Im Auge des Drachen

Von
Philipp Muigg
Andreas Stamminger
Für CG23 Lab SS2004

Philipp Muigg 0125958
Andreas Stamminger 0125381

Das Spiel

Hintergrund

Das Spiel „Im Auge des Drachen“ handelt von einem der letzen überlebenden Drachen, welcher versucht in einer Welt, die von goldgierigen und brutalen Felsenzwergen überrannt wird, zu überleben. Und so muss er sich gegen ständig anstürmende Horden verteidigen und versuchen, sofern dies möglich ist, die Wurzel allen Übels, nämlich die Dörfer der Zwerge, zu zerstören.

Spielmechanik und Spielziel

Die Aufgabe des Spielers ist es seine Burg zu beschützen (bzw seinen Goldvorrat nie unter 0 sinken zu lassen) und alle Zwerge (und ihre Häuser) zu zerstören. Diese versuchen ihrerseits durch ständige Angriffe an das Gold der heimischen Burg heranzukommen und den Spieler zu töten, falls dieser in Schussreichweite kommt. Wird ein Zwerg getötet wird er durch eine starke Magie in seinem Haus wieder zum Leben erweckt. Dort schläft er eine kurze Zeit um sich danach gleich wieder auf den Weg zur Burg des Spielers zu machen. Deshalb kann ein Zwerg nur dann dauerhaft getötet werden, wenn sein Haus zerstört wurde. Weiters baut er sein Heim wieder auf, falls er von seinem Streifzug zurückkehrt und dieses zerstört vorfindet.
Falls ein Zwerg die heimische Burg angegriffen und Gold geraubt hat trägt er dieses zurück zu seinem Haus, wo er es hortet. Wird er auf dem Heimweg getötet hinterlässt er selbstverständlich das geraubte Gold in form einer gelben Kugel. Ebenso bleibt eine solche Kugel mit dem entsprechenden Wert zurück, falls das Zwergenhaus zerstört wird. Um nun diese Goldkugeln wieder zurückzuerobern hat der Spieler zwei Möglichkeiten. Entweder kann er selbst das Gold aufsammeln (indem er es einfach vom Boden greift. Die Anzahl der Goldkugeln die getragen werden können wird durch deren Wert bestimmt, welcher sich in ihrem Gewicht ausdrückt.) und es zur Burg bringen, wo er es über dieser fallen lassen muss, oder er kann sich einen Ballon kaufen, welcher von befreundeten Gnomen gelenkt wird, und automatisch nach Gold auf der Insel sucht und dieses zurück zur Burg bringt (wie ein solcher Ballon gekauft werden kann wird im Abschnitt Menüs beschrieben).
Weil der Spieler beim Kampf gegen die Zwerge immer mehr Erfahrung sammelt gewinnt dieser von Zeit zu Zeit neue Erfahrungspunkte (dies wird durch eine Nachricht angezeigt, die kurz eingeblendet wird), welche er in vier Verschiedene Kategorien verteilen kann (mehr hierzu auch im Abschnitt Menüs).
Diese Vier sind:

- Maximale Lebenspunkte
  Dieser Wert bestimmt wie viele Lebenspunkte der Spieler maximal besitzt.
- Maximale Last
  Mehr Last als dieser Wert kann nicht mitgeführt werden.
- Feuerkraft
  Dieser Wert bestimmt die Geschwindigkeit mit der Feuerbälle geschossen werden können und wie viel Schaden diese anrichten.
- Geschwindigkeit
  Die Geschwindigkeit mit der sich der Drache bewegen kann.

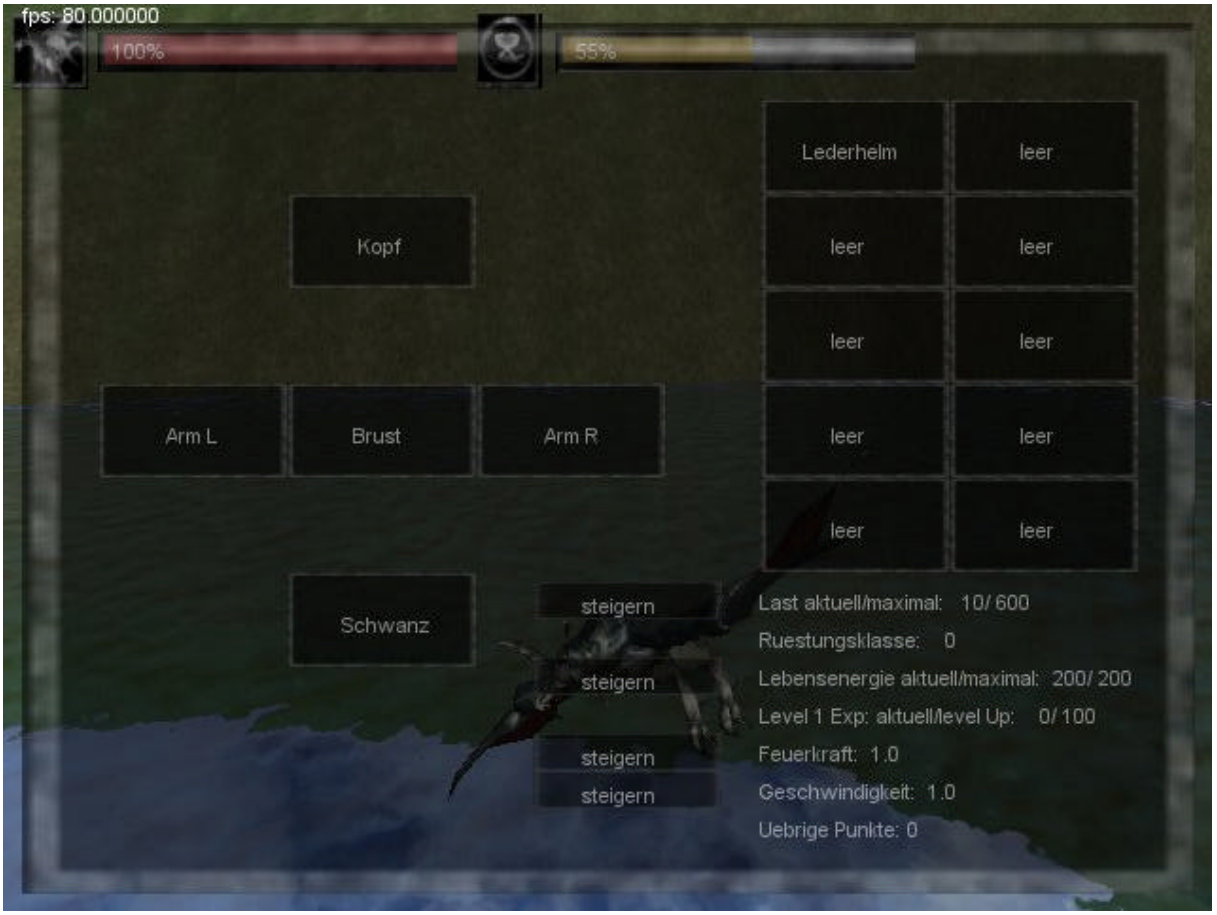
Philipp Muigg 0125958
Andreas Stamminger 0125381

Menüs

Burgmenü

Dieses Menü kann über die Taste „m“ erreicht werden, sobald sich der Spieler in der Nähe seiner Burg befindet. Auf der Linken Seite werden Gegenstände aufgelistet, die gegen Gold erworben werden können (der aktuelle Goldstand wird unter dieser Liste angezeigt.) Um die Kosten eines Gegenstandes zu sehen muss einfach auf diesen geclicked werden. Dann scheint dessen Name und der Preis unterhalb des aktuellen Goldstandes auf. Um den Gegenstand zu kaufen muss doppelt auf diesen geclicked werden. Hiernach scheint er in der Ausrüstungsliste auf der rechten Seite des Bildschirmes auf. Um ihn wieder loszuwerden einfach in dieser Liste auf ihn doppelt Klicken. Unter der Angebotsliste befinden sich zwei weitere Knöpfe. Über den einen kann der Spieler einen der zuvor erwähnten Goldsammelballone kaufen und der andere Heilt den Drachen des Spielers wieder vollständig. In der rechten unteren Ecke werden darüber hinaus die aktuellen Eigenschaften des Drachen angezeigt.

Ausrüstungsmenü



Dieses Menü kann über die Taste „e“ zu jeder Zeit im Spiel erreicht werden. Die Rechte Bildschirmseite ist äquivalent zur Rechten Seite des Burgmenüs. Allerdings wird ein Gegenstand nicht zerstört, falls doppelt auf ihn geclicked wird, sondern er wird benutzt (zur Zeit gibt es nur verschiedenste Rüstungsteile welche durch das Benutzen angezogen werden. Sie erhöhen die Rüstungsklasse des Drachen, was dazu führt, dass jeglicher Schaden der dem Spieler zugefügt wird reduziert wird). Auf der linken Seite sind die verschiedenen Rüstungsslots zu sehen, in denen die jeweiligen Rüstungsteile erscheinen, falls sie angezogen werden. Um diese Slots wieder zu leeren muss doppelt auf sie geclicked werden. Die Vier Knöpfe mit der Aufschrift steigern werden verwendet um, bei einem Levelaufstieg, erhaltene Punkte auf die jeweilig rechts daneben stehenden Eigenschaften zu verteilen (wie viele Punkte noch zu verteilen sind wird in der letzten Zeile angezeigt).

Minimap

Die Minimap zeigt die gesamte Insel von Oben. Der Spieler wird als blauer Pfeil dargestellt (welcher sich in Blickrichtung ausrichtet). Die weißen Punkte sind feindliche Gebäude (entweder Zwergenhäuser oder Verteidigungstürme). Die Roten Markierungen sind die Zwerge selbst. Rosa werden die Goldkugeln dargestellt, welche eingesammelt werden können. Grün sind die Markierungen der eigenen Goldsammelboallone.

Tastenbelegung und Steuerung

?	Vorwärts fliegen und aufsteigen
?	Rückwärts fliegen

?	Zur linken Seite fliegen
?	Zur rechten Seite fliegen
„linke Maustaste“	Feuerball schießen (Maustaste gedrückt halten für Dauerfeuer)
„space“	Gold fallenlassen.
„p“	Spiel Pausieren bzw Fortsetzen
„m“	Burgmenü, falls der Spieler sich in der Nähe der Burg befindet
„e“	Ausrüstungsmenü
“esc”	Spiel beenden
“c”	Cheattaste. Diese füllt sowohl Lebensenergie als auch Goldvorrat vollständig auf. Weiters verschafft sie dem Spieler pro Tastendruck 10 zusätzliche zu Verteilende Punkte.
„i“	Maus invertieren bzw nicht invertieren.

Tipps

Zur Steuerung

Da es keine Munitionsbeschränkung gibt immer mit Dauerfeuer drauflos ☺.
Um nicht von den Zwergen getroffen zu werden versuchen ständig in Bewegung zu bleiben. Dies kann am besten durch ständiges zur Seite und Rückwärtsfliegen erreicht werden. Um ein Dorf zu vernichten deshalb versuchen von schräg oben hinabzufeuern und durch rückwärts fliegen die Höhe und Entfernung halten. Dabei wie erwähnt immer versuchen durch seitliches Ausweichen den Geschossen der Zwerge zu entkommen.

Zur Strategie

Da es in unmittelbarer Nähe der Drachenburg ein Zwergendorf steht muss man sich als erstes um dieses Kümmern. Insofern sofort dorthin fliegen und versuchen alle Häuser zu zerstören. Die Zwerge, die währenddessen angreifen versuchen nebenher zu dezimieren. Das Hauptaugenmerk sollte allerdings auf jeden fall sein die Häuser zu zerstören, da die getöteten Zwerge sonst bereits nach wenigen Sekunden wieder aufstehen. Falls alle Häuser dem Erdboden gleich gemacht worden sind versuchen die dazugehörigen Zwerge endgültig über den Jordan zu schicken.
Wenn das Dorf zerstört ist kommen vermutlich bereits Zwerge von anderen etwas weiter entfernten Dörfern angelaufen. Diese müssen nun so schnell wie möglich abgeschossen werden um danach schnell ein weiteres Zwergendorf zu vernichten (am besten das, das am nächsten zur Drachenburg steht).
Einige Dörfer bestehen nicht nur aus einfachen Zwergenhäusern, sondern auch aus Wachtürmen, welche den Spieler mit Geschossen beschießen, welche diesem folgen. Allerdings ist es möglich diesen entweder auszuweichen, oder sie mit einem Feuerball abzuschießen. Natürlich ist es auch möglich die Wachtürme zu zerstören.
Ein weiterer strategisch günstiger Zug ist es bereits früh einen oder zwei Ballone zu kaufen, die das Gold, das in den Zwergenhütten lagert zur Burg bringen, damit man selbst die Burg verteidigen kann.

Zur Punkteverteilung

Die durch Levelaufstieg gewonnenen Punkten sollten am besten am Anfang in Feuerkraft und Geschwindigkeit investiert werden.

Philipp Muigg 0125958
Andreas Stamminger 0125381

Die Grafik

Kameramodell

Für die Engine wurde eine Kameraklasse (TKamera in den Dateien TKamera.h und TKamera.cpp) implementiert, welche über einen Up-Vector einen View-Direction-Vector einen Fokus Punkt, der Entfernung der Projektionsebene und des Projection Reference Points zu diesem, konfiguriert wird. Aus diesen Parametern wird eine Projektions und eine Modelview Matrix generiert welche, sobald die Kamera gesetzt wird, geladen werden.

Beleuchtung

Die einzelnen Objekte, die im Spiel zu sehen sind, werden über unterschiedliche Wege und Techniken beleuchtet auf die hier nun genauer eingegangen wird.

Terrain

Das Terrain wird über eine einfache Lightmap-Textur (welche von uns generiert wurde) mit Schatten versehen indem die Farbe des Terrains einfach mit dieser Moduliert wird.

Einfache Objekte

Einfache, nicht animierte, Objekte (wie die Zwergenhäuser oder die Burg) werden über ein einfaches Vertex-Program diffus beleuchtet. Das hierfür eingesetzte Programm ist in der Datei „programs\standard_vp.txt“ zu finden.

Animierte Objekte

Die einzigen Objekte dieser Kategorie sind zurzeit die Zwerge. Sie werden auch über ein Vertex-Program diffus beleuchtet. Allerdings ist anzumerken, dass dies hier in Kombination mit Skinning gemacht wird. Nachdem für einen bestimmten Vertex eine Matrix aus 4 weiteren Matrizen zusammengeblendet wurde, (die Art des Blendings wird über die dem Vertex zugeordneten Gewichte und die Matrixindizes bestimmt) wird diese nicht nur zur Transformation der Vertexposition, sonder auch zur Transformation des zugehörigen Normalvektors eingesetzt (hier wird darauf verzichtet die inverse transponierte zu berechnen, da die Matrix nicht ununiform skalieren sollte). Danach kann einfach, nachdem noch das Ergebnis mit der Modelviewmatrix multipliziert wurde, das Vektorielle Produkt mit dem in Eye-Space vorliegenden Lichtrichtungsvektor gebildet werden, was dann schließlich als Diffuse Lichtintensität eingesetzt werden kann. Das Vertex-Program befindet sich in der Datei „programs\noskinning_vp.txt“

Drachen

Der Spielerdrache Wird über Normalmapping beleuchtet, welches im Spezialeffekte-Abschnitt genauer beschrieben wird.

Partikeleffekte

Die Partikeleffekte werden alle über eine gemeinsame Klasse (TGlParticleEngine in den Dateien TGlParticleEngine.h und TGlParticleEngine.cpp definiert) koordiniert und realisiert. Jedes einzelne Partikel kann sich selbstständig bewegen, seine Größe und Transparenz ändern

und noch einige andere Parameter beeinflussen. Die Grundlegende Partikelklasse ist TFireParticle (in den Dateien TFireParticle.h und TFireParticle.cpp definiert). Diese enthält auch den Code zur Darstellung. Hierzu verwendet sie ein Vertex-Program („programs\particle_vp.txt“) welches die Position und einige andere Parameter (die z-Texturkoordinate, die Größe die Helligkeit und die Transparenz) als Program-Environment-Parameter erhält. Weiters wird eine einmal generierte Displaylist, welche ein einziges fixes Quad enthält eingesetzt. Die Vertizes dieses Quads werden dann im Vertex-Program mit der übergebenen Partikelgröße multipliziert und als offset in eyeSpace dazuaddiert. Hierdurch ist das Partikel immer dem Betrachter zugewandt. Danach wird noch die z-Texturkoordinate durch die im Program-Parameter übergebene ersetzt (Dies ist wichtig für die 3D-Texturen, die für die Texturanimation eingesetzt werden. Hierzu mehr im Abschnitt Spezialeffekte).

View Fustrum Culling

Um möglichst wenig unsichtbare Geometrie rendern zu müssen wird einfaches View Fustrum Culling eingesetzt. Allerdings werden nicht alle Spielobjekte gegen das Fustrum auf Sichtbarkeit getestet, da dies den Prozessor viel zu stark beanspruchen würde und somit den Geschwindigkeitsgewinn des Algorithmus zunichte machen würde. Stattdessen sind alle Spielobjekte dem Teil des Terrains zugeordnet über dem sie sich befinden. Nun werden alle Terrainteile gegen das Fustrum auf Sichtbarkeit getestet. Falls sie sichtbar sind werden sie gemeinsam mit ihren Spielobjekten gezeichnet.

Geometriedaten

Die Geometrie wird hauptsächlich in Vertex Buffer Objects abgelegt. Nur die Statischen Objekte (zur Zeit nur die Bäume) werden in Immediate Mode in Displaylists gerendert. Werden über die Taste F6 die VBOs deaktiviert wird alles über einfache Vertex Arrays gerendert. Falls mit der F7-Taste die Displaylists deaktiviert werden, werden alle statischen Objekte in immediate mode gerendert.

Die Spezialeffekte

Da Einige Effekte auf Fragment Programs aufbauen und die ARB Version dieser Extension auf nv-Hardware ein wenig langsamer läuft als die NV Version wurden für diese Effekte zwei Fragmentprogram Versionen geschrieben (es gibt zwei Ausführbare Binaries die entweder auf die ARB Extension oder auf die NV Extension aufbauen). Weiters gibt es natürlich auch immer zwei Textdateien, die die jeweiligen Programme enthalten, wobei folgende Namenskonvention gilt: Alle ARB Programme enden auf „.txt“ während alle NV Programme denselben Namen wie die ARB Entsprechung haben, allerdings auf „.txt.nv“ enden.

Philipp Muigg 0125958
Andreas Stamminger 0125381

Matrix Palette Skinning (fast ausschließlich auf der GPU mit bis zu 17 Matrizen auf nv bzw 32 Matrizen auf ati hw)

Skinning ist eine Methode um bei Animierten Modellen nicht jeden Vertex extra zu bewegen. Stattdessen wird zuerst ein Skelett für das zu animierende Objekt definiert. Danach werden jedem Vertex die Knochen dieses Skelettes über eine Gewichtungsfunktion zugewiesen, wobei die Summe der Gewichte für alle Knochen für jeden Vertex gleich eins sein muss (ausserdem müssen die Einzelgewichte im Intervall [0,1] sein). Um das Modell zu animieren muss nun nur mehr das Skelett animiert werden, welches eben über seine Knochen und die jeweiligen Gewichte die Position der Vertizes verändern kann.
In unserer Implementierung kann jeder Vertex von bis zu vier Knochen beeinflusst werden. Jeder Knochen besitzt eine Matrix, welche die relative Änderung der Position und Rotation bezüglich des Ruhezustandes des Knochen beschreibt. Werden diese nun für einen Vertex über dessen Gewichte aufsummiert erhält man die durch die Bewegung der Knochen implizierte neue Vertexposition. Bis auf das Interpolieren der zuvor erwähnten Knochenmatrizen geschieht der restliche Vorgang von Interpolation der Matrizen bis hin zur finalen Transformation des Vertex in dem Vertex Program „programs\noskinning_vp.txt“. Der Softwareteil ist in den Klassen TGlSkeletton und TGlSwSkinningModel implementiert.
Hier noch ein Link zu einer Präsentation in der Skinning ein wenig genauer erläutert wird (zum unterschied zu unserer Implementierung werden hier allerdings auch noch Tangenten Vektoren mitgeskinned):
http://developer.nvidia.com/object/skinning.html

Statisches Level of Detail (für das Terrain)

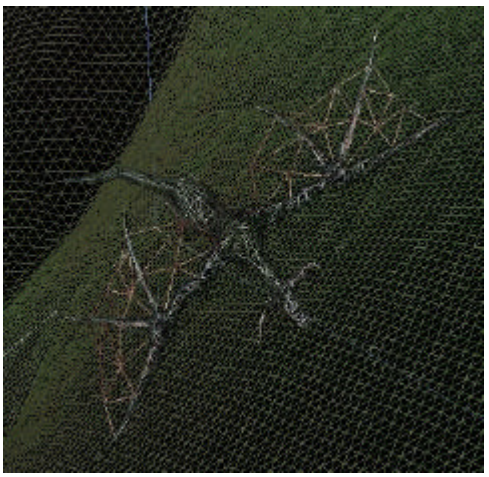
Um nicht alle ca. 8.300.000 Polygone (2048*2048 Vertizes) des Terrains zeichnen zu müssen wird dieses in 16 mal 16 Zellen aufgeteilt. Für diese werden dann Indexarrays angelegt, welche jeweils ein Viertel der Polygonanzahl des Vorherigen besitzten. Weiters werden auch Indexarrays für die Verbindungsstücke zwischen den Detailstufen angelegt. Bei unserer Implementierung können allerdings die Detailstufen benachbarter Zellen nicht um mehr als 1 unterschiedlich sein, da hierfür keine Übergänge generiert werden. Dies erfordert eine gewisse Sorgfalt bei der Auswahl des passenden Detaillevels (und in unserem Fall eine kleine Nachkorrektur um fehlerhafte Übergänge auszubessern). Die grundlegende Auswahl des Detaillevels geschieht zugegebenermaßen sehr einfach über die Entfernung zum Betrachter. Die Klassen, die Für die Darstellung des Terrains verantwortlich sind, sind TTerrain und TTerrainCell.

Bodentexturen (5 verschiedene Texturen in Fragment Program kombiniert)

Für das Terrain werden in einem simplen Vertex Program („programs\floor_vp_fp.txt“) zuerst Texturkoordinaten generiert (auch nur ein Texturkoordinatenset fix zu speichern hätte die Menge an Geometriedaten auf über 100MB in die höhe schnellen lassen, was selbst auf

heutigen Karten viel zu viel ist). Diese werden dann im Fragment Program „floor_fp.txt“ eingesetzt. Hier werden zunächst drei Lookups in Detailtexturen (d1..d3) und einer in eine vierte Textur (i) gemacht, welche die Anteile der drei Detailtexturen an der endgültigen Fragmentfarbe bestimmt. Das Zusammenblenden wird dann wie folgt vorgenommen: color=d1*i.r+d2*i.g+d3*i.b. Hierbei ist natürlich darauf zu achten, dass i.r+i.g+i.b=1 ist. Nachdem nun die Fragment Farbe bestimmt ist wird ein letzter Textur Lookup in eine Lightmap gemacht und die Farbe mit dieser Multipliziert.

Normalmapping (Charakter mit 2000 Polygonen für Beleuchtung äquivalent zu 34000 Polygonen)



Normalmapping ist eine Technik um Modelle mit niedriger Polygonanzahl für die Per-Pixel Beleuchtung wie sehr hochpolygonale Modelle aussehen zu lassen. Dies wird dadurch erreicht, dass die Normalvektoren des detailierten Modells in Farben kodiert und so in eine Textur gerendert werden (natürlich müssen Texturkoordinaten generiert werden, die ein gutes Mapping von allen Faces des detailierten Modells auf die Normalmap zulassen). Dieses Generieren der Normalmaps hat für uns das NVidia Tool Melody übernommen (http://developer.nvidia.com/object/melody_home.html). In unserer Implementierung werden die Normalen als Object Space Normals in die Normalmap gerendert, wodurch man sich jeglichen Zusatzaufwand, der durch die Verwendung von Tangentspace Normals entstehen würde (wir benötigen also keine Tangenten oder Binormals ☺) sparen kann. Da allerdings unser Modell in einem Vertexprogram geskinned wird (in diesem Fall das dem normalen Skinning sehr ähnliche Program in der Datei „programs\noskinning_vp_fp.txt“) müssen die Object Normals der Normalmap natürlich noch über eine interpolierte Bonematrix verändert werden. Diese Bonematrix wird im zuvor erwähnten Vertex Program ja bereits eingesetzt um die Vertexpositionen zu verändern und kann einfach in Form von drei Texturkoordinaten an das Fragment Program „programs\normalmap_fp.txt“ übergeben werden. Nachdem nun die Object Normals über diese Matrix zu den der Animation entsprechenden Object Normals transformiert worden sind, wird die Beleuchtung in Object Space durchgeführt (Dies erspart die Transformation des Normalvektors in Eye Space). Object Space Halfvector und Object Space View Direction werden

auf der CPU berechnet und dem Fragment Program als Parameter übergeben (wobei die View Direction in unserer Implementierung der Einfachheit halber immer die Blickrichtung ist).

Explosionen (3D Texturen für Texturanimation)

Die Partikel, die bei Explosionen zum Einsatz kommen sind mit einer 3D-Textur belegt, wobei die einzelnen Texturschichten die jeweiligen Animationsphase darstellen. Der Vorteil dieser Technik ist, dass der Übergang zwischen den einzelnen Texturen auch ohne Fragment Program (oder Env Combiners) sehr weich ist (falls die 3D-Textur Trilinear gefiltert wird).

Reflektierendes/Verzerrendes Wasser (PBuffer, projektive Texturierung und Fragmentprogram)

Der Wassereffekt wird durch die Kombination einiger Techniken erzielt. Grundlegend für alle ist, dass zuerst drei Lookups in Texturen mit zufällig orientierten Normalvektoren (allerdings alle mit positiver z-Richtung) gemacht werden. Hier entsteht auch die Wasseranimation, da die Lookups an Texturkoordinaten stattfinden, die ständig durch die CPU über die Texture Matrizen verändert werden. Die Ergebnisse dieser Lookups werden dann unterschiedlich gewichtet aufaddiert und Normalisiert (eine Normalisation Cubemap ist hierfür genau genug). Basierend auf diesem Ergebnis wird die Beleuchtung des Wassers durchgeführt. Die Reflexion wird dadurch erzielt, dass die Szene (mit reduziertem Detail) in einen PBuffer gerendert wird, welcher danach als Textur gebunden und auf den Wasserpolygon (aus Sichtrichtung) projiziert wird. Um nun die Reflektion an der rauen Wasseroberfläche zu simulieren wird die Texturkoordinate, die durch die Projektion auf den Wasserpolygon generiert wurde durch den Normalvektor verschoben. Dies ist zwar physikalisch hochgradig

inkorrekt, sieht aber plausibel aus (und darauf kommts in der CG ja an ☺). All dies geschieht im Fragment Program „programs\wasser_fp.txt“.

Sonstiges

Alle Modelle wurden Mit 3DSMax erstellt (bzw verändert) und mit Hilfe eines selbst geschriebenen Plugins exportiert. Die Heightmap für das Terrain wurde mit einem eigens dafür geschriebenen Editor generiert welcher unter anderem auch midpoint displacement Fraktale beherrscht. Das einzige Tutorial das verwendet wurde war NeHes Tutorial zu Bitmap Fonts (http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=13), welches so erweitert wurde, dass statt der Bitmaps, Texturen erzeugt werden die dann auf quads gemapped werden.
Zur Umsetzung des Sounds wurde die Soundlibrary fMod eingesetzt.
Wie bereits erwähnt wurden die Normalmaps mithilfe von NVidias Melody erstellt.