

**Praktikum Design und Entwicklung eines Computerspiels**

**OpenGL Odyssee  
oder: Wie man eine Terrain-Engine verbricht**

**Festini-Mira Alexander  
alexander.festini@web.de**

**19. März 2003**

## Zusammenfassung

Dieser Bericht enthält eine kurze Übersicht über verschiedene Methoden zur Landschaftsdarstellung, sowie eine Beschreibung der gewählten Methode und einer kurzen Dokumentation zum letztlich entstandenen Modul. Dabei sollen auch die Probleme und deren mehr oder weniger gelungenen Lösungen erläutert werden, sowie immer noch vorhandene Probleme und Anregungen zu deren Beseitigung.

Da auch nach dem Praktikum weiter an der Terrain-Engine gearbeitet wurde, ist die Version im Projekt nicht auf dem neuesten Stand und viele Änderungen und Verbesserungen sind hier zwar erwähnt, aber nicht implementiert. Diese dienen lediglich als Anhaltspunkt für spätere Praktika, die evtl. Änderungen an der Landschaft vornehmen wollen oder müssen.

## 1. Einführung

Es soll also ein Modul entstehen, welches aus wie auch immer gearteten Daten eine Landschaft erzeugt. Dabei stellen sich grundsätzliche Fragen, die Einfluß auf den weiteren Verlauf haben. Genügt eine vollkommen zweidimensionale Landschaft, oder soll es auch Überhänge, Brücken und Höhlen geben? Wie groß wird die Sichtweite und soll die Landschaft unauffällig im Hintergrund nachladen oder gibt es feste Grenzen, an denen geladen wird?

Gerade das Nachladen bei großen Landschaften ist bisher selten befriedigend gelöst worden. Kurze Unterbrechungen oder starkes Ruckeln bei ständigen Festplattenzugriffen sind in fast jedem Spiel bemerkbar und nur wenige versuchen ohne feste Ladegrenzen auszukommen.

Üblicher ist eine zweidimensionale Landschaft, in der Besonderheiten wie Brücken und Überhänge durch zusätzliche Objekte realisiert werden. Bisher ebenfalls üblich sind leider auch kantige Landschaften, die mit wenigen Polygonen auskommen müssen.

Diese beiden Schwachpunkte sollten hier besser gelöst werden. Das Nachladen soll den Spielfluß nicht beeinflussen und es soll die Leistungsfähigkeiten moderner Karten genutzt werden, um glattere Landschaften dazustellen.

## 2. State of the Art

Anfangs begann die Suche nach verschiedenen Methoden, um davon eine angemessene als Ausgangspunkt zu nutzen.

Ein immer noch überall anzutreffendes Artefakt aus der Prä-3D-Beschleunigungs-Ära ist der ROAM (Realtime Optimally Adapting Mesh) Algorithmus. Dieser entscheidet für jeden Frame und jeden Punkt, ob dieser nötig ist, oder ob der sichtbare Fehler, der durch das Weglassen entstehen würde, unterhalb eines Grenzwertes liegt. Hierbei gibt es ein paar zusätzliche Regeln, um aus den verbliebenen Punkten das Landschafts-Mesh zu erzeugen. Was damals allerdings noch sinnvoll war, ist heute reine Rechenzeitverschwendung am falschen Ende.

Besonders deutlich wird dies bei Unreal Tournament 2003. Hier wird die Landschaft lediglich in kleine Stück aufgeteilt und mittels Frustum Culling auf die sichtbaren beschränkt. Die Leistungsreserven einer aktuellen Grafikkarte machen LoD und andere Entlastungen des 3D-Chips unnötig und oft kontraproduktiv. Dabei sollte allerdings beachtet werden, daß in diesem Spiel relativ kleine Level mit einer bescheidenen Heightmap von 256x256 zum Einsatz kommen.

Das Beispiel führt auch schon zu den heute gängigeren Methoden. Statt die Landschaft in jedem Frame neu zu erstellen wird sie in einzelne Chunks aufgeteilt. Gibt es diese in verschiedene Detailgraden, so nennt sich die Methode GeoMipMapping, wegen ihrer Ähnlichkeit zu MipMapping bei Texturen. Welcher Detailgrad für einen Chunk gewählt wird, ist dann von beliebigen verschiedenen Faktoren abhängig, wie z.B. der Entfernung zur Kamera oder der Abweichung zum höchsten Detailgrad.

### 3. Related Work

Den Maßstab setzte für diesen Teil des Praktikums das IScape-Projekt (<http://www.futurenation.net/gibase/projects.htm>). Während der Entstehung wurde auch deren Heightmap verwendet, um einen direkten Vergleich zu haben. Diese beinhaltet zwar einige Features mehr, beschreitet aber insgesamt den gleichen Weg, mit kleinen Unterschieden. Vorberechnete Sichtbarkeitsinformationen sorgen zwar für schnelles und genaues Occlusion Culling, allerdings sind über 5MB für einen relativ kleinen Ausschnitt der Karte für ein Spiel mit großer Spielwelt nicht machbar. Dennoch ist es eine gute Darstellung der gesteckten Ziele.

### 4. Das Konzept

Das Ziel ist also eine nahtlose Welt mit hohem Detailgrad und wenig auffälligem LoD. Dazu sollte die Arbeit soweit wie möglich von der CPU auf die GPU verlagert werden, um den anderen Komponenten möglichst wenig Rechenzeit zu stehen.

Der Aufbau der Landschaft gliedert sich in mehrere Ebenen, mit dem Terrain an oberster Stelle. Dieses beinhaltet die Vertex- und IndexBuffer für die gesamte Geometrie. Das Terrain ist aufgeteilt in mehrere Sektoren, die wiederum ihre Heightmap, Lightmap, etc. beinhalten. Diese bestehen dann aus vielen Patches, die jeweils einen Chunk darstellen. Außerdem kennt jeder Patch seinen Nachbarn.

Beim Laden wird ausreichend Speicher reserviert und aus der Heightmap der VertexBuffer erstellt, sowie Texturgewichte festgelegt. Dabei füllen die Sektoren die ihnen zugehörigen Bereiche des VertexBuffers mit Vertices.

Beim Rendern wird dann das Terrain für jeden Sektor einen Sichtbarkeits auslösen, der dann für jeden Patch überprüft, ob dieser im ViewFrustum liegt und ihn gegebenenfalls in die Liste sichtbarer Patches einhängen und das zGrid (ein grob aufgelöster z-Buffer) aktualisieren. Sofern benutzt wird hier auch für jeden Patch der richtig LoD gewählt.

Diese Liste wird dann abgearbeitet, indem für jeden Patch geprüft wird, ob er vollständig im zGrid überlagert wird. Ist er nicht verdeckt wird er gerendert. Hierbei wird anhand der Detailstufen seiner Nachbarn und ihm selbst das passende Rumpfstück, die nötigen Verbindungsstücke und Ecken gerendert.

## 5. Implementierungsaspekte

Für die Landschaft selbst kommen GeoMipMaps zum Einsatz. Diese bestehen aus der Geometrie für die höchste Detailstufe, sowie einigen Index-Buffers für die verschiedenen Detailgrade und Verbindungsstücke zu anderen Chunks mit geringerem Detail. Wenn verfügbar wird außerdem die VertexArrayRange Extension von nVidia verwendet, die es erlaubt die Geometriedaten im AGP-Speicher abzulegen. Dadurch kann die Grafikkarte selbstständig darauf zugreifen und unabhängiger von der CPU arbeiten. Da 500-1000 Polygone pro Rendereaufruf optimal sind, wurde als Kompromiß die Größe der Chunks auf 16x16 festgelegt, wodurch sich 512 Dreiecke bei höchstem LoD ergeben. 32x32 wäre hier passender und würde das Culling beschleunigen, allerdings wird selbiges dadurch ungenauer und die Anzahl der Detailgrade/IndexBuffer würde ansteigen. Aufgrund mathematischer Alpträume blieb die Möglichkeit diesen Wert variabel zu halten im Laufe der Zeit auf der Strecke. Unnötigerweise wird mit Triangle-Strips gearbeitet, was nur unbedeutend schneller als eine Triangle-List ausfällt und teilweise langsamer sein könnte, als eine auf den Cache optimierte Triangle-List (und mehr zeitraubende Aufrufe erfordert).

Für die Texturen wurde ein einfacheres Prinzip gewählt, als bei IScape. Statt groben Texturen und "gesplatteten" Detail-Texturen kamen nur Detail-Texturen zum Einsatz. Davon werden 2-3 auf die gesamte Landschaft gekachelt (besonders unregelmäßige kachelbare Texturen sind also nötig) und mit verschiedenen Gewichten zusammengeblendet. Da auf Vertex/Pixel-Shader verzichtet wurde, war es nötig die Texturgewichte in der Vertexfarbe zu speichern. Da später auf 2 Texturen reduziert wurde genügte der Alphakanal und der Farbwert selbst war wieder frei. Für die Beleuchtung wird eine vorberechnete Lightmap aufmoduliert. Diese könnte man im Laufe eines "Tages" aktualisieren, um eine passende Beleuchtung und korrekte Schatten zu erhalten.

Culling geschieht in zwei verschachtelten Stufen. Zuerst wird ein etwas schnelleres Frustum Culling auf die einzelnen Chunks angewandt (ein Ansatz mit Quadtree zeigte sich als im besten Falle unmerklich schneller und auch das erst nach etlichen Optimierungen). Dieses testet nicht gegen das ViewFrustum, sondern transformiert die Punkte in das Kamera-Koordinaten-System und projiziert sie auf 2 Dimensionen, wodurch der Test selbst trivial wird. Liegen alle Punkte außerhalb einer einzelnen Ebene ist der Chunk nicht sichtbar.

Für einen sichtbaren Chunk wird seine Entfernung zur Kamera in einem 2D-Raster eingetragen, indem der Bereich unterhalb des tiefsten Punktes des Chunks dessen Entfernung als Wert bekommt, sofern er kleiner als der aktuelle Wert ist. Dies funktioniert nur deswegen (meistens), weil die Kamera nicht rollen kann. Wäre dies nötig, so müßte man die Form der Bounding Box mit Raster-Verfahren einschreiben, was wiederum soviel Aufwand verursacht, daß weniger Zeit eingespart als für das Verfahren aufgewandt wird.

Dieses war bei einigen anderen getesteten Methoden der Fall und auch die Occlusion Query Extension von nVidia ist in ihrem Nutzen eher eingeschränkt und in diesem Fall langsamer als ganz darauf zu verzichten. Daher der Entschluß zu obiger grob annähernder Methode.

Das Nachladen der Landschaft soll geschehen, indem einzelne Sektoren der Karte im Speicher überschrieben werden und die Nachbar-Zeiger neu gesetzt werden. Dabei wird das Laden der Texturen und der Heightmap, sowie das Erstellen der Geometrie in kleinen Stücken auf mehrere Frames verteilt. Da später ein Editor direkt die Geometrie exportieren soll entfällt das aufwändige Erzeugen später. Dies ist nicht mehr in der Praktikumsversion implementiert, läuft aber in der "Entwicklungs"-Version in ca. 1 Sekunde pro Sektor ab, ohne die Geschwindigkeit übermäßig herabzusetzen. In insgesamt 3 Sekunden könnte also ein "Schritt" auf der Karte erfolgen. Entsprechend sollte man zum Durchqueren eines Sektors mindestens solange benötigen (ein solcher hat im Spiel eine Kantenlänge von 2.5km).

Ein Hauptproblem war der Speicherverbrauch. Ursprünglich benötigte ein Sektor über 12MB Speicher. Eine sinnvolle Größe von 3x3 Sektoren wäre mit 100MB also kaum praktikabel. Durch Auslagern der Texturkoordinaten (sind für alle Sektoren/Chunks identisch) und Verwenden von Bytes statt Floats für die Farbe, sowie das Verzichtens auf Normalenvektoren wird die Größe allerdings deutlich erträglicher. 4.5MB pro Sektor plus einmalig 2.3MB.

## 6. Ergebnisse

Wie angedeutet entstand zuerst eine isolierte Version, die lediglich OpenGL und Glut verwendete. Alle Erweiterungen und Versuche wurden zuerst hier implementiert und erst relativ spät in das eigentliche Praktikum eingebaut. Bei diesem Schritt wurde dann auch ein Teil des -durch ständige Änderungen entstandenen- Chaos beseitigt. Dieses Vorgehen war einerseits angenehm, weil man ohne jede Abhängigkeit vom übrigen Projekt jederzeit neue Ideen testen konnte, gleichzeitig aber auch ein Grund, warum viele neue Features erst spät oder am Ende gar nicht mehr ins Praktikum einfließen. Hier wäre an vielen Stellen eine komplette Umstellung nötig gewesen. Der sinnvollste Weg scheint es wohl zu sein, das Modul für sich zu vervollständigen und erst dann in das Projekt einzubauen.

Die Arbeit an der Landschaft selbst war daher sehr entspannt, da in vorgegebener und vertrauter Umgebung. Das genaue Gegenteil war das Arbeiten mit dem eigentlichen Projekt. Von den üblichen Problemen mit verschiedenen Plattformen abgesehen wurden regelmäßig neue Bibliotheken eingebaut. Insgesamt hat die Arbeit am Projekt weniger Zeit gekostet, als das Identifizieren fehlender Header, Auffinden, Installieren und Kompilieren der dazugehörigen Bibliothek und letztlich das zum Laufen bringen unter Windows. Hier wäre es sehr wünschenswert gewesen, wenn neue Bibliotheken zuerst unter allen betroffenen Betriebssystemen getestet und danach zentral verfügbar oder wenigstens verlinkt worden wären.

Selbst die Vorlage für diesen Bericht benötigte erstmal zum Installieren eines 100MB Downloads, um mehr als den Quelltext zu sehen.

Leistungsmäßig ist die Landschaft durchaus eine positive Überraschung und auf einem halbwegs modernen System sollten keine Probleme auftreten. Die CPU hat wie geplant kaum Einfluß auf die Geschwindigkeit und schon eine GF4 bewältigt in der Praxis nicht auftretende Sonderfälle ausreichend schnell (2.28 Millionen dreifach texturierte Polygone bei 25fps).

Optisch ist sie ebenfalls in Ordnung, allerdings wenig aufregend, durch die Einschränkung auf 2 Texturen pro Chunk. Vegetation wird ebenfalls nicht von der Landschaft erledigt, sondern erfolgt "von außen". Für die Entwicklungszeit von knapp einem Semester sollte sie mit den richtigen Texturen allerdings genug Möglichkeiten bieten.

Triangle-Strips sind auf modernen Karten mit ausreichendem Vertex-Cache nicht mehr nötig. Auch hier zuviel Zeit und unschöner Code.

Weniger zufriedenstellend fallen LoD und Occlusion Culling aus. Zwar erfüllen sie ihren Zweck, allerdings ist GeoMorphing bei diesem Ansatz nicht ohne weiteres möglich, wodurch stellenweise noch sichtbare Wechsel zwischen einzelnen LoDs entstehen. Bei kleinerer Sichtweite wäre es wohl besser, den LoD nur zu Beginn aufgrund des Terrains zu wählen und diesen beizubehalten.

Occlusion Culling ist eine relativ schwerwiegende Fehlentscheidung gewesen. Zum einen macht es nur auf relativ schwachen Systemen einen Unterschied zwischen spielbar und unspielbar und zum anderen sind selbst in der momentanen Entwicklungsversion noch unbehebbar Fehler vorhanden. Diese sind zwar mittlerweile stark reduziert, aber dennoch wird die Engine dadurch eingeschränkt. Übermäßig hohe Landschaften (für tiefe Schluchten oder ähnliches) verdecken fälschlicherweise sichtbare Teile der Landschaft. Außerdem benötigt das Occlusion Culling eine aufwendigere Methode beim Frustum Culling. Dieses könnte abgebrochen werden, sobald feststeht, daß ein Chunk sichtbar ist. Stattdessen müssen die restlichen Punkte dennoch transformiert und projiziert werden. Zwar ist das Occlusion Culling abstellbar, aber ein paar Stellen könnten bei vollständigem Verzicht noch ein wenig optimiert werden. Damit würde das reine Frustum Culling samt Erstellen der Liste sichtbarer Chunks bei 4096 Tests unter 1ms laufen.

Der Quadtree war zum Glück ein schnell abgehaktes Experiment, das mit zusätzlichem Aufwand und Speicherbedarf für diese Anzahl von Chunks eher kontraproduktiv war (und es auch nicht ins Projekt geschafft hat). Ohne die KI Vorlesung wäre der Ansatz wohl nie über die kreuzlahme Rekursion hinausgekommen und selbst der Listenansatz ist durch das ständige Anlegen und Freigeben von Objekten extrem nutzlos. Erst ein verschwenderischer Ansatz mit einem Array kam wenigstens auf die gleiche Geschwindigkeit wie der Brute Force Test aller Chunks.

Insgesamt könnte der gesamte Code also kompakter und übersichtlicher sein, wenn auf Funktionen verzichtet würde, die sich als relativ nutzlos erwiesen haben. Für den ersten Kontakt mit dem Thema Terrain Renderer ist das Ergebnis und vor allem die gemachten Erfahrungen allerdings ok.

## 7. Zukünftige Entwicklung

Für das Praktikum wäre ein Entfernen des Occlusion Cullings eine gute Idee, sollten damit vermehrt Probleme auftreten. Ob darauf ganz verzichtet oder ein anderer Ansatz gewählt wird hängt dann von der Performance ab. Das Handhaben des LoD ist ein weiterer Punkt. Bei übermäßiger Sichtweite wird es nicht ohne gehen, wird diese allerdings auf übliche Werte eingeschränkt, dann könnte auch hier einiges entfallen. Statt Chunks in höchstem Detailgrad mit vielen IndexBuffern könnte der Editor einen Chunk mit der wirklich nötigen Geometrie erstellen. Das spart Code und extrem viel Speicher.

Das Nachladen ist noch nicht implementiert, sollte aber auch nicht gemacht werden, solange nicht der Editor steht und das endgültige Format festgelegt ist. Nachdenken könnte man über die Verwendung von Shadern, die wesentlich mehr Freiheit bei den Texturen lassen würden.

Die Zukunft der eigentlichen Landschaft vor Ort liegt in Direct3D und damit einer einheitlichen Grundlage für alle Karten. Damit erledigen sich etliche Abfragen und alternative Codepfade. Außerdem zwingt der Wechsel zu einem kompletten Neustart, bei dem die gemachten Erfahrungen von Anfang an einfließen können.

Außerdem wird früher oder später eine neue Version des Occlusion Cullings getestet. Der naive Erstanatz beinhaltet zwar einige Fehler, funktioniert aber relativ schnell. Statt dem sturen Schattenwurf nach unten soll der Umriss der Boundingbox korrekt in das Raster eingezeichnet werden. Perspektivische Fehler würden dadurch vermieden, gleichzeitig nimmt die Komplexität deutlich zu. Der Versuch wird vermutlich kurz gehalten und danach vollständig aufgegeben.

## 8. Literatur

Greg Snook, "Simplified Terrain Using Interlocking Tiles"  
(Charles River Media, 2001, "Game Programming Gems 2")

Jason Shankel, "Fast Heightfield Normal Calculation"  
(Charles River Media, 2002, "Game Programming Gems 3")

Willem H. de Boer, "Fast Terrain Rendering Using Geometrical MipMapping"  
(<http://www.flipcode.com/tutorials/geomipmaps.pdf>)

<http://www.futurenation.net/glbases/lscapetech.htm>

<http://www.gamedev.net/community/forums/>

<http://www.flipcode.com/forums/>

## Anhang A Code Dokumentation

camera.h

Die Kamera ist dafür verantwortlich, die Projektions- und Transformations-Matrix vorzubereiten und Objekte auf Sichtbarkeit zu prüfen.

Globale defines:

MOUSE\_SLOWDOWN: Faktor, um die Mausbewegung in Pixel in eine Drehung umzurechnen.

XROTATION\_LOCK\_UP, XROTATION\_LOCK\_DOWN: Grenzwerte für die Rotation, verhindert ein Überschlagen der Kamera.

CAM\_MOVE\_X, CAM\_MOVE\_Y, CAM\_MOVE\_Z: Konstanten für die Bewegungsachsen der Kamera.

CAM\_SPEED: Faktor für die Bewegungsgeschwindigkeit der Kamera

GridSizeX, GridSizeY: Größe des zGrids

Klasse Camera:

Public Attribute:

MovementType: Enum der Bewegungsarten, MOVE\_FREE bewegt die Kamera frei im Raum,

MOVE\_FOLLOW hängt die Kamera an ein Objekt.

VisibleObjects: Vector, in dem nach dem Culling alle sichtbaren Objekte abgelegt werden.

ms\_zGrid[GridSizeX][GridSizeY]: das zGrid als 2D-Array von z-Werten

Private Attribute:

m\_Transform[16]: Die Transformationsmatrix der Kamera.

m\_HSize, m\_VSize: Entspricht der horizontalen und vertikalen Größe des Bilds in Weltkoordinaten.

m\_Fovy: Vertikales Field of View

m\_Aspect: Verhältnis Höhe zu Breite des Blickfelds

m\_NearP: Near Clipping Plane

m\_FarP: Far Clipping Plane

xf, yf: Faktoren, um Punkte auf das Grid zu mappen

szx, szy: GridSize-1, entspricht höchstem Index der Gridzellen

gsx, gsy: GridSize/2, dient zur Optimierung

m\_AngleX, m\_AngleY: Winkel der Kamera zum Objekt im Follow-Mode

m\_Dist: Entfernung zum Objekt im Follow Mode

m\_MoveType: Momentan aktiver Bewegungsmodus

m\_pFollowEntity: Zeiger auf das Objekt, dem die Kamera folgt

Methoden:

Camera(float x=0.0f, float y=0.0f, float z=0.0f, float fovy=45.0f,  
float near=0.1, float far=512.0f)

Beschreibung: Konstruktor

Parameter:

x,y,z: Position der Kamera  
fovy: Vertikales FoV in Grad  
near: Near Clipping Plane  
far: Far Clipping Plane

void gridWrite(float xb0, float xb1, float x0, float x1, float y0, float y1, float z0, int\* c=0, bool  
nowrite=false)

Beschreibung: Füllt einen Bereich des zGrids mit einem z-Wert,  
speichert die Eckpunkte gegebenenfalls in übergebenem Array

Parameter:

x0, x1, y0, y1: Eckpunkte des zu füllenden Bereichs in  
Kamerakoordinaten  
z: Der zu schreibende z-Wert  
c: Zeiger auf ein IntegerArray, in dem die berechneten Indices  
gespeichert werden  
nowrite: Flag, falls nicht ins zGrid geschrieben werden soll

bool gridTest(int\* c, float z)

Beschreibung: Testet für einen Bereich, ob er verdeckt ist. Liefert  
falls der Bereich sichtbar ist

true,

Parameter:

c: Zeiger auf IntegerArray mit den Indices des zu testenden Arrays  
z: Der z-Wert, mit dem verglichen werden soll

void clearGrid():

Beschreibung: Setzt alle Werte im zGrid auf 0

void setMovementType(MovementType f\_Type,  
PositionalEntity\* f\_pFollowObject = NULL)

Beschreibung: Setzt den Bewegungsmodus für die Kamera

Parameter:

f\_Type: Der gewünschte Bewegungsmodus  
f\_pFollowObject: Zeiger auf das Objekt, dem die Kamera folgen soll

void setFarPlane(float f\_pFar)

Beschreibung: Setzt die Far Clipping Plane

Parameter:

f\_pFar: Neuer Wert für die Far Clipping Plane

float getFarPlane()

Beschreibung: Liefert die Far Clipping Plane

void setView()

Beschreibung: Setzt die Modelview Matrix entsprechend der Kameraposition. Muß aufgerufen werden, bevor etwas gerendert wird.

void calcPerspective(int w, int h)

Beschreibung: Berechnet für eine Bildhöhe/Bildbreite die nötigen Parameter und ruft setPerspective mit diesen auf.

Parameter:

w: Die neue Bildbreite

h: Die neue Bildhöhe

void setPerspective()

Beschreibung: Setzt die OpenGL Projektionsmatrix für die Kamera

void moveRelative(int direction, float speed)

Beschreibung: Verschiebt die Kamera relativ zu ihrer aktuellen Ausrichtung.

Parameter:

direction: Eine der Bewegungsrichtungs-Konstanten

speed: Die Geschwindigkeit in m/s

void rotateByMouseMotion(int x, int y)

Beschreibung: Dreht die Kamera in typischer FPS Art

Parameter:

x: horizontale Drehung in Grad

y: vertikale Drehung in Grad

void setPosition(float x, float y, float z)

Beschreibung: Plaziert die Kamera an eine neue Position

Parameter:

x,y,z: Die neue Position der Kamera

void setAt(float x, float y, float z)

Beschreibung: Richtet die Kamera auf einen Punkt aus

Parameter:

x,y,z: Der anvisierte Punkt

void setRelative()

Beschreibung: Richtet die Kamera relativ zu einem verfolgten Objekt aus

GLfloat\* getPosition()

Beschreibung: Liefert einen Zeiger auf die Position der Kamera

bool pointInFrustum(float\* p)

Beschreibung: Testet für einen Punkt, ob er im Frustum liegt.

Parameter:

p: Zeiger auf den zu prüfenden Punkt

float bboxVisible(GLfloat\* bbox, int\* coords)

Beschreibung: Für Landschaftspatches, prüft ob diese sichtbar sind und liefert 0 falls außerhalb, 1 oder zmin falls sichtbar.

Parameter:

bbox: Zeiger auf die Eckpunkte der Bounding Box

coords: Zeiger auf ein Array für Eckpunkte im zGrid

int bboxInFrustum(float\* bbox);

Beschreibung: Prüft für eine Bounding Box, ob diese im Frustum liegt.

Liefert 1 falls komplett sichtbar, 0 wenn teilweise sichtbar und -1 falls komplett unsichtbar

Parameter:

bbox: Zeiger auf die Eckpunkte der Bounding Box

void drawFrustum()

Beschreibung: Rendert das ViewFrustum der Kamera

void drawGrid()

Beschreibung: Rendert das zGrid in Farbkodierung

terrainEntity.h

Das TerrainEntity lädt die Landschaft aus einer Heightmap und erstellt alle benötigten Daten. Sämtliche Schnittstellen nach außen sollten sich hier befinden.

Klasse Camera:

Public Attribute:

Sector[4]: Zeiger auf die aktuell geladenen 4 Sektoren

VrtxOffset: Aktueller Offset ins Vertex Array (nächste leere Stelle)

VisPatches[4096]: Array, in dem die sichtbaren Patches abgelegt werden

numVisPatches: Die Anzahl momentan sichtbarer Patches

m\_pTerrainRenderer: Zeiger auf den Terrain Renderer

VrtxBuffer: Das Vertex Array

Methoden:

virtual void render(Camera\* f\_pCamera)

Beschreibung: Stößt die Landschaft an sich zu rendern

Parameter:

f\_pCamera: Ein Zeiger auf die zu verwendende Kamera

void loadSector(int i, char\* f\_pFile)

Beschreibung: Lädt einen Sektor aus einer Heightmap

Parameter:

i: Der Index des zu ladenden Sektors

f\_pFile: Dateiname der Heightmap

void getNormal(float f\_X, float f\_Z, Vector3d\* f\_pResult) const

Beschreibung: Berechnet den interpolierten Normalenvektor an einer Stelle

Parameter:

f\_X, f\_Z: Die Koordinaten, für die die Normale berechnet werden soll

f\_pResult: Zeiger auf einen Vektor, in den das Ergebnis geschrieben wird

float getHeight(float f\_X, float f\_Z, Vector3d\* normal=0) const

Beschreibung: Berechnet die interpolierte Höhe an einer Stelle

Parameter:

f\_X, f\_Z: Die Koordinaten, für die die Höhe berechnet werden soll

normal: Um gleichzeitig die interpolierte Normale zu speichern

void SetLevels(float\* f\_Pos, GeoPatch\*\* VisPatches, int idx)

Beschreibung: Setzt für die sichtbaren Patches den LoD

Parameter:

f\_Pos: Position der Kamera

VisPatches: Zeiger auf das Array sichtbarer Patches

idx: Anzahl sichtbarer Patches

void DrawNormals(int x, int y, int size)

Beschreibung: Zeichnet in einem Bereich die Normalen

Parameter:

x,y: Position der Bereichsmitte

size: Die Größe des Bereichs

Protected:

void initMembers()

Beschreibung: Das eigentliche Laden und Verknüpfen der Sektoren

virtual void registerEntity()

Beschreibung: Landschaft bei der Physik registrieren

geoConsts.h

Beinhaltet immer wieder benötigte Konstanten für die Landschaft

Globals:

g\_PatchSize: Die Größe für einen Patch

g\_SectorSize: Die Größe für einen Sektor (Heightmap muß 1 größer sein)

g\_PatchPerSector: Anzahl der Patches in einem Sektor (SectorSize/PatchSize)

g\_NumLevels: Anzahl der LoDs ( $2^{(\text{NumLevels}-1)} = \text{PatchSize}$ )

g\_VerticesPerSector: Anzahl der Vertices in jedem Sektor

g\_Scale: Skalierungsfaktor in der Breite

g\_HeightFac: Skalierungsfaktor in der Höhe

```
struct GeoVertex {float u,v,s,t; unsigned char r,g,b,a; float x,y,z;}
```

Beschreibung: Format für ein Vertex wie folgt

u,v, s,t: Zwei 2D Texturkoordinaten

r,g,b,a: Farbwert und Alpha

x,y,z: Position

```
struct Vector {float x, y, z;}
```

Beschreibung: Eins von vielen Vectorformaten

geoLevel.h

Hier werden die IndexBuffer für die einzelnen Detailgrade und Bauteile erstellt.

Globale Defines:

GEO\_RIGHT, ...: Die Nachbarindices für bessere Lesbarkeit

```
struct CBodyPiece {unsigned short* IdxBuffer; unsigned int IdxBufferSz;
```

```
    unsigned short* CornIdxBuffer; unsigned int CornIdxSz;};
```

Beschreibung: IndexBuffer für die Rümpfe der Patches

IdxBuffer: Der IndexBuffer für ein LoD

IdxBufferSz: Größe des IndexBuffers

CornIdxBuffer: IndexBuffer für die Ecken für ein LoD

CornIdxSz: Größe des Buffers für die Ecken

```
struct CLinkingPiece {unsigned short* IdxBuffer[4]; unsigned int IdxBufferSz[4];};
```

Beschreibung: Die Verbindungsstücke zu kleineren LoDs

IdxBuffer[4]: IndexBuffer für jede Seite

IdxBufferSz[4]: Größe des Buffers für jede Seite

Klasse GeoLevel

Public Attribute:

Bodies: Ein Array von BodyPieces

Linkers: Ein Array von LinkingPieces

Methoden:

void BuildBodies(unsigned lvl, unsigned PatchSize)

Beschreibung: Erzeugt alle BodyPieces für ein LoD

Parameter:

lvl: Der LoD, für den die Teile erstellt werden sollen

PatchSize: Die Größe eines Patches

void BuildLinkers(unsigned lvl, unsigned PatchSize)

Beschreibung: Erzeugt alle LinkingPieces für ein LoD

Parameter:

lvl: Der LoD, für den die Teile erstellt werden sollen

PatchSize: Die Größe eines Patches

terrainRenderer.h

Renderer für die Landschaft

Klasse TerrainRenderer:

Protected Attribute:

m\_pTextureManager: Zeiger auf den TexturManager

Levels: Zeiger auf die einzelnen LoD IndexBuffer

Methoden:

TerrainRenderer(TextureManager\* f\_pTexMngr)

Beschreibung: Konstruktor

Parameter:

f\_pTexMngr: Zeiger auf den TexturManager

void render(GeoPatch\*\* f\_pVisPatches, int f\_NumPatches)

Beschreibung: Rendert die sichtbaren Patches

Parameter:

f\_pVisPatches: Zeiger auf das Array sichtbarer Patches

f\_NumPatches: Anzahl sichtbarer Patches

geoSector.h

Beinhaltet die Sektoren Klasse.

Globale Defines:

RIGHT, TOP, ...: Die vier Richtungen für die Nachbar-Zeiger

Klasse GeoSector:

Public Attribute:

xpos, ypos: Die Position des Patches in Weltkoordinaten

m\_pHeightMap: Das Array für die Höhenwerte

Patches: Das Array für die Patches

Methoden:

void Render(GeoPatch\*\* VisPatches, int& idx)

Beschreibung: Nicht mehr verwendet und ohne Implementierung.

void SetNbr(GeoSector\* r, GeoSector\* t, GeoSector\* l, GeoSector\* b)

Beschreibung: Setzt für den Sektor die Zeiger auf seine 4 Nachbarn

Parameter:

r,t,l,b: Zeiger auf die Sektoren rechts, über, links und unter dem Sektor

void load(char\* file, unsigned int VrtxOffset, GeoVertex\* VrtxBuffer)

Beschreibung: Lädt eine Heightmap und stößt die Erstellung der Geometrie an

Parameter:

file: Dateiname der Heightmap

VrtxOffset: Der Offset dieses Sektors im VertexBuffer

VrtxBuffer: Ein Zeiger auf den VertexBuffer

Bemerkung: Wahrscheinlich unsinnig, stattdessen Zeiger auf die Stelle im Buffer übergeben

GeoSector(int x, int y):xpos(x), ypos(y)

Beschreibung: Konstruktor

Parameter:

x,y: Position des Sektors in Weltkoordinaten

geoPatch.h

Beinhaltet die Patch Klasse

Klasse GeoPatch:

Public Attribute:

m\_Level: Der aktuell für diesen Patch ausgewählte LoD  
VrtxData: Ein Zeiger auf die Geometrie des Patches im VertexBuffer  
m\_BaseTex: Die erste Textur für diesen Patch  
m\_SecondTex: Die zweite Textur für diesen Patch  
scoords[4]: Array mit Eckindices des belegten Bereichs im zGrid  
m\_Error[4]: Die Abweichung vom höchsten LoD für jeden Detailgrad  
m\_BaseLevel: Der aufgrund der Geometrie gewählte Basis LoD  
z: Die Entfernung zur Kamera, bzw. z-Koordinate des Patches  
radius: Der Radius der BoundingSphere des Patches  
center: Der räumliche Mittelpunkt des Patches in Weltkoordinaten  
BBox[6]: Die BoundingBox, gegeben durch zwei Eckpunkte Min/Max  
Nbr[4]: Zeiger auf die vier Nachbarn des Patches

Methoden:

```
void Build(int xoff, int yoff, unsigned xpos, unsigned ypos,  
          unsigned& VrtOff, const unsigned char* HeightMap, GeoVertex* VrtxBuffer)
```

Beschreibung: Erstellt einen Patch, schreibt die Geometrie in den Buffer und erzeugt BoundingVolumes

Parameter:

xoff, yoff: Offset relativ zur Welt, entspricht Position des Sektors

xpos, ypos: Position im Sektor

VrtOff: Offset in den VertexBuffer, wird beim Erstellen verändert

HeightMap: Zeiger auf das Array mit Höhenwerten

VrtxBuffer: Zeiger auf den VertexBuffer

Bemerkung: Für bessere Lesbarkeit/Verständlichkeit xoff, xpos zusammenfassen, evtl. Zeiger in den VertexBuffer übergeben und Zahl der erstellten Vertices zurückgeben.

## Anhang B

### Details zur Implementierung

#### Camera::gridWrite(...):

Die Minimum und Maximum x-y-Werte der Eckpunkte einer BBox in Kamerakoordinaten werden auf das Raster umgerechnet. Dabei wird beim Schreiben abgerundet (nur vollständige gefüllte Zellen werden geschrieben) und beim Lesen aufgerundet (alle berührten Zellen werden getestet). Außerdem wird beim Schreiben der Bereich unter der Bounding Box (0 - MinY) gefüllt, da die Box selbst nicht vollständig mit Terrain gefüllt ist. Fehlerverminderung: Nur die unteren 4 Punkte einer BBox sollten beachtet werden. Insgesamt sollte aber eine Methode benutzt werden, die die perspektivische Verzerrung berücksichtigt.

#### Camera::pointInFrustum(...):

Da für die Kamera die Matrix bekannt ist, lassen sich Punkte durch einfache Skalarprodukte in Kamerakoordinaten transformieren. Da das Frustum symmetrisch ist gibt es keinen Grund sich Ebenen zu berechnen und 6 Tests zu machen. Die Entfernung kann direkt mit der near und far clipping plane verglichen werden, für x und y wird ein Faktor benötigt. Von oben betrachtet entfernt sich die Seite des Frustums linear von der Mittelachse. Für einen Punkt muß also nur getestet werden, ob seine x Koordinate kleiner ist als seine z Koordinate, multipliziert mit diesem Faktor. Dieser ist einfach der Tangens des Winkels zwischen Mittelachse und Seite des Frustums, was wiederum die Hälfte des Blickfelds ist.

#### Camera::bboxInFrustum(...):

Wie oben, es werden alle Punkte getestet und falls alle Punkte außerhalb der gleichen Ebene liegen, so ist die Box nicht sichtbar.

#### TerrainEntity::SetLevels(...):

Die Zahlen hier sind relativ willkürlich und müssen einfach solange angepaßt werden, bis man mit dem Ergebnis zufrieden ist. Eine Änderung der Sichtweite würde das Ergebnis stark verändern, deswegen wird von einer festen Sichtweite ausgegangen.

#### GeoPatch::Build(...):

Hier sollten die Texturkoordinaten nicht mehr auftauchen. Stattdessen sollten sie einmalig für Patches und Sektoren gespeichert werden. Für die Verwendung der Vertex Array Range Extension müssen sie allerdings ebenfalls im gleichen Speicher liegen, wie die restlichen Daten. Es muß also der GeoVertex-Zeiger auf das Array vorübergehend in einen float-Zeiger gecastet und der Offset danach entsprechend angepaßt werden.

## Anhang C "Tagebuch"

### 1. Investigatives Programmieren

Roam (Realtime optimally adapting mesh) ist der Begriff, der einem als erster und später immer wieder begegnet. Dabei wird für ein Vertex (meistens anhand der Abweichung in Bildschirmkoordinaten) entschieden ob es weggelassen werden kann. Dabei entsteht zwar ein Mesh, daß die Form gut annähert und bereits Entfernung und Blickwinkel berücksichtigt, allerdings ist die Methode zu CPU-lastig und ignoriert munter das Bemühen der GPU-Entwickler möglichst viel Arbeit von der CPU auf den Grafikchip zu verlagern.

Also lieber einige Dreiecke mehr, aber dafür möglichst geringen Rechenaufwand. Anstatt regelmäßig ein komplett neues Mesh zu generieren (bzw. andere Indices für das vollständige Mesh) wäre eine gröbere Methode angebracht. Eine gute Lösung sind GeoMipMaps. Ähnlich den Textur MipMaps gibt es die Landschaft in mehreren Detailgraden, wobei der tatsächliche auf verschiedene Arten (oder eine Kombination von allen) bestimmt wird. Sinnvoll einsetzen läßt sich diese Methode natürlich nur, wenn die Landschaft in kleinere Abschnitte aufgeteilt wird, für die sich der Detailgrad einzeln wählen läßt. Dabei wird die maximale Abweichung für jeden Detailgrad und für jeden Abschnitt berechnet, indem pro Level die Abweichung der einzelnen Vertices bestimmt wird. Eine andere Annäherung wäre z.B. die maximale Steigung in einem Abschnitt. Dabei würden aber leichte Bodenwellen unter den Tisch fallen, die einen hohen Detailgrad benötigen würden, während umgekehrt bei günstiger Lage auch eine starke Steigung mit geringer "Auflösung" dargestellt werden könnte.

Die letztlich verwendete Methode sind GeoMipMaps mit einer Patchgröße von 17x17 Vertices und 5 Detailstufen. Für jeden Patch wird anhand des maximalen Fehlers eine Basis-Detailstufe bestimmt, die anhand der Entfernung verändert wird. Für nahe Bereiche wird die Stufe solange reduziert, bis Fehler/Entfernung einen Höchstwert überschreiten würde. So wird verhindert, daß komplett flache Bereiche nur aufgrund ihrer Nähe mit 512 statt 2 Dreiecken dargestellt werden. Ein besseres Verfahren bzw. genauere Anpassungen der Werte wird gegen Projektende vorgenommen.

### 2. Ich sehe was, was du nicht siehst

Neben LOD ist natürlich das Entfernen nicht sichtbarer Bereiche ein wichtiger Schritt, um die dargestellte Geometrie gering zu halten. Der einfache Teil ist dabei das Frustum Culling, bei dem die Bounding Box jedes Patches darauf geprüft wird, ob sie für die Kamera sichtbar ist. Das übliche Verfahren besteht dabei darin, gegen die 6 Flächen des Kamera-Frustums zu testen, was in meinen Augen für diesen Sonderfall unnötig ist. Statt 6 (bzw. 4, da die übrigen 2 trivial sind) Normalenvektoren für das Frustum zu bestimmen und zu transformieren werden stattdessen nur die ohnehin in der Transformationsmatrix der Kamera gespeicherten Vorne/Rechts/Oben-Vektoren verwendet. Dafür wird die Differenz des zu testenden Punktes und der Kameraposition berechnet. Das Skalarprodukt mit dem Vorne-Vektor ergibt die z-Koordinate im Kameraraum. Ist dieser Wert kleiner als die Near Clipping Plane oder größer als die Far Clipping Plane liegt der Punkt außerhalb. Wird z.B. nur eine Bounding Sphere getestet kann hier bereits der Test beendet werden. Ansonsten wird der Vorgang mit den beiden übrigen Vektoren wiederholt. Um die x/y-Koordinaten testen zu können benötigt man jeweils einen Faktor, der vom FOV der Kamera abhängt ( $\tan(\text{winkel}/2)$ ). Dieser Faktor entspricht der Breite/Höhe einer Clipping Plane in der Entfernung 1. Verglichen wird wahlweise x und z\*Faktor oder x/z und Faktor.

Einschub: sofern die korrekten Koordinaten nicht weiter benötigt werden ist ersteres zu bevorzugen. In dieser Implementierung dauert die Division (genauer: das Zurückschreiben des Ergebnisses vom Stack in den Speicher) jeweils ca. 800-1000 Takte und erhöht die durchschnittliche Ausführungszeit des Frustum Cullings um 25%. Wieso hier scheinbar vollkommen am Cache vorbeigeschrieben wird ist unklar und für das Praktikum auch eher bedeutungslos.

Für eine Bounding Box sieht das Verfahren anders aus, je nachdem, ob man lediglich zwischen sichtbar/unsichtbar oder zwischen komplett sichtbar/teilweise sichtbar/unsichtbar unterscheidet (z.B. weil man einen Quadtree verwendet und bei vollständiger Sichtbarkeit eines Knotens nicht nochmal alle Childs unnötig testen will). In ersterem Fall wird abgebrochen, sobald ein Punkt im Frustum liegt (die Box ist zwangsweise mindestens teilweise sichtbar). Für den Großteil der sichtbaren Volumen ist der Test also nach dem ersten Punkt beendet. Komplett unsichtbare lassen sich durch einen vorgeschobenen Bounding Sphere test ebenfalls frühzeitig erkennen. Da hier ein Quadtree benutzt wird entstand allerdings ein Problem. Rekursive Durchläufe erzeugten zuviel Overhead und waren dadurch langsamer als das sture Durchtesten aller 4096 BBs. Iterative Verfahren mit STL-Liste/Queue waren durch das ständige Anlegen und Freigeben einzelner Elemente ebenfalls deutlich langsamer. Letztlich wird ein einfaches Array von 4096 Zeigern reserviert, um möglichst effizient durch den Baum zu laufen.

Die zweite Hürde: für den Quadtree mußten jetzt grundsätzlich alle 8 Punkte getestet werden, statt wie bisher in der Hälfte der Fälle nach dem ersten abzubrechen. Auch hier war das Ergebnis ernüchternd und der Quadtree war weiterhin langsamer als stures Testen. Leichte Vorteile wurden erst erreicht, indem zusätzlich für die Knoten ein BoundingSphere Test durchgeführt wurde, der komplett sichtbare und komplett unsichtbare Knoten frühzeitig findet. Wirklich zu empfehlen ist der zusätzliche Aufwand für einen Quadtree aber offensichtlich erst bei deutlich mehr als 4000 Blättern oder einer sehr großen Sichtweite (die dann wieder andere Probleme mit sich bringt).

Der zweite Schritt besteht im Occlusion Culling. Es gibt zwar viele Methoden, aber jede einzelne hat hier mehr CPU-Zeit gekostet, als die eingesparte Geometrie an Render-Zeit gespart hat. Außerdem setzen die meisten Methoden voraus, daß möglichst gute Occluder gewählt werden. Also mußte etwas neues her, was möglichst geringen Aufwand verursacht, aber dennoch einiges vom Rendern ausschließt. Der logische Schritt vom Frustum Culling zum Occlusion Culling wird hier relativ klein, dadurch, daß schon oben ein eigener Ansatz verwendet wurde, der beiläufig die Bildschirm Koordinaten der Bounding Boxes liefert. Zusätzlich läßt sich durch bestimmte Einschränkungen eine sparsame Methode anwenden.

a) Die Kamera rollt nicht

Man kann also davon ausgehen, daß sich die Landschaft unendlich nach unten ausdehnt (sowohl in Welt- als auch in Bildschirmkoordinaten)

b) Eine BB ist nur die Krone auf einer unendlichen Säule, die ansonsten die Maße der BB vollkommen ausfüllt.

Ohne diese Einschränkungen müßte man für den allgemeinen Fall mehr Aufwand betreiben (z.B. eine "eingeschriebene" Box für große Objekte zusätzlich zur Bounding Box speichern).

Ursprünglich war angedacht, für jeden Patch den von ihm verdeckten rechteckigen Bereich und einen z-Wert zu speichern. Da hier aber jeder noch sichtbare Patch gegen sämtlich Rechtecke verglichen werden müßte wird stattdessen ein Raster benutzt. Dieses z-Grid besteht hier aus 32x32 Zellen, in denen jeweils der kleinste z-Wert gespeichert wird, der diese Zelle vollständig ausfüllt. Im wesentlichen also ein niedrig aufgelöster z-Buffer, der durch die enge Verbindung mit dem Frustum Culling wenig Aufwand erfordert. Eine ähnliche Methode in Hardware war das wesentliche Feature von Kryo 2 Karten und wird als Hyper-Z von ATI verwendet. Bei uns entfällt allerdings der Zwang von vorne nach hinten zu rendern (was allerdings für den normalen z-Buffer vorteilhaft wäre). Außerdem muß die Geometrie nicht erst unnötig zur Karte geschickt werden, um dort ignoriert zu werden.

Beim Schreiben ins z-Grid wird das Rechteck nebenbei kleiner gerundet (um nicht ganz gefüllte Zellen nicht zu überschreiben), während beim Lesen größer gerundet wird. Um den Test zu beschleunigen werden in jedem Patch seine beiden Eckpunkte für den Test gespeichert.

Auch diese Methode bremst manchmal mehr als sie beschleunigt, da wie schon beim Quadtree für sichtbare Boxen alle 8 Eckpunkte durchlaufen werden müssen (allerdings nur für Blätter, die den ersten Test bestanden haben). In flachen Gebieten wird sie zwar nur wenig Zeit kosten, aber dafür dennoch nichts bringen. Zum Ausgleich kann das Raster eingebledet werden, was für nicht-fantasy Spiele als Effekt mißbraucht werden kann, um die Landschaft grob zu erkennen, selbst wenn sie nicht dargestellt wird.

### 3. Rein damit

Um den immer mehr entarteten Terrainrenderer endlich ins eigentliche Projekt zu integrieren, ist erstmal alles was über die reine Darstellung hinausgeht weggelassen worden. Da ein großer Teil der zusätzlichen Funktionen entweder von der Kamera abhängt oder einige angepaßte Funktionen in der Kameraklasse nötig machen schien das wesentlich sinnvoller als stundenlange Fehlersuche und Anpassungen in 5x mal umfangreichem Code.

Über die Weihnachtsferien hätte jetzt der Rest eingebaut werden sollen, um im neuen Jahr mit Kamera und Landschaft fertig zu sein und sich um ein Partikelsystem zu kümmern. Der 'für jedes Problemchen gibts eine dicke Bibliothek'-Ansatz verhindert allerdings effizient jede Heimarbeit, da die Worldforge Mathebibliothek etliche Template-Sauereien benutzt, an denen sich der VC++ Compiler die Zähne ausbeißt.

### 4. Dann eben Partikel

Solange an Terrain und Kamera nicht weitergemacht werden kann wird jetzt eben ein isoliertes Partikelsystem entstehen, bzw. ein längst vorhandenes von D3D zu OpenGL portiert (hoffentlich ohne wieder die gleichen Fehler zu machen).

Nach einigen unerwarteten Problemen sieht es gut aus. Etwas entsetzt mußte ich feststellen, daß Pointsprites erst von Geforce 4 aufwärts in Hardware unterstützt werden und erst aktuelle Treiber den Support für alle Karten in Software mitbringen. Da die Emulation im Treiber allerdings mit großer Sicherheit schneller sein dürfte als manuelles Billboard-Ausrichten für tausende Partikel werden sie vorerst dennoch benutzt (später kann das System entsprechend für ältere Systeme angepaßt werden).

Um möglichst effizient zu rendern kommt natürlich wieder nur AGP-Speicher in Frage, um so das eigentliche Rendering von der CPU zu lösen. Der Gedanke ist, daß einige Partikel aktualisiert und dann gerendert werden, so daß während dem Rendern die CPU bereits das nächste Partikel-Paket vorbereiten kann.

Allerdings sollte man einiges beachten:

Zugriffe auf AGP-Speicher sind langsamer, daher wird das System NICHT komplett im AGP-Speicher gehalten und dort aktualisiert. Außerdem könnten sich hier Zugriffe durch CPU und GPU beeinflussen und Probleme machen.

Aktualisierte Partikel Stück für Stück in einer Schleife kopieren scheint einfach und bequem, ist aber nicht sinnvoll. Stattdessen wird der gesamte Speicher-Block kopiert. Dafür müssen die Partikel aber in einer Form vorliegen, die von der Karte direkt genutzt werden kann. Dieser Teil ist kein Problem, da man für Vertex, Farbe, etc. den Abstand vorgeben kann und so gezielt die relevanten Daten bekommt. Dabei würde aber erstens unnötig viel kopiert werden und zweitens wohl auch der Cache sabotiert. Deshalb besteht das System aus zwei Arrays: einmal mit den Punktdaten (Position und Farbe) und einmal der Rest (Geschwindigkeit, Lebensdauer, etc.).

Der Vorteil besteht bisher darin, daß zusätzlich zu den Daten im Partikelsystem nur relativ wenig AGP-/(evtl. auch Grafik-)Speicher reserviert werden muß, da dieser immer wieder mit dem nächsten Paket überschrieben wird.

Dabei sollte unbedingt die Fence-Extension benutzt werden, um nicht versehentlich den Speicher zu überschreiben, der gerade gerendert wird.

Sollte es hier größere Verzögerungen geben wird einfach Platz für zwei Pakete reserviert und diese im Wechsel benutzt.

Die eigentliche 'Partikel-logik' kann dann von meinem früheren D3D-System übernommen werden, wobei viele fehlende Funktionen noch für einige Flucherei sorgen werden. Zwar lassen sich problemlos OpenGL und die D3DX Bibliothek mischen, aber dadurch müßte für Linux erst eine entsprechende Bibliotheken geschrieben werden. Von diesem Problem abgesehen werde ich ohnehin einen Schreikampf bekommen, wenn noch mehr "Unter x läuft, unter y nur vielleicht und nur mit bestimmten Compilern"-Bibliotheken eingebaut werden.

## 5. Ein lästiges Zäunchen

Nach Anpassen der Werte in `wglAllocateMemory` (0.25 ist als obere Grenze nicht mehr im Bereich enthalten und dadurch wurden die Angaben ignoriert) läßt sich auch ein deutlicher Unterschied zwischen AGP und Video-Speicher erkennen. Bei den häufigen Schreibzugriffen ist Video-Speicher durch die bescheidenen Zugriffszeiten wesentlich langsamer. Gleichzeitig wurde die Fence-Extension eingebaut, nach einigen Experimenten aber wieder entfernt.

Sinn ist es hier, nach dem Aufruf von `DrawArrays` einen Marker zu setzen, für den geprüft werden kann, ob die Ausführung ihn schon erreicht hat. Man würde also nach dem Aufruf ein Paket zu rendern einen Marker setzen und könnte vor dem nächsten Zugriff auf den Speicher prüfen, ob dieser schon fertig gerendert wurde und gegebenenfalls abwarten. Der Unterschied war auch sofort deutlich erkennbar (bei geringer Partikelzahl Einbruch von mehreren hundert auf 33fps). Bisher wurde also offensichtlich der Speicher schon überschrieben, bevor der Inhalt fertig gerendert war. Der nächste Problemfall war, daß egal wie groß die eingebaute Verzögerung war, das letzte Paket nie fertig war, wenn das neue begonnen wurde. Der peinliche Fehler: ein `glFlush` sollte man halt doch aufrufen, wenn etwas gemacht werden soll. Indem man mitzählt, wie oft ein solcher Fall eintritt läßt sich die Größe eines Pakets auf den Aufwand abstimmen, der für die Aktualisierung der Partikel notwendig ist (auch wenn die Zeit dafür von CPU zu CPU verschieden ist und man nur schätzen kann).

Selbst jetzt war die Verwendung der Extension immer noch wesentlich langsamer, da damit ja aber ein Fehler vermieden werden sollte (Werte überschreiben, die eigentlich noch gebraucht werden), würde man ohne Fence eine falsche Darstellung erwarten (Partikel in falscher Farbe an falscher Stelle). Ein Test mit 10 Partikeln und kleiner Paketgröße (2) zeigte allerdings außer der Geschwindigkeit keinen Unterschied, so daß ich nur vermuten kann, daß die Treiber selbst schon solche Fehler verhindern.

## 6. Billboards und Krümel

Es bleibt zu Entscheiden, ob das Partikelsystem neben Billboard auch echte Geometrie darstellen können kann. Da allerdings erstes auf Pointsprites basiert und zweiteres einige Daten mehr und vor allem komplexere Physik benötigt, werden vorerst nur Billboards geplant. Ein System aus Objekten würde dann getrennt implementiert und ein Manager entscheidet welche Art von Partikelsystem angelegt wird. Dadurch können sich die beiden Arten soviel unterscheiden wie sie wollen, solange sie eine Update und Render Funktion bieten. Bei ausreichend vielen Gemeinsamkeiten läßt sich auch über eine gemeinsame Basisklasse nachdenken.

Die grundlegenden Eigenschaften sind implementiert. Um beim Berechnen der aktuellen Farbe den Aufwand klein zu halten wird lediglich die Endfarbe und die Differenz für jeden Kanal als ein Byte gespeichert. Die Startfarbe selbst spielt nach der Initialisierung keine Rolle mehr, anhand des relativen Alters des Partikels wird die Farbe interpoliert. Außerdem bewegen sich die Partikel bisher noch linear anhand ihrer vorgegebenen Startgeschwindigkeit durch den Raum. Das größte Problem wird bei den Partikelsystemen die Füllrate werden. 10000 und mehr sind als kleine Punkte keine Herausforderung, werden die einzelnen Partikel aber größer bricht die Geschwindigkeit drastisch ein. Grundsätzlich sollte jedes System also so wenig und so kleine Partikel wie für den Effekt nötig verwenden, nicht zuletzt, weil Pointsprites ein Größenlimit nach oben haben. Je größer sie von Haus aus sind, desto stärker fällt der Effekt auf, wenn man näher herankommt, weil sie nicht mehr weiter 'wachsen' wie man es erwarten würde.

Alle nötigen Variablen sind vorhanden, zufällige Startpositionen innerhalb vorgegebener Bereiche sind ebenfalls möglich. Einiges läßt sich aber nicht nur über die Variablen lösen, es muß sich also zeigen, ob verschiedene Funktionen für z.B. Initialisierung der Flugbahn eingebaut werden sollen, oder ob stattdessen spezielle Systeme mit anderen Funktionen abgeleitet werden. Bei einem Partikelspray müssen die Partikel ihre Startrichtung anders bestimmen als z.B. bei einer Explosion. Verschiedene Funktionen wären hier zwar flexibler, aber dadurch entstehen wesentlich zuviele Funktionsaufrufe. Speziell abgeleitete Systeme würden dies vermeiden, lassen sich dann aber nicht mehr so leicht kombinieren. Gleichzeitig wird eine einzelne Funktion mit vielen Alternativen sehr groß und unübersichtlich.

## 7. Schon vor Silvester Vorsätze gebrochen

Die Initialisierung ist vorübergehend doch in eine Funktion gewandert, da einzelne Partikel im Fall von endlosen System regelmäßig neu gesetzt werden müssen. Weitere Probleme ergeben sich durch die Art der Initialisierung. Für eine Explosion wird z.B. jedes Partikel sofort an einer zufälligen Startposition initialisiert, für einen anlaufenden Rasensprenger werden erst im Laufe der Zeit alle Partikel aktiv, während für einen Wasserfall die Partikel schon gleichmäßig für verschiedene Zeitpunkte (also 'mittendrin') gesetzt werden müssen.

Die eigentliche Initialisierung bleibt dabei immer gleich und ob ein Partikel schon zum Zeitpunkt  $t$  'vorgespult' wird läßt sich durch ein Flag vorgeben. Wieviele Partikel über welchen Zeitraum initialisiert werden läßt sich anhand der Anzahl der Partikel und der Lebensspanne berechnen.

Eingebaut sind jetzt auch die Möglichkeit Start-/Endfarbe und Varianz vorzugeben. Statt einem Konstruktor mit dutzenden Parametern werden nur die wichtigsten Werte im Konstruktor gefordert, während der Rest über Set-Methoden läuft. Alternativ wird es noch einen zweiten Konstruktor geben, der alle Werte aus einer Datei einliest, um so im Editor entstandene Partikelsysteme direkt einbauen zu können.

Das Partikelsystem selbst (oder anschaulicher: der Emitter) ist dabei ein ausgerichteter Punkt im Raum (sprich: er besitzt eine Transformations-matrix). Damit lassen sich Systeme leicht an andere Objekte koppeln und z.B. die Sprühhichtung drehen.

## 8. Jetzt bekommt die Sache die richtige Richtung

Für die angegebene Richtung läßt sich jetzt eine Abweichung angeben, indem in Rad die Rotation um die jeweilige Achse angegeben wird. Das Ergebnis kann dadurch zwar eckiger wirken, ist aber weniger eingeschränkt als nur eine Abweichung in alle Richtungen zuzulassen (was dann immer als Kegel enden würde).

Damit sind alle Punkte soweit umgesetzt und das System könnte eingebaut werden. Im nächsten Schritt werden jetzt bestimmte Sonderfälle umgesetzt, evtl. für besonders gebräuchliche Systeme eigene Klassen abgeleitet. Außerdem muß der Manager für die Systeme begonnen werden.

## 9. Überbezahlter Manager

Die Aufgabe des Managers beschränkt sich momentan darauf einen Vector von Systemen zu verwalten, neue Systeme zu erzeugen, vorhandene zu löschen bzw. 'ausgebrannte' zu entfernen. Um nicht bei der Initialisierung über  $x$  Schichten zu laufen liefert die NewSystem-Funktion einen Zeiger auf das erzeugte System zurück.

Außerdem wurden die ersten beiden Zustände für ein System eingeführt: aktiv und pausiert.

Peinlicher Fehler mit dem statisch für die gesamte PartikelSystem-Klasse allokierten AGP-Speicher wurde behoben. Dieser wurde im Destruktor freigegeben und auf 0 gesetzt, was logischerweise zum Absturz führte, sobald zwei Systeme gleichzeitig aktiv sind und das erste freigegeben wird.

Pünktlich zu Silvester ist das Feuerwerk fertig geworden, bisher fehlen allerdings noch Einflüsse wie Wind und Schwerkraft.

## 10. Back to the roots

Um endlich was voran zu bringen wurden ein halbes Dutzend Rechner samt Besitzer eingesperrt, bis ordentlich was erledigt wurde. Oder so ähnlich, jedenfalls war es extrem hilfreich die jeweiligen Personen direkt anwesend zu haben und Probleme gleich vor Ort lösen zu können.

Die Landschaft an sich war zwar schon grob in den Client eingebaut, hat aber alles andere als funktioniert. Kleinere Anpassungen (wird jetzt 512 oder 513 gespeichert und an welchen Stellen muß dann umgedacht werden), peinliche Fehler ( $(i > 1 \ \& \ 1) * 3$  ist eben nicht das gleiche wie  $(i > 1 \ \& \ 3)$ ) und viel Kopfzerbrechen (Patch\* statt Patch\*\* sorgt für etwas zu große Schritte beim Array durchlaufen) später war die Landschaft selbst endlich soweit. Frustum Culling war vorhanden und LOD war schmerzfrei von der isolierten Landschaft zu kopieren. Etwas mehr Fehlerbeseitigung war für das Occlusion Culling nötig, weil durch die jetzt deutlich größeren Boxen neue Probleme auftraten (im wesentlichen werden dadurch Boxen die zwar nicht sichtbar sind, aber weder komplett hinter noch komplett neben dem Frustum liegen als sichtbar betrachtet und füllen das gesamte zGrid mit einem kleinen Wert, der alles andere verdeckt). Die Notlösung ist jetzt, daß nur eine Box, die komplett vor der Kamera liegt etwas verdecken kann. Der Nachteil ist dabei, daß der Patch, in dem sich der Spieler befindet nichts mehr verdeckt. In der Praxis aber kein Verlust, da er sich auf dem Patch befindet und er deswegen ohnehin wenig bis nichts verdecken kann und der Patch direkt dahinter ja auch noch da ist.

Änderung: Das Problem bestand darin, daß hohe Boxen bei Blick nach oben komplett als 'über' der Kamera gewertet werden, obwohl sie von oben betrachten dahinter liegen. Der Fix besteht darin, zuerst das Skalarprodukt ohne y-Anteil zu berechnen, um zu überprüfen, ob der Punkt in der Ebene hinter der Kamera liegt. Der y-Anteil kann aufaddiert werden, womit kein zusätzlicher Rechenaufwand entsteht. Ist die Box in der Ebene hinter der Kamera wird sie nicht ins Raster geschrieben.

Für die Landschaft wurde außerdem ein schnellerer Test zur Sichtbarkeitsbestimmung einer Box in die Kamera eingebaut. Der Grund dafür ergibt sich durch die Nutzlosigkeit eines Quadtree bei der Landschaft. Der Overhead macht jeden Geschwindigkeitsvorteil zunichte, so daß das einfachere Brute Force Testen am Ende sogar schneller ist. Durch den Verzicht läßt sich der Test bereits abbrechen sobald eine Box teilweise sichtbar ist, was bei allen sichtbaren Boxen den Aufwand auf 1/8 reduziert.

Probleme bestehen noch auf den verschiedenen Plattformen. Zwar hat nVidia in neuen Treibern wohl einige Optimierungen vorgenommen, die bestimmte Extensions nicht mehr allzu wichtig machen (der Unterschied ist hier von 60 auf 10fps geschrumpft), aber davon haben Besitzer anderer Karten natürlich nichts. Gleichzeitig ist OpenGL nicht gleich OpenGL. Auf Mac fehlen mal eben ein paar verwendete Konstanten, um sämtliche State-Bits auf den Stack zu schieben, manche Dinge verhalten sich auf Linux anders als auf Windows und das Ergebnis ist Chaos. Windows läuft korrekt, Linux wechselt unmotiviert die Farben und Mac zeigt erst gar keine Landschaft an. Folgerung: nur Masochisten würden ohne jede Erfahrung mit einem Cross-Plattform Projekt in dieser Größe in die Spieleprogrammierung einsteigen. Spätere Praktika sollten hier allein aus Motivationsgründen erreichbare Ziele stecken, und den Schwerpunkt auf Spieleentwicklung statt Crossplattform Client/Serverentwicklung legen. Die größte Zeit geht nicht durch das Entwickeln verloren, sondern dadurch, daß jeder kleine Krümel Probleme auf Plattform X verursachen kann. Sinnvoll wird sowas frühestens, wenn ein Teil der Praktikanten sich wirklich mit der Programmierung auf jeder der drei Plattformen auskennen.

## 11. Licht und Farbe, aber nicht unter einem Hut

Der Grund für das Verschwinden einer Textur bei aktiverer Beleuchtung ist auch endlich gefunden. Das Problem besteht darin, daß die Beleuchtung die Farbwerte schon verändert, bevor Texturen ins Spiel kommen. Damit sind entweder die Farbwerte verändert oder der Alphawert auf 1 gesetzt. Da die Texturgewichte aber in einem von beidem gespeichert sein müssen ist es nicht ohne weiteres möglich gewichtete Texturen und Beleuchtungsrechnung gleichzeitig zu verwenden. Pixelshader könnten einem hier vermutlich mehr Kontrolle geben als eine handvoll vorgegebener Konstanten, aber nach dem Streß, den eine simple nVidia Extension auslöst findet sich wenig Motivation ein halbes Dutzend verschiedene Codepfade zu schreiben.

Stattdessen wird wohl doch wie geplant nur eine einfache Lightmap benutzt, auch wenn dafür mehr Textureinheiten benötigt werden (bzw. in mehreren Durchgängen gerendert werden muß, was dummerweise wohl gerade für die betroffenen Karten mit weniger als 4 Textureinheiten tödlich sein dürfte).

Umgekehrt benötigt die Beleuchtung für eine große Polygonzahl einiges an Zeit, während eine Lightmap auf moderneren Karten (bis auf den Speicher) kostenlos wäre.

Nachtrag: im Moment wird die Lichtberechnung von Hand erledigt und direkt in die Vertexfarben moduliert. Das kostet zur Laufzeit keine Zeit, keinen zusätzlichen Speicher und läßt den Alphakanal in Ruhe. Nachteil: die Farbe aufmodulieren wird zwangsweise von der dritten Textureinheit erledigt (oder alternativ im zweiten Renderdurchlauf).

## 12. Blau ist keine Farbe, sondern ein Zustand

Während der beiläufigen Entwicklung einer Simpleengine stolpert man zwangsweise über ähnliche Probleme wie im Praktikum. In dem Fall entstand dabei eine Idee für einen halbwegs effizienten Zustandsmanager (was zur Zeit ohnehin reines Chaos darstellt, solange man an keinem Zeitpunkt wissen kann, in welchem Zustand sich diese und jene Einstellung befindet und man zur Sicherheit lieber alles nochmal setzt, hinterher aber nicht weiß, was man davon rückgängig machen muß). Der Zustandsmanager speichert neben dem aktuellen Zustand für alle von ihm unterstützten Einstellungen einen Stack von bis zu 256 Differenz-Zuständen. Dabei wird eine Aufzeichnung der Änderungen gestartet, alle benötigten Änderungen gemacht und die Aufzeichnung beendet. Danach beinhaltet der Differenzzustand alle tatsächlichen Änderungen, die dann mit einem Restore rückgängig gemacht werden können. Dadurch ist sichergestellt, daß jeder Programmteil die Zustände so hinterlassen kann, wie sie zuvor waren. Durch den Stack lassen sich Änderungen sammeln und schrittweise rückgängig machen, was hilfreich wird, wenn nach benötigten Zuständen sortierte Objekte in einem Baum Depth-First durchlaufen und gerendert werden.

Komplex wird das Problem erst dadurch, daß nicht alles ein On/Off-Zustand ist und neben einem bool-Array noch mehr Konstrukte benötigt werden. Das betrifft mit etwas Glück aber überwiegend Einstellungen, die nicht ständig geändert werden (z.b. Fog, etc.)

## 13. Licht und Schattenspiele

Aus gewisser Langeweile und Neugier wurde ein wenig Schatten eingefügt. Im wesentlichen wird entlang der umgekehrten Lichtrechnung geprüft, ob das Gelände höher ist als der Lichtstrahl. Um keine häßlich abgestuften Schatten zu bekommen werden die Schnittpunkte gezählt, auf einen festen Bereich normiert und mit der Helligkeit verrechnet:  $c=c * (1-\text{Intersections})$ .

Die Startdauer wird dadurch deutlich länger, eine offline vorberechnete Light/Shadowmap wird immer angebracht.

Änderung: oder stattdessen ein paar zusätzliche Abbruchbedingungen. Den Strahl zu verfolgen, wenn er bereits höher als der Maximalwert der Heightmap ist bzw. überhaupt damit anzufangen, wenn der Punkt ohnehin kein Licht bekommt ist unsinnig. Jetzt wird der benötigte Zeitaufwand gleich wesentlich kleiner. Theoretisch sollten sich bei jedem Einfallswinkel Licht und Schatten die Waage halten, so daß sich insgesamt die benötigte Zeit nicht zu stark verändert.