Computergraphik 2 LU SS 4.0

Sommersemester 2007

LVA Nr. 186.165

3. Abgabe

Askari Albors	**0325682**	**E532**	**askari@chello.at**
Patronas Alexander	**0425487**	**E532**	**patronas@chello.at**

Inhaltsverzeichnis

1. Kurzbeschreibung / Anforderungsanalyse

1.1 Grundlegendes Gameplay

BSGTTT (Battlestar Galactica - The 13th Tribe) ist ein Weltraumshooter welches aus der FPS Sicht eines Viper-Cockpits gespielt wird. Ziel ist es im 1.Level die BSG und Colonial One solange zu beschützen bis diese ihr FTL-Drive wieder online hat um somit einen schnellen Rückzug mittels eines Jumps zu einem anderen Ort in der Galaxie zu ermöglichen.

Es sind die Battlestar-Galactica , Colonial One, zwei weitere Viper Fighter, der Spieler(Viper) selbst unter der BSG, ein Basestar(Gegner) und 3 Cylon Fighter im 1.Level anzutreffen.

Die Steuerung wurde grundlegend verbessert und nach langem Überlegen auf Physx umgestellt. Es stellte sich als große Herausforderung heraus, da wir bereits eine Matrix, Vektor und Kameraklasse vorher hatten die für alle diese Bewegungen (Player – Steuerung und andere Objekte) zuständig waren.
Doch wir konnten die Physx ziemlich gut integrieren, indem wir z.b. die Steuerung indirekter gemacht haben (nun fliegt man leicht rückwärts weiter wenn man sich schnell um 180 Grad dreht und selbst keinen Schub mehr gibt), weiters verwenden wir ContactReports um verschiedene Kollisionsabfragen abzuhandeln und auch deren Matrix, Vector Klasse ist nun bei uns in Verwendung.

Es ist möglich Raider Fighter abzuschießen, diese explodieren (Partikelengine) nach 5 Treffern.

Eine KI ist nur sehr grob vorhanden, ein Gegner reagiert auf die Gegenwart des Spielers und nimmt „Kontakt“ auf. Bis zum nächsten „Spiele-Event“ wollen wir das Handeln der Gegner von mehreren Kriterien abhängig machen.

1.2 Bewegte / Komplexe Objekte / Animierte Objekte

CYLONS (RAIDER, BASESTAR)
BATTLESTAR GALACTICA
VIPER
PLAYER(VIPER COCKPIT SICHT)

SKYBOX, welches ein texturiertes Rechteck ist

Die Objekte wurden alle mit 3D-Studio Max bearbeitet und alle Menütexturen, Skybox-Texturen, HUD Ansicht (Texturen) wurden selbstständig und eigens mit dem Programm Photoshop erstellt.

Ansatz-KI
Ein Cylon-Raider verfolgt den Spieler auf Grund seiner derzeitigen Position und Lage.

Animierte Textur als Radar

Animierte Objekte sind die **Flak Türme** auf der BSG welche extra OBJ- Models darstellen und auf der BSG richtig positioniert wurden und sich auf Grund der aktuellen Lage der BSG mit transformiert werden.

1.3 Effekte

Lens Flare

Unser Lens Flare Effekt besteht aus 3 unterschiedlich großen und färbigen Halo Texturen. Zusätzlich wird das Bild umso heller je näher man dem Zentrum der Lichtquelle kommt. Verdeckt ein Objekt die Lichtquelle so wird kein Effekt dargestellt. Die Überprüfung erfolgt mit Hilfe des Z-Buffers. Berechnet werden die Positionen der Texturen über den Up- und Forward-Vektor der Kamera. Auf Grund der verwenden 6DOF Steuerung und der 3 möglichen Rotationsarten (yaw, pitch und roll) kostete uns die Berechnung einige Nächte.

Für das Verständnis waren das „Lens Flare nehe-Tutorial“ und die Texte zum Thema „Billboards“ im Buch „More OpenGL Game Programming“ sehr vorteilhaft.

Partikelengine

Hierfür wurde eine eigene Klasse Emitter und Particle erstellt, welche auf Grund der verstrichenen Framedauer (deltaTime) neue Particle mit bestimmter pos, Lebensdauer, Textur, blending etc.. instanziert und aussendet.

Hierbei wurde im speziellen das Düsentriebwerk der Cylons, Explosionen (welche nach 5 Treffern auf die gegnerischen Raider-Raumschiffe zu sehen sind) mit der Partikelengine implementiert.

Schwierigkeiten (Blending) : Wir wollen hier nur erwähnen dass wir einige Schwierigkeiten hatten mit dem richtigen Blending aber dies nach längerer Arbeit daran gelöst werden konnte durch die Open-GL Befehle :

```
glDepthMask(GL_FALSE);
glEnable(GL_DEPTH_TEST);
glEnable(GL_BLEND);
```

Und am Ende wieder ein

```
glEnable(GL_DEPTH_TEST);
```

um die richtigen Z-Werte im Depth-Buffer für die nächsten Objekte abzufragen und abzuspeichern.

Komplexe Transparenz

Hier bei mussten wir besonder Acht geben auf die Reihenfolge der zu zeichnenden Objekte, da wir im Cockpit alles eigentlich transparent gemacht haben und auch zeitweise mit dem Alpha-Channel bei den TGA Files uns beschäftigt haben. Das HUD ist sehr komplex aber sehr übersichtlich gehalten.

Animiertes Radar

Rechts unten in der Ecke vom HUD wird das animierte 2D-Radar dargestellt, welches über eine animierte Textur implementiert wurde,

Info: Leider stellte sich die 2D Darstellung als nicht sehr brauchbar heraus für die genaue Lokalisation der Gegner. Da wir das Spiel weiterführen wollen und ausbauen wollen, möchte wir entsprechend hier ein 3D-Radar bei Gelegenheit darstellen / implementieren.

1.4 Frustum Culling

Dies wurde mittels der Referenz Lighthouse3D implementiert, eine eigene! AABBox.cpp Klasse wurde dafür erstellt und die Frustrum Klasse wurde als Ansatz für unsere Klasse verwendet und weiter ausgebaut. Die Objekte werden anhand ihrer Axis-Aligned-Bounding-Box welche jedes Frame neu berechnet wird, klassifiziert ob sie sich innerhalb des Frustrum befinden oder nicht. Dazu werden die 8 min / max Punkte des Objekts einzeln durchgetestet.

1.5 Experimentieren mit OpenGL (IM, VBO, Debug..)

Dies wurde in dem File Model.cpp bewerkstelligt. Immediate Mode, Vertex Arrays und Vertex Buffer Objects stehen zur Auswahl, als Standard haben wir uns für VBO entschieden auf Grund der Performance.

Debug Ausgaben hatten wir bereits bei der 2.Abgabe vorzulegen derzeit haben wir alles unnötige entfernt und nur die wichtigen Sachen werden angezeigt wie die Frames/sec. und die derzeitigen Einstellungen der F2-F9 Tasten. Wir haben das Help Menü (F1) und die Debug Ausgaben F2 – F10 in unser HUD integriert um so ein besseres Design bieten zu können. Die dargestellte Schrift wurde mittels Bitmap Font Builder und Photoshop erstellt. Diese einzelnen Buchstaben wurden in DisplayLists gespeichert und können dadurch schnell abgerufen werden.

1.6 Funktionierende Steuerung

Die Steuerung erfolgt über die Maus und das Keyboard.
Keyboard

W	THRUST
A	COMBAT ROLL LEFT
S	REVERSE THRUST
D	COMBAT ROLL RIGHT
Q	ROLL LEFT
E	ROLL RIGHT
SPACE	Überspringen des Videos (nur bei Wiedergabe von Videos)
ESC	Beenden des Spiels
P	PAUSE
F1	SHOW HELP
F2	SHOW FPS
F3	WIREFRAME MODE
F4	TEXTURE QUALITY
F5	MIPMAPPING QUALITY
F6	GEOMETRY MODE (VBO, IM , ..)
F7	DISPLAYLISTS
F8	FRUSTUM CULLING
F9	TRANSPARENCY
F10	DEBUG MODE (Display Status of F-Keys)

Maus

MOVE MOUSE (x-Axis)	YAW (horiz. Drehung) der Kamera/Viper
MOVE MOUSE (y-Axis)	PITCH (Neigung) nach oben bzw. nach unten
LEFT MOUSE	FIRE

2. Anforderungsanalyse – Features des Spiels

2.1 Model Loader (OBJ-Loader)

Es wurde ein eigens geschriebener OBJ- Model Loader erstellt welcher aus der Datei .obj, welches die Daten zu den Eckpunkten und den Normalen enthält und der Datei .mtl, welches die Eigenschaften zu den Materialien enthält das Model ausliest und in bestimmten selbst definierten Structures und Arrays (Vector, Verwendung von STL) abgespeichert wird.

Der geplante FBX-Loader konnte leider aus Zeitgründen nicht implementiert werden, was wir sehr bedauern.

2.2 PhysX Implementation

Dies bereitete uns starke Schwierigkeiten, da wir von Anfang unsere eigene Matrix, Vektor und Kamera Klassen verwendet haben und die PhysX „Umstellung“ erst vor ca. 2 Woche vollzogen haben. Im Nachhinhein betrachtet, ist es schade, dass wir nicht sofort von Anfang an mit der PhysX gearbeitet haben, da dies uns einiges an Nerven erspart hätte und natürlich viel Zeit.

2.3 DisplayLists / VBO / IM /VA

Dies wird unter anderem beim Model-Loader und bei der Font Darstellung genutzt, wobei hier 256 DisplayLists mittels GenList erstellt und mittels CallList abgerufen werden.

2.4 Komplexe / Einfache Objekte

CYLONS (RAIDER, BASESTAR)
BATTLESTAR GALACTICA
VIPER
PLAYER(VIPER COCKPIT SICHT)
SKYBOX, welche ein texturierter Würfel ist

MENÜOBJEKTE (Buttons)

2.5 Soundunterstützung FMODEx

Durch eine eigene Klasse wird die Soundsteuerung geregelt. Die Initialisierung und das Laden der verschiedenen Files finde hier statt, die notwendige „Soundbibliothek“ die benötigt wird fürs Spiel wird in einer MAP gespeichert, dies ermöglicht leichten Zugriff auf die Daten.

Ein Titel wird bereits im Hintergrund abgespielt und bei jedem Klick auf einen Menübutton wird ebenfalls ein Sample abgespielt.

2.6 Image-Loading (DeviL)

Dies wurde in der Klasse Texture implementiert. Anfangs war nur ein Laden von TGA Bildern (nur unkomprimierte) möglich. Auf Grund der Möglichkeit mittels DeviL verschiedene Formate zu laden wurde diese Option genutzt und die Klasse teilweise umgeschrieben bzw. um eine Funktion erweitert. Natürlich wird die Funktion ilutGL nicht verwenden sondern diese folgenden Zeilen zum Binden der Texture:

```
glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_2D, texID);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, type, width, height, 0, type, GL_UNSIGNED_BYTE,
imageData);
```

2.7 Preloader

Wir verwenden in unserem Spiel derzeit zwei Preloader. Zu Beginn des Spiels werden alle fürs Menü relevanten Daten geladen, sowie das Intro. Der zweite Preloader findet seine Anwendung während des Briefings. Dabei werden alle für das Level benötigten Daten geladen (Models, Texturen, usw.).

2.8 Menüsystem

Es ist ein übersichtliches Menüsystem erstellt worden, welches mittels GL_QUADS und verschiedenen Textures erstellt worden ist. Hintergrundmusik wird abhängig vom aktuellen Screen abgespielt und die Buttons haben verschiedene Funktionen. Wir haben sehr viel Zeit in das Menü investiert um das Ergebnis zu erhalten.

2.9 Abspielen von Videofiles (Xvid/Divx) – DirectShow

In der Klasse VideoPlayer wurden die nötigen Filter implementiert und Funktionen um ein Video im Spiel selbst abzuspielen. Es wird über das SDL Video Fenster gelegt und agiert als „Child Window“. Es kann ein beliebiges Video geladen und abgespielt werden.

Das Intro wurde mittels 3D-Studio Max erstellt und es befinden sich im Ordner data unter videos 2 verschiedenen Versionen, da wir Kompatibilitätsproblemen auf verschiedenen Rechnern festellen konnten. Die erste Version welche Standardweise aktiviert ist (Intro.avi) ist im Format 960 x 540, jedoch gab es auf einem Rechner starke Sprünge im Video beim Abspielen dieses Formates. Daher wurde auch ein Video in der Auflösung 640 x 480 beigelegt, welches normalerweise keine Probleme verursachen dürfte und somit kann man im Notfall das Video in flüssiger Darstellung gespannt verfolgen.

Ein zweites Video befindet sich bei uns noch in Produktion und war natürlich für das Spiel gedacht, wir waren bereits am Überlegen ob wir es nicht als SPOILER in das Menü integrieren sollen (also ein Button der nur das Spoiler Video abspielt), jedoch konnte dies aus Zeitgründen noch nicht zur Verfügung gestellt werden.

2.10 Beleuchtung (Licht Klasse)

Die Beleuchtung erfolgt mittels eines „**DIRECTIONAL LIGHT**“, welches in die negative x-Richtung strahlt, es vermittelt auf Grund der Beschaffenheit der Skybox den Eindruck als würde die Lichtquelle sich im Nebel befinden. Die Objekte werden richtig beleuchtet, für die OBJ Models wurden die entsprechenden Materialien im Modellierungsprogramm gesetzt.

2.11 Bitmap Font

Dies ermöglicht Debug Informationen anzeigen (FPS, F-Tasten, etc…).
Hierbei wird in der Klasse GLEngine, in welcher nur das Licht initialisiert wird und der TextureFont erstellt wird, eine Textur eingelesen und so unterteilt dass es nur noch jeweils die einzelnen Buchstaben enthält, diese werden in DisplayLists abgespeichert und können jederzeit über den Aufruf CallList abgerufen werden.

2.12 Particle Engine

Mit der wird es ermöglicht, Düsentriebwerke und Explosionen darzustellen.

2.13 Umfangreiches HUD

Über das HUD können alle wichtigen Funktionen angezeigt werden (Kommukation, Health-Anzeige) Außerdem haben wir die Hilfe und den Debug Modus in das HUD integriert. Auf dem Radar rechts unten werden feindliche Raumschiffe angezeigt.

2.14 Animiertes Radar

An dieser Stelle soll nicht unerwähnt bleiben, dass wir die Funktionalität des Radars aus der Serie besser nicht übernehmen hätten sollen. Das Prinzip mit der fehlenden Höheninformation ist schon etwas gewöhnungsbedürftig. Wir haben bei der Implementierung zum Glück darauf geachtet, dass es uns nachträglich jederzeit möglich ist das bestehende Radar ohne viel Aufwand auf ein dreidimensionales System umzurüsten (was auch noch passieren wird).

3. FAZIT

Leider funktioniert. das Spiel nur ansatzweise so wie wir es gerne hätten, denn obwohl wir sehr, sehr viel Zeit für die LVA investiert haben und uns eigentlich dieses Semester NUR auf das Spiel konzentriert haben, wurden wir nicht vollständig fertig.

Es war unser erstes größeres Projekt in C++ und OpenGL und dies hat viel Einarbeitungszeit für uns bedeutet.
Doch wir haben durch diese LVA unglaublich viel dazu gelernt, von Projektmanagement, Zeitkalkulierung, OpenGL, Grafik, Audio, Modellierung, C++ etc., etc.…

Weiters war es für uns schade, dass wir uns nicht gleich am Anfang mit der PhysX auseinandergesetzt haben, da wir damit uns sehr viel Zeit erspart hätten mit den eigens geschriebenen Klassen (Kamera, Matrix, Vektor..) und deren Testen auf Funktionstüchtigkeit.

Wir werden auf jeden Fall dieses Projekt noch weiterführen und noch sehr stark ausbauen.

4. Programmstruktur

4.1 Main

In der Main Klasse werden alle anderen wichtigen Klassen bzw. Header inkludiert und verwendet. Die Keyboard Tasten und Mausbewegungen werden hier abgefragt bzw. verarbeitet.

4.2 Render

Diese Klasse hat viele Methoden aus der damaligen MAIN übernommen,
(drawScene, drawLevel1, drawLoadingBar, drawMenü, etc..)
Hier erfolgt auch die Initialisierung und Darstellung von der 3D Viewing Pipeline.

Die Menüstruktur wird derzeit hier in der Render Klasse verwaltet, dies möchten wir bei Gelegenheit in eine eigene Klasse auslagern.

4.3 Vector3.h

Dies stellt die verschiedenen Funktionen für Vektorrechnung zur Verfügung durch Überladen der notwendigen Operatoren. Unter anderem kann das Kreuzprodukt (gibt Normale zurück der durch 2 Vektoren aufgespannten Fläche) zweier Vektoren abgefragt werden, ein Vektor normalisiert werden, auf Gleichheit überprüft werden und Skalarprodukt (wie Vektoren richtungsmäßig zu einander stehen) berechnet werden.

Update: Wird nicht mehr konsequent verwendet, stattdessen stützen wir uns auf das interne NxVec3 und NxMat34 Format der Physx

4.4 Matrix44.h

Eigens geschriebene Matrix Klasse. (inline definiert)

Update: Da wir erst sehr spät auf PhysX umgestellt haben wird diese Klasse in ganz wenigen Fällen noch benutzt.

4.5 Model

Dies ist die eigens geschriebene OBJ-Loader Klasse, welche die relevanten Daten lade, verwaltet und verwendet.

4.6 Texture

Hier können Textures mittels der Library DeviL geladen werden.

4.7 Time

Diese Klasse ermöglicht eine Game Time mittels QPC (Query Performance Counter) zu erstellen, wobei verschiedene Funktionen zur Verfügung stehen, unter anderem vergangene Zeit abfragen, DeltaTime und Startzeit seit der Instanzierung des Objekts.

4.8 VideoPlayer

Hier wird mittels DirectShow ermöglicht Divx/Xvid bzw. allgemein Avi-Dateien abzuspielen. Verschiedene Filter die nötig sind werden initialisiert und Funktionen wie Play, Stop und Load stehen zur Verfügung. Wir haben uns mit dieser Implementierung auseinandergesetzt, da wir durch Renderszenen in unserem Spiel Atmosphäre aufbauen möchten. Derzeit ist nur eines von unseren Videos zu sehen, und zwar das Intro.

4.9 Control

Diese Klasse beschreibt abstrakt die verschiedenen Funktionen die notwendig sind für die Klassen Button bzw. ListBox. Positionsangabe, Höhe, Breite und einige andere Sachen werden hier definiert respektive können diese abgefragt werden.

4.10 Button

Diese Klasse erbt die Eigenschaften der Control Klasse und erweitert diese entsprechend. Im Endeffekt werden derzeit alle Menüelemente mittels Buttons dargestellt, sogar der Hintergrund. Diese Buttons sind verschieden große texturierte GL_QUADs die unter anderem Hover- Effekt unterstützen.

4.11 Listbox

Diese Klasse wird derzeit nicht verwendet. Am Anfang wurde diese geschrieben um eine Listbox im Menü darzustellen, mit einer Auswahl und Markierung dieser Auswahl. Vorerst lassen wir diese Klasse bestehen, da die Möglichkeit besteht diese noch zu verwenden.

4.12 Light

Hier kann ein Licht mit bestimmten Lichttyp erstellt werden und daraufhin mittels verschiedener Funktionen Diffuse, Attenuation, Ambient und andere Eigenschaften des Lichts geändert werden.

4.13 HUD

Die HUD Klasse stellt unser Cockpit dar. (Radar, etc..)

4.14 GLEngine

In dieser Klasse wird derzeit nur das Licht initialisiert und die fontbase Textur welche jeden Buchstaben als Textur jeweils in einer DisplayList zur Verfügung stellt. Dadurch wird ermöglicht verschiedene Ausgaben am Bildschirm darzustellen.

4.15 FXSound

Diese Klasse beinhaltet alle relevanten Funktionen und Initialisierungen für die Unterstützung der Library FMODEx. In einer (STL) MAP Datenstruktur werden die notwendigen Samples (komprimiert im Arbeitsspeicher vollständig abgespeichert) und die Sounds (werden nur gebuffert abgespeichert) gespeichert und danach werden die erstellt und können jederzeit abgerufen werden.

4.16 Camera

Die Kamera Klasse ermöglicht verschiedene Funktionen wie unter anderem Roll, Pitch, Yaw, Move Along Side Axis, Move Along Forward Axis etc…

Update: Wird nicht mehr vollständig genutzt, sondern nur noch die Werte der PhysX werden übergeben und gespeichert für den Aufruf gluLookAt.

4.17 Physx

Diese Klasse verwaltet Initialisierungen der PhysX relevanten Daten.

4.18 AABBox

Axis Aligned BoundingBox – diese Klasse stellt mittels der min/max Werte eines Objektes seine BBox dar und wird entsprechend bei jedem Frame neu berechnet um Frustum Culling durchzuführen.

4.19 GameObject

Abstrakte Klasse nach welchem jedes Objekt in unserem 3D Spiel definiert und abgeleitet ist.

4.20 Battlestar : GameObject

Battlestar Galactica

4.21 BattlestarGUN : GameObject

BatllestarGalactica Flak Türme

4.22 Colonial One : GameObject

Colonial One Raumschiff welches zum Konvoi der BSG gehört.

4.23 Emitter

Emitter Klasse wird verwendet für unserer PartikelEngine.

4.24 ERaider : GameObject

Cylon Raider (Gegner)

4.25 Explosion

Verwaltete das “Abspielen” einer mittels der ParticleEngine erstellten Explosion.

4.26 Frustum

Wie der Name schon sagt, finden sich hier alle relevanten Methoden die zur Berechnung des Frustum Cullings benötigt werden.

4.27 Level1

Level1 enthält alle nötigen Initialisierungen der Objekte die für den Level benötigt werden und auch andere Methoden.

4.28 Particle

Stellt ein Particle in seiner reinsten Form dar (Position, Geschwindigkeit, etc…)

4.29 Plane

Dient für den Aufbau des Frustum Bereichs.

4.30 Projectile : GameObject

Einzelne Projectile werden in einer list verwaltet und dienen als Objekte für den Spieler als PrimärWaffe und für die Battlestar Galactica als Flak-Feuer.

4.31 PrimaryW

Hier werden die einzelnen Schüsse der Viper (Player) verwaltet und gezeichnet.

4.32 Viper : GameObject

Viper Objekt, aus diesem wird auch der Player erstellt.

5. LIBRARIES

5.1 DeviL 1.6.8-rc2

http://openil.sourceforge.net/download.php

5.2 FMODEx 4.06

http://www.fmod.org/

5.3 DirectShow (Platform SDK - Windows Server 2003 R2 Platform SDK)

http://www.microsoft.com/downloads/details.aspx?familyid=484269E2-3B89-47E3-8EB7-1F2BE6D7123A&displaylang=en

5.4 PhysX SDK

http://www.ageia.com/

6. EXTRAS

6.1 FONT BUILDER

LMNOpc BITMAP FONT BUILDER
http://www.lmnopc.com/bitmapfontbuilder/

6.2 SDL

SDL-1.2.11
http://www.libsdl.org/

6.3 Modellierungsprogramm + Models

Die Models wurden alle überarbeitet mittels Photoshop und 3dStudio-Max. Großteils wurden diese auch verändert.

www.scifi-meshes.com

7. BÜCHER

- Beginning OpenGL Game Programming
- More OpenGL Game Programming“
- 3D Math Primer for Graphics and Game Development (Fletcher Dunn and Ian Parberry)
- C++ von A bis Z (Jürgen Wolf)
- OpenGL Programming Guide (Fifth Edition, Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis)