera bewegt und immer im Hintergrund ist, erscheint er dem Betrachter unendlich groß.

Durch die Unterteilung der Szene in zwei Pässe kann Z-Fighting effektiv vermieden werden, allerdings ist das Verfahren nicht ohne Nachteile. Für die Technik spricht, dass sie selbst bei hardwarebedingt geringer Auflösung des Z-Buffers Z-Fighting-Artefakte beseitigen kann, ohne Zugeständnisse an die Sichtweite machen zu müssen oder in die Geometrie einzugreifen.

Leider ist es auf Grund von Rundungsfehlern bei der Rasterisierung möglich, dass die von den beiden Pässen erzeugten Bilder nicht exakt zusammenpassen, d.h. im zusammengesetzten Bild fehlen an der Bruchkante eventuell einzelne Pixel. Dieser Effekt kann theoretisch vermieden werden, indem die beiden Pässe sich leicht überlappen, also $n_1 = s - e$ und $f_2 = s + e$ für einen kleinen Wert $e > 0$. Praktisch führt dieses Vorgehen jedoch zu einem Problem, wenn die Szene halbtransparente Objekte – wie etwa eine Wasseroberfläche – enthält. Bei der Überlappung können dann einige halbtransparente Pixel doppelt gezeichnet werden, einmal von jedem Pass. Das Ergebnis ist, dass diese Pixel im fertigen Bild deutlich zu dunkel erscheinen, weil der Betrachter durch zwei halbtransparente Lagen blickt statt, wie eigentlich korrekt, nur durch eine.

Ein weiterer Nachteil ist die Geschwindigkeit:

Erstes Problem: Die Szene pro Bild doppelt zu zeichnen ist selbstverständlich langsamer, als sie nur ein einziges Mal zu zeichnen. Zwar können die meisten Objekte effizient komplett dem ersten oder komplett dem zweiten Pass zugeordnet werden, jedoch gibt es immer Objekte, die die Trennungsebene schneiden und deshalb tatsächlich in beiden Pässen, also doppelt, gezeichnet werden müssen.

Zweites Problem: Viele Pixel des Bildes werden doppelt gezeichnet. Dabei wird von *Overdraw* gesprochen, weil die Graphikkarte Pixel zeichnet, die später gar nicht zu sehen sind, da sie vorher von anderen Pixel überschrieben werden. Die Anzahl der gezeichnetem Pixel kann ein die Geschwindigkeit begrenzender Faktor sein, deshalb sollte die Szene möglichst von vorn nach hinten gezeichnet werden, damit der Z-Buffer das doppelte Zeichnen bei möglichst vielen Pixeln verhindern kann [RIG02]. Die hier beschriebene Multipasstechnik erfordert allerdings das genaue Gegenteil.

Drittes Problem: Das Rendern der Skybox vor allen anderen Objekten erzeugt ebenfalls einen massiven Overdraw. Es gäbe eine Möglichkeit, dieses zu verhindern; es könnte zuerst die gesamte Szene gerendert und anschließend die Skybox nur an den Stellen gezeichnet werden, an denen der Z-Buffer noch den Wert -1 hat, also wo noch keine anderen Pixel gezeichnet worden sind. Da der Z-Buffer zwischen den Pässen zurückgesetzt werden muss, ist diese Optimierung nicht mehr möglich.

3DView2 bietet die beschriebene Multipasstechnik als Option für Graphikkarten an, deren Z-Buffer-Auflösung nicht ausreicht, um Z-Fighting zu vermeiden. (Siehe dazu auch Kapitel 13: Konfigurationsmangement). Ob diese Option benötigt wird, ist auch abhängig von der Sichtweite, die ebenfalls vom Benutzer konfiguriert werden kann.

12.Schattendarstellung

Der Begriff „Schatten“ kommt vom althochdeutschen „scato“ (vgl.: Wikipedia, die freie Enzyklopädie) und bezeichnet einen Raum ohne Licht, der von einem vor der Lichtquelle befindlichen Gegenstand verursacht wird. Umgangssprachlich wird gesagt, der Gegenstand „wirft einen Schatten“. Der Schattenraum selbst ist für Menschen nicht wahrnehmbar; lediglich Objekte oder Teile von Objekten, die sich im Schattenraum befinden, erscheinen dunkler als ihre Umwelt, da sie kein direktes Licht von der Lichtquelle empfangen. Die Form des schattenwerfenden Objektes zeichnet sich als zweidimensionale Projektion auf der Umwelt ab; umgangssprachlich werden diese Projektion als Schatten bezeichnet. Flächen, die nicht von direktem Licht getroffen werden, können trotzdem indirekt beleuchtet sein, wenn Flächen in der Umgebung das Licht reflektieren und streuen. Schattenflächen sind daher selten völlig dunkel.

Wenn mehrere Lichtquellen beteiligt sind, kann eine Unterscheidung in Kernschatten und Halbschatten getroffen werden. Kernschatten erhalten überhaupt kein direktes Licht, Halbschatten erhalten nur von einigen Lichtquellen kein direktes Licht.

Schatten sind für die räumliche Wahrnehmung des Menschen sehr bedeutsam; sie helfen dabei, Objekte räumlich zueinander in Beziehung zu setzen und Größenverhältnisse abzuschätzen. Entsprechend müssen Schatten auch in virtuellen Welten dargestellt werden.

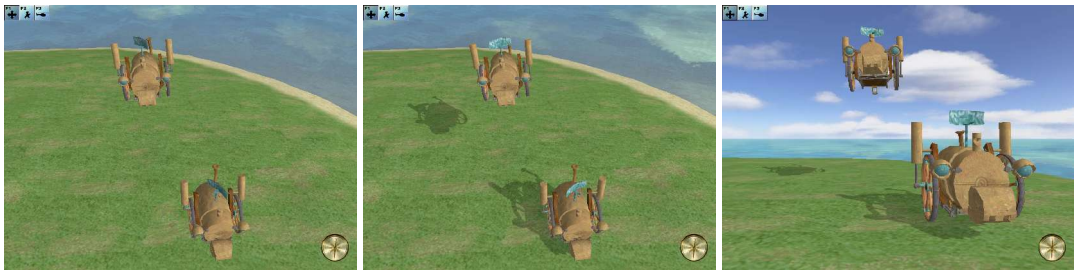


Abbildung 45: Schatten sind für die räumliche Wahrnehmung sehr bedeutsam. Hier eine Szene erst ohne, dann mit Schatten (links, Mitte). Eine andere Perspektive (rechts) verdeutlicht die wahre räumliche Position der Objekte.

Es gibt im Wesentlichen zwei unterschiedliche Ansätze in der dreidimensionalen Computergraphik. Bei Raytracing-Verfahren werden virtuelle Lichtstrahlen, die das Auge des Betrachters treffen, zu ihren Quellen zurückverfolgt. Schatten entstehen bei diesen Verfahren ganz automatisch und nach den gleichen (wenn auch vereinfachten) Gesetzmäßigkeiten wie in der realen Welt. Diese Verfahren sind jedoch trotz ihrer theoretischen Überlegenheit aus Geschwindigkeitsgründen noch nicht für den Einsatz in Echtzeitumgebungen geeignet. Statt dessen setzen die meisten Anwendungen – so auch 3DView2 – auf die Rasterisierung von Polygonen. Schatten entstehen dabei nicht automatisch, sondern müssen mit teilweise sehr aufwendigen Verfahren künstlich in die Szene eingefügt werden. Die gebräuchlichen Algorithmen können grob in drei Gruppen eingeteilt werden ([FAU03]):

- *Limitierte Ansätze* (*limited approaches*) machen sehr stark beschränkende Annahmen über das Anwendungsszenario; beispielsweise dass der Boden eine perfekte Ebene ist und Schatten nur auf den Boden geworfen werden. Diese Verfahren sind in der Regel sehr einfach, jedoch für die meisten Anwendungsfälle, einschließlich der vorliegenden Aufgabenstellung, aufgrund der Beschränkungen nicht geeignet – daher werden sie an dieser Stelle nicht weiter betrachtet.

- *Statische Ansätze* (*static approaches*) gehen von der Annahme aus, dass sowohl die Lichtquellen als auch die Schatten werfenden Objekte unbeweglich sind. Die Beleuchtungsinformation kann somit in einem uneingeschränkt komplexen Vorverarbeitungsschritt mit einem beliebigen anderen Verfahren, einschließlich Raytracing, berechnet und zur späteren Verwendung abgespeichert werden. Ein Beispiel für ein solches Verfahren sind *Lightmaps*. Lightmaps sind Texturen, die vorberechnete Lichtinformationen für eine Oberfläche enthalten und mit der farbgebenden Textur dieser Oberfläche gemischt werden. Für jede Oberfläche der Szene muss eine individuelle Lightmap berechnet und gespeichert werden.

- *Allgemeine Ansätze* (*general approaches*) machen möglichst wenig Annahmen über das Szenario und erlauben insbesondere, dass Schatten werfende Objekte und Lichtquellen sich frei im Raum bewegen. Diese Klasse von Algorithmen ist die komplexeste. Dank der Fortschritte auf dem Hardwaresektor ist es mittlerweile jedoch möglich, solche Verfahren auch im größeren Rahmen anzuwenden. Beispiele für allgemeine Ansätze sind *Shadow Volumes* (insbesondere *Stencil Shadows*) und *Shadow Maps*.

Die Forschung konzentriert sich in letzter Zeit zunehmend auf die allgemeinen Verfahren, da mit einer weiteren Leistungssteigerung bei PCs zu rechnen ist. Zum jetzigen Zeitpunkt verwenden die meisten Anwendungen jedoch eine Kombination aller hier genannter Ansätze, um ein sowohl optisch als auch leistungsmäßig optimales Ergebnis zu erzielen.

12.1.Shadow Volumes

Das Konzept der Schattenerzeugung mittels so genannter *Shadow Volumes* stammt bereits aus dem Jahr 1977 [CRO77], doch erst die Nutzung des *Stencil Buffers* [HEI91] ermöglichte die Nutzung des Verfahrens für Echtzeitschatten; daher wird häufig von *Stencil Shadows* gesprochen.

Shadow Volumes haben viele Vorteile:

- Sie ermöglichen Echtzeitschattenwurf in beliebige Richtungen und auf beliebige Oberflächenformen.
- Objekte können (bei entsprechender Implementation) Schatten auf sich selbst werfen (*Self-Shadowing*).
- Lichtquellen können sowohl Punktlichter, Spot-Lichter als auch direktionale Lichter sein.
- Schattenkanten werden pixelgenau oder sogar subpixelgenau (bei Antialiasing) aufgelöst.
- Eine beliebige Anzahl von Lichtquellen kann gleichzeitig Schatten werfen.

Andererseits gibt es auch eine Reihe ernsthafter Nachteile:

- Shadow Volumes sind, wie noch gezeigt werden wird, teuer in der Berechnung. Auf aktueller Hardware ist in Echtzeit nur eine kleine Anzahl schattenwerfender Objekte und Lichtquellen möglich.
- Die erzeugten Schattenkanten sind immer „hart“, d.h. es gibt eine scharfe Trennlinie zwischen Licht und Schatten. In vielen Fällen wäre jedoch eher eine „weiche“ Schattenkante, also ein Helligkeitsverlauf zwischen Licht und Schatten, wünschenswert.
- Der Schatten wird komplett aus der zugrunde liegenden Geometrie erzeugt, Texturen mit transparenten Teilen können nicht berücksichtigt werden. In der Praxis wird gern auf teilweise transparente Texturen zurückgegriffen, um komplexe Objekte effizient, d.h. mit wenig Geometrie, darstellen zu können. Ein Maschendrahtzaun beispielsweise müsste eigentlich mit tausenden Polygonen modelliert werden; durch die geschickte Verwendung passender Texturen kann das gleiche Objekte mit nur einigen Dutzend Polygonen dargestellt werden. Unter Verwendung von Stencil Shadows werfen solche Objekte aber keine korrekten Schatten.
- Es gibt eine Reihe von Sonderfällen, in denen visuelle Artefakte auftreten könne. Mittlerweile existieren Lösungen für diese Probleme, eine robuste Implementation von Shadow Volumes ist aber in keinem Fall trivial.

Stencil Shadow Rendering

Damit Stencil Shadows eingesetzt werden können, muss eine Reihe von Bedingungen erfüllt sein:

- Die Rasterisierungseinheit muss perfekt geschlossene Formen erzeugen, d.h. wenn zwei Dreiecke zwei gemeinsame Punke haben, dürfen an der gemeinsamen Kante keine Pixel fehlen und keine Pixel doppelt gezeichnet werden. Diese Bedingung sollte heutzutage auf allen Plattformen erfüllt sein.

- Es wird ein *Stencil Buffer* benötigt. Ein Stencil Buffer speichert pro Pixel im fertigen Bild einen Integer-Wert. Diese Werte können als Maske benutzt werden, um Zeichenoperationen nur in bestimmten Teilen des Bildes zuzulassen. D.h. bevor ein neuer Pixel gezeichnet wird, wird der Stencilwert an der entsprechenden Stelle mit einem Referenzwert verglichen, das Ergebnis des Vergleiches entscheidet darüber, ob der Pixel tatsächlich gezeichnet wird. Darüber hinaus kann fest gelegt werden, ob ein tatsächlich gezeichneter Pixel den Stencil Buffer an der entsprechenden Stelle inkrementieren, dekrementieren oder unverändert belassen soll. Stencil Buffer gehören schon seit einigen Jahren zur Standardausstattung von Graphikkarten.

- Es wird ein Z-Buffer benötigt.

- Die Schatten werfende Geometrie muss komplett geschlossen sein, d.h. jede Kante im Mesh muss zu genau zwei Polygonen gehören. Jede beliebige Geometrie kann so angepasst werden, dass diese Bedingung erfüllt ist.

- Die Geometrie darf nur aus planaren Polygonen bestehen. *Curved Surfaces* oder ähnliche Techniken müssten speziell behandelt werden.

Die Grundidee von Shadow Volumes besteht darin, dass Schatten ein Volumen haben. Ein Objekt, das von einer Lichtquelle beleuchtet wird, spannt einen in Bezug auf diese Lichtquelle „lichtleeren“ Raum hinter sich auf. Andere Objekte, die sich in diesem Raum befinden, liegen im Schatten. Der erste Schritt bei der Implementation von Stencil Shadows besteht darin, für diese Schattenvolumina Geometrie zu konstruieren, d.h. es muss ein Mesh konstruiert werden, dessen Begrenzungsflächen exakt denen des Schattenvolumens entsprechen. Pro Objekt müssen so viele Schattenvolumen konstruiert werden wie Lichtquellen in der Szene vorhanden sind. Ändert sich die Position oder Form eines Meshes oder einer Lichtquelle, so müssen alle betroffenen Schattenvolumina neu berechnet werden.

Der erste Schritt bei der Konstruktion des Schattenvolumens ist das Auffinden der Konturkanten; das sind diejenigen Kanten im Mesh, die aus Sicht der Lichtquelle die Umrisse des Objektes bilden. Anders gesagt: Konturkanten bilden die Grenze zwischen von der Lichtquelle beleuchteten (ihr zugewandten) und nicht beleuchteten (abgewandten) Bereichen des Objektes. Mit diesem Ansatz können die Konturkanten leicht ermittelt werden. Für jede Kante wird das Vektorprodukt zwischen der Lichtrichtung und den Flächennormalen der beiden angrenzenden Dreiecke gebildet. Ist das Vektorprodukt einmal größer und einmal kleiner als Null, so handelt es sich um eine Konturkante. Für die Berechnung muss die Lichtposition (bzw. Lichtrichtung, je nach Typ der Lichtquelle) in das Koordinatensystem des Objektes transformiert werden. Der Pseudocode zum Auffinden der Konturkanten nach [TSI04]:

```
LightPosition = LightPosition * Inverse(ObjectWorldMatrix);
for(every polygon)
{
      IncidentLightDir = AveragePolyPos - LightPosition;
      if(DotProduct(IncidentLightDir, PolygonNormal) >= 0.0)
      {
            for(every edge of the polygon)
            {
                  if(edge is already in the contour edge list)
                        remove edge from contour edge list
                  else
                        add edge to the contour edge list
            }
      }
}
```

Um aus den Konturkanten ein Volumen zu erzeugen, wird für jede Konturkante ein Quad aus zwei Polygonen konstruiert. Dabei sind die vier Eckpunkte des Quads die beiden Endpunkte der Kante und die gleichen beiden Endpunkte, verschoben entlang der Lichtrichtung um einen großen Betrag. Was ein „großer Betrag“ ist, hängt von der Anwendung ab. Theoretisch sollte das Schattenvolumen unendlich groß sein, praktisch ist aber die Größe der 3D-Szene oder zumindest die Sichtweite begrenzt, so dass für die Größe des Schattenvolumens eine sinnvolle Begrenzung gefunden werden kann.

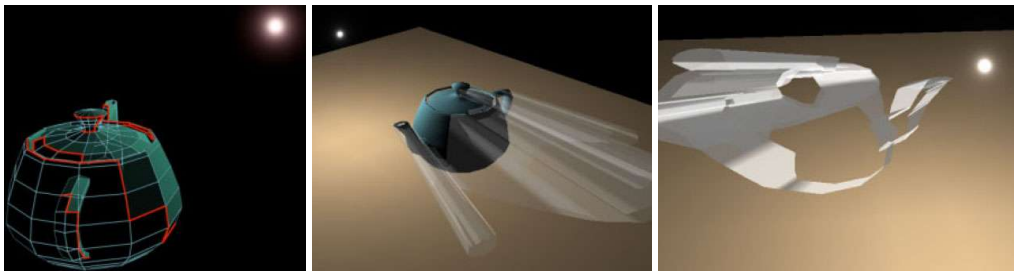


Abbildung 46: Konturkanten (links), und resultierendes Schattenvolumen (Mitte, rechts). Auf dem rechten Bild ist das Objekt nicht dargestellt, so dass die Löcher im Schattenvolumen erkennbar sind. (Quelle: [TSI04])

Das so entstandene Schattenvolumen ist nicht geschlossen, sondern weist vorn und hinten Löcher in der Form des Ursprungsobjektes auf. Abhängig vom weiteren Vorgehen kann es notwendig sein, dieses Loch zu schließen; dabei wird vom *Capping* des Schattenvolumens gesprochen. Dazu werden als vorderes Capping alle der Lichtquelle zugewandten Dreiecke des Objektes dem Schattenvolumen hinzugefügt. Das hintere

Capping entsteht aus den übrigen, der Lichtquelle abgewandten Polygonen, sie müssen punktweise entlang der Lichtrichtung um den bereits verwendeten „großen Betrag“ verschoben werden.

Der zweite Schritt ist das eigentliche Rendering der Schatten mit Hilfe der Schattenvolumina. Für jeden Pixel der Szene muss ermittelt werden, ob er innerhalb eines Schattenvolumens liegt oder nicht. Gegeben sei ein Strahl, der von der Kamera aus ein Objekt in der Szene trifft. Auf seinem Weg durchquert der Strahl möglicherweise ein oder mehrere Schattenvolumen. Es kann beobachtet werden, dass der Strahl in ein Schattenvolumen immer durch ein Frontface dieses Volumens eindringt und es durch ein Backface wieder verlässt. Ein Schattenvolumen kann teilweise oder vollständig innerhalb eines anderen Schattenvolumens liegen, so dass der Strahl auch zwei oder mehr Frontfaces unmittelbar nacheinander durchqueren kann. Um herauszufinden, ob ein Punkt im Schatten liegt, muss lediglich die Anzahl der Front- und Backfaces auf der Geraden zwischen diesem Punkt und der Kamera ermittelt werden. Jedes Frontface zählt +1, jedes Backface -1. Ist die Summe am Ende größer als Null, so liegt der Punkt im Schatten (Abbildung 47).

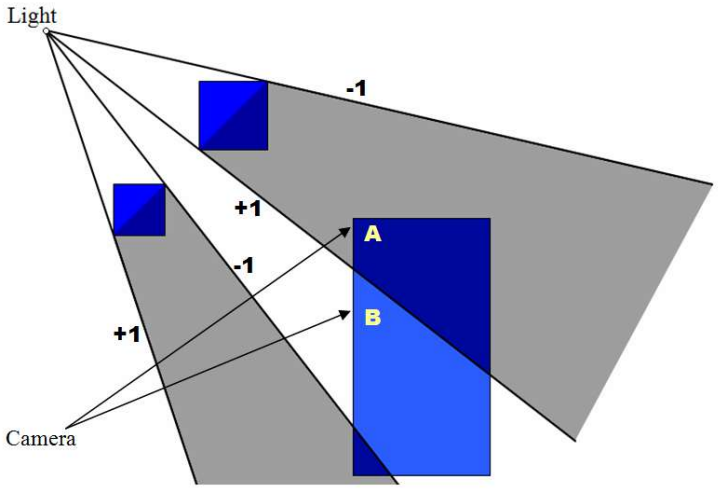


Abbildung 47: Szene mit Schattenvolumina (grau und dunkelblau). Punkt A hat eine Summe von 1 und liegt daher im Schatten, Punkt B hat eine Summe von 0 und wird daher von der Lichtquelle beleuchtet.

Dieses Zählen kann effizient mit Hilfe des Stencil Buffers implementiert werden. Der vollständige Algorithmus zum Rendern der Szene lautet:

1. Frame Buffer, Z-Buffer und Stencil Buffer löschen.

2. Gesamte Szene rendern, wobei nur ambientes Licht und Lichtquellen, die keine Schatten werfen sollen, berücksichtigt werden. Wichtigster Zweck dieses Schrittes ist den Z-Buffer zu füllen, so dass im nächsten Schritt nur die Teile der Schattenvolumina gerendert werden, die für den Betrachter nicht verdeckt sind. Daher darf dieser Schritt nicht ausgelassen werden; selbst wenn es kein ambientes Licht gibt, wodurch die Schatten völlig schwarz würden.

3. Schattenvolumen für eine Lichtquelle zeichnen. Der Z-Buffer wird benutzt, um nur nicht verdeckte Teile der Schattenvolumina zu zeichnen; er wird in diesem Schritt aber nicht beschrieben. Das Rendern der Schattenvolumen geschieht in zwei Schritten:

   (a) Alle Frontfaces der Schattenvolumina werden gezeichnet, d.h. normales Backface-Culling. Der Stencil Buffer wird beim Zeichnen eines Pixels jeweils inkrementiert.

   (b) Alle Backfaces der Schattenvolumina werden gezeichnet, d.h. invertiertes Backface-Culling. Der Stencil Buffer wird beim Zeichnen eines Pixels jeweils dekrementiert.

4. Gesamte Szene ein zweites Mal rendern, diesmal mit der Schatten werfenden Lichtquelle und nur an den Stellen, wo der Stencil-Buffer den Wert 0 hat.

In Schritt 2 wird die Szene ohne direkte Beleuchtung gerendert, in Schritt 4 werden Punkte, die nicht im Schatten liegen, mit direkter Beleuchtung ein zweites Mal gerendert, d.h. überschrieben. Punkte im Schatten bleiben durch die im Stencil Buffer abgelegte Maske so, wie sie in Schritt 2 geschrieben wurden.

Wenn die Szene mehrere Schatten werfende Lichtquellen enthält, müssen die Punkte 3 und 4 für jede Lichtquelle wiederholt werden. Das Rendern der Szene in Punkt 4 muss dann additiv erfolgen, d.h. die neuen Farbwerte werden mit den bereits im Frame Buffer vorhandenen addiert. Dabei muss darauf geachtet werden, dass der Wertebereich von 0 bis 1 pro Farbkanal eingehalten wird.

Anhand des Verfahrens wird klar, warum Stencil Shadows teuer sind. Die gesamte Szene wird bei n Lichtquellen pro Frame $n+1$ Mal komplett gerendert, jedes Schattenvolumen $2n$ Mal gerendert. Dazu kommen die Kosten, die Schattenvolumen bei jeder signifikanten Änderung neu zu berechnen. Eine häufige Optimierung ist daher, zur Berechnung des Schattenvolumens statt des Originalmeshs eine geometrisch reduzierte Variante zu verwenden. Dieses Vorgehen kann auch Nachteile haben, denn wie noch gezeigt werden wird, führen niedrig aufgelöste Schattenmeshes unter Umständen zu Darstellungsfehlern.

Auf neuerer Hardware ist es möglich, jedes Schattenvolumen nur einmal statt zweimal zu zeichnen, da getrennte Stenciloperationen für Front- und Backfaces definiert werden können [EVE+02]. Die Leistungsverbesserung für die GPU ist nur marginal, jedoch wird die Arbeitslast für den Graphiktreiber auf der CPU reduziert.

Probleme mit Stencil Shadows

Der bisher beschriebene Algorithmus leidet unter einer Reihe von ernsthaften Problemen. Die erste Schwierigkeit ergibt sich, sobald sich die Kamera innerhalb eines Schattenvolumens befindet. Bei den bisherigen Überlegungen wurde davon ausgegangen, dass ein von der Kamera ausgesandter Strahl beim Durchqueren des Frontfaces eines Schattenvolumens das Volumen betritt und es durch ein Backface wieder verlässt. Befindet sich die Kamera jedoch innerhalb eines solchen Volumens, ist die erste Fläche, die getroffen wird, ein Backface. Im Stencil Buffer passiert ein Unterlauf, wodurch in der Szene Licht und Schatten vertauscht werden. Der Unterlauf des Stencil Buffers kann verhindert werden, indem der Wert nach unten auf 0 begrenzt wird, dadurch verschwinden allerdings die meisten Schatten aus der Szene.

Eine offensichtliche Lösung für das Problem wäre, zu testen, ob sich die Kamera innerhalb eines Schattenvolumens befindet. Ist das der Fall, könnte z.B. der Stencil Buffer

mit 1 statt 0 initialisiert werden. Dieser Test ist jedoch aufwändig und daher kaum praktikabel.

Eine andere Lösung für das Problem wäre, das Schattenvolumen an der Near Clipping Plane zu schneiden und die Öffnung mit Polygonen zu schließen. Dieses Vorgehen ist aber ebenfalls sehr aufwändig, zumindest, wenn eine hundertprozentig robuste Lösung anstrebt wird.

Es gibt ein zweites, verwandtes Problem: Wenn die Near Clipping Plane einen Abstand $n > 0$ von der Kamera hat, können Schattenvolumina genau wie alle Objekte regelrecht aufgeschnitten werden (siehe Kapitel 11, Abbildung 43); für die Far Clipping Plane gilt das selbe. Auch durch dieses Aufschneiden kommt es in Teilen des Bildes zu den eben beschriebenen Problemen mit der Schattendarstellung. Eine mögliches Vorgehen wäre wieder, das Schattenvolumen an den beiden Clipping Planes künstlich zu verschließen [DIE96]. Dieses Lösung ist aber kaum praktikabel, Carmack bezeichnete sie als „fragil“ [CAR00], da viele Spezialfälle den Algorithmus verkomplizieren.

Das Clipping an der Far Plane kann vermieden werden, denn es ist möglich, die Projektionsmatrix so zu gestalten, dass die Far Plane sich rechnerisch in der Unendlichkeit befindet, d.h. es findet kein Clipping an der Far Plane mehr statt [EVE+02]. Die Präzision des Z-Buffers wird dadurch nur marginal beeinträchtigt.

Carmacks Reverse

Eine sehr elegante Lösung für das Clipping an der Near Plane fand 2000 John Carmack [CAR00]. Er entdeckte, dass es genügt, den Z-Test umzukehren – daher wurde diese Idee auch unter dem Namen *Carmacks Reverse* bekannt. Allerdings hat er sie nie offiziell publiziert, sondern sie nur auf einer privaten Mailingliste diskutiert.

Carmacks Reverse funktioniert exakt wie das bisher beschriebene Stencil-Shadow-Verfahren, nur, dass der Stencil Buffer inkrementiert bzw. dekrementiert wird, wenn ein Pixel den Z-Test nicht besteht – statt wie bisher, wenn er ihn besteht. Daher wird die Technik auch als *Depth Fail* bzw. *Z-Fail* und die bisherige Vorgehensweise als *Depth Pass* bzw. *Z-Pass* bezeichnet. Dabei muss lediglich eine Randbedingung erfüllt sein – die Schattenvolumina müssen an beiden Enden geschlossen werden, was für die Depth-Pass-Technik nicht notwendig ist.

Depth Fail ist die weniger intuitive Technik. Der wesentlichen Unterschied besteht darin, dass Depth Pass die Front- und Backfaces der Schattenvolumina zwischen einem sichtbaren Punkt (Pixel) in der Szene und der Near Clipping Plane zählt; Depth Fail zählt statt dessen in umgekehrter Richtung die Front- und Backfaces der Schattenvolumina zwischen diesem Punkt und der Unendlichkeit. Da die Schattenvolumina nicht wirklich unendlich groß und an den Enden verschlossen sind, kann davon ausgegangen werden, dass die „Unendlichkeit“ immer außerhalb aller Schattenvolumina liegt. Damit ist nicht nur das Problem des Clippings an der Near Plane gelöst, sondern es entfallen auch die Probleme, die bisher entstanden sind, wenn die Kamera sich innerhalb eines Schattenvolumens befindet.

Depth Pass und Depth Fail lassen sich kombinieren. Die Depth-Pass-Technik ist schneller, da sie ohne das Capping der Schattenvolumina auskommt. Daher ist es sinnvoll, Depth Fail nur dann einzusetzen, wenn Schattenvolumina sich sehr nah an der Near Clipping Plane befinden, so dass ein Schneiden der Plane zu befürchten ist oder die Möglichkeit besteht, dass die Kamera sich innerhalb eines dieser Volumina befindet. Ist das nicht der Fall, wird Depth Pass eingesetzt.

Darstellungsartefakte

Einige Darstellungsfehler sind im Verfahren begründet und können daher nicht generell vermieden werden. Ein Problem von Shadow Volumes ist, dass die Schattenkante auf dem Schatten werfenden Objekt selbst nur entlang der Konturkanten verlaufen kann. Physikalisch gesehen ist das sogar korrekt – in der Praxis wird jedoch bei der Darstellung gerundeter Oberflächen eine Technik verwendet, der hier zum Problem wird. Die gerundete Oberfläche wird mit Dreiecken approximiert und die Lichtberechnung geschieht in einer Weise, welche die Oberfläche rund erscheinen lässt (z.B. *Geround Shading* oder *Phong Shading*). Abbildung 48 zeigt einen typischen Problemfall – der beleuchtete Teil der Kugel erscheint dank geschickter Lichtberechnung rund. Die gezackte Schattenkante offenbart jedoch, dass die Kugel tatsächlich aus verhältnismäßig wenigen Dreiecken zusammengesetzt ist. Dieses Artefakt kann nicht völlig vermieden, sondern nur abgemildert werden, indem die Geometrie der Kugel höher aufgelöst wird; das wiederum wirkt sich negativ auf die Darstellungsgeschwindigkeit aus.

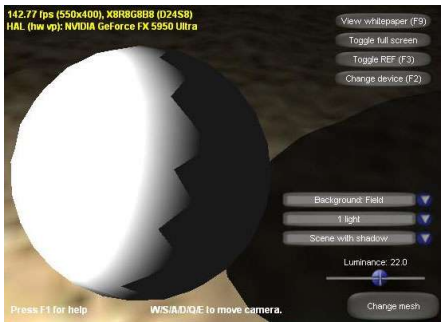


Abbildung 48: Zu niedrig aufgelöste Geometrie führt zu unansehnlichen Schattenkanten. (Quelle: [MS04a])

Ein weiteres mögliches Problem liegt darin begründet, dass die Schattenvolumen unendlich groß (oder in der Praxis zumindest sehr groß) sind. Die resultierende Darstellung ist nur dann korrekt, wenn ausnahmslos die gesamte Geometrie der Szene Schatten mittels Stencil Shadows wirft. In den meisten Anwendungen ist das aber schon aus Geschwindigkeitsgründen unmöglich zu realisieren. Meist werden Schattenvolumen nur für bewegliche Objekte eingesetzt, während statische Geometrie wie Boden und Wände mittels statischer Schattenverfahren, z.B. Lightmaps, behandelt werden. Dadurch können an Wänden und Böden falsche Schattenbilder sichtbar sein. Abbildung 49 zeigt einen typischen Fall. Ein Beobachter, der zwischen den beiden in hellblau eingezeichneten Wänden steht, sieht an beiden Wänden einen Schatten des Quaders, obwohl er den Quader selbst nicht sehen kann. Diese falschen Schattenbilder wären nicht sichtbar, wenn die Wände ebenfalls Stencil Shadows werfen würden.

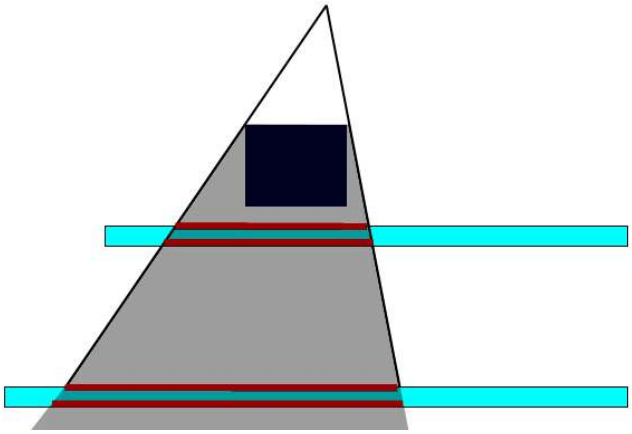


Abbildung 49: Das sein Schattenvolumen unendlich groß ist, wirft der Quader vier Schatten (rot). Die falschen Schatten wären nicht sichtbar, wenn die Wände (hellblau) ebenfalls Schatten werfen würden.

Shadow Volumes im Shader

Die Einführung von *Vertex Shadern* hat die Tür für eine möglicherweise effizientere Implementation von Stencil Shadows geöffnet [MS04a]. Bisher musste die Erzeugung der Schattenvolumen auf der CPU erfolgen und das Resultat musste bei jeder Neuberechnung jeweils komplett über den BUS auf die Graphikkarte kopiert werden. Mit Hilfe eines Vertex Shaders können Schattenvolumen komplett auf der GPU berechnet werden. Das entlastet CPU und BUS.

Vertex Shader bis einschließlich Version 3.0 können nur Geometrie transformieren, aber keine neuen Vertices oder Dreiecke erzeugen oder existierende entfernen. Zur Erzeugung von Schattenvolumen ist es jedoch erforderlich, an den Konturkanten neue Dreiecke einzufügen, um das Mesh anschließend entlang der Lichtrichtung extrudieren zu können. Da der Vertex Shader das nicht leisten kann, muss dieser Schritt auf der CPU durchgeführt werden. Welche Kanten des Meshes Konturkanten sind, hängt von Licht- und Objektposition ab und kann sich von Frame zu Frame ändern. Da das Schattenmesh nicht in jedem Frame auf der CPU neu berechnet und keine Annahme über die Position der Lichtquelle gemacht werden sollen, werden an allen Kanten des Objektes Quads eingefügt. Die so eingefügten Dreiecke sind normalerweise degeneriert, d.h. haben eine Fläche von Null, da je zwei Eckpunkte zusammenfallen. Es ist sinnvoll, ein vom Originalmesh getrenntes Schattenmesh zu erzeugen. Das Schattenmesh benötigt lediglich Positions- und Normalenvektoren, aber z.B. keine Texturkoordinaten oder Farbinformationen. Der Vertex Shader extrudiert das Schattenmesh dann anhand der Lichtrichtung, wodurch diejenigen degenerierten Quads, die an Konturkanten liegen, zu echten Begrenzungsflächen des Schattenvolumens werden.

Das Erzeugen des Schattenvolumens auf der CPU geschieht folgendermaßen [MS04a]: Es wird über die Dreiecke des Originalmeshes iteriert; für jedes Dreieck erfolgen drei Schritte:

- Es werden drei neue Vertices und ein neues Dreieck für das Schattenmesh erzeugt. Im Originalmesh teilen sich höchstwahrscheinlich benachbarte Dreiecke die Vertices entlang ihrer gemeinsamen Kante. Da das Schattenmesh extrudiert wird, muss hier jedes Dreieck drei eigene Vertices besitzen.

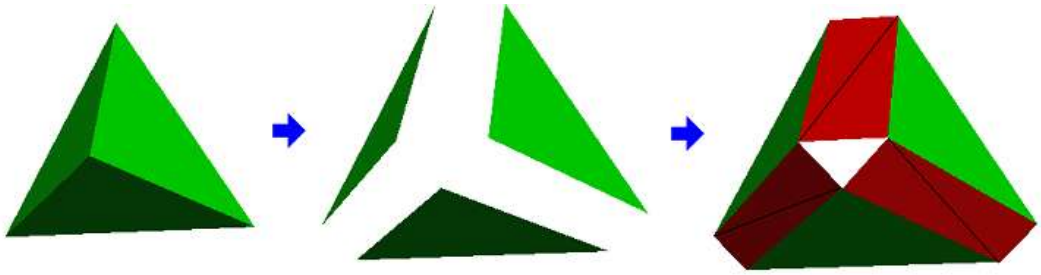


Abbildung 50: An allen Kanten des Meshes (grün) werden Quads (rot) eingefügt. Die Vertices an den Kanten können daher nicht mehr geteilt werden, jedes Dreieck muss drei eigene Vertices besitzen. (Quelle: [MS04a])

- Die Normalen der Vertices werden auf die Flächennormale ihres Dreieckes gesetzt. Ein Vertex Shader bis Version 3.0 verarbeitet nur einzelne Vertices und hat keine Informationen über die zugehörige Fläche. Aufgrund der neu berechneten Normale kann er pro Vertex entscheiden, ob er zu einer der Lichtquelle zugewandten oder abgewandten Fläche gehört.

- Es wird ein Eintrag in einer Kantenzuordnungstabelle erzeugt. Die Kantenzuordnungstabelle enthält eine Zuordnung von Kanten des Originalmeshes zu neu erzeugten Kanten des Schattenmeshes. Für jede Kante des neu erzeugten Dreiecks wird überprüft, ob bereits ein Eintrag in der Tabelle existiert. Ist das nicht der Fall, wird ein entsprechender Eintrag angelegt. Existiert bereits ein Eintrag, so ist das zweite zu dieser Kante gehörende Dreieck bereits bearbeitet worden. In diesem Fall liegen nun vier Eckpunkte vor, so dass ein Quad erzeugt und dem Schattenmesh hinzugefügt werden kann. Der Eintrag in der Kantenzuordnungstabelle wird gelöscht.

Am Ende sollte die Kantenzuordnungstabelle wieder leer sein. Ist das nicht der Fall, so ist das Originalmesh nicht geschlossen, was die ursprüngliche Forderung verletzt. Als Resultat ist das Schattenmesh ebenfalls nicht geschlossen, was – wie schon beschrieben – zu fehlerhafter Schattendarstellung führen würde. Das Schattenmesh kann automatisch geschlossen werden, indem jeweils zwei verbleibende Kanten gesucht werden, die sich einen Vertex teilen, und ein neues Dreieck über ihnen aufgespannt wird. Dieses wird wiederholt, bis die Kantenzuordnungstabelle leer ist. Das so erzeugte Schattenvolumen entspricht aber nicht notwendigerweise genau der Form des Originalmeshes.

Das Rendering geschieht so wie bisher beschrieben. Das Schattenvolumen wird mit einem speziellen Vertex Shader gezeichnet, der für jeden Vertex aufgrund seiner Normale entscheidet, ob er zu einem dem Licht zugewandten oder abgewandten Dreieck gehört. Letztere Vertices werden entlang der Lichtrichtung verschoben, wodurch das korrekte Schattenvolumen entsteht.

Shadow Volumes per Vertex Shader können, aber müssen nicht notwendigerweise effizienter sein als die klassische Implementation auf der CPU. Zwar wird die Hauptlast an die GPU abgegeben und das Kopieren über den BUS entfällt, andererseits haben die Schattenvolumen typischerweise bis zu drei Mal mehr Vertices als die Originalmeshes, was die Geschwindigkeit wieder reduziert. In der Praxis sollte experimentell bestimmt werden, welcher Ansatz sich in einer gegebenen Situation als vorteilhafter erweist.

12.2.Shadow Maps

Das zweite momentan populäre Verfahren zur Echtzeitschattendarstellung sind *Shadow Maps* [WIL78]. Sie wurden nur ein Jahr nach den Shadow Volumes vorgestellt und waren zu dieser Zeit ebenfalls nicht für Echtzeitanwendungen geeignet; erst die Einführung spezieller Hardware änderte dieses. Shadow Maps wurden zuvor beispielsweise vom Animationsfilmspezialisten Pixar in Filmproduktionen wie „Toy Story“ eingesetzt [EVE01].

Die Vorteile von Shadow Maps sind:

- Schattenwurf auf beliebige Oberflächen ist möglich.
- Self Shadowing, also Schattenwurf eines Objektes auf sich selbst, ist möglich.
- Mehrere Schatten werfende Lichtquellen sind möglich.
- Teilweise transparente Texturen können berücksichtigt werden.
- Weiche Schattenkanten sind möglich.

Auf der anderen Seite bestehen folgende Nachteile:

- Lichtquellen können nur direktionale oder Spot-Lichter sein; Punktlichtquellen sind zwar theoretisch möglich, aber wesentlich aufwändiger.
- Da es sich um ein bildbasiertes Verfahren handelt, können Schattenkanten grob („pixelig“) aufgelöst sein
- Shadow Maps verwenden Z-Buffer, daher treten die als Z-Fighting bekannten Genauigkeitsprobleme auf und können zu flimmernden Schatten führen

Im direkten Vergleich zu Stencil Shadows kann festgehalten werden, dass Shadow Maps in der Regel deutlich effizienter sind [MS04b], insbesondere wenn die Szene eine größere Zahl Schatten werfender Objekte enthält. Shadow Maps benötigen im Gegensatz zu Shadow Volumes keine zusätzliche Geometrie und stellen auch keine speziellen Anforderungen an die Objektgeometrie, selbst teilweise transparente Texturen werden korrekt behandelt. Ihr größter Nachteil ist die schlechte Auflösung der Schattenkanten und Artefakte durch Z-Fighting; Stencil Shadows hingegen liefern immer perfekte Schattenkanten. Die Vermeidung dieser Probleme ist ein Gegenstand aktueller Forschung. Die Schattenkanten auf dem Schatten werfenden Objekt selbst, die bei Shadow Volumes ein Problem darstellten, werden hingegen korrekt gehandhabt. Die Hardwarevoraussetzungen für Shadow Maps sind höher als die für Stencil Shadows. Insbesondere, wenn mehrere Lichtquellen gleichzeitig Schatten werfen sollen, ist die Anwendbarkeit von Shadow Maps durch die maximale Anzahl gleichzeitig benutzbarer Texturen eingeschränkt. Im Gegensatz zu Stencil Shadows sind Punktlichtquellen mit Shadow Maps nur auf sehr ineffiziente Weise realisierbar.

Basisalgorithmus

Shadow Maps beruhen auf folgender Überlegung: Ein Beobachter, der sich exakt an der Position der Lichtquelle befindet, würde keine von dieser Lichtquelle geworfenen Schatten sehen. Schatten sind nur an Stellen vorhanden, die keine direkte Sichtlinie zur Lichtquelle haben. Jeder von der Lichtquelle ausgesandte Lichtstrahl trifft nach einer bestimmten Entfernung auf ein blockierendes Objekt. Alle Punkte, die hinter diesem Schnittpunkt auf dem Strahl liegen, liegen im Schatten. Dieser Umstand kann mit Hilfe von Z-Buffern (vgl. Kapitel 11) ausgenutzt werden.

Es wird zunächst davon ausgegangen, dass die Lichtquelle ein Spot-Licht ist – andere Arten von Lichtquellen werden später in diesem Kapitel diskutiert. Eine Kamera wird so positioniert, dass ihr Sichtkegel exakt dem Kegel der Lichtquelle entspricht. Mit dieser Kamera wird ein Bild gerendert. Der Z-Buffer dieses Bildes enthält nun für jeden Punkt die Entfernung zur Lichtquelle und wird daher als *Depth Map* oder entsprechend seiner Verwendung auch als *Shadow Map* bezeichnet. Genaugenommen ist nur dieses Tiefenbild relevant und nicht das herkömmliche Rendering der Szene.

Im zweiten Schritt wird die Szene mittels der normalen Kamera gerendert. Mit Hilfe der Shadow Map kann für jeden Pixel dieses neuen Bildes ermittelt werden, ob er sich im Schatten befindet oder nicht. Dazu müssen

1. die Entfernung des Pixels (d.h. des dem Pixel entsprechenden Punktes in der Szene) zur Lichtquelle berechnet werden,

2. der korrespondierende Pixel in der Shadow Map ermittelt und

3. beide Entfernungen verglichen werden. Ist die Entfernung zur Lichtquelle größer als der in der Shadow Map gespeicherte Wert, so liegt der Punkt im Schatten; andernfalls ist er beleuchtet.

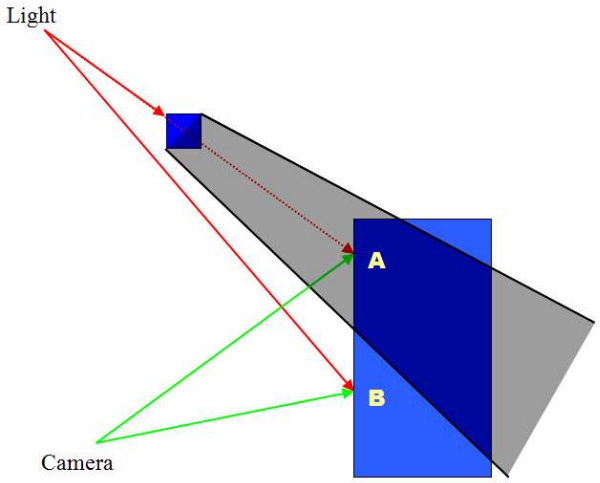


Abbildung 51: Punkt A liegt im Schatten, da die Shadow Map zeigt, dass die Lichtquelle an dieser Stelle ein anderes, näher gelegenes Objekt "sieht".

Abbildung 51 verdeutlicht das Prinzip noch einmal. Für Punkt A ist der berechnete Abstand zur Lichtquelle (rot gestrichelter Pfeil) kleiner als der in der Shadow Map gespeicherte Abstand zum nächsten blockenden Objekt (kurzer roter Pfeil). Punkt A muss daher im Schatten liegen. An Punkt B hingegen ist der berechnete Abstand zur Lichtquelle identisch mit dem Wert aus der Shadow Map. Der Punkt ist daher beleuchtet.

Das Verfahren wird typischerweise mittels Pixel- und Vertex-Shadern implementiert [MS04b], obwohl einige Graphikkarten bestimmter Hersteller auch andere, shaderunabhängige Implementationen beherrschen [EVE+01]. Da Shader sich als allgemeinste Lösung durchgesetzt haben, wird hier vorrangig dieser Ansatz beschrieben.

Es gibt zwei Möglichkeiten die Shadow Map abzuspeichern. Einige Graphikkarten unterstützen spezielle Floating-Point-Texturformate, bei denen jeder Texel durch eine Fließkommazahl dargestellt wird. Diese Lösung ist ideal, alternativ ist es aber auch möglich, ein herkömmliches RGBA-Texturformat zu verwenden und die Tiefenwerte darin zu kodieren. Bei einigen Karten ist es sogar so, dass angebliche Fließkommatexturformate tatsächlich vom Treiber mit Hilfe herkömmlicher Texturformate realisiert werden.

Zur Erzeugung der Shadow Map sind ein spezieller Vertex Shader und ein dazugehöriger Pixel Shader notwendig. Die gesamte Szene muss mit diesen Shadern gerendert werden. In der Praxis enthält eine Szene meist eine Vielzahl von verschiedenen Shadern; beispielsweise verwenden animierte Objekte andere Vertex Shader als nicht animierte. Daher kann es unter Umständen notwendig sein, von vielen oder allen diesen Shadern jeweils eine zweite Version zum Erstellen der Shadow Map zu schreiben.

Der Vertex Shader transformiert die Vertices aus dem World Space zunächst in den Clipping Space der Lichtquelle, d.h. in den Einheitswürfel $[-1, 1]^3$. Die x- und y-Koordinaten der resultierenden Punkte entsprechen ihren Koordinaten im Bild, während z/w die Tiefe der Punkte angibt. Diese Tiefeninformation wird jedoch pro Pixel, nicht nur pro Vertex benötigt. Daher wird das Koordinatenpaar z/w als Texturkoordinatenpaar im Vertex abgespeichert. Texturkoordinaten sind zur Zeit die einzige Möglichkeit, diese Art von Daten an den Pixel Shader durchzureichen. Texturkoordinaten werden automatisch interpoliert, d.h. für jeden Pixel erhält der Pixel Shader z/w-Werte, die sich aus der gewichteten Interpolation der Eckpunkte seines Dreieckes ergeben; somit liegt im Pixel Shader für jeden Pixel der Shadow Map ein korrekter Tiefenwert vor, der nun in die Textur geschrieben werden kann.

Im zweiten Schritt wird die Szene mittels der normalen Kamera gerendert. Wieder werden für alle Objekte, auf die Schatten geworfen werden können, spezielle Vertex- und Pixel Shader benötigt. Als zusätzliches Eingabedatum benötigt der Vertex Shader die kombinierte View- und Projektionsmatrix der Lichtquelle, wie sie im letzten Schritt verwendet wurde. Mit Hilfe dieser Matrix kann jeder Vertex im World Space, den die Kamera sieht, in das Koordinatensystem der Lichtquelle transformiert werden. Dadurch ergeben sich einerseits die Koordinaten dieses Punktes in der Shadow Map (xy) und andererseits die Entfernung des Punktes von der Lichtquelle (z/w). Dieses Vorgehen entspricht dem *Projective Texture Mapping* [SEG+92]; die Shadow Map wird gleichsam von der Lichtquelle aus auf die Szene projiziert. Die xy-Koordinaten des Punktes sind genaugenommen noch keine Texturkoordinaten, da sie den Wertebereich [-1, 1] statt [0, 1] aufweisen; die Umwandlung ist jedoch trivial. Diese Koordinaten werden wieder als Texturkoordinaten im Vertex gespeichert, so dass der Pixel Shader pro Pixel korrekt interpolierte Werte erhält. Der Pixel Shader kann nun den Tiefenwert des aktuellen Pixels mit dem Wert aus der Shadow Map vergleichen und so entscheiden, ob der Pixel im Schatten liegt oder nicht.

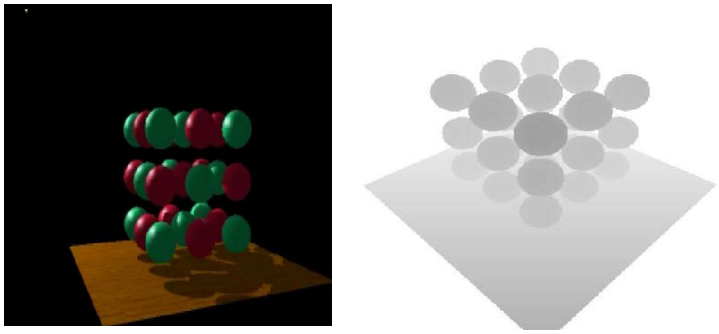


Abbildung 52: Eine Szene mit Schatten (links) und die dazugehörige Shadow Map (rechts). Die Shadow Map ist hier in Graustufen dargestellt, dabei sind hellere Pixel weiter von der Lichtquelle entfernte Punkte. (Quelle: [EVE01])

Bisher zu diesem Abschnitt wurde davon ausgegangen, dass die Lichtquelle ein Spot-Licht ist. Das beschriebene Verfahren kann auch für direktionale Lichtquellen anwendet werden; der einzige Unterschied besteht darin, dass die Transformationsmatrix der Lichtquelle orthographisch statt projektiv wird. Im Gegensatz zu Spot-Lichtern haben direktionale Lichtquellen keine eindeutige Position im Raum. Es ist daher möglich, die Shadow Map von einer beliebigen Position im Raum aus zu rendern. Diese Position sollte so gewählt werden, dass sie die sichtbare Szene genau erfasst; dadurch wird die Auflösung der Shadow Map optimal ausgenutzt.

Punktlichtquellen stellen für Shadow Maps ein Problem dar. Die einzige Möglichkeit, eine Punktlichtquelle zu realisieren, liegt darin, sie durch sechs Spot-Lichter mit einem Öffnungswinkel von je 90 Grad zu emulieren. Das Ergebnis sind sechs Shadow Maps, die sich auf den meisten Graphikkarten am besten gemeinsam in einer *Cube Map* unterbringen lassen; das ist eine Textur, die in Wahrheit aus sechs Texturen besteht, die den Flächen eines Würfels entsprechen. Nichtsdestotrotz ist diese Lösung vergleichsweise so aufwändig, dass davon in den meisten Fällen abgeraten werden muss.

Genauigkeitsprobleme

Shadow Maps bergen einige Schwierigkeiten. Das Verfahren geht gewissermaßen vom Idealfall aus, in dem ein beleuchteter Punkt exakt den Abstand zur Lichtquelle aufweist, der in der Shadow Map vermerkt ist. In der Praxis ist jedoch sowohl die Präzision der Berechnungen als auch die Auflösung der Shadow Map pro Pixel begrenzt. Ergebnis dieser Ungenauigkeiten ist, dass Flächen fälschlicherweise Schatten auf sich selbst werfen können. Theoretisch könnte dieses Problem vermieden werden, indem ein geringer Korrekturwert (*bias*) auf alle Pixel der Shadow Map addiert wird. Dies kann beispielsweise durch eine leichte Translation der Lichtquellenmatrix gegenüber der realen Lichtposition geschehen.

Everitt et al. [EVE+01] merken jedoch an, dass diese Lösung suboptimal ist. Die Shadow Map kann an verschiedenen Stellen des Bildes unterschiedlich grob aufgelöst sein, unter Umständen deutlich grober als das zu rendernde Bild. Dadurch wird der tatsächlich vorhandenen Geometrie unzureichend Rechnung getragen. Abbildung 53 (links) zeigt ein Beispiel: Die Shadow Map (gelbe Punkte) ist grob aufgelöst, dadurch wird die zwischen den Shadow-Map-Pixeln befindliche Spitze in der Geometrie nicht berücksichtigt. Als Ergebnis scheinen Pixel an dieser Spitze fälschlicherweise im Schatten zu liegen (schwarze Punkte). Dieser Effekt muss mit einem ausreichend großen Korrekturwert ausgeglichen werden. Das

führt jedoch zu einem anderen Problem – der Korrekturwert verschiebt tatsächlich die Schatten in der Szene nach hinten. Für kleine Werte ist diese Verschiebung kaum sichtbar, wird der Bias jedoch zu groß, entsteht ein falscher optischer Eindruck und Objekte scheinen in der Szene zu schweben. OpenGL bietet mit *Polygon Offset* eine Möglichkeit, Vertices abhängig vom Neigungswinkel ihres Dreiecks zu verschieben. Damit kann das beschriebene Problem mit einer geringeren Verschiebung der Schatten in der Szene gelöst werden. In Microsoft DirectX fehlt eine analoge Funktion bislang.

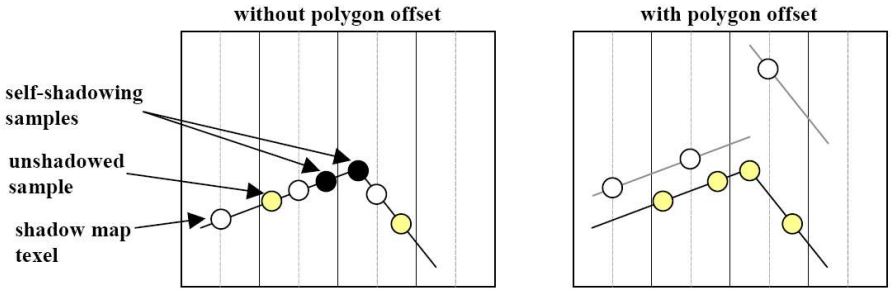


Abbildung 53: Die grobe Auflösung der Shadow Map führt zu fälschlicherweise schattierten Pixeln (links). Polygon Offset in OpenGL kann dieses Problem verhindern. (Quelle : [EVE+01])

Generell bleibt das Problem, dass die nötigen Korrekturwerte stark von der Szene und dem subjektiven optischen Eindruck abhängen und daher nur manuell sinnvoll festgelegt werden können. Artefakte können zwar minimiert, aber nicht sicher vermieden werden.

Ein weiteres, verwandtes Problem entsteht, wenn ein Schatten werfendes und ein Schatten empfangendes Objekt so nahe beieinander liegen, dass sie aufgrund der begrenzten Auflösung der Tiefenwerte in der Shadow Map nicht mehr unterschieden werden können. Der Effekt ist mit Z-Fighting vergleichbar und kann ähnlich bekämpft werden (vgl. Kapitel 11)

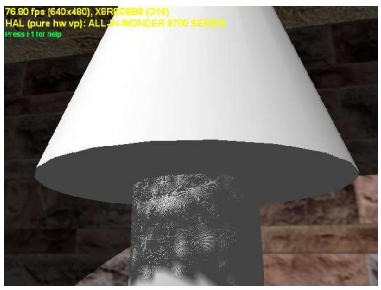


Abbildung 54: Die begrenzte Präzision der Shadow Map führt zu Darstellungsfehlern ähnlich dem Z-Fighting. (Quelle: [MS04b])

12.3. Vergleich von Stencil Shadows und Shadow Maps

Zusammenfassend die Vor- und Nachteile von Stencil Shadows und Shadow Maps im direkten Vergleich:

	Stencil Shadows	**Shadow Maps**
Performance	schlecht; n+1 Pässe über komplette Szene für n Lichtquellen, 2 Pässe mit jedem Schattenvolumen, Erzeugung der Schattenvolumen	moderat, n+1 Pässe über komplette Szene für n Lichtquellen
Self Shadowing	ja	ja
Multiple Lichtquellen	ja	ja
Schatten auf beliebigen Oberflächen	ja	ja
Spot-Lichtquellen	ja	ja
Direktionale Lichtquellen	ja	ja
Punktlichtquellen	ja	ja, aber sehr ineffizient
Schattenkanten auf Umgebung	perfekt	u.U. grob (sichtbares Pixelraster)
Schattenkanten auf Schatten werfendem Objekt	u.U. unsauber, benötigt hoch aufgelöste Geometrie	wie auf Umgebung
Spezielle Anforderungen an Geometrie	ja, geschlossene Geometrie	nein
Teilweise transparente Texturen berücksichtigt	nein	ja
Weitere Probleme	Doppelte Schattenbilder, wenn nicht ausschließlich Stencil Shadows verwendet	Z-Fighting; Ungenauigkeit der Z-Berechnung und der Shadow Map
Hauptschwierigkeit bei der Implementation	Vermeidung von Clipping an Near und Far Plane; Lösung: Far Plane entfernen, Carmacks Reverse	Schlechte Auflösung der Shadow Map, ungleiche Verteilung im Blickfeld der Kamera; Lösung: Perspective Shadow Maps, Light Space Perspective Shadow, Überblenden mehrere Shadow Maps für unterschiedliche Entfernungen

12.4.Echtzeitschatten in 3DView2

3DView2 soll eine komplett veränderbare Welt visualisieren, d.h. es können jederzeit Objekte in der Welt erscheinen, aus ihr verschwinden, sich bewegen oder ihre Form ändern; daher ist für die Schattenberechnung nur ein Echtzeitverfahren denkbar. In Betracht kamen somit nur die beiden zur Zeit gängigen Verfahren, *Shadow Volumes* und *Shadow Maps*. Die Wahl fiel auf Shadow Maps; ausschlaggebend dafür waren die folgenden Gründe:

- Das implementierte Inselszenario enthält eine große Anzahl von Objekten, z.B. ausgedehnte Wälder mit einer Vielzahl von Pilzen und kleineren Pflanzen. In einer solchen Situation erweisen sich Shadow Maps als das weitaus effizientere Verfahren.

- Einige Objekte machen von teilweise transparenten Texturen Gebrauch. Beispielsweise können Bäume mit ihrem dichten Blätterdach oder ihren feinen Nadeln nur schwer allein mit Geometrie dargestellt werden; transparente Texturen sind eine einfache und zudem um ein vielfaches effizientere Alternative. Shadow Maps berücksichtigen diese Transparenzen bei der Schattenberechnung, während Shadow Volumes allein aus der zugrundeliegenden Geometrie erzeugt werden.

Die Einschränkung von Shadow Maps gegenüber Stencil Shadows bezüglich des Lichtquellentyps stellt kein Problem dar, da für das Inselszenario lediglich eine direktionale Lichtquelle vorgesehen ist.

Shadow Maps ohne Pixel Shader

3DView2 sollte nach Möglichkeit auf mit weniger leistungsstarken Rechnern kompatibel bleiben. Pixel Shader sind eine Technologie, die momentan noch nicht auf jedem Gerät vorausgesetzt werden kann. Äquivalente Lösungen, die ohne Pixel Shader funktionieren, sind auf spezielle Graphikkarten bestimmter Herstelle beschränkt.

Es gibt jedoch eine Möglichkeit, Shadow Maps in einer vereinfachten Form ohne Pixel Shader zu realisieren. Es werden dabei vereinfachende Annahmen darüber gemacht, welche Objekte auf welche anderen Objekte Schatten werfen können. Im Falle von 3DView2 wird davon ausgegangen, dass lediglich Objekte wie z.B. Bäume oder Agenten Schatten werfen können. Das Terrain wirft selbst keine Schatten; Schatten können nur auf das Terrain geworfen werden. Objekte werfen keine Schatten auf sich selbst oder andere Schatten werfende Objekte.

Diese Einschränkungen sind akzeptabel; es gibt darüber hinaus noch einen zweiten Grund, warum das Terrain keine Schatten werfen sollte. Aus Effizienzgründen wurde der Bereich, in dem Objekte Schatten werfen, auf die unmittelbare Nähe des Betrachters beschränkt. 3DView2 ist für große Sichtweiten von einem Kilometer und mehr ausgelegt. Zeitweise können hunderte von Bäumen gleichzeitig im Sichtfeld sein, die meisten sind jedoch hunderte Meter entfernt. Wenn Objekte in dieser Entfernung keine Schatten werfen, fällt das optisch kaum auf, da sie auf dem Bildschirm sehr klein erscheinen. Die Zeitersparnis, die entsteht, weil diese Objekte nicht in die Shadow Map gerendert werden müssen, ist jedoch groß. Desweiteren verteilt sich die Auflösung der Shadow Map auf alle Schatten werfenden Objekte. Weniger Schatten werfende Objekte bedeuten eine bessere Schattenqualität für die verbleibenden Objekte.

Beim vereinfachten Shadow Mapping werden nicht Tiefenwerte, sondern einfache Verdeckungsinformationen in die Shadow Map gerendert. Alle Schatten werfenden Objekte – und nur diese – werden aus der Perspektive der Lichtquelle in eine Textur gerendert, wobei nur der Alphakanal ihrer jeweiligen Texturen geschrieben wird. Der Alphakanal

enthält normalerweise den Transparenzgrad eines Texels. Der Alphakanal der Shadow Map wird zuvor mit 1.0 (d.h. vollständig transparent) initialisiert. Der Vorgang ist ein normales Rendering in eine normale Textur, die Hardware muss dazu weder Pixel Shader noch spezielle Texturformate beherrschen.

Die Shadow Map wird im zweiten Schritt auf die Szene – d.h. konkret auf das Terrain – projiziert, statt eines Vergleiches von Z-Werten wie beim herkömmlichen Shadow Mapping kann bei der vereinfachten Variante direkt am Alphakanal der Shadow Map abgelesen werden, ob ein Punkt im Schatten liegt oder nicht.

Der Verzicht auf Tiefeninformationen in der Shadow Map begründet die Einschränkung, dass nur Schatten von Objekten auf das Terrain möglich sind und keine anderen Kombinationen. Die vereinfachte Shadow Map speichert für jeden Punkt des Terrains die Information, ob sich zwischen diesem Punkt und der Lichtquelle ein das Licht blockierendes Objekt befindet. Es gibt keine Information darüber, in welcher Entfernung sich das Objekt befindet. Dieser Umstand kann leider zu Darstellungsfehlern führen, die es beim klassischen Shadow Mapping nicht gibt.

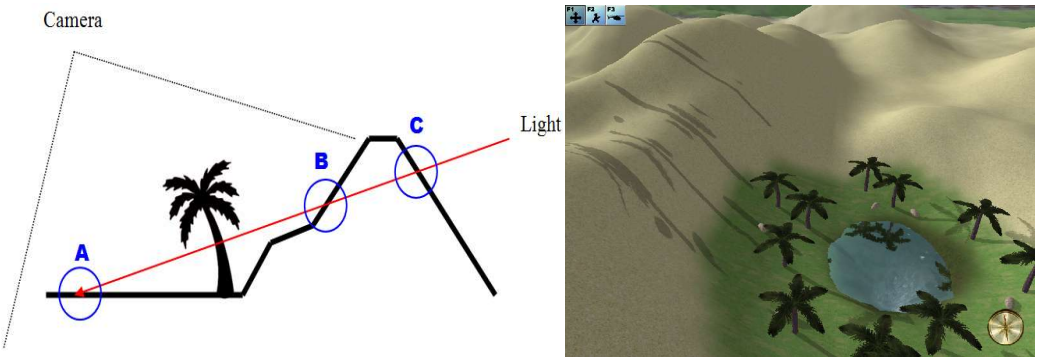


Abbildung 55: Bei der vereinfachten Shadow-Mapping-Technik wird den Punkten A, B und C der gleiche Punkt der Shadow Map zugeordnet. Dadurch kann es zu falschen Schattenbildern kommen (rechts).

Abbildung 55 veranschaulicht ein solches Problem. Die Punkte A, B und C auf dem Terrain liegen aus Sicht der Lichtquelle auf einer Geraden; daher wird ihnen der gleiche Punkt der Shadow Map zugeordnet. In die Shadow Map wird nur die Palme, nicht aber das Terrain gerendert. Der rot dargestellte Lichtstrahl schneidet die Palme, daher zeigt der entsprechende Pixel der Shadow Map ein blockierendes Objekt und Punkt A liegt korrekt im Schatten. Gleichzeitig liegen allerdings auch die Punkte B und C im Schatten, was nicht korrekt ist – die Schatten werfende Palme liegt auf dem Lichtstrahl hinter diesen Punkten. Da die vereinfachte Shadow Map jedoch nur anzeigt, dass ein Objekt das Licht blockiert, nicht aber, wo es geschieht, ist das später nicht mehr erkennbar. Dieses Problem wird vermieden, so lange das Licht in einem relativ steilen Winkel auf das Terrain fällt und sich daher nicht mehr als ein Schnittpunkt mit dem Terrain ergibt.

Die vereinfachte Shadow Mapping Technik hat neben den geringeren Hardwarevoraussetzungen noch einen anderen echten Vorteil gegenüber dem klassischen Shadow Mapping. Die Shadow Map speichert pro Pixel nicht nur die Information „Licht“ oder „kein Licht“, sondern vielmehr einen Grad an Lichtdurchlässigkeit, nämlich den Alpha-Wert aus den Texturen der blockierenden Objekte. Somit können bei diesem Verfahren auch halbtransparente Objekte korrekte Schatten werfen.

Anwendung auf das Terrain

Das Terrain in 3DView2 wird durch ein Multipassverfahren erzeugt (siehe Abschnitt 9.5.3, Seite 80). Da die meisten Terrainblöcke durch sukzessives Blending verschiedener Materialtexturen entstehen, können Schatten nicht einfach durch *Multitexturing* hineinmultipliziert werden. Statt dessen wird die Shadow Map in einem weiteren Blendingpass nach allen Bodenmaterialien aufgebracht. Es können eine beliebige Schattenfarbe C_{Shadow} sowie eine Schattentransparenz $A_{ShadowMax}$ gewählt werden. Die endgültige Terrainfarbe C_{final} ergibt sich aus

$$C_{final} = C_{Terrain} \cdot (1 - (A_{ShadowMap} \cdot A_{ShadowMax})) + C_{Shadow} \cdot (A_{ShadowMap} \cdot A_{ShadowMax}) \quad ,$$

wobei $C_{Terrain}$ die Terrainfarbe als Ergebnis der bisherigen Pässe und $A_{ShadowMap}$ der Alphawert aus der Shadow Map sind. Insofern ein Terrainblock nur ein einziges Bodenmaterial verwendet oder er aufgrund seiner Entfernung zum Betrachter nur mittels der Basistextur gerendert wird, muss dies nicht in einem separaten Pass erfolgen, sondern kann aus Effizienzgründen auch mittels Multitexturing realisiert werden.

13.Konfigurationsmanagement

Eines der Hauptprobleme bei der Entwicklung von 3DView2 bestand in der breiten Palette von Hardware, die unterstützt werden sollte. Dies bezieht sich einerseits auf die sehr unterschiedlichen technischen Fähigkeiten von Graphik- und Soundkarten, andererseits aber auch auf die unterschiedliche Leistungsfähigkeit verschiedener PCs. Beispielsweise verfügten einige der Rechner, auf denen 3DView2 laufen sollte, über 1GB RAM, während andere nur 128MB vorweisen können. Einige Graphikkarten hatten 256MB RAM, andere weniger als 32MB. Die besten Graphikprozessoren unterstützten das Shadermodell 2.0, während andere bereits an Tripple-Buffering und Kantenglättung scheiterten.

Die Schwierigkeit besteht nun darin, die Applikation entsprechend flexibel zu gestalten. Einerseits sollen die Fähigkeiten leistungsfähiger Rechner ausgenutzt werden, um eine möglichst hohe graphische Darstellungsqualität zu erreichen, andererseits muss die Kompatibilität zu weniger leistungsstarken Geräten gewährleistet bleiben. Um dies zu erreichen, sind drei Dinge notwendig:

1. Die Anwendung muss, wann immer es möglich ist, die Fähigkeiten des Systems automatisch erkennen.

2. Die Anwendungskomponenten sollten hinsichtlich ihres Ressourcenbedarfs skalierbar sein.

3. Die Anwendung muss, wo immer es möglich ist, durch den Benutzer konfigurierbar sein.

Der erste Punkt bedarf keiner großen Erklärung; wenn die Hardware- oder Softwareumgebung bestimmte technische Limitationen aufweist, müssen Anwendungsfeatures abgeschaltet oder durch alternative Implementationen ersetzt werden. Dies sollte automatisch geschehen, um den Anwender zu entlasten. Viele alternative Codepfade erhöhen allerdings den Entwicklungs-, Test- und Wartungsaufwand drastisch.

Auf den zweiten und dritten Punkt soll kurz eingegangen werden. Bei graphischen Anwendungen ist es meist so, dass der Ressourcenbedarf eng an die Darstellungsqualität gekoppelt ist. Je nach Situation kann die Darstellungsqualität ein Parameter sein, der variiert werden kann, ohne die Funktion der Anwendung zu beeinträchtigen. Um ein Beispiel zu nennen: In Kapitel 9 zum Thema Terrainvisualisierung wurde gezeigt, dass die Auflösung der Geländeform (das *Mesh*) zugunsten besserer Darstellungsgeschwindigkeit reduziert werden kann. Dabei kommt es aber zu einer Verfälschung der ursprünglichen Form, beispielsweise können sich Sichtlinien verändern. Ob das akzeptabel ist, hängt vollkommen von der konkreten Anwendung ab. Insofern es nur um einen subjektiv korrekten Eindruck geht, beispielsweise in einem Computerspiel, ist eine solche Vereinfachung vertretbar, für Architekturvisualisierungen oder Flugsimulationen hingegen kann sie problematisch sein.

Auf einige graphische Effekte, beispielsweise Schatten oder Wasserreflexionen, kann je nach Leistungsfähigkeit des Rechners eventuell auch komplett verzichtet werden. Auch hier ist Vorsicht geboten – beispielsweise verbessern Schatten nicht nur subjektiv die Optik, sondern sind auch von großer Bedeutung für die Orientierung im Raum.

Ein letzter Punkt, über den nachgedacht werden sollte, ist die Frage, inwiefern es zu Problemen führt, wenn die Applikation auf unterschiedlichen Rechnern unterschiedliches Verhalten zeigt oder auch nur eine unterschiedliche visuelle Darstellung liefert. Für Experimente, wie beispielsweise Dörner sie mit seiner Psi3D-Software durchgeführt hat

[GER+01], ist es von entscheidender Bedeutung, dass alle Versuchspersonen von exakt gleichen Voraussetzungen ausgehen.

Innerhalb der eben genannten Grenzen – einerseits die Eigenschaften des Rechnersystems, andererseits die Erfordernisse der Anwendung – sollte die Konfiguration dem Benutzer überlassen werden. Durch die Vielzahl der Einstellungsmöglichkeiten ist es gar nicht möglich, für jeden Rechner eine optimale Konfiguration zu finden. Was optimal ist, ist gerade bei graphischen Anwendungen eine zutiefst subjektive Frage.

Für den Benutzer ist wichtig, dass die ihm zur Verfügung stehenden Anwendungsparameter gut dokumentiert sind. Es sollte für alle Parameter sinnvolle Standardwerte geben. Außerdem müssen die Parameterwerte auf den zulässigen Bereich eingegrenzt werden, d.h. Fehlbedienungen sollten ausgeschlossen sein.

Das 3DView2-Konfigurationssystem

Über die eben genannten Überlegungen hinaus gab es bei der Entwicklung von 3DView2 einige spezielle Erfordernisse:

- Die Anwendung sollte sowohl über eine persistente Konfigurationsdatei als auch über Kommandozeilenparameter konfigurierbar sei. Letzteres ist praktisch, um die Anwendung aus anderen Programmen heraus, beispielsweise aus Eclipse, starten zu können.

- Da sich das Projekt ständig weiterentwickelt, ändern sich die zur Verfügung stehenden Parameter und Wertebereiche von Zeit zu Zeit. Andererseits ist die Software bereits in Benutzung, d.h. Nutzer haben bereits individuelle Konfigurationen für ihre Rechner angelegt. Diese Konfigurationen sollten beim Update auf eine neue Version nicht komplett überschrieben, sondern so weit wie möglich übernommen und aktualisiert werden.

Aufgrund dieser Überlegungen wurde für 3DView2 ein XML-basiertes Konfigurationssystem implementiert. Im Zentrum des Systems stehen die so genannten Parameter. Zu jedem Parameter gehören:

- Name
- Gruppe
- Datentyp
- Wertebereich
- Standardwert
- aktueller Wert
- Beschreibungstext

Der Name eines Parameters ist ein beliebiger Textstring, beispielsweise *Volume*. Zur besseren Übersicht können Parameter in Gruppen geordnet werden werden. Gruppen funktionieren ähnlich wie Verzeichnisse in einem Dateisystem; der Gruppenname wird dem Parameter mit einem Schrägstrich vorangestellt und bildet so den vollständigen Namen des Parameters, beispielsweise *Sound/Volume*. Gruppen können ihrerseits wieder Gruppen enthalten, beispielsweise kann die eine Gruppe *Sound* eine Gruppe *Music* enthalten, der vollständige Gruppenname von *Music* wäre dann *Sound/Music* und ein vollständiger

Parametername könnte *Sound/Music/Volume* lauten.

Jeder Parameter hat einen festgelegten Datentyp. Momentan gibt es vier Datentypen, das System ist bei Bedarf aber erweiterbar. Je nach Datentyp kann ein Wertebereich für den Parameter festgelegt werden:

- Typ Bool: Wahrheitswerte, Wertebereich ist true/false
- Typ Integer: ganze Zahl, Wertebereich wird durch Minimal- und Maximalwert definiert
- Typ Float: gebrochen rationale Zahl, Wertebereich wird durch Minimal- und Maximalwert definiert
- Typ String: Zeichenkette, Wertebereich ist entweder nicht eingeschränkt oder wird durch eine vollständige Liste möglicher Werte definiert

Darüber hinaus existiert für jeden Parameter ein Standardwert, der angenommen wird, wenn der Parameter gar nicht oder inkorrekt spezifiziert wird, sowie eine textuelle Beschreibung des Parameters.

Die Definition der Parameter mit ihren Eigenschaften geschieht in der Applikation, d.h. im Quelltext über den Aufruf von C++-Funktionen. Eine externe Definition würde wenig Sinn machen, da die Parameter ohnehin im Quelltext evaluiert werden und eine externe Definition somit in jedem Fall mit dem Quelltext synchron gehalten werden müsste. Eine typische Parameterdefinition sieht folgendermaßen aus:

```
m_xConfiguration.AddParameterFloat( "general/rangeofvision",
      "Determines the range of vision in the application. Reduce
this value if     the application runs too slow.",
      1000.0f, 100.0f, 2000.0f);
```

Die Funktionsparameter in diesem Beispiel sind: vollständiger Name des Parameters (d.h. einschließlich Gruppen), Beschreibung, Standardwert, Minimalwert, Maximalwert. Der Typ des Parameters leitet sich aus der verwendeten Funktion ab; in diesem Fall ist es Float. Nach der Definition der Parameter ist der aktuelle Wert jedes Parameters gleich seinem Standardwert.

Das Konfigurationssystem kann Parameterwerte aus XML-Dateien lesen und wieder schreiben. Zuerst wird immer versucht, eine Konfigurationsdatei einzulesen. Alle in der Datei spezifizierten Parameterwerte werden übernommen, sofern der Parameter definiert ist und der Wert dem Wertebereich entspricht. Enthält die Datei unbekannte Parameter, so werden diese verworfen; sind Werte für Parameter nicht oder falsch spezifiziert, so nehmen diese Parameter ihre Standardwerte an. Auf diese Weise werden aus veralteten oder inkorrekten Konfigurationsdateien alle sinnvollen Daten übernommen, während die übrigen keinen Schaden verursachen können. Auch durch das komplette Fehlen einer Konfigurationsdatei entsteht kein Problem.

Anschließend wird die Konfigurationsdatei mit den nun aktuellen Werten komplett neu geschrieben. Aus dem Beschreibungstext, dem Wertebereich und dem Standardwert werden automatisch Kommentare erzeugt, die dem Benutzer den Umgang mit der Konfigurationsdatei erleichtern sollen. Hier als Beispiel eine kleine, auf die beschriebene Weise entstandenen Konfigurationsdatei:

```
<configuration>
   <sound>
        <!--Determines whether sound is enabled-->
        <!--Possible Values: true, false-->
        <!--Default Value: true-->
        <soundenabled>true</soundenabled>
    </sound>
    <shadow>
        <!--Shadow map texture resolution in pixels. -->
        <!--Possible Values: 1024, 512, 256, 128, 64, 32-->
        <!--Default Value: 512-->
        <shadowmapresolution>512</shadowmapresolution>
        <!--Determines whether shadows are enabled or not.-->
        <!--Possible Values: true, false-->
        <!--Default Value: true-->
        <shadowsenabled>true</shadowsenabled>
    </shadow>
</configuration>
```

Aus Gruppen werden Tags, die dann ihrerseits Gruppentags oder Parameter enthalten können. Durch die Einrückung gestalten Gruppen die Datei übersichtlicher. In einem letzten Schritt kann das Konfigurationssystem nun die Kommandozeile der Applikation auswerten. Alle definierten Parameter können im Format

–<*Parametername*> <*Parameterwert*>

auf der Kommandozeile angegeben werden, also beispielsweise

```
3dview2.exe -soundenabled true -shadowsenabled false
```

Auch hierbei wird die Einhaltung des Wertebereiches forciert. Die Applikation kann die Parameterwerte danach zu jedem beliebigen Zeitpunkt auslesen; dazu wird lediglich der vollständige Name des Parameters benötigt:

```
bool bSound = m_xConfiguration.GetValueBool("sound/soundenabled");

float fDist = m_xConfiguration.GetValueFloat
("general/maxdistancefromterrain");
```

Die Parameter werden intern in einem Rot-Schwarz-Baum gehalten, so dass sie anhand ihres Namens über eine binäre Suche effizient gefunden werden können. Trotzdem muss festgehalten werden, dass dieses System auf Benutzerfreundlichkeit und Robustheit und nicht auf Geschwindigkeit ausgelegt ist.

Das Konfigurationssystem in dieser Form hat sich in der Praxis bewährt und bietet viele Vorteile:

- Parameter werden dort definiert, wo sie auch verwendet werden – im Quelltext; alle Änderungen müssen nur an einer Stelle vorgenommen werden
- Die Beschreibungstexte werden zusammen mit dem Parameter definiert und sind somit leicht aktuell zu halten; die Dokumentation der Parameter wird automatisch aus diesem Text und dem tatsächlich verwendeten Wertebereich und Standardwert erzeugt
- Das Programm erzeugt eine gültige Konfigurationsdatei, wenn keine vorhanden sein sollte.
- Alte oder fehlerhafte Konfigurationsdateien werden von der Applikation automatisch aktualisiert und korrigiert, d.h. nicht mehr vorhandene Parameter werden gelöscht, neu hinzugekommene eingefügt, Werte außerhalb des gültigen Wertebereiches werden auf Standardwerte gesetzt.
- Der Applikationsentwickler kann davon ausgehen, dass jeder definierte Parameter zu jedem Zeitpunkt einen gültigen Wert hat, zusätzliche Prüfungen können entfallen

14.Zusammenfassung und Ausblick

Zusammenfassung

Ziel der vorliegenden Arbeit war die Entwicklung von 3DView2, einer Visualisierungs- und Steuerungskomponente für die virtuelle Weltsimulation des MicroPsi-Toolkits. 3DView2 sollte in der Lage sein, die simulierte Welt in Echtzeit dreidimensional darzustellen, sollte als Editor für diese Welt fungieren können und die manuelle Steuerung von Agenten ermöglichen. Desweiteren sollte auf Basis der 3DView2-Technologie ein Prototyp des 3DEmotion-Viewers entwickelt werden. Der 3DEmotion-Viewer soll auf einem dreidimensional modellierten Gesicht verschiedene Mimiken erzeugen können und so die Visualisierung von Emotionen erlauben.

Zu Beginn dieser Arbeit wurde das Psi-Projekt von Professor Dietrich Dörner vorgestellt, welches das Vorbild für das MicroPsi-Projekt und den entwickelten 3DViewer ist. Die dem Agentenmodell zugrunde liegende Theorie wurde bewusst außer Acht gelassen, statt dessen wurden ausführlich die technischen Grundlagen der virtuellen Weltsimulation und der dreidimensionalen Visualisierung der Psi-Welt diskutiert. Das Objektverwaltungs- und Nachrichtensystem von Psi Reality 3D erlaubt das Erstellen neuer Objekte und Verhaltensweisen über Textdateien bzw. proprietäre Tools und ohne Modifikation des eigentlichen Programmquelltextes. Das System ist flexibel, jedoch schwer erweiterbar und mit erheblichem Arbeits- und Einarbeitungsaufwand verbunden. Es wurde auch analysiert, welche Möglichkeiten zur Verbesserung der Software existieren. Überarbeitungsbedürftig ist einerseits die Steuerung; sie ist für Anfänger unnötig schwer erlernbar und ignoriert etablierte Standards. Hauptkritikpunkt ist die Qualität der graphischen Darstellung; sie ist angesichts der heutigen technischen Möglichkeiten nicht mehr zeitgemäß und hat teilweise sogar negative Auswirkungen auf Bedienbarkeit und Verständlichkeit der Anwendung.

Aufbauend auf Dörners Psi Reality 3D wurde das MicroPsi-Projekt vorgestellt, welches eine Formalisierung und Erweiterung der Psi-Theorie anstrebt und eine umfangreiche Softwareumgebung zur Erstellung und Simulation künstlicher Agenten entwickelt. Auch hier wurde auf eine tiefgreifende Diskussion des Agentenmodells verzichtet, statt dessen sollte nur der zum Verständnis der Ziele dieser Arbeit notwendige Kontext vermittelt werden.

Entgegen der ursprünglichen Planung wurde bereits in einem frühen Stadium der vorliegenden Diplomarbeit entschieden, dass statt einer Open Source Graphikengine eine komplette Eigenentwicklung auf Basis von Microsoft DirectX und einiger ausgewählter Open-Source-Bibliotheken eingesetzt werden soll. Kapitel 5 erläuterte die Gründe hierfür – die ursprünglich anvisierte Nebula2-Engine war zu diesem Zeitpunkt noch jung und instabil, zudem war sie unzureichend dokumentiert. Die Entscheidung zugunsten einer eigenen Technologie brachte ein hohes Maß an Flexibilität, es konnten für jeden Aspekt des 3DViewers maßgeschneiderte Lösungen implementiert werden, ohne eine bestehende rigide Softwarearchitektur berücksichtigen zu müssen.

Es muss eingeräumt werden, dass die Implementation eigener Technologien einen erheblichen Teil der Entwicklungszeit beansprucht hat, was ursprünglich nicht so vorgesehen war. Die Nutzung einer existierenden 3D-Engine hätte mit Sicherheit in vielen Bereichen zu schnelleren Ergebnissen geführt und Entwicklungsressourcen freigesetzt, die in die eigentliche Funktionalität des 3DViewers hätten fließen können. Beispielsweise hätte Nebula2 fertige Implementationen eines Terrainsystems und dynamische Echtzeitschatten geboten – allerdings beides auf Grundlage von Techniken, die für den 3DViewer nicht

unbedingt geeignet gewesen wären. Ob Nebula2 letztendlich die adäquatere Wahl gewesen wäre, konnte somit im Rahmen dieser Arbeit nicht geklärt werden.

Obwohl die Implementierung einer eigenen 3D-Engine in DirectX durch die gewonnene Flexibilität viele Probleme löste, führte sie gleichzeitig zu neuen. Die verwendeten Graphikexportprogramme für das Microsoft X-File-Format erwiesen sich in der Praxis als instabil bzw. sehr schwierig in der Handhabung. Ausnahmslos alle Personen, die Graphiken für den 3DViewer oder 3DEmotion erstellt haben, stießen auf Probleme beim Export in das X-File-Format. Teilweise musste durch mehrere Tage dauerndes Experimentieren herausgefunden werden, unter welchen Randbedingungen die Exporter korrekt funktionieren; in einigen Fällen, beispielsweise beim 3DEmotion-Gesicht oder dem 3D-Modell des MicroPsi-Agenten, musste daraufhin fast die gesamte Arbeit noch einmal von vorn begonnen werden. Die Exporttools der Nebula2-Engine hingegen gelten als stabil, sind aber auf das Graphikprogramm Maya beschränkt und wie bereits erläutert nicht kostenlos erhältlich.

Kapitel 6 gab einen kurzen Überblick über Animationstechniken und stellte ausführlich Vertex Skinning, das heute gebräuchlichste Verfahren, vor. Vertex Skinning kommt in 3DView2 und dem 3DEmotion-Viewer zum Einsatz. Die für den Prototypen der 3DEmotion-Komponente gestellten Ziele wurden erreicht – das von einem Künstler modellierte und mit Bones versehene Gesicht lässt sich über ein graphisches Interface steuern; viele einzelne Partien wie Augen, Augenlider, Mundwinkel, Wangen etc. können unabhängig voneinander angesteuert und bewegt werden. Ob die 3DEmotion-Komponente indes tatsächlich zur Visualisierung von simulierten Emotionen künstlicher Agenten geeignet ist, wird erst die Zukunft zeigen. Es wird eine Komponente entwickelt werden müssen, die simulierte Emotionen in geeigneter Weise auf die virtuellen Muskeln des Gesichtes überträgt. Eine sinnvolle Anbindung von 3DEmotion an das MicroPsi-Toolkit setzt auch eine Weiterentwicklung des MicroPsi-Agenten voraus und wird daher vorraussichtlich Thema einer anderen Arbeit sein.

Kapitel 7 beschäftigte sich mit dem Objektverwaltungssystem des MicroPsi-Toolkits. Im Gegensatz zu Dörners Psi Reality 3D verwendet die MicroPsi-Weltsimulation Javaklassen um Objektklassen zu repräsentieren; eine Erweiterung der Weltsimulation bedeutet eine Erweiterung des Quelltextes. Die MicroPsi-Weltsimulation ist hinsichtlich Umfang und Komplexität noch sehr weit von Dörners Psi-Szenario entfernt, daher wäre ein wertender Vergleich beider Implementationsansätze verfrüht.

Es wurde festgestellt, dass eine dreidimensionale Visualisierung der Simulationswelt Anforderungen stellt, die über das existierende System hinausgehen. Da die Weltkomponente viele für die Visualisierung notwendige Daten nicht liefern kann, müssen diese Daten dem 3DViewer in gesonderten Konfigurationsdateien zur Verfügung gestellt werden. Für die angestrebte optische Qualität ist eine größere graphische Vielfalt erforderlich als das bei der existierenden, symbolhaften zweidimensionalen Darstellung der Fall ist. Als Lösung können im 3DViewer pro Objekttyp beliebig viele mögliche Visualisierungen definiert werden. Die Darstellung von mehreren tausend Objekten in einer dreidimensionalen Perspektive führt zu Geschwindigkeitsproblemen; als Konsequenz wurde ein LOD-Verfahren implementiert, das Objekte in ausreichender Entfernung durch vordefinierte, weniger detailierte Modelle ersetzt oder komplett ausblendet. Die dreidimensionale Geländeform, die in der existierenden Weltsimulation nicht berücksichtigt wird, führt zur Notwendigkeit, die Rotation einiger Objekttypen der jeweiligen Bodenform anzupassen. Durch die Definition von Bodenkontaktpunkten für die betroffenen Objekte lässt sich ein korrekter optischer Eindruck erzielen. Zum Abschluss des Kapitels wurde auf

die Notwendigkeit sehr effizienter Kollisions- und Sichtbarkeitstests für Objekte hingewiesen; 3DView2 verwendet zu diesem Zweck ein Standardverfahren – Quadtrees.

Kapitel 8 beschrieb das bisher in dieser Form nicht dokumentierte Kommunikationssystem der MicroPsi-Komponenten. Da 3DView2 als erste Komponente nicht in Java, sondern in C++ implementiert ist, konnte nicht auf die bestehende Implementation zurückgegriffen werden; statt dessen mussten die für einen Client notwendigen Teile des Systems in C++ dupliziert werden. Die resultierende Implementation steht damit in Zukunft auch anderen Entwicklern zur Verfügung, die MicroPsi um C++-Komponenten erweitern wollen.

Kapitel 9 beschäftigte sich mit der Darstellung von Terrain und wurde, obwohl es bei der ursprünglichen Konzeption der vorliegenden Arbeit nicht geplant war, zu einem der umfangreichsten. Es wurde gezeigt, welche Verfahren zur Zeit zur Visualisierung dreidimensionaler Terrains existieren und es wurde deutlich gemacht, dass die Einführung dedizierter Graphikhardware in den letzten zehn Jahren die Entwicklung neuer Verfahren auf diesem Gebiet notwendig gemacht hat. Ein weiterer Schwerpunkt des Kapitels war die Texturierung von Terrain, auch hier wurden mehrere Standardverfahren diskutiert. Die vorgestellten Verfahren wurden miteinander verglichen und es wurde erläutert, dass für den 3DViewer GeoMipMapping für die Geometriedarstellung und Splatting für die Texturierung die am besten geeigneten Techniken sind. Obwohl es Verfahren gibt, die zu besseren visuellen Ergebnissen führen, spricht für diese beiden die Tatsache, dass sich Terrains auch von graphischen Laien mit sehr geringem Aufwand erstellen lassen. Somit sind diese Verfahren zweckmäßig für die Verwendung in einem Experimentierbaukasten wie dem MicroPsi-Toolkit.

Es wurde dargelegt, wie sich Splatting und GeoMipMapping auf effiziente Weise miteinander kombinieren lassen. Die Kombination der Verfahren führte dazu, dass die von de Boer, dem Autor des GeoMipMapping, vorgeschlagene Technik zur Vermeidung von Löchern im Terrain nicht mehr anwendbar ist. Dieses Problem wurde gelöst, indem Anleihen bei einer anderen Terrainvisualisierungstechnik, dem Chunked LOD nach Ulrich, getätigt wurden.

Kapitel 10 schnitt das Thema Wasserdarstellung an; Schwerpunkt war die Darstellung von Reflexionen auf der Wasseroberfläche. Es wurde aufgezeigt, dass sich trotz der sehr komplexen Materie mit geringem Aufwand sehr gute Ergebnisse erzielen lassen, wenn das Anwendungsszenario entsprechend eingeschränkt wird. Die Implementation in 3DView2 kann Spiegelungen an der Wasseroberfläche effizient darstellen, da angenommen wird, dass die Wasseroberfläche eine perfekte Ebene ist. Auf die physikalische Simulation von Wellen wurde verzichtet, statt dessen wird ein leichter Wellengang mit Normal Maps simuliert.

Abbildung 56: 3DView2 integriert sich nahtlos in Eclipse

Ein Problem, welches während der Arbeit an 3DView2 unerwartet auftrat, war massives Z-Fighting. Kapitel 11 beschäftigte sich daher mit diesem Phänomen, erklärte, warum es auftritt und mit welchen Methoden es bekämpft werden kann. Für Graphikkarten, deren Z-Buffer-Auflösung nicht ausreichend ist, um Z-Fighting zu vermeiden, bietet 3DView2 eine Multipass-Rendering-Technik, die Z-Fighting verhindern kann – der Preis dafür ist eine etwas langsamere Darstellungsgeschwindigkeit.

Kapitel 12 beschäftigte sich mit der Echtzeitdarstellung von Schatten. Es wurden die beiden zur Zeit gängigsten Verfahren zur Darstellung komplett dynamischer Schatten vorgestellt – Shadow Maps und Stencil Shadows. Beide Verfahren wurden detailiert diskutiert und es wurde gezeigt, dass sie sehr unterschiedliche Stärken und Schwächen aufweisen. Der Vorzug bei der Implementation von 3DView2 wurde den Shadow Maps gegeben; ausschlaggebend dafür war die höhere Effizienz angesichts der großen Zahl von Objekten, die in einer typischen MicroPsi-Welt enthalten sind, und ebenso die Tatsache, dass Shadow Maps im Gegensatz zu Stencil Shadows Objekte mit teilweise transparenten Texturen korrekt handhaben können. Es wurde gezeigt, wie Shadow Maps unter vereinfachenden Annahmen auch auf Graphikkarten implementiert werden können, die nicht über Pixel Shader verfügen. Desweiteren wurde erläutert, wie Shadow Maps in das in Kapitel 9 vorgestellte Terrainsystem integriert wurden.

Schließlich wurde in Kapitel 13 darauf hingewiesen, dass eine komplexe graphische Anwendung wie der 3DViewer vom Benutzer leicht konfigurierbar sein muss, um auf einer möglichst großen Bandbreite unterschiedlich leistungsfähiger Hardware zufriedenstellend zu arbeiten. Es wurde diskutiert, welche Anforderungen an ein Konfigurationssystem gestellt werden müssen und es wurde ein neuartiges System vorgestellt, welches aus im Quelltext definierten Parametern automatisch gut dokumentierte XML-Konfigurationsdateien erstellt, diese bei Änderungen automatisch aktualisiert und die Gültigkeit aller Einstellungen überprüft. Dieses Konfigurationssystem kommt in 3DView2 und dem 3DEmotion-Viewer zum Einsatz, ist aber aufgrund seiner Allgemeinheit für beliebige andere Anwendungen geeignet.

3DView2 und der 3DEmotion-Viewer bestehen insgesamt aus über 80.000 Zeilen C++-Quelltext, verwendete Open-Source-Bibliotheken selbstverständlich nicht mitgerechnet. Auch wenn ein großer Teil davon nicht neu geschrieben werden musste, sondern aus vorhergehenden Projekten bereits vorhanden war, zeigt diese Zahl doch, wie komplex die Applikationen sind. Dies spiegelt auch die vorliegende Diplomarbeit wider – es wurden viele Probleme und Techniken angerissen, aber kaum ein Thema wurde in der Tiefe behandelt, wie es möglicherweise angemessen wäre. Alle Themen erschöpfend zu behandeln war nicht die Intention dieser Arbeit und konnte es aus Zeitgründen auch nicht sein. Ziel war vielmehr, existierende Standardverfahren auszuwählen und sie zu einer komplexen und benutzbaren Anwendung zu kombinieren – dieses Ziel wurde erreicht.

3DView2 ist zu einer leistungsfähigen Anwendung geworden, welche die MicroPsi-Weltsimulation eindrucksvoll graphisch darstellen kann, als Editor für die Welt fungiert und die Fernsteuerung von Agenten erlaubt. Seine Feuertaufe hat 3DView2 mit der Einführung von Psitopia[7] bestanden. Psitopia ist eine MicroPsi-Weltkomponente, die 24 Stunden täglich über das Internet erreichbar ist. Mit dem MicroPsi-Toolkit ist es möglich, von jedem internetfähigen Rechner auf der Welt Agenten in diese Simulationswelt zu schicken und sie dort miteinander interagieren zu lassen. Über die World Console oder 3DView2 kann diese Welt beobachtet oder manipuliert werden. Auf der Psitopia-Homepage steht zu diesem Zweck ein vorkonfigurierter 3DViewer zum Download. Mit dem 3DViewer ist es dem Benutzer auch möglich, selbst Agenten durch die Welt zu steuern.

7 http://psitopia.cognitive-agents.org:8080/org.micropsi.psitopia/

Abbildung 57: Im Editormodus (links) können Objekte platziert und gelöscht werden. Im Agentenmodus (rechts) können Agenten in der Welt gesteuert werden. Beliebig viele 3DViewer können gleichzeitig mit dem Server verbunden sein.

Obwohl das 3DViewer-Projekt erfolgreich verlaufen ist, konnten viele der ehrgeizigen Ziele und Ideen nicht verwirklicht werden. Beispielsweise waren ursprünglich ein dynamischer Tag-/Nachtwechsel geplant, das Erzeugen stereoskopischer Bilder oder die Möglichkeit, die Weltsimulation mit einem Datenhandschuh zu beeinflussen. All diese Ideen mussten aus Zeitgründen zurückgestellt werden, um die zentralen Ziele dieser Arbeit innerhalb des vorgegebenen Zeitrahmens zu erreichen. Ein Hauptgrund, dass die anderen ehrgeizige Pläne bis jetzt nicht verwirklicht werden konnten, ist der hohe zusätzliche Arbeitsaufwand, der durch die Entscheidung zugunsten einer selbst implementierten 3D-Engine entstanden ist.

Es muss in diesem Zusammenhang auch erwähnt werden, dass das MicroPsi-Toolkit ebenfalls noch in ständiger Entwicklung begriffen ist. Während der Entwicklung von 3DView2 traten viele Probleme und Fehler im MicroPsi-Toolkit zu Tage, die teilweise von anderen Entwicklern im Rahmen des Gesamtprojektes behoben werden mussten; die Anbindung des 3DViewers erforderte auch die Implementation neuer Features seitens des MicroPsi-Teams. Der MicroPsi-Agent ist zur Zeit erst zu sehr rudimentärem Verhalten fähig, entsprechend ist auch die Weltsimulation noch sehr einfach gehalten, d.h. es gibt nur wenige Objekttypen und nur minimale Interaktionsmöglichkeiten. Die Anbindung verschiedener Animationen an die Agenten konnte bisher nur sehr elementar realisiert werden, was erstens auf noch sehr rudimentäre Weltsimulation zurückzuführen ist und zweitens auch auf Zeitmangel.

Graphisch bleibt 3DView2 bisher leider hinter seinen technischen Möglichkeiten zurück, da es sehr schwer war, Graphiker an das Projekt zu binden. Die Erstellung von Graphiken, also 3D-Modellen, Animationen und Texturen, ist enorm zeitaufwändig – allein die Erstellung des Gesichtes für 3DEmotion hat einen Künstler mehr als 50 Arbeitsstunden gekostet. Aus diesem Grund konnten einzelne am Projekt beteiligte immer nur einen Beitrag leisten, was letztendlich nicht zu einem stilistisch konsistenten Gesamtbild geführt hat. Aus dem gleichen Grund mussten viele graphische Ideen bislang unverwirklicht bleiben.

Genau wie das MicroPsi-Projekt sich in Zukunft noch weiter entwickeln wird, sollte auch die vorliegende Version des 3DViewers nicht als fertiges Produkt, sondern nur als Zwischenschritt gesehen werden.

Ausblick

Das MicroPsi-Toolkit wird in der Zukunft weiterentwickelt werden und mit dem Toolkit wird auch der 3DViewer wachsen müssen. Während das Hinzufügen neuer Graphiken und Objekttypen nicht mit Programmierarbeit verbunden ist, erfordert die Einführung neuer Agententypen oder Aktionen eine Weiterentwicklung des Programms. Doch auch die Entwicklung des 3DViewers könnte umgekehrt das MicroPsi-Toolkit beeinflussen. Der 3DViewer hat unterschiedliche Bodenmaterialien und ein Terrain mit unterschiedlichen Höhenstufen eingeführt, jedoch auf einer rein optischen Ebene. Wünschenswert wäre, dass auch die Weltsimulation in Zukunft mit einem dreidimensionalen Terrain und verschiedenen Terraintypen arbeiten kann und diese tatsächlich Auswirkungen auf die Simulation haben.

Für die Weiterentwicklung des 3DViewers hat diese Arbeit eine solide Grundlage geschaffen. Die bisher nicht verwirklichten Ideen bleiben auf der Wunschliste für die Zukunft:

- dynamische Tag-/Nachtwechsel mit realistischer Beleuchtung der Umgebung
- Wettereffekte wie Regen, Schnee, Nebel
- realistisch wirkendes Gras, das sich im Wind wiegt
- dynamische Lichtquellen; beispielsweise um Lichtquellen in Braitenbergvehikel-Szenario darstellen zu können
- Partikeleffekte, beispielsweise Rauch und Feuer an den Vulkanen; Wasserfälle
- Animationen, die das Verhalten von Agenten visualisieren; dies setzt eine entsprechende Weiterentwicklung des MicroPsi-Toolkits voraus
- Refraktionen an Wasseroberflächen

Generell wäre es wünschenswert, einen Graphiker zu gewinnen, der die notwendige Zeit in das Projekt investieren kann, um die Landschaft und alle Objekte stilistisch konsistent zu gestalten. Denkbar wäre auch, verschiedene Visualisierungen der gleichen Welt zu erstellen, beispielsweise eine realistisch anmutende und eine im Comicstil. Die Graphikhardware entwickelt sich momentan in rasantem Tempo. Künftige Generationen von Graphikkarten werden dank freierer Programmierbarkeit und höherer Leistung extrem realistische Licht- und Schattenberechnungen ermöglichen. Von diesen Features können Anwendungen wie 3DView2 nur dann profitieren, wenn auch ein entsprechend höherer Aufwand bei der Erstellung graphischer Inhalte betrieben wird.

15.Danksagungen

An dieser Stelle möchte ich allen Menschen danken, die mich fachlich oder persönlich bei dieser Diplomarbeit unterstützt haben. Mein spezieller Dank gilt:

- meinem Freund und Kollegen Daniel Matzke, der wesentliche Teile des Frameworks geschrieben hat, auf dem unter anderem auch 3DView2 basiert. Daniel hat mir auch mehr als einmal bei technischen Schwierigkeiten zur Seite gestanden.

- meinem Betreuer Joscha Bach, der MicroPsi und das großartige Team überhaupt erst auf die Beine gestellt hat. Joscha ist ein unerschöpflicher Quell neuer Ideen, was die Ausführenden zuweilen an den Rand der Verzweiflung brachte, letztendlich aber das Projekt zu immer neuen Höhenflügen geführt hat. Darüber hinaus hat er sich regelmäßig als Graphiker und Inseldesigner betätigt.

- Ronnie Vuine, der für das MicroPsi-Toolkit technisch verantwortlich zeichnet und für mich immer ein verlässlicher und hilfreicher Ansprechpartner war.

- Mathias Füssel, der die MicroPsi-Weltsimulation betreut.

- meinem Freund und Kollegen Florian Busse, der sehr viel Arbeit und Energie in 3D-Modelle und Texturen für 3DView2 gesteckt hat und dabei zum Experten für X-File-Exporter avanciert ist.

- Henning Zahn und Jens Meisner, die einige 3D-Modelle und Texturen beigesteuert haben.

- meinen Eltern, die mir das Studium überhaupt erst ermöglicht haben.

- meinem Bruder Konstantin für das Korrekturlesen.

- und meiner Freundin Mildred für das Korrekturlesen und die liebevolle psychologische Betreuung, die mich vor einem sicheren Nervenzusammenbruch bewahrt hat.

16.Literaturverzeichnis

DOE98	Dietrich Dörner, Harald Schaub, *„Das Leben von Psi“,* Otto-Friedrich-Universität Bamberg, 1998 Internet: http://www.uni-bamberg.de/ppp/insttheopsy/projekte/psi/ (10.2.2005)
DOE99	Dietrich Dörner, *„Bauplan für eine Seele“,* Reinbek bei Hamburg: Rowohlt 1999. ISBN: 3-498-01288-6
BAC03	Joscha Bach, *„The MicroPsi Agent Architecture“,* Proceedings of ICCM-5, Universitäts-Verlag Bamberg, 2003
BAC+03	Joscha Bach, Ronnie Vuine, *„The AEP Toolkit for Agent Design and Simulation“,* MATES 2003, LNAI 2831, pp. 38-49, Springer-Verlag Berlin, Heidelberg 2003 Internet: http://www.cognitive-agents.org/publications.php (2.6.2005)
VUI+03	Ronnie Vuine, Joscha Bach, *„The Artificial Emotion Project Handbook Rev. 0.5b“,* 2003 Internet: http://www.cognitive-agents.org/publications.php (2.6.2005)
BAC+05	Joscha Bach, Ronnie Vuine and others, „*The MicroPsi Project - building cognitive agents“,* Project Homepage Internet: http://www.cognitive-agents.org/ (2.7.2005)
DET99	Frank Detje, *„Insel. Programmbeschreibung (Stand Jan. 2002)“,* Institut für Theoretische Psychologie, Universität Bamberg, 1999 Internet: http://www.uni-bamberg.de/ppp/insttheopsy/dokumente/Detje_Insel_Programmbeschreibung.pdf (10.2.2005)
DET00	Frank Detje, *„Insel. Dokumentation Versuche Ergebnisse (Memorandum 39)“*, Institut für Theoretische Psychologie Otto-Friedrich-Universität Bamberg, 2000 Internet: http://www.uni-bamberg.de/ppp/insttheopsy/dokumente/Detje_Insel_-_Dokumentation_-_Versuche_-_Ergebisse_(Memo39).pdf (10.2.2005)
GER+01:	Jürgen Gerdes, Frank Detje, „*Aufbau einer komplexen Umwelt für den Multi-Agenten-Betrieb und Hybridgesellschaften“*, Institut für Theoretische Psychologie Otto-Friedrich-Universität Bamberg, 2001
HÄM03:	Viola Hämmer, *„Psi Reality 3D - Instruktionen“*, Institut für Theoretische Psychologie Otto-Friedrich-Universität Bamberg, 2003
GER+03:	Jürgen Gerdes, Viola Hämmer, „*Psi Reality 3D –Programmdokumentation“*, Institut für Theoretische Psychologie Otto-Friedrich-Universität Bamberg, 2003
GER+99	Jürgen Gerdes, Maja Dshemuchadse, Dietrich Dörner *„Face - Das Gesicht von Psi“*, 1999 Internet: http://giftp.ppp.uni-bamberg.de/projekte/psi/face/ (2.6.2005)
SAL04	David Salz, *„3DView - Eine dreidimensionale Visualisierung der MicroPsi-Multiagentenplattform“,* Studienarbeit, Humboldt-Universität zu Berlin, 2004 Internet: http://www-agents.org/webdoc/space/3dviewer (11.4.2005)
WIE+04	Christian Wiech, Tino Naphtali, *„3DEmotion“,* 2004 Internet: http://www.cognitive-agents.org/micropsi/space/3demotion (11.4.2005)

MS05 *„Microsoft DirectX SDK“*, Microsoft Corp., Redmondt, WA, USA, 2005
Internet: http://msdn.microsoft.com/directx (24.6.2005)

GRE02 Jason Gergory, *„Animation in Video Games“*, Presentation at University of Southern California, 2002
Internet: www-scf.usc.edu/~gamedev/animation.ppt (11.4.2005)

SHO85 Ken Shoemake, *„Animating rotation with quaternion curves“,*
Proceedings of the 12rd annual conference on Computer graphics and interactive techniques, pp. 245-254, 1985

EBE01 David H. Eberly, *„3D Game Engine Design – A Practical Approach To Real-Time Computer Graphics“,* Academy Press, San Diego, USA , 2001

HOP96 Hugues Hoppe (Microsoft Research), *„Progressive Meshes“,* Proceedings of ACM SIGGRAPH 1996, 1996, pp. 99-108
Internet: http://research.microsoft.com/~hoppe/pm.pdf (26.3.2005)

DUCH+97: Mark Duchaineau et al., *„ROAMing Terrain: Real-time Optimally Adapting Meshes“*,
Proceedings of the 8th conference on Visualization '97, Phoenix, Arizona, USA, 1997, pp. 81-88
Internet: http://www.llnl.gov/graphics/ROAM/ (10.2.2005)

TUR00: Turner, *„Real-Time Dynamic Level of Detail Rendering with ROAM“*, Gamasutra, 2000
Internet: http://www.gamasutra.com/features/20000403/turner_01.htm (10.2.2005)

BOE00: Willem H. de Boer, *„Fast Terrain Rendering Using Geometrical MipMapping“*, Flipcode, 2000
Internet: http://www.flipcode.com/articles/article_geomipmaps.shtml (10.2.2005)

LIN+96: Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faus *„Real-Time, Continuous Level of Detail Rendering of Height Fields“*,
Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pp. 109-118, 1996
Internet: http://www.gvu.gatech.edu/people/peter.lindstrom/papers/siggraph96/ (21.2.2005)

ULR02: Thatcher Ulrich, *„Rendering Massive Terrains using Chunked Level of Detail Control (DRAFT)“*, „Super-size it! Scaling up to Massive Virtual Worlds“ course, SIGGRAPH 02, 2002
Internet: http://www.tulrich.com/geekstuff/chunklod.html (5.12.2004)

PEA01: Mark Peasley, *„Tiled Terrain“*, Gamasutra, 2001
Internet: http://www.gamasutra.com/features/20011024/peasley (1.12.2004)

BLO00: Charles Bloom, *„Terrain Texture Compositing by Blending in the Frame-Buffer (aka. 'Splatting')“*, 2000
Internet: http://www.cbloom.com/3d/techdocs/splatting.txt (1.12.2004)

RAD05 Nebula2, Radon Labs Gmbh, Berlin, 2005
Internet: http://www.radonlabs.de/nebula.html (7.4.2005)

FOS96 Nick Foster, Dimitri Metaxas, *„Realistic Animation of Liquids“,* in *„Graphical Models and Image Processing: GMIP“*, Academic Press, 1996 v.58, n.8, pp. 471-483
Internet: http://www.cis.upenn.edu/~fostern/pdfs/gmip96.pdf (21.2.2005)

TES01 Jerry Tessendorf, *„Simulating Ocean Water“,*
SIGGRAPH 2001 Course Notes, 2001

MAT05 Daniel Matzke, *„3D-Visualisierung von Wasser – eine Betrachtung von Simulations-, Animations- und Shadingalgorithmen unter Echtzeitaspekten“,*
Diplomarbeit, Humboldt-Universität zu Berlin, 2005

JEN01 Lasse Staff Jensen, Robert Golias, *„Deep-Water Animation and Rendering“,*
Gamasutra, 2001
Internet: http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm (21.2.2005)

VLA02 Alex Vlachos, John Isidoro, Chris Oat, *„Rippling Reflective and Refractive Water“,* in *„ShaderX: Vertex and Pixel Shader Programming Tips and Tricks“*, edited by Wolfgang F. Engel, Wordware Publishing Inc. Plano, Texas, 2002 also published online by ATI Technologies Inc. Markham, Ontario, Canada
Internet: http://www.ati.com/developer/shaderx/ (21.2.2005)

SEG92 Mark Segal et al, *„Fast shadows and lighting effects using texture mapping“*, Proceedings of SIGGRAPH 1992, 1992, pp. 249-252

EVE01b Cass Everitt, *„Projective Texture Mapping“,* NVIDIA Corp., Austin, Texas, 2002
Internet: http://developer.nvidia.com/object/Projective_Texture_Mapping.html (26.6.2005)

LOM04 Yann Lombard, *„Realistic Natural Effect Rendering: Water I“*, GameDev.net, 2004
Internet: http://www.gamedev.net/reference/articles/article2138.asp (26.6.2005)

VOL97 J. D. Folay, A. van Dam, S. K. Feiner, J. F. Hughes, *„Specular Reflection“* in *„Computer Graphics Priciples and Practice“,* Addision-Wesley Publishing Company, New York, Amsterdam, Sydney, Tokyo, Madrid, Paris, Bonn 1997, pp. 730, 756

CAT74: Edwin Catmull, *„A subdivision algorithm for computer display of curved surfaces“*, PhD Thesis, Computer Science Dept., University of Utah, 1974

STA+02 Mark Stamminger, George Drettakis, *„Perspective Shadow Maps“*, Proceedings of SIGGRAPH 2002, 21(3), 2002, pp. 557-562
Internet: http://www-sop.inria.fr/reves/personnel/Marc.Stamminger/psm/ (10.2.2005)

RIG02: Guennadi Riguer, *„Performance Optimization Techniques for ATI Graphics Hardware with DirectX9.0“*, ATI Technologies Inc., 2002
Internet: http://www.ati.com/developer/dx9/ATI-DX9_Optimization.pdf (9.2.2005)

FAU03: Kasper Fauerby, Carsten Kjaer, *„Real-time Soft Shadows in a Game Engine“*, 2003
Internet: www.peroxide.dk/papers/realtimesoftshadows.pdf (11.2.2005)

CRO77	Frank C. Crow, *„Shadow algorithms for computer graphics“*, Proceedings of SIGGRAPH 1977, 11 (2), 1977, pp. 242-248
HEI91	Tim Heidmann, *„Real Shadows Real Time“*, IRIS Universe, Number 18, 1991, pp. 28-31 Internet: http://developer.nvidia.com/object/robust_shadow_volumes.html (10.2.2005)
EVE+02	Cass Everitt, Mark J. Kilgard, *„Practical and Robust Stenciled Shadow Volumes for Hardware Accelerated Rendering“*, NVIDIA Corp., Austin, Texas, 2002 Internet: http://developer.nvidia.com/object/robust_shadow_volumes.html (10.2.2005)
TSI04	John Tsiombikas, *„Volume Shadows Tutorial“,* GameDev.net, 2004 Internet: http://downloads.gamedev.net/pdf/VolumeShadowsTutorial-2036.pdf (10.2.2005)
DIE96	Paul Diefenbach, *„Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions“,* Ph.D. Thesis, University of Pennsylvania, 1996
CAR00	John Carmack, Unpublished Personal Correspondence, 2000 Internet: http://developer.nvidia.com/object/robust_shadow_volumes.html (10.2.2005)
MS04a	Microsoft Corp., *„ShadowVolume Sample“*, Microsoft DirectX 9.0 SDK Update (Summer 2004), 2004 Internet: http://msdn.microsoft.com/directx/ (10.11.2004)
WIL78	Lance Williams, *„Casting curved shadows on curved surfaces“*, 1978 Proceedings of SIGGRAPH 1978, 112 (3), 1978, pp. 270-274
EVE+01	Cass Everitt, Ashu Rege, Cem Cebenoyan, *„Hardware Shadow Mapping“*, NVIDIA Corp., 2001 Internet: http://developer.nvidia.com/object/hwshadowmap_paper.html (14.2.2005)
MS04b	Microsoft Corp., *„Shadow Map Sample“*, Microsoft DirectX 9.0 SDK Update (December 2004), 2004 Internet: http://msdn.microsoft.com/directx/ (14.2.2005)
EVE01	Cass Everitt, *„Shadow Mapping“,* Presentation at the Game Developer Conference (GDC) 2001, 2001

17.Abbildungsverzeichnis

18.Glossar

Ambient Light / Ambientes Licht

Eine Fläche kann direktes und indirektes Licht erhalten. In der Realität reflektiert beinahe jede Fläche, die direktes Licht erhält, dieses und beleuchtet somit andere Flächen, die kein direktes Licht erhalten. In der Computergraphik ist die Simulation indirekter Beleuchtung extrem aufwändig, daher wird insbesondere im Echtzeitbereich häufig darauf verzichtet und statt dessen werden alle Flächen pauschal mit einem so genannten Ambienten Licht beleuchtet, das keine spezielle Richtung besitzt.

Backface

Ein dem Betrachter abgewandtes, also normalerweise nicht sichtbares, Polygon. Siehe *Backface-Culling*.

Backface-Culling

Polygone werden normalerweise als einseitig betrachtet, d.h. sie werden nur gezeichnet, wenn ihre Vorderseite (Frontface) dem Betrachter zugewandt ist. Andernfalls werden sie automatisch aussortiert, dieses wird als Backface-Culling bezeichnet. Der Cullingmodus lässt sich verändern, so dass auf Wunsch statt den Frontfaces die Backfaces oder sowohl Front- als auch Backfaces gezeichnet werden können.

Clipping

Clipping bezeichnet das Abschneiden von Geometrie an einer Ebene. Beispielsweise wird Geometrie von der GPU automatisch an den Ebenen des View Frustums geclippt, auf den meisten Karten lassen sich vom Benutzer auch individuelle Clipping-Ebenen einstellen.

Cube Map

Ein spezielles Texturformat, das eigentlich aus sechs gleich großen Texturen besteht, diese Texturen werden als die Flächen eines Würfels interpretiert. Es stehen spezielle Befehle zum Zugriff auf Cube Maps zur Verfügung.

Culling

Culling bezeichnet in der Computergraphik das komplette Ausschließen von Geometrie vom Rendering. Siehe Backface-Culling und View Frustum Culling.

FPS

Abkürzung für Frames Per Second (Bilder pro Sekunde).

Frame

Ein einzelnes gerendertes Bild.

Frame Buffer

Ein Speicherbereich, in den das Bild gerendert wird. Sobald der Frame Buffer komplett geschrieben ist, wird er auf dem Monitor als Bild angezeigt.

Frontface

Ein dem Betrachter zugewandtes, also normalerweise sichtbares, Polygon. Siehe *Backface-Culling*.

GPU

Graphics Processing Unit. Die Prozessoren moderner Graphikkarten sind von der Komplexität mit CPUs vergleichbar. Sie sind programmierbar und CPUs durch in Hardware implementierte Algorithmen und massive Parallelverarbeitung bei der Graphikdarstellung weit überlegen.

High Order Surfaces / Curved Surfaces

Eine Technik, bei der geometrische Rundungen nicht durch eine feste Anzahl von Polygonen, sondern durch ein mathematisches Modell beschrieben werden. Aufgrund dieses Modells kann die Hardware die Rundung durch automatisch erzeugte Polygone beliebig fein Auflösen. Dadurch wird Speicherplatz und Bandbreite auf dem BUS gespart, außerdem kann die Auflösung der Geometrie dynamisch den aktuellen Erfordernissen angepasst werden.

Indexbuffer

Ein Indexbuffer enthält Indizes in einen Vertexbuffer. Der Indexbuffer gibt an, wie die Vertices im Vertexbuffer zu Dreiecken zu kombinieren sind. Der Vorteil dabei ist, dass ein Indexbuffer einen Vertex beliebig oft referenzieren kann, ohne dass der Vertex tatsächlich dupliziert werden muss. Auch können mehrere Indexbuffer den gleichen Vertexbuffer verwenden.

Kamera

Hier: die Position, Blickrichtung, Sichtwinkel und Sichtweite des Betrachters in der virtuellen Welt. Eine Kamera definiert einen Sichtkegel, der sich durch eine Transformationsmatrix beschreiben läßt.

LOD

Abkürzung für Level of Detail. Bezeichnet Verfahren, bei denen Daten oder Prozesse aus Effizienzgründen situationsabhängig vereinfacht werden. Beispielsweise werden in der dreidimensionalen Computergraphik oft vereinfachte 3D-Modelle und kleinere Texturen verwendet, je weiter ein Objekt vom Betrachter entfernt ist. Es gibt stufenweise und stufenlose Algorithmen (continuous LOD).

Mipmapping

Wenn eine Textur auf Grund der Entfernung zur Kamera auf dem Bildschirm sehr klein skaliert dargestellt wird, kann ein unangenehmes Flackern auftreten, da schon sehr geringe Änderungung der Kameraposition dazu führen, dass andere Pixel aus der Textur dargestellt werden. Das Filtern der Textur bei der Darstellung kann dieses Problem nur bedingt lösen, da nur eine begrenzte Anzahl von Pixeln in jede Richtung beim Filtern berücksichtigt werden können. Daher erzeugen Graphikkarten von jeder Textur eine Reihe von (mit aufwändigeren Algorithmen) herunterskalierten Versionen, auch Mipmaps genannt. Jede Mipmap hat die halbe Auflösung der vorhergehenden Stufe. Mit steigender Entfernung zwischen Kamera und Textur wird statt der Originaltextur eine der Mipmaps verwendet.

Multitexturing

Bezeichnet die Fähigkeit einer Graphikhardware, ein Polygon gleichzeitig mit mehreren verschiedenen Texturen zu versehen. Die einzelnen Texturen werden dabei mit einer arithmetischen Operation, beispielsweise additiv oder multitplikativ, verknüpft.

Pixel Shader

Pixel Shader sind Programme, die auf der GPU genau einmal pro Pixel ausgeführt werden. Sie können beispielsweise Lichtberechnungen pixelgenau durchführen.

Rendering, rendern

Mit Rendering wird in der Computergraphik der Prozess der Bilderzeugung bezeichnet.

Self Shadowing

Die Fähigkeit eines Objektes, Schatten auf sich selbst zu werfen. Nicht alle Algorithmen zur Schattendarstellung ermöglichen dieses.

Stencil Buffer

Speichert für jeden Pixel auf dem Bildschirm einen Integer-Wert. Zeichenoperationen können einerseits Werte in den Stencil Buffer schreiben, andererseits kann aufgrund von Stencilwerten entschieden werden, ob bestimmte Pixel gezeichnet werden oder nicht, d.h. der Stencil Buffer dient als Maske beim Beschreiben des Frame Buffers.

Textur

Eine Textur ist eine Bitmap, die von der Graphikhardware ganz oder teilweise auf ein Polygon projiziert werden kann (Texture Mapping). Durch diese Technik können dreidimensionale Modelle mit einer Bitmaphaut überzogen werden.

Texturkoordinaten

Vertices können Texturkoordinaten haben. Sie ordnen einen Vertex einen Punkt in der Textur zu. Durch die Texturkoordinaten aller Eckpunkte eines Polygons ergibt sich, wie genau die Textur auf das Polygon projiziert wird.

Vertex

Ein Vertex (Plural: Vertices) ist ein Eckpunkt eines 3D-Modelles. Vertices spannen die Flächen (Polygone bzw. Dreieicke) auf, aus denen das Modell besteht. Zu einem Vertex können eine Vielzahl von Informationen gehören, z.B. räumliche Koordinaten (x,y,z), ein Normalenvektor, verschiedene Farbinformation, verschiedene Texturkoordinaten usw.

Vertexbuffer

Ein Vertexbuffer ist ein Array von Vertices, das der Graphikkarte zum Zeichnen übergeben wird. Die Vertices definieren Dreiecke, die von der GPU gezeichnet werden können. Es gibt verschiedene Möglichkeiten, die Vertices im Buffer zu Dreiecken zusammenzufassen. Ein Vertexbuffer kann direkt oder unter Verwendung eines Indexbuffers gezeichnet werden.

Vertex Shader

Ein Programm, dass pro Vertex auf der GPU ausgeführt wird. Ein Vertex Shader erhält pro Durchlauf als Eingabedaten genau einen Vertex und eine Reihe von durch die Applikation definierten Parametern. Ausgabedatum ist genau ein Vertex, der dann zum Rendern verwendet wird. Typischerweise transformieren Vertex Shader Vertices vom Objekt- in den Clip Space und berechnen die Beleuchtung, aber es sind auch viele andere Verwendungen möglich.

View Frustum / View Frustum Culling

Der Begriff View Frustum bezeichnet den von der Kamera erfassten Sichtkegel. Genau genommen handelt es sich um einen Pyramidenstumpf, der durch sechs Ebenen definiert werden kann. An diesen Ebenen findet durch die Graphikkarte ein automatisches Clipping statt. In den meisten Fällen ist es effizienter, wenn bereits die Applikation registriert, welche Geometrie sich ausserhalb des View Frustums befindet und diese Geometrie nicht zum Rendering an die GPU übergibt. Dieses wird dieses als View Frustum Culling bezeichnet.

Z-Buffer

Ein Z-Buffer speichert für jeden Pixel auf dem Bildschirm einen Tiefenwert, also die Entfernung des Punktes zur Kamera, ab. Anhand dieser Tiefenwerte kann für später gezeichnete Geometrie entschieden werden, einzelne Pixel gezeichnet werden dürfen oder von vorhandener Geometrie verdeckt würden.

Z-Fighting

Ein Darstellungsfehler, bei dem sich Flächen scheinbar durchdringen und zu flimmern beginnen, obwohl eine von der anderen verdeckt werden müsste. Entsteht durch die begrenzte Auflösung des Z-Buffers.

Anhang A – 3DView2-Anleitungstexte

I. Vorbemerkung zu den Anleitungstexten

Die in diesem Anhang abgedruckten Texte richten sich an MicroPsi-Entwickler und erfahrene Benutzer. Im folgenden wird beschrieben, wie neue Objekte und Animationen für 3DView2 erstellt und in das Programm eingebunden werden können, wie neue Terrains und Umgebungen erstellt werden und wie 3DView2 die Weltdaten des MicroPsi-Server visualisiert.

Diese Texte wurden ursprünglich auf dem MicroPsi-Projekt-Wiki publiziert* und später in das MicroPsi-Entwicklerhandbuch aufgenommen, welches ebenfalls auf der Projekthomepage zum Download bereit steht. Dort werden auch in Zukunft aktualisierte Fassungen der Anleitungstexte erscheinen. Die wichtigsten Texte sind an dieser Stelle in der momentan aktuellen Version abgedruckt.

Da MicroPsi mittlerweile auch international verwendet wird und die Texte für die Publikation im Internet geschrieben wurden, ist der folgende Teil dieses Anhangs auf Englisch verfasst.

*http://www.cognitive-agents.org/micropsi/space/3dviewer

II. Using the 3DViewer

1. Running 3DView2

There are two ways to install 3DView2.

A release version is available through the Eclipse Software Update mechanism, just like the rest of the microPSI toolkit (see installation). If you are a microPSI user then this is the right choice for you.

If you are a developer and have access to our CVS server you can always get an up-to-date version (or even the source code) from there. In fact, if you work with the CVS version of microPSI, you should also work with CVS version of 3DView2. There is always a precompiled version available, there is no need to compile 3DView2 yourself unless you absolutely want to.

Downloading 3DView2 from the CVS

- Checkout the module 'org.micropsi.3dview2' from the CVS (this contains the complete release version of 3DView2)

- If you want to integrate 3DView2 into Eclipse, use Eclipse to checkout the modules! Otherwise use any CVS program you like.

3DView2 can be used as a stand alone program. If you do not want integration into Eclipse, you can stop right here and move on to the next section.

- Checkout the module 'org.micropsi.3dviewer' in Eclipse (this contains code that integrates 3DView2 into Eclipse)

- In the Eclipse view of the project 'org.micropsi.3dviewer' there should be a folder named 'build'. Inside that folder, there is a file named 'build.xml'. Left click on that file, choose 'Run', then 'Ant build'.

Starting 3DView2

- To launch 3DView2 as a stand-alone application, run bin3dview2.exe in the 3dview2 folder.

- It is also possible to run 3dview2 in a separate window from within Eclipse. Go to the World Perspective and click the 3dviewer button in the tool bar.

- Finally, 3DView2 can be completely integrated into the Eclipse GUI by running it as a View. Go the the World Perspective and select *"Window -> Show View -> Other ... -> MicroPsi World Console -> 3DView"* from the menu.

III.Graphics and Content Creation

1.Creating 3D Models and Animations for 3DView2

3D Objects and Animation for 3DView2 can be created using standard 3D modeling programs. The most well-known professional programs are probably "Maya" by Alias Wavefront and "3D Studio Max" by discreet. There are also a number of freeware and open source programs that are suitable for the task but for the remainder of this article we will focus on the major applications.

Internally, 3DView2 uses the Microsoft X-File format (*.x), so all models must be converted to X-Files before they can be used. X-File exporters are available for both Alias Wavefront Maya and discreet 3D Studio Max (and a few other modeling packages). A X-File always contains mesh data. It can (but does not have to) include animation data, both simple keyframe animation and skeletal animation ("skinning") are supported. It can, however, be tricky at times to export animations since not every exporter supports all features of a particular modeling program.

X-Files also contain material definitions, but you must be aware that materials as seen by 3DView2 and other real time applications are somewhat different from the materials you use in your modeling program. Every modeling program has a different approach to materials anyway. Any "fancy" shaders your program offers will usually not get exported, most exporters only support simple textures that must be available as bitmap files (jpg, png, tga or the like). More sophisticated effects are possible through DirectX shaders in the form of Microsoft FX-Files (.fx). Some modeling programs support FX-Files; for instance, 3D Studio MAX 7 does and Maya 5 (or better) does with the help of a plugin by Microsoft. Using FX-Files the object in your modeling program will appear exactly as it will in 3DView2. Most FX-shaders have parameters like textures, colors or ranges that you must set. Naturally, your X-File exporter must also support shader assignments and parameters.

The following Table summarizes the features of the most popular exporters:

name	modeling program(s)	file name	version	skinning	shader parameters	web
Pandasoft	3DStudio MAX6, 3DStudio MAX7	PandaDXExport6.dle	4.6.0.57 (11/20/2004)	physique modifier, skin modifier	yes	http://www.andytather.co.uk/Panda/directxmax_downloads.aspx
Quest3D (Q3D, improved Microsoft Exporter)	3DStudio MAX6, 3DStudio MAX7	XSkinExp.dle	10/31/2003	physique modifier only	no	http://www.quest3d.com/index.php?id=102
TiburonGX Exporter	3DStudio MAX6	?	0.7	physique modifier, skin modifier	?	http://www.tiburongx.com/new_release.htm
Microsoft DirectX-SDK Exporter	Maya 5	D3DMaya5ExportX.mll	Feb 2005 SDK	yes	yes	http://msdn.microsoft.com/directx/ (download SDK)
Microsoft DirectX-SDK Exporter	Maya 6	D3DMaya6ExportX.mll	Feb 2005 SDK	yes	yes	http://msdn.microsoft.com/directx/ (download SDK)

The best exporter for 3D studio MAX is the **Pandasoft Exporter**. The best exporter for Maya is the one available from the Microsoft **DirectX SDK**.

General Exporting Guidelines

- The exporter tools export the **hierarchy** exactly as it is in the modeling program. (In Maya the hierarchy is displayed in the Outliner or Hypergraph, in 3DS Max the same feature is called Schematic View). While it is a good idea to have a well-structured hierarchy while working on the model it is usually not a good idea to export the hierarchy like this. The reason is that every mesh group in the modeling program leads to a so-called *vertex group* in 3DView2. On most systems application performance largely depends on the number of vertex groups and not so much on the number of polygons. Therefore, you should merge all groups that use the same material (or texture) before exporting; this will greatly improve performance.
- The number of textures a model uses also has an impact on performance. It is better to have few large textures than a lot of smaller ones. Combine and reuse textures wherever possible.
- If the **orientation of an object** matters, make sure that the 'front' points towards the positive x axis in your modeling program

Exporting Models and Animations with 3DStudio MAX

- The default unit in 3DS MAX is 'meters'. This often results in exported objects being extremely large in 3DView2; you will need to scale them down before exporting. It is advisable to set your unit scale to 'centimeters' when working on models for 3DView2 - that way the size in MAX and the 3D engine will match. Go to *Customize - Units Setup...* to do this.
- Scaled transform nodes cause trouble with 3DView2. Therefore, you will need to reset all transformations before exporting. Use the 'Reset XForm' command to do this.
- Only polygon data gets exported. Before exporting you will need to triangulate your complete scene.
- All known exporters have trouble with transformed bones, e.g. mirrored ones. You will need to reset these transformations before exporting.

The Pandasoft Exporter for 3D Studio MAX

The Pandasoft Exporter is available for 3D Studio Max starting with version 3. The following settings are recommended when exporting models or animations for 3DView2:

- Output Options:
  - Mesh Definition, Materials
  - Optimize Mesh = Normal or Optimized
  - Include Animation, Bones (if needed)
- Object Types
  - Geometric, Dummy
- Mesh Options
  - Mesh Normals
  - Mapping Coordinates
- Animation (if needed)
  - Key Options: Matrix
  - Timeline: 3DS Max ticks
- .fx Effect Files
  - Include .fx file
  - Include .fx parameters
- DX File Type
  - Binary with Compression
- DX Frame
  - Sub Frame Hierarchy (does not matter if you use skinning)

Exporting Models and Animations with Maya

- Scaled transform nodes cause trouble with 3DView2. Therefore, you will need to reset all transformations before exporting. Use the 'Freeze Transformations' command to do this.
- Only polygon data gets exported. Before exporting you will need to triangulate your complete scene.

2.Creating Terrains for 3DView2

Terrains for 3DView2 consist of

- a height map
- a material map
- a number of terrain parameters
- a number of terrain textures

The terrain textures are defined in the visualization. A visualization is like a skin for 3DView2, it is described in an xml file defines the look of the world. Any number of terrains can use the same visualization and will therefore use the same terrain textures and have a similar look. You can also use the same terrain with different visualizations. For general information about visualizations see Adding Things to 3DView2; the definition of terrain textures is covered later in this document.

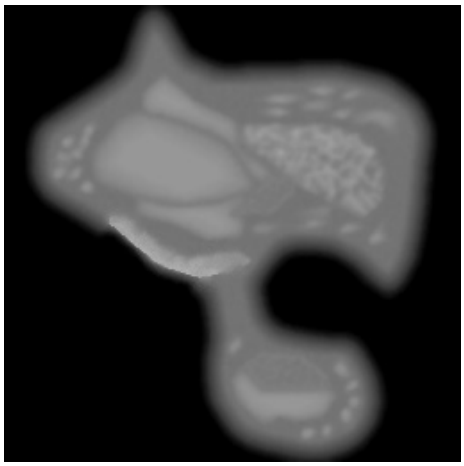
The Height Map

A height map is a grayscale bitmap image. The color information in the image is interpreted as height data. The brighter a pixel is the higher is the terrain in that location. Consider a height map a top-down view of your terrain where mountains and hills are bright pixels while valleys are represented by dark pixels.

Height Maps for 3DView2 must be square and the edge length should be a power of two plus one, e.g. 129, 257, 513 or 1025. If these requirements are not met the bitmaps is automatically converted and resized accordingly; but this can lead to small unwanted artifacts and distortions.

Height Maps can be in almost any image format. JPG creates very small files which are ideal for people with a slow network connection, but the compression is lossy. PNG has lossless compression but the files are a good deal larger.

Keep in mind that larger maps are more demanding in terms of processing power and memory. It is not advisable to create maps much larger than 2049 pixels unless the machines you are targeting are very powerful.



The Material Map

A material map is a palette color bitmap image. The color information in the image is interpreted as a map of different ground materials.

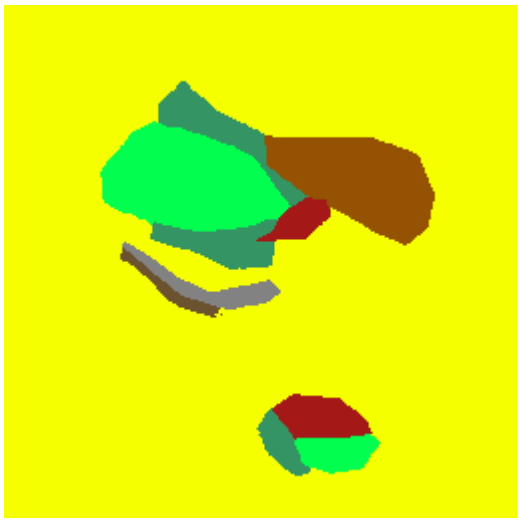
Material Maps for 3DView2 should be square and the edge length must be the same as the height map minus 1 pixel. For instance, if your height map is 1025x1025 pixels, your material map should have a size of 1024x1024 pixels. If this condition is not met, the image is resized automatically but this can of course lead to unwanted artifacts in the image.

The material map **must** be a **palette image**. That means that it uses a palette of 256 colors and stores only a color index per pixel instead of storing individual color values per pixel. RGB color images will not work! A typical color palette image format is GIF. PNG supports palettes as well but is a bit more tricky since it also supports RGB and RGBA formats. All image processing programs should be able to handle GIF images and allow you to edit the color palette. GIF also has very nice lossless compression.

Every pixel in the material map represents a small portion of the terrain. The color index of the pixel defines the terrain type, i.e. the ground texture that is applied to this portion of the terrain. So color index 0 is terrain type 0, color index 1 terrain type 1 and so on. What the different terrain types are and how many different terrain types are available is defined by the visualization (see below). The actual color values (the palette colors) are not used. Therefore you can define the palette colors in a way that is convenient for editing. For instance, if color index 0 symbolized terrain type 'grass' then you could define palette entry 0 as green. Again, 3DView2 does not use this color.

3DView2 automatically creates soft transitions between different ground materials. There is no way you can influence these transitions; the system is designed for ease of use which means that artistic freedom has to be limited somewhat.

If a material map uses more terrain types than are defined by the visualization the excessive ones are displayed with a pink terrain texture in 3DView2.



Creating a Map File

There are two types of world map files: those for online and those for offline use. Online world files are used when you connect to a MicroPsi server. Offline files can be loaded and displayed without a server connection. The file format for both types is basically the same; there are some minor differences that will be covered in a moment.

World map files are xml files. The following is a typical example:

```
<world>
    <description>The default island.</description>
    <defaultvisualization>standard.xml</defaultvisualization>
    <terrainmap>
        <heightmap>square_128.jpg</heightmap>
        <materialmap>square_material_128.gif</materialmap>
        <offset>
            <x>-18.000000</x>
            <y>-2.000000</y>
            <z>-78.000000</z>
        </offset>
        <scaling>
            <x>0.750000</x>
            <y>0.040000</y>
            <z>0.750000</z>
        </scaling>
        <observerstartpos>
            <x>-11.000000</x>
            <y>13.500000</y>
            <z>1.200000</z>
        </observerstartpos>
        <observerlookat>
            <x>14.000000</x>
            <y>3.050000</y>
            <z>-18.000000</z>
        </observerlookat>
        <wraparound>false</wraparound>
    </terrainmap>
    <objects />
</world>
```

The *description* tag contains a textual description of the map file; it is used by the GUI. *defaultvisualization* is the name of the visualization xml file this map will use by default; a different visualization may be selected at run-time. *heightmap* and *materialmap* are the file names of the height map and material map; both files must reside in the *data/terrain* folder in the 3DView2 directory.

offset is the point in 3d world coordinates where the upper left corner of the height map will be. From there it will stretch in the direction of the positive x- and z-axis. Note that 3DView uses a different coordinate system that MicroPsi. In 3DView the positive y-axis points up while the z-axis ist the equivalent to the y-axis in MicroPsi. *scaling* defines the size of the map in 3d space. With a scaling of (1, 1, 1) a height map of 257x257 pixels will be 256x256 units (meters) in 3d space and the highest elevation possible (perfectly white pixel) will be 256 units high. With a scaling of (2, 1, 2) the map will be 512x512 meters with the maximum terrain height unchanged. *observerstartpos* and *observerlookat* specify the point in 3d space where the camera will initially be situated after loading the map and the point the camera will initially look at.

The *objects* tag contains world objects. You should use the built-in editor to place on the map, simply leave the tag empty.

Offline world files belong in the folder *data/offlineworlds* in the 3DView2 directory and can have arbitrary names.

Online world files belong in the folder *data/offlineworlds* in the 3DView2 directory. The names of online world files must match those of the terrain files on the server. The server currently uses bitmap files as terrain maps. For instance, when 3DView2 connects to a server that displays the terrain map "island.png" it will look for an online world file called "island.png.xml". The *objects* tag of an online world file should always be empty since the object list is received from the server. If the file does contain any objects they are simply ignored. As far as *offset* and *scaling* are concerned, only the y coordinate is actually taken from the xml file. The values for the other axis are matched with the servers settings. The world on the server is essentially only two-dimensional, that is why the xml file is still necessary for the y-axis settings.

Terrain Textures

As stated in the beginning, terrain textures are part of the visualization xml file. There is a default visualization called *standard.xml* which is a good starting point for creating your own visualizations.

Every visualization xml file must contain a *terrain* block. Here is an example:

```
<terrain>
    <mapdefaults> … </mapdefaults>
    <texturetilesize>4.0</texturetilesize>
    <materials>
        <material>
            <name>grass (0)</name>
            <texture>terrain_grass.tga</texture>
        </material>
        <material>
            <name>sand (1)</name>
            <texture>terrain_sand.tga</texture>
        </material>
        <material>
            <name>rock (2)</name>
            <texture>terrain_rock.tga</texture>
        </material>
        <material>
            <name>dense grass (3)</name>
            <texture>terrain_forestfloor.jpg</texture>
        </material>
    </materials>
</terrain>
```

The *mapdefaults* tag contains map information just like an offline world file. This is meant as a default in case a world does not specify a map file (which hopefully never happens). There is no reason to change anything inside this tag, simply take it over from the default visualization.

The *materials* tag defines the terrain types. The terrain types are assigned ID numbers in the order they appear in this file, starting with 0. So the first *material* tag is terrain type 0, the second terrain type 1, the third 2 and so on. Every material tag contains a name, which is an arbitrary string and only for information purposes, and a texture file. The texture file can be in any well-established RGB or RGBA image file format. Terrain textures should - like all other textures - be square and should have an edge length that is a power of two.

Terrain textures are tiled over the whole terrain so you must make sure that terrain textures are tileable, i.e. the upper border of the texture must match the lower border and the left border must match the right one. The textures are laid out on a rectangular grid. With the *texturetilesize* tag you can define the the size of that grid. This influences how much the textures are stretched - this is only a cosmetic issue, a little experimenting is necessary. It is important to understand that the value you set here is only a guiding value. The actual tile size must be aligned with the terrain chunk size; the terrain system will try to match the value you give as closely as possible.

3.Adding 3D Models and Object Types to 3DView2

Object Types are defined by the microPSI world server. 3DView2 connects to the server and receives a list of world objects including their positions and states. Every object has a **class name**. The class name is basically the object type, for instance 'Tree' or 'Mushroom'. In order to visualize the world with 3DView2 you must associate one or more 3D models with every object class. This is done in a so-called *visualization file* for 3DView2. The microPSI core system and the world server have no knowledge about the viewer or 3d models. The creation of 3d models is covered in Creating 3D Models and Animations for 3DView2

A visualization completely defines the look of the world in 3DView2. There can be more than one visualization. Think of visualizations as skins - there could, for instance, be a Middle European visualization where objects of type *Tree* appear as broad-leafed trees and a Caribbean visualization where the same object type looks like a palm tree.

Visualizations are defined in XML-Files. All visualization files are located in the folder *3dview2/data/visualizations*. Only one visualization file is used at a time. There is always a default visualization called *standard.xml*. If you want to create a new visualization it is a good starting point to copy this file and edit it as needed.

The basic structure of a visualization xml file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- microPSI 3Dview2 visualization file -->
<visualization>
    <description> … </description>
    <terrain> … </terrain>
    <skybox> … </skybox>
    <objects> … </objects>
</visualization>
```

The **description** tag contains a descriptive text, for instance “my very own Caribbean island visualization”. The terrain tag is not covered here, it is discussed in Creating Terrains for 3DView2. The *skybox* tag defines a model that represents the sky, the objects tag contains the definitions of all world objects.

Skybox

The **skybox** tag contains the name of a model file that is used as a sky box, for instance

```
<skybox>skybox.x</skybox>
```

The skybox model is different from all other models. It is the sky you see behind the terrain and all other objects. Typically, it is a box with a sky texture on the inside (therefore the name). The skybox model moves with the camera, so the viewer is always in the center of it, that means at (0,0,0) in the local coordinate system of the model. The skybox is rendered in a way so that it always appears to be in the background. It does not obscure any other objects but is obscured by everything else. The actual size of the skybox does not matter; it can be as small as 1x1x1 meter. Because of the special rendering it will appear to have an infinite size. The skybox can in fact be an arbitrary model, it does not have to be a box. It should, however, be a closed mesh. The viewer should be on the inside of the mesh and it sound not be possible to look out of it.

World Objects

The **objects** tag contains information how to visualize different **object** types. In the ideal case this section contains an entry for every object class the microPSI world server defines. If 3DView2 encounters an object with a type which has no such entry it will display the object with a default model, e.g. a small blue box with the text 'unknown object type' printed on it.

You can define an number of **variations** for every object class. Variations are alternative models that can be used to display a particular object. This is because a large forest, for instance, which consists of many objects of type 'Tree' would look pretty dull if all trees looked the same. Therefore, you can create a number of different tree models and 3DView2 will pick a random variation for every Tree object in the world.

For every variation you can define a number of **LOD levels**. LOD stands for 'Level of Detail'. In 3DView2 hundreds of objects can be visible at the same time and 3d models can become very complex. For performance reasons it is often necessary to reduce the visual detail of objects that are far away from the camera, i.e. you can use a very detailed model for an object if it is close to the camera but use a very simple one if the object is far away and therefore appears only very small on the screen anyway. 3DView2 allows you to define a number of LOD models for every variation of an object. For every LOD level you define a maximum distance in meters where this model can be used. Let us have a look at a few examples:

```
    <object>
        <classname>PalmTree</classname>
        <variation>
            <lodlevel>
                <model>bananatree.x</model>
            </lodlevel>
        </variation>
    </object>
```

This is the easiest way to define an object, it contains only one variation with only one LOD level. This piece of xml code says that objects of type *PalmTree* will always be displayed with model file *bananatree.x*. Note that the LOD level in this example does not contain a distance limit, the limit it therefore considered to be infinite.

```
    <object>
        <classname>Grass</classname>
        <variation>
            <lodlevel>
                <model>grass.x</model>
                <maxdistance>8.0</maxdistance>
            </lodlevel>
            <lodlevel>
                <model>grass_reduced.x</model>
                <maxdistance>16.0</maxdistance>
            </lodlevel>
            <lodlevel>
                <model>grass_billboard.x</model>
            </lodlevel>
         </variation>
    </object>
```

This second example contains two LOD levels. Objects of type *Grass* will use model *grass.x* up to a distance of 8.0 meters from the viewer. Between 8.0 and 16.0 meters the application will use *grass_reduced.x*. Beyond 16.0 meters *grass_bill.x* will be used. If more than one LOD level exists for an object 3DView2 will always use the one with the *smallest acceptable limit*. The order in which you specify the LOD levels in the xml file does *not* matter. Again, there is one LOD level in the example that has no limit (i.e. an infinite limit). This does not have to be the case.

```
<object>
    <classname>FlyAgaric</classname>
    <variation>
        <lodlevel>
            <model>element_fliegenpilz_a.x</model>
            <maxdistance>100.0</maxdistance>
        </lodlevel>
    </variation>
</object>
```

In this third example there is only one LOD level but it has a limit of 100.0 meters. That means that objects of type *FlyAgaric* will not be visible beyond a distance of 100.0 meters. They will completely disappear which saves performance and is acceptable for smaller objects.

Finally, here's an example of a tree object which defines three variations with two levels of detail each:

```
<object>
    <classname>PineTree</classname>
    <variation>
        <lodlevel>
            <model>element_nadelbaum_a.x</model>
            <maxdistance>150.0</maxdistance>
        </lodlevel>
        <lodlevel>
            <model>element_nadelbaum_a_bill.x</model>
        </lodlevel>
    </variation>
    <variation>
        <lodlevel>
            <model>element_nadelbaum_b.x</model>
            <maxdistance>150.0</maxdistance>
        </lodlevel>
        <lodlevel>
            <model>element_nadelbaum_b_bill.x</model>
        </lodlevel>
   </variation>
   <variation>
       <lodlevel>
           <model>element_nadelbaum_c.x</model>
           <maxdistance>150.0</maxdistance>
       </lodlevel>
       <lodlevel>
           <model>element_nadelbaum_c_bill.x</model>
       </lodlevel>
   </variation>
</object>
```

IV.Programmers Guide to 3DView2 and 3DEmotion

1.3DView2 Workspace Setup

What you need

In order to compile 3DView2 or 3DEmotion you need:

- **Microsoft Visual Studio 7.1** (aka Visual Studio .NET 2003) or better (other compilers might work, too, but I cannot help you with that.)
- The latest version of the **Microsoft DirectX SDK** installed. (DirectX 9 SDK, June 2005 Update at the time I write this. Later version should work, too.). The SDK is available here: http://msdn.microsoft.com/directx.
- The EAX 2.0 SDK by Creative Labs installed; it is needed by the sound system. Download: http://developer.creative.com/articles/article.asp?cat=1&sbcat=31&top=71&aid=138
- The current Java SDK installed if you plan to compile the Eclipse plugin. It is available from http://java.sun.com (Without the Java SDK, you will still be able to compile the standalone version of 3DView.)

I assume you have at least a basic understanding of C++, Microsoft Windows Programming and Microsoft DirectX. If not, there are excellent tutorials out there. It is not that difficult, really :)

Getting the source code

Grab the project org.micropsi.3dview2.code from our CVS. (Careful: it's huge! About 70MB at the time I write this, but chances are it has grown since then :)) It is split into several subprojects witch reside in separate folders. See *3DView2 Project Structure* for details.

Compiling

In the folder *3dview2/code/3dview2*, there should be a Solution File called *3dview_with_libs.sln*. The solution file for 3demotion is *3demotion_with_libs.sln* in the folder *3dview2/code/3demotion*. Open one of these files with Visual Studio and *Batch Build* all targets and all configurations. You're done!

Setting the Working Directory

You must set the working directory for both 3dview2.exe and 3demotion.exe to *../../bin* in MS Visual Studio (under *project properties --> debugging*). Otherwise you will get error messages (file not found etc.) when running the applications from within Visual Studio.

2.3DView2 Project Structure

The following projects belong to the 3DView2 source code:

- **BaseLib**(*) - contains base functionality and utilities used by all other projects
- **UILib** - a graphical user interface library
- **SoundLib** - a sound library
- **e42**(*) - the 3D engine
- **HTTPLib** - a very simple http library (rather a MFC-wrapper)
- **GameLib**(*) - a framework for Computer Games
- **3dview2** - the 3dview2 application

All projects marked with (*) were written in collaboration with my colleague Daniel Matzke - Thanks a lot, it's a pleasure to work with you!

These projects use some external open source libraries with have also been included in the CVS repository for your convenience:

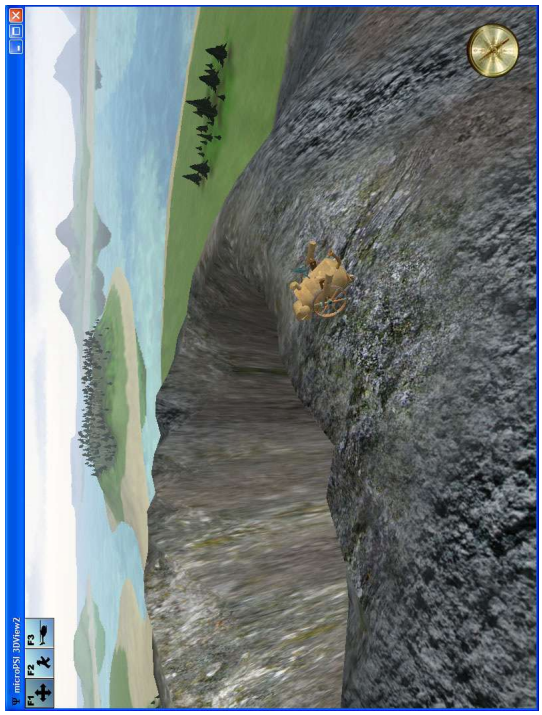
- **DevIL** - an open source image library. See http://openil.sourceforge.net/
- **TinyXML** - a small and easy-to-use XML parser. See http://www.grinninglizard.com/tinyxml/
- **Lib Vorbis** and **Lib Ogg**: an easy-to-use audio compressor, not unlike MP3. See http://www.vorbis.com/
- **Opcode**: a very fast mesh collision library. See http://www.codercorner.com/Opcode.htm

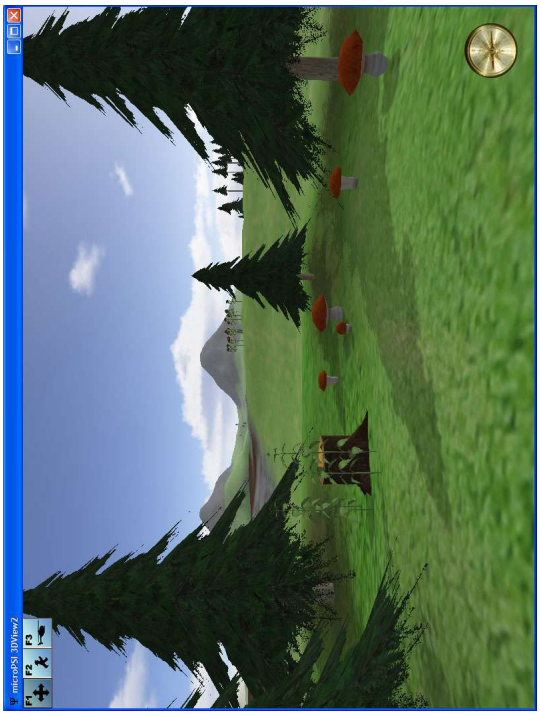
Anhang B – Screenshots

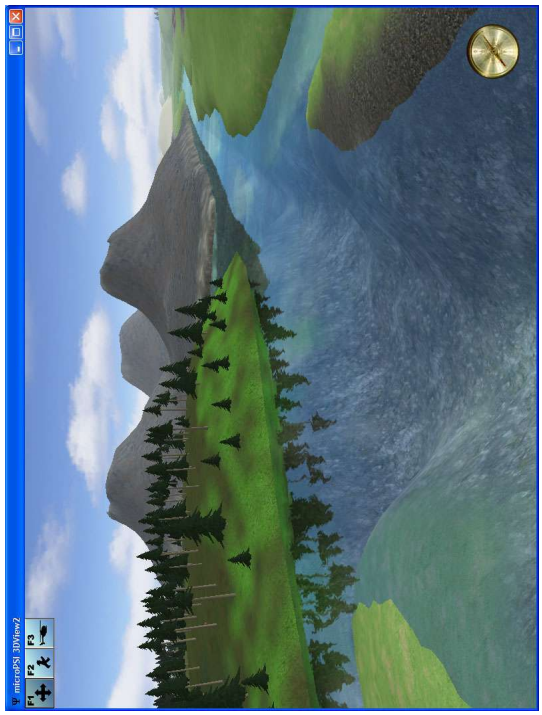

microPSI 3DView2
F1
F2
F3


microPSI 3DView2
F1
F2
F3


F1
F2
F3


F1
F2
F3


F1
F2
F3


F1
F2
F3


microDSI 3DView2 - 3DEmotion Viewer
microDSI 3DView2 - 3DEmotion Viewer


microDSI 3DView2 - 3DEmotion Viewer
microDSI 3DView2 - 3DEmotion Viewer

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich die vorstehende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die dafür verwendete Literatur habe ich vollständig verzeichnet und Textstellen, die wörtlich oder inhaltlich anderen Werken entnommen wurden, sind mit entsprechenden Quellenangaben gekennzeichnet.

Hohen Neuendorf, 02.07.2005 David Salz

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Hohen Neuendorf, 02.07.2005 David Salz