"Untersuchung von Techniken beim Level- und Characterdesign zur Laufzeitoptimierung von 3D-Echtzeitanwendungen, dargestellt am Beispiel eines Computerspiels"

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

an der
Fachhochschule für Technik und Wirtschaft Berlin
Fachbereich Wirtschaftswissenschaften II
Studiengang Angewandte Informatik

angefertigt in der Firma *ZeroScale*

1. Betreuer: *Prof. Dr. Thomas Jung*
2. Betreuer: *Christian Kinne*

eingereicht von: *Stefan Schubert*
email: *herr.schubert@web.de*

Berlin, 04. Mai 2002

Danksagung

Mein ganz besonderer Dank gilt meinen Eltern, die mir das Studium und damit diese Arbeit erst ermöglicht haben.

Ich danke aber auch dem Rest der Familie sowie meinem Freundeskreis für das Verständnis, das sie mir während dieser Zeit entgegengebracht haben.

Abschließend geht mein Dank an die Betreuer dieser Arbeit - Prof. Dr. Thomas Jung und Christian Kinne, die sich Zeit genommen haben und mit ihrer Sachkenntnis sowie ihren Rat zur Verbesserung der Arbeit beigetragen haben.

Inhaltsverzeichnis

I Einleitung

1 Vorwort

Vor gut 15 Jahren waren die technischen Möglichkeiten noch so stark begrenzt, dass man einen Punkt der zwischen zwei Strichen hin und her prallte als geniales Spiel bezeichnete und vor Begeisterung über dieses "Wunder der Technik" die Zeit davor vergaß. Viele Menschen haben damals ihren Computer genutzt, um sich an ersten Spielen zu versuchen.

Inzwischen hat sich durch den Fortschritt auf diesem Gebiet Unglaubliches getan und die Computerspielindustrie ist zu einem milliardenschweren Wirtschaftsfaktor geworden. "Nach einer Studie des Marktforschungsinstituts NPD Group überschritten die Verkaufserlöse von Computer- und Konsolenspielen im Jahr 1999 $8,8 Mrd. alleine in den USA (einschließlich Educationsoftware und Kinder-Lernspiele)“[1]. Neuerdings entstehen sogar Kinofilme, bei denen die Story der zugrundeliegenden Spiele verfilmt wird, wie zum Beispiel Tomb Raider oder Final Fantasy - allein die Produktionskosten beider Filme zusammen überstiegen, wenn auch nur knapp, die $200 Mio. Marke ($137 Mio. Final Fantasy / $85 Mio. Tomb Raider).

Wie bei allen Industriezweigen gibt es auch hier zwei Seiten: eine konsumierende und eine produzierende. Viele, die auf der verbrauchenden Seite stehen, hegen aufgrund ihrer Spielleidenschaft den Wunsch, einmal selbst ein Spiel zu entwickeln, dass dann genau ihren Vorstellungen entspricht. Doch nur wenigen gelingt es ihr Hobby zum Beruf zu machen, obwohl die Einsatzgebiete in dieser Branche sehr vielseitig sind, da für die Erstellung eines Computerspiels neben den Programmierern auch Musiker, Geräusch-Designer, Grafiker, 3D-Künstler und andere Mitarbeiter benötigt werden. Inzwischen ist es sogar möglich, Spieldesigner zu studieren[2].

Als sich mir während des Studiums die Möglichkeit bot, bei der Gestaltung eines Computerspiels maßgeblich mitzuwirken, erfüllte sich für mich ein seit Jahren gehegter Wunsch. Das Außergewöhnliche daran war jedoch, dass seinerzeit nur eine einzige Person – Programmierer, 2D-und 3D-Künstler in einem – daran entwickelte und dennoch beeindruckende Ergebnisse vorweisen konnte. Durch das sehr kleine Team schien sich für mich die Möglichkeit zu ergeben, in nahezu alle Bereiche der Spielentwicklung einen Einblick zu bekommen und dementsprechend Erfahrungen zu sammeln.

Mittlerweile ist aus dem ehemaligen „Ein-Mann-Projekt“ die Firma ZeroScale[3] hervorgegangen, zu der sich weitere Personen eingefunden haben, die bei der Realisierung dieses Spiels mithelfen.

Bei ZeroScales aktuellem und gleichzeitig ersten Computerspiel, handelt es sich um eine Skateboard-Simulation, die dem Spieler neben einer rasanten 3D-Grafik auch Action-Elemente bieten soll. Bis zur Fertigstellung ist es allerdings noch ein weiter Weg, da bis jetzt ein sogenannter Publisher fehlt, der Spiel-Entwicklungen vorfinanziert und anschließend die Vermarktung übernimmt. Um möglichst bald einen derartigen Sponsor zu finden, arbeitet das gesamte Team ehrgeizig an einer aussagekräftigen Demo-Version von „Skate Attack“, so der Arbeitstitel des Spiels.

[1] [Saltzmann 00, S.15f]
[2] http://www.gameprogrammer.com/links/schools.html
[3] http://www.zeroscale.com

Da bisher keine der beteiligten Personen an einer vergleichbaren Spiel-Entwicklung teilgenommen hat, stellen sich natürlich zahlreiche Fragen bezüglich der Herangehensweise und nach Möglichkeiten der Ergebnisoptimierung.

2 Ziel der Diplomarbeit

Inwieweit es im Bereich des Level- und Characterdesigns möglich ist, auf die Performanz des Spiels Einfluss zu nehmen, um sie daraufhin optimieren zu können, soll Gegenstand der Diplomarbeit sein.

In diesem Rahmen sollen allgemeine Methoden und Arbeitsweisen auf ihre Vor- und Nachteile bezüglich der Auswirkungen auf die Laufzeit eines Computerspiels untersucht werden. Ein wichtiger Aspekt ist dabei die Berücksichtigung der Qualitätskriterien eines Spiels. Neben den Anwendungsmöglichkeiten für „Skate Attack“ sind im Falle beeinträchtigender Ergebnisse auch Alternativen zu betrachten. Es bietet sich ebenfalls an, Techniken einzubeziehen, die bisher noch keine Berücksichtigung in der Spiel-Engine von „Skate Attack“ finden. Da diese vom Initiator des Spiels entwickelt wird, besteht nicht nur die Möglichkeit sich über nutzbringende Erkenntnisse verständigen zu können, sondern auch mit in die Spiel-Software einfließen zu lassen.

Ein weiterer wichtiger Punkt der Arbeit ist die Anwendung der auf diese Weise gewonnenen Ergebnisse und Prinzipien zur Verbesserung der Laufzeit beziehungsweise der Qualität des Spiels bei der Erstellung der Spielumgebung, sowohl im 3D- als auch im 2D-Bereich. Welche Schwierigkeiten und Vorteile sich dabei konkret für „Skate Attack“ ergeben, sollen Performanztests sowie Bilder der Anwendung zeigen. Für die Durchführung der Messungen ist ein Tool geplant, dass sowohl die Untersuchungen erleichtern als auch die Auswertung in einer geeigneten Form ermöglichen soll.

Letztendlich soll ein Prototyp – die Demo-Version des Spiels - entstehen.

II Grundlagen

1 3D-Computergrafik

"Computergrafik ist die Wissenschaft von der rechnergestützten Verarbeitung grafischer Daten." [Tönnies 94, S.11]

Die 3D-Computergrafik ist, neben zahllosen anderen, nur ein Teilgebiet dieser Wissenschaft. Sie hat sich in den letzten Jahren äußerst rasant entwickelt und ermöglicht inzwischen Dinge, die noch vor 10 Jahren undenkbar waren. Forciert durch das starke Interesse an 3D-Grafik, gibt es heute ein unglaublich großes Angebot an Soft- und Hardware, rund um dieses Thema.

Im täglichen Leben begegnet uns mehr 3D, als je zuvor – Fernsehen, Werbung, Kino oder Internet bedienen sich mit steigendem Interesse der „dritten Dimension".

Ein elementarer Begriff, der in diesem Zusammenhang immer wieder fällt und hin und wieder zu Verwirrungen führt, ist das sogenannte „Rendern". Sehr häufig wird vom Rendern gesprochen, wenn es um Bilder geht, bei denen Reflexionen und Spiegelungen in Glaskugeln, die auf einem Schachbrett liegen, zu sehen sind. Da es sich dabei aber um Ray-Tracing - ein spezielles Renderingverfahren - handelt, sei an dieser Stelle ausdrücklich daraufhingewiesen, dass sich hinter dem Wort „Rendern" (engl. to render - "wiedergeben") nichts anderes verbirgt, als das Erzeugen eines 2-dimensionalen Bildes, basierend auf Daten einer 3D-Szene [Watt 89, S.97].

1.1 3D-Engines

Ein Programm, das in der Lage ist, aus einer 3D-Szene ein zweidimensionales Bild zu generieren wird als 3D- oder Render-Engine bezeichnet.

Grundsätzlich werden zur Erzeugung von 3D-Grafiken zwei Arten von Programmen benötigt: Tools zum Erstellen von 3D-Szenen - die sogenannten Modeller und die oben erwähnten Renderer, die diese Informationen in Bilder umwandeln. Von beiden Anwendungstypen sind dutzende Vertreter der unterschiedlichsten Hersteller auf dem Markt zu finden. Das Angebot reicht dabei von Free- und Shareware bis hin zu Highend-Komplettpaketen, die für private Endverbraucher nahezu unbezahlbar werden. Die Preise richten sich dabei nach dem Umfang der angebotenen Möglichkeiten und deren Qualität, zum Beispiel der Geschwindigkeit und Bildqualität.

Durch das anhaltende starke Interesse an 3D-Grafik entstehen immer wieder neue und bessere Engines. Einen Überblick über das enorm große Angebot an 3D-Engines, bietet eine Auflistung im Internet[4], die zur Zeit mehr als 640 Einträge zählt.

Die Dauer eines Renderprozesses kann sehr stark variieren. Sie hängt nicht nur von der eingesetzten Hardware ab. Faktoren wie Szenenkomplexität, Anzahl der Lichtquellen, Rendermethode oder der Renderer selbst, beeinflussen die Berechnungszeit in einem so hohem Maße, dass eine nach oben offene Skala erreicht wird, beginnend mit einem Bruchteil einer Sekunde. Aufgrund dieser extrem großen Zeitunterschiede werden Renderer, wie auf der folgenden Seite näher erläutert, nach ihren Einsatzgebieten klassifiziert.

[4] http://cg.cs.tu-berlin.de/~ki/engines.html

1.1.1 Offline-Render

Heutige Renderer sind in der Lage extrem wirklichkeitsnahe Bilder zu generieren, was ihre steigende Verwendung in der Filmindustrie bestätigt. Effekte wie Schattenwurf, Lichtbrechungen an gekrümmten Oberflächen (Caustics) oder Tiefenschärfe (Depth of Field) dienen nur einem Ziel – die Wiedergabe der Wirklichkeit. Es wird daher auch von der Fotorealistischen Computergrafik gesprochen. Die Qualität dieser Bilder fordert jedoch ihren Preis in Form recht langer Renderzeiten, meist mehrere Sekunden oder Minuten für ein einzelnes Bild. Daher ist der Einsatz solcher Renderer in Echtzeitsystemen völlig unmöglich. Oft werden diese auch als Offline-Renderer bezeichnet.

1.1.2 Realtime-Render

Um Echtzeit-Rendering zu gewährleisten, muss die Render-Engine in der Lage sein, die Bilder in Sekundenbruchteilen zu generieren. Die Rate, mit der die Bilder (Frames) gerendert, beziehungsweise dargestellt werden, wird in Bildern pro Sekunde (Frames Per Second - FPS) angegeben. Erst wenn ein System in der Lage ist, 15 FPS zu erzeugen, lässt sich von Echtzeit sprechen [Möll 99, Kapitel 1].

Im Gegensatz zu den Offline-Renderern, bieten Echtzeitsysteme dem Benutzer die Möglichkeit zur Interaktion. Prinzipiell funktionieren solche Anwendungen so, dass ein Bild angezeigt, der Betrachter agieren oder reagieren kann und in Abhängigkeit dessen, das nächste Bild generiert wird [Möller 99, Kapitel 1]. Dabei ist es besonders wichtig, dass keine Verzögerung (Latenz) zwischen Aktion und Reaktion des Systems auftritt, ansonsten wird es dem Anwender kaum gelingen, auf das was er sieht, korrekt zu reagieren. Allerdings werden Echtzeitberechnungen nicht selten durch parallel arbeitende Prozesse realisiert. Dabei übernimmt beispielsweise ein Prozess die Generierung der Bilder um stets eine konstante und hohe Frame-Rate zu erzielen, während getrennt davon ein zweiter Prozess eine komplexe Kollision berechnet. Weitere Prozesse sorgen dann für eine Aktualisierung der Zustände beziehungsweise Verarbeiten die Eingabe des Benutzers. Demnach sind Frame-Rate und Verzögerung nicht unbedingt gekoppelt, dass heißt eine hohe Frame-Rate bedeutet nicht immer eine niedrige Latenz. Neben Simulationen beim Militär oder im Automobilsektor werden Echtzeitsysteme auch in der Medizin oder der Unterhaltungsindustrie (Computerspiele) eingesetzt.

Abb. II-1 Vergleich zwischen Realaufnahme (a) und 3D-Computergrafik (b) und (c)

Der Vergleich zwischen einem Foto[5] (a), einer Szene des Films „Final Fantasy“[6] (b) und einer Spielfigur aus „Dead Or Alive 3“[7] (c), macht die Leistungsfähigkeit moderner Computergrafik deutlich.

[5] Rhona Mitra - Fotomodell und erste menschliche Verkörperung der Lara Croft – Hauptfigur des Spiels Tomb Raider von Core Design
[6] ein komplett computergenerierter Spielfilm der auf der gleichnamigen Computerspiel-Serie „Final Fantasy“ von Square, Ltd. basiert
[7] Konsolenspiel aus dem Hause Tecmo, Ltd.

1.1.3 3D-Game-Engines

Hinter dem Begriff Game-Engine, oder auch Spiel-Engine, verbirgt sich das „Herz“ eines Computerspiels, dass sämtliche Daten verarbeitet die notwendig sind, um überhaupt spielen zu können. Sie lässt sich grob in weitere Engines unterteilen, die für spezifische Teilbereiche zuständig sind:

- Animation
- Grafik
- Sound
- Musik
- Eingabeverarbeitung (Tastatur, Joystick, Gamepad)
- Stats / Behavoir (Zustände / Verhalten)

Demnach ist die Grafik- oder auch Render-Engine nur eine Komponente, innerhalb eines Computerspiels. Mittlerweile verwendet nahezu jedes Spiel, das auf den Markt kommt, zur Darstellung dreidimensionaler Grafiken, so dass die Grafik-Engine fast immer eine 3D-Engine ist.

1.2 3D-Welten

Grundlage jeder Visualisierung bilden die Informationen, die das beschreiben, was dargestellt werden soll. Im Falle eines Computerspiels geht es um die Darstellung der sogenannten Spielwelt, mit der der Benutzer interagieren kann. Eine derartige Umgebung besteht im wesentlichen aus folgenden Elementen:

- Objekte
- Lichter
- Materialien
- Kameras

Sämtliche Informationen werden von der Render-Engine verarbeitet und in 2-dimensionale Bilder umgewandelt. Die dazu notwendigen Schritte, werden im Abschnitt 1.6 „3D-Grafik-Pipeline“ näher erläutert.

1.2.1 Objekte

Die Objektdaten werden als geometrische Formen im Speicher gehalten. Wobei es prinzipiell genügt deren Oberfläche zu beschreiben, da im Normalfall nur diese vom Anwender gesehen werden kann. Diese Art der Objektdarstellung wird als Oberflächenrepräsentation bezeichnet.

Da die Form eines Objektes beliebig sein kann, ist die Oberfläche selten durch eine Formel beschreibbar und wird daher aus einer Vielzahl einzelner Flächenelemente zusammengesetzt [Tönnies 94, S.38] Diese werden auch als Facetten oder Polygone bezeichnet. Wörtlich übersetzt, bedeutet Polygon nichts anderes als Vieleck. Genauer spezifiziert ist „ein reguläres, geschlossenes Polygon P eine zweidimensionale Struktur, die durch einen geschlossenen, sich nicht schneidenden Kantenzug mit den Kanten $K = \{\vec{k}_1, \vec{k}_2, ..., \vec{k}_n\}$ begrenzt wird (...). Eine Kante $\vec{k}_i$ besitzt zwei Eckpunkte (...).“ [Tönnies 94, Seite 38] Diese, sich im Raum befindenden Punkte, werden auch als Vertices (Einzahl - Vertex) bezeichnet.

Ein polygonales Objekt, setzt sich demzufolge aus 3 wesentlichen Komponenten zusammen:

- Punkte
- Kanten und
- Facetten, auch Faces genannt.

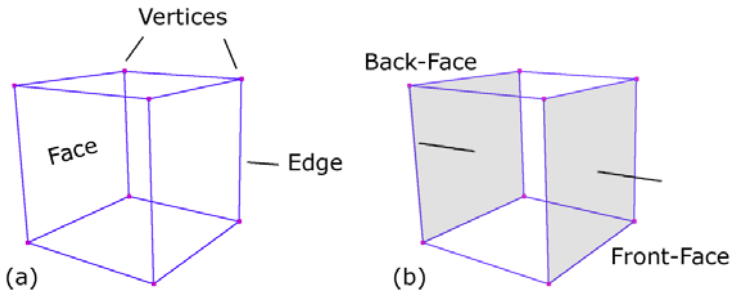

Abb. II-2 Die Komponenten eines polygonalen Würfels

Werden die benachbarten Vertices durch Linien miteinander verbunden (Veranschaulichung der Kanten), wie es die Abb. II-2 in (a) darstellt, ergibt sich ein sogenanntes Drahtgitter (Wire Frame). Es stammt aus dem CAD-Bereich und stellt eine primitive aber sehr schnelle Art der Objektrepräsentation dar. Drahtgitter-Modelle werden sehr häufig bei der Arbeit mit komplexen 3D-Szenen verwendet. Polygonale Objekte werden zudem auch als Polygongitter (Polygonmeshes) oder Mesh bezeichnet.

Die Facetten besitzen eine Orientierung, dass heißt ein Polygon kann dem Betrachter zu- oder abgewandt sein - es wird von Front- und Back-Faces gesprochen. In welche Richtung ein Polygon zeigt, hängt von dessen Normale ab – ein Vektor, der senkrecht auf der Fläche steht (Flächennormale) - Teil (b) der Abb. II-2, macht dies deutlich. Auch die Vertices haben aufgrund ihrer Kanten- und damit Flächen-Zugehörigkeit eine Normale (Punktnormale). Normalen sind für die Berechnung der Beleuchtung von größter Bedeutung.

Ein Punkt kann neben dem Wert der Normale und den Daten zur räumlichen Lage (Koordinaten x, y und z) weitere Informationen besitzen, zum Beispiel einen Farbwert oder UV-Koordinaten, deren Bedeutung unter 1.5 „Texturen“ dieses Kapitels beschrieben wird. Anzahl und Art der Attribute eines Punktes werden im sogenannten Vertex-Format beschrieben.

Bei einem polygonalen Objekt ist es möglich, dass sich benachbarte Facetten Punkte teilen (Shared Vertices). Die gemeinsamen Punkte speichern zwar für alle angrenzenden Faces dieselben Raumkoordinaten, bedeuten aber nicht, dass die übrigen Attribute der Punkte ebenfalls für alle Flächen gleich sein müssen. Intern werden sie als Punktgruppe verwaltet, deren Größe durch die Menge der angrenzenden Facetten bestimmt wird. Daher kann immer noch jeder einzelne, zu einer Fläche gehörende, Vertex unterschiedliche Eigenschaften besitzen. Auf diese Weise kann jedes Face seine eigene Farbe besitzen, trotz gemeinsamen Vertex. Sobald sich die Shared Vertices jedoch in einem Attribut unterscheiden, werden sie von der Engine zerlegt und als einzelne Punkte verarbeitet.

Wie bereits darstellt, sind Polygone nichts anderes als Vielecke. Beim kleinstmöglichen Vieleck - dem Dreieck (Triangle), handelt es sich um DAS Basisprimitiv der 3D-Welt. Die Gründe dafür sind in den Eigenschaften zu suchen. Ein Dreieck ist:

- einfach zu beschreiben
- stets (!) planar und konvex.

Außerdem lässt sich jedes Polygon durch Dreiecke beschreiben. Oft wird der Begriff Polygon verwendet, obwohl ein Dreieck gemeint ist. Da moderne Grafikkarten für deren Verarbeitung optimiert sind, werden die Objekte der 3D-Szene, zur einfacheren Manipulation in eine Vielzahl von Dreiecken zerlegt – dieser Vorgang wird als Tesselierung bezeichnet.

1.3 Beleuchtung

Ohne Licht, würde sich eine 3D-Szene für den Betrachter als schwarzes Bild darstellen. Wie Licht in eine Szene gelangt, soll im folgenden Abschnitt erklärt werden.

1.3.1 Lichtquellen

Im wesentlichen werden folgende Typen unterschieden:

- **Gerichtetes Licht (Directional Light)**
  hat neben Farb- und Intensitätswert eine feste Richtung. Die emittierten Lichtstrahlen sind unendlich und verlaufen parallel - ähnlich den Sonnenstrahlen, die auf die Erde gelangen.
- **Punkt- oder Omnidirektionales Licht (Point / Omnidirectional Light)**
  streut sein abgestrahltes Licht, wie eine Glühlampe, in sämtliche Richtungen. Es besitzt sowohl Farbe, Intensität als auch eine Position innerhalb der Szene. Ein Punktlicht kann, wie gerichtetes Licht unendlich weit strahlen, aber auch in seiner Reichweite begrenzt werden.
- **Lichtkegel (Spot Light)** verfügen über Farbe, Intensität, Position und Richtung. Die Wirkungsweise lässt sich mit einer Taschenlampe oder einem Scheinwerfer vergleichen, deren kegelförmig entsendetes Licht zum Rand hin schwächer wird.

Daneben gibt es noch das **Ambiente Licht (Ambient Light)**. Diese Form des Lichtes stellt eine Art Grundbeleuchtung (engl. Ambient – Umgebung) dar, die sich in der gesamten 3D-Szene befindet und durch seine Farbe und Intensität sämtliche Objektflächen in gleicher Weise beeinflusst, wodurch keine Differenzierung der Objektoberfläche erkennbar ist. [DirectX 00, „Light Objects“] zählt diese Lichtform aufgrund seiner Eigenschaften nicht zu den Lichtquellen.

Damit aus einem 3D-Objekt ein zweidimensionales Bild generiert werden kann, muss festgestellt werden, wie auf der Objektoberfläche auftreffendes Licht in Richtung Anwender reflektiert wird. Genau dieses Verhalten wird durch sogenannte Beleuchtungsmodelle beschrieben [Tönnies 94, S.101].

1.3.2 Beleuchtungsmodell

Was die Aufstellung eines solchen Modells so schwierig macht, sind die vielen ineinandergreifenden Bedingungen (Licht wird von verschiedensten Quellen emittiert, an Oberflächen unterschiedlichster Beschaffenheit gebrochen, reflektiert, zum Teil absorbiert, usw.). Daher stellen sämtliche Beleuchtungsmodelle, die in der Computergrafik verwendet werden, nur Annäherungen an die Wirklichkeit dar. Generell werden zwei Modell-Arten unterschieden:

- globale und
- lokale

"Durch ein lokales Beleuchtungsmodell wird nur die direkt von Oberflächen zum Betrachter reflektierte Strahlung berücksichtigt, während durch ein globales Modell auch die Reflexionen zwischen Oberflächen in die Beschreibung einfließen." [Tönnies 94, S.101]

Die Berücksichtigung von Mehrfachreflexionen bietet zwar realistische Ergebnisse, erfordert jedoch einen so hohen Rechenaufwand, dass sie in heutigen Echtzeitanwendungen nicht eingesetzt werden kann.

Das durch Hardwareunterstützung wesentlich schnellere lokale Beleuchtungsmodell ist dagegen für den Einsatz in Echtzeitsystemen geeignet. Aufgrund der Vernachlässigung der Reflexion zwischen Oberflächen sind keine Spiegelungen und Schlagschatten möglich. Diese Effekte müssen in Echtzeitanwendungen auf anderem Wege realisiert werden. Mögliche Techniken werden im folgenden Teil dieses Kapitels beschrieben.

Sind die Lichtquellen bekannt (Art, Position, etc.), kann zusammen mit den Eigenschaften der Objekte (Position und Ausrichtung der Oberflächenelemente) sowie der Betrachterposition das vom Objekt abgestrahlte Licht errechnet werden. Dafür stehen verschiedene sogenannte Reflexionsmodelle zur Verfügung:

- Ambiente Reflexion
- Diffuse Reflexion
- Spiegelnde Reflexion

Die **Ambiente Reflexion** ist das einfachste Modell, bei der „(...) eine richtungsunabhängige und gleichmäßige, indirekte Reflexion als alleiniger Einfluss angenommen (...)“ [Tönnies 94, S.114] wird. Bei der daraus resultierenden konstanten Beleuchtung handelt es sich um das zuvor unter „Lichtquellen“ beschriebene Ambiente Licht. Objekte, die auf diese Weise erhellt werden, erscheinen als einfarbige Silhouetten, die keine Intensitätsunterschiede oder Glanzlichter aufweisen.

Zur visuellen Verbesserung des Ergebnisses, kann die sogenannte **Diffuse Reflexion** hinzugenommen werden. Sie berücksichtigt das Licht, das in das Objekt eindringt und wieder in die Umgebung abgegeben wird. Die Helligkeit der Oberfläche hängt dabei sowohl von dessen Orientierung als auch vom Abstand zur Lichtquelle ab. Da das Licht in alle Richtungen gleichmäßig reflektiert wird, ist es vom Betrachter unabhängig. Als Ergebnis entsehen matte und unterschiedlich helle Oberflächen.

Glänzende Flächen lassen sich weder mit der Ambienten noch mit der Diffusen Reflexion veranschaulichen. Erst mit der **Spiegelnden Reflexion** wird auch das Licht berücksichtigt, dass an der Oberfläche gespiegelt (Einfallswinkel = Ausfallswinkel) wird. Bei dieser Reflexion ist die Position des Betrachters (Winkel zur Lichtquelle) entscheidend, ob auf der Objektoberfläche sogenannte Glanzlichter (Highlights) entstehen.

1.3.3 Schattierungsverfahren

Schattierungsverfahren (Shading-Verfahren) werden dazu benutzt, um die Helligkeitsverteilungen in den Polygonflächen zu bestimmen, die sich aufgrund des verwendeten Beleuchtungsmodells ergeben.

Beim sogenannten **Flat** oder **Constant Shading** findet die Beleuchtung anhand der Flächennormale statt. Dass heißt zusammen mit dem Lichtvektor und unter Berücksichtigung des Reflexionsmodells ergibt sich ein neuer Farbwert, der anschließend für die Schattierung des gesamten Polygons verwendet wird.

Der erforderliche Rechenaufwand ist der geringste von allen Schattierungsverfahren [DirectX, „Flat Shading“], das sichtbare Ergebnis aber entsprechend schlecht, da keine Schattierungen innerhalb der Polygone entstehen ist die facettenartige Struktur des Objektes deutlich zu erkennen. Durch das in der Literatur als „Mach-Band-Effekt“ beschriebene Phänomen werden die gemeinsamen Kanten, an denen helle auf dunkle Flächen treffen vom menschlichen Auge zusätzlich verstärkt. Dabei treten auf der helleren Seite der Kante ein schmales Band auf, dass noch heller erscheint und auf der dunklen Seite ein Band das noch dunkler erscheint. Eine Verminderung dieses Effekts kann durch eine Erhöhung der Polygonzahl erreicht werden, bedeutet aber auch mehr einen höheren Daten- und Rechenaufwand.

Für bessere Ergebnisse, ohne mehr Polygone zu verwenden, kommen Schattierungsverfahren mit Interpolation zum Einsatz, die Informationen angrenzender Polygone mitberücksichtigen.

Ein nach Henri Gouraud benannter Algorithmus (**Gouraud Shading**), benutzt bei der Beleuchtungsberechnung die Vertex-Normalen. Dass heißt, die Beleuchtung wird für die einzelnen Punkte des Objektes berechnet. Anschließend wird durch Interpolation der Farbwerte das Polygon gefüllt, wodurch Helligkeitsverläufe möglich sind, die gekrümmte Flächen „weich“ erscheinen lassen. Selbst Glanzlichter können auf diese Art berechnet werden, hängen aber stark von der Lage der verwendeten Polygone ab [Watt 00, Seite 183]. So würde ein Licht, dass nur auf ein Vertex der Fläche fällt, in die Farbberechung des gesamten Polygons einfließen, wodurch zum Beispiel feine Glanzlichter nur möglich sind, wenn das Objekt aus sehr vielen Einzelflächen besteht. Gouraud Shading kann aber auch bestimmte Details gar nicht darstellen – so würde beispielsweise ein Spotlicht, das auf ein Polygon fällt ohne dessen Vertices zu „berühren“, erst gar nicht in Erscheinung treten.

Durch die wesentlich besseren Bildergebnisse und immer noch relativ einfache Beleuchtungsberechung wird dieses Verfahren in nahezu allen Echtzeitanwendungen eingesetzt.

Ein weiteres Schattierungsverfahren ist das **Phong Shading** – benannt nach Phong Bui-Tuong, der 1975 Gouraud‘s Verfahren aufgriff und dahingehend verbesserte, dass Specular Highlights ermöglicht werden, die nicht von den verwendeten Polygonen abhängen. Für jedes Pixel der Polygonfläche, die schattiert werden soll, wird eine Normale interpoliert und mit deren Hilfe die Beleuchtung durchgeführt.

Aufgrund des enormen Rechenaufwandes und einer fehlenden Hardwareunterstützung findet das Phong Shading keine Verwendung in Echtzeitanwendungen.

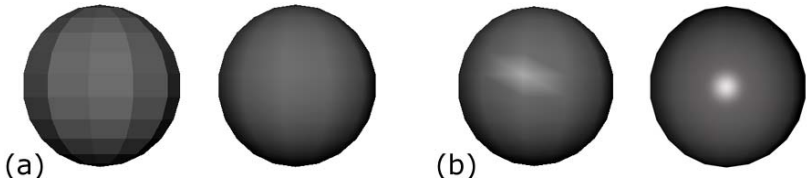

Abb. II-3 Eine Kugel (220 Dreiecke) mit verschiedenen Shading-Verfahren gerendert

In (a) der Abb. II-3 stehen Flat (links) und Gouraud (rechts) Shading gegenüber – die Überlegenheit von Gouraud ist deutlich zu erkennen. Teil (b) der Abbildung zeigt den Vorteil von Phong Shading (rechts) bei Glanzlichtern im Vergleich zum Gouraud Shading (links).

1.4 Materialien

Laut [DirectX 00, „Material Properties“] beschreiben Materialien, wie Polygone Licht reflektieren oder selbst in die Szene abgeben. Dass heißt, von ihnen hängt es ab, ob und inwiefern die zuvor beschriebenen Reflexionstypen eine Auswirkung haben. Dazu können Farbwerte sowohl für die Ambiente, Diffuse als auch die Spiegelnde Reflexion festgelegt werden. Die Stärke, des zur spiegelnden Reflexion gehörenden Glanzlichtes, wird über einen extra Wert festgelegt. Ein Selbstleuchten wird über das Emission-Attribut des Materials gesteuert.

1.5 Texturen

Die meisten Texturen sind 2-dimensionale Felder aus Farbwerten – also Bilder. Sie werden in 3-dimensionalen Welten dazu verwendet, Objekte mit Strukturen zu versehen (engl. Texture – Struktur). Ein einzelner Farbwert, das kleinste Textur-Element, wird Texel genannt und hat eine feste Position, die sich, ähnlich wie in einer Tabelle, aus Spalten und Reihen ermitteln lässt – sie werden als U und V bezeichnet. Diese UV-Texturkoordinaten befinden sich im sogenannten Texturraum, der im Normalfall von 0 bis 1 reicht.. Über die UV-Werte, die in den Vertices gespeichert sind, kann in der Render-Phase zugeordnet werden wie die Textur über die Geometrie gelegt werden soll (Texture Mapping).

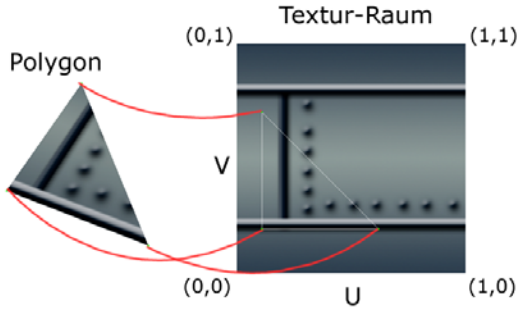

Abb. II-4 Zusammenhang zwischen Texturraum und UV-Koordinaten der Vertices

Die Farbe, die ein Texel repräsentiert, wird durch eine Bit-Kombination bestimmt, wobei ein Bit die Werte 0 und 1 annehmen kann. Die Bit-Zahl (Farbtiefe) eines Bildes gibt die maximale Zahl der darstellbaren Farben an. Bei einer 1 Bit-Grafik (0=schwarz / 1=weiß) handelt es sich um reines Schwarz-Weiß-Bild. Ein Farbbild mit 24 Bit dagegen, stellt für die Grundfarben Rot, Grün und Blau, die sich auf getrennten Kanälen befinden, jeweils 8 Bit zur Verfügung, woraus sich circa 16,7 Mio. verschiedene Farbwerte ergeben.

1.5.1 Alpha-Kanal

Neben den Farbkanälen kann eine Textur auch einen sogenannten Alpha-Kanal beinhalten, mit der es möglich ist Transparenzen oder Maskierungen zu erstellen. Die Informationen dieses Kanals können als Schwarz-Weiß-Bild verstanden werden, deren weiße Stellen dafür sorgen, dass die Texturfarben zu sehen sind, schwarze hingegen verbergen die Textur. Auch hier ermöglich eine höhere Bit-Zahl mehr Abstufungen, so dass auch zum Teil transparente Stellen möglich sind – üblich sind 1 und 8 Bit Alpha-Kanäle.

1.5.2 Texturgröße

Der allgemein oft verwendete Begriff Texturgröße ist für spätere Betrachtungen zu ungenau und wird daher, wie folgt, differenziert:

- die Auflösung - bedeutet in diesem Fall aus wie vielen Zeilen sich die Bitmap zusammensetzt (Höhe) beziehungsweise aus wie vielen Bildpunkten eine einzelne Zeile besteht (Breite)
- der benötigte Speicherplatz - ergibt sich aus der Auflösung und der Farbtiefe des Bildes

Beispiel:
Eine Textur der Auflösung 640 x 480 Pixel und einer Farbtiefe von 16,7 Mio. Farben benötigt 640x480 x24 = 7.372.800 Bit, das entspricht einem Speicherbedarf von 900 KByte.

1.6 3D-Grafik-Pipeline

Wenn eine 3D-Szene auf einen Monitor oder Fernseher ausgegeben werden soll, müssen dafür ganz bestimmte Berechnungen stattfinden, die immer nach einem bestimmten Schema ablaufen. Diese Abfolge wird in der Literatur auch als Grafik-Render-Pipeline oder 3D-Pipeline bezeichnet.

Wesentliche Funktion dieser Pipeline ist es, aus sämtlichen Ausgangsdaten, wie den darzustellenden 3D-Objekten, deren Materialien, dem zugrundeliegenden Beleuchtungsmodell, dem Standort des Betrachters und so weiter ein zweidimensionales Bild zu generieren (rendern).

Neben der eigentlichen Umwandlung und Ausgabe der Informationen, beinhaltet die Pipeline viele Techniken zur Effizienzsteigerung. Die wahrscheinlich größte Bedeutung wird der Sichtbarkeitsbestimmung beigemessen. Diese Verfahren bestimmen, welche Objekte oder Teile von Objekten vom Betrachter wahrgenommen werden können beziehungsweise welche Objektflächen durch Flächen anderer Objekte oder des eigenen Objekts verdeckt werden.

Wie die Abfolge dieser Funktionskette im Falle eines lokalen Beleuchtungsmodells aussieht, soll im Folgenden gezeigt werden. Da das Rendern jedoch von der verwendeten Hardware und der Render-Engine abhängt und sich diese in ihrer Funktionsweise und Vielfalt implementierter Möglichkeiten unterscheiden, legt die Darstellung keinen Wert auf Vollständigkeit, sondern soll einen Überblick über die Problematik gewähren und sowohl Grundlagen als auch Zusammenhänge verdeutlichen, die für spätere Erklärungen wichtig sind.

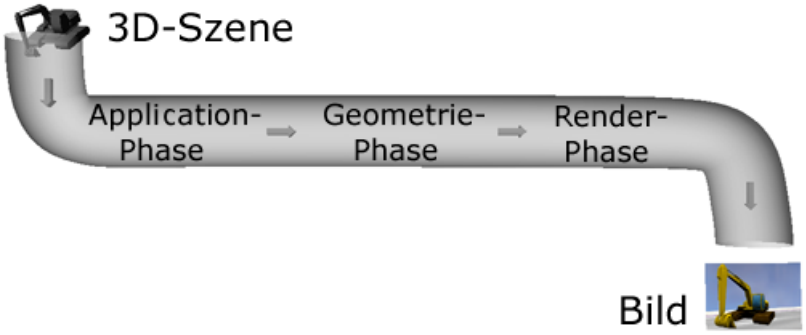

Abb. II-5 Schema der 3D-Pipeline

Das Schema der Abb. II-5 verdeutlicht, dass sich die Pipeline in 3 wesentlichen Abschnitte unterteilt:

- Application – Phase
- Geometrie – Phase
- Render – Phase, auch Rasterisierungs – Phase genannt.

Den Ausgangspunkt bildet die 3D-Szene mit den dazugehörigen Informationen. Eine Besonderheit der 3D-Pipeline ist, dass sie eine Form der Parallelverarbeitung darstellt, da sich Berechnungen auf die CPU und den Grafikprozessor der Plattform verteilen. Noch vor 6 Jahren war diese Verteilung sehr ungleich – fast alle Berechnungen fanden auf der CPU statt, der Grafik-Prozessor war nur für das Rendern zuständig. Seit 1999 ist das anders - die Firma nvidia[8] entwickelte einen Grafik-Prozessor mit integrierter Transform&Lighting Engine – die sogenannte GPU (Graphics Processing Unit). Diese GPU ist in der Lage sämtliche Rechenoperationen der Geometrie- und Render-Phase zu übernehmen. Dadurch wird wertvolle Rechenzeit auf der CPU frei, die für die zentralen Aufgaben der Application-Phase genutzt werden kann.

1.6.1 Application - Phase

Die erste Stufe verdient ihren Namen angesichts der Tatsache, dass der Softwareentwickler die volle Kontrolle darüber hat, was in ihr passiert. Das bedeutet, er hat die Möglichkeit Implementierungen zu verändern und nimmt damit direkten Einfluss auf die Performance. Veränderungen an der Implementierung der zweiten oder dritten Phase gestalten sich dagegen erheblich schwieriger, da heutzutage der Großteil in die Grafik-Hardware integriert ist [Möller 99, Kapitel 2]. Zu den zentralen Aufgaben gehören:

- Signalverarbeitung der Eingabegeräte (wie Tastatur oder Joystick)
- Bewegung der Kamera
- Künstliche Intelligenz (KI) von Gegnern etc.
- Kollisionserkennung (Collision Detection)
- Physik (Physics)

Eine wesentliche Aufgabe der Application-Phase ist es, jene Objekte herauszufiltern, die vom Betrachter zum gegenwärtigen Zeitpunkt potentiell sichtbar sind. Das heißt, aufgrund der aktuellen Position und Blickrichtung des Anwenders, ergibt sich ein möglicher Sichtbereich. Dieser Bereich wird in

[8] http://www.nvidia.com/

der Tiefe durch zwei Flächen (Front- und Back-Plane) eingegrenzt. Das auf diese Weise entstehende Sichtvolumen (Frustum), entspricht bei der perspektivischen Sicht einem Pyramidenstumpf, siehe Abb. II-6.

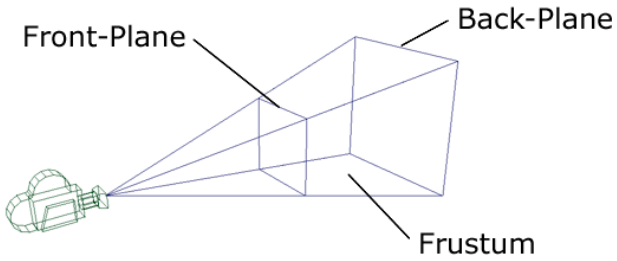

Abb. II-6 Das perspektivische Sichtvolumen

Sämtliche Szenen-Objekte die sich außerhalb dieses Frustums befinden, können problemlos entfernt werden (View Frustum Culling). Aufgrund der Tatsache, dass diese Objekte einen komplizierten Aufbau haben könnten, wäre die für diesen Test benötigte Rechenzeit immens. Geprüft werden daher nur einfache Begrenzungsgeometrien, die das gesamte Objekt einschließen (Bounding Box). Nur wenn sich die Bounding Box komplett außerhalb des Frustums befindet, wird das Objekt definitiv nicht "gesehen" und kann demzufolge entfernt werden. Bounding Boxen können wiederum in einer hierarchischen Struktur geordnet sein, so dass beim Eliminieren einer übergeordneten Box, sämtliche untergeordneten Objekte wegfallen (Hierarchical View Culling), wodurch Rechenzeit eingespart werden kann. Die darzustellenden Objekte werden dagegen in eine Liste (Display List) aufgenommen und weitergeleitet. Weitere Verfahren zur Sichtbarkeitsbestimmung werden unter „Culling-Techniken“ der allgemeinen Optimierungstechniken im Kapitel „Analyse“ beschrieben. In der Literatur wird diese Problematik als Hidden-Surface-Removal (HSR) bezeichnet.

Ferner wird in Abhängigkeit der Entfernung des Betrachters zu den Objekten, deren Detaillierungsgrad bestimmt (Level of Detail - LOD). Eine detailliertere Beschreibung ist im gleichnamigen Teil des zuvor erwähnten Kapitels zu finden.

1.6.2 Geometrie - Phase

In der Geometrie-Phase durchlaufen die Objekte der Display List, vielmehr deren Vertices, zahlreiche mathematische Operationen, die wie bereits erläutert, heutzutage fast ausschließlich von der Grafik-Hardware übernommen werden können. Zur schnelleren Verarbeitung, werden die Daten in spezielle Koordinatensysteme umgewandelt (transformiert). Dadurch ergeben sich verschiedene Koordinatensysteme, auch Spaces genannt, wie im folgenden Schema dargestellt:

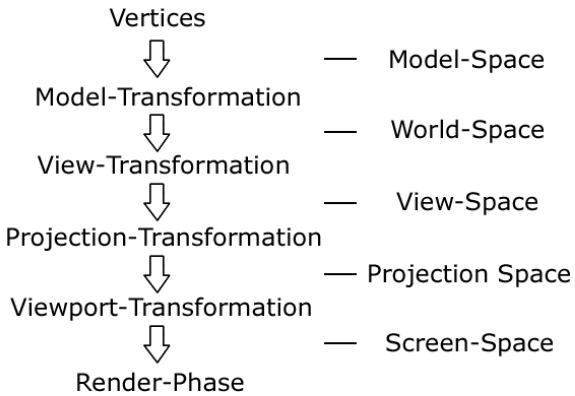

Abb. II-7 Schema der Geometrie-Phase

Die so errechneten Koordinaten werden für jeweils unterschiedliche Rechenoperationen benötigt. Zur näheren Erläuterung:

Bei der Erstellung der Objekte befindet sich jedes in seinem eigenen Model Space, auch Local Space genannt, was bedeutet, dass sie noch nicht transformiert wurden.

Zur Positionierung und Ausrichtung innerhalb der 3D-Szene wird auf das Objekt eine Modell-Transformation angewendet. Es ist möglich, einem Modell mehrere Modell-Transformationen zuzuweisen. Dies ermöglicht verschiedene Kopien desselben Objekts, die sich in Position, Richtung und Größe unterscheiden, ohne die Objektgeometrie duplizieren zu müssen, sogenannte Instanzen [Möller 99, Kapitel 2].

Ist die Modell-Transformation angewandt - das Objekt transformiert - befindet es sich im Welt-Koordinatensystem (World Space). Falls ein Objekt bewegt wird, muss das Bewegungssystem das Objekt Frame für Frame ins Weltkoordinatensystem transformieren [Watt 00, S. 143]. Letztlich müssen alle Objekte, die gerendert werden sollen, in dieses globale Koordinatensystem übertragen werden. Dazu gehören neben den Objekten, auch Materialien, Kameras und Lichtquellen, die zur Berechnung des Bildes benötigt werden.

Wie die Objekte der Szene, hat auch der Betrachter eine bestimmte Position und Ausrichtung. Mit Hilfe dieser Informationen werden als Nächstes die Objekte und Flächen bestimmt, die auch wirklich vom Benutzer gesehen werden können und somit gerendert werden müssen. Zur Vereinfachung dieser Berechnungen wird auf den Betrachter und die Objekte der Display List eine View-Transformation angewendet. Ziel ist ein Koordinatensystem, bei dem sich der Betrachter im Ursprung befindet. Das so entstandene System wird häufig als Camera-, Eye- oder View-Space bezeichnet.

Die Eye Space Koordinaten werden sowohl zur Beleuchtung als auch für ein weiteres HSR-Verfahren – dem Back-Face Culling benötig. Eine Erklärung kann im Kapitel III „Analyse“ unter 2.1.4 „Culling-Techniken“ gefunden werden.

Ein weiterer wichtiger Schritt ist die sogenannte Projektion. Erst sie ermöglicht das Abbilden einer dreidimensionalen Welt auf eine zweidimensionale Ebene. Wie die Objekte später dargestellt werden hängt von der sogenannten Projektions-Transformationen ab. Generell werden 2 Projektionsarten unterschieden:

- orthogonale (oder auch parallele) Projektion und
- perspektivische Projektion

Bei der Parallelprojektion wird davon ausgegangen, dass der Betrachterstandpunkt im Unendlichen liegt wodurch die Projektionsstrahlen parallel zueinander verlaufen. Die daraus resultierenden Abbildungen werden zum Beispiel in CAD- oder anderen Modellierungstools bei den orthogonalen Sichten (Top, Side, Front) verwendet.

Die perspektivische Projektion berücksichtig dagegen den Abstand zwischen Objekt und Betrachter, indem entfernte Objekte kleiner abgebildet werden als nähere derselben Größe. Dies dient dazu, dem Anwender eine glaubhafte Tiefe der Szene zu vermitteln, wie sie im täglichen Leben vorkommt.

Das bei der Projektion zugrundeliegende Sichtvolumen wird für ein weiteres Verfahren angewendet, das nicht sichtbare Flächen beseitigt - es wird als Kappen (Clipping) bezeichnet. Beim Clipping wird die Lage der Polygone zum Frustum geprüft. Flächen, die sich außerhalb des Volumens befinden, können problemlos entfernt werden. Befinden sie sich dagegen vollständig im Inneren, werden sie beibehalten. Tatsächlich gekappt werden nur die Flächen, die sich teilweise innerhalb befinden. [Möller 99, S. 16] Für die entsprechenden Schnittstellen müssen neue Vertices, einschließlich deren Eigenschaften, berechnet und die alten, außen liegenden, gelöscht werden.

Die verbleibenden Vertices werden im letzten Schritt der Geometrie-Phase, entsprechend den Werten des Anzeigefensters (Viewport) auf dem Ausgabegeräte, transformiert und damit in den sogenannten Screen Space umgewandelt [DirectX 00, „Fixed Function Vertex and Pixel Processing“].

1.6.3 Render - Phase

Aufgabe dieser Phase ist es, aus den verbliebenen Daten, den Vertices, ein Bild im Anzeigefenster des Ausgabegerätes zu generieren. Für die Darstellung steht allerdings nur eine begrenzte Zahl an Pixel zur Verfügung, so dass die Formen, die die Objektgeometrien beschreiben, mit dieser zur Verfügung stehenden Anzahl an sogenannten Rasterpunkten angenähert werden müssen. Je mehr Pixel zur Verfügung stehen (höhere Auflösung), desto genauer wird dieses Annäherung und verbessert das Ergebnis.

Neben der Rasterung, gehört das Füllen der Flächen mit Farbe zu den wesentlichen Aufgaben dieser Phase. Die Farbwerte der Polygonflächen beziehungsweise der resultierenden Bildpunkte werden durch die Beleuchtung, den Materialien, sowie Textur- und Alpha-Werte bestimmt.

Da polygonweise gerendert wird, sich in der 3D-Szene jedoch Flächenelemente gegenseitig verdecken können, kommt ein sogenannter Tiefenspeicher zum Einsatz. Mit dessen Hilfe ist es möglich, nur die Punkte des finalen Bildes darzustellen, die nicht von anderen Flächen verdeckt werden. Dieser z-Buffer, speichert die Tiefeninformation (z-Werte) der bereits gerenderten Pixel. Beim zeichnen eines weiteren Polygons, wird der z-Wert jedes einzelnen Pixels mit dem Wert der entsprechenden Stelle im z-Buffer verglichen. Ist der Wert des aktuellen Pixels kleiner – liegt das zuvor gerenderte Pixel räumlich tiefer – wird dessen Wert im z-Buffer übernommen und gleichzeitig der Farbwert des aktuellen Pixels in den Farbspeicher gespeichert. Damit wird gewährleistet dass zum Schluss nur die Pixel gezeigt werden, die nicht verdeckt sind.

Erst wenn sämtliche Polygone auf diese Weise verarbeitet wurden, ist ein Frame fertig und kann dargestellt werden. Damit jedoch der Bildaufbau für den Anwender verborgen bleibt, erfolgt dieser zunächst im sogenannten Frame-Buffer, der sich in weitere Speicherbereiche unterteilt. Während der Inhalt eines Speicherbereichs angezeigt wird, kann im Zwischenspeicher das nächste Bild zusammengesetzt werden.

Ein weiterer, sehr wichtiger Buffer, der sogenannte Stencil-Buffer sei an dieser Stelle ebenfalls kurz erklärt. Der Stencil-Buffer arbeitet nach [Microsoft 01], wie der z-Buffer, auf Pixel-Basis und teilt sich mit dem z-Buffer vorhandenen Speicher, so dass ein gebräuchliches z-/Stencil-Buffer-Format: 15/1 oder 24/8 ist. Der Stencil ermöglicht Maskierungen des Bildes, die zum Beispiel verhindern, dass in einen bestimmten Bereich des Ausgabefenster Pixel gerendert werden. [DirectX, „What is a Stencil Buffer“].

1.7 Zusätzliche Effekte

Wie aus der zuvor erläuterten Beleuchtung hervorgeht, entstehen aufgrund des lokalen Beleuchtungsmodells in Echtzeitanwendungen weder durch Lichtquellen noch durch Shading-Verfahren Schlagschatten. Auch Spiegelungen müssen auf gesondertem Wege in Echtzeitanwendungen eingebracht werden. Nachfolgend werden die wichtigsten Methodiken beider Bereiche erörtert.

1.7.1 Schlagschatten

Bei einer sehr einfachen und schnellen Methode werden Geometrien projiziert (**Projizierter Schatten**). Was das bedeutet, soll mit Teil (a) der Abb. II-8 und wie folgt erklärt werden: Ausgehend von der Lichtquelle kann die mit Schatten bedeckte Region der Projektionsfläche, durch eine Projektion der Objektgeometrie, berechnet werden. Das Objekt wird dann zweimal gerendert, einmal mit der normalen View/Model-Transformation sowie den Farbinformationen und ein weiteres mal mit der Projektions-Transformation und dem Farbwert des Schattens [Heidrich 99].

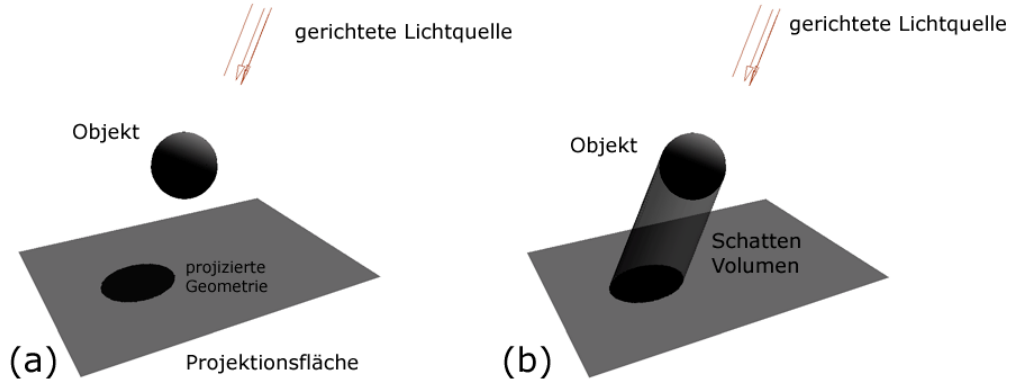

Abb. II-8 Möglichkeiten der Schlagschattenberechnung in Echtzeitanwendungen

Eine weitere Methode stellen volumetrische Schatten dar (**Schatten Volumen** oder **Volume Shadows**), siehe (b) der Abb. II-8, die folgendermaßen berechnet werden:

Aus Sicht der Lichtquelle ergibt sich für das schattenwerfende Objekt eine Silhouette, die vom Licht weg, in eine bestimmte Richtung und Tiefe des Raumes „gestreckt“ einen sogenannten Extrusionskörper ergibt, der durch ein polygonales Objekt beschrieben wird. Eine 3D-Szene mit Schatten Volumen wird nach [Microsoft 01, Heidrich 99] in drei Stufen gerendert: Als erstes normal über den z-Buffer, allerdings ohne Schatten. Mit Hilfe der Informationen des z-Buffer werden dann die Schatten im Stencil-Buffer maskiert. In einem abschließenden Render-Durchlauf kann die Maske des Stencils dazu genutzt werden, die Schattenstellen abzudunkeln.

Eine weitere Möglichkeit zur Echtzeitberechnung von Schlagschatten stellen die sogenannten **Shadow Maps** dar, wobei es verschiedene Implementierungen gibt. Prinzipiell wird jedoch zuerst das schattenwerfende Objekt aus Sicht der Lichtquelle gerendert - das so entstandene Bild wird als Shadow Map bezeichnet. Anschließend werden im Bereich des Schattens befindliche Geometrien mit der Shadow Map gerendert.

2 Entwicklung von Computerspielen

2.1 PC kontra Konsole

Ein nicht zu verachtender Unterschied bei der Spiel-Entwicklung ist, ob für den PC- oder Konsolen-Markt produziert wird:

Es existiert eine enorme Zahl an PC-Plattformen unterschiedlichster Hardware-Ausstattungen (Prozessoren, Grafik-, Soundkarten, ...) und -Konfigurationen, die es zu berücksichtigen gilt, damit das Spiel auf dem neuesten PC genauso spielbar ist, wie auf einem älteren. Da dies nicht ohne weiteres möglich ist, nutzen die meisten PC-Spiele in den seltensten Fällen die allerneusten Grafikfeatures und gehen somit auch nicht an die absoluten Leistungsgrenzen der PCs. Eine denkbare Lösung sind sogenannte skalierbare Engines, die erkennen, wie leistungsstark das System ist auf dem sie laufen und entsprechend reagieren. Dass heißt automatisch die Frame-Rate konstant halten, indem sie zum Beispiel die Szenen-Objekte mit einer geringeren Anzahl an Polygonen zeigen oder beim Einsatz von Partikelsystemen die Anzahl an Teilchen reduzieren. Der Programmieraufwand wäre allerdings enorm, so dass es die gebräuchlichere und auch vernünftigere Alternative ist, dem Benutzer die Möglichkeit zu geben, in bestimmten Bereichen wie Grafik oder Ton, zwischen verschiedenen Qualitätsstufen auszuwählen. Die Berücksichtigung vieler Systeme kostet aber auch wesentlich mehr Zeit und Geld - ein aus heutiger Sicht nicht unerheblicher Aspekt bei der Entwicklung neuer Spiele.

Standardisierte 3D-Schnittstellen (auch Application Programming Interfaces – kurz APIs genannt) tragen zwar zur Erleichterung der Entwicklung im PC-Bereich bei, indem auf den API-Befehlsvorrat zurückgriffen werden kann ohne die Grafik-Hardware direkt ansprechen zu müssen, eine sich in Bezug auf Hardware-Ausstattung und -Konfiguration nicht ändernde Plattform, lässt Entwicklern jedoch die größeren Möglichkeiten ihr Produkt optimal daran anzupassen. Dies ist einer der Hauptgründe, weshalb Spiele, die speziell für eine Konsole entwickelt wurden, oft brillant umgesetzt werden können, obwohl die zur Verfügung stehende Hardware nicht mehr den aktuellen Stand der Technik repräsentiert. Aber auch die Entwicklung für den Konsolen-Bereich bringt nicht nur Vorteile mit sich.

Die sogenannten Entwickler-Kits für Konsolen, sind nicht nur teuer sonder auch sehr schwer zu bekommen. Erst mit deren Hilfe kann jedoch getestet werden, wie das Spiel in der Konsolen-Umgebung funktioniert.

Darüber hinaus bietet der überwiegende Teil der Videokonsolen nur Anschlussmöglichkeiten für Fernsehgeräte. Aufgrund der Bauweise bieten diese, verglichen mit PC-Monitoren, eine wesentlich schlechtere Bildqualität. Hat sich im PC-Bereich eine Bildschirmauflösung von 1280x1024 Pixel schon fast als Standard etabliert, müssen sich Fernsehzuschauer mit weit weniger zufrieden geben. Hierbei treten allerdings regional bedingte Unterschiede auf, die bei der Spielentwicklung für ein Konsolensystem beachtet werden müssen:

- in Europa (PAL) - Bildschirmauflösung: 720x576 Bildpunkte mit 50 Hz
- in USA (NTSC) - Bildschirmauflösung: 720x480 Bildpunkte mit 60 Hz

Sowohl bei den PCs als auch den Konsolen haben sich als Speichermedien für die Spiele CDs oder DVDs durchgesetzt, da sie viel Platz für die anfallenden Daten (Grafik, Video, Sprache, ...) bieten. Damit das Spiel jedoch ausgeführt werden kann, müssen die Daten in den Hauptspeicher beziehungsweise Grafik-Speicher geladen werden. Die zuvor erwähnten Medien bieten allerdings nur relativ hohe Zugriffszeiten, daher werden im PC-Bereich Spiele normalerweise vorher auf die wesentlich schnelleren Festplatten übertragen. Dennoch bleibt das Problem, der geringen Größe von Haupt- und Grafik-Speicher, verglichen mit den Speichermedien.

Die neueste Konsole, Microsofts Xbox, überträgt den Vorteil des PCs - eine Festplatte, erstmals in den Bereich der Spielkonsolen. Eine Möglichkeit Daten für schnellere Zugriffe abzulegen oder Veränderungen im Spiel beziehungsweise Spielstände permanent zu speichern.

Pro	PC	Contra
- schneller Zugriff auf Festplatten		- Unmengen an unterschiedlicher Hardware-Ausstattungen und –Konfigurationen möglich
	Konsole	
- ein Konsolen-Typ hat eine feste Hardware-Ausstattung - da Ausgabe auf Fernsehgeräte erfolgt, sind geringere Textur-Auflösungen und weniger leistungsstarke Game-Engines notwendig		- teure Entwickler-Kits und Werkzeuge

Tab. II-1 Ein Vergleich der Spiel-Entwicklung für die Zielplattform PC bzw. Konsole

Auf dem heißumkämpften Konsolen-Markt existieren derzeit nur noch 3 große Hersteller. Zwar haben die seit längerem im Geschäft verweilenden Firmen Nintendo und Sony erst jüngst einen ihren Konkurrenten - Sega verloren, jedoch gibt es bereits einen neuen Hersteller - Microsoft. Dessen Gründer, Bill Gates, hat sich persönlich der Herausforderung angenommen eine Spiel-Konsole, die Xbox, zu entwickeln und zu etablieren. Da diese Konsole eher einem PC gleicht und durch die 3D-API DirectX ähnlich „leicht“ zu programmieren sein wird, sehen viele Spiel-Hersteller die Xbox schon als Referenz-Plattform, für die in Zukunft vorzugsweise Spiele produziert werden, gefolgt von den anderen Konsolen und dem PC.

Hardware-Eckdaten der Xbox laut [Gieselmann 02]	
Prozessor	733 MHz CPU von Intel
Grafikkarte	Spezieller Gforce 3-Chip der Firma Nvidia (XGPU mit 233 Mhz)
Hauptspeicher	2x32 MB DDR-SDRAM (200 MHz)
Festplatte	8 GB
Soundkarte	Dolby Digital 5.1 fähig, bis zu 64 Stimmen
Ethernet-Karte	100 Mbit

Tab. II-2 Die Hardware der Xbox zeigt deutlich die Nähe zum PC

2.2 Level- & Characterdesign

Der Begriff **Level** stammt aus dem Bereich der Computerspiele und bezeichnet die Umgebung in der sich der Anwender beziehungsweise der Spieler bewegt – also den Ort der Handlung, auch Spielwelt genannt.

In ein Level gehören, neben der reinen Architektur, auch Objekte und verschiedene Missionen – Aufgaben die der Spieler erfüllen muss, um innerhalb des Levels weiterzukommen oder das nächste Level zu erreichen (zum Beispiel eine Tür die geöffnet werden muss, um in den nächsten Raum zu gelangen).

Character sind im engeren Sinn Personen oder Lebewesen, die sich in den verschiedensten Levels befinden und diese mit Leben füllen. Im weitesten Sinne können jedoch sämtliche Gegenstände die sich bewegen und/oder mit denen eine Interaktion möglich ist als Charaktere bezeichnet werden. Für deren Erstellung sind wiederum spezielle Programme – sogenannte Animationstools notwendig.

Ein **Level- & Characterdesigner** ist demnach der Schöpfer einer künstlichen Welt. Ihm obliegt die Aufgabe, diese Spielumgebung:

- zu erstellen
- Objekte in ihr zu platzieren und
- Missionen einzubauen

Bei der Entwicklung von Spielen, kommen beim Leveldesign-Prozess inzwischen immer häufiger sogenannte Leveleditoren zum Einsatz. Bei einem solchen Editor handelt es sich um einen Modeller, der speziell zur Erstellung von Levels für eine spezielle Spiel-Engine entwickelt wurde. Der Vorteil solcher Programme ist, dass sie perfekt an die jeweilige Engine angepasst werden können und damit der Prozess des Leveldesigns schnell und Engine-konform geschehen kann. Egal ob ein spezieller Leveleditor oder ein herkömmliches Modellierungstool eingesetzt wird, als Ergebnis sollte ein möglichst atmosphärisches und beeindruckendes Szenario entstehen, das den Benutzer fesselt und in dem es Spaß macht sich zu bewegen.

Im Rahmen dieser Diplomarbeit sollen jedoch die Bereiche Missionsdesign und Entwurf (Skizzen,...) ebenso wenig betrachtet werden wie die Verwendung eines speziellen Leveleditors. Daraus ergeben sich folgende zu untersuchende Bereiche:

- Modellierung
- Texturierung
- Beleuchtung
- Animation
- Verteilung der Objekte

2.2.1 Modellierung

Die Erstellung 3-dimensionaler Objekte, genauer gesagt die geometrische Form der Oberflächen, wird als Modellierung bezeichnet (engl. Modelling – Formung oder Formgebung). Da die möglichen Methoden und Strategien vom Modeller abhängen, werden nachfolgend die Wichtigsten in ihren Charakteristiken.

Heutige 3D-Welten bestehen praktisch immer aus **Polygonen**. Diese Vielecke haben allerdings einen entscheidenden Nachteil bei der Erstellung gekrümmter beziehungsweise weicher, organisch wirkender Flächen, beispielsweise von Personen. Eine derartige Form aus Polygonen zu erzeugen, bedeutet den Einsatz einer relativ hohen Zahl an Einzelflächen, durch die die Handhabung eines solchen Modells sehr kompliziert wird [Himmelein 01, S.143].

Daher existieren neben dieser traditionellen Form der Modellierung weitere Geometrie-Werkzeuge, die den Umgang mit solchen Formen vereinfachen.

„Dem CAD-Bereich entstammen **Spline-Kurven und –Flächen**, die auch runde Figuren exakt beschreiben“ [Himmelein 01, S.143]. Diese Art der Flächendarstellung basiert auf mathematisch exakten Formeln und ermöglichen eine Formung mit Hilfe von Kontrollpunkten. Diese bieten dem Designer die Möglichkeit Oberflächenformen auf sehr einfache und intuitive Weise zu bearbeiten. Splines wurden erstmals 1940 zur Modellbeschreibung eingesetzt [Weisskopf 01]. Mittlerweile existieren viele verschiedene Spline-Formen, wie **Bézier-Patches**, **B-Splines** oder **Non-uniform Rational B-Splines** (NURBS). Die Schwierigkeit und zugleich größter Nachteil bei der Verwendung von Splines ist, dass die Herausmodellierung von Details an bestimmten Stellen des Objekts, eine Erhöhung der Komplexität der gesamten Oberfläche nach sich zieht. Soll das verhindert werden, muss das Objekt über sogenannte Patches modelliert werden – eine recht komplizierte Modellierungsmethode, bei der die Objektoberfläche aus Einzelflächen zusammengesetzt wird.

Subdivision Surfaces stellen eine relativ neue Art von Modellierungswerkzeug dar, das immer häufiger eingesetzt wird. Auch diese Flächenart basiert auf mathematischen Formeln. Der Designer arbeitet zwar an einem simplen Polygonmodell, das System führt jedoch vor der Darstellung Berechnungen daran durch, die die Oberflächen glätten und verfeinern [Himmelein 01, S.143]. Für Details können einzelne Flächenbereiche ausgewählt und in extra Ebenen bearbeitet werden, so dass die einfach zu manipulierende polygonale Grundform bestehen bleibt.

Egal welche Technik zum Einsatz kommt, bieten sich dem Designer 2 Herangehensweisen. Entweder wird das gewünschte Objekt, ähnlich dem Bildhau-Prozess, aus *einem* Körper (meist ein Grundkörper wie Kugel oder Würfel) geformt oder es wird aus mehreren Einzelteilen zusammengesetzt.

Die erste Variante eignet sich, orientiert an der realen Welt, besonders gut für die organische Modellierung (Charaktere). Dagegen sind technische Dinge erfahrungsgemäß so konstruiert, dass sie aus vielen Einzelteilen bestehen.

2.2.2 Texturierung

In den Bereich der Texturierung fallen 2 notwendige Arbeitsschritte:

- die Textur-Erstellung und dessen
- das Anbringen auf dem 3D-Objekt.

Zur Erstellung von Texturen kann sowohl 3D- als auch 2D-Software eingesetzt werden. Auch Photos und andere Bilder können geeignete Ausgangsmaterialien für Texturen sein. Im Normalfall werden jedoch 2D-Bildbearbeitungprogramme, wie zum Beispiel Photoshop von Adobe, Paint Shop Pro der Firma JascSoftware oder Corel's Painter verwendet.

Das Zuweisen eines Bildes beziehungsweise einer Textur zu einem Modell wird als **Texture-Mapping** bezeichnet [Daughtry 01, S. 106].

Einem 3D-Objekt lassen sich allerdings auch mehrere Texturen gleichzeitig zuordnen. Eine Möglichkeit besteht darin, dass die unterschiedlichen Bilder auf unterschiedliche Faces eines Objekts aufgetragen werden (**Multi-Materials**). Sollen jedoch mehrere Texturen über demselben Objektbereich erscheinen, muss die Render-Engine **Multi-Texture** fähig sein.

2.2.3 Beleuchtung

Beleuchtung bedeutet nicht nur ein einfaches Ausleuchten der 3D-Szene. Es ist, neben Farben (Texturen), ein Mittel zur Erzeugung von Atmosphäre und ermöglicht die räumlichen Erfahrung der Spielumgebung (Tiefe durch Licht und Schatten).

Dazu stehen dem Leveldesigner die im vorhergehenden Kapitel erläuterten Lichtquellen und Materialien sowie die Methoden zur Erzeugung von Schlagschatten und Spiegelungen zur Verfügung.

Abb. II-9 Screenshots eines 3D-Spiels, basierend auf der Q3A-Engine[9]

Die Grafiken der Abb. II-9 sind Beispiele für ein gelungenes Beleuchtungsdesign und überzeugen mit viel Atmosphäre.

[9] Die Q3A-Engine ist eine sehr leistungsfähige Game-Engine, die für viele 3D-Spielen genutzt wird. Nähere Informationen können unter http://www.idsoftware.com/corporate/idtech gefunden werden.

2.2.4 Animation

Wörtlich übersetzt, bedeutet Animation Belebung, Bewegung beziehungsweise Leben. In diesem Zusammenhang bedeutet es jedoch die Änderung von Objekteigenschaften innerhalb eines Zeitraums.

Generell lässt sich sagen: je mehr sich in einer Spielumgebung (ver-)ändert, desto lebendiger und „realistischer“ wirkt sie auf den Spieler. Je nach Game-Engine, lässt sich nahezu alles, was sich in der 3D-Szene befindet, animieren. Zum Beispiel:

- Objekte bzw. Objekteigenschaften
- Teile von Objekten
- Lichtquellen
- Texturen oder
- Kameras

Damit sich in der Spielumgebung etwas bewegt, kann im einfachsten Fall auf Game-Engine-spezifische Möglichkeiten zurückgegriffen werden (zum Beispiel die Rotation um eine der drei Raum-Achsen) oder die Bewegung wird im Vorfeld mit einem externen Animationsprogramm erstellt und bei Bedarf vom Spiel aufgerufen. Um ein Objekt, präziser gesagt dessen Eigenschaften, mit einem solchen Tool zu animieren, bieten sich je nach dessen Möglichkeiten die verschiedensten Methoden an. Allerdings muss darauf geachtet werden, dass sich diese Daten dann auch in die Spiel-Engine übertragen lassen.

Eine Technik, die von sehr vielen Animationstools unterstützt wird, ist die sogenannte **Keyframe-Animation**. Das Prinzip, das sich dahinter verbirgt, ist das Speichern der gewünschten Objekteigenschaften (es werden sogenannte Keys gesetzt) zu verschiedenen Zeiten (oder auch Frames). Allerdings wäre es extrem zeitaufwendig für jedes einzelne Bild der Animation entsprechende Änderungen vorzunehmen und zu speichern. Die Tatsache, dass sich ein Bewegungsablauf grob durch wenige charakteristische Posen beziehungsweise Positionen beschreiben lässt, hilft insoweit, dass zur Erstellung einer Animation „nur“ noch die gewünschten Eigenschaften an den Schlüsselpositionen gespeichert werden müssen und die fehlenden Zwischenschritte vom System berechnet werden.

Mit der Keyframe-Methode lassen sich nur sehr mühsam weiche Animationen erstellen, wie sie beispielsweise für Kamerafahrten nötig wären. Dafür stehen andere, hilfreichere Methoden zur Verfügung. Zum Beispiel die **Pfad-Animation** (**Path Animation**). Hierbei wird ein Objekt entlang eines vordefinierten Pfades bewegt, diese Wege werden mit Hilfe von Kurven (Splines) realisiert.

Zur Festlegung wann sich das Objekt an welcher Stelle des Pfades befinden soll, werden wieder Keyframes eingesetzt.

Eine weitere Form zur Animation von Objekten ist die **Skelett-Animation.** Ein Skelett ist in diesem Zusammenhang eine vereinfachte abstrakte Darstellung eines Objekts. Es setzt sich aus Gelenken - den Verbindungselementen (Joints) und den dazwischen befindlichen Knochen zusammen, ähnlich der Struktur einer Gliederpuppe.

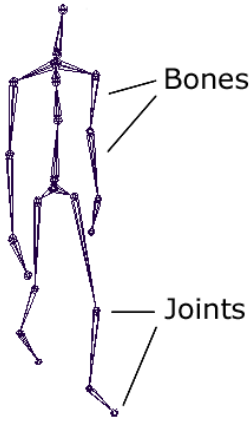

Abb. II-10 Ein Skelett mit seinen Bestandteilen

Werden dem Skelett die Objektgeometrien zugewiesen, vollziehen diese die Bewegungen des Skeletts nach. Dadurch können komplexe Bewegungen mehrerer zusammenhängender Teile wesentlich einfacher erstellt werden. Handelt es sich allerdings nicht um einzelne Objektteile, die den Gelenken zugeordnet werden können (Gliederpuppe) sondern um ein zusammenhängendes Mesh, muss dieses über ein Skinning-Verfahren an das Skelett gebunden werden (Binding). Beim sogenannten **Rigid Binding** wird jedem Vertex genau ein Joint zugeordnet. Infolgedessen kann es jedoch passieren, dass die Geometrie an den Gelenken einknickt und die Bewegungen eigentümlich starr wirken. Wesentlich bessere Ergebnisse ergibt das **Smooth Binding**, bei dem ein Punkt von mehreren Gelenken beeinflusst wird.

Zur Erstellung einer Skelett-Pose, müssten sämtliche Glieder in die entsprechende Position gebracht werden (Vorwärts- oder **Forward-Kinematik**). Da sich der Arbeitsaufwand dadurch nicht wesentlich verringert, wird die Skelett-Animation üblicherweise zusammen mit **Inverse-Kinematik** (IK) benutzt, die den Animationsprozess erheblich vereinfacht. Das Funktionsprinzip, das sich hinter dieser Umkehrung der klassischen mechanischen Bewegungsdefinition verbirgt, ist dass zunächst eine kinematische Kette definiert wird (über die Gelenke, die miteinander verbunden sind, beispielsweise beim menschlichen Arm: Schultergelenk-Ellenbogen-Handgelenk). In dieser Kette dürfen allerdings keine Verzeigungen auftreten [Immler 98, S. 1049]. Bei der menschlichen Hand müsste folglich für jeden einzelnen Finger eine solche IK-Kette angelegt werden. Zur Animation muss dann nur noch das Ende der Kette in die gewünschte Position gebracht werden und die übrigen Joints der Kette werden entsprechend mitbewegt.

Ein über IK animierbares Skelett, bietet eine ausgezeichnete Grundlage zur Erstellung natürlicher Bewegungen, wenn es allerdings um die Wiedergabe extrem komplizierter, meist menschlicher, Bewegungsabläufe geht, so zum Beispiel die Posen eines Kampfsportlers kommt oft ein Verfahren namens **Motion-Capture** zum Einsatz, mit dessen Hilfe die Bewegungen in Echtzeit aufgenommen werden. Beim optischen Motion Capture werden beispielsweise an einem Menschen Kontrollpunkte befestigt, die von einem Kamerasystem erfasst und direkt an einen Computer weitergeleitet werden. Bewegt sich die so präparierte Person, werden die Positionsänderungen der Markierungen aufgezeichnet.

Die so gewonnenen Animationsdaten können nach entsprechender Bearbeitung und Übertragung auf ein Skelett, durch ein hohes Maß an Realismus beeindrucken. Größter Nachteil dieses Verfahrens sind die immensen Kosten für ein solches System oder für dessen Nutzung, so dass Motion Capturing nur von sehr wenigen Firmen eingesetzt werden kann.

Abb. II-11 Der Profi-Golfspieler Tiger Woods beim Einsatz eines Motion Capturing Verfahrens

Electronic Arts (EA Sports) setzt dieses Verfahren sehr häufig zur Gewinnung von Animationsdaten für ihre Sportspiele ein. Im obigen Bild wird Tiger Woods Schlagbewegung für das neue Golfspiel „Tiger Woods PGA Tour 2002“ gecaptured.

2.2.5 Dynamics

Zum Abschluss sollen kurz Möglichkeiten erläutert werden, wie sich Objekte, basierend auf physikalischen und mathematischen Berechnungen, animieren lassen.

Partikel-Systeme werden dann eingesetzt, wenn es um die Erzeugung und Bewegung unzähliger Objekte geht (zum Beispiel eine Wasserfontäne, Rauch, Feuer, etc.) Diese werden dann automatisch, aus Angaben zu Richtung, Geschwindigkeit, Gravitaion sowie Zufallseinflüssen, animiert. Spiele können allerdings auch eine sogenannte **Physik-Engine** enthalten, die dazu verwendet werden kann, den Verlauf einer Bewegung, anhand physikalischer Eigenschaften (Masse, Reibung und andere Größen) zu errechnen. Das Ergebnis sind meist sehr überzeugend und realistisch wirkende Animationen.

Abb. II-12 Ein Screenshot aus dem Spiel „Midtown Madness“

Die qualmenden Reifen in Midtown Madness sind nur ein Beispiel wie Partikelsysteme verwendet werden können. Die Animation der Wagen errechnet die Physik-Engine des Spiels.

2.2.6 Qualitätskriterien

Computerspiele haben alle die gleiche Funktion - sie sollen den Anwender unterhalten. Dieser Aufgabe werden sie allerdings nur dann gerecht, wenn sie es schaffen, den Spieler zu fesseln und ihn in die virtuelle Welt eintauchen lassen, mit der er interagieren kann. Kurt Arnlund, Programmierer der Spielschmiede Accolade, sagte dazu folgenden treffenden Satz: "Erschaffe eine Spielumgebung, die so detailliert ist, dass der Spieler regelrecht in ihr verloren gehen kann - und letztlich vergisst, dass er eigentlich nur spielt." [Salt 00, S.26] Er spricht damit wesentliche Eigenschaften eines Spiels an, auf die der Level- und Characterdesigner einen direkten Einfluss hat:

Damit die künstliche Welt den Benutzer in ihren Bann ziehen kann, sollte sie detailreich sein – der realen Welt möglichst ähnlich sehen. Demnach darf ein Stadtszenario, wie es in „Skate Attack“ der Fall sein wird, nicht nur Häuser und Straßen bestehen. Es müssen Autos, Fußgänger, Straßenschilder, Papierkörbe, etc. vorhanden sein. Allerdings ist nicht nur die Zahl der Objekte wichtig, sondern auch deren technische Umsetzung. Fahrzeuge können beispielsweise durch Quader repräsentiert werden, eine glaubhafte Wirkung wird dadurch allerdings nicht erzielt. Dafür müssen vom Designer objekttypische Geometrien modelliert und mit entsprechenden Feinheiten versehen werden (im Falle des Fahrzeugs beispielsweise Außenspiegel oder Stoßstangen).

Details werden aber nicht nur über Geometrien geschaffen – Texturen auf denen Einzelheiten zu erkennen sind, können den Objekten eine charakteristische Oberflächenstruktur geben und damit das visuelle Ergebnis erheblich verbessern.

Ferner sollte sich das Umfeld dem Spieler gegenüber nicht starr verhalten. Einerseits könnte der Anwender schnell gelangweilt sein, andererseits sind Animationen wichtig für eine überzeugende Illusion. So wie sich in der realen Welt durch Passanten, Verkehr oder sich drehende Werbeschilder ein reges Stadtbild entsteht, muss auch die Spielumgebung „lebendig“ sein.

Licht und Schatten sind ebenfalls von großer Bedeutung. Durch die Beleuchtung erfährt die Szene nicht nur mehr Tiefe, sondern auch Atmosphäre – eine für das Spielerlebnis sehr wichtige Komponente. Außerdem dienen Schatten zur besseren Orientierung des Benutzers.

Reich an Details bedeutet aber auch Abwechslung und Vielfältigkeit. Zum Beispiel sehen Personen normalerweise nie gleich aus und bewegen sich individuell, im Straßenverkehr sind gleiche Fahrzeuge derselben Farbe eher eine Ausnahme und die Fenster einer Hausfassade unterscheiden sich durch Gardinen, Jalousien oder dadurch, dass sie offen oder zu sind. All dies muss vom Designer in den Bereichen Modellierung, Texturierung, Beleuchtung und Animation beachtet werden, damit das Spiel möglichst realistisch und für den Benutzer interessant wird.

Neben der optischen Wirkung der Spielumgebung hat auch die Frame-Rate einen wesentlichen Einfluss auf das Spielerlebnis. Sie sollte möglichst hoch und konstant sein, da hohe Schwankungen zu ruckenden Bildern führen und den Spielfluss mindern. Zusätzlich können die Interaktionsmöglichkeiten des Spielers entscheidend negativ beeinflusst werden, wodurch der Spielspass weiter sinkt. Doch dieser stellt das Ziel einer solchen Anwendung dar. Bei Beachtung aller Qualitätsmerkmale, sollte als Ergebnis ein attraktives, den Anwender mitreißendes Spiel-Szenario entstehen.

III Analyse

1 Rahmenbedingungen

Wie im Abschnitt „PC kontra Konsole“ des vorhergehenden Kapitels beschrieben, ist die Spielentwicklung für den PC-Bereich sehr aufwendig. Da es sich bei „Skate Attack“ um ZeroScales erstes 3D-Computerspiel handelt, für das eigens eine Game-Engine programmiert wird, ist die Zielplattform eine Konsole - Microsofts Xbox. Neben der einfacheren Entwicklungsmöglichkeit bietet der Konsolenbereich auch die vom Spiel angestrebte Zielgruppe sowie erforderliche Eingabegeräte – Gamepads, die zur Standardausrüstung dieser Geräte gehören.

Wie einleitend bereits erwähnt, soll das Spiel eine Skateboard-Simulation werden, die dem Anwender neben einer rasanten 3D-Grafik auch Action-Elemente bietet. Die Stadt, in der sich der Spieler völlig frei bewegen kann, soll möglichst groß und mit vielen Objekten gefüllt sein, so dass viele Interaktionsmöglichkeiten entstehen.

1.1 Das Entwicklungssystem

Egal für welche Zielplattform ein Computerspiel entwickelt wird, es kommen üblicherweise Windows-PCs als Entwicklungsplattform zum Einsatz. Ein relativ leistungsstarker und somit für die Entwicklung gut geeigneter PC ist mittlerweile preisgünstig und zudem bietet sich dem Benutzer eine breite Palette aus inzwischen ausgereifter Software, die zur Produktion nötig ist.

Da sich Xbox und PC zudem sehr ähneln, lässt sich im Falle von „Skate Attack“ der Großteil der Entwicklung, ohne mit größeren Schwierigkeiten bei der Konvertierung rechnen zu müssen, auf PCs erledigen. Die Hardware-Ausstattung reicht dabei von 600 bis 1457 MHz-CPUs bis zu Grafikkarten von Nvidia der GeForce-Generation.

Als Betriebssystem wird Microsofts Windows 2000 eingesetzt, da es ein verhältnismäßig stabiles System ist und die Xbox aufgrund der Netzwerkfähigkeit ein sehr ähnliches System verwendet.

Die Schnittstelle der Xbox wird zwar DirectX sein, bezüglich [Abrash 01] allerdings in einer abgewandelten Form. Normalerweise muss DirectX ein großes Spektrum unterschiedlichster Hardware und Systemdienste unterstützen. Die Xbox-Version konnte dagegen speziell auf diese Plattform abgestimmt werden. Außerdem werden neue Funktionen, entsprechend den technischen Möglichkeiten der Konsole, enthalten sein.

1.2 3D-Game-Engine

Die schon des öfteren erwähnte Game-Engine, die für „SkateAttack“ entwickelt wird, trägt den Namen Prana-Engine. Sie basiert auf DirectX 8 und ist bereits funktionsfähig. Die Eigenentwicklung der Game-Engine bietet neben dem Vorteil, dass sie genau an die Bedürfnisse des Spiels und etwaigen Forderungen seitens des Level- & Characterdesigns angepasst werden kann, auch die Möglichkeit leicht Veränderungen vorzunehmen beziehungsweise mehr Features zu implementieren. Darüber hinaus ist sie nicht an Lizenzgebühren gebunden. Demgegenüber stehen allerdings relativ hohe Entwicklungskosten (Zeit und Geld).

Zum derzeitigen Entwicklungsstand unterstützt sie folgende, für den Bereich Level- und Charakterdesign interessante, Features:

- Polygonale Geometrien
  - Vertex Colors
- Materialien
  - Farbkanäle für ambiente, diffuse, spiegelnde Reflexion und Selbstleuchten (Emmission)
  - Multi-Materials
- Texturen
  - Datei-Formate: Bitmap, DDS, und Targa
  - Alpha-Kanal
  - Multi Textures
  - ...
- Beleuchtung
  - Ambientes , Punkt sowie Gerichtetes Licht
  - Gouraud Shading
  - Volumetrische Schatten
- Animation
  - Keyframe-Interpolation
  - Pfad-Animation
  - Skelett-Animation mit IK
  - ...
- Dynamics – Partikelsysteme
- LOD
  - Static LOD
- Billboards
- Culling Techniken
  - View Frustum Culling
  - Backface Culling
  - Portal Culling
  - ...
- Scriptsprache – Python
- ...

1.3 Modellierungs- und Animationswerkzeug

Zur Erstellung der Level und Charaktere sowie deren Animationen, kurzum zur Erzeugung der gesamten Spielwelt, wird Maya von Alias Wavefront[10] eingesetzt. Selbst Menüs und Elemente des Spiel-Interfaces lassen sich mit dessen Hilfe arrangieren. Bei Maya handelt es sich um ein 3D-Komplettpaket, das ursprünglich für Charakter-Modelling und Animation sowie für Visuelle Effekte konzipiert war und in erster Linie im digitalen HighEnd-Bereich, wie Film und Werbung, eingesetzt wurde und immer noch wird.

Mittlerweile wird Maya, aufgrund seiner Erweiterungen für den Echtzeitbereich, immer häufiger für Spielentwicklungen eingesetzt. So benutzte zum Beispiel Sony Computer Entertainment Inc. bei der Entwicklung seiner Rennwagensimulation „Grand Turismo 3“ fast ausschließlich Maya. Aber auch für andere Spiel-Hersteller wie Electronic Arts (NHL 2001), NAMCO (Tekken Tag Tournament), oder Midway Games Inc (Ready 2 Rumble Boxing Round 2) ist Maya inzwischen ein fester Bestandteil in der Entwicklungs-Pipeline.

Für ZeroScale kam Maya in erster Linie aufgrund der Philosophie, die hinter diesem Programm steckt, in Betracht. Das heißt die objektorientierte Verwaltung der Szenen-Elemente und dem dazugehörigen Connection-System, das die Verbindungen zwischen den Elementen beschreibt. Hinzu kommt die Möglichkeit, mit der integrierten Scriptsprache MEL auf die gesamte interne Struktur zugreifen zu können. Wie „mächtig“ diese Scriptsprache ist, lässt sich daran ersehen, dass die komplette Benutzeroberfläche von Maya in MEL entwickelt ist. Dass ermöglicht natürlich auch dem Benutzer eine absolut freie an spezifische Bedürfnisse gebundene Gestaltung des Interfaces und damit eine Erleichterung der Arbeitsweise.

[10] http://www.aw.sgi.com

Aufgrund der Anpassung des Interfaces kann darauf verzichtet werden, einen eigenen Level-Editor für „SkateAttack“ zu entwickeln, dessen Vorteil es wäre, dass er genau auf die Eigenschaften der Prana-Engine abgestimmt werden könnte, was im Endeffekt zu einer besseren Performance führt und den Arbeitsaufwand minimiert.

Ein weiterer Vorteil, der durch die Verwendung nur eines Tools für die Realisierung des Projektes entsteht, ist ‚dass keine Daten zwischen verschiedenen Programmen ausgetauscht werden müssen, was nicht nur Zeit spart sondern auch vor Kompatibilitätsproblemen schützt. Davon wären besonders Objekte mit zusätzlichen Informationen speziell für die Prana-Engine betroffen.

1.3.1 Modellierungstechniken

Die Version 4.0 von Maya, bietet zur Erzeugung von 3D-Objekten drei verschiedene Modellierungstechniken mit entsprechenden Hilfsmitteln zur Bearbeitung an:

- Polygone
- NURBS-Kurven und -Flächen oder
- Subdivision Surfaces.

Zusätzlich bietet Maya die Möglichkeit, die verschiedenen Geometrie-Arten ineinander umzuwandeln. So lassen sich die parametrischen Formen durch Polygonnetze annähern, dieser Konvertierungsprozess wird auch Tesselierung genannt. Den Grad der Annäherung, dass heißt aus wie vielen Einzelflächen das neue Objekt bestehen soll, bestimmt der Benutzer durch Angabe verschiedener Parameter. Aus Polygonen und Subdivision Surfaces lassen sich allerdings keine NURBS-Flächen erzeugen.

Das für das Projekt „Skate-Attack“ die polygonale Modellierung der Level und Charaktere am besten geeignet ist, kann wie folgt begründet werden:

- Die Prana-Engine verarbeitet nur Polygone als Geometrieform. Die Objekterstellung über NURBS- oder Subdivsion-Flächen würde damit nur die Arbeit innerhalb des Modellierungstools erleichtern und eine anschließende Konvertierung bedeuten.
- Da die Spielumgebung ein komplexes und vielfältiges Stadtszenario beschreiben soll, können einzelnen Objekte - egal ob Gegenstände oder lebende Kreaturen - zu keiner Zeit aus einer unüberschaubaren Zahl an Polygonen bestehen. Hinzukommt, dass für die Stadtarchitektur einfache plane Flächen eingesetzt werden können (Häuser, Straßen, etc.).

Damit fehlt in diesem Fall die Notwendigkeit NURBS- oder Subdivision Flächen als Geometrieformen einzusetzen.

Bei der Frage, aus wie vielen Polygonen eine komplette 3D-Szene in „SkateAttack“ besitzen darf, woraus sich dann abschätzen lässt, wie viel für einzelne Objekte bleiben, lässt sich nicht einfach beantworten. Die Xbox ist zwar in der Lage 125 Millionen Dreiecke pro Sekunde (MTris/sec) zu berechnen, aber dieser Wert hat keinen wahren praktischen Nutzen, denn die Angabe bezieht sich nur auf die Möglichkeit polygonale Dreiecke zu berechnen, ohne Materialien, Texturen, Beleuchtung und so weiter. Soll mehr als nur Geometrie verarbeitet werden, ergeben sich sehr schnell wesentlich kleinere Werte:

- 1 unendliche Lichtquelle, 1 Textur ~ 85 MTris/sec
- 1 unendliche Lichtquelle, 2 Texturen ~ 75 Mtris/sec
- 1 unendliche Lichtquelle, 2 Spotlichter,2 Texturen ~ 20 MTris/sec

[Abrash 01 B]

Soll das Spiel dann noch mit einer Frame-Rate von 60 oder mehr laufen, blieben beim letzten Beispiel der Tabelle (20 MTris/sec) für die 3D-Szene „nur“ noch rund 334.000 Dreiecke. Da sich mit 2 Texturen aber kein abwechslungsreiches Umfeld schaffen lässt, zudem noch Animationen, Partikeleffekte, Schatten etc. fehlen, wird der ursprünglich so hohe Wert von 125 Mio. immer geringer und zeigt, dass auch die modernste Konsole ihre Leistungsgrenzen hat. Damit wird deutlich, wie sehr ein Level durchdacht sein muss, damit aufgrund der maximalen Gesamtzahl an Dreiecken sich ungefähr abschätzen lässt, wie viele Polygone für einzelne Objekte verwendet werden können. Eine derartige Planung ist jedoch nicht so einfach möglich. Neben der Xbox spielt die Prana-Engine eine entscheidende Rolle, die sich allerdings noch in der Entwicklung befindet. So werden stets neue Features implementiert oder auch ältere wieder verändert oder herausgenommen, wodurch sich wieder andere Möglichkeiten ergeben, ein bestimmtes Ergebnis zu erzielen. In diesem Fall findet der Erstellungsprozess der Level und Character so statt, dass von Anfang an darauf geachtet werden muss, mit so wenig wie möglich Datenaufwand auszukommen und die derzeit zur Verfügung stehenden Features der Engine zu nutzen, um bestmögliche Ergebnisse zu erzielen. Die Erzeugung von Polygon-Modellen mit sehr geringer Flächenzahl wird auch als Low-Poly Modellierung bezeichnet.

Auch aus parametrischen Flächen lassen sich, nach einer Umwandlung in Polygongeometrien, Low-Poly Objekte erstellen – entweder bei der Tesselierung oder durch eine anschließende Reduzierung der Flächenzahl. Beides ist zwar möglich, führt jedoch nur im High-Poly Bereich zu wirklich sinnvollen Ergebnissen. Dies bedeutet, dass sich beispielsweise ein Charakter der aus 10.000 Einzelflächen besteht, problemlos auf 5.000 verringern lässt. Meshes die dagegen nur aus 10 bis 1000 Einzelflächen bestehen dürfen, erfordern vom Designer eine maximale Konzentration auf die charakteristische Form (Shape). Ein Programm hingegen „versteht“ nicht was das Objekt, das reduziert werden soll (beispielsweise ein menschliches Bein und eine Regenrinne), verkörpert. Demzufolge gehen sehr schnell Details an den falschen Stellen verloren, so dass eine manuelle Nachbearbeitung meist unumgänglich ist und die Verwendung solcher Geometrien keine Arbeitserleichterung mehr darstellt.

1.4 Exporter

Damit die mit Maya erstellten Animations- und Szenendaten (Geometrien, Lichter, Kameras usw.) von der Prana-Engine verarbeitet werden können, ist ein sogenannter Exporter notwendig. Dieser ist als Plugin für Maya realisiert und bietet dem Designer die Möglichkeit Objekten bei der Modellierung Attribute zu geben, die es ermöglichen Features der Engine zu nutzen. Anschließend kann und muss die Szene in ein Prana-eigenes Dateiformat umgewandelt werden. Dabei werden die 3D-Objekte automatisch tesseliert.

Der Exporter ermöglicht aber auch die Vorschau der aktuellen 3D-Szene, direkt aus Maya heraus. Das hat den Vorteil, dass die Szene noch im Design-Prozess auf Funktionalität überprüft werden kann, ohne sie ins Spiel einbinden zu müssen.

1.5 Zusatz-Software

Ein weiteres wichtiges Werkzeug auf das nicht verzichtet werden kann, ist eine Applikation zur Textur-Erstellung. Dafür wird das wohl bekannteste und am meisten verbreitete Programm - Adobes Photoshop - genutzt. Es bietet dem Anwender eine bemerkenswert großes Repertoire an Hilfsmitteln um Bilder zu erstellen oder vorhandenes Material so zu verändern, dass daraus Texturen gewonnen und in üblichen Dateiformaten abgespeichert werden können.

Um das speziell für den Spielbereich von DirectX entwickelte DirectDrawSurface-Format (DDS) mit der Prana-Engine nutzen zu können, ist ein DDS-Konverter notwendig, mit dessen Hilfe die herkömmlichen Bildformate (BMP, TARGA, ...) umgewandelt werden können. Dieser Konverter ist einerseits als Plugin für Photoshop erhältlich, andererseits als eigenständiges Programm - das sogenannte DxTex-Tool. Dieses ist so integriert, dass die Prana-Engine beim Start der 3D-Szenen Texturen die nicht im DDS-Format vorliegen zunächst konvertiert. Die Vorteile, die das DDS-Format bietet, sind neben dem Alpha-Kanal Texturkompression (DXTn) und MipMapping. Nähere Beschreibungen dazu können unter 2.2.2 und 2.2.3 im anschließenden Teil dieses Kapitels gefunden werden.

Auch wenn damit prinzipiell alle Bereiche mit entsprechender Software abgedeckt sind, sollten stets die gesamte, am Markt erhältliche, Produktpalette an Werkzeugen, Zusatzmodulen, kleine hilfreiche Tools und so weiter, im Auge behalten und nach Möglichkeit ausprobiert werden. Oft sind es die unscheinbaren Programme von Drittherstellern, die die Arbeit auf einem speziellen Gebiet erleichtern und damit schneller zu gewünschten Ergebnissen führen.

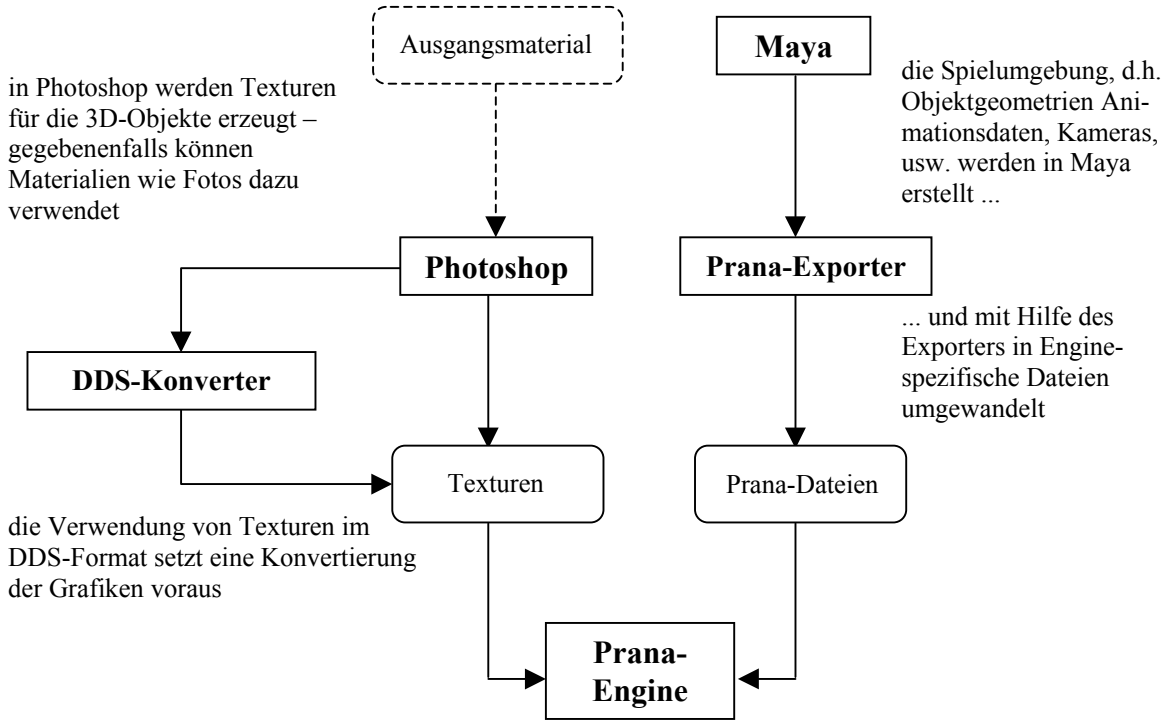

Abb. III-1 Schematische Darstellung des Design-Prozesses der Spielumgebung

2 Allgemeine Optimierungstechniken

Wie im Kapitel II „Grundlagen“ unter 2.2.6 „Qualitätskriterien“ beschrieben, sollte sowohl die Bildqualität als auch die Frame-Rate eines Spiels möglichst hoch sein. Da sämtliche Maßnahmen zur Verbesserung der Optik und Glaubhaftigkeit der Spielumgebung Unmengen an Daten (Geometrien, Texturen, Animationen, ...) verursachen, die transportiert und zur Laufzeit berechnet werden müssen, steigt die Renderzeit für ein einzelnes Bild und die Frame-Rate sinkt.

Genau da setzten verschiedenste Optimierungstechniken an, deren Ziel es einerseits ist, die Datenmenge zu minimieren und andererseits den Berechnungsaufwand, den diese Daten verursachen, so gering wie möglich zu halten, um höhere Frame-Raten zu erzielen.

2.1 Modellierungsbereich

Szenen, die in 3D-Spielen dargestellt werden, dürfen eine gewisse Anzahl an Polygonen, die zur gleichen Zeit sichtbar sind, nicht überschreiten um Echtzeit und damit eine exakte Interaktion zu gewährleisten . Die Grenzen werden sowohl von der verwendeten Plattform als auch der Game-Engine bestimmt. Somit unterliegt die Objekt-Modellierung strengen Vorgaben, wobei grundsätzlich gilt: nur so viele Polygone wie nötig zu verwenden.

Laut [Microsoft 01] ist DirectX daraufhin optimiert Polygone schubweise und in großer Zahl zu rendern. Je mehr Polygone mit einem Aufruf (Call) gezeichnet werden können, desto besser (für jedes Objekt der Szene muss 1 sogenannter Call durchgeführt werden). Nach Möglichkeit sollten mit einem Call durchschnittlich über 100 Polygone gezeichnet werden. Anderenfalls würde die maximale Leistung nicht ausgeschöpft oder andere Prozesse gestört, die sonst parallel dazu stattfinden könnten. [Huddy 99] empfiehlt sogar 200 oder mehr.

2.1.1 Instanzen

Durch den Einsatz von Instanzen ist es möglich eine 3D-Szene komplexer zu gestalten, ohne das der Datenaufwand sowohl auf dem Speichermedium als auch im Hauptspeicher entsprechend mitsteigt. Für die Kopien werden nur noch die Transformationsdaten benötigt - siehe 1.6.2 „Geometrie-Phase“ im Kapitel II „Grundlagen“.

Da die Objekte völlig gleich aussehen, scheint der Einsatz von Duplikaten sehr stark eingeschränkt zu sein, dennoch bieten sich entsprechend der realen Welt viele Möglichkeiten. Für das Stadtszenario von „Skate Attack“ wären dass zum Beispiel Parkbänke, Fahrzeuge, oder Straßenlaternen - selbst Personen können auf diese Weise geometriesparend eingesetzt werden.

Um zu verhindern dass die Spielumgebung zu monoton wird und sich der Spieler dadurch langweilt, können Instanzen auch mit unterschiedlichen Materialien beziehungsweise Texturen versehen werden.

2.1.2 Level Of Detail

Hinter dem Begriff Level Of Detail (LOD) verbergen sich eine Reihe sehr wichtiger Methoden zur automatischen und dynamischen Kontrolle der Komplexität einer 3D-Szene, die besonders im Bereich der interaktiven 3D-Echtzeit einen hohen Stellenwert haben.

Die zugrundliegende Idee ist folgende: Sämtliche Objekte einer Szene befinden sich in unterschiedlicher Entfernung zum Betrachter. Da das menschliche Auge, wie im echten Leben, nur bei den Objekten, die in unmittelbarer Nähe stehen, Details erkennen kann, ist es auch sinnvoll nur an diesen entsprechende Feinheiten darzustellen – für Objekte die sich weiter vom Betrachter wegbefinden ist eine geringere Detailgenauigkeit ausreichend. Die Verwendung eines weniger komplexen 3D-Modells für entferntere Objekte hat den Vorteil dass Rechenzeit gespart werden würde. Genau dieses Ziel wird mit unterschiedlichen LOD-Strategien verfolgt, von denen nachfolgend die wichtigsten untersucht werden.

Bei **Static LOD** handelt es sich um die konventionellste Methode, auf die vom Level- & Characterdesigner direkt Einfluss genommen werden kann. Die prinzipielle Vorgehens- und Funktionsweise sieht folgendermaßen aus:

- In einem Vorprozess werden vom Designer mehrere Versionen des Modells in unterschiedlichen Detaillierungsgraden erstellt. Diese reichen von sehr einfachen Grundkörpern bis hin zur möglichst realistischen Nachbildung.
- Durch das anschließende Zusammenfassen der Objektstufen, entsteht eine sogenannte LOD-Gruppe.
- In welcher Entfernung die nächst gröbere Version des Objekts angezeigt werden soll, wird vom Designer im sogenannten Threshold-Attribut festgelegt.
- Entsprechend den Threshold-Werten, werden einzelne Objekte dieser Gruppe sichtbar gemacht - die anderen ausgeblendet. Diese Aufgabe wird von der Game-Engine zur Laufzeit übernommen (siehe 1.6.1 „Applikation-Phase“ in Kapitel II „Grundlagen“).

Die Vorteile dieser Methode sind laut [Luebke 00, A7], dass

- die Berechnung der vereinfachten Objektmodelle nicht zur Laufzeit stattfinden und sich somit wertvolle Rechenzeit sparen lässt
- das interne Management eines solchen Systems recht simpel und damit der dafür notwendige Rechenaufwand sehr gering ist
- die einzelnen LOD-Stufen sich bezüglich der verwendeten Plattform (Grafikhardware) und Render-Engine optimieren lassen.

Wesentliche Nachteile von Static LOD sind, neben dem erhöhte Arbeitsaufwand für den Designer:

- dem System stehen relativ wenig verschiedene Stufen zur Auswahl
- die Stellen, an denen zwischen zwei verschiedenen Stufen gewechselt wird, können bei zu großen Unterschieden zu sichtlichen Sprüngen führen (auch „Popping“ oder „Pop-Effekt“ genannt).

Um die Auffälligkeit dieses Effekts so gering wie möglich zu halten, gibt es 3 verschiedene Maßnahmen. Entweder wird eine weitere Zwischenstufe angefertigt, der Wert für die Entfernung, bei der zwischen zwei Objekten gewechselt wird, erhöht oder die Unterschiede zwischen den Abstufungen verringert. Diese Maßnahmen können jedoch wieder dazu führen, dass mehr Polygone gleichzeitig dargestellt werden müssen.

Static LOD hat jedoch seine Grenzen. Spätestens bei extrem großen Geometriemodellen wie Landschaften oder Modellen, die aus mehreren tausend Polygonen bestehen (im CAD-Bereich keine Seltenheit), ist eine manuelle Vereinfachung sehr schwierig oder unmöglich. Eine automatische Reduktion dieser Objekte wird als **Dynamic LOD** bezeichnet [Luebke 00, A13]. Dieser Vorgang wird vom System übernommen und findet während der Laufzeit statt. Neben dem Einsatz in CAD-Bereich oder bei Simulationen, wird dieses Verfahren auch in Computerspielen eingesetzt. Dynamic LOD entbindet den Designer zwar vom Anfertigen der verschiedenen Detailstufen, verursacht dadurch aber mehr Rechenaufwand, da die Abstufungen zur Laufzeit generiert werden. Eine Optimierung dieser Geometrien ist daher kaum bis gar nicht möglich. Der größte Vorteil dieser LOD-Form ist das Ausbleiben des „Pop"-Effekts. Da sich für jede Entfernung exakt benötigte Modellstufen generieren lassen, verlaufen die Übergänge viel „weicher" und sind vom Betrachter im Prinzip nicht mehr wahrzunehmen. Dieses LOD-System ist für den Einsatz in „Skate Attack" allerdings weniger geeignet. Das Stadtszenario besteht fast ausschließlich aus planen Geometrien und die Polygonmeshes der Objekte werden nicht aus unzähligen Einzelflächen bestehen können, siehe Kapitel III, 1 „Rahmenbedingungen".

2.1.3 Billboard

Ein Billboard, ist ein Objekt, dass stets zum Betrachter zeigt. Das hat den Vorteil, dass nur dessen Vorderseite existieren muss. Sämtliche Polygone, die für die Rückseite nötig wären, können auf diese Weise eingespart werden. Im einfachsten Fall könnte dies eine Fläche mit einer Textur sein – (a) der Abb. III-2. Eine Kameraorientierung wird jedoch nur möglich, wenn das Objekt um mindestens eine Raum-Achse drehbar gelagert ist. In diesem Fall würde der Spieler ein Billboard als solches erkennen, wenn er aus einer Richtung schaut, in das es sich nicht drehen kann – wie in (b) der Abb. III-2 dargestellt ist.

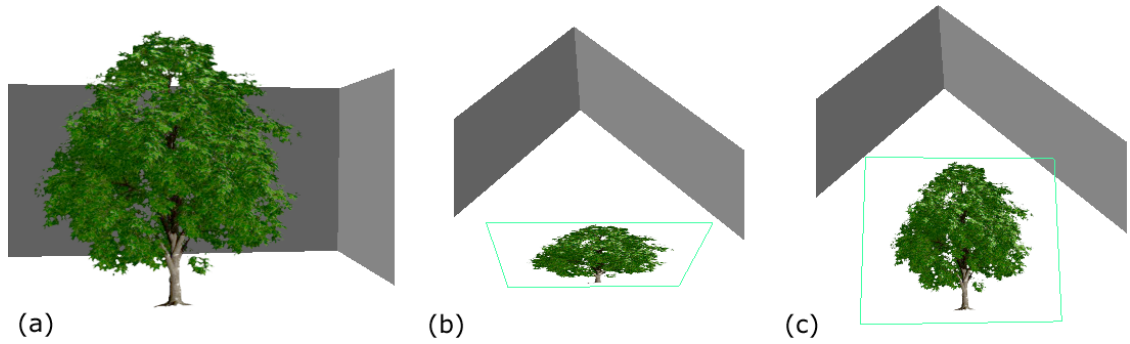

Abb. III-2 Ein mit Hilfe eines Billboards realisierter Baum

Aber selbst mit Hilfe mehrere Raumachsen, tritt noch ein erkennbares Problem auf. Wie (c) der Abb. III-2 zeigt, scheint sich der Baum von oben betrachtet „auf den Boden zu legen" und zerstört dadurch die Illusion. Daraus ergibt sich für Billboards ein stark begrenztes Einsatzgebiet auf Objekte mit bezüglich der Raumachsen rotationssymmetrischen Formen. Wenn der Spieler allerdings keine Gelegenheit mehr hat, diese von oben oder unten zu betrachten, kommen auch Formen in Frage, die um weniger Raumachsen rotationssymmetrisch sind.

2.1.4 Culling-Techniken

Nach [Möller 99, Kapitel 7] bedeutet Culling – das Selektieren aus einer Gruppe. In diesem Zusammenhang bedeutet dass die Entfernung der Objekt und Flächen, die vom Spieler nicht gesehen werden können. Wenn sich jedoch Objekte aufgrund ihrer Lage im Raum, ganz oder teilweise verdecken, können diese Flächen nur über spezielle HSR-Verfahren ermittelt und entfernt werden, die hier nicht näher erläutert werden.

Das in den Grundlagen zur 3D-Grafik beschriebene **View-Frustum Culling** ist ein Vertreter dieser Techniken. Da die Bounding Boxen der Objekte gegen das Sichtvolumen überprüft werden, hat der Designer einen Einfluss auf die Effizienz der Methode, wie die folgende Überlegung zeigt: Ein extrem großes Objekt besitzt eine entsprechend mächtige Bounding Box. Befindet sich der Spieler in einer Position, von der nur ein sehr geringer Teil des Objektes beziehungsweise der Box zu sehen ist, würde der Culling-Test negativ ausfallen und die gesamten Geometriedaten zur Verarbeitung an die Grafikkarte weitergeleitet werden. Die Unterteilung des Objekts hätte in diesem Fall einen Vorteil, steht jedoch im Gegensatz zu den Optimierungsempfehlungen des Geometriebereiches, wonach die Anzahl der Polygone möglichst hoch sein sollte. Dieses Beispiel zeigt einmal mehr, dass sich der Optimierungsprozess als Balanceakt darstellt.

Beim **Back-Face Culling** handelt es sich um ein weiteres Optimierungsverfahren seitens der Hardware. Es bestimmt und entfernt die Objektflächen, deren Normalen vom Betrachter wegzeigen und somit eindeutig nicht sichtbar sind. Dadurch kann der Anteil an Dreiecken einer komplexen Szene, durchschnittlich um etwa die Hälfte verringert werden.

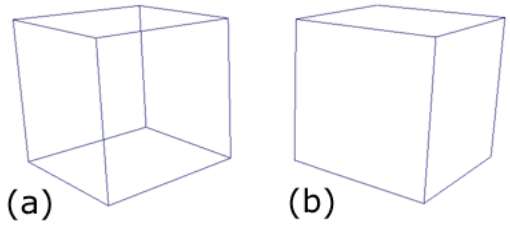

Abb. III-3 Verwendung von Back-Face Culling, dargestellt an einem Würfel

Back-Face Culling ist aufgrund seines geringen Rechenaufwandes das wahrscheinlich "günstigste" Verfahren dieser Art und dennoch sehr effizient. Der Designer hat zwar keinen Einfluss darauf, muss allerdings bei der Modellierung auf die Auswirkungen achten, was im nachfolgenden Kapitel näher beschrieben wird.

Eine weitere Methode auf die der Level- und Characterdesigner einen direkten Einfluss hat, ist das **Portal Culling**. Dieses Verfahren beruht auf der Tatsache, dass in architektonischen Levels (z.B. Räume) Objekte vorhanden sind, die dem Spieler den Blick auf große Teile der dahinterliegenden Geometrie der Spielumgebung versperren (z.B. Wände). Bei einer im Vorfeld stattfindenden möglichst sinnvollen Unterteilung der 3D-Szene in Abschnitte fungieren die Schnittstellen angrenzender Bereiche, die ein Hindurchblicken des Betrachters ermöglichen (z.B. Türen), als sogenannte Portale [Möller 99, Kapitel 7]. Zur Laufzeit werden die Portale daraufhin geprüft, ob sie vom Spieler aus sichtbar sind. Ist das der Fall müssen die Objekte der angrenzenden Bereiche gerendert werden.

2.2 Textur-Bereich

Durch die Einschränkungen im Modellierungsbereich gehen Details verloren, was der Glaubhaftigkeit der Darstellung schadet. Um die Beschaffenheit und typischen Merkmale der Objekte wieder „sichtbar“ werden zu lassen, kommen Texturen zum Einsatz. In erster Linie dienen sie zur Oberflächengestaltung von Objekten.

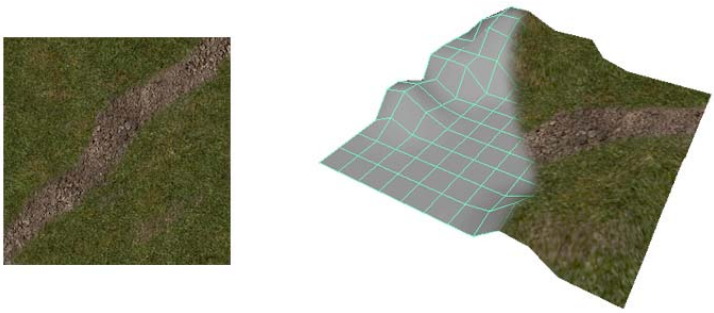

Abb. III-4 zeigt eine Textur (links) und eine partiell damit bedeckte „Polygon-Landschaft“(rechts)

Wie das obige Beispiel demonstriert, kann mit Hilfe von Texturen die Oberfläche auf sehr einfache Art und Weise optisch verbessert werden. Allerdings bieten Texturen weitaus eindrucksvollere Möglichkeiten, als nur Strukturen unterschiedlicher Komplexität, wie Sand- und Grasflächen oder Holzmaserungen darzustellen. Mit ihrer Hilfe lassen sich beispielsweise:

- Details (Nägel oder Schrauben in Wänden, Falten von Haut oder Kleidung, Haare, usw.)
- Beschriftungen
- Transparenzen (von Glas, etc.)
- Spiegelungen
- Glanzeffekte (von z.B. Metallen) oder auch
- Licht und Schatten

darstellen, beziehungsweise vortäuschen.

Abb. III-5 Ein Screenshots des Spiels „Tony Hawk“

Neben der Verbesserung der Optik, tragen Texturen erheblich zur Einsparung von Geometrien bei. Details wie zum Beispiel Schnürsenkel an Schuhen wären nicht nur aus Sicht der Modellierung ein erheblicher Mehraufwand, sondern wären auch seitens der Game-Engine nahezu unmöglich zu berechnen.

2.2.1 Auflösung

Um den benötigten Speicherplatz so gering wie möglich zu halten, sollte die Auflösung einer Textur relativ klein sein. Der Nachteil einer zu gering dimensionierten Textur zeigt sich, wenn der Spieler direkt vor einem Objekt mit einer solchen Textur steht – denn dann werden mehreren Pixel, der Farbwert eines Texels zugewiesen. Das daraus resultierende Ergebnis sind grob wirkende Pixelblöcke (Teil (b) der unteren Abbildung). In diesem Fall hilft eine Textur höherer Auflösung.

Allgemein gilt, je größer die Dimension einer Grafik, desto besser die Bildqualität. Dies zieht wiederum einen höheren Speicherbedarf nach sich.

Die einzige Möglichkeit ist dies allerdings nicht, da moderne Grafikkarten Filtermethoden anbieten, die den zuvor geschilderten Effekt ebenfalls verhindern. **Bilineares Filtering** beispielsweise nimmt zur Berechnung des Farbwertes eines Pixels nicht nur den Farbwert eines Texels, sondern benutzt dazu vier angrenzende Texel (links, rechts, oben, unten) [DirectX 00, „Linear Texture Filtering“]. Diese Methode findet in der dritten Phase der Grafikpipeline statt.

Abb. III-6 Ein virtueller PKW mit Detailansichten der Fahrertür

Die Abbildung zeigt ein texturiertes 3D-Modell eines Fahrzeugs, dessen Details aus der Entfernung korrekt dargestellt werden (a). Ein geringer Abstand führt zu erkennbaren Pixelgruppen (b), die jedoch von der Grafikkarte „weich“-gezeichnet werden können, siehe (c) der Abbildung.

[DirectX 00, „Texture Size“] empfiehlt die Auflösung einer Textur immer so klein wie möglich zu halten und wann immer es geht quadratische Grafiken zu benutzen. Texturen mit einer Auflösung von 256x256 seinen die „schnellsten“.

2.2.2 MipMap

Wie im Geometriebereich unter LOD beschrieben, müssen weit entfernte Objekte nicht so detailliert sein – das trifft auch auf die Texturen zu. Befindet sich beispielsweise ein Objekt in so großer Entfernung zum Spieler, dass die Größe auf dem Ausgabegerät nur noch 30x40 Pixel beträgt, wäre eine Textur der Dimension 256x256 völlig unnötig. Darüber hinaus kann die Darstellung einer hochauflösenden Grafik in weiter Ferne zu Bildfehlern führen, wie dem Moiré-Effekt.

Das sogenannte MipMap-Verfahren (oder auch MipMapping), das mittlerweile von allen modernen Grafikkarten unterstützt wird, kann derartige Fehler verhindern, wie aus der folgenden Abbildung hervorgeht.

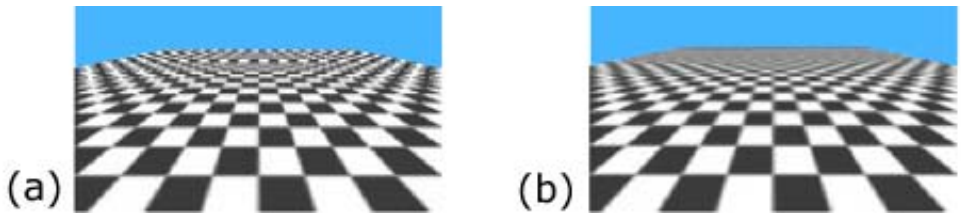

Abb. III-7 MipMapping verbessert die Qualität des Bildes – Teil (b) der Grafik

Eine MipMap ist nach [Ertl 01] eine Textur, in der neben der Originalgröße, vorgefilterte Versionen in gröberen Auflösungen gespeichert sind. MIP leitet hierbei vom lateinischen „Multum In Parvo“ ab („Vieles in Kleinem“). Da die gröberen Auflösungen ein Viertel der Größe der vorhergehenden Stufe einnehmen, benötigt eine MipMap-Textur, verglichen mit der Originaltextur, ein Drittel mehr Speicherplatz.

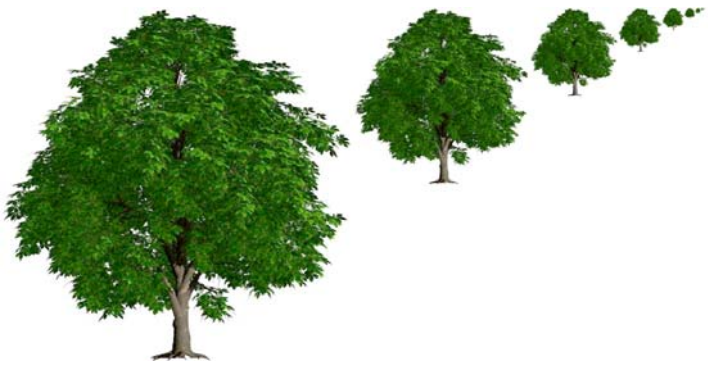

Abb. III-8 Die verschiedenen Stufen einer MipMap Textur

Die Erzeugung der gröberen Auflösungen erfordert nach [DirectX 00, „DXTex Tool“], dass Höhe und Breite der Originaltextur Potenzen von 2 sind. Darauf muss beim Design geachtet werden, da sich der Initialisierungsprozess, bei dem die MipMaps von der Engine generiert werden, durch eine entsprechende Skalierung der Grafiken verzögert. Die Erstellung und Speicherung der MipMap Stufen kann aber auch im Vorhinein vom Leveldesigner übernommen werden, siehe Kapitel III, Abschnitt 1.5, wodurch der Startvorgang des Spiels verkürzt wird.

[Dietrich 99] zufolge kann das MipMapping Verfahren, neben der Behebung von Bildfehlern, die Performance steigern, da zum Rendern der entfernteren Objekte wesentlich kleinere Texturen verwendet werden.

2.2.3 Speicherplatz

Texturen benötigen mit steigender Qualität (Farbtiefe, Dimension) mehr Speicherplatz. Sowohl im Texturspeicher der Plattform (beim PC auf der Grafikkarte) als auch auf dem Medium, auf dem das Computerspiel gespeichert ist. Platz bieten die CDs oder DVDs, auf denen heutzutage fast ausschließlich alle Spiele ihre Inhalte ablegen, zwar reichlich, jedoch müssen die Daten zunächst in den Texturspeicher transportiert und nach Möglichkeit dort gehalten werden. Meist fasst dieser 32 oder 64 MB.

Herkömmliche Bildformate wie TIFF oder JPEG, die Verfahren zur Komprimierung bieten, können den benötigten Speicherbedarf einer Textur auf ein Bruchteil minimieren und damit den Datenaufwand sowohl auf dem Medium als auch beim Transport möglichst niedrig halten. Im Texturspeicher liegen die Bilder jedoch in reinen RGB-Werten vor. Das hängt damit zusammen, dass der Grafikchip möglichst schnell auf die Farbwerte der Texel zugreifen muss – das würde eine Dekomprimierung der oben aufgeführten Bildformate zur Laufzeit erfordern, die nicht möglich ist. Auf diesem Wege kann die Zahl an Texturen im Grafik-Speicher daher nicht erhöht werden.

Abhilfe schaffen da nur hardwareunterstützte Textur-Kompressionen. Sie ermöglichen dem Grafikchip auf die RGB-Werte eines Texels zugreifen zu können obwohl die Texturen komprimiert im Speicher gehalten werden. Dieser wird entsprechend weniger „belastet“ und bietet Platz für mehr und größer dimensionierte Grafiken. Die erste Technik dieser Art wurde von der Firma S3 entwickelt und nannte sich S3TC. Sie blieb längere Zeit ein Privileg der firmeneigenen Grafikkarten und führte zu weiteren Entwicklungen anderer Hardware-Hersteller (z.B. FXT1 der Firma 3DFX). Microsoft lizenzierte diese Technologie von S3, um sie als Standard Kompression für deren DirectX zu implementieren und ebnete damit den Weg für eine freie Weiterentwicklung und Nutzung [Cross 99]. Mittlerweile ist DXTn, so die Bezeichnung von Microsoft, ein Feature das von allen gängigen Grafikkarten unterstützt und von entsprechend vielen Computerspielen genutzt wird.

DXTn ist ein mit Verlusten behaftetes Komprimierungsverfahren, dass sich nicht nur für „normale“ Texturen, sondern auch für Bilder mit Alpha-Kanal eignet. Im Abhängigkeit der Qualität dieses Kanals (Bit-Tiefe), ergeben sich nach [Doug 01] unterschiedliche Kompressionsraten. DXT1 ermöglicht ein Verhältnis von 8:1. Für die Informationen des Alpha-Kanals steht bei dieser Methode lediglich 1 Bit zur Verfügung. Transparenzen mit mehr Abstufungen lassen sich dagegen mit Hilfe von DXT3 komprimieren, wobei derartige Texturen nur noch 4 mal so klein wie ihr Original sind.

2.2.4 Tiles

Hinter Tiling, verbirgt sich die Methode, eine Textur wiederholt auf ein Polygon zu „legen“ – zu kacheln (engl. Tile – Kachel). Die Texturkoordinaten liegen in diesem Fall nicht mehr im Bereich zwischen 0 und 1 – wie in (b) der folgenden Abbildung.

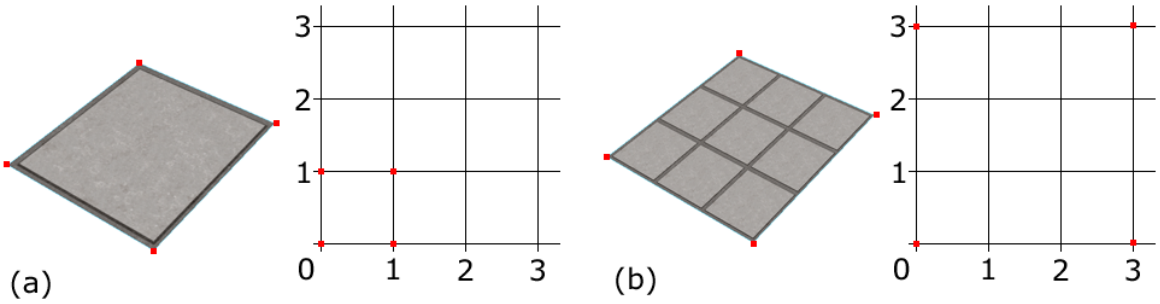

Abb. III-9 Texturkoordinaten außerhalb von 0 und 1lassen Texturen mehrfach erscheinen

Der Vorteil dieser Technik ist, dass mit Hilfe sehr kleiner Bilder (Auflösung und notwendiger Speicherplatz) große polygonale Flächen mit einer Struktur versehen werden können, beispielsweise eine Steinplatte aus der ein Plattenweg entstehen kann, wie im oberen Beispiel gezeigt.

Ein Problem das bei kachelbaren Texturen auftritt, ist dass die Stellen an denen das Bild wiederholt dargestellt wird, dem Betrachter sofort auffallen, wenn diese nicht nahtlos ineinander übergehen - in (a) der folgenden Abbildung dargestellt. Seamless Tiles sind dagegen so angelegt, dass die Ränder, an denen eine Wiederholung stattfinden soll, nahtlos (engl. seamless) ineinander übergehen - wie in (b) der folgenden Grafik.

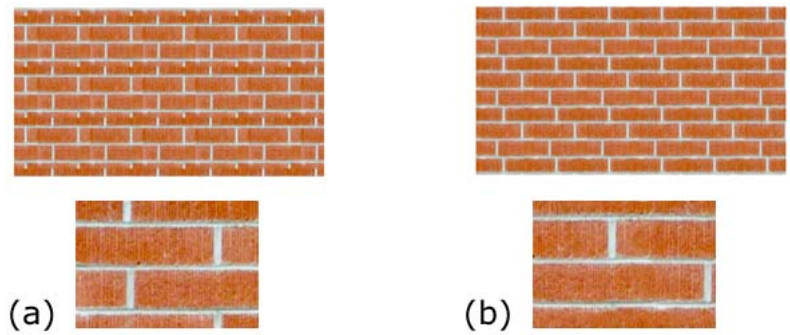

Abb. III-10 Texturen als Kacheln eingesetzt

Durch die Wiederholung der Texturen kann die in der oberen Grafik abgebildete Wand aber sehr schnell eintönig werden.

Damit Details, wie beispielsweise ein Fenster, dargestellt werden können, stehen dem Designer mehrere Möglichkeiten zur Verfügung: Die Aufnahme in das Tile würde nicht nur dessen Dimensionsvergrößerung bedeuten, sondern kann auch zu einem ungewollten Ergebnis führen, da sich das Fenster sehr oft wiederholt. Zur weiteren Nutzung des Vorteils einer „kleinen“ Wandkachel und zur Verhinderung der Wiederholung der Fenstertextur, kann der Designer auf Multi-Materials zurückzugreifen, wie auf der nächsten Seite näher erläutert wird.

Durch den Einsatz von Multi-Materials muss in diesem Falle die Wandfläche so unterteilt werden, dass an der geplanten Fenster-Position eine Fläche entsteht, auf die die Detailgrafik gemappt werden kann. Dies hat eine Unterteilung der Wandfläche in mehr Polygondreiecke zur Folge, wie die nachfolgende Grafik zeigt. Sind in (a) nur 2 Dreiecke notwendig - allerdings ohne Fenster, besteht die Mauer in (b) aus 10 Dreiecken, davon 2 für die Fensterfläche.

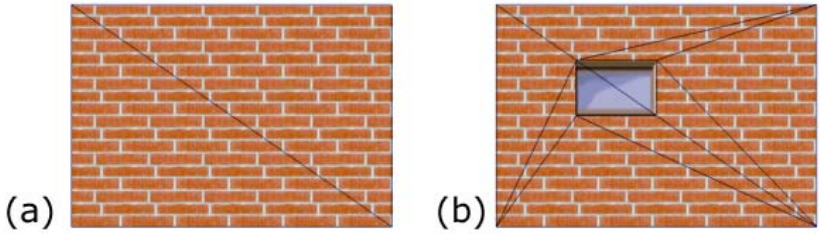

Abb. III-11 Für die Verwendung von Multi-Materials muss die Wandfläche unterteilt werden

Eine wesentlich polygonsparendere Methode ist die Platzierung einer Fläche, für die Fenstergrafik, kurz vor der Wand. Auf diese Weise kommen nur 4 Dreiecke (für Wand und Fenster jeweils 2) zum Einsatz. In der Praxis ergibt sich aber auch hierbei ein Problem – damit der Spieler nicht erkennt, dass es sich um zwei Flächen handelt muss sich das Fenster möglichst nahe der Wand befinden. In größer werdenden Entfernungen können aber Darstellungsprobleme auftreten. Die Bildfehler werden vom z-Buffer verursacht, beziehungsweise der eingeschränkten Genauigkeit (Bit-Tiefe) mit der dieser Speicher arbeitet. Sollen wie in diesem Fall zwei Polygone fast millimetergenau hintereinander positioniert werden, kann in sehr großer Entfernung der Mindestabstand unterschritten werden der vom Buffer noch korrekt berechnet wird. Durch sich daraus ergebene Rundungsfehler kann die korrekte Tiefenreihenfolge der Pixel durch falsche z-Werte beeinflusst werden und zu mangelhaften Ergebnissen in der Darstellung führen, wie im nachfolgenden Bild dargestellt.

Abb. III-12 Screenshots eines aktuellen 3D-Spiels der Firma id software[11]

Teil (a) der Abb. III-12 zeigt vier Fenster, die auf die zuvor beschriebene Weise erstellt wurden. Mit Hilfe der integrierten Fernglasfunktion des Spiels ist es dem Benutzer aus einer sehr entfernten Position möglich, das Haus zu betrachten - mit entsprechendem Ergebnis (b). Einige Engines bieten dem Designer für derartige Situationen den sogenannten z-Bias als Lösung an. Dieses Attribut sorgt dafür, dass die z-Werte des Objekts, in diesem Falle die Fenstergrafik, automatisch um einen intern festgesetzten Wert erhöht werden, so dass diese Pixel „bevorzugt“ gerendert werden, siehe 1.6.3 „Render – Phase“ im Kapitel II „Grundlagen“.

Mit Hilfe von Multi-Textures ist es möglich das Fenster ohne einen Mehraufwand an Polygonen darzustellen. Die Positionierung der Detailgrafik geschieht über eine eigene UV-Map, die auf der Wandgeometrie gespeichert wird. In diesem konkreten Beispiel müsste zusätzlich die Kachelfähigkeit der Fenstertextur deaktiviert werden, da sie sonst die gesamte Wandfläche bedeckt.

[11] http://www.idsoftware.com

2.2.5 Transparenzen

Im Bereich der Texturen, spielen Bilder mit einem Alpha-Kanal eine wichtige Rolle. Sie bieten dem Designer zwei Einsatzmöglichkeiten:

- optische Verbesserung der Spielumgebung und
- Ersparnis von Geometriedaten

Ersteres ergibt sich dadurch, dass mit Hilfe des Alpha-Kanals ganz oder teilweise durchsichtige Objekte kreiert werden können, die zur Erhöhung des Realismus der Szene beitragen, wie zum Beispiel die Fenster eines Hauses. Bei der Verwendung einer solchen Textur muss jedoch das Back-Face Culling berücksichtigt werden. Denn durch die Transparenz der Flächen, ist es dem Spieler möglich ins Innere, in diesem Fall des Hauses, zu schauen. Da die Rückwand vom Culling-Verfahren entfernt wurde, ist die Illusion des Hauses sehr schnell zerstört. In solchen Fällen müssen vom Designer zusätzlich Flächen geschaffen werden, die sich hinter den transparenten Objekten befinden und in Richtung Spieler zeigen.

Eine Einsparung von Levelgeometrien ergibt sich dadurch, dass mit Hilfe des Alpha-Kanals komplizierte und verzweigte Strukturen, wie Äste und Blätter eines Baumes, als Textur ins Level gebracht werden können, anstatt die Objekte aus unzähligen Polygonen herauszumodellieren.

2.3 Beleuchtung

Wird die Szene zur Laufzeit beleuchtet, sind es zunächst die Lichtquellen-Typen, die einen Ansatzpunkt für eine Optimierung bieten. Laut [Rogers 00] kann eine Szene zwar beliebig viele Lichtquellen beinhalten, aber nur 8 davon dürfen maximal zur Beleuchtung eines Objektes der Szene eingesetzt werden. Der Grund für diese Beschränkung ist, dass die Hardware nicht mehr als 8 Lichter gleichzeitig verarbeiten kann. Gerichtetes Licht ist sehr schnell zu berechnen, während Spotlicht die rechenaufwändigste Form der Lichtquellen darstellt, so [Rogers 00] weiter.

[DirectX 00, „Lighting Tips“] empfiehlt bei der Beleuchtung:

- so wenig wie möglich Lichtquellen verwenden
- für die Erhöhung der Lichtintensität der gesamten Szene nach Möglichkeit das Ambient Light zu nutzen, als eine neue Lichtquelle einzufügen
- Directional Lights sind „günstiger“ als Spot oder Point Lights
- Spotlights können einfacher zu berechnen sein als Point Lights. Ob sie tatsächlich schneller sind, hängt vom Einflussbereich des Spotlichts – der Kegelgröße, ab.
- Specular Highlights können die „Kosten“ einer Lichtquelle fast verdoppeln, sind daher nur dann zu verwenden wenn sie wirklich benötigt werden.

2.3.1 Prelighting

Echtzeitbeleuchtung hat den Vorteil, dass die 3D-Szene immer korrekt, entsprechend der Position der Lichtquellen, ausgeleuchtet wird. Allerdings müssen die dafür nötigen Berechnungen für jedes Frame stets neu durchgeführt werden, was mit einem entsprechendem Rechenaufwand verbunden ist. Daher ist es sehr oft üblich die Beleuchtung weitestgehend ins Vorfeld – den Designprozess zu verlegen, indem die Objekte um Beleuchtungsinformationen ergänzt werden – auch Prelighting genannt.

Was beim Prelighting ausgenutzt wird, ist dass die meisten Lichtquellen der Szene eine feste Position haben (Lage der Sonne, Laternen, Reklameschilder, etc.) und viele Objekte der Umgebung statisch sind (Häuser, Straßen, ...). Für die Übertragung der Helligkeitsinformationen auf ein Objekt, stehen dem Level- und Characterdesigner 2 Möglichkeiten zur Verfügung:

- Light Maps und
- Vertex Colors

Light Maps sind nichts anderes als Bilder, die Beleuchtungsinformationen enthalten. Sämtliche auf der Q3A-Engine basierende Spiele setzen diese Technik ein und erzielen damit sehr überzeugende Ergebnisse, siehe Abb. III-1. Light Maps sind für gewöhnlich relativ klein dimensionierte Texturen (16x16 oder 32x32) die durch Filtermethoden „weich“ gerechnet und über Multi-Texturing auf wesentlich größer dimensionierte Objektflächen, wie beispielsweise Wände, gelegt werden. Das hat den Vorteil, für den Großteil der Umgebung Tiles verwenden zu können und mit Hilfe der Light Maps nicht nur die Beleuchtung sondern auch mehr „Abwechslung“ ins Level zu bekommen. Die Beleuchtungsinformation kann aber auch vom Designer direkt in die Textur eingearbeitet sein – eine Möglichkeit die sich besonders bei den nicht kachelbaren Texturen der Character eignet.

Bei der zweiten Variante, werden die Farbwerte, die sich für ein Objekt aus den Eigenschaften einer Lichtquelle ergeben, auf dessen Geometriendaten, dass heißt auf dessen Vertices übertragen (Vertex Colors). Der Designer muss anschließend dafür sorgen, dass die Game-Engine beim Rendern des Objekts die gespeicherten Vertex Colors benutzt, anstatt die Farben in Echtzeit zu berechnen. Durch die Übertragung der vorberechneten Werte auf den Emission-Kanal des Materials entsteht dann das beleuchtete Objekt.

Wie zu Beginn der Seite angesprochen, eignen sich diese Methoden besonders gut für statische Objekte. Im Falle der Character erschwert sich diese Art der Beleuchtung, da sich durch deren Animation schnell neue Beleuchtungssituationen ergeben können. Ungeachtet dessen, lassen sich auch bei diesen Bereiche finden, die aufgrund der natürlichen Beleuchtung eher hell oder dunkel sind – das Dach eines Autos verglichen mit dessen Seite, die Achselpartie bei Personen, Falten in deren Kleidung und so weiter.

2.3.2 Schlagschatten

Im Bereich der Echtzeitschatten ergibt sich aus der Betrachtung der Berechnungsweise, dass die projizierten Schatten die schnellste Variante darstellen. Sie bieten allerdings auch die geringste Qualität und stark eingeschränkte Einsatzmöglichkeiten. Da sie auf eine Ebene projiziert werden, eignen sie sich auch nur für einen derartigen Untergrund. Sobald die Oberfläche uneben wird, zum Beispiel durch einen Hügel, ragt der Schatten durch diese Geometrie hindurch, anstatt sie zu bedecken.

In diesem Fall eignen sich Volumen Schatten und Shadow Maps. Sie bieten eine größere Flexibilität, erfordern aber auch wesentlich mehr Rechenaufwand. Einen klaren Vorteil bezüglich der Qualität, bieten die Shadow Maps, da nur sie einen weichen Schattenrand ermöglichen. Projizierte und volumetrische Schatten erzeugen harte Ränder, da die Schatten aus Geometrien errechnet werden.

Zur Optimierung der Volumen Schatten kann der Designer beitragen, indem er bei der Erstellung der 3D-Szene die Modellierung der Extrusionskörper übernimmt, vorausgesetzt die Game-Engine ist darauf ausgelegt und bietet eine Kennzeichnung derartiger Geometrien.

Wie bei der Beleuchtung ergibt sich auch in diesem Bereich die Möglichkeit für den Designer, den gewünschten Effekt im Vorfeld zu erzeugen. Der einfachste Weg geht über die zuvor, unter „Prelighting“, beschriebene Methode der Light Maps. Vertex Colors können ebenfalls eingesetzt werden, erfordern jedoch vom Designer die Unterteilung der Geometrie, auf der die Schatten dargestellt werden sollen. In Anhängigkeit von der Genauigkeit der Silhouette, entstehen mehr oder weniger zusätzliche Polygone. Unabhängig davon bildet auch hier der Übergang zwischen Licht und Schatten eine scharfe Kante aufgrund der Verwendung von Geometrie.

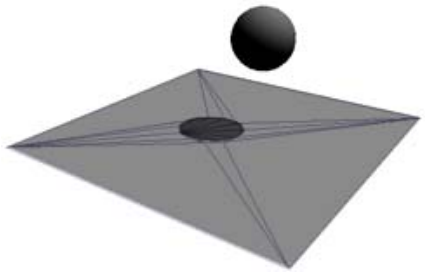

Abb. III-13 Konstruktion eines Schlagschattens mit Hilfe von Vertex Colors

Die in der oberen Abbildung dargestellte Szene enthält einen von der Kugel geworfenen Schatten, der mit Hilfe von Vertex Colors realisiert wurde. Die in diesem Fall relativ genaue Silhouette der Kugel, erforderte eine Unterteilung der Bodenfläche in 26 Polygondreiecke, davon 11 für die Kreisfläche des Schattens. Im Normalfall wären 2 Dreiecke zur Beschreibung des Bodens ausreichend gewesen.

Alle im Vorfeld erstellten Schlagschatten haben den großen Nachteil, dass sie nur in Verbindung mit statischen Objekten funktionieren. Außerdem haben sie keinen Einfluss auf die Geometrie, die sich während der Laufzeit in den Schattenbereich bewegt. Für dynamische Objekte müssen die Schatten in Echtzeit berechnet werden, wobei die projizierten ebenfalls keinen Einfluss auf andere Objekte haben, die sich im Schattenbereich befinden.

2.4 Animation

Techniken wie Pfad- und Skelettanimation mit IK, erleichtern zwar den Arbeitsaufwand des Designers und führen zu geschmeidigeren und somit optisch besseren Ergebnissen, in Echtzeit berechnet bedeuten sie dagegen einen erhöhten Rechenaufwand, da die Spiel-Engine beispielsweise bei einem Skelett mit IK, die Position der Vertices über die Keyframes der IK-Kette ermitteln muss. Wie des öfteren schon beschrieben sollten diese Animationen vom Designer, sofern dies möglich ist, innerhalb der Entwicklungsphase in Keyframes umgewandelt werden, so dass die Engine zur Laufzeit nur noch auf die Ergebniswerte zugreifen muss.

Es ist allerdings nicht immer möglich beziehungsweise sinnvoll Pfade oder Skelette durch Keyframes zu ersetzten. Sollen zum Beispiel viele Fahrzeuge einem vorgegebenen Straßensystem folgen, kann anstatt mit Hilfe eines einzigen Pfades, die Animation aller Fahrzeuge berechnet werden. Auf die gleiche Art und Weise lassen sich auch Skelette mehrfach verwenden.

Zur Minimierung des Rechenaufwandes den Skelette mit IK verursachen, werden in nahezu allen Spielen nur 2-Knochen IK-Ketten verwendet. Dass heißt die Echtzeit-Berechnung der Inversen Kinematik erfolgt nicht über beliebig viele Knochen sondern nur über das sinnvollste Minimum - zwei Knochen.

Das in Zusammenhang mit organischen Objekten benötigte Skinning bei Skelettanimationen, lässt sich ebenfalls, in Bezug auf den Rechenaufwand der zur Laufzeit notwendig ist, optimieren. Das rechenintensive Smooth Binding kann dadurch beschleunigt werden, dass die Zahl der Gelenke, die einen Vertex beeinflussen, verringert wird. Es kann auch das einfache Rigid Binding eingesetzt werden, besonders bei Personen, die in ihren Bewegungen keine Posen enthalten, die zu erkennbaren Knicken an den Gelenken führen (einfache Gehbewegungen der Passanten auf der Straße) Mit Hilfe einer Gliederpuppe kann die Zeit für das Skinning auch ganz eingespart werden. Allerdings entstehen auf diese Weise beim Menschen sehr viele kleine Meshes (Hände, Unter- und Oberarme, Kopf, usw.) – nach den Erkenntnissen des Modellierungsbereichs nicht optimal und bei der Bewegung wird die Zusammensetzung aus Einzelteilen erkennbar.

Aber auch die Verwendung von Keyframes lässt sich optimieren. Dazu wird die ursprünglich fest definierte Keyframe-Animation in kleine sich wiederholende Animationssequenzen unterteil – sogenannte **Clips**. Bei der Bewegung eines Menschen wäre dies zum Beispiel eine kompletter Geh-Zyklus (siehe Abbildung). Soll die Person dann eine Straße entlanggehen, kann der Weg mittels einer Pfadanimation festgelegt werden und für die Geh-Bewegung käme der sich ständig wiederholende Clip zum Einsatz. Die Clip-Technik wird auch als nichtlineare Animation bezeichnet. Durch die Möglichkeit Sequenzen beliebig zu kombinieren und miteinander vermischen (überblenden) zu können, eignen sie sich besonders gut zur Erstellung umfangreicher Animationsabläufe aus relativ wenigen Bewegungsdaten.

Eine weitere Möglichkeit Rechenaufwand einzusparen ist das **Ausblenden** der Animationen sichtbarer Objekte in weiter Entfernung. Dass heißt nicht die Bewegung von A nach B sondern, die individuellen Bewegungen des Objektes, wie die Armbewegungen eines Menschen beim Gehen.

2.4.1 Dynamics

Sämtliche Effekte, die auf physikalischen Berechungen beruhen, können nicht nur den Spieler sehr beeindrucken, sondern auch viel Rechenzeit kosten, da sie zur Laufzeit stattfinden.

Wie in den anderen Bereichen, ist es auch hier möglich Animationen, die auf diese Weise entstehen würden, ins Vorfeld zu verlegen. Dazu muss dem Designer allerdings eine Anwendung zur Verfügung stehen, die es ihm ermöglicht derartige physikalische Simulationen durchzuführen und die Ergebnisse abzuspeichern - im Normalfall Keyframes. Mit Maya ist dies zum Teil möglich. Auf diese Weise wird nicht nur kostbare Systemzeit frei, sondern gelangen auch naturgetreue Bewegungsabläufe in Spiele, in denen keine Physik-Engine implementiert ist, wie der Prana-Engine.

Selbst für Partikelsysteme, wäre dieses Prinzip denkbar. Aufgrund der hohen Datenmenge, die die Partikel verursachen, wird dies jedoch nicht praktiziert. Neben einem sorgsamen Einsatz können diese Systeme aber auch über die Anzahl der ausgestoßenen Partikel und deren „Lebensdauer“ an die Rechenleistung der Spiel-Engine angepasst werden.

Eine sinnvolle Anwendung im Bereich Partikelsystem stellen Sprites dar, Texturen die stets zum Betrachter zeigen, ähnlich den Billboards, siehe 2.1.3 dieses Kapitels. Da die zum Einsatz kommenden Partikel meist relativ klein sind (Funken, Wassertropfen, etc.), bleiben diese Täuschungen vom Spieler eher unbemerkt.

2.5 Frame-Rate & V-Sync

Allgemein gilt: je höher die Bildwiederholfrequenz, desto besser das Spielerlebnis beziehungsweise das Spielgefühl. Umgekehrt sollte, wie zu Beginn des Kapitels II „Grundlagen“ beschrieben, die Bildrate nie unter 15 fallen. Allerdings begrenzt die Bildwiederholfrequenz des Ausgabegerätes die Frame-Rate wenn die Anwendung im sogenannten V-Sync-Modus läuft.

V-Sync bedeutet, dass die Grafikkarte solange mit dem Senden eines neuen Bildes aus ihrem Frame-Buffer wartet, bis der Fernseher oder Monitor das aktuelle Bild komplett dargestellt hat und damit ein vertikales Synchronisationssignal (daher V-Sync) zurücksendet. Dadurch wird gewährleistet, dass der zeilenweise Bildschirmaufbau vom Spieler nicht wahrgenommen werden kann beziehungsweise nicht unterschiedliche Bildfragmente zur gleichen Zeit dargestellt werden. Diese Maßnahme zur Verbesserung der Bildqualität erfordert allerdings auch eine konstante und hohe Frame-Rate der Engine. Dazu folgende Überlegung: Einem Ausgabegerät, dessen Bildwiederholfrequenz bei 3 Hz liegt, würde eine Spiel-Engine nur 2 Bilder pro Sekunde liefern. Die Folge – das Ausgabegerät muss ein Bild mehrfach anzeigen. Bei diesem Beispiel würde der Versatz dem Spieler wohl extrem auffallen - das Prinzip greift jedoch auch bei Fernsehgeräten mit 50 oder 60Hz. Ist die Engine nicht in der Lage eine entsprechend hohe Framezahl zu rendern, kann sie, zur Erlangung eines „flüssigeren“ Ergebnisses, auch auf ein Vielfaches der Frequenz des Ausgabegerätes reduziert werden (zum Beispiel bei einem 60Hz Bildschirm – 30 FPS). Eine derartige Reduzierung ist zwar möglich, widerstrebt aber dem Ziel eine möglichst hohe Framezahl zu erreichen. Da „Skate Attack“ nicht nur für den nationalen Spiel-Markt entwickelt wird, ist wegen der NTSC-Norm in den USA eine konstante Frame-Rate von 60 FPS angestrebt.

IV Laufzeitoptimierung am Beispiel von „Skate Attack“

Mit dem Wissen über allgemeine Funktionsprinzipien im 3D-Echzeitbereich und den Kenntnissen über Rahmenbedingungen und Optimierungstechniken ergeben sich für den Bereich des Level- und Characterdesigns verschiedene Herangehensweisen zur Realisierung der Spielumgebung beziehungsweise zur Erzeugung von Effekten. Dabei ist es natürlich von Interesse wo sich im Falle von „Skate Attack“, mit Rücksicht auf die Bildqualität, Möglichkeiten zur Verbesserung der Laufzeit ergeben.

1 Messprogramm

Im Zusammenhang mit Performanz-Tests fällt immer wieder der englische Begriff „Benchmark“. Wörtlich übersetzt bedeutet er „Bezugsgröße“ oder „Vermessung“. Ein Benchmark-Test ist demzufolge ein Leistungstest. Programme die bestimmte Bereiche (wie Bildqualität, Übertragungsgeschwindigkeiten, 3D-Funktionen, etc.) oder die Gesamtleistung einer Plattform messen können gibt es viele. Gerade im nahezu unüberschaubaren Grafikkarten-Sektor herrscht ein großes Interesse an Vergleichsmöglichkeiten – der Grundgedanke, der hinter solchen Tests steht. So lassen sich zum Beispiel 2 verschiedene Plattformen bezüglich ihrer Render-Geschwindigkeit gegenüberstellen, indem sie unter gleichen Voraussetzungen dieselbe 3D-Szene rendern. Nach diesem Prinzip lassen sich auch die für die Entwicklung von „SkateAttack“ zur Verfügung stehenden Techniken des Level- und Character-Designs testen.

Die Entwicklungsversion der Prana-Engine gibt zwar die Möglichkeit sich zur Laufzeit über Werte wie Anzahl der zur Grafikkarte geschickten Objekte und Polygone, Auslastung des Texturspeichers, Renderzeiten der einzelnen Frames, etc. einen Überblick zu verschaffen, eine Protokollierung der Daten ist jedoch nicht möglich. Allerdings verfügt die Engine über eine Python-Schnittstelle sowie ein Initialisierung-Script, das beim Start der Engine ausgewertet wird. In dieser Datei können der Engine Parameter (beispielsweise zur Ausgabeauflösung) und Funktionen übergeben werden, so dass sich auch die ausgegebenen Werte aufzeichnen lassen. Dies führte zur Idee ein Prana-Bench-Mark Tool, kurz PBM, zu entwickeln. Dieses Programm sollte nicht nur eine komfortable Modifikation beziehungsweise Generierung der Initialisierungsdatei ermöglichen sondern auch eine geeignete Form der Auswertung bieten sowie die Untersuchungen erleichtern. Dass heißt zum Beispiel, dass unterschiedliche Testszenen geladen, Einstellungen für die Engine vorgenommen, die Messergebnisse dargestellt, separat gespeichert und zu Vergleichszwecken wieder geladen werden sollen. Für die Ausgabe der Daten bieten sich sowohl Diagramme als auch Zahlenwerte an (Menge der Polygonmeshes, durchschnittliche Renderzeit, ...). Eine übersichtliche Gestaltung und vereinfachte Bedienung kann mit einer grafischen Benutzeroberfläche erreicht werden.

Die Gelegenheit sich Kenntnisse über Python anzueignen und mit Hilfe dieser Sprache PBM zu realisieren, ergab sich aus folgenden Gründen:

- eine Kommunikation mit der Prana-Engine ist nur über Pythonscripts möglich und Python selbst eignet sich sehr gut zur dynamischen Generierung von Quellcode (Init-Script)
- an PBM werden keine besonderen Performanzanforderungen gestellt, so dass die Vorteile einer Scriptsprache genutzt werden können:
  - relativ einfache Handhabung
  - keine Programmierumgebung oder Compiler notwendig
- mit wxPython, einem Zusatzmodul für Python, lässt sich die grafische Benutzeroberfläche implementieren

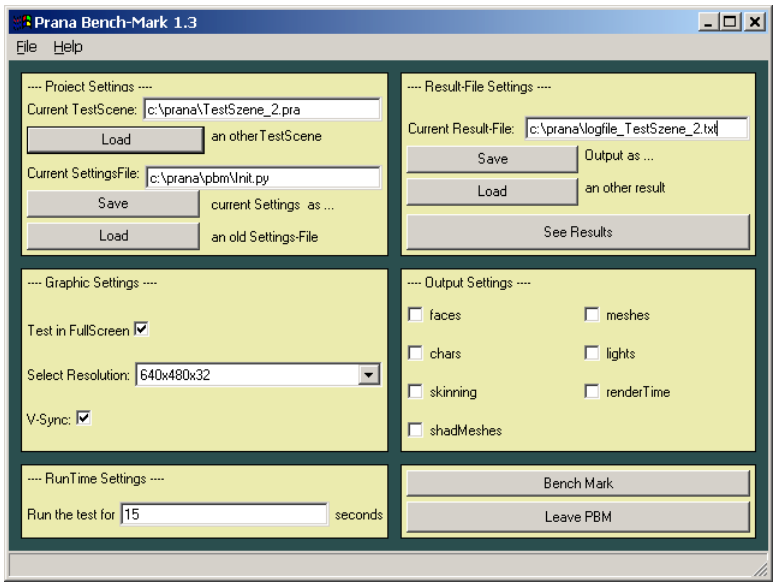

Abb. IV-1 Die Benutzeroberfläche von PBM

Zur Erklärung der Abb. IV-1: Die Oberfläche von PBM teilt sich in 6 Themen-Bereiche, die jeweils gelb gefärbten Blöcke, in denen verschiedene Einstellungsmöglichkeiten vorzunehmen sind.

Unter **Project Settings**, das Feld links oben, lässt sich die zu prüfende Szene laden und die aktuellen Einstellungen der anderen Felder in einem Profil abspeichern beziehungsweise wiederherstellen.

Darunter befinden sich die **Graphic Settings**, die festlegen in welcher Ausgabeauflösung die Szene dargestellt werden soll. Außerdem gibt es die Möglichkeit zwischen Fenstermodus und Vollbild zu wählen sowie den V-Sync-Modus ein- oder auszustellen.

Im untersten Teil der linken Hälfte – den **RunTime Settings** – kann die Messdauer eingestellt werden.

In den **Result File Settings** kann festgelegt werden, wohin die protokollierten Werte gespeichert werden sollen. Das Laden einer vorhandenen Ergebnisdatei ist ebenfalls möglich. Der Button „See Results“ stellt die Protokollwerte grafische in Form eines Liniendiagramms dar. Damit ein einfacherer Vergleich zwischen mehreren Testergebnissen möglich ist, wird für jede Ergebnisdatei ein neues Fenster geöffnet. Wie ein solches Ergebnis aussieht, zeigt die Abb. IV-2.

Über die **Output Settings**, im mittleren Bereich der rechten Hälfte, lassen sich Profiles auswählen, beispielsweise die Polygonanzahl, Zeiten für Animationsberechnungen oder Verarbeitung der Display List, die während der Testphase protokolliert werden sollen.

Die letzten beiden Buttons sind zum Starten der Testszene beziehungsweise zum Verlassen von PBM.

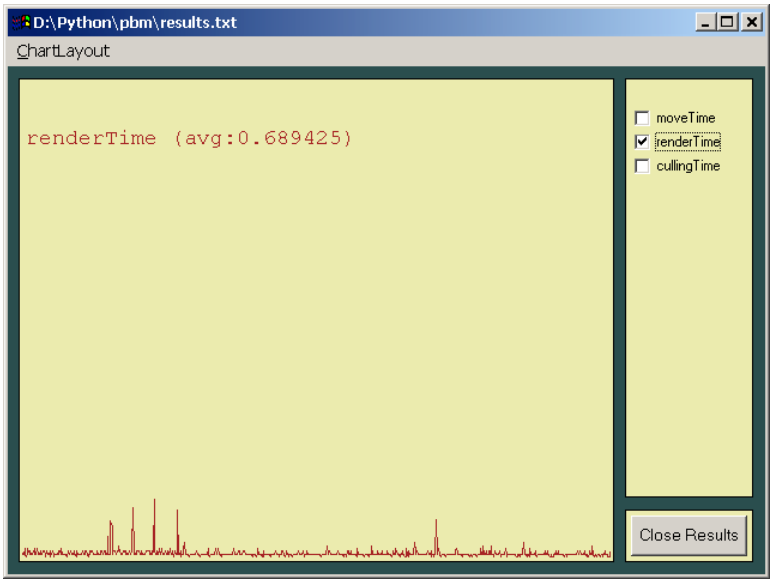

Abb. IV-2 Die über ein Liniendiagramm dargestellten Ergebniswerte

Die Abb. IV-2 stellt die grafische Ausgabe der Protokoll-Daten dar. Werden während des Tests mehrere Profiles aufgezeichnet, können die verschiedenen Datensätze über Checkboxen ausgewählt und damit ein- beziehungsweise ausgeblendet werden.

2 Versuche

Für die Untersuchungen sind zwei verschiedene Arten von Testszenen möglich. Einerseits spezielle Versuchsanordnungen mit einem realen Anwendungszweck und andererseits das aktuelle Demolevel, dass im finalen Spiel verwendet werden soll. Beides hat einen begründeten Vorteil. Mit Hilfe der speziellen Testszenen lassen sich einzelne Sachverhalte genauestens überprüfen. Die Ergebnisse können dann den weiteren Prozess des Level-& Character-Designs beeinflussen. Ebenso aufschlussreich sind Testergebnisse, die unter genau den Bedingungen stattfinden, wie sie sich im Spiel ereignen. Da zur Zeit noch keine Kamerafahrten möglich sind, werden in den Szenen Kameras platziert, die unterschiedliche Blickwinkel im Laufe einer Kamerafahrt nachstellen beziehungsweise für spezielle Untersuchungen geeignet sind.

Folgende wichtige Punkte, gilt es in jedem Fall zu beachten: Um zwei Techniken oder Sachverhalte miteinander vergleichen zu können, müssen die übrigen Bedingungen gleich sein, da sonst falsche Messwerte entstehen können. Dass heißt zum Beispiel, gleiche Ausgabeauflösung, dieselben Kameraeinstellungen und so weiter. Da auch das System die Messergebnisse beeinflussen kann, muss darauf geachtet werden, dass keine weiteren Anwendungen offen oder unnötige Systemdienste aktiv sind. Damit vom System verursachte Messengenauigkeiten weitestgehend minimiert werden können, bieten sich längere Messzeiten mit anschließender Durchschnittsbildung der Messdaten an. Mehrmalige Test-Durchläufe senken ebenfalls die Beeinflussung des Systems auf die Ergebnisse. Durchgeführt werden die Tests auf einem PC mit 600 MHz und einer GeForce 2 Grafikkarte. Aufgrund der Tatsache das „Skate Attack“ ein Konsolenspiel wird, werden sämtliche Tests in einer entsprechenden Ausgabeauflösung von 800x600 Bildpunkten durchgeführt. Die aufgenommenen Messwerte werden aus Gründen der Übersichtlichkeit in tabellarischer Form im Anhang aufgeführt.

3 Anwendungsbeispiele

Für die Erstellung des Spielinhalts war sowohl eine intensive Beschäftigung mit dem Themen Low-Poly-Modellierung, Textur-Erstellung als auch Texture-Mapping notwendig.

Im folgenden Teil des Kapitels soll exemplarisch dargestellt werden, wie sich in „Skate Attack“ Optimierungstechniken anwenden lassen beziehungsweise welche Schwierigkeiten sich dabei ergeben. Außerdem werden mit Hilfe von PBM Messungen an Testszenen durchgeführt, um unterschiedliche Herangehensweisen auf ihre Vor- oder Nachteile zu überprüfen.

3.1 Fahrzeug-Stoßstange

Damit beispielsweise Autos den optischen Ästhetikansprüchen von „Skate Attack“ gerecht werden, dürfen keine texturierten Quader zum Einsatz kommen. Gefordert ist eine wagentypische Gestalt, die allerdings aus so wenig wie möglich Polygonen bestehen soll. Details, die für eine ansprechende Optik nötig sind, wie Kühlergrill, Scheinwerfer, Türgriffe und so weiter benötigen für die Modellierung zu viel Geometriedaten und werden daher in der Textur untergebracht, wie im späteren Teil dieses Kapitels naher erläutert wird.

Eine Herausforderung, besteht in der Fertigung der Stoßstangen, Außenspiegel und Räder. Da sie nicht als „flache“ Grafik auf das Fahrzeug gelegt werden sollen, bieten sie verschiedene Möglichkeiten der Realisierung als Geometrie.

Die Stossstange eines Autos besitzt üblicherweise eine U-Form. Für die vereinfachte polygonale Beschreibung, wie sie in (a) der Abb. IV-3 dargestellt ist, sind 16 Punkte, beziehungsweise 22 Dreiecke notwendig - die nach Innen, zur Karosserie, zeigenden Flächen müssen nicht vorhanden sein, da sie vom Spieler nicht gesehen werden können.

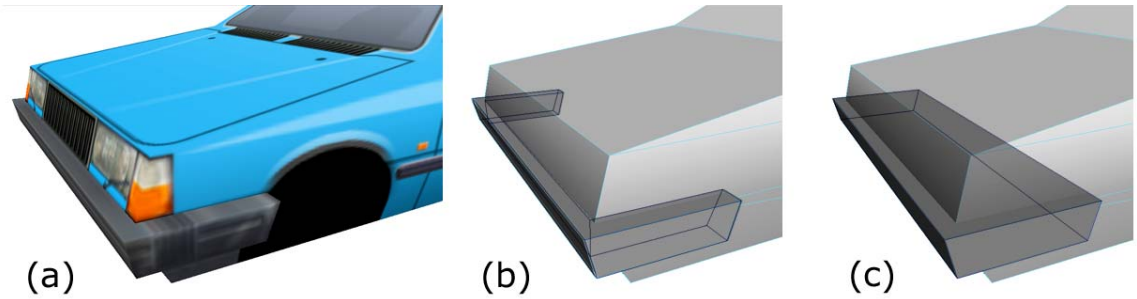

Abb. IV-3 Zwei Möglichkeiten die Stoßstange eines Fahrzeugs zu modellieren

Ein identisches Ergebnis kann allerdings auch mit wesentlich weniger Flächen erzielt werden. Der Trick ist, ein quaderförmiges Objekt in das Auto zu schieben, so dass nur noch ein U-förmiger Teil herausragt – (c) in Abb. IV-3. Da die Karosserie des Autos nicht transparent ist, kann die Methode vom Spieler auch nicht wahrgenommen werden. So entsteht im Gegensatz zur ersten Variante, eine Stoßstange, die nur aus 8 Vertices und 12 Dreiecken besteht.

Variante 1	Variante 2
16 Vertices	8 Vertices
22 Dreiecke	12 Dreiecke

Tab. IV-1 Vergleich der benötigten Geometrie für die Stoßstange des Autos

Die Einsparung von 10 Polygonen mag auf den ersten Blick gering erscheinen. Aber in diesem Fall handelt es sich um eine Geometrieersparnis von etwa 50 Prozent für eine einzige Stoßstange. Da für jedes Auto zwei Stoßstangen eingesetzt werden und sich in der Stadt von „Skate Attack“ viele Fahrzeuge bewegen sollen, kommen schon bei 5 Fahrzeugen hundert Flächen zusammen, die für andere Objekte genutzt werden können.

Wie in (c) der Abb. IV-3 angezeigt, entstehen allerdings durch die zweite Modellierungsvariante größere Flächen, die gezeichnet werden müssen. Dadurch erhöht sich die Zahl der Pixel, die vom Grafikchip verarbeitet werden muss (Füllrate). Welche Methode bezüglich der Laufzeit optimaler ist, soll der anschließende Test zeigen.

3.1.1 Versuch „Fahrzeug-Stoßstange“

Der Test wird in speziell hierfür erstellten Szenen durchgeführt, in der sich 10 Fahrzeuge befinden, die nebeneinander angeordnet sind, siehe Abb. IV-4.

Szene 1 enthält neben der Fahrzeugkarosserie, mit 36 Polygonen, die Stoßstangen der ersten, realistischeren Modellierungsmethode - pro Auto weitere 44 Polygone. In Szene 2 werden dagegen die Geometrie-sparenden Stoßstangen eingesetzt – 24 Dreiecke für jedes Fahrzeug.

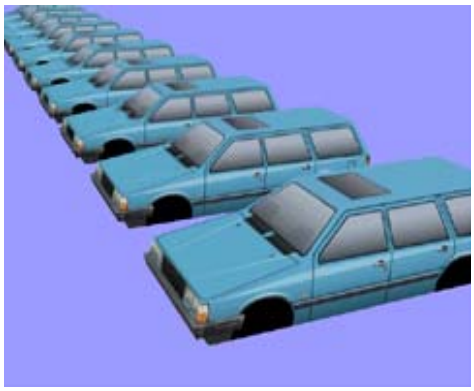

Abb. IV-4 Versuchsanordnung – Versuch „Fahrzeug-Stoßstange“

Aus jeweils 5 Durchläufen zu 10 Sekunden Messdauer wird dann die durchschnittliche Frame-Rate der Szenen ermittelt.

Das Resultat dieses Versuchs ist, dass die Szene 1 in der mehr Polygone gezeichnet werden müssen im Vergleich zur Szene 2 rund 5 Prozent langsamer gerendert wird.

Dass die Füllrate kaum ins Gewicht fällt, häng vor allem mit der Bildschirmauflösung von 800x600 Punkten zusammen. Die Zahl der zu berechnenden Pixel ist dadurch verhältnismäßig gering. Ein weiterer Versuch mit den gleichen Testszenen brachte bereits in einer Auflösung von 1280x1024 Bildpunkten einen kaum mehr messbaren Geschwindigkeitsunterschied. Da „Skate Attack“ jedoch als Konsolenspiel geplant ist, hat diese Modellierungsmethode seine Berechtigung.

3.2 Fahrzeug – Räder

Runde Objekte, wie in diesem Fall die Räder der Autos, stellen im Bereich der Low-Poly Modellierung ein Problem dar. Da die Rundungen durch einzelne Flächen angenähert werden, können sie vom Spieler auch schnell als solche erkannt werden. Bei den Reifen kommt hinzu, dass der Spieler direkt auf die Silhouette der Geometrie sieht. Ein überzeugend rundes Rad, bedeutet daher eine sehr hohe Polygonzahl. Abb. IV-5 demonstriert, wie die Illusion eines runden Reifens mit abnehmender Flächenzahl sinkt (von links nach rechts).

Abb. IV-5 Die Silhouette der Autoreifen lässt die Zusammensetzung aus Einzelflächen erkennen

Für einen Reifen, wie er ganz links im Bild dargestellt ist, ist ein Zylinder notwendig, der aus mindestens 60 Dreiecken besteht.

Eine wesentlich bessere Lösung bieten Texturen mit Alpha-Kanal. Die Reifentextur wird dazu nicht auf einen Zylinder gemappt, sondern eine quadratische Fläche, die der Größe des Reifens entspricht. Mit Hilfe des Alpha-Kanals ist es dann möglich, einen exakt runden Reifen auszumaskieren. Der alleinige Einsatz solcher Texturen führt aber nicht ganz zum gewünschten Ergebnis. Blickt der Spieler von vorne auf das Auto, ist eindeutig zu erkennen, dass die Reifen keine Tiefe besitzen. Hinzukommt, dass von der Seite betrachtet, die vom Spieler abgewandten Reifen nicht mehr zu sehen sind – wie in (a) von Abb. IV-6 demonstriert.

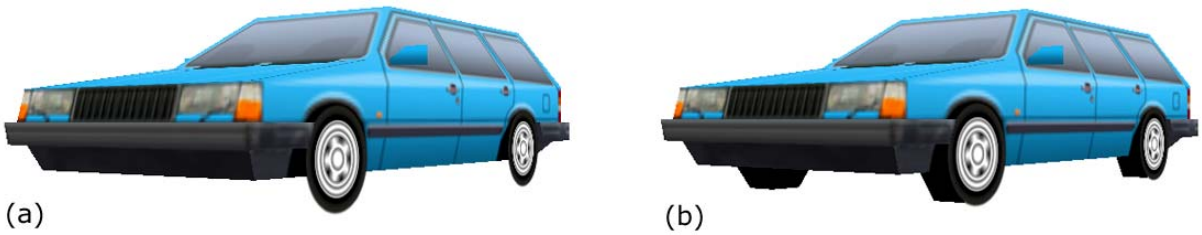

Abb. IV-6 Mit Hilfe von Alpha-Texturen realisierte Reifen

Da die GeForce-Grafikkarte automatisch die Back-Faces entfernt werden (siehe 2.1.4 „Culling Techniken“ im Kapitel III), muss dies bei der Erstellung der Spielumgebung von „Skate Attack“ beachtet werden. Wenn, wie in diesem Fall, aus Effizienzgründen zur Modellierung eine polygonale Fläche eingesetzt wird, besitzt diese nur Front-Faces (Modeller sind meist so voreingestellt, dass sie Flächen beidseitig anzeigen – erst bei eingeschaltetem Back-Face Culling wird dieser Effekt dann deutlich). Damit die Rückseite ebenfalls sichtbar ist, gibt es zwei Möglichkeiten. Entweder wird ein zweites genau entgegengesetzt orientiertes Objekt erstellt, oder die Facetten, die von der Rückseite aus sichtbar sein sollen, werden dupliziert und deren Normale umgekehrt. Bei den Rädern besteht allerdings weiterhin die Problematik, der fehlenden Tiefe. Diese kann durch ein simples geometrisches Objekt, wie in Abb. IV-5 ganz rechts abgebildet, erzeugt werden, so dass als Ergebnis eine glaubhaft rundes Rad entsteht, siehe (b) der Abb. IV-6.

3.2.1 Versuch „Fahrzeug-Reifen“

Die beiden zuvor beschriebenen Möglichkeiten ein rundes Rad zu erstellen, unterscheiden sich stark im Anspruch an Geometriedaten. Werden für die modellierte Variante 60 Polygone benötigt, sind es mit Hilfe der Alpha Textur nur noch 15 (2 für die Fläche der Textur, 13 für die dahinter befindliche Geometrie des Reifens).

Wie sich diese Unterschiede auf die Laufzeit auswirken soll mit einem weiteren Test überprüft werden. Dazu wird die Testszene aus 3.1.1, in der sich das Auto mit der optimierteren Stoßstange befindet, um die jeweils verschiedenen Reifenvarianten ergänzt.

Das Resultat ist, verglichen mit den Ergebnissen des ersten Versuchs, aufgrund der stärkeren Differenz der Polygonzahl wesentlich deutlicher. Szene 2, die aufgrund der verwendeten Alpha-Textur nur noch 1200 Dreiecke benötigt, läuft verglichen mit Szene 1, die infolge der aufwendigeren Reifen 3000 Dreiecke enthält, um rund 43 Prozent schneller. Somit tragen die transparenten Grafiken durch die drastische Reduzierung der benötigten Geometriedaten zur Verbesserung der Laufzeit bei und ermöglichen in diesem Fall ein ausgezeichnetes optisches Resultat.

3.3 Fahrzeug - Außenspiegel

Bei der Modellierung der Wagenaußenspiegel kann nach den bisherigen Kenntnissen die polygonreduzierende Methode, wie sie bei der Stoßstange erläutert wurde, in Erwägung gezogen werden.

Im Gegensatz zur Stoßstange, wird jedoch nicht das gleiche sichtbare Ergebnis erzielt. In (a) der Abb. IV-7 ist der Spiegel aus nur einem Objekte modelliert. Da Wert auf die wagentypische Form der Außenspiegel – zu sehen in (b) - gelegt wird, werden in diesem Fall zwei einzelne Objekte benötigt.

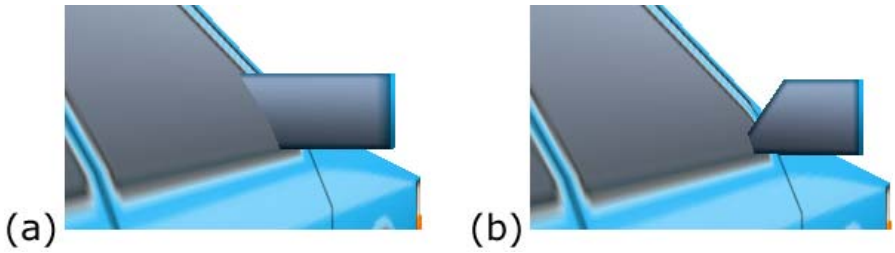

Abb. IV-7 Optischer Vergleich zweier Modellierungsmöglichkeiten der Wagenaußenspiegel

Zusammen mit der Karosserie, den zwei Stoßstangen und den vier Reifen, die wiederum aus einer einfachen Geometrie und der davor befindlichen texturierten Fläche bestehen, entsteht ein Fahrzeug, das sich aus 13 Einzelteilen zusammensetzt.

Wie aus der Analyse der allgemeinen Optimierungstechniken hervorgeht, werden für die Grafikkarte allerdings Meshes empfohlen, deren Polygonzahl bei 200 Dreiecken liegt, beziehungsweise darüber. Es bietet sich daher an, das gleiche Fahrzeug aus nur einem polygonalen Grundkörper herauszumodellieren.

Eine wesentliche Änderung gegenüber der Methode, das Fahrzeug aus einzelnen Objekten zusammenzusetzen, besteht in der Anbindung der Details (Außenspiegel, ...). War es vorher möglich die Karosserie aus nur wenigen großen Flächen entstehen zu lassen, müssen diese jetzt unterteilt werden, da sich sonst keine Feinheiten herausmodellieren lassen. In Abb. IV-8 wird die Problematik anhand der Außenspiegel verdeutlicht.

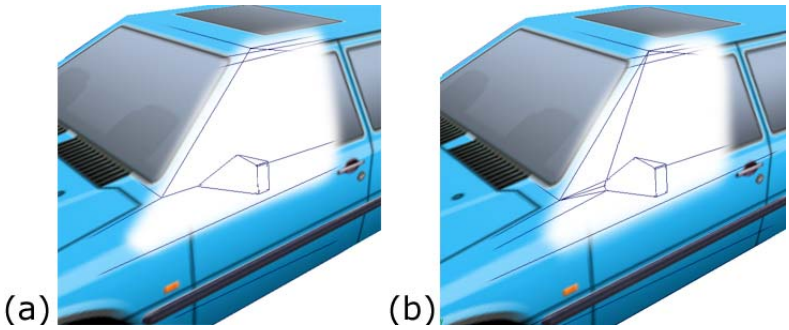

Abb. IV-8 Zusammengesetzte Objekte sparen Verbindungspunkte

Die Modellierung der Außenspiegel erfordert zusätzliche Vertices am Fahrzeug. Diese Verbindungspunkte führen jedoch dazu, dass zur Beschreibung der Karosserie mehr Polygone notwendig sind – siehe (b) der Abb. IV-8. Im Vergleich dazu stellt (a) die Methode des Zusammensetzens dar.

Das auf diese Weise erzeugte Fahrzeug mit all seinen Details, liegt zwar am Ende als einzelnes Objekt vor, benötigt insgesamt aber mehr Dreiecke:

Aus einem Objekt modelliert	Aus Einzelteilen bestehend
154 Vertices	74 Vertices
292 Dreiecke	108 Dreiecke

Tab. IV-2 Vergleich des Geometrieaufwandes

Wie aus Tab. IV-2 hervorgeht, ist der Unterschied zwischen beiden Modellierungsmethoden enorm. Das aus einem Körper herausmodellierte Fahrzeug benötigt fast die dreifache Menge an Dreiecken.

3.3.1 Versuch „Fahrzeug-Modellierung“

Mit diesem Versuch soll getestet werden, welchen Vorteil die zuvor beschriebene Modellierungsmethode bietet.

Die Versuchsanordnung ähnelt der Testszene aus Versuch „Fahrzeug-Stoßstange“. Die erste Szene enthält den Wagen der aus Einzelstücken zusammengesetzt ist und in der zweiten Szene wird die ein-Objekt-Version eingesetzt.

Die Messwerte ergeben ein bemerkenswertes Resultat. Szene 2, in der wesentlich weniger, dafür polygonstärkere Objekte vorhanden sind, wird mit rund 14 Prozent deutlich langsamer gerendert, als Szene 1 mit vielen Objekten, die sich ihrerseits aus wenigen Dreiecken zusammensetzen. Das Ergebnis lässt sich sicher auf das ungleiche Verhältnis der Polygonzahl zurückführen, zeigt aber, dass die Modellierung aus einem Grundkörper nicht zur Effizienzsteigerung der Laufzeit beiträgt.

Es gibt jedoch noch eine dritte Möglichkeit, die die Vorzüge beider zuvor genannten Methoden vereint. Das Objekt, in diesem Fall das Fahrzeug, wird dafür zunächst aus den notwendigen Einzelteilen modelliert. Mit Hilfe der Combine-Funktion in Maya lassen sich anschließend die separaten Polygonmeshes zu einem Einzigen zusammensetzen. Da an den Berührungsstellen zweier Objekte keine Verbindungspunkte entstehen, bleibt die Polygonzahl gleich. Auf diese Weise lassen sich auch Meshes verbinden, die räumlich voneinander getrennt liegen. In einem Test soll überprüft werden, inwieweit sich diese Maßnahme auf die Laufzeit auswirkt.

3.3.2 Versuch „Fahrzeug-Modellierung B“

Die Versuchsanordnung entspricht der Testszene des vorigen Versuchs. Nur wird in diesem Fall der aus Einzelstücken zusammengesetzte Wagen mit der Version verglichen, die mit Hilfe der Combine-Funktion erstellt wurde.

Die Messwerte zeigen, dass sich viele kleine Objekte nachteilig auf die Laufzeit auswirken – die Szene in der der Wagen aus 13 Einzelteilen besteht wird rund 6 Prozent langsamer gerendert, als die Szene in der aufgrund der Combine-Funktion ein Mesh pro Wagen entsteht. Damit ist diese Form der Modellierung im Design-Prozess zu bevorzugen.

Bei der Erstellung der Character kann die Combine-Funktion allerdings nicht unbedacht eingesetzt werden. Soll nicht nur das gesamte Objekt animiert werden, sondern einzelne Teile, wie die Räder der Autos oder der Greifarm eines Baufahrzeugs, können diese nicht mit dem restlichen Objekt vereint werden.

3.4 Nicht-drehende Reifen

In „Skate Attack“ weist der Verkehr eine Besonderheit auf. Die Fahrzeuge bewegen sich permanent mit gleichbleibender Geschwindigkeit durch die Straßen. Diese Tatsache lässt sich bezüglich der Performanz zweifach nutzen. Da die Wagen nie stehen bleiben, kann die normale Radtextur so bearbeitet werden, dass sie der eines fahrenden Autos gleicht – siehe (a) der folgenden Abbildung. Durch diese optische Täuschung, kann die Berechnung der Animation für sämtliche Räder eingespart werden und die Reifengeometrie muss nicht vom Fahrzeug getrennt werden - beides Faktoren, die auf die Laufzeit des Spiels einen Einfluss haben. Außerdem gelangt auf diese Weise, ohne Rechenaufwand, realistisch wirkende Bewegungsunschärfe (Motion Blur) in das Spiel.

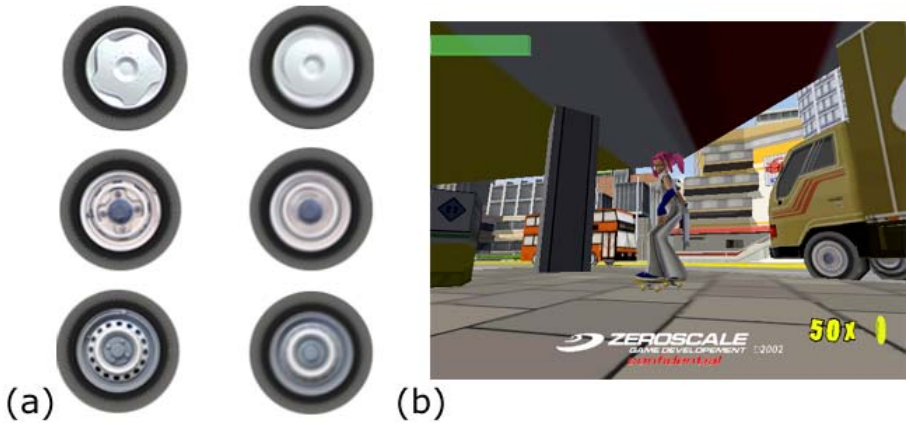

Abb. IV-9 Die Illusion eines sich drehenden Rades

Der Screenshot in (b) der Abb. IV-9 zeigt, wie mit Hilfe von Texturen die Bewegung der Autoreifen vorgetäuscht wird.

3.5 Bounding Box Problem

Das die Combine-Funktion zur Verbesserung der Laufzeit führen kann, hat das Ergebnis des unter 3.3.2 durchgeführten Versuchs gezeigt. Neben den Animationen, müssen aber auch noch andere Aspekte beim Einsatz dieser Funktion beachtet werden.

Wie bereits in Kapitel „Analyse allgemeiner Optimierungstechniken“ unter „View-Frustum Culling“ beschrieben, können sich besonders große Objekte auch nachteilig auf die Performance des Spiels auswirken. Dies soll am Beispiel einer Straßenüberführung, siehe (a) der Abb. IV-10, des Demolevels dargestellt werden.

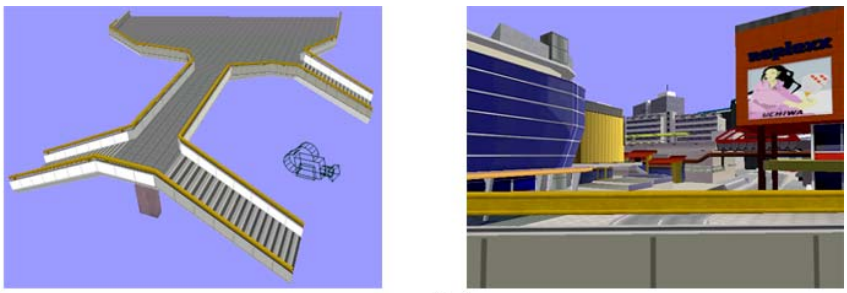

Abb. IV-10 Eine Fußgängerbrücke aus „Skate Attack“

Für die Geometrie der relativ mächtigen Brückenkonstruktion (mehrere Treppenaufgänge - wobei die Treppenstufen aus Effizienzgründen als Textur realisiert sind, eine sich über das gesamte Objekt erstreckende Brüstung, Pfeiler, etc.) werden insgesamt 664 Dreiecke benötigt. Wird mit Hilfe der Combine-Funktion daraus ein einzelnes Mesh erzeugt, entsteht aufgrund der Bauweise eine sehr große Bounding Box.

3.5.1 Versuch „Bounding Box“

In der Spielumgebung soll daher untersucht werden, wie sich diese Bounding Box in speziellen Situationen auf die Laufzeit auswirkt. Dazu werden zwei zusätzliche Kameras in das Demolevel positioniert (Szene 1). Kamera 1 befindet sich in Höhe der Straße, zwischen zwei Treppenaufgängen, wobei der Betrachter mit dem Rücken zur Brücke steht – in (a) der Abb. IV-10 als grüne Kamera dargestellt. Das Sichtfeld der zweiten Kamera ist in Teil (b) der oben aufgeführten Grafik abgebildet. Sie simuliert einen Spieler, der sich direkt auf der Überführung befindet. Die Blickrichtung entspricht der Kamera1. Als Vergleich dient eine zweite Version des Demolevels (Szene 2), in der die Brücke in verschiedene Segmente unterteil ist.

Da sich in den bisherigen Versuchen die Messwerte der verschiedenen Durchläufen kaum unterschieden haben, wird in diesem und den folgenden Tests nur noch ein Durchlauf gemessen, dessen Dauer allerdings von 10 auf 20 Sekunden erhöht wird, um eventuelle Schwankungen des Systems besser ausgleichen zu können.

Die Messergebnisse verdeutlichen, dass sich über einen großen Raum erstreckende Objekte in bestimmten Situationen nachteilig auf die Laufzeit auswirken können. Szene 1, in der sich die Brücke als einzelne Geometrie befindet, wird durchschnittlich 5 Prozent langsamer gerendert, als Szene 2, in der die Brücke in mehrere Ausschnitte unterteilt ist. Ein Vergleich der Angaben der Prana-Engine bezüglich der Objekte und Polygone, die zur Grafikkarte geleitet werden, ergibt, dass in Szene 1 stets die gesamte Anzahl an Polygonen übertragen wird. Selbst bei der Sicht aus Kamera 1, obwohl sich keine

einzige Fläche der Brücke im Bild befindet. In Szene 2 dagegen, werden aus dieser Kameraposition alle Segmente der Brücke durch das View-Frustum Culling entfernt. Aus Sicht der zweiten Kamera, in der der Spieler noch einen kleinen Teil der Brücke sieht, wird nur noch das entsprechende Segment der Grafikkarte übermittelt.

Das Ergebnis zeigt, dass auch dieser Punkt bei der Erstellung der Spielumgebung vom Designer berücksichtigt werden muss. Objekte die, wie in diesem Fall die Brückenkonstruktion, sich über weite Teile des Levels erstrecken aber vom Spieler oft nur zum Teil gesehen werden, sollten entsprechend unterteilt werden, das sie sonst zu Performanzverlusten führen können.

3.6 Instanzen

Instanzen haben zwar den Vorteil, dass sie weniger Speicher benötigen, in „SkateAttack“ werden sie jedoch vor allem zur Arbeitserleichterung eingesetzt. Sämtliche Objekte, die als Instanzen realisiert sind – Fußgänger, Fahrzeuge, Telefonzellen und so weiter, werden über Dummies, die auf die Prana-Datei des entsprechenden Objektes weisen, in die Spielumgebung platziert. Soll ein auf diese Weise ins Level aufgenommener Character verändert werden, muss dies nur noch an dessen Referenzdatei geschehen. Außerdem kann so sichergestellt werden, dass alle Objekte geändert werden und das Aufsuchen in der Szene entfällt.

Um zu verhindern, dass die Spielumgebung von „Skate Attack“ durch die Verwendung von Instanzen nicht zu monoton wird, werden wie in Kapitel III unter 2.1.1 „Instanzen“ bereits angedeutet, unterschiedliche Texturen für derartige Objekte eingesetzt. Vom Designer müssen dazu verschiedene Grafiken angelegt werden (zum Beispiel die Kleidung der Personen - verschiedenfarbig gestaltet, abwechselnde Muster, Aufschriften, ...). Mit Hilfe des Prana-Exporters ist es dann möglich, ein „texArray“ - ein Pool von Grafiken für einen Character - festzulegen, auf den die Prana-Engine beim Start des Spiels zugreift. Die Zuweisung erfolgt dabei zufällig.

3.7 Häuser und Straßenelemente

Die bisherigen Versuchsergebnisse haben gezeigt, dass es bei der Modellierung darauf ankommt, stets so wenig wie möglich Polygone zu benutzten, die notwendig sind die äußere Form des Objektes zu beschreiben.

Das bedeutet natürlich auch, dass Facetten die vom Spieler nicht gesehen werden können auch nicht vorhanden sein müssen. Wird beispielsweise ein Haus aus einem Würfel beziehungsweise Quader erstellt, kann das dem bodenzugewendete Face problemlos entfernt werden, ebenso die Hausseiten, wenn andere Objekte angrenzen. Auf diese Weise entstehen Gebäude-Attrappen, die im einfachsten Fall nur aus der Hausfront bestehen. Gibt es für den Spieler jedoch eine Möglichkeit auf das nichtvorhandene Dach zu schauen, wäre die Illusion zerstört – auch so etwas muss beachtet werden.

Wie bereits angedeutet, werden für Straßensegmente oder entferntere Gebäude in „Skate Attack“ häufig einfache Elemente wie Flächen oder Quader eingesetzt. Bei der Verwendung der Grundkörper aus Maya, muss jedoch darauf geachtet werden, dass diese nicht, wie voreingestellt, aus mehr Einzelflächen bestehen als für die Beschreibung der äußeren Gestalt notwendig ist.

3.8 Lüftungsrohr

Wie in „Fahrzeug - Räder“ dieses Kapitels beschrieben, ließ sich bei den Wagenrädern die vereinfachte Geometrie noch kaschieren, bei freistehenden Objekten, wie Lüftungsrohren oder Laternen ist die Situation etwas komplizierter.

Bei der Darstellung des reduzierten Körpers treten die einzelnen Kanten hervor, wodurch die facettenartige Zusammensetzung der Objektoberfläche deutlich wird. Abb. IV-11 zeigt den Ausschnitt eines Lüftungsrohres, an dem dieser optische Nachteil deutlich wird.

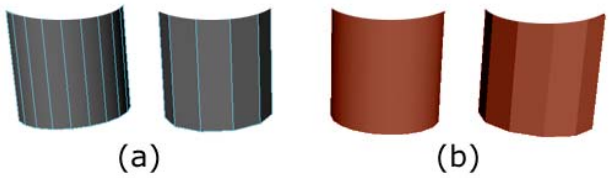

Abb. IV-11 Ein Lüftungsrohr unterschiedlich stark tesseliert.

In (a) der Abb. IV-11 sind zwei Beispiele für die untexturierte Oberfläche des Rohres dargestellt, wobei die blauen Linien die Facettenkanten kennzeichnen. Der Unterschied zwischen beiden Ausschnitten liegt in der verwendeten Zahl an Polygonen. Während für die linke Version 40 Dreiecke benötigt werden, sind es bei der anderen nur noch 22. Die Auswirkung auf das texturierte Objekt ist in (b) der Abb. IV-11 dargestellt.

Diese wahrnehmbare Verschlechterung bei gekrümmten Flächen kann beim Design behoben werden, da in „SkateAttack“ Gouraud Shading zum Einsatz kommt und mit Hilfe von Maya die Vertex-Normalen verändert werden können.

Dazu werden in Maya die Kanten selektiert, die einen „weichen“ Übergang der angrenzenden Flächen ermöglichen sollen (orangefarbene Kante in (1a) der Abb. IV-12) und anschließend die Operation „Soft-Edge“ ausgeführt. Diese bewirkt, das die ursprünglich in verschiedene Richtungen zeigenden Normalen der Flächeneckpunkte (rotfarbene Markierungen in (1a)) hinterher die selbe Orientierung besitzen (rot umrandet in (1b)). Im Ergebnis sieht die Hülle des aus 22 Dreiecken zusammengesetzten Lüftungsrohres (2a) genauso rund aus wie sein komplexeres Ebenbild in (2b) aus 40 Dreiecken.

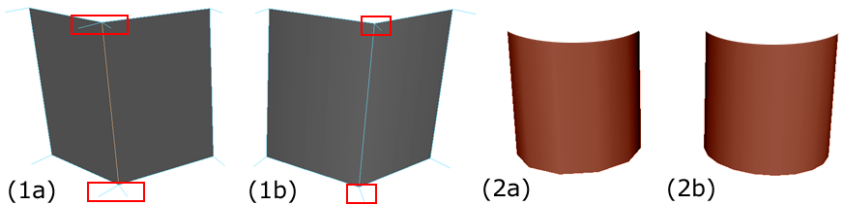

Abb. IV-12 Die „Soft-Edge“ Funktion in Maya ermöglicht weiche Flächenübergange

Auf diese Art können in einer Szene gekrümmte Flächen noch viel stärker reduziert werden als gerade dargestellt. In „Skate Attack“ werden daher Objekte, wie das Belüftungsrohr, durch 5-eckige Formen beschrieben, siehe (a) der Abb. IV-13.

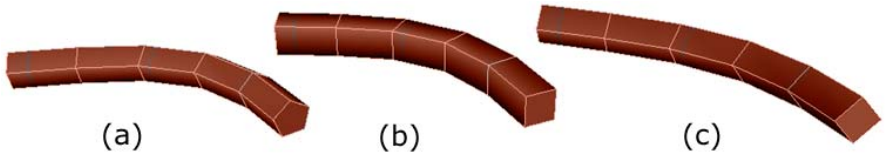

Abb. IV-13 Drastisch reduzierte Geometrieform eines Belüftungsrohres in „Skate Attack“

In manchen Teilen der Spielumgebung werden sogar 4-kantige Rohre verwendet. Dabei wird jedoch ein optisch besseres Ergebnis erzielt, wenn statt der Quader-Form eine Rauten-Form verwendet wird (Vergleich zwischen (b) und (c) der Abb. IV-13). Eigene Erfahrungen haben bestätigt, dass der Einsatz derartiger Objekte nicht nachteilig wahrgenommen wird, besonders wenn sie in Bereichen platziert werden, die sich nicht im direkten Fokus des Spieler befinden (Dachregionen, etc).

Zusätzlich kann verhindert werden, dass die Silhouette der Rohre die stark reduzierte Form sofort erkennen lässt, indem die Enden in andere Objekte (Hauswände, etc.) „hineingeschoben“ werden.

3.9 Baufahrzeug-LOD

Dass das von der Prana-Engine unterstützte Static LOD zur Performanzsteigerung beitragen kann, lässt sich schon aus dem ersten Versuch „Fahrzeug – Stoßstange“ ableiten, der gezeigt hat, dass Objekte mit weniger Polygonen auch tatsächlich schneller gerendert werden, selbst wenn dadurch Meshes entstehen, die nur noch aus 12 Polygonen bestehen.

Wie in 2.1.2 in Kapitel III ausführlich beschrieben, müssen bei dieser Art von LOD die gröberen Objektstufen vom Designer erstellt werden. Wobei sich zwei mögliche Herangehensweisen für die Fertigung der Zwischenstufen ergeben. Einerseits können während der Modellierungsphase des Modells Kopien erstellt werden, andererseits lassen sich ausgehend vom fertigen Modell Polygone reduzieren. Während letzteres zunächst nur nach einem höheren Arbeitsaufwand klingt, hat sich bei der Arbeit an „Skate Attack“ gezeigt, dass dieses Methode mehr Vorteile bietet: Zwar steht schon zu Beginn der Modellierung fest wie ein Objekt am Ende aussehen soll, dennoch entstehen an der finalen Version gelegentlich Änderungswünsche um die Wirkung der äußeren Gestalt optisch zu verbessern. Da sich dieser Feinschliff erst an der endgültigen Form vornehmen lässt, wären zuvor angefertigte LOD-Stufen nicht mehr optimal an diese Version angepasst und müssten ebenfalls bearbeitet werden. Ein Vorteil entsteht aber auch beim Erzeugen der UV-Map des Modells, dass heißt beim Ausrichten der UV-Koordinaten, so dass die Texturen korrekt platziert sind, wenn sie der Oberfläche zugewiesen werden. Bei der Reduzierung der höchsten Detaillierungsstufe, bleibt die UV-Map zu weiten Teilen erhalten, so dass dadurch nicht nur Arbeitsaufwand gespart werden kann, sondern auch die Textur zwischen zwei LOD-Stufen nicht aufgrund unterschiedlicher UV-Maps „springt“ und damit den „Pop-Effekt“ verstärkt.

Die Zahl der gröberen Objektstufen sowie der Grad der Reduzierung hängt stark vom Objekt ab. Ein Fahrkartenautomat, dessen Form schon sehr reduziert ist bietet nicht die gleichen Möglichkeiten wie beispielsweise ein Baufahrzeug, mit vielen Details wie runde Hydraulikstangen, et cetera. Als zweckmäßig stellen sich jedoch 3 Stufen dar – für kurze, mittlere und weite Entfernungen, beziehungsweise eine vierte „leere“ LOD-Stufe, so dass das Objekt in weiter Distanz zum Spieler ausgeblendet wird. Wie sehr sich die Polygonzahl beim bereits angesprochenen Baufahrzeug reduzieren lässt, zeigt die Abb. IV-14.

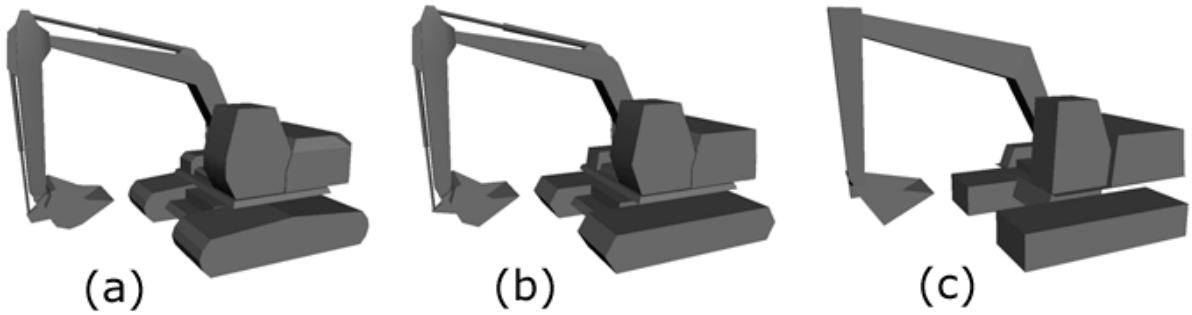

Abb. IV-14 verschiedene LOD-Stufen (LODs) eines Objektes

Die detailreichste Version benötigt 563 Dreiecke – (a) Abb. IV-14. Durch Reduzierungen an den Rundungen der Ketten, Schaufel, Hydraulikstangen (nur noch 3-eckige Gebilde) und so weiter, besteht das mittlere Polygonmeshes bereits nur noch aus 349 Dreiecken, siehe (b) der Abb. IV-14. Die letzte Detailstufe erfordert gerade einmal 109 Dreiecke, wie in (c) der Abb. IV-14 dargestellt.

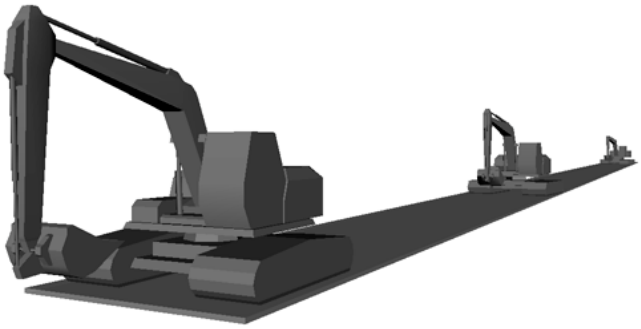

Abb. IV-15 Entferntere Objekte benötigen weniger Details

Abb. IV-15 macht deutlich, dass der Einsatz der reduzierten Baggerstufen nicht zwangsläufig zu einer Verschlechterung der Bildqualität führt, da ab einer bestimmten Entfernung, aufgrund der geringen Größe, Details nicht mehr dargestellt beziehungsweise vom Spieler wahrgenommen werden können. Dazukommt, dass sich die Wahrnehmung des Betrachters viel stärker auf die nähere Umgebung konzentriert, als auf sehr weit entfernte Objekte.

Bei den optischen Tests zur Einstellung der Threshold-Werte hat sich interessanterweise ein Vorteil für einige der Character ergeben, die, wie unter 3.3.2 beschrieben, aufgrund von Animationen aus Einzelteilen bestehen müssen. Während normalerweise ab genau einem Entfernungswert das gesamte Objekt zwischen zwei Stufen wechselt, können in diesem Fall für die Einzelteile unterschiedliche Werte festgelegt werden. So können bei den aus einzelnen Glieder bestehenden Beinen von „Jar“ – einem gegnerischen Kampfroboter, die stärker differenzierten Füße durch gröbere Stufen ersetzt werden, bevor zum Beispiel die Auswechslung der weniger detaillierten Oberschenkel nötig ist. Auf diese Weise werden schon bei näherer Distanz weniger Polygone benötigt.

3.10 Billboard-Baum

Wie im Kapitel III unter 2.1.3 “Billboards“ bereits ausführlich erklärt, sind die Anwendungsmöglichkeiten dieser Technik relativ stark eingeschränkt. Um den Vorteil der Polygonersparnis den diese Technik bietet, dennoch zu nutzen, werden bei den Laubbäumen Stamm und Baumkrone als getrennte Objekte erstellt. Die annähernd runde Kronenform eignet sich sehr gut für die Rotation um mehrere Raumachsen – der grünumrandete Teil in (a) der nachfolgenden Abbildung. Der Stamm, ein polygonales Objekt, kann indessen stets senkrecht auf dem Boden stehen bleibt, während das Blattwerk zum Spieler zeigt.

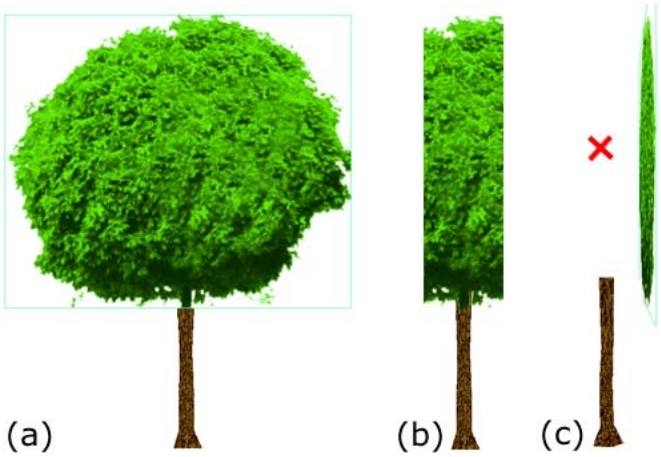

Abb. IV-16 Ein zum Teil über die Billboard-Technik realisierter Laubbaum in „Skate Attack“

Diese Methode hat allerdings auch “Schönheitsfehler“ – damit der Übergang zwischen der transparenten Blatttextur und dem Stamm nicht sofort vom Spieler wahrgenommen wird, darf der Baumstamm nicht nur bis zur Grenze des Billboards gehen, sondern muss ins Laubwerk hineinragen – Vergleich zwischen (a) und (b) der Abb. IV-16. Damit das 3-dimensionale Objekt die Textur jedoch nicht „schneidet“ befindet sich das Billboard versetzt, so dass es sich vor dem Stamm befindet, wie in (c) der Abbildung dargestellt ist. Für eine exakte Rotation muss sich das Rotationszentrum allerdings über dem Holz befinden – rotes Kreuz in (c) der Abbildung. In dieser Konstellation kann es zwar passieren, dass sich, bei einer Sicht vom Boden aus, das Blattwerk durch den Baumstamm hindurch bewegt, die Möglichkeit, dass der Spieler in diese Lage gerät, ist allerdings sehr gering.

3.11 richtig dimensionierte Texturen

Da Texturen ein sehr wichtiger Bestandteil der Spielumgebung von „Skate Attack“ sind, ist es auch wichtig zu wissen, wie sie am effizientesten eingesetzt werden können. Wie schon aus den allgemeinen Optimierungstechniken bekannt, sollten Texturen möglichst klein und quadratisch sein. Noch viel wichtiger ist es jedoch, dass die Dimension einer Textur immer der Form 2^x x 2^y entspricht, demzufolge Höhe und Breite Potenzen von 2 sind. Der Grund dafür liegt in der automatischen MipMap-Generierung der Prana-Engine, die eine derartige Dimension erfordert.

3.11.1 Versuch „Textur-Dimension“

Mit diesem Test soll gezeigt werden, wie wichtig es ist, dass sich die Texturen in der zuvor beschriebenen Auflösung befinden.

Als Testszenario dient das Demolevel, in dem sämtliche Texturen im korrekten Format (32x64, 128x128, ...) vorliegen. In die Szene werden dazu mehrere Kameras platziert, die den Betrachter auf verschiedene Bereiche des Levels blicken lassen. Abb. IV-17 stellt zwei dieser Kamerapositionen dar.

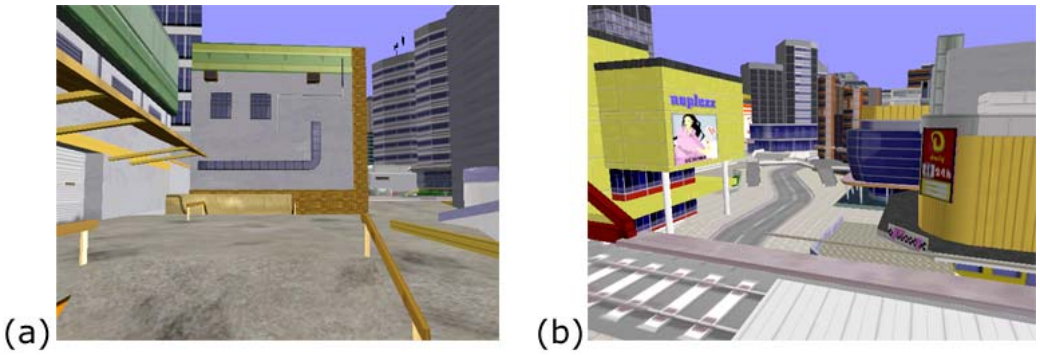

Abb. IV-17 zwei Kamerapositionen des Versuchs „Textur-Dimension“

Zum Vergleich wird die gleiche Szene mit „falsch-dimensionierten“ Texturen geprüft, indem die Ausgangstexturen (BMP und TGA Dateien) um jeweils ein Pixel in Höhe und Breite vergrößert werden. Während diese Dateien (BMPs und TGAs) auf dem Speichermedium dadurch kaum mehr Platz benötigen, zeigt sich die Beeinträchtigung bei den MipMap-Texturen (DDS-Dateien) beziehungsweise bei der Auslastung des Texturspeichers. Benötigten die DDS-Dateien der ersten Szene noch rund 6 MB, waren es jetzt fast 24 MB, 4 mal soviel. Dies liegt daran, dass sämtliche Texturen auf die nächst größere Dimensionsstufe, die eine Potenz von 2 darstellt, skaliert werden. Die DDS-Version einer 65x129 Pixel Grafik beispielsweise benötigt somit 128x256 Pixel.

Das Ergebnis zeigt wie schnell unnötigerweise mehr Texturspeicher belegt werden kann, wenn die Auflösung der Texturen nicht der Form 2^x x 2^y entspricht. Erstaunlicherweise sind die Renderzeiten in beiden Versionen des Demolevels nahezu gleichgeblieben. Die Ursache dafür liegt vermutlich in der Funktion der MipMaps. Da keine Kamera direkt vor eine Wand positioniert ist, sondern den Betrachter mitten ins Level blicken lässt, verwendet die Grafikkarte zur Darstellung der meisten Texturen die gröberen MipMap-Stufen wodurch die Performanz nicht negativ belastet wird.

3.11.2 Versuch „Textur-Dimension B“

Ob die zuvor beschriebene Annahme richtig ist, soll mit diesem Versuch überprüft werden.

Dazu wird aus Versuch „Textur-Dimension“ das Demolevel mit den „falschen“ Texturgrößen verwendet und noch einmal die Frame-Rate gemessen. Zum Vergleich werden dann sämtliche MipMap-Stufen aus den bereits vorliegenden DDS-Dateien entfernt.

Das Ergebnis ist eindeutig – die Frame-Raten sinken dabei um durchschnittlich 70 Prozent. Dies zeigt, welchen Einfluss MipMaps auf die Laufzeit des Spiels haben können beziehungsweise welchen Vorteil Texturen mit geringerer Auflösung beim Rendern bieten.

Wie aus den letzten beiden Versuchen hervorgeht, können größer dimensionierte Grafiken eingesetzt werden, ohne mit beträchtlichen Performanzeinbußen rechnen zu müssen. Diese benötigen allerdings auch mehr Platz im Texturspeicher und wirken damit der Texturenvielfalt entgegen. Dagegen stehen die schnelleren „kleinerer“ Bilder, die dem Designer aufgrund der geringeren Texelzahl, auch weniger Möglichkeiten zur Unterbringung von Details bieten. Andererseits müssen die Grafiken nicht extrem

hochauflösend sein, da „Skate Attack“ als Konsolenspiel entwickelt wird und somit das Ausgabegerät in erster Linie ein Fernseher sein wird. Ein weiterer wichtiger Aspekt bei der Bestimmung der Auflösung ist daher die tatsächlich dargestellte Größe der Textur auf dem Bildschirm. Für Hausfassaden, an die der Spieler nie sehr nah herankommt, sowie für kleine Objekte beispielsweise Briefkästen, Papierkörbe, Hydranten, etc. die stets nur einen kleinen Teil des Anzeigefensters einnehmen, werden auch nur entsprechende Grafiken benötigt. Fahrzeuge, an die sich die Spielfigur dranhängen und mitziehen lassen kann, werden indessen oft vom Spieler sehr nah gesehen und vergleichsweise groß dargestellt. Die gilt besonders für den beziehungsweise die Hauptcharaktere des Spiels. Für die korrekte Dimensionierung einer Textur, die in „Skate Attack“ eingesetzt werden soll, spielen daher folgende Faktoren ein wesentliche Rolle:

- die PAL bzw. NTSC Auflösung eines Fernsehgerätes
- die Größe der Textur auf dem Bildschirm
- die Häufigkeit, mit der die Textur vorkommt
- der Grafikspeicher der Xbox als Limit

3.12 problematische MipMaps

Aufgrund der bildverbessernden Eigenschaften und der Möglichkeiten zur Performanzsteigerung (siehe 3.11.1 Versuch „Textur-Dimension“) werden in „SkateAttack“ fast ausschließlich MipMaps verwendet. Eine Ausnahme bildet beispielsweise die „Wolken-Textur“ des Horizonts. Sie befindet sich, im Verhältnis zur restlichen 3D-Szene, immer in der gleichen Entfernung zum Betrachter und muss daher nicht in unterschiedlichen Detaillierungsstufen vorliegen. Das gleiche gilt für die Elemente des Benutzerinterfaces.

Neben diesen Ausnahmen können MipMaps auch Probleme bezüglich der Bildqualität verursachen. Werden für feine Strukturen wie den Maschen eines Zaunes transparente Texturen eingesetzt um Polygone einzusparen, können durch die gröberen Texturstufen entscheidende Details verloren gehen.

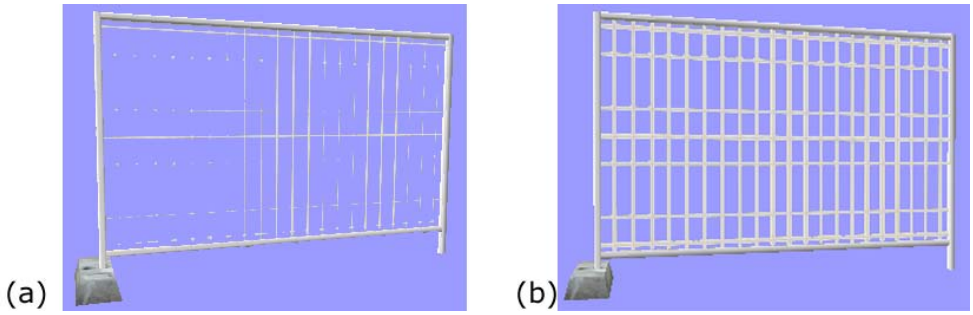

Abb. IV-18 Problematische Situationen für MipMaps

In (a) der Abb. IV-18 ist ein Bauzaun abgebildet, bei dem durch das MipMapping viele Maschen nicht mehr als solche zu erkennen sind. Um die Performanzvorteile, die durch den Einsatz der transparenten Grafik sowie der MipMaps entstehen, nutzen zu können erleiden ohne einen optischen Nachteil für den Spieler zu erleiden, haben sich bei der Arbeit an „Skate Attack“ zwei mögliche Wege gezeigt: Einerseits lassen sich mit Photoshop die MipMap-Stufen in den DDS-Dateien nachträglich bearbeiten und andererseits verhindern dickere Linien in den Ausgangsgrafiken, dass sich schon in den ersten gröberen Texturstufen so schlechte Resultate, wie sie in (a) der Abb. IV-18 zu sehen sind, einstellen. Beide Möglichkeiten führen zu einem wesentlich besseren optischen Ergebnis und lassen den Zaun wie in (b) der Abb. IV-18 dargestellt, aussehen.

3.13 Objekttextur

Egal ob es sich um einen Baum, eine Laterne oder um eine Telefonzelle handelt, die Oberfläche setzt sich aus jeweils verschiedenen Strukturen zusammen. Wie im Modellierungsbereich bieten sich zwei Herangehensweisen an: Die Erzeugung entsprechend vieler Grafiken die für ein Objekt benötigt werden oder einer großen Textur auf der sämtliche Details untergebracht sind. Aufgrund der Ergebnisse aus Versuch „Fahrzeug-Modellierung B“ unter 3.3.2 dieses Kapitels, lässt sich letztere Variante favorisieren – da für das Rendering eines Objekts nicht zwischen verschiedenen Texturen gewechselt werden muss. Im anderen Fall sind Grafiken für verschiedene Objektbereiche wesentlich „kleiner“, wie in 3.11 „richtig dimensionierte Texturen“ beschrieben, ebenfalls ein Vorteil. Dazu folgender Versuch:

3.13.1 Versuch „zerstückelte Objekttextur“

Das schon für viele andere Versuche verwendete Fahrzeug ist für diesen Test besonders gut geeignet, da es sich in viele Bereiche teilt, für die eine eigene Textur infrage kommt, zum Beispiel die Motorhaube, Dach, Außenspiegel, Frontscheibe, Stoßstangen und so weiter. Auf diese Weise entstehen relativ schnell 10 Texturen, wobei die Dimensionen von 16x16 Pixel für die Außenspiegel bis hin zu 256x64 Pixel für die Seiten reichen. Bei der zweiten Variante befinden sich dagegen sämtliche Details der Einzeltexturen auf einer 256x256 Pixel großen Grafik vereint.

Getestet wird, wie schon in Versuch „Fahrzeug-Stoßstange“, anhand spezieller Szenen, in der sich jeweils 10 Fahrzeuge befinden.

Die gemessenen Renderzeiten zeigen einen klaren Performanzvorteil für die Szene mit nur einer Wagentextur. Sie läuft um circa 30 Prozent schneller als die Vergleichsszene, in der zwar „kleinere“ aber mehr Texturen vorhanden sind. Das Ergebnis zeigt, dass es dem Designer möglich ist bei der Texturerstellung positiv auf die Laufzeit des Spiels einzuwirken. Allerdings ist die Erstellung einer solchen Textur schwieriger, da schon zu Beginn die Größe für einzelne Details genau überlegt und entsprechend angeordnet werden muss.

Abb. IV-19 Sämtliche Details eines Objekts werden auf einer Textur zusammengefasst

Der Vorteil dieser Methode beschränkt sich allerdings nicht nur auf einzelne Objekte. Das Zusammenfassen von Grafiken eignet sich besonders gut, wenn eine Vielzahl kleiner Bilder vorliegt. So ergeben zum Beispiel vier 32x128 Texturen genau ein 128x128 großes Bild beziehungsweise wiederum vier von diesen eine 256x256 Textur – die, wie aus der Analyse der allgemeinen Optimierungstechniken bekannt, eine optimale Dimension besitzt.

Kachelbare Texturen lassen sich allerdings nicht so einfach vereinen, da sonst die anderen Grafiken in die Wiederholungen mit einfließen. Tiles die sowohl in U- als auch in V-Richtung wiederholt werden sollen, dürfen gar nicht zusammengefasst werden. Anders bei Kacheln die nur vertikal oder horizontal funktionieren müssen - sie können ebenfalls „verschmolzen“ werden.

3.13.2 Versuch „Reklameschilder“

Es stellt sich allerdings die Frage, inwieweit es sinnvoll ist die Texturen von Objekten zusammenzufassen, die weit über das Level verteilt sind und deshalb nie gleichzeitig dargestellt werden. Würde normalerweise zum Rendern eines Objekts eine kleinere Textur verwendet, käme dann eine wesentlich größere Grafik zum Einsatz. Ein sehr gutes Beispiel dafür sind die Werbeschilder des Demolevels, die sich über die gesamte Stadt verteilen. Ob die Laufzeit negativ beeinflusst wird, wenn nur eine einzige Werbetafeln sichtbar ist, soll mit diesem Versuch getestet werden.

In der ersten Variante des Demolevel liegen sämtliche Werbetexturen als separate Grafiken vor, während in der zweiten Version dafür nur noch ein 1024x1024 Pixel große Bild existiert, siehe (a) der Abb. IV-20. Für den Test wird eine Kamera so in die Spielumgebung platziert, dass der ungünstigste Fall eintritt: aus Sicht des Betrachters, wie in (b) der Abb. IV-20 dargestellt, ist nur ein Schild zu sehen.

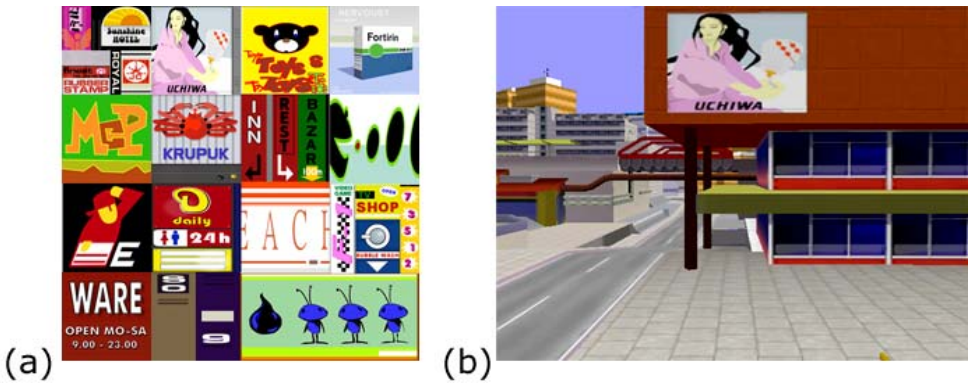

Abb. IV-20 Versuchsanordnung – Versuch „Reklameschilder“

Die gemessenen Renderzeiten der Szenen weisen allerdings keinen Unterschied auf. Das Resultat kann mit dem relativ kleinen Bildschirmanteil und der Verwendung von MipMaps begründet werden (siehe 3.11.2 Versuch „Textur Dimension B“). Das bedeutet, dass im Fall der Reklameschilder die Zusammenfassung aller Werbegrafiken keine negative Auswirkung auf die Laufzeit des Spiels hat.

3.14 Beleuchtung der Stadt

Soll die Spielumgebung in Echtzeit ausgeleuchtet werden, sind Lichtquellen notwendig. Wird nur das Ambiente Licht benutzt, entstehen stark unrealistische Bilder, wie in (a) der Abb. IV-21 dargestellt, die keine räumliche Tiefe vermitteln. Erst wenn „echte“ Lichtquellen dazukommen, wirkt die Szene realistischer und eine 3-dreidimensionale Wahrnehmung der Objekte wird möglich, siehe (b) der Abb. IV-21. Da das von der Oberfläche der Objekte abgestrahlte Licht keinen Einfluss auf die Umgebung hat, bleiben bei nur einer Lichtquelle viele Stellen des Levels im „Dunkeln“. Um jeden Winkel der Umgebung mit Licht zu versehen, müssen mehrere Lichtquellen eingesetzt werden, wodurch jedoch die Performanz des Spiels negativ beeinflusst wird. Um zusätzlich die Diffusion des Sonnenlichts zu simulieren, wären neben mehreren gerichteten Lichtquellen unzählige kleine Punktlichter notwendig, zum Beispiel für die in Abb. IV-21 dargestellte Situation, in der das Wasser Licht reflektiert und dadurch den unteren Teil des Brunnens erhellt.

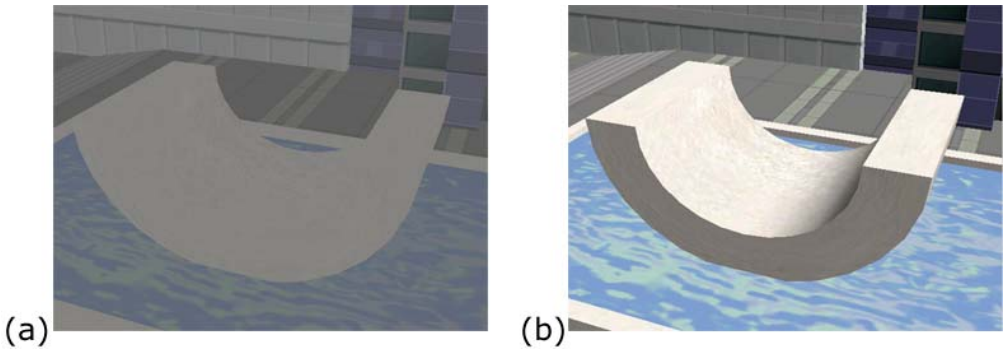

Abb. IV-21 2 Beleuchtungssituationen dargestellt am Stadtbrunnen von „Skate Attack“

Da die Prana-Engine in der Lage ist Vertex Color zu verarbeiten, kann auf das sogenannte Prelighting zurückgegriffen werden, siehe Abschnitt 2.3.1 im Kapitel III „Analyse“.

3.14.1 Versuch „Echtzeitbeleuchtung vs. Prelighting“

Anhand des Demolevels soll überprüft werden, welchen Einfluss die Echtzeitberechnung der Lichtquellen auf die Laufzeit hat beziehungsweise welche Rechenersparnis die Vorbeleuchtung der Szene bietet.

Dazu wird die Zahl der Lichtquellen, die zur Ausleuchtung der Spielumgebung benötigt wird schrittweise erhöht und jedes Mal die Frame-Rate von drei Kamerapositionen aus gemessen. Da eine Tagessituation dargestellt werden soll, werden gerichtete Lichter verwendet. Anschließend wird das auf diese Weise mit Lichtquellen ausgestattete Demolevel in Maya prelighted, wobei die Renderzeiten dieser Szene die Basis für folgendes Verlustdiagramm darstellt:

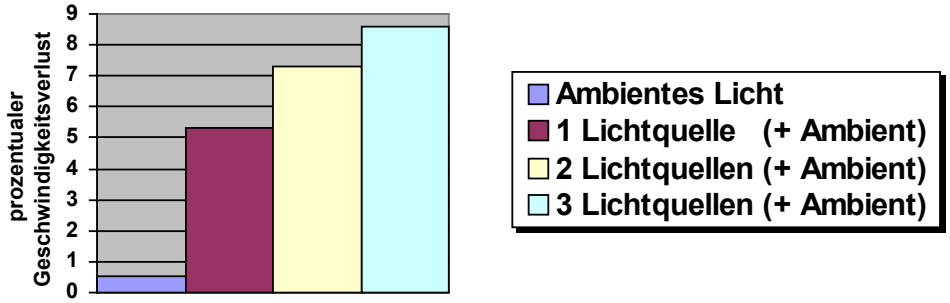

Abb. IV-22 Geschwindigkeitsverlust durch Echtzeitbeleuchtung verglichen mit Prelighting

Die Berechung des Ambienten Lichts zur Echtzeit verursacht keinen nennenswerten Performanzverlust. Das optische Ergebnis ist allerdings sehr mangelhaft, so dass ein Ambientes Licht alleine nicht verwendet werden kann, wie in (a) der Abb. IV-21 bereits gezeigt. Abb. IV-22 macht weiter deutlich, wie der Verlust der Rendergeschwindigkeit mit jeder hinzugefügten Lichtquelle steigt. Da für die Beleuchtung des Levels ein Ambientes Licht und 3 gerichtete Lichquellen benötigt werden, können mit Hilfe von Prelighting rund 8 Prozent Rechenzeit eingespart werden. Mehr Directional Lights führen allerdings zu unschönen Bildergebnissen, da sich die Intensitäten der Lichter immer weiter aufaddieren. Wie zu Beginn dieses Abschnitts erklärt, müssten für eine diffuse Beleuchtung unzählige Punktlichter eingesetzt werden, deren Echtzeitansatz jedoch utopisch ist. Der Vorteil den Prelighting bietet, ist, dass diese Berechnungen im Vorfeld durchgeführt beziehungsweise die erforderlichen Veränderungen der Vertex Colors von Hand vorgenommen werden können.

In „Skate Attack“ werden daher sämtliche statischen Objekte vorbeleuchtet. Aber auch Character können bis zu einem gewissen Grad prelighted werden. So enthalten beispielsweise die Texturen der Fahrzeuge Beleuchtungsinformationen, so dass die Seiten von oben nach unten einen Helligkeitsverlauf aufweisen beziehungsweise das Dach und die Motorhaube heller sind, als andere Bereiche des Wagens.

3.15 Tiefe über Texturen

Da es die Aufgabe beim Design der Spielumgebung ist, mit möglichst wenig Datenaufwand die bestmöglichen optischen Ergebnisse zu erzielen, müssen durch die Einsparungen an den Objektgeometrien Details über die Texturen erzielt werden. Dazu zählen aber auch Effekte, die sich normalerweise nur durch Beleuchtung plastischer Formen ergeben.

Sowohl der in (a) der Abb. IV-23 dargestellte Greifarm des Baggers als auch die Halterung der Videokamera in (b) der Abbildung sind aus Effizienzgründen aus Quadern modelliert, dass heißt der Querschnitt entspricht in beiden Fällen einem Viereck. Damit sie dennoch den typischen Formen entsprechen – wie in den jeweiligen Schemen angedeutet, muss die Textur entsprechend gestaltet werden. Dass heißt helle Linien entlang den Kanten, die der Lichtquelle (Sonne) zugewandt sind beziehungsweise dunkle Linien im umgekehrten Fall.

Durch diese Finessen können nicht nur unzählige Polygone eingespart, sondern auch die optischen Resultate wesentlich verbessert werden. Auf diese Weise können sowohl die Laufzeit als auch die Bildqualität optimiert werden.

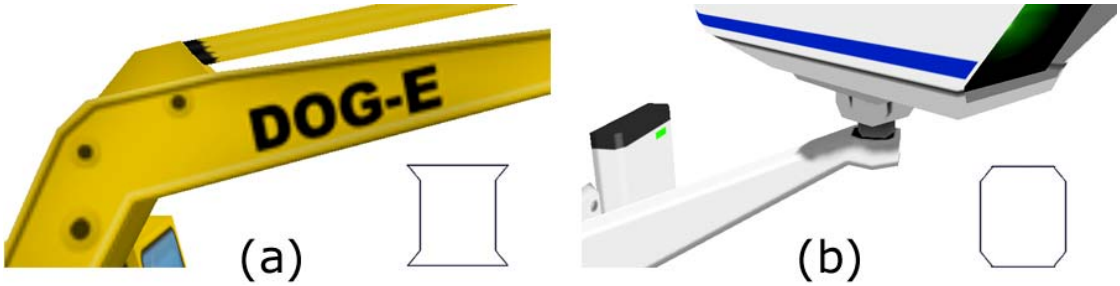

Abb. IV-23 in Texturen aufgenommen Tiefen- beziehungsweise Beleuchtungsinformationen

3.16 Schlagschatten

Eine im Vorfeld definierte Beleuchtungssituation, die sich zur Laufzeit nicht ändert, hat den Vorteil, dass sich die Gestalt und Position der Schlagschatten von statischen Objekten wie Bäumen, Häusern oder Brücken ebenfalls nicht ändert. In diesen Fällen lassen sich die Berechnungen ganz oder teilweise vom Designer übernehmen. Entscheidend dabei ist allerdings, ob Character die sich in den Schatten hineinbewegen auch wirklich dunkler werden sollen. In diesem Fall können nur Volume Shadows eingesetzt werden.

Zur Optimierung der Laufzeit können, wie im Kapitel III „Analyse“ unter 2.3.2 beschrieben, diese Volumen im Vorfeld erstellt werden, da die Prana-Engine beziehungsweise der Prana-Exporter entsprechende Funktionen bereithalten. Seitens der Engine gibt es jedoch eine Bedingung für die Geometrieform – sie muss quaderförmig sein. Wie ein solcher Körper aussieht, zeigt Teil (a) der Abb. IV-24, am Beispiel der Bahnbrücke. Bei der Erstellung in Maya muss zusätzlich darauf geachtet wer-

den, dass die Schattengeometrie stets durch den Boden ragt, da der Schatten sonst nicht korrekt dargestellt wird. Das Ergebnis zeigt Teil (b) der Abbildung.

Eine ganz andere Möglichkeit, bei der die gesamte Schattenberechnung eingespart werden kann, stellen die Vertex Colors dar. Allerdings hat diese Methode den optischen Nachteil, dass die Fahrzeuge, die sich durch den Schattenbereich der Brücke bewegen, nicht abgedunkelt werden. Kann darauf verzichtet werden, erfordert diese Methode jedoch mehr Polygone, wie bereits in Kapitel III Abschnitt 2.3.2 an einem Beispiel gezeigt. Im Fall der Bahnbrücke kommt hinzu, dass sich der Schatten über sehr viele Geometrien des Levels erstreckt, daher muss nicht nur die Fahrbahn unterteilt werden, sondern auch Bordsteine, Bürgersteige und Wände. Aber auch die beiden grünen Klimaschächte beziehungsweise die in der Abbildung weniger gut erkennbaren Schrägen an den Straßenrändern, die dem Spieler als Rampen dienen, müssen gemäß der Silhouette des Schattens unterteilt werden. Es stellt sich jedoch die Frage ob der Mehraufwand an Polygonen nicht wieder zur Verschlechterung der Frame-Rate führt. Dazu folgender Versuch:

3.16.1 Versuch „Schlagschatten-Bahnbrücke“

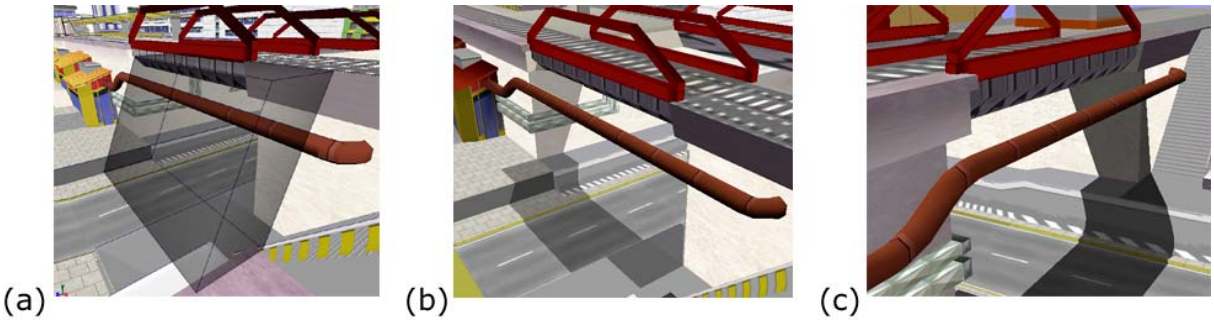

Abb. IV-24 Schlagschatten der Bahnbrücke

In einer ersten Variante des Demolevels, wird der Schatten der Brücke durch ein Schattenvolumen realisiert und dafür die Geometrie erzeugt.. Die zweite Version des Levels, in der die Vertex Colors zum Einsatz kommen, benötigt für ein ähnliches Schattenprofil, siehe (c) der Abb. IV-24, zusätzliche 149 Dreiecke.

Beide Testszenen werden aus jeweils zwei Perspektive gerendert, wie in (b) und (c) der Abb. IV-24 dargestellt.

Die Messwerte zeigen einen klaren Vorteil für die zweite Version des Levels. Trotz der höheren Zahl an Polygonen wird die Szene, gegenüber der ersten Variante des Levels, im Schnitt 13 Prozent schneller gerendert. Das Ergebnis zeigt, dass die Berechnung der volumetrischen Schatten, wie in Kapitel II unter 1.7.1 beschrieben, wesentlich mehr Rechenzeit in Anspruch nehmen, als die Verarbeitung zusätzlicher Polygone. In diesem Fall steht demnach eine höhere Performanz des Spiels einem geringeren Grad an Realismus gegenüber.

Es gibt aber auch Situationen, in denen auf den Vorteil von Volumen Schatten verzichtet werden kann. Wie beispielsweise die Videokamera in Abb. IV-23. Der Schatten, den die Kamera auf die Halterung wirft, kann ohne große Nachteile in der Textur des Objekts, siehe Abb. IV-19, aufgenommen werden. Der einzige „Schönheitsfehler“ dabei ist, dass sich trotz der Kamerabewegung die Gestalt des Schattens nicht ändert, wobei diese Feinheit im Spiel nahezu unbemerkt bleibt.

Um bei den Bäumen einen möglichst realistischen Schatten zu erzeugen sind Schattenvolumen völlig ungeeignet. Aufgrund der zuvor beschriebenen Einschränkung der Prana-Engine, wäre die Silhouette des Schattens viereckig und damit für einen Baum absolut unrealistisch. Der Einsatz von Vertex Colors stell sich insofern als problematisch dar, dass für einen naturgetreuen Schatten mit einer erkennbaren Struktur des Laubwerks, unzählige Polygone in Anspruch nehmen würde. In diesem Fall bieten transparente Texturen den mit Abstand größten Vorteil. Die Vorgehensweise ähnelt der Beschreibung in Abschnitt 3.2 dieses Kapitels: Nur wird diesmal die transparente Grafik auf eine Geometrie gemappt, die direkt über den Bereich der Spielumgebung gelegt wird, der vom Schatten abgedunkelt werden soll. Die Form der Geometrie ist dabei nicht auf eine Ebene beschränkt. Das optische Ergebnis, das dadurch erreicht wird, ist sehr überzeugend – wie die Abb. IV-25 zeigt. Auf einen Performanztest kann an dieser Stelle verzichtet werden, da schon Versuch „Fahrzeug-Reifen“, siehe 3.2.1 dieses Kapitels, in einer ähnlichen Situation einen klaren Vorteil für die transparenten Texturen gezeigt hat.

Damit keine Bildfehler durch die übereinanderliegenden Polygone entstehen, siehe Abb. III-12, muss mit Hilfe des Prana-Exporters die Geometrie der Schattentextur das Attribut „z-Bias“ bekommen, damit dessen Polygone „bevorzugt“ gerendert werden.

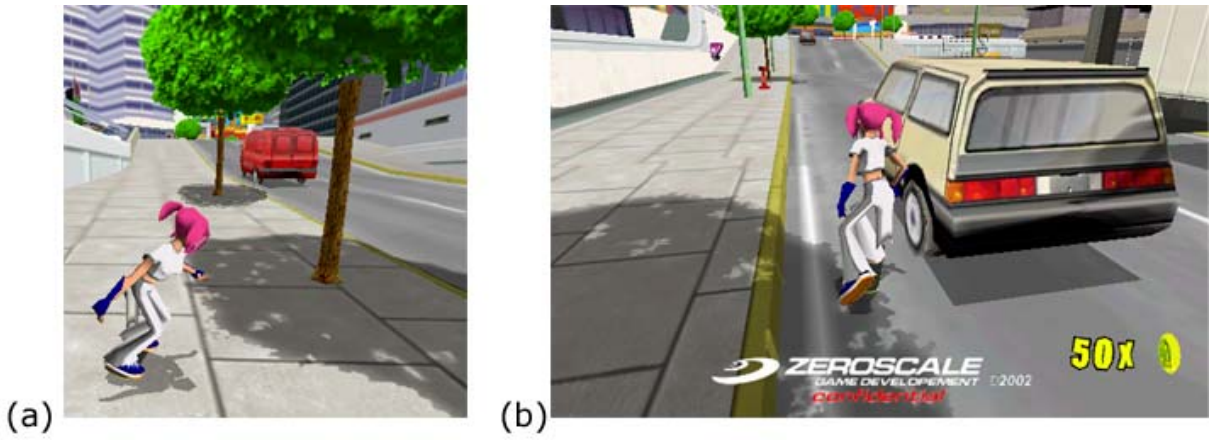

Abb. IV-25 Beispiele für Schlagschatten in „Skate Attack“

Character bieten nur noch wenig Möglichkeiten die Erstellung ihrer Schatten ins Vorfeld zu übertragen, da sich ihr Schatten ebenfalls bewegt und in den meisten Fällen auch die Silhouette ändert. Da es bei den Fahrzeugen allerdings völlig ausreichend ist, dass sich unterhalb des Wagens ein rechteckiger Schatten befindet, wie in (b) der Abb. IV-25 dargestellt, werden die Schattenvolumen ebenfalls im Vorfeld erzeugt. Dieser Teil der Abbildung macht ebenfalls deutlich, dass für weit entfernte Autos ein Schatten völlig überflüssig ist, da er nicht mehr vom Spieler gesehen werden kann. Zur Reduzierung der Laufzeitberechnungen werden die Schattenvolumen daher als 2-Stufen LOD-Gruppen erstellt, die es ermöglichen das Volumen ab einer bestimmten Entfernung auszublenden.

Die Schatten der anderen Character lassen sich bezüglich der Laufzeit nur noch dadurch optimieren, dass vereinfachte Geometrien erstellt werden, aus denen die Prana-Engine dann die Silhouette beziehungsweise die Extrusionskörper erstellt. Bedingt durch die Engine, müssen diese eine konvexe Form besitzen, so dass die Character selbst relativ selten eingesetzt werden können. Gegen den Einsatz der Figuren spricht aber auch, dass es sich „nur“ um dessen Schattenbild handelt, für das wesentlich gröbere Formen völlig ausreichend sind. So benötigt der Hauptcharacter des Spiels zwar 2098 Dreiecke, für die Berechnung des Schattens wird jedoch nur eine Gliederpuppe bestehend, aus 232 Dreiecken

verwendet – etwa ein Zehntel! Wie diese Nachbildung aussieht, zeigt Abb. IV-26, während in (a) der Abb. IV-25 das Resultat zu erkennen ist.

Abb. IV-26 Die Schattengeometrie des Hauptcharacter

Auf diese Weise können ohne einen auffälligen Nachteil bezüglich der Bildqualität, die zur Laufzeit stattfindenden Schattenberechungen vereinfacht werden.

3.17 Collision-Meshes

Damit der Anwender die Spielfigur nicht durch die Objekte der künstlichen Welt hindurchbewegen kann, müssen zur Laufzeit permanent Kollisionsberechnungen stattfinden. Die notwendige Rechenzeit steigt natürlich mit der Anzahl der Objekte beziehungsweise deren Polygonanzahl. Eine Vereinfachung kann allerdings dadurch erreicht werden, dass, ähnlich den zuvor beschriebenen Schattengeometrien in 3.16, im Vorfeld vereinfachte Kollisionsgeometrien erstellt werden. Eine Möglichkeit, die besonders bei komplexeren Figuren viele Polygone einsparen kann. Als besonders nützlich bei der Fertigung dieser Collisionmeshes haben sich die jeweils niedrigsten Detailstufen bereits erstellter LOD-Gruppen erwiesen.

Darüber hinaus können und müssen mit diesen Meshes für die Kollisionsberechnung kritische Situationen vermieden beziehungsweise beseitigt werden. Zur Erklärung: der Kollisionsalgorithmus der Prana-Engine verwendet für die Spielfigur eine Kugel, die sich in etwa der Höhe des Gesäßes befindet und einen Radius bis etwa zur Schulter besitzt. Um daher auszuschließen dass sich die Spielfigur in Geometrien „verfängt“ dürfen die Collisionmeshes keine Lücken, etc. aufweisen.

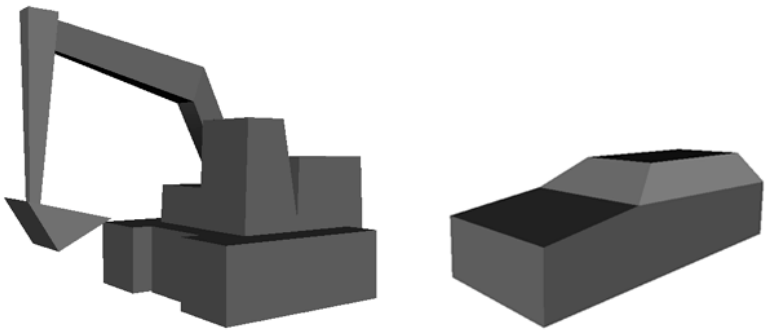

Abb. IV-27 Collisionmeshes zweier Fahrzeuge

Die in Abb. IV-27 dargestellten Collisionmeshes der Beispiele Baufahrzeug und Fahrzeug benötigen nur noch 19 beziehungsweise 14 Prozent der ursprünglichen Polygonzahl.

3.18 Personen

Damit die Strassen und Wege in „Skate Attack“ einer belebten Stadt gleichen, sollen sich möglichst viele Passanten in der Spielwelt bewegen. Je weniger Daten- beziehungsweise Rechenaufwand dabei von einer Person verursacht wird, um so mehr können für das Stadtbild eingesetzt werden. Wie im Kapitel III „Analyse“ unter 2.4 „Animation“ eingehend beschrieben, lassen sich im Bereich der Animation viele Techniken verwenden, um derartige Berechnungen zu minimieren. So werden zum Beispiel die in Maya mit Hilfe von Skelett- und IK-Technik erstellten Bewegungen der Passanten in Keyframes umgewandelt, um die IK-Echtzeitberechungen einzusparen. Eine Ausnahme bildet jedoch der Hauptcharacter – im Spiel soll es nämlich möglich sein, Posen zu erzeugen, die schwer vorhersagbar beziehungsweise zu mannigfaltig sind, als dass es sinnvoll wäre für sämtliche Möglichkeiten einen entsprechenden Clip zu erstellen.

Wie bereits erwähnt, bedeutet eine hohe Zahl an Passanten möglichst wenig Berechnungsaufwand für eine Einzelperson. Dies wird in erster Linie durch einen geringeren Grad an Realismus erzielt – so besteht eine Person aus „nur“ 346 Dreiecken – der Hauptcharacter benötigt 6 mal soviel. Die ebenfalls in der Analyse aufgezeigten Methoden Rechenzeit für das Skinning einzusparen, stellen bei der Gliederpuppe allerdings eine untersuchungswürdige Situation dar. Denn einerseits kann komplett auf das Skinning verzichtet werden, aber andererseits entsteht dadurch eine Vielzahl einzelner Meshes (2 Füße, 2 Unterschenkel, 2 Oberschenkel, ...). Wie die Testergebnisse aus Versuch „Fahrzeug-Modellierung B“, siehe Abschnitt 3.3.2 dieses Kapitels gezeigt haben, erfordern diese wiederum mehr Rechenzeit. Ob eine Gliederpuppe daher noch zur Verbesserung der Laufzeit beträgt, soll mit folgendem Versuch überprüft werden.

3.18.1 Versuch „Gliederpuppe vs. Skinning“

Für diesen Versuch werden spezielle Testszenen erstellt, in denen sich jeweils 20 Passanten befinden, die eine einfache Gehbewegung vollführen. Abb. IV-28 stellt die Versuchsanordnung dar.

Abb. IV-28 Versuchsanordnung – Versuch „Gliederpuppe vs. Skinning“

Die Passanten der Szene 1 sind als Gliederpuppen realisiert – eine einzelne Person besteht dabei aus insgesamt 15 Einzelteilen. In zwei weiteren Szenen werden dagegen komplette Meshes eingesetzt und Rigid-Binding (Szene 2) beziehungsweise Smooth-Binding (Szene3) für das Skinning verwendet. Die Polygonzahl ist in allen Szenen gleich. Neben der Frame-Rate werden die Zeiten für das Skinning und die Verarbeitung der Display List (RenderTime) gemessen.

Das Messergebnis ist, dass die Gliederpuppenszene gegenüber dem Rigid-Binding, mit 13 Prozent, deutlich langsamer gerendert wird, verglichen mit dem Smooth-Binding läuft sie jedoch rund 4 Prozent schneller. Die zusätzlich aufgenommenen Werte zeigen, wie das Ergebnis zustande kommt: Während die Verarbeitungszeit der Meshes in Szene 2 und 3, aufgrund der gleichen Objektanzahl identisch

ist – etwa 10,3 ms, werden für die Vielzahl der Objekte in Szene 1 rund 15,5 ms benötigt. Da für das Skinning in Szene 3 mehr als 6 ms benötig werden, läuft die Szene insgesamt langsamer als die mit den Gliederpuppen. Dagegen nimmt das Skinning der Szene 2 nur 2,6 ms ein, so dass in diesem Fall das Rigid-Binding die schnellste und geeignetste Variante für die Passanten darstellt.

V Ergebnisse und Ausblick

Die Entwicklung eines Computerspiels, speziell im Bereich Level- und Characterdesign, stellt eine Gradwanderung zwischen schönem Aussehen und Performanz dar. Einerseits muss die Spielumgebung interessant gestaltet sein, damit sich als Ergebnis dem Anwender ein attraktives Spielszenario darbietet. Andererseits soll die Frame-Rate möglichst hoch sein, um dem Benutzer ein angenehmes Spielgefühl zu vermitteln. Sämtliche Maßnahmen zur Gestaltung eines abwechslungsreichen Spiels (zum Beispiel möglichst viele Objekte, Texturen, Animationen, etc.) bedeuten jedoch einen erhöhten Datenaufwand und erfordern mehr Berechnungen zur Laufzeit, was der Frame-Rate entgegenwirkt.

Die Forschungsarbeit, die einen wesentlichen Bestandteil dieser Arbeit darstellt, hat gezeigt, dass es eine Vielzahl an Techniken und Strategien gibt, die zur Optimierung der Laufzeit eingesetzt werden können, ohne den visuellen Eindruck stark zu beeinträchtigen. Dabei werden zwei signifikante Ansätze verfolgt:

- Minimierung der Daten
- Berechnungen und Ergebnisspeicherung im Vorfeld

Die Reduzierung von Daten kann, wie im Fall der Texturkompression allerdings Informationsverluste bedeuten, wodurch die Bildqualität der Grafiken beziehungsweise die Optik des Spiels beeinträchtigt wird. Dagegen nutzt Static LOD zwar die natürlichen Limitierungen des menschlichen Auges beziehungsweise des Ausgabegerätes, der Einsatz kann unter Umständen aber auch vom Spieler erkannt werden. Die Mehrfachverwendung von Daten trägt ebenfalls zur Minimierung des Datenaufwandes bei. Aber egal welche Verfahren zum Einsatz kommen, wichtig ist die Berücksichtigung der individuellen Situation und die technische Umsetzung, so dass diese vom Spieler unbemerkt bleiben.

Eine Verlagerung von Laufzeitberechnungen ins Vorfeld, hat den Vorteil, dass die Game-Engine nur noch auf verfügbare Resultate zugreifen muss und somit Rechenzeit eingespart werden kann, zum Beispiel durch Prelighting. Außerdem bieten sich so Möglichkeiten um beispielsweise die MipMap-Stufen einer Textur zu korrigieren oder die gröberen Objektstufen bei Level Of Detail besser aufeinander abzustimmen. In den meisten Fällen bedeutet dies allerdings wiederum eine Zunahme der Datenmenge (Texturen, Geometrien). Der entscheidende Vorteil ist jedoch, dass es sich dabei um Daten handelt, auf deren Verarbeitung die Grafikkarte optimiert ist. Oft gehen Laufzeitoptimierungen zu Lasten von Speicher oder Qualität. Die Schwierigkeit beim Design ist es, den bestmöglichen Mittelweg zu finden, so dass das Spiel funktioniert und dem Anwender Spaß macht.

Durch die praktischen Untersuchungen konnten nicht nur theoretische Kenntnisse überprüft, sondern auch optimierte Modellierungs- und Texturierungsmethoden ermittelt werden, die bei der Erstellung der Spielwelt von „Skate Attack“ zum Einsatz kommen. Ebenso wurden verschiedenste Möglichkeiten aufgezeigt, wie beim Design die Bildqualität verbessert werden kann und wie sich Nachteile durch den Einsatz von bestimmten Techniken beheben lassen, so dass insgesamt ein wesentlich besseres Ergebnis erzielt wird.

Ein Demolevel, in dem die durch Theorie und Praxis gewonnenen Erkenntnisse zum Einsatz kommen, ist bereits fertig und hat auch schon von vielen Seiten Zustimmung erhalten, unter anderem bei

einer Präsentation an der Games Academy[12] in Berlin. Aber nicht nur auf regionaler Ebene konnte damit Interesse geweckt werden. Bei der Bewerbung für die Milia 2002[13], im Februar diesen Jahres, schaffte es „Skate Attack“ unter die 30 Besten zu gelangen, woraufhin ZeroScale nach Frankreich geladen wurde, um das Spiel einem internationalen Publikum vorstellen zu können. Seitdem steht die Firma mit verschiedenen Publishern in Kontakt. Neben den positiven Reaktionen, die vor allem der Handhabung und Spielbarkeit galten, wurde von den meisten die Entwicklung für den Konsolenbereich begrüßt, aber die Konzentration auf die Xbox bemängelt. Da jedoch nur eine Person an der Engine programmiert, ist eine Anpassung der Prana-Engine an drei verschiedene Konsolensysteme nahezu unmöglich. Aus diesem Grund wird vom Programmierer derzeit Renderware-Graphics evaluiert, ein Render-Modul, das für alle gängigen Spielplattformen erhältlich ist. Damit wäre die Prana-Engine portabel und könnte auch auf den anderen Konsolen verwendet werden.

Eine Integration von Renderware hat auch einen großen Einfluss auf den Bereich der Level- und Character-Erstellung, da sie neue Features wie zum Beispiel Shadow Maps ermöglicht, dagegen andere Features nicht mehr unterstützt, beispielsweise Volume Shadows. Ob die bisherigen Messungen mit dem neuen Render-Modul die gleichen Ergebnisse liefern, ist ebenfalls fraglich. Interessant sind daher erneute Messungen der existierenden Testszenen.

Die Integration von RenderWare-Graphics bleibt zunächst jedoch noch offen und letztendlich auch eine Frage der Finanzierungsmöglichkeit, da dieses Modul mit Lizenzgebühren verbunden ist, die pro Konsolensystem gezahlt werden müssen.

Unabhängig davon sollen die Untersuchungen mit PBM weitergeführt werden, da sich immer wieder unterschiedliche Herangehensweisen für die Lösung einer Problematik ergeben. Außerdem eignet sich die bereits implementierte grafische Ausgabe, um zum Beispiel problematische Bereiche der Spielumgebung ausfindig zu machen, die zu hohen Performanzeinbußen führen. Dazu müsste über eine Kamerafahrt, die inzwischen möglich ist, zunächst die Spielumgebung „durchflogen“ werden, praktischerweise entlang einer Strecke, wie sie vom Spieler zurückgelegt werden könnte. Anschließend kann die grafische Darstellung der Entwicklung der Frame-Rate zeigen, zu welchen Zeitpunkten große Schwankungen im Level auftreten. Mit Hilfe weiterer Diagramme, beispielsweise über Polygonzahl, Anzahl der Bounding Boxen oder der für das Skinning benötigten Zeit, ließe sich dann nach den Schwankungsursachen suchen.

Doch zunächst konzentriert sich die Arbeit auf die Komplettierung der Demo-Version, dass heißt die Erstellung weiterer Levelgeometrien, Texturen und ähnliches. Darüber hinaus sind für die endgültige Version des Spiels sind insgesamt 7 Level geplant, die dem Anwender viele Interaktionsmöglichkeiten bieten sollen, unter anderem gegnerische Character. Außerdem sind mindestens 5 Hauptcharacter geplant, aus denen der Spieler auswählen kann.

Auch zukünftig wird es trotz steigender Leistung der Hardware nötig sein, bekannte sowie neue Optimierungstechniken zu nutzen, um ein Maximum an Qualität der Computerspiele zu erreichen und damit den stets wachsenden Ansprüchen der Spieler gerecht zu werden – die Herausforderung Spiel-Entwicklung wird immer eine solche bleiben.

[12] http://www.games-academy.de
[13] Eine internationale Fachmesse für junge Spielentwickler, deren Produkt noch unfinanziert ist. - http://www.milia.com

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Applikation Programmer Interface
CAD	Computer Aided Design
FPS	Frames Per Second – Bilder pro Sekunde
GPU	Graphics Processing Unit
HSR	Hidden Surface Removal
IK	Inverse Kinematik
KI	Künstliche Intelligenz
LOD	Level Of Detail
MTris	Million Triangles
PBM	Prana Bench Mark

Glossar

3D-API	Programmierschnittstelle, die →Renering für 3D Echtzeitanwendungen ermöglicht.
3D-Game-Engine	→3D-Spiel-Engine
3D-Objekt	Ein 3-dimensionales Objekt, dass Höhe, Breite und Tiefe besitzt und meist nur durch seine Oberfläche repräsentiert wird.
3D-Pipeline	→Grafik-Render-Pipeline
3D-Spiel-Engine	Eine →Spiel-Engine beziehungsweise ein Computerspiel, das zur Darstellung 3-dimensionale Grafiken benutzt.
3D-Szene	Enthält sämtliche Objekte und Informationen der 3 dimensionalen Welt, die notwendig sind ein Bild zu generieren.
Backface	Ein →Face, das vom Betrachter wegzeigt.
Back-Face Culling	Ein Optimierungsprozess, bei dem die →Back-Faces eines →3D-Objekts entfernt werden.
Billboard	Ein Objekt, dass so ausgerichtet werden kann, dass es stets zum Betrachter zeigt.
Bounding Box	Eine quaderförmige Begrenzungsgeometrie, die ein →3D-Objekt vollständig einschließt.
Character	Hier die Bezeichnung für sämtliche Gegenstände der Spielumgebung, die sich bewegen und/oder mit denen eine Interaktion möglich ist.
Clip	In diesem Fall ein aus dem Bereich der →Animation kommender Begriff. Gleichbedeutend mit Animationssequenz, d.h. ein Teil einer größeren Animation
DirectX	Eine →3D-API
Display List	Enthält sämtliche →3D-Objekte, die für das →Rendern eines Bildes zur Grafikkarte geleitet werden.
Dreieck	Kleinstmögliches →Polygon, stellt das Basisprimitiv in der 3D-Welt dar.
Engine	Ein (Software-) Programm oder Programmmodul.
Face / Facette	Ein Flächenelement eines polygonalen Objekts
Frame	Ein einzelnes Bild.

Frame-Buffer	Bildspeicher der Grafikkarte
Frame-Rate	Die Rate, mit der eine Render-Engine Bilder (Frames) generieren kann. Wird in Bildern pro Sekunde angegeben.
Füllrate	Die Geschwindigkeit mit der die Grafikkarte Pixel rendern kann, normalerweise in Pixel pro Sekunde angegeben
Game-Engine	→Spiel-Engine
Grafik-Render-Pipeline	Beschreibt den Weg, wie aus den Informationen einer → 3D-Szene ein zweidimensionales Bild berechnet wird, das anschließend auf einem Ausgabegerät wie Fernseher oder Monitor dargestellt wird.
Graphics Processing Unit	Moderner und sehr leistungsfähiger Grafikprozessor der den Großteil der Berechungen übernimmt
Instanz	Hier: ein Duplikat eines →3D-Objektes, für das nur Transformationsinformationen gespeichert werden.
Kachelbare Textur	Eine Grafik, deren Ränder nahtlos ineinanderübergehen, so dass sie sich mehrfach aneinandergereiht über ein Polygon legen lässt.
Konsole	Eine spezielles Computersystem, das in erster Linie für Computer- bzw. Videospiele konzipiert ist.
Level	Im Bereich der Computerspiele die Bezeichnung für die Spielwelt. Meist besteht diese jedoch aus mehreren Levels.
Level Of Detail	Ein Verfahren zur Anpassung der Detailgenauigkeit eines →3D-Objekts in Abhängigkeit der Entfernung zum Betrachter. Nur wenn sich der Betrachter nah am Objekt befindet, werden alle Details dargestellt.
Maya	Professionelle 3D-Software, die sowohl Modellierung, Animation als auch → Rendern ermöglicht. Wird bei der Produktion von Kinofilmen, Werbung und Echtzeitanwendungen eingesetzt.
Mesh	→Polygonnetz
MipMap	Eine →Textur, in der neben der Originalgröße, vorgefilterte Versionen in gröberen Auflösungen gespeichert sind, ähnlich →Level Of Detail
Modeller	Software zur →Modellierung
Modellierung	Erstellung eines 3D-Modells (Objekts) mit Hilfe einer Software. Dieses ist in verschiedenen sogenannten → Modellierungstechniken möglich.
Modellierungstechniken	Techniken zur Erzeugung 3 dimensionaler Objekte (→Polygone, →Splines, →Subdivison Surfaces

Modelling	→Modellierung
Multi-Material	Bezeichnung für die Möglichkeit mehrere Materialien beziehungsweise →Texturen für verschiedene Polygone eines →3D-Objekts benutzen zu können.
Multi-Texturing	Eine Technik, bei der mehrere Texturen auf ein Polygon gelegt werden.
NURBS	Non-uniform Rational B-Spline (nicht-gleichförmiger rationaler B-Spline), ein spezieller →Spline-Typ
Polygon	Ein Vieleck beziehungsweise eine geometrische Form die zur Oberflächenbeschreibung von 3D-Objekten verwendet wird.
Polygonnetz	Bezeichnung für die aus →Polygonen zusammengesetzte Oberfläche eines → 3D-Objektes.
Publisher	Ein Unternehmen, das die Entwicklung eines Computerspiels vorfinanziert und meist auch die anschließende Vermarktung übernimmt.
Render-Engine	Eine Software oder ein Softwaremodul zum →Rendern, auch Renderer genannt.
Rendering	→Rendern
Rendern	Prozess, bei dem aus den Informationen einer →3D-Szene ein zweidimensionales Bild erzeugt wird
Shape	In diesem Zusammenhang die äußere Gestalt bzw. Form eines Objektes.
Spiel-Engine	Hinter diesem Begriff verbirgt sich das „Herz“ eines Computerspiels, dass sämtliche Daten verarbeitet, die notwendig sind, um überhaupt spielen zu können.
Spline	→Modellierungstechnik, bei der Flächen und/oder Kurven, die auf mathematisch exakten Beschreibungen basieren, mit Hilfe von Kontrollpunkten „geformt“ werden.
Sprite	Textur die stets zum Betrachter zeigt, siehe → Billboard.
Subdivision Surface	Eine →Modellierungstechnik, die besonders zur Erzeugung organischer Formen geeignet ist.
Tesselierung	Prozess der Zerlegung eines →3D-Objekts in →Polygone, meist Dreiecke.
Texel	Das kleinste Element einer → Textur, leitet sich von „texture element“ ab.
Textur	Üblicherweise ein 2 dimensionales Bild, das zur Erzeugung von Oberflächenstrukturen auf →3D-Objekten eingesetzt wird.

Textur-Mapping	Prozess bei dem eine →Textur einem →3D-Objekt zugewiesen und daran ausgerichtet wird.
Tile	→kachelbare Textur
Triangle	→Dreieck
Vertex	Ein Punkt im Raum
Vertices	Mehrzahl von →Vertex
Whitepaper	Eine offizielle Richtlinie oder Vorgabe bezüglich eines Themas.

Literaturverzeichnis

[Abrash 01A] Michael Abrash, Advanced Technology Group, Microsoft
„About Xbox“
http://www.reactorcritical.de

[Abrash 01B] Michael Abrash, Advanced Technology Group, Microsoft
„Lessons learned from a year with Xbox“
Game Developers Conference 2001

[Cross 99] Jason Cross
„DXTn Texture Compression“
Computer Games Online
http://www.cdmag.com, 1999

[Daughtry 01] Lane Daughtry
„Maya Illuminated: Games“
Mesmer Inc., 2001

[Dietrich 99] Sim Dietrich, Nvidia Corporation
„Maximizing D3D Programming“
Game Developers Conference, 1999

[DirectX 00] DirectX 8.0
Offizielle Dokumentation, 2000

[Doug 01] Doug, Nvidia Corporation
„Nvidia DXT Compression Tool“
Hilfe zum DDS-Plugin für Photoshop, 2001

[Ertl 01] Thomas Ertl, Institut für Informatik der Universität Stuttgart
Visualisierung und interaktive Systeme (VIS)
Vorlesung „3D-Grafik Teil 2“
http://wwwvis.informatik.uni-stuttgart.de, 2001

[Gieselmann 02] Hartmut Gieselmann
„Die Konsolen greifen an“
c’t – Magazin für Computer und Technik, Heft 5/2002, S107-113]

[Heidrich 99] Wolfgang Heidrich
„High-quality Shading and Lighting for Hardware-accelerated Rendering“
Kapitel 6
Doktorarbeit
An der Technischen Fakultät Universität Erlangen-Nürnberg, 1999

[Himmelein 01] Gerald Himmelein
„So rendert Hollywood“
c’t – Magazin für Computer und Technik, Heft 11/2001, S.140-152

[Huddy 99] Richard Huddy, Nvidia Corporation
„High Performance D3D“
Game Developers Conference, 1999

[Immler 98] Christian Immler
„Das Große Buch - 3D Studiuo Max2“
DataBecker, 1998

[Luebke 00] David Luebke, University of Virginia
„Advanced Issues in Level of Detail“
Course 41, SIGGRAPH 2000

[Microsoft 01] Microsoft Corporation
„Microsoft DirectX 8 Developer FAQ“, 2001

[Möller 99] Tomas Möller; Eric Haines
„Real-Time Rendering“
A.K. Peters Ltd., 1999

[Rogers 00] Douglas H. Rogers
„Optimizing Direct3D for the GeForce 256“
Whitepaper von Nvidia, 2000

[Saltzmann 00] Marc Saltzmann
„Game Design - zweite Ausgabe“
X-Games, 2000

[Tönnies 94] Dr. Klaus D. Tönnies; Prof. Dr. Heinz U. Lembke
„3D-Computergrafische Darstellung“
R. Oldenbourg Verlag, 1994

[Watt 89] Alan Watt
„Fundamentals of three-dimensional computer graphics“
Addison-Wesley, 1989

[Weiskopf 01] Daniel Weiskopf
„Modellierung und Animation„
http://wwwvis .informatik.uni -stuttgart.de/ger/teaching/lecture/ws01/modanim,2001

Messwerte

1 Versuch „Fahrzeug- Stoßstange“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:		
	Szene 1	Szene 2
Durchlauf 1	251,39	264,81
Durchlauf 2	250,95	264,32
Durchlauf 3	250,82	264,43
Durchlauf 4	250,75	264,34
Durchlauf 5	250,92	264,62
Durchschnittswert	250,97	264,5

Tab. V-1 Messreihe Versuch „Fahrzeug-Stoßstange“

2 Versuch „Fahrzeug-Reifen“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:		
	Szene 1	Szene 2
Durchlauf 1	170,63	244,56
Durchlauf 2	171,29	244,12
Durchlauf 3	171,12	244,59
Durchlauf 4	170,33	245,20
Durchlauf 5	171,09	244,86
Durchschnittswert	170,89	244,67

Tab. V-2 Messreihe Versuch „Fahrzeug-Reifen“

3 Versuch „Fahrzeug-Modellierung“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:		
	Zusammengesetzt	Ein Objekt
Durchlauf 1	237,53	203,27
Durchlauf 2	238,17	203,19
Durchlauf 3	237,29	203,46
Durchlauf 4	237,83	203,59
Durchlauf 5	237,29	203,83
Durchschnittswert	237,62	203,47

Tab. V-3 Messreihe Versuch „Fahrzeug-Modellierung“

4 Versuch „Fahrzeug-Modellierung B“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:		
	Zusammengesetzt	Combined
Durchlauf 1	238,43	252,31
Durchlauf 2	238,17	252,87
Durchlauf 3	237,34	253,12
Durchlauf 4	237,31	252,46
Durchlauf 5	237,98	252,21
Durchschnittswert	237,85	252,59

Tab. V-4 Messreihe Versuch „Fahrzeug-Modellierung B“

5 Versuch „Bounding Box“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:		
	Brücke Combined	Brücke Separated
Kamera 1	68,47	71,54
Kamera 2	66,28	70,69

Tab. V-5 Messwerte Versuch „Bounding Box“

6 Versuch „Textur-Dimension“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:			
	Kamera 1	Kamera 2	Kamera 3
196 Texturen (2X x 2Y)	77,53	73,95	45,31
196 Texturen (2X + 1 x 2Y+1)	77,21	73,44	45,67

Belegter Texturspeicher	
196 Texturen (2X x 2Y)	23,9 MB
196 Texturen (2X + 1 x 2Y+1)	5,9 MB

Tab. V-6 Messwerte Versuch “Textur-Dimension”

7 Versuch „Textur-Dimension B“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:			
	Kamera 1	Kamera 2	Kamera 3
Texturen mit MipMaps	76,96	73,73	45,03
Texturen ohne MipMaps	17,86	21,22	12,90

Tab. V-7 Messwerte Versuch “Textur-Dimension B”

8 Versuch „zerstückelte Objekttextur“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:		
	10 Texturen	1 Textur
Durchlauf 1	176,34	251,78

Tab. V-8 Messwerte Versuch „zerstückelte Objekttextur“

9 Versuch „Reklameschilder“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:		
	Viele Grafiken	Eine Grafik
Durchlauf 1	85,78	85,03

Tab. V-9 Messwerte Versuch „Reklameschilder“

10 Versuch „Echtzeitbeleuchtung vs. Prelighting“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:			
Szene (Demolevel)	Kamera 1	Kamera 2	Kamera 3
Prelighted	89,91	87,76	121,84
Ambient	89,51	87,20	121,32
1 Directional	84,35	82,68	117,09
2 Directional	82,36	80,38	115,55
3 Directional	80,75	79,92	113,81

prozentualer Geschwindigkeitsverlust im Vergleich zur **vorbeleuchteten** Szene:			
Ambient	0,44	0,64	0,43
1 Directional	6,18	5,79	3,90
2 Directional	8,40	8,41	5,16
3 Directional	10,19	8,93	6,59

Durchschnittliche Geschwindigkeitsverminderung in %:	
Ambient	0,5
1 Directional	5,29
2 Directional	7,32
3 Directional	8,57

Tab. V-10 Messwerte Versuch “Echtzeizbeleuchtung vs. Prelighting”

11 Versuch „Schlagschatten-Bahnbrücke“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:		
	Kamera 1	Kamera 2
Volumen Schatten	79,28	83,70
Vertex Colors	90,21	94,49

Tab. V-11 Messwerte Versuch “Schlagschatten-Brücke”

12 Versuch „Gliederpuppe vs. Skinning“

Angabe der durchschnittlichen Render-Geschwindigkeit in Bilder Pro Sekunde:			
	Gliederpuppe	Rigid Binding	Smooth Binding
Durchlauf 1	51,10	58,79	49,14

Angabe der durchschnittlichen Zeit für das Skinning, in ms:			
	0	2,60	6,12

Angabe der durchschnittlichen Zeit für die Verarbeitung der Display List, in ms:			
	15,57	10,41	10,23

Tab. V-12 Messwerte Versuch “Gliederpuppe vs. Skinning ”

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, 04. Mai 2002

- Stefan Schubert -