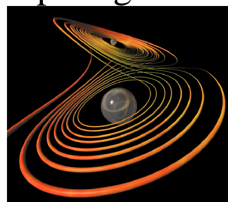
GL*ego*-Viewer

Qualitativ hochwertige Darstellung von LEGO-Modellen in Echtzeit mit OpenGL

Studienarbeit, Tomas Neumann 2001
Betreuer: Sven Havemann

Technische Universität Braunschweig
Institut Computergrafik



Inhaltsverzeichnis

1 Einleitung **2**
1.1 DISCLAIMER 2
1.2 ZUSAMMENFASSUNG 2
1.3 MOTIVATION 2
1.4 ZIEL VON GLEGO-VIEWER 3
1.5 AUFBAU DER ARBEIT 3
2 Überblick von LDraw **4**
2.1 DAS DAT-FORMAT 4
2.1.1 Zeilen-Typen *4*
2.1.1.1 0-Zeile : Kommentar oder Meta-Befehl 4
2.1.1.2 1-Zeile : Referenz auf Baustein / Teilstück 4
2.1.1.3 2-Zeile : Linie 5
2.1.1.4 3-Zeile : Triangle 5
2.1.1.5 4-Zeile : Quadrilateral 5
2.1.1.6 5-Zeile : optionale Linie 5
2.2 FARBEN IM DAT-FORMAT 6
2.3 BEISPIEL DAT-DATEI 6
3 Konzept **7**
3.1 GEPLANTE FEATURES 7
3.1.1 Darstellung: *7*
3.1.2 Optimierung: *7*
3.1.3 Schatten: *7*
3.1.4 Handhabung: *7*
3.2 AUFTRETENDE SCHWIERIGKEITEN 8
3.2.1 Orientierung von Flächen *8*
3.3 INTERNE TECHNIKEN 9
3.3.1 Partmap – Der Cache von Teilstücken *9*
3.3.2 Die Struktur der gespeicherten Daten *9*
3.3.3 Verschachtelte Display-Listen *9*
4 Implementierung **10**
SCHEMATISCHER AUFBAU 10
4.2 DIE KLASSE CLEGOMODEL 11
4.3 DIE KLASSE CFILEDAT 11
4.3.1 Speichern der Informationen der DAT-Datei *11*
4.3.1.1 Kommentar 11
4.3.1.2 Referenz auf Teilstück 11
4.3.1.3 Linie 12
4.3.1.4 Transparente und solide Dreiecke und Vierecke 12
4.4 PARTMAP 12
4.5 ABLAUF ZUR VERWENDUNG DER KLASSEN 12
4.5.1 Schritt 1 – Einlesen *12*
4.5.1.1 Parsen 12
4.5.2 Schritt 2 – Aufbereitung *13*
4.5.2.1 Das Füllen der Displaylisten 13
4.5.3 Schritt 3 – Darstellung *13*
4.6 WEITERE DETAILS 14
4.6.1 Texturen des Untergrundes *14*
4.6.2 Fit-View *14*
4.6.3 Konfiguration der INI-Datei *15*
4.7 GRAFISCHE OBERFLÄCHE 15
5 Ergebnisse und Vergleich **16**
5.1 BERECHNUNGSZEITEN: 16
5.2 GRAFISCHER VERGLEICH 17
6 Ausblick **19**
7 Zusammenfassung **19**
8 Quellenverzeichnis **20**

1 Einleitung

1.1 Disclaimer

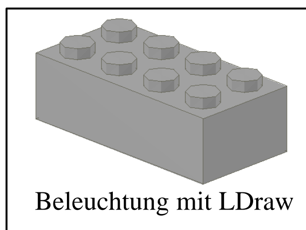
LEGO® is a registered trademark of the Lego Group, which does not sponsor, endorse, or authorize this project.

1.2 Zusammenfassung

Im Internet gibt es mehrere CAD-Programme, mit deren Hilfe man LEGO-Modelle konstruieren kann. Diese Modelle werden in einem speziellen Dateiformat spezifiziert, dem sogenannten *DAT-Format*[1] (siehe 2.1). Der GL*ego*-Viewer stellt in hoher grafischer Qualität LEGO-Modelle dar, die durch diese CAD-Werkzeuge konstruiert wurden und vorliegen. Die grafische Ausgabe geschieht durch *OpenGL*[2]. Der Benutzer kann sich somit schnell ein recht realistisches Bild des Modells machen, Lichtverhätnisse während der Laufzeit veränderen, Schattenwurf an/ausschalten und eine Untergrundtextur varieren.

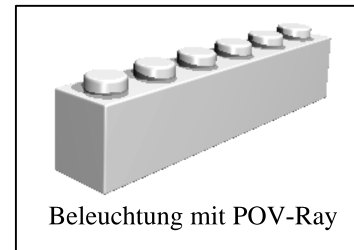
1.3 Motivation

LEGO gehört seit seiner Gründung vor fast 70 Jahren zu den bekanntesten Spielzeug-Marken weltweit. Seit ca. 45 Jahren haben die kleinen bunten Plastiksteinchen Kinder und Erwachsene zu Hunderttausenden fasziniert. Durch das Internet haben sich viele Liebhaber zusammengefunden und bilden eine kreative Plattform mit innovativen Ideen und Ansätzen. Durch die recht hohen Anschaffungskosten der vielen verschiedenen *Bausteine* entstand das Bedürfnis, LEGO-Modelle virtuell und somit kostengünstig konstruieren zu können. *LDraw*[3] ist eines der bekanntesten CAD-Programme, mit dem man kostenlos Modelle bauen kann. Es wurde ein spezielles Datei-Format spezifiziert und jeder einzelne LEGO-Baustein wurde und wird weiterhin darin abgebildet. Die vielen LEGO-Bausteine werden darin in ihre Komponenten aufgeteilt. Bis jetzt sind ca. 1.650 Bausteine und deren *Teilstücke* für LDraw erfasst.



Beleuchtung mit LDraw

Die interne Darstellung der Modelle in LDraw ist sehr einfach gehalten, ohne aufwendige Beleuchtungsberechnung. Um die LEGO-Modelle realistisch darzustellen, wurde ein Export zu *POV-Ray*[4] entwickelt. Dies ist ein klassischer *Raytracer*, welcher durch Strahlenschuß eine 3D-Szene darstellt und längere Zeit benötigt, um ein Einzelbild zu berechnen. Diese Darstellung ist statisch und wenn z.B. die Kameraposition auch nur leicht verändert werden soll, muß das gesamte Einzelbild neu berechnet werden. Eine interaktive Darstellung, bei der man in Echtzeit die Kameraposition verändern kann, bietet folglich einen höheren Komfort zur Betrachtung der LEGO-Modelle, auch wenn es Einschränkungen gegenüber der Qualität der optischen Darstellung des Raytracing gibt. Der GL*ego*-Viewer soll eine qualitativ hochwertig Darstellung der 3D-LEGO-Modelle in Echtzeit bieten.



Beleuchtung mit POV-Ray

1.4 Ziel von GLego-Viewer

Der GL*ego*-Viewer soll eine qualitativ hochwertige Darstellung der LEGO-Modelle in Echtzeit ermöglichen. Die Modelle, die Bausteine und auch die Teilstücke, zu denen immer wieder neue hinzukommen, liegen im DAT-Format vor. Dieses Format spezifiziert Linien, Dreiecke, Vierecke und Referenzen auf Bausteine oder Teilstücke in verschiedenen Farben. Das Dateiformat beinhaltet jedoch keinerlei Angaben zu Normalen, Materialeigenschaften oder Nachbarschaftsinformationen der Polygone. Trotzdem soll eine möglichst realistische Beleuchtung erreicht werden, indem sogenannte Highlights, verschiedene Helligkeiten und berechnete Schatten die Szene aufwerten. Beeinhalten die Modelle z.B. Fenster oder Windschutzscheiben, so sollen diese transparenten Bausteine korrekt dargestellt werden. Um den Realismus der Darstellung zu erhöhen, soll wahlweise eine Grundfläche eingeblendet werden, welche die Orientierung erleichtert, auf der das Modell wie auf einer Bodenfläche ruht. Dieser Boden soll mit einer Textur belegt werden, z.B. Holz, Stein, Asphalt, etc. Das Betrachten des Modells in Echtzeit soll gewährleistet werden. Je nach verwendeter Hardware (CPU, Grafikkarte) soll eine flüssige Rotation auch bei komplexen Modellen möglich sein. Der Betrachter soll die Möglichkeit haben, das Modell von beliebigen Kamerapositionen aus zu betrachten und jederzeit eine Darstellung der aktuellen Szene sehen können. Den Nachteil des Raytracing, langwierig statische Einzelbilder zu generieren, soll der GL*ego*-Viewer aufheben, sich aber an der optischen Qualität des Raytracers orientieren. Um dies zu erreichen, muß die Geometrie mit den korrekten Farbwerten an OpenGL übergeben werden. Ausgehend von dem DAT-Format sind auch andere Projekte möglich, z.B. geometrischer Netzaufbau, generative Modeller, Editoren für kleine oder sehr große Modelle. Der Schwerpunkt dieser Arbeit liegt eindeutig darin, die LEGO-Modelle interaktiv und in hoher optischen Qualität betrachten zu können.

1.5 Aufbau der Arbeit

Im ersten Kapitel erfolgt die Einführung in das Thema, die Motivation, das angestrebte Ziel des Projekts und die Einführung relevanter Ausdrücke.

Im zweiten Kapitel wird die Ausgangssituation näher beschrieben. Es wird aufgeführt, wie innerhalb des DAT-Formats Geometrie oder Teilstücke referenziert werden.

Im dritten Kapitel wird auf das Konzept des Projekts eingegangen. Schon in der Planungsphase sind mögliche Optimierungen bekannt. Es wird beschrieben, warum und über welche Eigenschaften der GL*ego*-Viewer verfügen soll.

Im vierten Kapitel wird die erfolgte Implementation erklärt. Es wird beschrieben, wie GL*ego*-Viewer intern aufgebaut ist, welche Datentypen zur Speicherung der Daten dafür entwickelt wurden und welche Techniken zur Darstellung angewandt wurden.

Im fünften Kapitel wird das Ergebnis der Arbeit, primär also die grafische Ausgabe des GL*ego*-Viewers, präsentiert und mit den grafischen Ausgaben von LDraw und dem Raytracer POV-Ray verglichen.

Im sechsten Kapitel werden mögliche Verbesserungen und Vorschläge für eine Weiterentwicklung des GL*ego*-Viewers aufgeführt.

Im siebten Kapitel erfolgt eine kurze Zusammenfassung des Projekts und eine Einschätzung, ob und in welchem Maße die Ziele des Projekts erreicht wurden.

2 Überblick von LDraw

Die meisten der computerunterstützten Konstruktionsprogrammen für LEGO-Modelle (u.a. MLCad, LeoCAD , BLockCAD..) setzen als *Frontend* auf LDraw auf und benutzen dessen Grafik-Routinen und Darstellung. Wie bei der Mehrzahl der Programme basieren die LDraw-Modelle auf dem DAT-Format, welches im Folgenden genauer erklärt wird. Mit LDraw ist es möglich sehr schnell und angenehm eigene Modelle zu generieren, unter allen bisher von LEGO veröffentlichen Bausteinen und deren Teilstücke (ca. 1.650 Stück) auszuwählen, zu plazieren, zu rotieren, zu skalieren und zu kolorieren. Das zugehörige Dateiformat wurde wahrscheinlich schon vor mehreren Jahren spezifiziert.

2.1 Das DAT-Format

Die vorliegenden LEGO-Modelle sind im DAT-Format[1] spezifiziert. Ein LEGO-Modell besteht typischerweise aus mehreren Bausteinen, die wiederum aus Teilstücken aufgebaut sind. Ein LEGO-Baustein ist hierarchisch aufgebaut. Am wichtigsten sind wohl die Referenzen auf Teilstücke, die erst auf tieferen Ebenen aus Zeilen bestehen, welche Geometrie definieren. Die Angaben einer DAT-Datei werden zeilenweise aufgeführt. Pro Zeile können Kommentare, grafische Primitive wie Linien, Dreiecke und Vierecke sowie Referenzen auf Modelle, Bausteine oder deren Teilstücke angegeben werden. Die Zeilen müssen in keiner bestimmten Reihenfolge angegeben werden, außer den nahenliegenden Konventionen, den Kommentar, oder wenn ein Teilstück innerhalb einer Datei definiert wird, den zugehörigen Befehl `File` voranzustellen.

2.1.1 Zeilen-Typen

Es gibt insgesamt sechs verschiedene Zeilen-Typen. Sie unterscheiden sich durch eine Zahl (0-5) am Anfang der Zeile. Abhängig von dem Zeilen-Typ folgen weitere Angaben.

2.1.1.1 0-Zeile : Kommentar oder Meta-Befehl

Die Null definiert einen Kommentar oder einen Meta-Befehl, wie z.B. `Model Title`, `Step`, `Write`, oder `Clear` . Durch den Meta-Command `File` können auch Teile innerhalb einer Datei definiert werden. `Step` definiert einen Zwischenschritt, um den Aufbau und Nachbau des Modells zu verdeutlichen. Bis auf `File` können alle 0-Zeilen übergangen werden. Geometrie oder Refernzen, die einem `File` folgen gehören nicht zu der DAT-Datei in der sie auftauchen, sondern zu der Datei, die angegeben wird.

Syntax:
```
0 string
0 meta-command
```
Beispiel:
```
0 Dies ist ein Kommentar
0 Step
0 File SOMETHING.DAT
```

2.1.1.2 1-Zeile : Referenz auf Baustein / Teilstück

Die Referenz auf andere Bausteine oder Teilstücke ist der Kern des Aufbaus eines LEGO-Modells im DAT-Format. Die Zeile einer Referenz besteht aus der „1“, einem Farbwert, einer Transformationsmatrix und dem Dateinamen des Teils.

`colour` bezeichnet den Farbwert (siehe auch 2.1.2. Farben im DAT-Format)
`x, y, z` definieren die Translation des Teils
`a – i` definieren Rotation und Skalierung des Teils. Die Matrix hat folgende Form:

```
| a d g 0 |
| b e h 0 |
| c f i 0 |
| x y z 1 |
```

`PART.DAT` ist der Dateinamen der refernzierten Datei

Syntax:
```
1 colour x y z a b c d e f g h i PART.DAT
```
Beispiel:
```
1 0 5 5 5 1 0 0 0 1 0 0 0 1 ComGuard.dat
```

2.1.1.3 2-Zeile : Linie

Linien dienen zur Kantenvisualisierung und werden meistens im Farbwert mit entgegengesetzter Helligkeit der zu umschließenden Fläche gezeichnet. Die Zeile besteht aus der „2“, dem vorangestellten Farbwert und zwei Punkten im Raum.

Syntax: `2 colour x1 y1 z1 x2 y2 z2`
Beispiel: `2 24 0 0 0 1 1 1`

2.1.1.4 3-Zeile : Triangle

Ein Triangle beschreibt ein Dreieck. Die Zeile besteht aus der „3“, dem vorangestellten Farbwert und drei Punkten im Raum. Die Reihenfolge der Punkte hat keine Auswirkungen auf die Füllung oder die Ausrichtung der Fläche. Es gibt keine Vorder- oder Rückseite.

Syntax: `3 colour x1 y1 z1 x2 y2 z2 x3 y3 z3`
Beispiel: `3 16 0 0 0 1 0 0 0 1 0`

2.1.1.5 4-Zeile : Quadrilateral

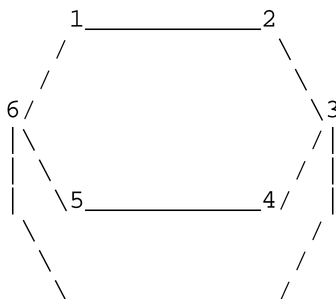
Ein Quadrilateral beschreibt ein Viereck. Die Zeile besteht aus der „4“, dem vorangestellten Farbwert und vier Punkten im Raum. Die Reihenfolge der Punkte hat keine Auswirkungen auf die Füllung oder die Ausrichtung der Fläche. Es gibt keine Vorder- oder Rückseite.

Syntax: `4 colour x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4`
Beispiel: `4 16 0 0 0 0 1 0 1 1 0 1 0 0`

2.1.1.6 5-Zeile : optionale Linie

Eine optionale Linie wird von LDraw abhängig von der Position einer zweiten Linie dargestellt. Die Linie zwischen Punkt 1 und Punkt 2 wird gezeichnet, wenn die Projektion der Punke 3 und 4 auf der selben imaginären Seite liegen wie die Projektion der ersten Linie auf den Bildschirm. Diese optionale Linie wird von LDraw benutzt um eine Tiefeninformation vorzutäuschen und entscheiden zu können, ob eine Linie verdeckt ist oder gezeichnet werden soll. Für diese Arbeit ist die optionale Linie nicht wichtig, da Linien in der Darstellung herausgefiltert werden, um den Realismus zu erhöhen.

Syntax: `5 colour x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4`
Beispiel: `5 24 1 0 0 1 1 0 0.9239 0 0.3827 0.9239 0 -0.3827`



Die Linie unter 1 soll nicht dargestellt werden, da 6 und 2 auf verschiedenen Seiten liegen.
Die Linie unter 2 soll nicht dargestellt werden, da 1 und 3 auf verschiedenen Seiten liegen.
Die Linie unter 3 soll dargestellt werden, da 2 und 4 auf der selben Seiten liegen.
Die Linie unter 4 soll nicht dargestellt werden, da 3 und 5 auf verschiedenen Seiten liegen.
Die Linie unter 5 soll nicht dargestellt werden, da 4 und 6 auf verschiedenen Seiten liegen.
Die Linie unter 6 soll dargestellt werden, da 1 und 5 auf der selben Seiten liegen.

2.2 Farben im DAT-Format

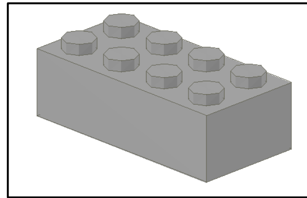
Um Linien, Dreiecke, Vierecke oder Referenzteile einzufärben wird eine Index-Farbpalette benutzt. (0=Schwarz, 1=Blau, 2=Grün, etc..) Addiert man 32 zum Farbwert hinzu, erhält man dieselbe Farbe mit Transparenz. Es gibt jedoch zwei entscheidene Ausnahmen im Farbindex. Die Zahl 16 definiert den Farbwert, der in der Referenz angeführt ist. Die Zahl 24 definiert die entsprechende Linienfarbe. Die Linienfarbe wird nicht durch die Komplementärfarbe bestimmt, vielmehr ist es dieselbe Farbe mit entgegengesetzter Helligkeit. Aus Hellblau wird somit Dunkelblau. Wird ein Teil mit einem Farbwert referenziert, so haben innerhalb dieses Teiles definierte Farben Vorrang, ausgenommen den Farben 16 und 24.

LDraw Farbindex:

0 black	**16 main color**
1 blue	17 Pastel-Green
2 green	18 Light-Yellow
3 dark cyan	19 Tan
4 red	20 Light-Purple
5 magenta	21 Glow-In-The-Dark
6 brown	22 Purple
7 grey	23 Purple-Blue
8 dark grey	**24 edge color**
9 light blue	25 Orange-Solid
10 light green	26 Dark-Pink
11 cyan	27 Lime-Green
12 light red	28 Tan-Solid
13 pink	29 unused
14 yellow	30 unused
15 white	31 unused

Ab 32 folgen die entsprechenden transparenten Farben.

2.3 Beispiel DAT-Datei



Beispielcode für ein 2x4 Baustein

```
0 Brick  2 x  4
0 Studs ontop
1 16  30 0 10     1 0 0 0 1 0 0 0 1 stud.dat
1 16  10 0 10     1 0 0 0 1 0 0 0 1 stud.dat
1 16  -10 0 10    1 0 0 0 1 0 0 0 1 stud.dat
1 16  -30 0 10    1 0 0 0 1 0 0 0 1 stud.dat
1 16  30 0 -10    1 0 0 0 1 0 0 0 1 stud.dat
1 16  10 0 -10    1 0 0 0 1 0 0 0 1 stud.dat
1 16  -10 0 -10   1 0 0 0 1 0 0 0 1 stud.dat
1 16  -30 0 -10   1 0 0 0 1 0 0 0 1 stud.dat

0 Studs below
1 16  20 4 0      1 0 0 0 -5 0 0 0 1 stud4.dat
1 16  0 4 0       1 0 0 0 -5 0 0 0 1 stud4.dat
1 16  -20 4       0 1 0 0 0 -5 0 0 0 1 stud4.dat

0 inner and outer box
1 16  0 24 0      36 0 0 0 -20 0 0 0 16 box5.dat
1 16  0 24 0      40 0 0 0 -24 0 0 0 20 box5.dat

0 Quads to close boxes from below
4 16   40 24  20   36 24  16  -36 24  16  -40 24  20
4 16  -40 24  20  -36 24  16  -36 24 -16  -40 24 -20
4 16  -40 24 -20  -36 24 -16   36 24 -16   40 24 -20
4 16   40 24 -20   36 24 -16   36 24  16   40 24  20
```

3 Konzept

Der GL*ego*-Viewer sollte unter VisualC++ entwickelt werden und als Frontend und *GUI*, der grafischen Oberfläche, fiel die Wahl auf *FLTK* (Fast Light Tool Kit). Die Entscheidung, die grafische Ausgabe über OpenGL zu realisieren, wurde schon in der Planungsphase gefällt, da OpenGL plattformübergreifend und weit verbreitet ist. Dennoch kollidieren manche Spezifikationen des DAT-Formats mit den Anforderungen von OpenGL. Als Beispiel ist im DAT-Format die Reihenfolge der Punkte eines Dreiecks oder Vierecks beliebig, dagegen ist sie in OpenGL entscheidend. Der GL*ego*-Viewer soll duch Datenstrukturen realisiert werden, die leicht in ein Hauptprogramm einzubinden sind und nur wenige Aufrufe für den Ablauf benötigen.

3.1 Geplante Features

- Jede beliebige DAT-Datei soll einzulesen und grafisch auszugeben sein
- Grafisches User Interface (GUI), zur angenehmen Bedienung

3.1.1 Darstellung:

- Möglichst realistische Beleuchtung und Materialeigenschaften (Plastik-Look)
- Um einen gewissen „Zeichentrick“-Effekt wie bei LDraw zu vermeiden, soll die Liniendarstellung herausgefiltert werden
- Um den Realismus zu erhöhen, soll eine optionale Grundfläche, die dynamisch über eine Bounding-Box generiert wird, eingeblendet werden können
- Texturierung oben genannter Grundfläche (Holz, Stein, Stoff, etc..)
- Automatisches Anpassen der Kameraposition, sodaß das Modell in optimaler Größe dargestellt wird

3.1.2 Optimierung:

- Einlesen des Modells durch *Caching*, also durch einmaliges Speichern, der oft benutzen Teilstücke
- Um die OpenGL-Ausgabe zu beschleunigen sollen *Display-Listen* verwendet werden. Diese werden vor der eigentlichen Darstellung generiert und können damit beschleunigt ausgegeben werden. Vergleichend mit der Struktur des DAT-Formats sollen diese Display-Listen hierarchisch aufgebaut werden.
- Durch notwenige Verdopplung der Grafikflächen (siehe auch 3.2.1) kann durch *Backface Culling*, das Unterdrücken der Anzeige der Flächen, deren Vorderseite nicht zum Betrachter zeigt, die Ausgabe beschleunigt werden

3.1.3 Schatten:

- Dynamischer Schatten, bei beweglicher Lichtquelle
- Schattenberechnung durch Projektion des Modells auf den Untergrund aus Sicht der Lichtquelle mit Hilfe der Projektionsmatrix
- Schattendarstellung über Volumenberechnung ist sehr aufwenig, da keinerlei Nachbarschaftsinformationen der Polygone vorliegen und diese erst generiert werden müßten
- Wenn Transparente Flächen, wie Fenstern, einen helleren Schatten werfen würden, wäre dies ein angenehmes Bonus-Feature

3.1.4 Handhabung:

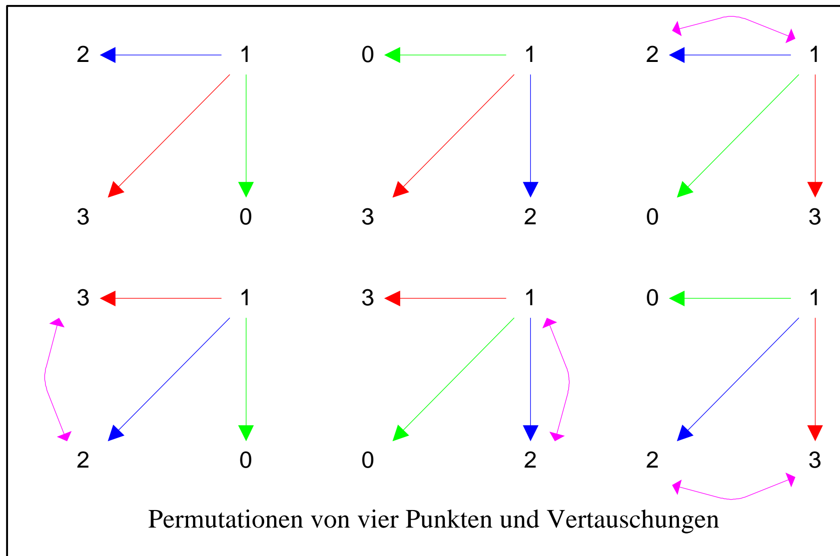
- Grafische Benutzeroberfläche, die dem Benutzer eine angenehme Bedienung ermöglicht
- Damit der Benutzer den frei wählbaren Blickwinkel in Echtzeit verändern kann, braucht er nur diesen intuitiv per Maus zu beeinflußen
- Um dem Benutzer eine möglichst große Freiheit zu geben, selbstständig die Darstellung zu verändern, sind die Schattenberechnung, die Darstellung der Untergrundfläche (mit Textur belegbar) und die Anzeige der Normalen ein- und ausschaltbar
- Um auch weitläufige Modelle anzeigen zu können, oder um gewollte Linseneffekte wie ein „Fischauge“ zu erzielen, ist der Kamerawinkel (Field of View) veränderbar
- Für Ausdrucke ist ein weißer Hintergrund sinnvoll, eine andere Farbe kann einen positiven Eindruck auf den Betrachter haben, darum ist die Hintergrundsfarbe direkt veränderbar
- Um die Installation zu vereinfachen, sind veränderbare Optionen wie Pfade oder Farbwerte über eine *INI-Datei*, eine Text-Datei mit Namen GLEGO.INI, einstellbar

3.2 Auftretende Schwierigkeiten

Das DAT-Format spezifiziert in recht eigentümliche Weise die Modelle. Dadurch ist es verständlich, daß es zu gewissen Komplikationen mit der Darstellung der Modelle durch OpenGL kommt.

3.2.1 Orientierung von Flächen

Egal in welcher Reihenfolge die drei oder vier Punkte, die eine Fläche definieren, im DAT-Format angegeben werden, zeichnet LDraw ein Fläche, die keine Vorder- oder Rückseite hat. In OpenGL verändert die Reihenfolge der Punkte der Fläche die *Orientierung*. Werden die Punkte eines Vierecks nicht kreisförmiglaufend angegeben, so werden zwei verdrehte Dreiecke mit unterschiedlicher Beleuchtung dargestellt. Der GL*ego*-Viewer überprüft, ob die Punkte kreisförmig angegeben wurden und muß diese gegebenenfalls vertauschen.



Permutationen von vier Punkten und Vertauschungen

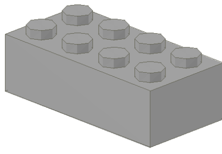
Durch folgendes Verfahren kann man vier Punkte in kreisförmige Reihenfolge bringen:

```
// getting vectors from points
a10 = p0 - p1;
a12 = p2 - p1;
a13 = p3 - p1;
// building normals (with cross)
n0 = a12 * a10;
n1 = a10 * n0;
n2 = n0  * a12;
// checking skalar
s1=a13.dot(n1);
s2=a13.dot(n2);
if (s1 < 0) switch(p1,p2);          // switch 1-2
if (s2 < 0) switch(p2,p3);          // switch 2-3
```

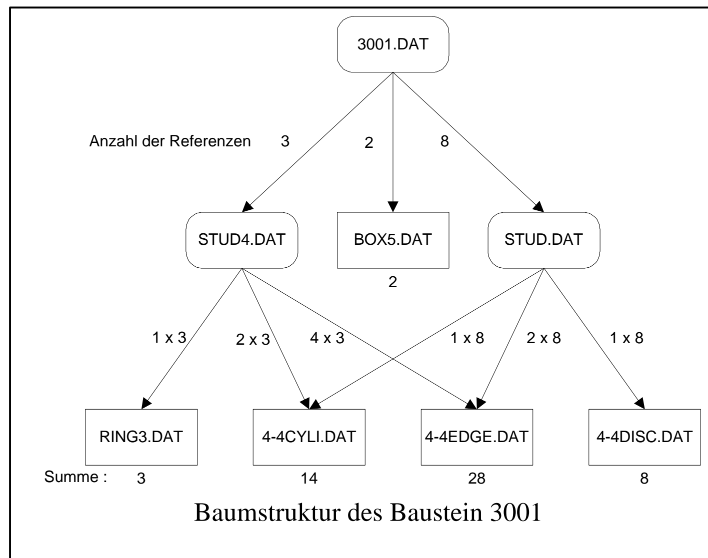
In LDraw besitzt eine Fläche keine Vorder- oder Rückseite, beide Seiten werden angezeigt. In OpenGL entscheidet aber der Umlauf der Punkte, mit bzw. gegen den Uhrzeigersinn, über die Orientierung der Fläche. Somit käme es zu einer falschen Darstellung, da manche Flächen nicht gezeichnet würden, wenn sie als Rückseite betrachtet werden. Daher ist es notwendig, jede Fläche zweimal in die Szene einzufügen, einmal als Vorder- einmal als Rückseite. Durch dieses Verfahren wird die Anzahl der Geometrie verdoppelt, was sich negativ auf die Performance für komplexe Szenen auswirkt. Man kann aber wiederum die Anzeige beschleunigen, indem man durch OpenGL durch Backface-Culling, die Rückseiten der Flächen nicht darstellt. Leider kommt es immer noch in sehr vereinzelten Fällen dazu, daß die Normalen bei manchen Flächen umklappen und die Fläche nicht die gewünschte Beleuchtung zeigt.

3.3 Interne Techniken

3.3.1 Partmap – Der Cache von Teilstücken



Wenn man an ein LEGO-Baustein denkt, hat man typischerweise das Bild von einem 2x4 Baustein im Sinn. Wie im Beispielcode (siehe 2.1.3) zu erkennen, werden auf der ersten Ebene nur drei verschiene Teilstücke 13mal referenziert. Steigt man in die zweite Ebene ab, so wird z.B. die Datei `4-4edge.DAT` 28mal und `4-4cyli.DAT` 14mal referenziert. Der Baustein `3001.DAT` ist also aus nur fünf verschiedenen Teilstücken aufgebaut. Selbst wenn man nun ein Modell aus mehreren dieser Bausteine aufbaut, bleibt die Anzahl der Teilstücke gleich, nur die Anzahl der Referenzen steigt. Dies wird auch als Vorteil in Kapitel 3.5. ausgenutzt durch die Verwendung von Display-Listen. Um nicht mehrmals dasselbe Teilstück einlesen und seine Geometrie und Referenzen speichern zu müssen, wird ein eingelesenes Teilstück in den Cache, der Partmap, eingefügt. Bei der nächsten Referenz wird überprüft, ob es schon eingelesen wurde oder dies noch nötig ist. So kann der Zugriff auf die Festplatte und auch der Aufwand zur Aufbereitung des Modells deutlich reduziert werden.



Baumstruktur des Baustein 3001

3.3.2 Die Struktur der gespeicherten Daten

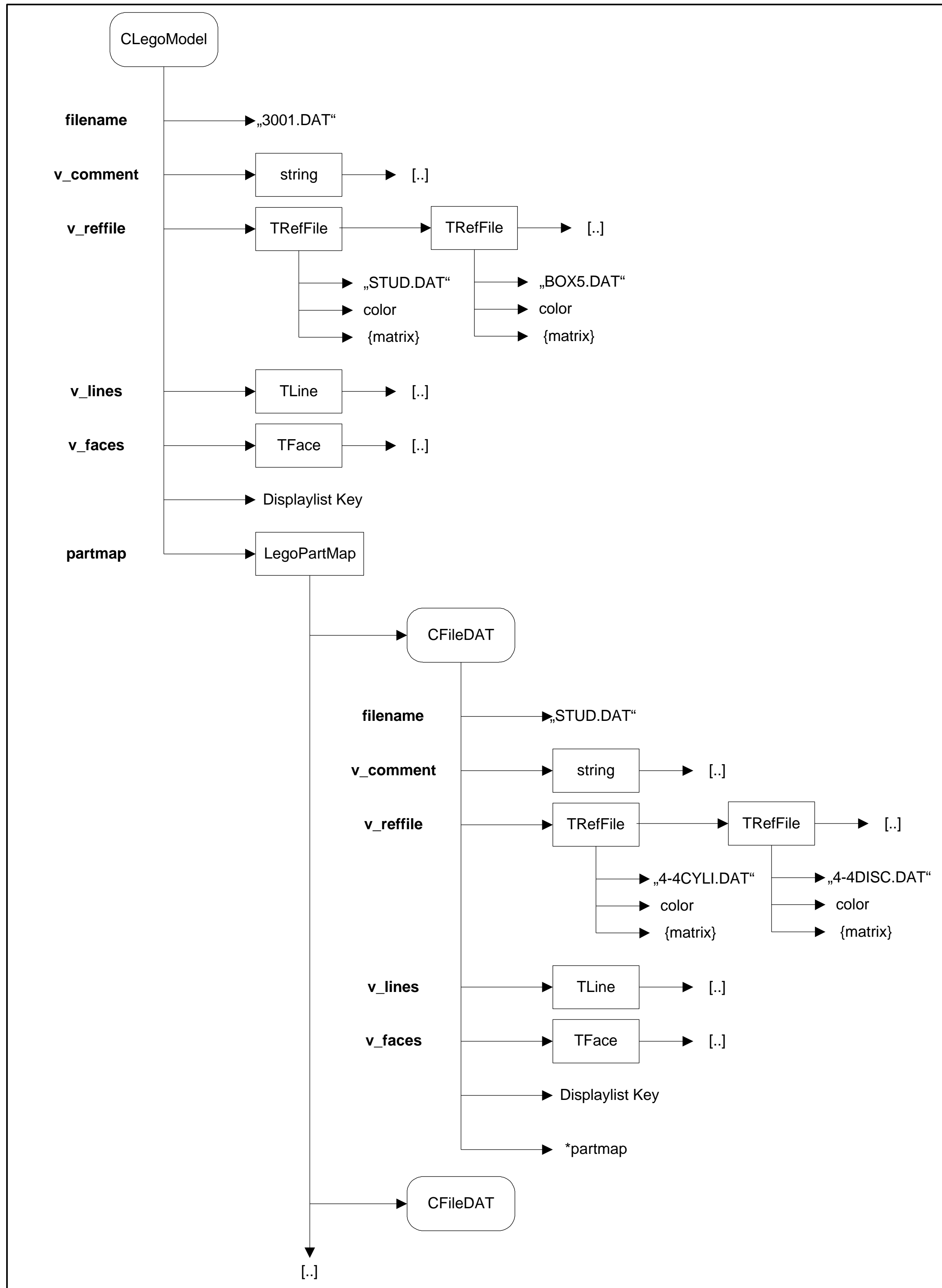
Beim Aufbau einer passenden Struktur zur Aufbereitung der Daten sind besonders die Angaben einer Referenz wichtig. Da ein Baustein an einer Stelle grün und an einer anderen Stelle vielleicht transparent-blau referenziert werden kann, scheint es sinnvoll, eine Baumstruktur aufzubauen. Analog zur DAT-Datei werden in den Knoten die Referenzen auf andere Bausteine / Teilstücke hierarichisch gespeichert. Darin wird die Transformationsmatrix, der Farbwert und der Dateiname gespeichert. Zugriff auf die eigentlichen Daten (Geometrie) eines Knoten, also eines Bausteins / Teilstücks, geschieht dann über die Partmap. So wird jedes Teilstück nur einmal bearbeitet und abgespeichert, durch die Baumstruktur der Referenzen ist aber das komplette Modell nachvollziehbar. Der genaue Aufbau dieser Datenstrukturen ist in Kapitel 4 Implementierung im Abschnitt 4.1. Schematischer Aufbau nachzuvollziehen.

3.3.3 Verschachtelte Display-Listen

Das DAT-Format ist hierarchisch aufgebaut und referenziert oft (hundert- oder sogar tausendfach) auf dieselben Teilstücke. Dies kann man als Vorteil mit Hilfe der Display-Listen in OpenGL umsetzen. Display-Listen werden vor der eigentlichen Darstellung aufgebaut und können durch OpenGL optimiert werden, da sie statische Befehle enthalten. Ein weiterer Vorteil ist, daß diese Display-Listen auch verschachtelt aufgebaut werden können. Beispielsweise kann Liste2 also innerhalb des Aufbaus der Liste1 generiert und aufgerufen werden. Die Transformationsmatrix, die bei einer Referenz angegeben ist, wird also mit der aktuellen OpenGL Matrix multipliziert und dann braucht einfach nur die dem referenzierten Teilstück dazugehörige Display-Liste aufgerufen werden.

4 Implementierung

4.1 Schematischer Aufbau



Schematischer Aufbau der Speicherung der Daten

4.2 Die Klasse `CLegoModel`

`Die Klasse CLegoModel` umfasst die Eigenschaften eines kompletten LEGO-Modells im DAT-Format und ist abgeleitet von der Klasse `CFileDAT`, die folgend genauer beschrieben wird. `CLegoModel` wurde um einen Cache der Teilstücke erweitert, welche in `partmap` gespeichert werden. Diese Map besteht aus einem String, der den Dateinamen speichert, und einem LEGO-Modell der Klasse `CFileDAT`.

```
typedef map<string, CFileDAT>      LegoPartMap; //defining Cache

class CLegoModel : public FileDAT{
      public:
            LegoPartMap  partmap;              // Cache
            CLegoModel();
            ~CLegoModel();
      private:
};
```

4.3 Die Klasse `CFileDAT`

`CFileDAT` ist die eigentlich Klasse, in der eine DAT-Datei gespeichert ist. Sie speichert den Dateinamen, den zugehörigen Pfad, die referenzierte Farbe, verschiedene Listen der Zeilentypen und weitere verschiedene interne Datentypen.

4.3.1 Speichern der Informationen der DAT-Datei

Die Daten der DAT-Datei werden komplett gelesen und in verschiedenen Datenstrukturen gespeichert. Sie werden in verschiedenen Listen, die als Vektoren realisiert wurden, der Klasse CFileDAT gespeichert, die wiederum in der Partmap gesammelt werden. Obwohl der GL*ego*-Viewer den Kommentar oder die Linien nicht anzeigt, werden sie denoch für zukünftige Zwecke gelesen und gespeichert. Den recht vollständigen Aufbau dieser Datentypen ist in 4.1. nachzuvollziehen.

4.3.1.1 Kommentar

Die Kommentare werden als String gespeichert. Bis auf den Meta-Command `File` werden andere Zeilen einfach ignoriert und im Objekt gespeichert.

```
typedef vector<string>        ListComment;
ListComment                   v_comment;
```

Ein `File-`Befehl startet einen Prozess, der die Datei durchliest und innerhalb dieser definierte DAT-Objekte erkennt und als Datei speichert. Ein erneutes Auswählen dieser Datei ist dann nötig. Es kommt vereinzelt zu Abbrüchen, wenn eine leere Kommentarzeile (nur eine 0) einer Leerzeile (kein Zeichen) folgt.

4.3.1.2 Referenz auf Teilstück

In dieser Liste werden alle aufgeführten Referenzen gespeichert. Über den Dateinamen als Schlüssel kann auf das eigentliche Objekt in der Partmap zugegriffen werden. Die Transformationsmatrix wird direkt als Matrix in der in 2.1.1.2. beschriebenen Form gesichert. „Matrix“ ist eine Klasse, die Matrixmanipulationen vereinfacht und die für die Transformationsmatrix benutzt wurde.

```
struct TRefFile {
      string                  filename;
      matrix                  transmatrix;
      int                     color;
};
typedef vector<TRefFile>      ListTReffile;
ListTReffile                  v_reffile;
```

4.3.1.3 Linie

Beide Linien-Typen (2 und 5, siehe 2.1.) werden als zwei Punkte im Raum und dem Farbwert gespeichert.

```
struct TLine {
      t_3DVector                  vektors[2]; // 0-1 vectors
      int                         color;
};
typedef vector<TLine>         ListTLine;
ListTLine                     v_lines;
```

4.3.1.4 Transparente und solide Dreiecke und Vierecke

Die eigentliche Geometrie wird in `ListTFace` gespeichert. In einem Array von sechs `t_3Dvector` werden drei (bzw. vier) Punkte der Fläche, die dazugehörige Normale und der Mittelpunkt der Fläche zusammengefaßt. `Facetype` spezifiziert ob die Fläche ein Dreieck (3) oder ein Viereck (4) ist.

```
struct TFace {
      int                         facetype;   // 3=Tri or 4=Quad
      t_3DVector                  vektors[6]; // 0-3 vectors,
                                              // 4 normal, 5 middle
      int                         color;
};
ListTFace                     v_faces;
typedef vector<TFace>         ListTFace;
```

4.4 Partmap

In der Partmap werden die Teilstücke gesammelt. Um in der DAT-Struktur abzusteigen, läuft man `CFileDAT v_reffile` ab und greift über den Filename auf das Teilstück im Partmap zu.

```
typedef map<string, FileDAT>  LegoPartMap;
```

`Beispieldurchlauf um die Displaylisten Schlüssel auszugeben:`

```
for (int i = 0; i < v_reffile.size(); i++)
      cout << (*p_partmap)[v_reffile[i].filename)].listkey;
```

Weitere Variablen, Strukturen und benutzte Klassen werden im Quellcode aufgeführt und hier nicht näher erklärt.

4.5 Ablauf zur Verwendung der Klassen

Soll ein LEGO-Modell angezeigt werden, so geschieht das in drei unterscheidbaren Schritten. Das Modell wird *geparset*, also eingelesen, dann aufbereitet und schließlich angezeigt.

4.5.1 Schritt 1 – Einlesen

Nachdem ein Lego-Objekt angelegt wurde, wird der gewünschte Dateiname eingetragen und die Partmap definiert. Dann startet man das Einlesen des Objekts an (beispielhafter Code):

```
CLegoModel Lego;
Lego.filename = filename;
Lego.partmap[Lego.filename] = Lego;
Lego.parse();
```

4.5.1.1 Parsen

Bei Parsen wird überprüft, ob das Objekt mit dem Dateinamen schon vorher referenziert und in die Partmap eingetragen wurde. Sonst wird in den verschiedenen Pfaden, die in der „INI-Datei" angegeben wurden, die entsprechende Datei gesucht und geöffnet. Entsprechend der aufgeführten Zeilen in der Datei werden die Listen mit Referenzen, Linien oder Flächen gefüllt. Eine Baumstruktur wird innerhalb der Referenzen aufgebaut.

4.5.2 Schritt 2 – Aufbereitung

Während der Aufbereitung der Daten werden verschiedene Displaylisten erstellt und eine Bounding-Box, eine imaginäre Hülle um das Modell, aufgebaut. Zuerst werden in den Objekten der Partmap verschiedene Displaylisten-Schlüssel generiert und dann gefüllt, dabei gibt es eine Schatten-Displayliste, die keine Farbwerte enthält.

```
Lego.generateLists();
Lego.fillingLists(1);
Lego.fillingShadow();
Lego.setBBox();
```

4.5.2.1 Das Füllen der Displaylisten

Das Füllen der Displaylisten muß etwas genauer angeführt werden. Dabei wird während der Definition einer Displayliste zuerst die im Objekt liegenden Flächen angegeben. Dann werden die Referenzen auf andere Teilstücke durchlaufen, die aktuelle Transformationsmatrix mit der referenzierten multipliziert, gegebenenfalls der Farbwert gesetzt und deren Displayliste aufgerufen. Man kann Displaylisten aufrufen, die erst durch den rekursiven Ablauf später gefüllt werden. Es muß beachtet werden, daß die speziellen Farbwerte 24 und 16 ihre aktuelle Farbe an Kindteile weitergeben, aber trotzdem transparente Flächen separat behandelt werden (siehe auch Schritt 3 – Darstellung). Der Farbwert einer Displayliste wird somit nicht „darin" sondern „vor" ihrem Aufruf definiert, damit das selbe Teil durch eine andere Referenzierung auch einen anderen Farbwert annehmen kann.

Pseudocode:

```
glNewList (listkey, GL_COMPILE_AND_EXECUTE);
for (i = 0; i < v_faces.size(); i++)
        if (aktuelleFarbe == solide) {
                richtige Farbe setzen;
                Face aufrufen;
        }
for (i = 0; i < v_reffile.size(); i++) {
        glPushMatrix();
        glPushAttrib(GL_CURRENT_BIT);
        richtige Farbe setzen;
        glMultMatrixd(v_reffile[i].transmatrix);
        glCallList ((*p_partmap)[(v_reffile[i].filename)].listkey);
        glPopAttrib();
        glPopMatrix();
}
glEndList();
```

4.5.3 Schritt 3 – Darstellung

Zu Beginn der Darstellung wird über ein entwickeltes Verfahren, dem *Fit-View* (siehe 4.5.1), die Kameraposition so gewählt, daß das Modell möglichst optimal den Bildschirm ausfüllt. Dann wird eine Schleife abgearbeitet, die in der entsprechenden Reihenfolge die Szene darstellt. So wird erst die Hintergrundsfarbe gesetzt, dann optional der Untergrund gezeichnet. Diese wird anhand der Unterseite der aufgebauten Bounding-Box ausgerichtet. Der Schatten wird abhängig zu der Lichtposition durch ein „Flachdrücken" und dem Aufruf der farblosen Displayliste des Modells realisiert. Dazu wird aus Sicht der Lichtquelle das Modell auf die Ebene des Untergrunds gezeichnet. Auch die Normalen der Flächen können wahlweise dargestellt werden. Nun wird der „solide" Teil, dann der transparente Teil des Modells gezeichnet. Diese Reihenfolge ist wichtig, damit man durch die transparenten Teile die soliden sieht, bei einer anderen Reihenfolge käme es zu optischen Fehlern.

```
glClearColor (back_r, back_g, back_b, 1.0);
if (underground)  Lego.drawUnderground();
if (drawshadow)   Lego.drawShadows(lightpos);
if (drawnormals)  Lego.drawNormals();
glCallList(Lego.listkey);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glCallList(Lego.translistkey);
glDisable(GL_BLEND);
```

4.6 *Weitere Details*

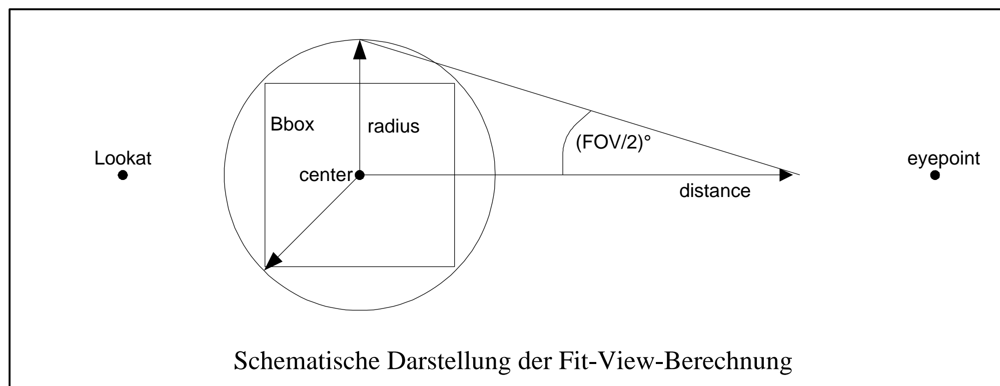
Im Folgenden werden einzelne Details der Implementierung beschrieben, die wichtig zur vollständigen Beschreibung dieser Arbeit sind.

4.6.1 Texturen des Untergrundes

Die Texturen liegen im TGA-Bildformat vor. Die Höhe und Breite des Bildes müßen dieselbe Anzahl von Bildpunkten haben. Weitere Texturen können hinzugefügt werden, indem sie einfach in des entsprechende Textures-Verzeichnis kopiert werden. Dann können sie über das Menü der grafischen Oberfläche dem Untergrund zugewiesen werden.

4.6.2 Fit-View

Fit-View beschreibt ein Verfahren, welches zur optimierten Darstellung des Modells entwickelt wurde. Nach dem Laden einer DAT-Datei oder nach dem Anklicken des Fit-View-Knopfes durch den Benutzer wird dieses Verfahren aktiviert. Es berechnet eine Kameraposition, aus der das Modell möglichst bildschirmfüllend dargestellt wird und setzt die Kamera an diesen Punkt. Dabei wird aus dem Öffnungswinkel der Kamera (*Field of View*), dem Mittelpunkt des Modells und dem umschließenden Radius eine Kameraposition berechnet, sodaß das Modell eine optimale Größe annimmt. Der bestehende Öffnungswinkel der Kamera wird halbiert und dann kann über den Radius der Kugel um das Modell mit Hilfe der Tangensfunktion die Entfernung zum Mittelpunkt des Modells berechnen. Dann wird über Vektorberechnung der aktuelle Blickvektor normalisiert, mit der berechneten Distanz gestreckt und durch Translationen so verändert, daß man die Kameraposition direkt verändern kann. Zuhilfe kam dabei eine Hilfsfunktion `unProjectGL` von Sven Havemann, die die aktuelle Kameraposition und den Blickpunkt berechnet.



Schematische Darstellung der Fit-View-Berechnung

Entscheidener Teil der Fit-View-Berechnung:

```
unProjectGL(eyepoint,lookat,v_width>>1,v_height>>1);
bspherecenter=Lego.boundbox.center();
bsphereradius =   (float) sqrt(
(Lego.boundbox.v_max.x - bspherecenter.x)
*(Lego.boundbox.v_max.x - bspherecenter.x)
+(Lego.boundbox.v_max.y - bspherecenter.y)
*(Lego.boundbox.v_max.y - bspherecenter.y)
+(Lego.boundbox.v_max.z - bspherecenter.z)
*(Lego.boundbox.v_max.z - bspherecenter.z));
distance = bsphereradius / tanf(fovy * 0.5 * M_PI / 180.0);
t_3DVector le = lookat-eyepoint;
le.normalize();
le *= distance;
le += eyepoint;
le = bspherecenter - le;
glTranslatef(-le.x,-le.y,-le.z);
```

4.6.3 Konfiguration der INI-Datei

Um den GL*ego*-Viewer flexibel an verschiedene Systembedingungen anzupassen, sind einige Parameter über eine Text-Datei zeilenweise einstellbar. Jede Zeile kann auch über das #-Zeichen am Ende mit einem Kommentar ergänzt werden.
Damit optimal nach den DAT-Dateien, in denen Modelle, Bausteine und Teilstücke definiert sind, während des Einlesens gesucht werden kann, sind die verschiedenen Pfade flexibel. Sie können über den Zeile-Befehl `Path` angegeben werden. Leider werden Leerzeichen in dem Pfadnamen nicht unterstützt.

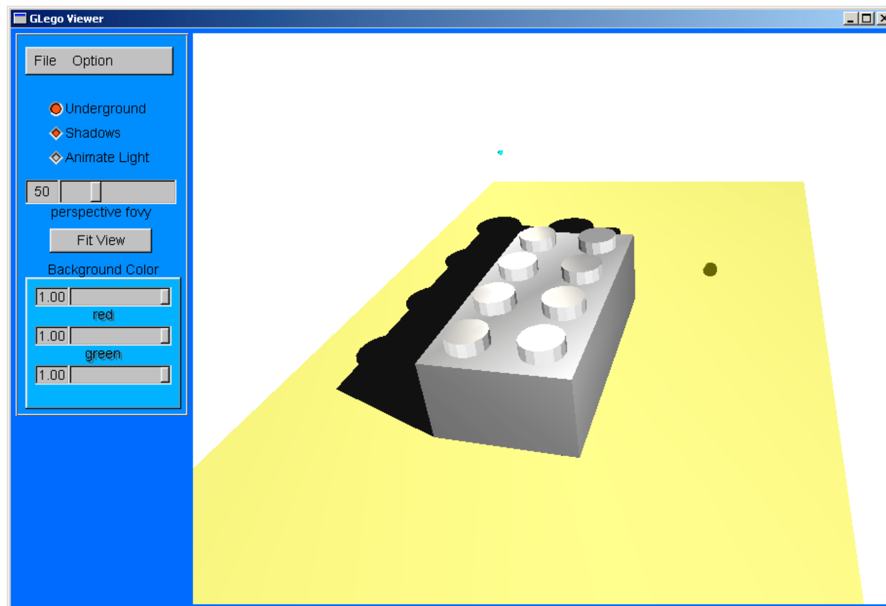
Syntax: `PATH string`
Beispiel: `PATH ../Models/PARTS/   # Find parts here`

Außerdem sind alle Farbwerte des Farbindexes einzeln einstellbar. So können eigene Farbwert erstellt oder bestehende den eigenen Bedürfnissen angepaßt werden. Farbwert werden über den Zeilen-Befehl `Color` angegeben. Dabei wird die Indexnummer angegeben und dann der Rot/Grün/Blau-Anteil und die Deckungsintensität in Gleitkommazahlen.

Syntax: `COLOR int   float float float float`
Beispiel: `COLOR 34    0     0.5   0.2   0.6   #34 Trans-Green!`

4.7 Grafische Oberfläche

Die Oberfläche des GL*ego*-Viewers ist benutzerfreundlich ausgelegt und ermöglicht einen schnellen Zugang zu den dynamischen Einstellungen. So kann ein neues Modell leicht über die gängige Menü-Struktur geöffnet oder das Programm beendet werden. Über den Menü-Punkt „Option" kann die gewünschte Textur des Untergrunds gewählt oder die Darstellung der Normalen kontrolliert werden. Oft benutzte Einstellungen sind direkt einstellbar. So kann die Darstellung des Untergrundes und des Schattens oder die Animation der Lichtquelle direkt angewählt werden. Unter diesen Kontrollfeldern befindet sich ein Schieberegeler, mit dem man direkt den Öffnugswinkel der Kamera einstellen kann. Darunter liegt der Fit-View-Knopf, mit dem die Größe des Modells optimal gewählt wird. Schließlich kann man die Hintergrundfarbe direkt über drei Schieberegler durch ihre Rot/Grün/Blau-Anteile einstellen.



Grafische Oberfläche des GL*ego*-Viewers

5 Ergebnisse und Vergleich

Um den GL*ego*-Viewer hinreichend evaluieren zu können, muß die Ausgangsposition und das angestrebte Ziel erneut betrachtet werden. Da es vor allem um die grafische Umsetzung geht, sei gestattet, primär die jeweilige optische Qualität zu vergleichen. Der Vergleich von Interface, Frame-Raten, Speicherbedarf oder *Preprocessing*, der Zeit zum Anstarten eines Modells, ist einmal schwer durchzuführen, zum anderen nicht so entscheidend, wie die tatsächlich erreichte optische Qualität. Als Ausgangspunkt wurde hierzu LDraw mit dem Frontend MLCad benutzt. Als optimales Ziel war die optische Qualität des Ray-Tracers POV-Ray vorgegeben. Dabei stellte sich heraus, daß der Export zu POV-Ray recht umständlich war und auch einige Szenen nicht anzuzeigen war (siehe Modell Castle Front), fehlerhafte Farbwerte oder das Fehlen von Modellkomponenten aufwies. Die Auswahl der Modelle erfolgte primär aus ästethischen Gesichtspunkten. Es sollte hervorgehoben werden, daß bei komplexen Geometrien durch das notwendige doppelte Einfügen von Flächen, die Performance des GL*ego*-Viewer deutlich nachgiebt, welches aber durch eine geeignete Optimierung (siehe Kapitel 6) behoben werden kann.

5.1 Berechnungszeiten:

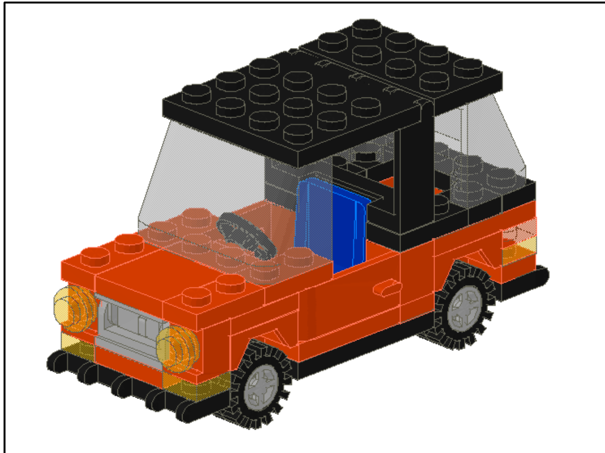
Verwendete Hardware: AMD 850MHz, 256MB RAM, GeForce 1 DDR RAM

Car:	43.892 Dreiecke		
Preprocessing.	GL:ca. 1 Sek.	LD: ca. 1 Sek.	POV-Ray: 10 Sek.
Render time:	GL:flüssig	LD: ca. 5 fps	POV-Ray: 17 Sek.
SnowSpeeder:	212.656 Dreiecke		
Preprocessing.	LD: ca. 2 Sek.	GL:ca. 5 Sek.	POV-Ray: 37 Sek.
Render time:	LD: ca. 5-10 fps	GL:ca. 2fps	POV-Ray: 11 Sek.
Warhammer:	783.200 Dreiecke		
Preprocessing.	LD: ca. 2 Sek.	GL:ca. 5 Sek.	POV-Ray: 50 Sek.
Render time:	LD: ca. 1-2fps	GL:ca. 2fps	POV:Ray: 22 Sek.
Castle Front:	1.224.618 Dreiecke		
Preprocessing.	GL:ca. 5 Sek.	LD: ca. 6 Sek.	Nicht lauffähig
Render time:	GL:ca. 1 fps	LD: ca. 1 fps	Nicht lauffähig

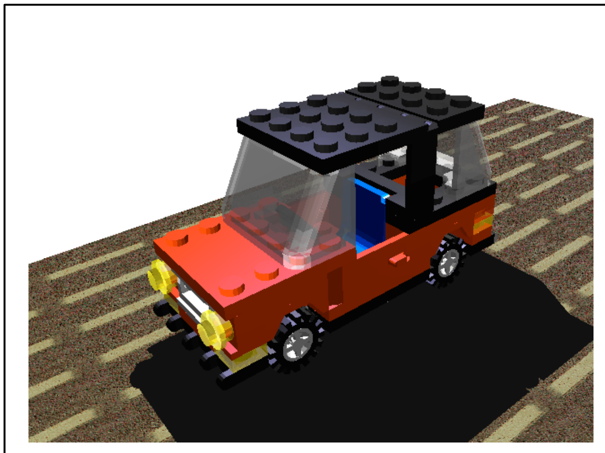
5.2 Grafischer Vergleich

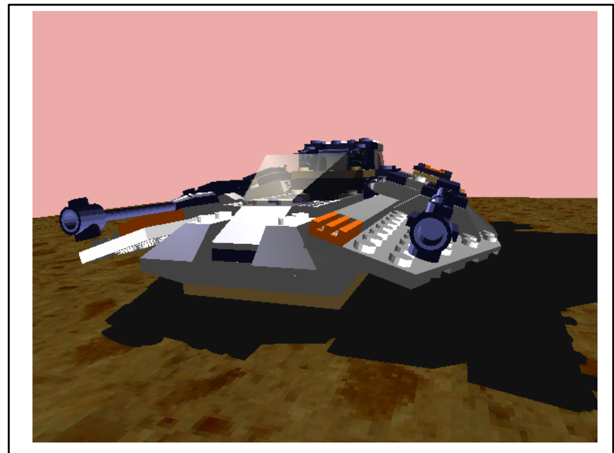
Car **Snowspeeder**

LDraw:

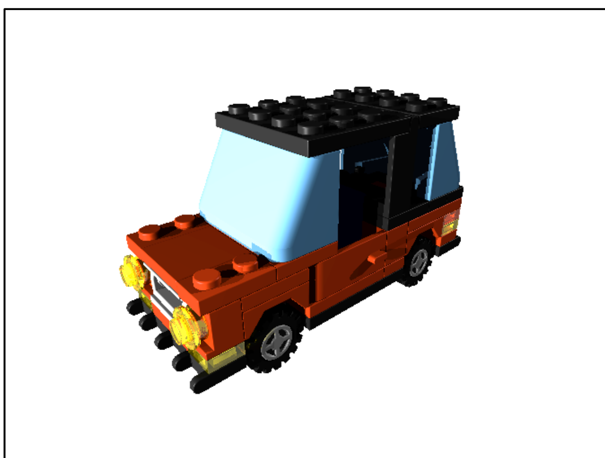


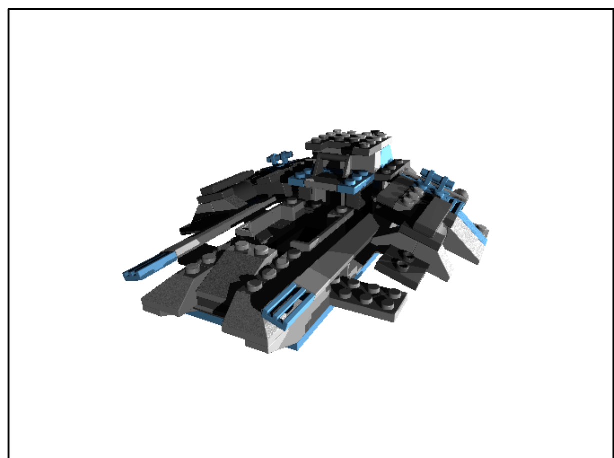
GL*ego*-Viewer:





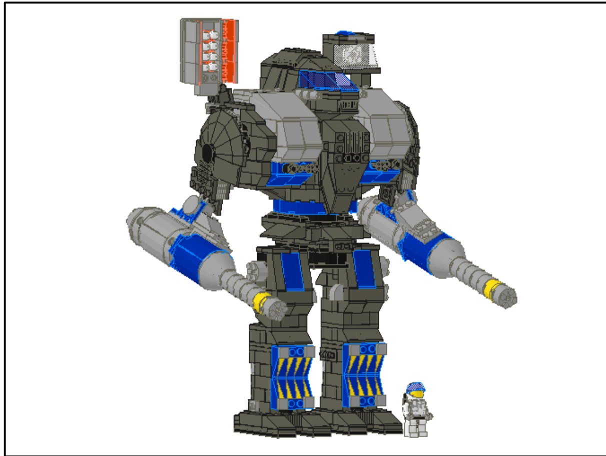
POV-Ray:



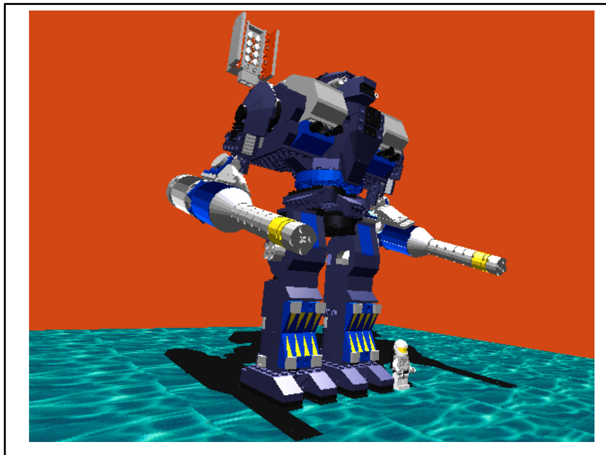


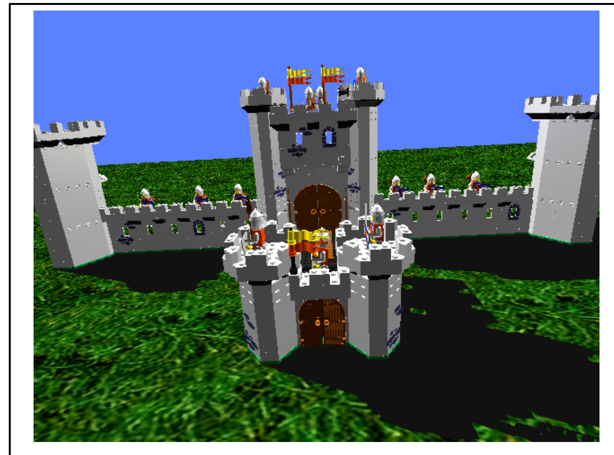
Warhammer | **Castle Front**

LDraw:

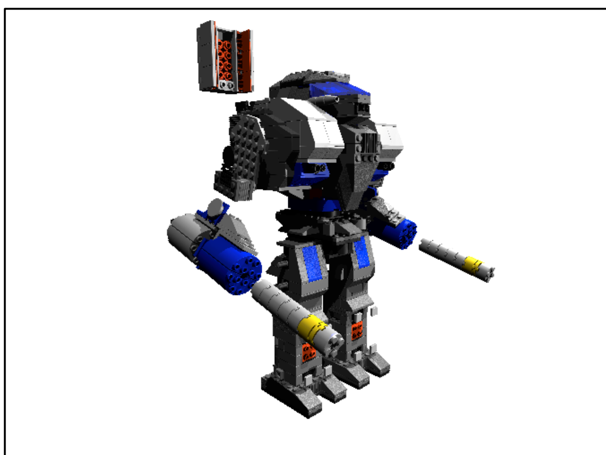


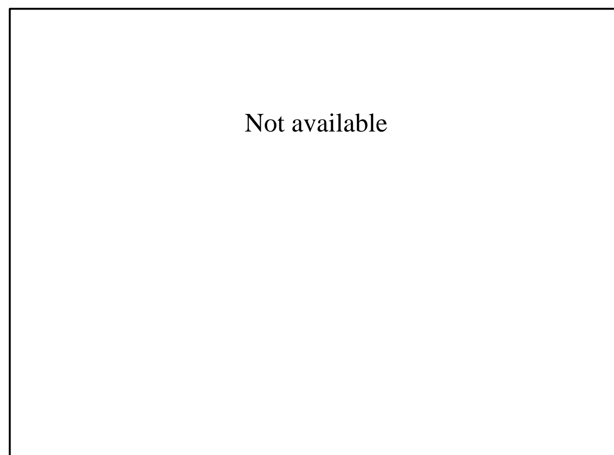
GL*ego*-Viewer:



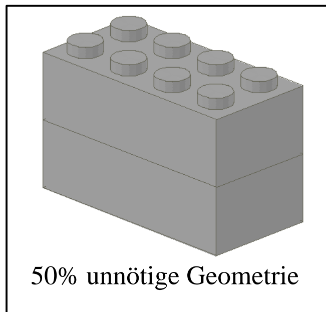


POV-Ray:



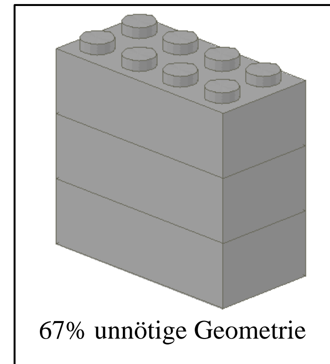


6 Ausblick



50% unnötige Geometrie

Im Rahmen dieser Studienarbeit konnten mehrere Ansatzpunkte und Optimierungsmöglichkeiten nicht bearbeitet werden. So hat sich herausgestellt, daß eine sehr große Menge an Geometrie in der Szene insofern unnötig ist, als sie innerhalb geschlossener Flächen liegt. LEGO-Steine sind gerade so konzipiert, daß Aus- und Einbuchtungen ineinander passen und bündig abschliessen. Schon bei zwei soliden und übereinander gesteckten 2x4-Bausteinen, die jeweils durch ca. 1400 Dreiecke dargestellt werden, wären ca. 50% der Geometrie überhaupt nicht sichtbar. Bei drei Steinen werden schon ca. 67% und bei vier ca. 76% Flächen unnötig dargestellt. Hier sollte dringend eine Optimierung greifen, indem überprüft wird, welche Flächen von anderen bündig eingeschlossen sind. Die Zusammenführung der Außenseiten führt wohl eher zu geringer Verbesserung.



67% unnötige Geometrie

Ein sehr großer Nachteil, der auch massiv diese Arbeit geprägt hat, ist die Tatsache, daß das DAT-Format keine einheitliche Orientierung der Flächen vorschreibt. Hier eine geeignete Lösung zu finden, damit nicht die doppelte Geometrie durch Vor- und Rückseite entsteht, macht Sinn.

Ein mögliche Lösung wäre der Aufbau eines geeignetes Netz der Flächen, mit dem man eingeschlossene Flächen erkennen und herausfiltern kann. Dies hätte noch weitere Vorteile. So könnte man je nach *dihedralem* Winkel, dem Neigungswinkel aneinander stoßender Flächen, diese Kante durch interpolierte Normalen die Beleuchtung glätten. Auch eine höherwertige Schattenberechnung durch Volumen ist dann möglich, weil man Außenkanten leicht finden kann. Bei dieser Schattenberechnung werden aus der Lichtquelle über die Kanten des Modells Strahlen geschossen und ein Volumen konstruiert, in dem die Lichtquelle verdeckt wird. Der Vorteil dieser Berechnung liegt darin, daß alle Objekte innerhalb dieses Volumens mit einem Schatten versehen werden; die in dieser Arbeit verwendete Berechnung wirft nur einen Schatten auf den Untergrund. Durch Volumenberechnung kann ein Objekt auf sich selbst einen Schatten werden. Um bei dem Baustein `3001` zu bleiben, würden dann die acht Ausbuchtungen einen Schatten auf die Oberfläche des Quaders werfen.

7 Zusammenfassung

Der GL*ego*-Viewer sollte die bestehenden Modelle im DAT-Format mit guter Qualität interaktiv darstellen. Trotz einiger grafischen Fehler durch umklappende Normalen aufgrund bisher ungeklärter Matrixmultiplikation in OpenGL, ist diese Aufgabenstellung voll erreicht worden. Durch die recht genaue Modellierung der Teilstücke wächst die Anzahl der Dreiecke in einer Szene bei größeren Modellen recht stark an. Dadurch ist eine „flüssige" Betrachtung ab ca. 300.000 Dreiecke je nach benutzter Hardware nur eingeschränkt möglich. Doch meistens entsteht diese hohe Zahl durch unnötige Geometrie, die sich in eingeschlossenen Bereichen befindet. Es wird erwartet, daß durch eine Optimierung in diesem Bereich die Geschwindigkeitsprobleme behoben werden können. Die grafische Qualität der Ausgabe ist gegeben. Die Lichtverhältnisse sind während der Laufzeit veränderbar und sogar die berechneten Schatten werden der Lichtposition nach angepasst. Das Einblenden des Bodens erhöht meistens drastisch die Orientierung und den Realismus, besonders wenn eine passende Textur geladen wird. Vor allem das interaktive Betrachten des Modells und die dynamische Auswahl der Perspektive sind große Vorteile gegenüber dem Raytracing, bei dem man vor der Berechnung erst eine Perspektive angeben muß. Dadurch kann man Standbilder der gewünschten Perspektive durch virtuelles Abfotografieren des Bildschirms angenehm und schneller realisieren als durch einen Raytracer, während die grafische Qualität kaum unterscheidbar und weniger fehleranfällig ist.

Ein persönlicher Dank geht an Sven Havemann, der diese Arbeit qualifiziert und (fast schon zu) motivierend begleitet hat. Es freut mich auch, daß Felix Funke die Herausforderung antritt und basierend auf meiner Arbeit weitere Energie und Forschung in dieses „Spielzeug" investieren wird, welches allzu schnell mit einem Augenzwinkern beurteilt wird.

8 Quellenverzeichnis

[1] http://www.ldraw.org/faq/
[2] http://www.opengl.org
[3] http://www.ldraw.org
[4] http://www.povray.org