

Dokumentation Softwarepraktikum “Rendering relativistischer Effekte mit Vertex Shadern”

4. Februar 2003

Inhaltsverzeichnis

1	Einleitung	1
2	Lorentz Transformation	1
3	Dateiformat zur Beschreibung der Szenen	2
3.1	Murray-Tag	2
3.2	Szene-Tag	3
3.2.1	Objekt-Tags	3
3.2.2	Light-Tag	4
4	Aufbau des “Viewers”	4
4.1	Grundaufbau	4
4.2	GLFrame	5
4.3	Camera	5
4.4	Objekte	5
4.4.1	Zylinder	6
4.4.2	Kegel	6
4.4.3	Kugel	7
4.4.4	Quader (Box)	7
4.4.5	Ebene	7
4.5	Parser	7
4.6	Mathematische Grundklassen	8
5	Abschließende Worte	8

1 Einleitung

Ziel des Softwarepraktikums war es, eine dreidimensionale Szene in OpenGL mittels der Vertex-Shader-Extension anzuzeigen, durch die man sich gerade mit hoher Geschwindigkeit bis hin zu Lichtgeschwindigkeit bewegt. Dabei treten relativistische

Effekte auf, z.B. Änderungen der Form der dargestellten Objekte, oder auch Farbveränderungen durch den Doppler Effekt. Wir haben uns hier allerdings auf die korrekte Darstellung der Form beschränkt, der Dopplereffekt wurde von uns vernachlässigt, kann aber implementiert werden. Durch das Rendering mit OpenGL wurde ein polygonbasierter Ansatz für das Rendering verwendet, welcher so hohe Darstellungsraten ermöglicht, daß interaktiv mit dem Programm gearbeitet werden kann. Dazu wurden die Objekte in der Szene als Polygon-Mesh-Modelle repräsentiert, deren Auflösung angepasst werden kann. Ab der Geforce 1 von nVidia wurden Vertex-Shader in Grafikkarten implementiert, die es erlauben, für jeden ausgegebenen Vertex ein kleines Programm mit assembler-ähnlichen Befehlen auf der Hardware laufen zu lassen, um bestimmte Eigenschaften des Vertex zu verändern, z.B. die Position oder die Farbe. Dieses Vertex-Programm wurde benutzt, um die Lorentz Transformation umzusetzen.

2 Lorentz Transformation

Diese Transformation ist ein Mapping zwischen 2 Koordinatensystemen, welche sich mit einer konstanten Geschwindigkeit relativ zueinander bewegen. Dabei treten die o.g. relativistischen Effekte auf. Gegeben seien 2 Koordinatensysteme K (Weltkoordinatensystem) und K' (Betrachterkoordinatensystem), wobei sich K' von K mit einer konstanten Geschwindigkeit v entlang der positiven x -Achse entfernt. Die Lorentz-Transformation, um einen Punkt P in K nach P' in K' zu transformieren:

$$P'_x = \gamma(P_x + vt), \quad P'_y = P_y, \quad P'_z = P_z, \quad t' = \gamma\left(t + \frac{v}{c^2}x\right)$$

wobei $\gamma = \frac{1}{\sqrt{1-\beta^2}}$ und $\beta = \frac{v}{c}$, c ist die Lichtgeschwindigkeit. Der zeitliche Zusammenhang zwischen der Emission eines Photons ct_e und dessen Beobachtung ct_o in K stellt sich wie folgt dar:

$$(ct_o - ct_e) = \sqrt{(x_e - x_o)^2 + (y_e - y_o)^2 + (z_e - z_o)^2}$$

wobei die Raumzeitkoordinaten des Emissions-Ereignisses (x_e, y_e, z_e, ct_e) und des Beobachtungs-Ereignisses (x_o, y_o, z_o, ct_o) sind. Mit der Lorentz-Transformation können dann die Koordinaten des Emissions-Ereignisses in K' berechnet werden. Dabei ist die Zeitkomponente der Emission in K' für das Rendering irrelevant, und die Raumkoordinaten der Emission in K' legen die Richtung fest, aus der das Licht kommt. Folgende Matrix führt die Transformation durch:

$$\begin{pmatrix} (\gamma-1)n_x^2 + 1 & (\gamma-1)n_x n_y & (\gamma-1)n_x n_z & -\beta\gamma n_x \\ (\gamma-1)n_x n_y & (\gamma-1)n_y^2 + 1 & (\gamma-1)n_y n_z & -\beta\gamma n_y \\ (\gamma-1)n_x n_z & (\gamma-1)n_y n_z & (\gamma-1)n_z^2 + 1 & -\beta\gamma n_z \\ -\beta\gamma n_x & -\beta\gamma n_y & -\beta\gamma n_z & \gamma \end{pmatrix}$$

Dabei ist $n = (n_x, n_y, n_z)$ die normalisierte Richtung, in die sich der Betrachter bewegt, γ und β wie oben. Diese Transformationsmatrix darf nur angewendet werden, wenn sich der Ursprung von K an der Position des Beobachters befindet, daher muss eventuell vorher noch eine Translation durchgeführt werden, welches im Falle unserer Applikation jedoch nicht nötig war.

3 Dateiformat zur Beschreibung der Szenen

Die Szenen die vom Viewer dargestellt werden können, müssen im XML Format vorliegen. Die XML Szenen können die folgenden Tags beinhalten und beginnen dabei mit folgendem Header:

```
<?xml version="1.0" ?>
<!DOCTYPE murray SYSTEM "murray.dtd">
```

3.1 Murray-Tag

Alle für die Szene nötigen Daten wie Lichtquellen, Kamera und die Objekte müssen innerhalb des Murray-tags stehen, das mit `<murray>` eingeleitet und mit `</murray>` geschlossen wird. Innerhalb des Tags sind folgende zwei Möglichkeiten zulässig.

Kamera-Tag Das Tag `<camera />` hat die folgenden Parameter:

- `pos="x y z"` ist die Position der Kamera
- `dir="x y z"` ist die Richtung in die die Kamera zeigt
- `up="x y z"` ist ein Vektor, der von der Kamera aus nach oben zeigt. Er stellt sozusagen die Drehachse dar
- `fovy="a"` ist der Öffnungswinkel der Kamera

3.2 Szene-Tag

Innerhalb des Szene-Tags (`<scene>` — `</scene>`) können folgende Tags vorkommen:

Background Das Tag `<background r="a" g="b" b="c">` gibt die Farbe des Szenenhintergrundes an, wobei r, g und b für die Farbwerte Rot, Grün und Blau stehen.

t-Tag Das Tag `<t ...>` `</t>` wird dazu verwendet, alle vom t-Tag eingeschlossenen Daten zu transformieren. Dazu stehen folgende Möglichkeiten zu Verfügung:

- `translation="x y z"` verschiebt den Inhalt des t-Tags um den übergebenen Vektor.
- `rotation="x y z a"` nimmt eine Drehung vor, wobei mit x, y und z jeweils die Drehachse bestimmt wird und mit a der Winkel angegeben wird.
- `scaling="x y z"` streckt die betroffene Szene um den im Vektor angegebenen Faktor.

Das t-Tag darf in beliebiger Reihenfolge ineinander verschachtelt werden.

3.2.1 Objekt-Tags

Shape-Tag Innerhalb des Shape-Tags ist entweder wieder ein Shape-Tag zulässig oder eines der Objekt-Tags.

Sphere-Tag Das Tag `<sphere radius="r" q="a"/>` erstellt ein Kugel mit dem Radius r und einer Dichte an Dreiecken von a .

Plane-Tag Das Tag `<plane p="x y z" x="x y z" y="x y z" q="a"/>` erstellt eine Fläche, die aus den Vektoren x und y aufgespannt wird und ihren Mittelpunkt bei p hat. Mit q wird wie bei allen Objekten die Dichte an Dreiecken angegeben.

Box-Tag Das Tag `<box s="x y z" q="a"/>` erstellt ein Quader mit den Seitenlängen x , y , z und der Dichte q .

Cone-Tag Das Tag `<cone radius="r" height="h" q="a"/>` erstellt einen Kegel mit dem Radius r und der Höhe h .

Zylinder-Tag Das Tag `<cylinder radius="r" height="h" q="a"/>` erstellt einen Zylinder mit dem Radius r und der Höhe h .

Material-Tag Mit den Tags `<material> <diffuse r="a" g="b" b="c"/> </material>` lässt sich für die innerhalb des Shape-Tags stehenden Objekte die Farbe mit den Farbwerten r , g und b angeben.

3.2.2 Light-Tag

Das Light-Tag (`<light type="..." .../>`) gibt eine Lichtquelle in der Szene an. Zulässige Parameter sind:

- `type="..."`, wobei mögliche Typen *global*, *positional* und *spot* sind.
- `r="a"`, `g="b"` und `b="c"` als die Farbwerte Rot, Grün und Blau (müssen zwischen 0 und 1 liegen)
- `pos="x y z"` gibt die Position der Lichtquelle an (nur für *positional* und *Spot*)
- `dir="x y z"` gibt die Richtung an, in die die Lichtquelle strahlt (nur für *Spot*)
- `angle="a"` gibt den Abstrahlwinkel der Lichtquelle an (nur für *Spot*)

4 Aufbau des “Viewers”

Der Viewer der im Zuge dieses Praktikums erstellt wurde, basiert auf der QT-Library von Trolltech, welche mindestens in der Version 2.2 vorhanden sein sollte. Während der Programmierung wurde die Version 2.23 der QT verwendet, Kompatibilitätstests mit anderen Versionen der QT wurden von uns nicht durchgeführt, aufgrund der Einfachheit des Applikationsaufbaus gehen wir jedoch davon aus, daß es sich auch mit anderen QT-Versionen kompilieren lässt, solange diese QGL unterstützt. Als Betriebssystem wurden SuSE Linux mit den Versionen 7.2(i386) sowie 8.0(i386) benutzt, daß

Projekt kann per Makefile mit gcc kompiliert werden. Für die Portierung auf andere Betriebssysteme müssen die Bibliotheken im Verzeichnis *other* für dieses Betriebssystem vorhanden sein (im Web: www.xmlsoft.org für libxml und sourceforge.net/projects/libxmlplusplus für libxml++). Wichtig ist, daß die installierte OpenGL Implementierung mindestens die Version 1.3 hat, und die NV_VERTEX_PROGRAM Extension unterstützt, welche von nVidia eingeführt wurde. Desweiteren werden zum Einlesen der Szenen-Dateien, welche im XML-Dateiformat kodiert sind, die Bibliotheken libxml und libxml++ benötigt.

4.1 Grundaufbau

Das Hauptfenster der Applikation ist in den Dateien *appwin.qt.cc* und *appwin.qt.h* implementiert. Die Klasse *AppWin* ist von der QT-Klasse *QMainWindow* abgeleitet und benutzt Standard-QT-Widgets um ein Menü, eine Toolbar und die aktuelle Sicht in die Szene in einem Objekt der Klasse *GLFrame* darzustellen. Über das Menü sind die Funktionen zum Laden einer Szene und zum Beenden des Programms erreichbar. Die Toolbar implementiert einen *QSlider* welcher zur Einstellung der aktuellen Bewegungsgeschwindigkeit benutzt werden kann, sowie eine *QCheckbox* welche zum Feststellen der aktuellen Bewegungsrichtung benutzt werden kann.

4.2 GLFrame

Das *GLFrame* Objekt aus *glframe.qt.cc* und *glframe.qt.h* ist das zentrale Objekt der Applikation und wird verwendet um die Welt aus der Sicht des Betrachters darzustellen. Es enthält dazu Datenstrukturen, um eine Repräsentation der Szene im Speicher zu halten. Dies sind eine Kamera, eine Liste von Lichtquellen (verschiedene Lichttypen siehe *light.h*) und eine Liste der in der Szene vorhandenen Objekte (diese sind in *scene.cc* und *scene.h* implementiert). Um eine Szenendatei zu laden, wird ein Objekt der Klasse *Parser* instanziiert, welches eine XML-Szene lädt und in die interne Repräsentation überträgt. Ausserdem werden hier die für den Vertex-Shader notwendigen Initialisierungsschritte vorgenommen, d.h. es wird ein Vertex-Programm erstellt und auf die Grafikkarte geladen. Die Darstellung der Szene wird mit der Methode *paintGL* ausgelöst. Diese ruft nach einigen Initialisierungsschritten zunächst die *performGLtasks*-Methode der Kamera und danach die *performGLtasks*-Methoden der einzelnen Objekte nacheinander auf. Mit der Maus können Kamera-Position und Blickrichtung eingestellt werden. Dabei kann mit gedrückter linker Maustaste durch Bewegung der Maus die Position der Kamera nach links/rechts/oben/unten verschoben werden. Mit gedrückter mittlerer Maustaste ist es möglich, sich vorwärts bzw. rückwärts zu bewegen. Mit gedrückter rechter Maustaste kann schliesslich die Blickrichtung angepasst werden. Ist die Bewegungsrichtung anhand der Checkbox festgestellt, sind die Funktionen der linken und mittleren Maustaste deaktiviert, und der Benutzer kann sich nur noch umsehen, allerdings folgt die Bewegungsrichtung dann nicht der Kamera.

4.3 Camera

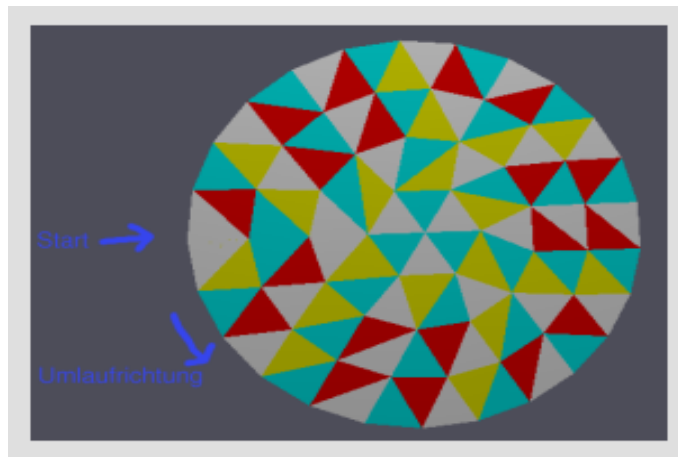
Das *Camera* Objekt aus *camera.cc* und *camera.h* hält die Daten für aktuelle Kamera-Position, Blickrichtung, Up-Vektor und Beobachtergeschwindigkeit. Dabei ist es über die QT eigenen Signal/Slot-Mechanismen mit dem Geschwindigkeitsslider verbunden, und erhält Nachricht, sobald der Benutzer die Geschwindigkeit verstellt, wodurch dann ein Update der Darstellung im GLFrame erfolgt. Die Methode *performGLtasks* sorgt dann für eine Umsetzung der Kamera-Parameter in OpenGL-Befehle. Dabei wird auch anhand der aktuellen Beobachtergeschwindigkeit die o.g. Lorentz-Matrix berechnet, und in den Parameter-Speicher des Vertex-Programmes übertragen.

4.4 Objekte

Alle Objekte, die in der Szene vorkommen, sind von einem gemeinsamen Grundobjekt *Entity* abgeleitet. Direkt von *Entity* abgeleitet sind die Objekte *Camera* und die verschiedenen Licht-Objekttypen, also alle Objekttypen, die kein Volumen im Raum ausfüllen, sondern nur als Punkt repräsentiert werden. Das von *Entity* abgeleitete Objekt *Object* ist Grundbaustein für alle *Entity*s die ein bestimmtes Volumen ausfüllen, und somit auch gerendert werden können. Davon sind dann wiederum alle Objekte ableitbar. Um optimale Darstellungsqualität zu erreichen, müssen alle Objekte aus möglichst vielen Vertices und damit Dreiecken bestehen. Für jedes Objekt kann in der Szenendatei ein Parameter *q* angegeben werden, der die gewünschte Auflösung dieses Objektes festlegt. Je kleiner *q*, desto feiner die Auflösung. Wird *q* nicht angegeben, so geht das Programm von einem Standardwert von 1.0 aus. Jedes Objekt hat eine Methode *performGLtasks*, die die Darstellung dieses Objektes mit OpenGL übernimmt.

4.4.1 Zylinder

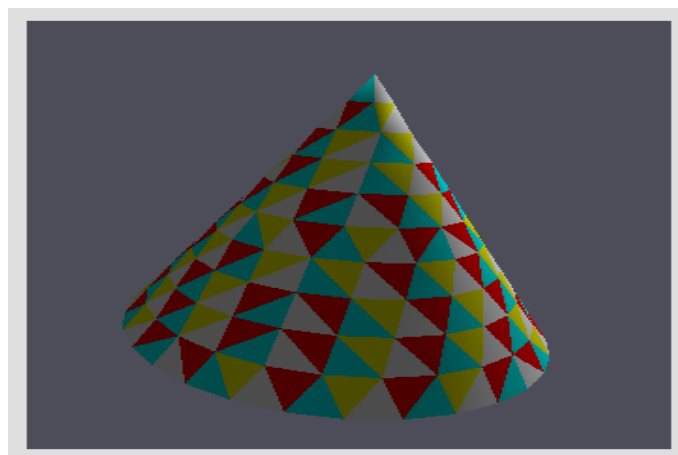
Der Zylinder (*cylinder.cc* und *cylinder.h*) besteht aus zwei Deckeln und dem restlichen Mantel. Die Deckel werden als erstes in Kreise mit der entsprechenden Dichte aufgeteilt. Zwischen jeweils zwei der Kreise werden dann die Dreiecke gespannt, wobei natürlich darauf zu achten ist, daß auf dem jeweils inneren Kreis weniger Vertices sind.



Der Mantel wird Vertikal in einzelne Ringe aus Dreiecken aufgeteilt. Man muß nur beachten, daß die Vertices der aneinanderliegenden Ringe extakt übereinstimmen, um Lücken und Überschneidungen zu vermeiden. Besonders wichtig ist es natürlich auch darauf zu Achten, daß die Vertices der Außenhülle mit den des Deckels und Bodens übereinstimmen.

4.4.2 Kegel

Der Kegel (*cone.cc* und *cone.h*) besteht aus dem Boden und dem oberen Teil des Mantels, dem eigentlichen Kegel. Der Boden wird wie beim Zylinder durch aufteilen in Ringe, welche dann mit Dreiecken ausgefüllt werden, erstellt. Der Obere Teil des Kegels wird im Prinzip genau gleich generiert, nur findet die Unterteilung nicht in der Ebene statt, sondern jeder der einzelnen Kreise wird auf der Y-Achse nach oben geschoben. Die Dichte der Kreise muß natürlich angepasst werden, da durch die Verschiebung eine größere Fläche abgedeckt werden muß als es beim Boden der Fall ist.



4.4.3 Kugel

Um die Kugel (*sphere.cc* und *sphere.h*) möglichst gleichmäßig mit Dreiecken zu überziehen, wurde auf eine Aufteilung von Breiten- und Längengrade verzichtet und statt dessen eine Implementierung gewählt, die einen Ikosaeder rekursiv so lange weiter unterteilt, bis die gewünschte Seitenlänge erreicht ist.

4.4.4 Quader (Box)

Der Quader (*box.cc* und *box.h*) besteht aus 6 begrenzt großen Flächen, die sich einfach mit jeweils zwei FOR-Schleifen in Dreiecke unterteilen lassen. Man muß allerdings darauf Achten, daß die Vertices von jeweils zwei Flächen an den Kanten übereinstimmen.

4.4.5 Ebene

Die Ebene (*plane.cc* und *plane.h*) ist einfach ein Fläche, die eigentlich unendlich groß sein sollte. Weil aber eine solche Fläche für den Vertex-Shader in ein Netz aus Dreiecken unterteilt werden muß, werden bestimmte Ausmaße festgelegt. Eine Fläche fester Größe lässt sich nun einfach in der gewünschten Dichte in ein gleichmäßiges Netz von Dreiecken unterteilen.

4.5 Parser

Der Parser (*parser.cc* und *parser.h*) sorgt für eine Übersetzung der XML-Szenendateien in der interne Speicherformat. Dabei müssen Änderungen, wie Hinzufügen neuer Objekte oder Parameteränderungen direkt hier implementiert werden. Sollten während des Parsens der Szene Fehler auftreten, wird eine entsprechende Fehlermeldung ausgegeben, das Laden der Szene jedoch nicht abgebrochen.

4.6 Mathematische Grundklassen

Wichtige Klassen für Vektor- und Matrizenrechnung finden sich in *v3d.h*, *v4d.h* und *m4x4.h*. Die Klassen für drei- bzw vierdimensionale Vektoren sind "operator-overloaded", und erlauben direkte Vektoradditions- und Multiplikationsoperationen mit einem Skalar. Wichtige Funktionen wie Normierung, Kreuzprodukt und Skalarprodukt sind ebenfalls vorhanden.

5 Abschließende Worte

Diese Dokumentation soll einen Überblick darüber geben, welche Quelldateien welche Funktionen haben. Für eine genauere Analyse des Quellcodes haben wir noch eine Doxygen-Dokumentation beigelegt, welche den Quellcode en detail beschreibt. Diese befindet sich im Unterverzeichnis *doku/doxygen/index.html*. Im Verzeichnis *data* befinden sich einige Testszenen, die wir während der Entwicklung erstellt haben. Durch

die einfache Szenenbeschreibungssprache ist es möglich sowohl mit Skripten als auch von Hand schnell eigene Szenen zu erstellen.

Das gesamte Projekt ist modular erweiterbar um weitere Objekte, dafür muss einfach eine Ableitung vom Typ *Object* erfolgen, ausserdem muss das Objekt noch in *parser.cc* eingetragen werden. Ausserdem könnte noch an der Geschwindigkeitsoptimierung gearbeitet werden. Momentan ist es so, daß mit jedem Aufruf einer *perform-GLtasks*-Methode die komplette Objektgeometrie neu berechnet und dann an OpenGL weitergeleitet wird. Dies kann bei sehr grossen bzw. sehr fein aufgelösten Szenen zu Geschwindigkeitseinbußen führen. Ein Ansatz wäre, die Objektgeometrien einmal beim Start (z.B. im Objektkonstruktor) zu berechnen und in Vertexlisten zu speichern. Diese könnten dann ohne viel Aufwand an Performance schnell gezeichnet werden. Nichtsdestotrotz werden auch im jetzigen Zustand der Applikation interaktive Frameraten erreicht.

Die Programmierung dieses Softwarepraktikums hat uns faszinierende Einblicke in die 3D-Programmierung mit OpenGL, Benutzung der Vertex-Shader sowie die sich der täglichen Wahrnehmung entziehenden relativistischen Effekte bei sehr hohen Bewegungsgeschwindigkeiten gegeben. Deshalb möchten wir uns hiermit auch noch einmal ganz herzlich bei unserem netten, sehr geduldigen Tutor, Dr. Daniel Weiskopf, bedanken - Axel Niese, Gernot Saul und Martin Schrodt.