Fachhochschule Köln
Campus Gummersbach
Fakultät für Informatik und
Ingenieurwissenschaften

Entwicklung eines JOGL-basierten Frameworks zur Darstellung von Game Physics

Diplomarbeit

zur Erlangung des Diplomgrades
Diplom - Informatiker (FH)
in der Fachrichtung Allgemeine Informatik

ausgearbeitet von:

Adrian Dietzel
Hofwiese 3 (Benroth)
51588 Nümbrecht

Matrikelnummer: 11031862 1 5

im Studiengang Allgemeine Informatik

Vorgelegt am 27. Februar 2007

Erster Prüfer:	Prof. Dr. Wolfgang Konen
Zweiter Prüfer:	Prof. Dr. Erich Ehses

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

2D	2-Dimensional
3D	3-Dimensional
API	application programming interface
ARB	Architecture Review Board
AWT	Abstract Window Toolkit
CPU	Central Processing Unit
FPS	frames per second
GHz	Gigahertz
GL4Java	OpenGL for Java
GLU	OpenGL Utitlity Library
GLUT	OpenGL Utility Toolkit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
JAR	Java Archive
JDK	Java Development Kit
JFC	Java Foundation Classes
JNI	Java Native Interface
JOGL	Java OpenGL
JSR	Java Specification Request
JSR-231	Java Binding for OpenGL
JVM	Java Virtual Machine
LWJGL	Lightweight Java Game Library
MHz	Megahertz
OpenGL	Open Graphics Library
RAM	Random Access Memory
RGB	Red, Green Blue (Additives Farbmodell)
SGI	Silicon Graphics Incorporated
UML	Unified Modeling Language

1 Einleitung

Der rasante Fortschritt in der Entwicklung von Computersoftware und Computerhardware hat in den letzen Jahren seine Spuren hinterlassen. Neben der fast unüberschaubaren Menge an verschiedener, angebotener Software hat sich auch die Hardware überproportional schnell entwickelt. Somit sind komplexe Multimedia-Anwendungen, welche vorher nur Großrechnern vorbehalten waren, ohne Probleme auf einem aktuellen Heimcomputer einsetzbar. Gerade im Bereich der Darstellung von komplexer Computergrafik ist der technische Fortschritt auf Hochtouren. Es ist mittlerweile sogar möglich, sehr realistische 3D-Welten mit fast allen erdenklichen Effekten, wie zum Beispiel Licht- und Schatteneffekte, auf dem Personalcomputer zu simulieren. Das Spektrum der Anwendungsmöglichkeiten in diesem Gebiet reicht von einfachen Bildbearbeitungsprogrammen bis hin zu hochkomplexen 3D-Modellierungsprogrammen[1] wie beispielsweise Autodesk Maya[2], mit dem bereits bekannte Filme wie zum Beispiel Shrek[3] erstellt wurden. Vorallem aber die Spieleindustrie erzielt dank der fortgeschritten Technik in der Grafikdarstellung wirtschaftliche Erfolge wie nie zuvor. Fast jedes aktuelle Computerspiel, welches sich auf dem Markt etabliert, ist mit neuester 3D-Computergrafik ausgestattet. Doch neben der Grafikdarstellung spielt bei einem Computerspiel auch die Physik eine erhebliche Rolle. Erst in Verbindung mit korrekten physikalischen Berechnungen wirken Spielszenarien oder Animationen lebendig und täuschen eine wirkliche Welt vor. Dynamische Bewegungen oder realistische Kollisionen mit unterschiedlichen Kräfteverhältnissen sind ohne dementsprechende Physik undenkbar. Genau dieses Thema wird in dieser Diplomarbeit behandelt. Es soll ein einfaches Rahmenwerk (Framework) erstellt werden, mit dem es möglich ist, in der Programmiersprache Java physikalische Vorgänge, welche meist in modernen Computerspielen eingesetzt werden (Game Physics), in einer 3D-Welt darzustellen. Dieses Framework soll dann später in dem dazugehörigen Wahlpflichtfach namens „Spiele, Simulation, dynamische Systeme“ zum Einsatz kommen, um so die Studenten bei ihrer Arbeit mit Game Physics zu unterstützen. Bisher existiert zwar ein unterstützendes Framework, welches aber in C/C++ realisiert worden ist. Da aber an der Fachhochschule Köln mittlerweile der Fokus auf die Programmiersprache Java gelegt worden ist, ist die Einarbeitung in C/C++ oft zu mühsam für die Studenten, um sich auf die Erforschung der Gamephysik konzentrieren zu

[1] Programm zum erstellen von 3D-Figuren/Modellen und/oder 3D-Animationen

[2] Berühmtes 3D-Modellierungsprogramm mit Fokus auf 3D-Animationen, welches bereits zahlreiche Awards gewonnen hat.(<http://usa.autodesk.com>)

[3] Aufwendiger, computeranimierter Kinderfilm

können. Das neue Framework soll wie das bisherige Framework die Grafikbibliothek Open Graphics Library (OpenGL) benutzen, welche heutzutage neben der Grafikbibliothek DirectX den Standard für die Darstellung von 3D-Grafik darstellt. OpenGL stellt hier deshalb eine gute Wahl dar, da es im Gegensatz zu DirectX plattformunabhängig ist und somit auf fast allen Plattformen zu Verfügung steht. Detailierte Informationen hierzu und die Gründe für die Wahl von OpenGL sind in Kapitel 2.2 ausführlich beschrieben. Weil aber nun OpenGL vollständig in C implementiert wird, kann dieses nicht ohne weiteres von Java verwendet werden. Hierfür wird noch ein sogenanntes Java Binding[4] benötigt, welches Java mit OpenGL verknüpft. Mithilfe eines solchen Bindings können OpenGL-Funktionen von Java angesprochen werden. Ein sehr aktuelles und einfach zu bedienendes Binding nennt sich JOGL, welches für „Java OpenGL“ steht. Dieses soll zusammen mit OpenGL die Basis des neuen Frameworks bilden. Zwar gibt es noch einige andere mögliche Alternativen zu JOGL wie beispielsweise Java3D, aber in Kapitel 3.1 werden die Gründe genannt, warum sich gerade für JOGL entschieden wurde.

1.1 Ziele der Diplomarbeit

Es ist nicht das Ziel dieser Diplomarbeit, eine komplette „3D-Engine“ zu schreiben, welche meist das Herzstück in kommerziellen Computerspielen darstellt. Eine solche 3D-Engine ist meist ein *eigenständiger* Teil eines Computerprogramms und dient ausschließlich der schnellen und optimierten Darstellung von Szenarien und wiederverwendbaren Objekten. Vielmehr soll im Umfang dieser Diplomarbeit ein Framework entwickelt werden, mit dem es möglich ist, zum Beispiel einfache 3D-Objekte darzustellen und diese mit physikalischen Berechnungen zu verknüpfen. In erster Linie soll so dem Benutzer (Studenten der FH Köln) eine *Arbeitserleichterung* verschafft werden, in dem das Framework eine für die Physik-Simulationen notwendige Umgebung darstellt. Da das Framework zudem komplett in Java programmiert wird, sollte es aus bereits genannten Gründen weniger Probleme für die Studenten mit der Einarbeitung geben. Auch soll das Framework in seinem Aufbau und seinem Programmcode relativ einfach gehalten werden, damit es für interessierte Benutzer nachvollziehbar bleiben kann. Auch eventuelle Erweiterungen für die Zukunft sind so leichter handzuhaben. Somit muss ein Kompromiss zwischen dem Umfang/Komplexität des Frameworks und dessen Funktion der Arbeitserleichterung gefunden werden. Wichtig ist aber auch, dass existierende C/C++ Projekte des bisherigen Frameworks leicht in das neue Framwork zu portieren sind. Diese Möglichkeit soll später durch Portierung eines größeren Projektes getestet werden. Da Java einen langsamen, unabhängigen Bytecode verwendet, und keinen nativen[5] Maschinencode, stellt sich hier zudem zwingend die Frage nach der neuen Performance des Frameworks, welche

[4] Softwareschnittstelle zwischen Java und OpenGL
[5] hier: native = Prozessor-spezifisch

ebenfalls unter verschiedenen Gesichtspunkten ausführlich untersucht und getestet wird. Das letze Ziel dieser Diplomarbeit ist es dann, eine neue Game Physics Simulation auf dem neuen Framework durchzuführen. Sie behandelt das Thema „Real-Time Fluid Dynamics“, mit denen es möglich ist, dynamischen Rauch zu erzeugen. Da dieser Bereich sehr komplex ist, wird dieses Thema nur angerissen und auch nur die wichtigsten Grundlagen erläutert/implementiert.

1.2 Aufbau der Diplomarbeit

Der Aufbau der Diplomarbeit beginnt damit, dass erst wichtige Grundlagen erläutert werden, bevor genauer auf das Framework eingegangen wird. Zuerst wird eine Einführung in OpenGL (Kapitel 2) und JOGL (Kapitel 3) vorangestellt. Dies ist wichtig, damit später nicht zuviele Begriffserklärungen notwendig sind, welche den Lesefluß unnötig beeinträchtigen würden. Dann wird auf Basis dieser Grundlagen die Planung (Kapitel 4) und Entwicklung (Kapitel 5) des Frameworks beschrieben. Es wird hier vorrausgesetzt, dass gute Java-Kenntnisse und einfache Grundlagen der Softwaretechnik vorhanden sind. Danach folgt in Kapitel 6 eine exemplarische Portierung eines C/C++ Projektes des bisherigen Frameworks in das neue Framework. Unterschiede und eventuelle Probleme hierbei werden genauer untersucht. Das Kapitel 7 behandelt darauf die Performance-Untersuchungen bezüglich des Frameworks. Zuletzt wird in das Thema „Real-Time Fluid Dynamics“ (Kapitel 8) eingeführt und eine 2D-Rauch-Simulation entwickelt. Abbildung 1.1 zeigt noch einmal grafisch den Aufbau und die Struktur dieser Diplomarbeit.

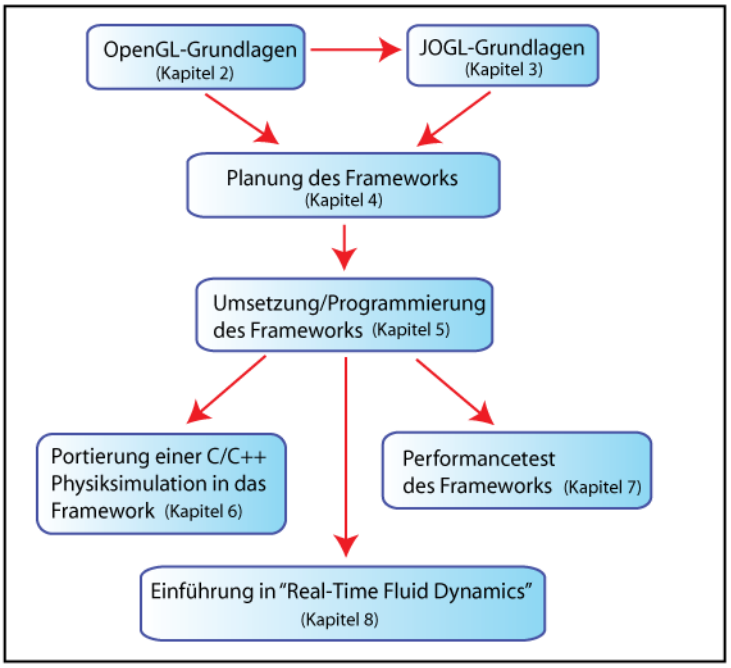


Abbildung 1.1: Aufbau und Struktur der Diplomarbeit

2 Grundlagen von OpenGL

2.1 Was ist OpenGL ?

OpenGL ist die Abkürzung für Open Graphics Library und ist eine Spezifikation für ein plattform- und programmiersprachenunabhängiges API[1] zur Entwicklung von 3D Computergrafik. Diese wird seit 1992 von dem OpenGL ARB (Architecture Review Board) festgelegt und hat heute unter anderem wichtige Mitglieder wie Sun, HP, Intel, ATI, NVIDIA und 3DLabs. Ursprünglich wurde OpenGL von Silicon Graphics Incorporated (SGI) entwickelt und stammt von IrisGL ab, einer Grafik-API, die damals nur mit dem Unix Derivat Irix kompatibel war. Ähnlich wie OpenGL entwickelte auch Microsoft eine Schnittstelle zur 3D-Hardware (Grafikkarte), welche sich Direct3D nannte. Im Jahr 1997 schlossen sich darauf SGI und Microsoft in dem Projekt „Fahrenheit“ zusammen, um dort ihre 3D-Standards zu vereinheitlichen. Aus finanziellen Gründen aber brach SGI das Projekt seinerseits ab. Microsoft, welches ebenfalls ein Mitglied des OpenGL ARB's war, verließ nun im Jahr 2003 überraschenderweise das Komitee, um seinen eigenen API-Standard DirectX durchzusetzen, welcher Direct3D beinhaltet.[2]

2.2 Alternativen zu OpenGL und warum OpenGL ?

Die Frage, warum für das zu entwickelnde Framework die 3D-Grafik-API OpenGL benutzt werden soll und keine anderen möglichen Alternativen, ist leicht beantwortet. Es gibt außer der bereits erwähnten API namens DirectX von Microsoft keine wirkliche Alternative. Aktuelle Grafikkarten, welche heute hauptsächlich von ATI und NVIDIA produziert werden, können nur mit diesen beiden Schnittstellen zusammenarbeiten. Welche Alternative nun besser von beiden ist, darüber gibt es bereits zahlreiche Diskussion und Gerüchte von Anhängern beider Parteien im Internet. Eine seriöse, unabhängige Informations-Quelle konnte aber nicht bestimmt werden. Die Entscheidung zur Wahl einer der beiden Grafik API's sollte von der Art der Anwendung abhängen. Da DirectX nicht nur eine Grafik-API darstellt, sondern noch zusätzlich optimale Unterstützung für Sound, Netzwerk, Maus und Keyboard besitzt, ist dieses für Spiele optimal geeignet und wird auch dementsprechend oft eingesetzt. Ein weitaus wichtigerer Fakt aber ist, dass DirectX nur auf dem Betriebssystem Windows

[1] API = Application Programming Interface = Schnittstelle zum Programmieren von Anwendungen
[2] Vgl. [Com07d]

funktioniert. OpenGL dagegen ist plattformunabhängig und kann auf jedem beliebigen Betriebssystem eingesetzt werden. Da das Framework in dieser Diplomarbeit mit Java erstellt werden soll, wäre es unvorteilhaft, eine plattformunabhängige Sprache mit einer betriebssystemabhängigen Grafikschnittstelle wie DirectX zu verwenden. Hierdurch würde der Sinn von Java zum größten Teil verloren gehen.

2.3 Grundlagen der Programmierung von OpenGL

2.3.1 Implementierung von OpenGL

Da OpenGL nur eine Spezifikation ist, stellt sie also nur einen Standard dar, wie bestimmte Befehle und Funktionen auszusehen haben, um auf die 3D-Hardware eines Computers zugreifen zu können. Die konkrete Implementierung der ca. 150 definierten Befehle und Funktionen ist dagegen in der Treibersoftware[3] der jeweiligen Grafikkarte realisiert. Dieses abstraktere Modell mit dem Kommunikationsumweg über OpenGL-kompatible Treiber hat den Vorteil, dass nicht spezifisch für bestimmte Grafikkarten programmiert werden muss (siehe Abbildung 2.1). Jedes OpenGL Programm läuft so theoretisch auf jedem Grafikkartentyp. Da einige rechenintensive Grafikoperationen direkt in Hardware umgesetzt werden müssen, wie zum Beispiel das Berechnen von Schatten, stellt sich das Problem, dass nicht jede Grafikkarte auch wirklich alle Operationen unterstützt. OpenGL löst dieses Problem, indem es solche Hardwareoperationen per Softwareoperationen emuliert[4], welches aber meist im Bezug auf die Geschwindigkeit große Einbußen mit sich zieht.

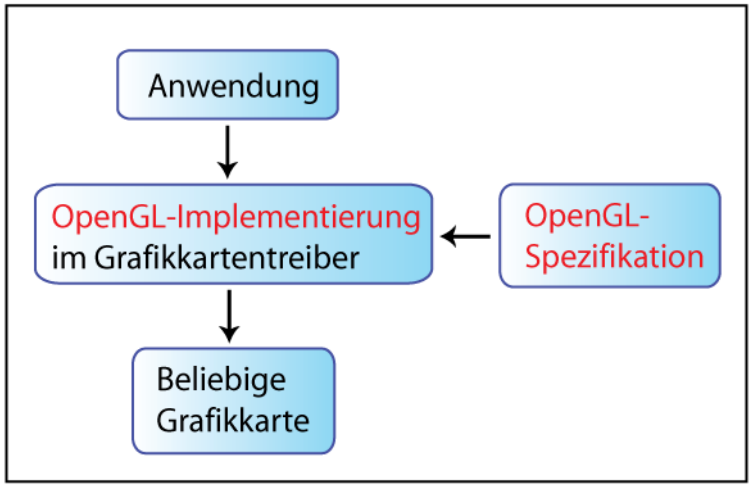


Abbildung 2.1: OpenGL-Implementierung in den Treibern [Ron98], Seite 42

[3] Über eine Treibersoftware kann direkt mit der dazugehörigen Hardware kommuniziert werden
[4] emulieren = nachahmen

2.3.2 Programmiermodell von OpenGL

Das Programmiermodell von OpenGL ist so gestaltet, dass einfache Objekte mit Hilfe von 3D-Koordinaten erzeugt werden können und anschließend gerendert werden. Unter „Rendern“ ist hierbei das Darstellen der primitiven Objekte auf dem Bildschirm gemeint. Die Objekte werden von der Grafikkarte ausgewertet, umgerechnet und an den Bildschirm weitergeleitet. Dabei können die Objekte mit Farben versehen und auch deren Lage im Raum definiert werden. Besonders hierbei ist, dass OpenGL wie ein Zustandsautomat arbeitet. Das bedeutet, dass solange bestimmte Werte und Einstellungen verwendet werden, bis diese einen anderen Zustand bekommen. Wird zum Beispiel eine bestimmte Farbe für ein Objekt definiert, dann gilt diese nicht nur ausschließlich für dieses Objekt, sondern auch für alle weiteren Objekte.[5] Diese Vorgehensweise hat den Vorteil, dass so meist wenig Einstellungen und Übergaben von Parametern nötig sind, um eine Vielzahl von Objekten zu erstellen. Folgend werden die Grundlagen für die Programmierung mit OpenGL erklärt. Zum Vertiefen und Nachschhlagen der nächsten Unterkapitel sei hier vorallem auf [Ron98] und [ND97] verwiesen.

2.3.3 Primitive Objekte

OpenGL arbeitet mit sogenannten Vertices, welche auch Eckpunkte genannt werden. Ein Vertex hat die Raumkoordinaten X, Y und Z und stellt das kleinstmögliche zu zeichnende Objekt dar, nämlich einen Punkt im 3D-Raum. [6] Verbindet man nun einige unterschiedliche Vertices mit jeweils einer Linie, so lässt sich fast jedes beliebige Objekt erzeugen. Diese Objekte werden in OpenGL mit der Bezeichnung „Primitive“ klassifiziert. Zehn wichtige Primitive können bereits über vordefinierte Funktionen gezeichnet werden, von denen Abbildung 2.2 einige exemplarisch zeigt. Bei Primitiven, welche dort mit einer ausgefüllten Fläche dargestellt sind, gehört die Fläche selbst zum Primitive dazu, welches zum Beispiel bei `GL_LINE_STRIP` und `LINE_LOOP` nicht der Fall ist, da diese Primitive nur aus reinen Linien bestehen. Bei `GL_POLYGON` ist zu beachten, dass sich hier die Linien nicht kreuzen dürfen und die Hülle konvex[7] sein muss.[8]

[5] Vgl. [ND97], Seite 18
[6] Ein Punkt im 3D Raum wird auch Voxel genannt (Volumetric Pixel)
[7] konvex = nach außen gewölbt, keine Kante zeigt nach innen
[8] Vgl. [ND97], Seite 38

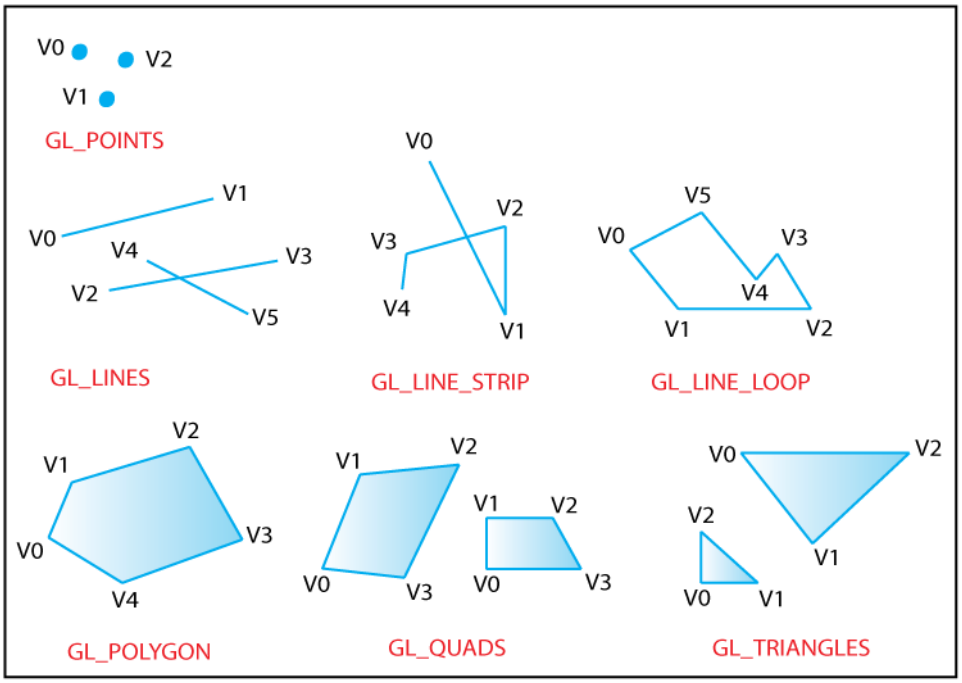


Abbildung 2.2: Einige vordefinierte Primitive [ND97], Seite 37

2.3.4 Koordinatensystem

Um Primitive in einem virtuellen Raum darstellen zu können, wird ein Koordinatensystem benötigt, indem die Primitive erscheinen können. Man nennt dieses Koordinatensystem auch Weltkoordinatensystem und es besteht aus den 3 Achsen X, Y und Z. In OpenGL wird ein rechtshändiges, kartesisches Koordinatensystem verwendet. Hierbei sind die X- und Y-Achsen wie gewohnt mit aufsteigenden Werten von links nach rechts und von unten nach oben belegt (siehe Abbildung 2.3). Die Z-Achse dagegen hat abnehmende Werte in die Tiefe. Je weiter ein Objekt entfernt ist, desto kleiner ist sein Z-Wert. Zum Erstellen und Darstellen von Primitiven stellt OpenGL den Befehl `glVertex(x, y, z)` bereit, der es erlaubt einzelne Eckpunkte an den im Parameter angegeben Koordinaten zu definieren. Diese werden dann darauf miteinander verbunden. Wichtig ist, dass die Definition in einem dafür vorgesehen Block geschieht, welcher im Kopf die Primitive-Typ Bezeichnung stehen haben muss. Der Block fängt mit dem Befehl `glBegin(primitiveTyp)` an und hört mit `glEnd()` auf.[9] Ohne solch einen Block inklusive Typ-Bezeichnung könnte nicht ermittelt werden, wieviele Vertices miteinander verbunden werden sollen. Innerhalb des Blocks dürfen mehrere Primitive gleichen Typs definiert werden. Abbildung 2.4 zeigt das Erstellen eines einfachen Quadrates.

[9] Vgl. [Ron98], Seite 56-60

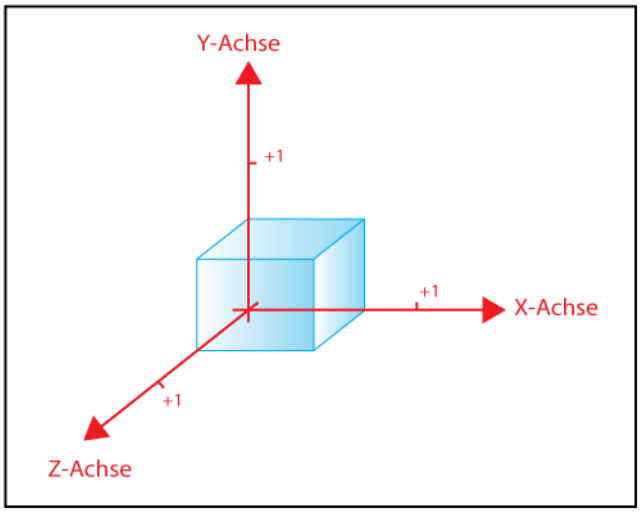


Abbildung 2.3: Rechtshänd., kartesisches Koordinatensystem [Ron98], Seite 72

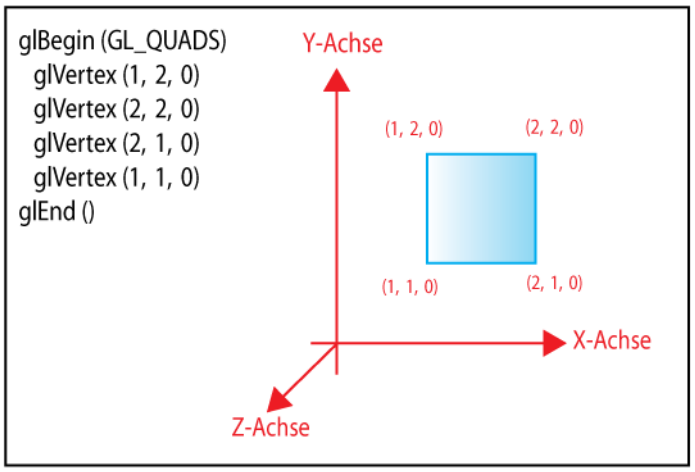


Abbildung 2.4: Zeichnen eines einfachen Quadrates

2.3.5 Matrizen

Zum weiteren Verständis für die Programmierung von OpenGL ist es wichtig zu wissen, dass hier mit sogenannten Matrizen gearbeitet wird. Da der Computerbildschirm nur eine 2D-Darstellung erlaubt, werden mit Hilfe solcher Matrizen die virtuellen 3D-Primitive auf 2 Dimensionen transformiert. Die genaue Mathematik, welche hinter den Transformationen steckt, wird aber nicht benötigt. Es reicht aus zu wissen, welche Transformationen es gibt und wie die dazugehörige Matrix anzuwenden und einzustellen ist. Die Transformationen dienen unter anderem auch der Verschiebung, Bewegung und Rotation der Primitive und wie sie darauf im Weltkoordinatensystem dargestellt werden. Es gibt folgende Transformationen und dazugehörige Matrizen in OpenGL:[10]

[10] Vgl. [ND97], Seite 73-95

Transformation	**Matrix**
Modeling-Transformation	Modelview-Matrix
Viewing-Transformation	Modelview-Matrix
Projection-Transformation	Projection-Matrix
Vieport-Transformation	nicht vorhanden

Tabelle 2.1: Transformationen und Matritzen in OpenGL

1. **Modeling-Transformation**
   In OpenGL sind verschiedene Sichtweisen möglich, wie die Transformationen zu verstehen sind. Aus der Sichtweise, welche hier benutzt wird, besitzt jedes Primitive ein eigenes lokales Koordinatensystem, in dem es definiert wird. Es ist unabhängig von den anderen lokalen Koordinatensystemen anderer Primitive.[11] Die Modeling-Transformation ist nun dafür zuständig, solch ein lokales Koordinatensystem in das Weltkoordinatensystem zu transformieren. Durch die Modeling-Transformation können Primitive verschoben, rotiert und skaliert werden, welche dann dementsprechend im Weltkoordinatensystem auftauchen. Für die Modeling-Transformation ist die Modelview-Matrix notwendig, welche ihren Namen daher hat, dass sie ebenfalls zuständig für die Viewing-Transformation ist.

2. **Viewing-Transformation**
   Die Weltkoordinaten der Primitive werden mit Hilfe dieser Transformation in Eye-Koordinaten umgewandelt. Hier wird der Standort des Betrachters mit in die Transformationsberechnung einbezogen, wodurch die Eyekoordinaten sozusagen die Sicht des Betrachters auf die Primitive darstellen. Wie bereits erwähnt, wird hierfür ebenfalls die Modelview-Matrix verwendet, welches daran liegt, dass beide Transformationen schwer voneinander gedanklich zu trennen sind. Geht zum Beispiel der Betrachter näher an die Primitive heran, hat dies den gleichen Effekt, als würden die Primitive selber skaliert werden.[12]

3. **Projection-Transformation**
   Mit Hilfe dieser Transformation, wird die Sichtweite des Betrachters mit einkalkuliert. Ein Primitive, das aufgrund zu weiter Entfernung nicht mehr im sichtbaren Bereich des Betrachters ist, wird geclipped (ausgeschnitten). Die Koordinaten in diesem Stadium werden somit als Clip-Koordinaten bezeichnet, für deren Berechnung hier nun die Projection-Matrix verwendet werden muss.

4. **Vieport-Transformation**
   Letztendlich werden die Clip-Koordinaten in 2D-Viewport-Koordinaten umgewandelt und auf dem Bildschirm dargestellt. Man nennt den sichtbaren Bild-

[11] Vgl. [ND97], Seite 75
[12] Vgl. [ND97], Seite 80

schirmbereich auch Viewport, welcher gewöhnlich genauso groß wie die Auflösung des Bildschirms ist.

Abbildung 2.5 zeigt noch einmal grafisch die Reihenfolge der einzelnen Transformationsschritte, die benötigt werden, um ein Primitive auf den 2-dimensionalen Bildschirm umzurechnen. Es sei hier noch erwähnt, dass dieses Modell der Übersicht halber nur die einzelnen Transformation aufzeigt. Weitere interne Zwischenschritte, wie Perspektiven-Korrekturen, sind bewusst ausgelassen worden und nachzuschlagen unter [ND97], Seite 67.

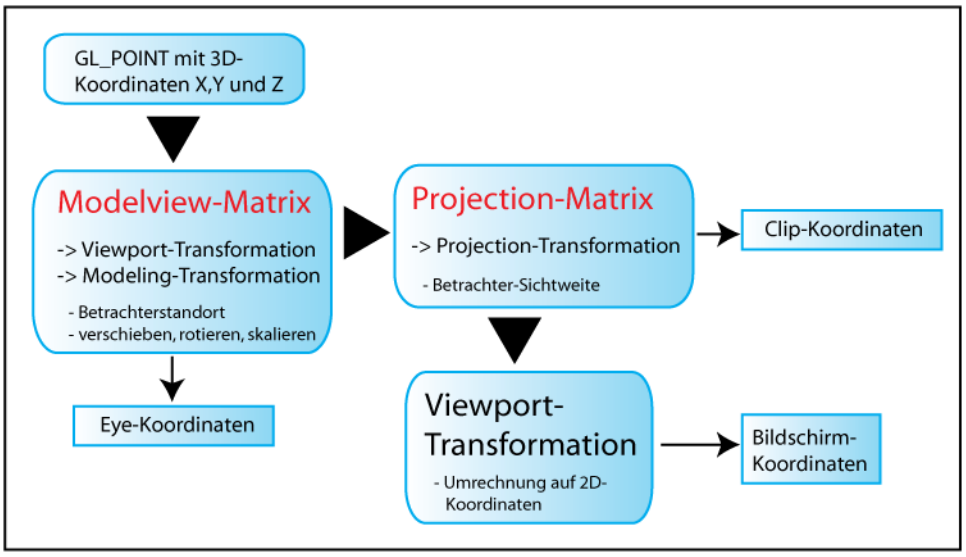


Abbildung 2.5: Transformationswege eines 3D-Primitives [ND97], Seite 67

2.3.6 Verschieben, Rotieren, Skalieren

Mit den 3 Befehlen `glTranslate, glScale und glRotate` lassen sich fast alle erdenklichen Darstellungen eines Primitives in OpenGL erzeugen. Mit dem Befehl `glTranslate` (siehe Abbildung 2.6) lässt sich das lokale Koordinatensystem eines Primitives verschieben und an bestimmten Stellen im Weltkoordinatensystem platzieren. Dabei ist wie bei den anderen 2 Befehlen zu berücksichtigen, dass ein Translations-Befehl auch für alle folgenden Primitive gilt. Außerdem wirken mehrere Translationsbefehle hintereinander kumulativ[13], welches auch für `glRotate` und `glScale` gilt. Wird beispielsweise ein Primitive um eine Einheit nach rechts verschoben, und darauf ein weiteres Primitive um 2 Einheiten nach rechts, dann wird letzeres schließlich um 3 Einheiten nach rechts verschoben. Mit `glScale` dagegen wird ein lokales Koordinatensystem eines Primitives um einen bestimmten Faktor skaliert. Dieser kann separat für X, Y und Z angegeben werden. Ein Faktor < 1 ergibt ein stauchendes Ergebnis und ein Faktor > 1 hat ein Stretchen (Auseinanderziehen) zur

[13] kumulativ = Hier: gesammelt, anwachsend

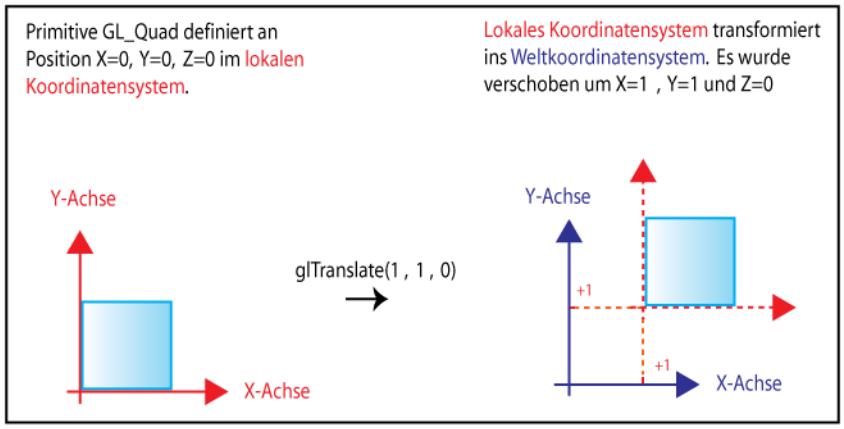


Abbildung 2.6: Primitive verschieben mit glTranslate [ND97], Seite 77

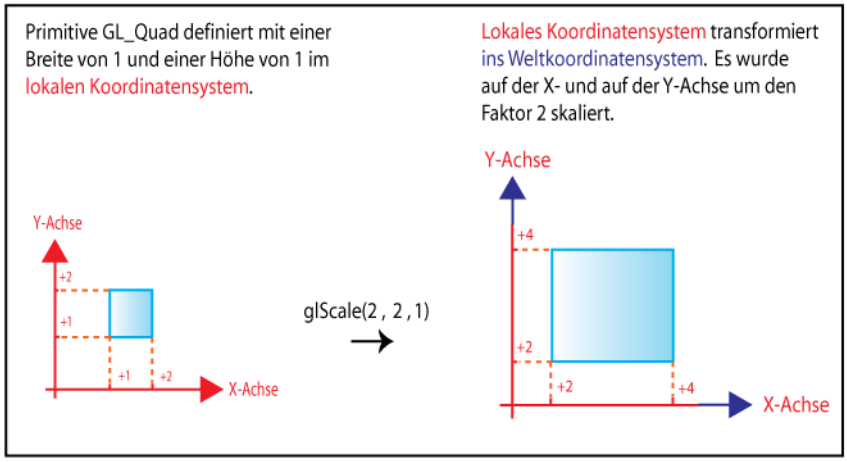


Abbildung 2.7: Primitive stauchen/stretchen mit glScale

Folge (siehe Abbildung 2.7). Der letze Befehl `glRotate` lässt ein beliebiges Primitive um einen bestimmten Winkel rotieren. Dabei wird als 1. Parameter der Drehwinkel in Grad angegeben. Danach folgen 3 Parameter, welche die Drehachsen X, Y und Z darstellen, um die rotiert werden soll. Sie bekommen den Wert 1, wenn um diese Achse rotiert werden soll und den Wert 0, wenn dort keine Rotation gewünscht ist. Dabei bezieht sich die Rotation immer auf den *Ursprung* des lokalen Koordinatensystems und erfolgt gegen den Uhrzeigersinn. Eine wichtige Anmerkung sei hier, dass es unterschiedliche Ergebnisse bei der Kombination von `glTranslate` und `glRotate` gibt, je nachdem in welcher Reihenfolge sie kombiniert werden. Bei der Rotation wird nicht das Objekt selbst rotiert, sondern dessen lokales Koordinatensystem. Somit wirkt ein nachfolgender Translationsbefehl anders, als wenn er vorangestellt worden wäre.[14]

[14] Vgl. [ND97], Seite 74

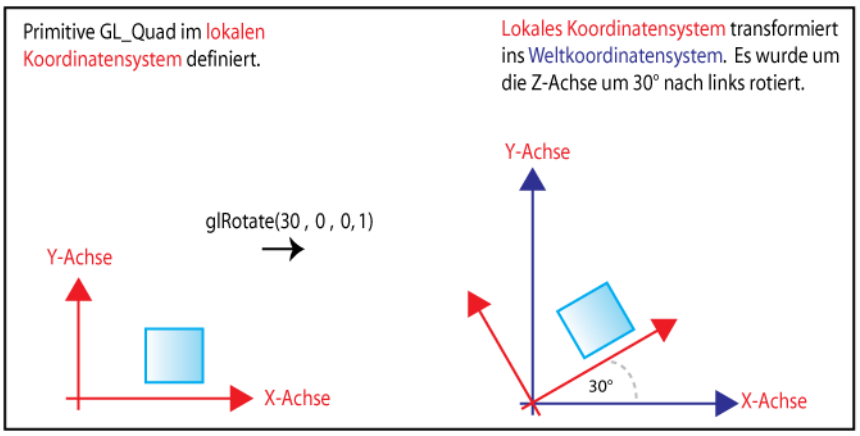


Abbildung 2.8: Primitive um Ursprung rotieren mit glRotate [ND97], Seite 77

2.3.7 Kamera

Da OpenGL einen nur verhältnismäßig geringen Umfang an Funktionen und Befehlen zu Verfügung stellt, welche sehr einfache Operationen darstellen, ist es mühsam, komplexere Dinge darzustellen, wie zum Beispiel eine Kugel oder andere Dinge, die aus den Primitiven zusammengesetzt werden müssen. Auch eine direkte Kamerafunktion ist nicht implementiert um die Sicht des Betrachters dynamisch behandeln zu können. Glücklicherweise gibt es eine Bibliothek die auf OpenGL aufsetzt und so Grafikfunktionen höherer Ebene bereitstellt. Sie nennt sich OpenGL Utitlity Library (GLU) und wird standardmäßig mit der OpenGL-Implementierung ausgeliefert. Sie beinhaltet unter anderem die Funktion `gluLookAt`, welche die Simulierung einer Kamera realisiert.

```
gluLookAt(eyeX, eyeY, eyeZ, centerX, centerY, centerZ,
                                          upX, upY, upZ)
```

Die ersten 3 Parameter `eyeX`, `eyeY` und `eyeZ` stellen den Standort der Kamera dar. Sie lässt sich dabei beliebig im Weltkoordinatensystem aufstellen. Die 3 weiteren Werte geben einen Zentrumspunkt an, auf welchen geblickt wird. Somit entsteht eine Sichtlinie, vom Standort zum Zentrum auf der entlang geschaut wird. Die Länge der Sichtlinie ist dabei aber unerheblich, nur dessen Richtung ist relevant. Die letzte Parametergruppe definiert einen sogenannten *UP-Vektor*. Er gibt die Richtung der Oberseite der Kamera an, weil ohne diese die Funktion nicht wissen kann, wo gerade oben und unten der Kamera ist (siehe Abbildung 2.9).[15]

[15] Vgl. [ND97], Seite 82-85

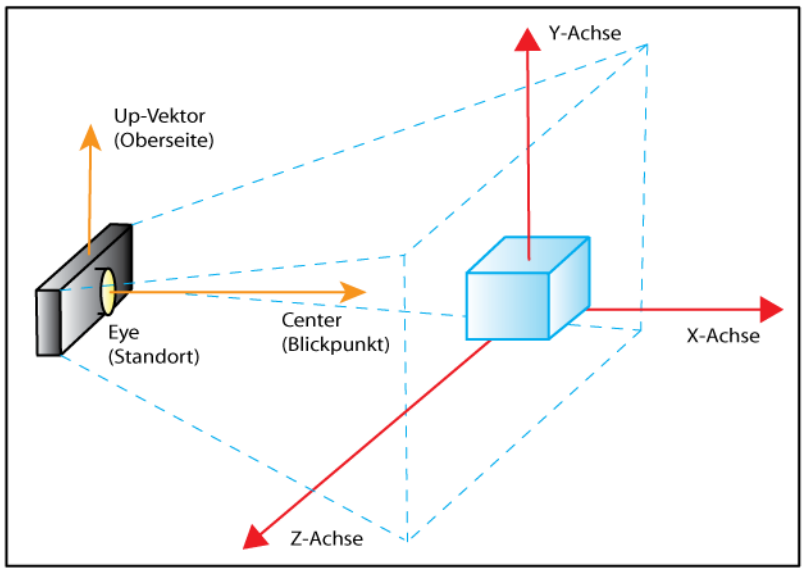


Abbildung 2.9: Kamerarealisierung mit gluLookAt [ND97], Seite 83

2.3.8 Farben und Texturen

Um Primitive mit einer Farbe zu versehen, steht in OpenGL die Funktion `glColor` zur Verfügung.[16] Sie kann unter anderem mit dem RGB-Farbmodell[17] arbeiten, und bekommt in diesem Fall 3 verschiedenen Farben als Parameter übergeben. Hierbei können die 3 Farbanteile Rot, Grün und Blau beliebig gemischt werden. Der Farbanteil einer Farbe liegt im Wertebereich 0 bis 255, wobei der Wert 255 den größten Farbanteil darstellt. Es werden somit 3 Farbanteile im Parameter von `glColor` angegeben, mit denen sich ca. 16,5 Millionen verschiedene Farben erzeugen lassen. Es reicht aus, die Funktion einmalig vor der Erstellung eines Primitives aufzurufen. Die eingestellte Farbe gilt darauf solange für alle Primitive, bis diese erneut verändert wird. Auch ist es möglich, die Farbe mehrfach *während* der Erzeugung eines Primitives zu wechseln. Da ein Primitive durch Vertices definiert wird, kann so die Farbe vor jeder Definition eines einzelnen Vertex geändert werden, welches Farbverläufe ergibt. Oft reicht aber ein Farbverlauf nicht aus, um bestimmte Oberflächen der Primitive darzustellen. Realistisch aussehende Materialien, wie beispielsweise Holz oder Eisen, lassen sich kaum mit einem Farbverlauf erzeugen. Hierfür müssen sogenannte Texturen verwendet werden, die sich wie eine Art Spanntuch vorzustellen sind. Eine Textur wird mit einer beliebigen Grafik aus einer Grafikdatei belegt und kann dann über gewünschte Vertices eines Primitives gespannt werden. Diese äußerst dynamische Methode wird sehr häufig in der Praxis verwendet, da sie nicht nur zum Darstellen von Oberflächen dienen kann, sondern auch eingesetzt wird, um bestimmte Objekte einer Welt zu simulieren. Beispielsweise kann auf eine 3D-Wand

[16] Vgl. [ND97], Seite 64
[17] Additives Farbmodell mit den Farben **R**ot, **G**rün und **B**lau

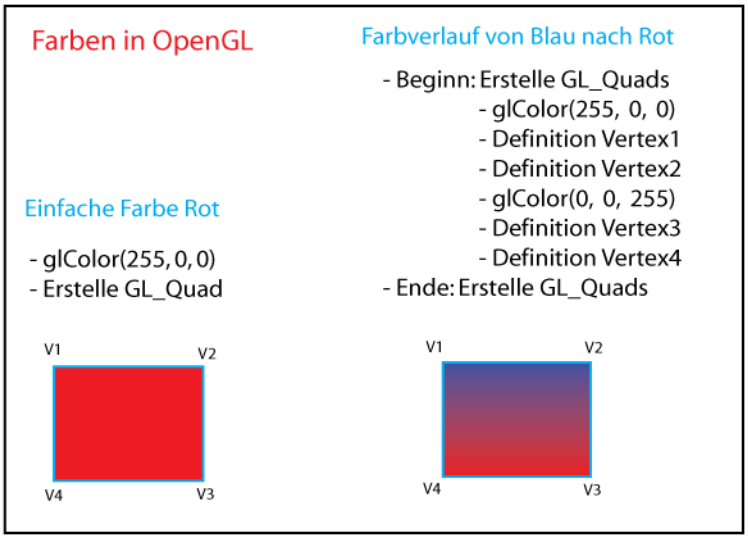


Abbildung 2.10: Farben mit glColor in OpenGL

eine Grafik von einer Tür aufgespannt werden, welche sonst mühsam aus einzelnen Primitiven nachgebaut werden müsste. Da die Benutzung von Texturen komplexerer Natur ist und der Rahmen der OpenGL-Grundlagen überschaubar gehalten werden soll, wird auf deren Beschreibung vollständig verzichtet. Auch weitere Funktionen wie Licht- und Schattendarstellungen sollten bei Bedarf nachgelesen werden. Als beste Informationsquelle dient hier wieder das OpenGL Redbook ([ND97]).

2.3.9 Ein erstes OpenGL Programm

In diesem Kapitel wird nun ein Codebeispiel vorgestellt, wie ein komplettes OpenGL-Programm erstellt werden kann. Es ist zwar nicht lauffähig, stellt aber den üblichen Aufbau eines OpenGL-Programmes dar. Ein lauffähiges Programm würde noch einen grafischen Zeichenbereich benötigen, auf dem OpenGL seine Grafik darstellen kann. Da dieser aber betriebssystemspezifisch programmiert werden muss, wird er hier außen vor gelassen. Der Programmcode für viele lauffähige OpenGL-Programme, zum Beispiel für Windows, ist unter [Com07b] zu finden. Die vorigen Kapitel haben wichtige Grundbefehle von OpenGL aufgezeigt, mit denen es möglich ist, eine kleine 3D-Welt aufzubauen. Um nun ein vollständiges Programm zu realisieren ist es notwendig, OpenGL mit Hilfe der Matrizen richtig einzustellen und die Befehle anzuwenden. Bisher wurde nur Pseudocode verwendet, dieses Beispiel wird sich aber nun konkret auf die Programmiersprache Java beziehen. Es zeigt, wie ein einfaches rotes Dreieck dargestellt werden kann, welches im Raum um die Y-Achse rotiert.

```
1 glShadeModel(GL_SMOOTH);
2 glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
3 glClearDepth(1.0f);
4 glEnable(GL_DEPTH_TEST);
```

```
5  glDepthFunc(GL_LEQUAL);
6  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
7  float ratio = 320 / 200;
8  glViewport(0, 0, 320, 200);
9  glMatrixMode(GL_PROJECTION);
10 glLoadIdentity();
11 gluPerspective(45.0f, ratio, 0.1, 1000.0);
12 glMatrixMode(GL_MODELVIEW);
13
14 glColor3f(255.0f , 0.0f, 0.0f);
15 float angle = 0.0f;
16 while(true)
17 {
18         glLoadIdentity();
19         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
20         glTranslatef(0.0f, 0.0f,-10.0f);
21         glRotatef(angle+=1.0f,0.0f, 1.0f,0.0f);
22         glBegin(GL_TRIANGLES);
23                 glVertex3f( -1.0f, -1.0f, 0.0f);
24                 glVertex3f(  1.0f, -1.0f, 0.0f);
25                 glVertex3f(  0.0f, 1.0f,  0.0f);
26         glEnd();
27 }
```

Hier sind einige neue Befehle dazugekommen, welche aber hauptsächlich nur der Initialisierung von OpenGL dienen. Auffällig ist, dass die bereits bekannten Grundfunktionen wie beispielsweise `glRotate` nun ein Suffix[18] tragen. Es besteht aus der Kombination von einer Zahl und einem Buchstaben. Die Zahl steht für die Anzahl der Parameter, die von der Funktion erwartet werden. Der Buchstabe, hier als „f“ auftretend, steht für den Datentyp, den die Parameter besitzen müssen. In diesem Fall ist es der Datentyp `float`. Durch solch ein Suffix kann in OpenGL zwischen mehreren Versionen einer Funktion unterschieden werden, von denen es recht viele gibt. Es hat den Sinn, dass so möglichst viele verschiedene Eingabeparameter-Typen möglich sind, um eine flexiblere Handhabung der Funktionen zu gewährleisten. Die ersten 12 Zeilen werden im Folgenden kurz beschrieben. Sie dienen dem richtigen Einstellen von OpenGL und sind am Anfang von fast jedem OpenGL-Programm zu finden.

[18] Eine an ein Wort angehängte Zeichenfolge

1. `glShadeModel(GL_SMOOTH);`

   Es wird eine Färbetechnik ausgewählt. `GL_SMOOTH` bedeutet hierbei, das Farbverläufe möglich sind und berücksichtigt werden. [19]

2. `glClearColor(0.0f, 0.0f, 0.0f, 0.0f);`

   Diese Funktion definiert einen RGB-Wert zum löschen des Anzeigebereichs (Viewport), auf dem später die Primitive gezeichnet werden sollen. Meistens wird dort die Farbe schwarz definiert. Der 4. Parameter stellt hierbei keine Farbe dar, sondern einen sogenannten Alpha-Wert. Dieser ist für Transparenz-Berechnungen zuständig, und wird mit dem Wert 0.0 in diesem Beispiel außer Acht gelassen.[20]

3. `glClearDepth(1.0f);`

   Diese und die nächsten 2 Zeilen beziehen sich auf einen sogenannten Tiefenpuffer. OpenGL speichert dort zu jedem Primitive Informationen über deren Tiefenlage (Negative Z-Achse). Das ist wichtig, damit sich eventuell hintereinander liegende Primitive richtig überlagern und zu weit entfernte Primitive nicht gezeichnet werden. `glClearDepth(1.0f)` bedeutet, der Tiefenpuffer soll vollständig gelöscht werden, wenn der entsprechende Befehl zum löschen verwendet wird. [21]

4. `glEnable(GL_DEPTH_TEST);`

   `glEnable(GL_DEPTH_TEST)` schaltet dann den Tiefentest ein, welcher dann Tiefenvergleiche von Primitiven mithilfe des Tiefenpuffers tätigt.

5. `glDepthFunc(GL_LEQUAL);`

   Mit `glDepthFunc(GL_LEQUAL)` wird zuletzt eine Bedingung für den Tiefentest erstellt, wann ein Primitive gezeichnet wird und wann nicht. `GL_LEQUAL` ist hier die Standardeinstellung und sollte so übernommen werden.[22]

6. `glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);`

   Diese Zeile steuert das Verhalten von OpenGL. Es wird dabei nur ein Vorschlag abgegeben, der aber nicht von OpenGL eingehalten werden muss. Mit `GL_PERSPECTIVE_CORRECTION_HINT` wird eine perspektivisch korrekte Farbinterpolation von Farbverläufen und Texturen verlangt, welches aber mehr Rechenleistung erfordert. Mit `GL_NICEST` soll OpenGL besonderen Wert auf die Qualität der Perspektivendarstellung legen.[23]

[19] Vgl. [ND97], Seite 68
[20] Mehr Details hierzu in [ND97], Seite 64-65
[21] Mehr Details hierzu in [ND97], Seite 124
[22] Mehr Details hierzu in [ND97], Seite 126
[23] Mehr Details hierzu in [ND97], Seite 89

7. `float ratio = 320 / 200;`

   Das Verhältnis von Bildschirmbreite (320 Pixel) zu Bildschirmhöhe (200 Pixel) wird in der Variablen ratio abgespeichert. Sie wird später in Zeile 11 bei `gluPerspective(...)` verwendet werden.

8. `glViewport(0, 0, 320, 200);`

   Hier wird die Größe und Lage des Viewports definiert. Er befindet sich hier an den X, Y-Koordinaten 0, 0 und hat ein Ausmaß von 320 x 200 Pixeln.

9. `glMatrixMode(GL_PROJECTION);`

   Mit `glMatrixMode` kann eine Matrix aktiv geschaltet werden. In diesem Fall wird die bereits bekannte Projection-Matrix aktiviert.

10. `glLoadIdentity();`

    Dieser Befehl setzt die aktuelle, aktive Matrix zurück. Das bedeutet, dass alle eventuellen Veränderungen an der Matrix auf einen bestimmten Ursprungszustand zurückgesetzt werden. Die Matrix wird mit einer Identitätsmatrix überschrieben, die auch als Einheitsmatrix bekannt ist. Ein tieferes Verständnis der dahinter steckenden linearen Algebra ist aber für die Anwendung nicht notwendig.

11. `gluPerspective(45.0f, (float)320/200, 0.1f, 1000.0f);`

    Mit Hilfe diesen Befehls wird ein Betrachtungsfeld definiert. Diese Einstellungen verändern unmittelbar die Projection-Matrix. Der erste Parameter gibt den Winkel des Betrachtungsfeldes entlang der Y-Achse in Grad an. Mithilfe des 2. Parameters, dem Verhältnis von Breite zu Höhe des Viewports, kann so eindeutig ein Betrachtungsfeld generiert werden, welches der Betrachter sieht. Die letzten beiden Parameter bestimmen, wie weit Primitive sichtbar sind. Das am nähesten gelegene Primitive, dass noch sichtbar für den Betrachter ist, liegt auf der negativen Z-Achse bei dem Wert 0.1. Ein Primitive welches eine Entfernung größer als 1000 besitzt, ist nicht mehr sichtbar. [24]

12. `glMatrixMode(GL_MODELVIEW);`

    In dieser Zeile wird die Modelview-Matrix aktiv geschaltet. Es können hiernach Zeichenoperationen erfolgen, welche die Modelview-Matrix beeinflussen.

Die Zeilen 14-21 sollten nun fast selbsterklärend sein. Sie beinhalten die Hauptschleife des Programmes, in dem stetig ein Dreieck gezeichnet wird, desses Rotationswinkel sich erhöht. Wichtig ist hier, dass bei jedem Schleifendurchgang der Tiefenpuffer und der Viewport mit `glClear` gelöscht wird. Somit wird das Dreieck oft hintereinander an versetzter Position dargestellt, welches dann die Animation ergibt. Der

[24] Mehr Details hierzu in [ND97], Seite 43

erste Befehl `gLoadIdentity` der Schleife ist notwendig, damit die ModelView-Matrix und somit die Translations- und Rotationseinstellungen zurückgesetzt werden. Ohne `gLoadIdentity` würde beispielsweise das Dreieck immer weiter nach hinten verschwinden, da sich die Verschiebungen der Translationsbefehle addieren.

```
14 glColor3f(255.0f , 0.0f, 0.0f);
15 float angle = 0.0f;
16 while{true}
17 {
18         glLoadIdentity();
19         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
20         glTranslatef(0.0f, 0.0f,-10.0f);
21         glRotatef(angle+=1.0f,0.0f, 1.0f,0.0f);
22         glBegin(GL_TRIANGLES);
23                 glVertex3f( -1.0f, -1.0f, 0.0f);
24                 glVertex3f(  1.0f, -1.0f, 0.0f);
25                 glVertex3f(  0.0f, 1.0f,  0.0f);
26         glEnd();
27 }
```

2.4 Weitere Funktionalitäten

Wie aus der Einführung in die Grundlagen von OpenGL hervorgeht, ist OpenGL eine ausgereifte und gut strukturierte API. Die beschriebenen Grundlagen reichen bereits aus, um eine kleine 3D-Welt aufzubauen. Auch ist zu erkennen, das OpenGL sehr funktional gehalten ist und keine objektorientierten Ansätze aufweist. Somit sind kleine, einfache Anwendungen schnell zu programmieren. Wächst aber die Komplexität und die Anzahl von Primitiven, so kann ein Programm schnell unübersichtlich werden. Die Kombination aber von Java und OpenGL, welche in dieser Diplomarbeit verwendet werden soll, dürfte bezüglich objektorientierter Programmierungsmöglichkeiten keine Wünsche offen lassen. Da OpenGL in seiner mittlerweile aktuellen Version 2.1 noch viele weitere wichtige Features besitzt, welche weit über diese Grundlagen hinaus gehen, werden die Wichtigsten der Vollständigkeit halber noch kurz aufgelistet:[25]

- **Texture Mapping**
  Primitive werden mit 2-dimensionalen Grafiken (Texturen) belegt. Dieser Vorgang wird auch „Mapping“ genannt. So können realistisch aussehende Oberflächen wie beispielsweise Stein oder Eisen erzeugt werden.

[25] Nachzuschlagen in [ND97] und [Com07b]

- **Lightening**
  OpenGL erlaubt es, verschiedene Lichtquellen an bestimmten Positionen zu platzieren. Das Licht kann dabei in beliebigen Farben definiert werden und auch die Art, wie beispielsweise diffuses Licht, ist auswählbar.

- **Alphablending**
  Mit Alphablending können Transparenzeffekte realisiert werden. Dabei kann eine Farbe eines Primitives zusätzlich noch einen Alpha-Wert, welcher die Stärke der Transparenz definiert, besitzen. So kann beliebig stark durch ein gefärbtes Primitiv geschaut werden, je nachdem wie der Alphawert gewählt wird.

- **Antialising**
  Ein Verfahren, bei dem die Kanten von Primitiven geglättet werden, denn oft treten unschöne, treppenartige Kanten auf, welche häufig bei dünnen, schrägen Linien zu finden sind.

- **Display Lists**
  OpenGL Zeichenbefehle, welche oft benötigt werden, können in so genannten Display-Listen gecached werden. Eine Befehlssequenz, die in einer Displayliste einmal definiert ist, kann dann beliebig oft ausgeführt werden, ohne das die einzelnen Befehle erneut an die Grafikkarte geschickt werden müssen.

- **Extensions**
  Extensions erweiterten OpenGL um Funktionen, die nicht in dessen Spezifikation definiert sind. Beispielsweise stellt der Grafikkartenhersteller NVIDIA einige OpenGL-Extensions in seiner mitgelieferten Treibersoftware zur Verfügung. Somit sind Extensions abhängig von jeweiligen OpenGL-Implementierungen.[26]

[26] Siehe hierzu auch: <http://developer.nvidia.com/object/opengl_extensions_tutorial.html>

3 Grundlagen von JOGL

3.1 Was ist JOGL ?

JOGL steht für Java OpenGL und erlaubt es, OpenGL mit der Programmiersprache Java zu verbinden. Schnittstellen solcher Art werden auch als Java Bindings bezeichnet. In seiner aktuellen Version 1.1.0 unterstützt es den Zugriff für alle Funktionen und Befehle von OpenGL in der Version 2.1. Dabei hält sich JOGL bei seiner Implementierung strikt an die Spezifikation JSR-231 (Java Binding for OpenGL) von Sun Microsystems, welches die offizielle Spezifikation der Schnittstelle von Java zu OpenGL darstellt. Java OpenGL wurde ursprünglich von Kenneth Russel und Chriss Kline unter dem namen Jungle begonnen, wird aber mittlerweile sogar von der Game Technology Group, welche zu Sun Microsystems gehört, unter dem Namen JOGL mit weiterentwickelt. Somit stehen die Chancen recht hoch, dass JOGL die offizielle Implementierung von JSR-231 sein wird und somit in der nächsten Java Version enthalten ist.[1]

3.2 Grundlagen der Programmierung von JOGL

3.2.1 Funktionsweise von JOGL

JOGL stellt eine externe Java-Bibliothek dar, über die ein Zugriff auf die OpenGL-Funktionen, welche in nativem Code vorliegen, möglich ist. Dabei ist JOGL fast ausschließlich in Java realisiert und enthält nur ca. 150 Zeilen reinen C-Code. Dieser ist teilweise nötig, um einige Fehler in den Treiber-Implementierungen von OpenGL zu umgehen.[2] Damit Java überhaupt in der Lage ist, Funktionen von OpenGL aufzurufen, muss es das JNI (Java Native Interface) verwenden, welches die Schnittstelle zwischen nativen Code und dem plattformunabhängigen Java-Code darstellt. Für jede OpenGL-Funktion, welche in C geschrieben wurde und aufgerufen werden soll, muss dementsprechender JNI-Code vorliegen. Da aber OpenGL eine ganze Fülle an Funktionen bereitstellt, würde es einen enormen Aufwand bedeuten, den zuständigen JNI-Code zu erzeugen. Aus diesem Grund wurde ein Automatisierungstool namens „gluegen“ entwickelt, mit dem es möglich ist, Teile des Quellcodes von OpenGL durchzuscannen und den nötigen JNI-Code für JOGL automatisch zu generieren. Gluegen[3]

[1] Vgl. [Com07c] und [RKZ07], Seite 16
[2] Vgl. [Dev07c]
[3] Vgl. [RKZ07], Seite 15

stellt somit weitgehendst den Kern der Technik von JOGL dar, welches zudem noch den Vorteil verschafft, JOGL schnell auf dem neuesten Stand zu halten, falls neue Funktionen in OpenGL hinzukommen oder sich verändern sollten. (Abbildung 3.1).

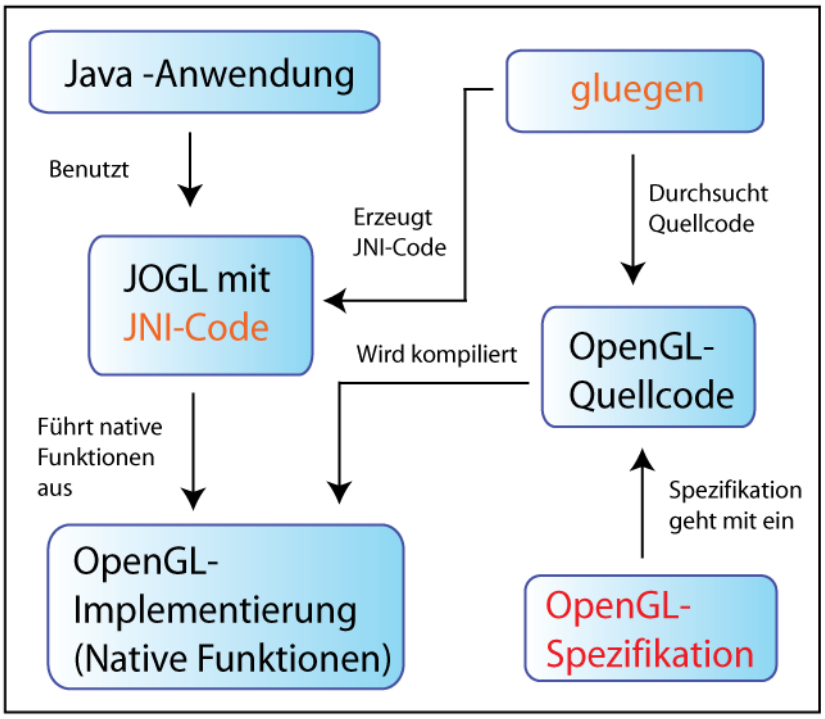


Abbildung 3.1: JOGL verwendet OpenGL über das JNI

3.2.2 Zusammenarbeit von JOGL und AWT/Swing

Da ein JOGL-Programm eine Zeichenfläche besitzen muss, auf der die OpenGL-Grafik gezeichnet wird, kann JOGL nicht ohne weiteres mit einer einfachen Java-Konsolen-Anwendung verwendet werden, welche nur Text darstellen kann. JOGL muss mit einer grafischen Benutzeroberfläche zusammenarbeiten, welche auch meist als GUI[4] bezeichnet wird. Eine GUI in Java verwendet dabei bekannte Formen, wie grafische Fenster, Scroll-Down Menüs usw., die auch von Windows bekannt sind. Um nun ein einfaches grafisches Fenster in Java zu erzeugen, gibt es 2 verschiedene Bibliotheken mit unterschiedlichen Schwerpunkten.

1. **AWT - Abstract Window Toolkit**
   Das Abstract Window Toolkit ist ein Teil der JFC (Java Foundation Classes), der Standard API für das Programmieren von GUI's unter Java. AWT stellt sogenannte Heavyweight-Komponenten bereit, mit denen plattformunabhängige Benutzeroberflächen programmiert werden. Die Komponenten werden deshalb so bezeichnet, weil die grafischen Komponenten des jeweiligen Betriebssystems

[4] Graphical User Interface

verwendet werden. Wird zum Beispiel ein Button mit AWT unter Windows angelegt, dann ist dies ein richtiger Windows-Button aus der Windows-Bibliothek. Es werden also native Elemente des aktuellen Betriebssystems verwendet und keine eigenen von Java. Da aber nicht jedes Betriebssystem die gleichen grafischen Komponenten besitzt, muss sich das AWT an den kleinsten gemeinsamen Nenner aller Betriebssysteme halten. Somit unterstützt das AWT nur sehr wenige, recht einfache Darstellungsmöglichkeiten von Komponenten, wie zum Beispiel einfache Fenster, Buttons und Listen. Trotz dieser starken Einschränkung ist das AWT aufgrund seiner Kompatibilität zur Java MicroEdition recht beliebt.

2. **Swing**
   Swing ist ebenfalls ein Teil der JFC und geht aber den entgegengesetzten Weg des AWT's. Hier werden alle grafischen Komponenten 100% in Java selbst erzeugt, wodurch diese als Lightweight-Komponenten bezeichnet werden. Diese Verfahrensweise stellt sicher, das alle Swing-Komponenten auf jedem Betriebssystem verfügbar sind. Somit gibt es in Swing eine Vielzahl von Möglichkeiten zur Gestaltung der GUI. Drag'n Drop, Layoutmanager und Tooltips sind nur einige der vielen Features die Swing unterstützt. Auch Komponenten zur Darstellung von Bäumen oder Tabellen werden hier bereitgestellt. Da die Komponenten selber von Java erstellt werden, besteht das Problem, das Swing-GUI's nicht dem Aussehen des aktuellen Betriebssystems gleichen. Dieses Erscheinungsbild kann aber durch sogenannte „Look and Feels" wieder korrigiert werden. Swing ist mittlerweile so ausgereift, dass bereits umfangreiche kommerzielle Produkte wie beispielsweise Poseidon[5] auf dem Software-Markt bestehen können.[6]

Es ist nun wichtig, die Richtige der beiden Alternativen auszuwählen, da JOGL 2 verschiedene Zeichenbereiche zur Darstellung der OpenGL-Grafik zur Verfügung stellt. Der erste heisst `GLCanvas` und arbeitet mit dem AWT zusammen. Hier erlaubt JOGL uneingeschränkte Hardwarebeschleunigung der Grafikkarte. Mit dem Zeichenbereich `GLJPanel` ist dagegen eine 100% kompatible Swing Integration möglich. Allerdings sind hier die Zeichenoperationen zwar hardwarebeschleunigt, aber doch langsamer als die des `GLCanvas`.[7] Für Anwendungen, welche viele grafische Steuerelemente verwenden (beispielsweise ein Photobearbeitungsprogramm mit Werkzeugleisten) ist der `GLJPanel` somit erste Wahl. Für umfangreiche 3D-Darstellungen, welche viele Rechenoperationen der Grafikkarte in Anspruch nehmen, ist dagegen der `GLCanvas` zu empfehlen. Da das zu entwickelnde Framework in dieser Diplomarbeit den Fokus auf die Darstellung von Physiksimulationen legt, welche schnell komplex und rechenin-

[5] Siehe [Gen07]
[6] Vgl. für Swing und auch AWT [Chr05], Seite 781
[7] Vgl. [Dev07c]

tensiv werden können, wird desweiteren auch nur auf die Verwendung des `GLCanvas` eingegangen.

3.2.3 Ein erstes JOGL Programm

GLCanvas

Folgendes Beispiel zeigt, wie in Java die Zeichenfläche `GLCanvas` in einem einfachen Fenster des AWT's erzeugt werden kann. Die grobe Funktionsweise des AWT's selber wird hier nur kurz beschrieben werden, um nicht von der eigentlichen Anbindung von JOGL zu weit abzuschweifen. Auch ist noch einmal ausdrücklich darauf hingewiesen, dass grundlegende Kenntnisse der Programmiersprache Java und Techniken wie Vererbung, Abstrakte Klassen, Interfaces usw. in dieser Diplomarbeit vorrausgesetzt werden.[8]

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import javax.media.opengl.*;
4
5  public class JoglApp
6  {
7      public static void main(String[] args)
8      {
9          Frame fenster = new Frame("Jogl Fenster");
10         GLCanvas canvas = new GLCanvas();
11         fenster.add(canvas);
12
13         fenster.addWindowListener(new WindowAdapter()
14         {
15             public void windowClosing(WindowEvent e)
16             {
17                 System.exit(0);
18             }
19         });
20
21         fenster.setSize(320,200);
22         fenster.setVisible(true);
23     }
24 }
```

In den ersten beiden Zeilen werden die Java-Pakete eingebunden, welche für die AWT-Darstellung nötig sind. Die Klassen/Methoden von JOGL, und somit auch

[8] Ausführliche Informationsquellen hierzu [Chr05]

der `GLCanvas`, stehen durch laden des Paketes `javax.media.opengl.*` zur Verfügung. In der `main(...)`-Methode, dem Einstiegspunkt des Programmes, wird ein Objekt vom Typ `Frame` erzeugt. Dies ist ein grafisches AWT-Fenster, welches darauf über den Befehl `add(canvas)` den zuvor erstellten `GLCanvas` zugewiesen bekommt. In Zeile 13 muss dem Fenster noch ein sogenannter „WindowListener“ hinzugefügt werden. Ein WindowListener wacht über bestimmte Ereignisse, die in einem AWT-Fenster auftreten können. Bei einem bestimmten Ereignis, wird automatisch eine dazugehörige Methode ausgeführt. In diesem Fall wird überwacht, wann ein Schließen des AWT-Fensters durch den Benutzer geschieht. Die dafür zuständige Methode lautet `windowClosing(...)` und führt in diesem Fall `System.exit(0)` aus, welches das Java-Programm beendet. Ohne den Listener wäre es auf normalen Wege (Beispielsweise auf dem Kreuz in der rechten, oberen Ecke des Fensters) nicht möglich, das grafische AWT-Fenster zu schließen. Mit `setSize(320,200)` wird anschließend noch die Größe des Fensters definiert und zuletzt mit `setVisible(true)` auf dem Bildschirm sichtbar gemacht. Wie zu erkennen ist, ist es relativ unkompliziert, einen `GLCanvas`, auf dem die OpenGL-Zeichenoperationen später stattfinden sollen, in Java einzubinden.

GLEventListener

Um OpenGL-Befehle verwenden zu können, muss dem `GLCanvas` ein GLEventListener hinzugefügt werden. Er ist für alle Ereignisse des `GLCanvas` zuständig und stellt den eigentlichen Einstiegspunkt zum Arbeiten mit OpenGL dar. Um nun einen eigenen `GLEventListener` zu erstellen, muss dieser das Interface `GLEventlistener` implementieren.

```
1  import javax.media.opengl.*;
2
3  public class JoglEventListener implements GLEventListener
4  {
5      public void init(GLAutoDrawable glDrawable) {}
6
7      public void reshape(GLAutoDrawable glDrawable,
8                          int x, int y,int width, int height)
9      {}
10
11     public void display(GLAutoDrawable glDrawable){}
12
13
14     public void displayChanged(GLAutoDrawable glDrawable,
15                                boolean modeChanged,
16                                boolean deviceChanged)
```

```
17      {}
18  }
```

Die 4 vorhandenen Methoden sind fest durch das Interface `GLEventListener` vorgeschrieben und werden je nach Ereignis automatisch aufgerufen.

- **init(...)-Methode**
  Die `Init(...)`-Methode wird automatisch einmalig beim Starten des Java-Programmes aufgerufen. Hier sollten alle nötigen OpenGL-Initialisierungen vorgenommen werden. Sie ist somit die erste der 4 Methoden, die aufgerufen wird.

- **reshape(...)-Methode**
  Verändert sich die Größe des AWT-Frames und somit die Größe des `GLCanvas`, wird diese Methode aufgerufen. So kann auf Größenänderungen des Zeichenbereichs, rechtzeitig reagiert werden. Sie wird allerdings automatisch einmalig direkt nach der `init()`-Methode aufgerufen, obwohl hier in dem Sinne noch keine Größen*änderung* stattgefunden hat, sonder nur eine Größendefinition durch `setSize(320,200)`.

- **display(...)-Methode**
  Dieses ist die wichtigste Methode. Sie wird später die OpenGL-Hauptschleife darstellen, welche permanent aufgerufen wird. Die eigentlichen Zeichenbefehle von OpenGL sollten somit hier ihren Platz einnehmen, um ständig wiederholt werden zu können. Ohne eine Hauptschleife wäre es nicht möglich, Animationen darzustellen oder beispielsweise Berechnungen zu aktualisieren. Problem ist nur, dass die `display(...)`-Methode standardmäßig nur ein einziges mal direkt nach der `init()`-Methode ausgeführt wird. Weitere Aufrufe müssen manuell durchgeführt werden.

- **displayChanged**
  Wird die Anzeige auf dem Bildschirm auf einer andere Auflösung umgestellt, tritt diese Methode in Kraft.

Ist der `GLEventlister` fertiggestellt, kann dieser schließlich dem `GLCanvas` hinzugefügt werden und ist somit registriert.

```
canvas.addGLEventListener(new JoglEventListener());
```

Animator

Wenn der Zeichenbereich definiert und ein `GLEventListener` registriert wurde, ist der Grundaufbau eines JOGL Programmes weitgehendst fertiggestellt. Doch es gibt noch ein wichtiges Element, welches nicht fehlen darf. Die `display(...)`-Methode des `GLEventListeners`, welche die Hauptschleife darstellt, wird wie bereits erwähnt

nicht automatisch permanent aufgerufen. Hier ist ein manuelles Aufrufen notwendig. Das hat den Vorteil, dass so eine volle Kontrolle über die Hauptschleife möglich ist. Ist das AWT-Fenster mit dem `GLCanvas` beispielsweise gerade durch ein weiteres AWT-Fenster verdeckt, wäre es unnötig, die OpenGL-Zeichenoperationen weiterhin fortzuführen und Ressourcen zu verbrauchen. In einfachen Programmen aber wird diese flexible Handhabungsmöglichkeit der Hauptschleife oft nicht benötigt. Deshalb stellt JOGL einen sogenannten `Animator` bereit, der die einfache Aufgabe hat, innerhalb eines separaten Threads ständig die `display(...)`-Methode des `GLEventListener` aufzurufen. Folgendes Code-Beispiel zeigt noch einmal die Klasse `JOGLApp` mit Anbindung des `GlEventlisteners` und dem Einsatz des `Animators`.

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import javax.media.opengl.*;
4  import com.sun.opengl.util.*;
5
6  public class JoglApp
7  {
8      public static void main(String[] args)
9      {
10         Frame fenster = new Frame("Jogl Fenster");
11         GLCanvas canvas = new GLCanvas();
12         canvas.addGLEventListener(new JoglEventListener());
13
14         final Animator anim = new Animator(canvas);
15
16         fenster.add(canvas);
17
18         fenster.addWindowListener(new WindowAdapter()
19         {
20             public void windowClosing(WindowEvent e)
21             {
22                 new Thread(new Runnable()
23                 {
24                     public void run()
25                     {
26                         anim.stop();
27                         System.exit(0);
28                     }
29
30                 }).start();
```

```
31                  }
32              });
33
34              fenster.setSize(320,200);
35              fenster.setVisible(true);
36              anim.start();
37          }
38      }
```

Um den Animator benutzen zu können, ist ein zusätzlicher Import des Jogl-Paketes `com.sun.opengl.util.*` nötig. In Zeile 14 wird darauf ein `Animator` mit dem Namen `anim` erstellt und über seinen Konstruktor mit dem `canvas`-Objekt verknüpft. Der Zusatz `final` ist wichtig, damit der Animator auch innerhalb der `windowClosing`-Funktion des WindowListeners platziert werden darf. Das liegt daran, dass im Parameter von `addWindowListener(...)` eine eingebettete, anonyme Klasse definiert wird. Objekte und Variablen die außerhalb definiert wurden, haben hier, mit Ausnahme von static- und final-Variablen, keinen Eintritt (Vgl. [Mic07b], Kapitel 8). In der neu hinzugefügten Zeile 37 wird schließlich der Animator gestartet, womit dieser in einem separaten Thread läuft. Eine wichtige Neuerung tritt in der Ereignis-Methode `windowClosing(...)` auf. Hier wird vor dem Beenden des Programmes der Animator-Thread mit der Methode `stop()` angehalten. Das ist wichtig, damit OpenGL nicht noch am Zeichnen ist, während das Programm geschlossen wird, und so eventuelle Abstürze hervorruft. Damit auch wirklich sichergestellt wird, dass der Animator rechtzeitig gestoppt ist, wird das Stoppen wiederum in einem eigenen Thread ausgeführt. Tests ergaben zwar, dass es keinerlei Probleme gibt, wenn der `stop()`-Befehl nicht in einem eigenen Thread eingebettet ist, aber laut der JOGL-API Dokumentation sollte das Stoppen nicht *direkt* in dem Listener erfolgen (vgl.([Dev07a], Animator). Die Folgende Abbildung 3.2 verdeutlicht noch einmal den Aufbau eines JOGL-Programmes:

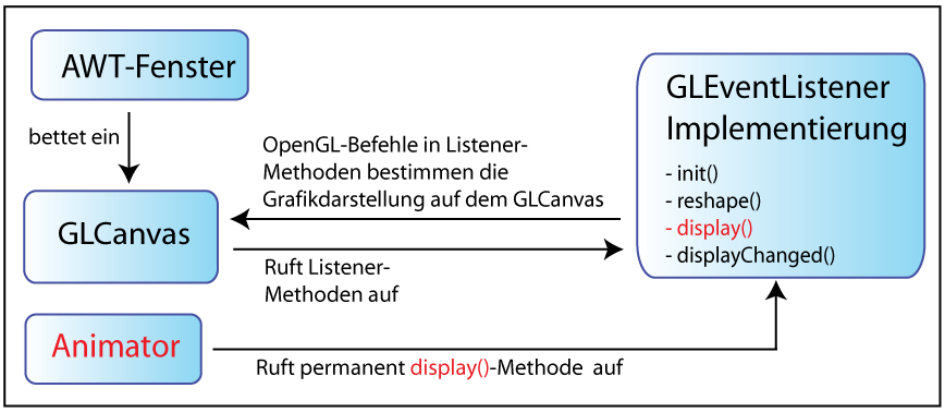


Abbildung 3.2: Interner Ablauf eines JOGL Programmes

OpenGL Befehlssatz verwenden

Wie an den 4 Methoden aus dem `GLEventListener` zu erkennen ist, besitzen alle einen gemeinsamen Parameter `GLAutoDrawable`. Dieses stellt ein Interface dar, welches von `GLCanvas` implementiert wird. Über dieses Interface lassen sich somit einige Methoden des zugehörigen `GLCanvas` aufrufen. Die wichtigste Methode heisst `getGL()` und gibt ein `GL`-Objekt zurück, das auch als *OpenGL Pipeline* bezeichnet wird. `GL` stellt somit schließlich die eigentliche Schnittstelle zu OpenGL dar. Hierüber können alle OpenGL-Befehle bis zur Version 2.1 „abgeschickt“ bzw. ausgeführt werden. Wichtig ist zu erwähnen, dass `getGL()` nur innerhalb der 4 obligatorischen Methoden[9] benutzt werden darf. Außerhalb dieser kann nicht gewährleistet werden, dass ein Wert zurückgegeben wird. Auch wird empfohlen, das GL-Objekt nicht zwischenzuspeichern, sondern bei *jedem* Aufruf der einzelnen 4 Methoden mit `getGL()` erneut anzufordern (vgl.([Dev07a], GLAutoDrawable). Die nützlichen Befehle der GLU-Bibliothek sind ebenfalls verfügbar und können einfach über ein `GLU`-Objekt, dass mit `GLU gluObjekt = new GLU()` angelegt wird, verwendet werden. Folgende 2 Klassen zeigen nun ein vollständiges, lauffähiges JOGL-Programm, welches das OpenGL-Programm aus Kapitel 2.3.9 S. 23 umsetzt:

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import javax.media.opengl.*;
4  import com.sun.opengl.util.*;
5
6  public class JoglApp
7  {
8      public static void main(String[] args)
9      {
10         Frame fenster        = new Frame("Jogl Fenster");
11         GLCanvas canvas      = new GLCanvas();
12         canvas.addGLEventListener(new JoglEventListener());
13         final Animator anim = new Animator(canvas);
14         fenster.add(canvas);
15
16         fenster.addWindowListener(new WindowAdapter()
17         {
18             public void windowClosing(WindowEvent e)
19             {
20                 new Thread(new Runnable()
21                 {
22                     public void run()
```

[9] init(), reshape(), display() und displayChanged()

```
23                      {
24                          anim.stop();
25                          System.exit(0);
26                      }
27
28                  }).start();
29              }
30          });
31
32          fenster.setSize(320,200);
33          fenster.setVisible(true);
34          anim.start();
35      }
36  }
```

```
1  import javax.media.opengl.*;
2  import javax.media.opengl.glu.*;
3
4  public class JoglEventListener implements GLEventListener
5  {
6      private GL    gl          = null;
7      private float angle       = 0.0f;
8      private final GLU glu     = new GLU();
9
10     public void init(GLAutoDrawable glDrawable)
11     {
12         gl = glDrawable.getGL();
13
14         gl.glShadeModel(GL.GL_SMOOTH);
15         gl.glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
16         gl.glClearDepth(1.0f);
17         gl.glEnable(GL.GL_DEPTH_TEST);
18         gl.glDepthFunc(GL.GL_LEQUAL);
19         gl.glHint(GL.GL_PERSPECTIVE_CORRECTION_HINT,
20                                                 GL.GL_NICEST);
21
22     }
23
24     public void display(GLAutoDrawable glDrawable)
25     {
26         gl = glDrawable.getGL();
```

```
27
28          gl.glClear(GL.GL_COLOR_BUFFER_BIT |
29                     GL.GL_DEPTH_BUFFER_BIT);
30
31          gl.glLoadIdentity();
32          gl.glTranslatef(0.0f, 0.0f,-10.0f);
33
34          gl.glRotatef(angle+=1.0f,0.0f, 1.0f,0.0f);
35
36          gl.glBegin(GL.GL_TRIANGLES);
37              gl.glVertex3f( -1.0f, -1.0f, 0.0f);
38              gl.glVertex3f(  1.0f, -1.0f, 0.0f);
39              gl.glVertex3f(  0.0f, 1.0f,  0.0f);
40          gl.glEnd();
41      }
42
43      public void reshape(GLAutoDrawable glDrawable,
44                          int x, int y,int width, int height)
45      {
46          gl    = glDrawable.getGL();
47
48          if (height <= 0) height = 1;
49
50          final float ratio = (float)width / (float)height;
51
52          gl.glViewport(0, 0, width, height);
53
54          gl.glMatrixMode(GL.GL_PROJECTION);
55          gl.glLoadIdentity();
56
57          glu.gluPerspective(45.0f, ratio, 0.1, 1000.0);
58
59          gl.glMatrixMode(GL.GL_MODELVIEW);
60          gl.glLoadIdentity();
61          gl.glColor3f(255.0f , 0.0f, 0.0f);
62      }
63
64      public void displayChanged(GLAutoDrawable glDrawable,
65                                 boolean modeChanged,
66                                 boolean deviceChanged)
67      {
```

```
68        }
69  }
```

3.3 Alternativen zu JOGL und warum JOGL ?

Neben JOGL gibt es noch einige Alternativen, mit denen es möglich ist, OpenGL von Java aus anzusprechen. Auch sollte ein Vergleich mit Java3D nicht fehlen, welches ebenfalls eine Alternative darstellt. Im Folgenden werden die bekanntesten Alternativen mit deren Vorteilen und Nachteilen kurz beschrieben:

- **GL4Java (Frei verfügbar: Open Source)**
  GL4Java steht für OpenGL for Java und ist einer der ältesten und bekanntesten Bindings. Es läuft nahezu auf jeder Plattform und unterstützt die Integration von AWT und Swing. Da aber die Entwicklung bereits 2002 eingestellt worden ist, unterstützt es nur OpenGL bis zur Version 1.3 und ist somit auf die Zukunft gesehen keine Alternative.[10]

- **LWJGL (Frei verfügbar: Open Source)**
  LWJGL (Lightweight Java Game Library) ist ein sehr aktuelles Binding. Es wird ständig aktualisiert und unterstützt bereits OpenGL in der Version 2.0. Aufgrund seines prozeduralen Aufbaus ist es auch für Anfänger leicht zu bedienen und stellt außerdem Möglichkeiten zur Sound-Ausgabe bereit. LWJGL wird sehr klein gehalten und beinhaltet nur das Nötigste, weil es für die Zukunft auch auf kleinen Endgeräten wie beispielsweise Handys funktionieren soll. Ein wichtiger Makel ist aber, das LWJGL keine Integration mit AWT oder Swing zulässt. Ein Blick in Code-Beispiele verrät den Grund. LWJGL stellt bereits komplette Komponenten wie zum Beispiel ein Hauptfenster bereit, in dem die OpenGL Operationen geschehen müssen. Die Gestaltung einer eigenen Oberfläche ist somit über AWT oder Swing sehr eingeschränkt bis unmöglich.[11]

- **Mesa 3D (Frei verfügbar: Open Source)**
  Mesa 3D stellt eine weitere Alternative dar, um OpenGL mit Java zu verbinden. Zwar soll Mesa nach einigen Artikeln zu Folge nicht mehr unterstützt und weiterentwickelt werden, nach den Informationen der aktuellen Mesa-Homepage wird dieses aber dennoch weiterentwickelt (Letzte Erneuerung Dezember 2006). In seiner aktuellen Version 6 unterstützt es aber nur den OpenGL Befehlssatz der Version 1.5 und hinkt daher anderen Bindings etwas hinterher.[12]

[10] Vgl. [RKZ07], Seite 11
[11] Vgl. [RKZ07], Seite 12 und [Dev07d]
[12] Vgl. [Dev07e]

- **Java3D (Frei verfügbar: Open Source)**
  Java3D ist eine objektorientierte API, welche aber auf einer höheren Ebene aufsetzt. Sie ist sehr umfangreich und besitzt über 100 Klassen, mit denen 3D-Darstellungen realisiert werden können. Hierbei besitzt Java3D ein ganz eigenes Modell zum Erstellen von geometrischen 3D-Objekten, welche in einem sogenannten Universum dargestellt werden. Um eine möglichst gute Performance zu erhalten, setzt es dabei intern auf OpenGL auf. Auch die interne Verwendung von DirectX ist möglich, benötigt aber zusätzliche Installationen. Seit der aktuellen Version 1.5 von Java3D kann dieses mittlerweile auch die JOGL Pipeline zur Grafikausgabe benutzen![13]

Warum nun JOGL für das neue Framework gewählt werden soll, ist folgendermaßen zu beantworten. Da Mesa und GL4Java nicht aktueller Stand der Technik sind, kamen diese beiden Bindings erst garnicht zur Diskussion. Wichtig sind Bindings, welche die aktuellen OpenGL-Versionen unterstützen und auch auf die Zukunft gesehen eine hohe Lebenserwartung haben. LWJGL entspricht zwar diesen Kriterien, aber besitzt doch Funktionalitäten wie beispielsweise Soundunterstützung, welche für das Physik-Framework überflüssig sind. Der wichtigste Punkt ist aber, dass LWJGL aus oben genannten Gründen nicht mit Swing oder AWT zusammenarbeitet. Mit LGJWL wäre das Framework in diesem Punkt sehr eingeschränkt und würde vielleicht an einer späteren Weiterentwicklung (Bsp.: Hinzufügen von Benutzersteuerelementen) gehindert werden. Java3D wäre eine gute Alternative gewesen, ist aber nach Einblick in die Tutorials und in die API[14] viel zu umfangreich und zeitaufwendig in der Einarbeitung. Da der Hauptgrund des zu entwickelnden Frameworks in dieser Diplomarbeit darin besteht, dem später damit arbeitenden Studenten Einarbeitungszeit zu ersparen, käme Java3D dem nicht zu gute. JOGL ist dagegen recht einfach gehalten und dessen Funktionsweise kann relativ schnell nachvollzogen werden. Auch das direkte Arbeiten mit der OpenGL-API dürfte wegen dem prozeduralen Aufbau leichter fallen.

[13] Vgl. [Mic07f] und [Mic07e]
[14] Siehe [Mic07d]

4 Planung des Frameworks

4.1 Anforderungen an das Framework

Wie bereits in der Einleitung und in den Zielen der Diplomarbeit erklärt ist, soll ein Framework erstellt werden, dass die Studenten bei Ihrer Arbeit mit Game Physics unterstützen soll. Warum dieses auf JOGL und somit folglich auch auf OpenGL basieren soll, ist in den dazugehörigen Kapiteln 2 und 3 auch bereits klargestellt worden. Auch ist in dem JOGL-Grundlagenkapitel 3.2.3 schon ein vollständiges, einfaches JOGL-Programm aufgelistet, mit dem sich theoretisch schon einige Physik Simulationen realisieren lassen. Doch es gibt viele wichtige Funktionalitäten, die dieses einfache Programm nicht bietet, welche aber oft für Physiksimulationen benötigt werden. Erste Gehversuche, einfache Simulationen zu erstellen, ergaben schnell, dass neben der reinen Möglichkeit, die OpenGL-API nutzen zu können, auch folgende Punkte sehr wichtig sind.

- **Darstellung einer Simulation in verschiedenen Vollbildauflösungen**
  Neben der Darstellung einer Simulation in einem normalen GUI-Fenster (Fenstermodus), gibt es auch die Möglichkeit, diese in einem sogenannten Full-Screen Exclusive Mode (Vollbildmodus) laufen zu lassen. Wie der Name schon sagt, wird hier der gesamte Bildschirm zu Darstellung verwendet. Dies ist zwar mit einem normalen GUI-Fenster auch möglich, aber der entscheidene Unterschied ist, dass hier die Auflösung, Farbtiefe und Frequenz des Bildschirms verändert werden können.[1] So kann die Anzeige der Simulation individuell für jedes Rechnersystem abgestimmt werden. Auch ist das Anzeigen von Grafik im Full-Screen Exclusiv Mode erheblich schneller, da hier das Programm selber aktiv für die Zeichnung der Grafik zuständig ist.

- **Benutzerinteraktionen durch Maus und/oder Keyboard**
  Viele Physik-Simulationen erfordern eine Interaktion durch den Benutzer. Vielleicht soll dieser nur einige Parameter zur Laufzeit verändern, oder sogar Objekte hin und her bewegen. In beiden Fällen ist eine Eingabe des Benutzers über Maus oder Keyboard erforderlich.

[1] Vgl. [Mic07c]

- **Bewegliche Kamera für dynamische Ansichten in einer Simulation**
  Sinnvoll wäre die Entwicklung einer dynamischen Kamera, mit der das Gefühl von freier Bewegung im Raum erzeugt werden kann. Der Betrachter könnte sich frei im Raum bewegen und interessante Punkte eine Physiksimulation aus verschiedenen Sichten anschauen. Oft sehen Simulationen sehr statisch aus und können nur aus einer bestimmten Ansicht betrachtet werden, weil gerade diese nützliche Funktionalität nicht implementiert wurde.

- **Möglichkeit für Zeitmessungen in einer Simulation**
  Zeitmessungen dienen sehr gut als Hilfsmittel zum Optimieren oder Vergleichen von bestimmten Algorithmen. Die Möglichkeit, diese benutzen zu können sollte somit nicht fehlen.

- **Möglichkeit zum Ausgeben von Text in der Simulation**
  Die Ausgabe von Text, beispielsweise zum Debuggen[2], ist wie üblich über die Java-Konsole mit `System.out.print` möglich. Sobald sich eine Simulation aber in einem Vollbildmodus befinden sollte, wird die Java-Konsole überdeckt und kann nur umständlich erreicht werden. Auch eventuelle Anwendungshinweise sind dann nicht mehr lesbar. Es sollte eine Möglichkeit geboten werden, wichtigen Text direkt auf der Zeichenfläche (wo auch die Simulation dargestellt wird) anzuzeigen. Da OpenGL hier keine direkte Funktion zur Verfügung stellt, muss dieses über Umwege realisiert werden.(Siehe Kapitel 5.6)

Es gibt sicher noch viele weitere Punkte, die man berücksichtigen könnte wie beispielsweise das Erstellen von Physik-Bibliotheken, welche viele vordefinierte Funktionen und Algorithmen enthalten. Da das Framework aber in einem überschaubaren Rahmen gehalten werden soll, werden nur die oben genannten 5 essenziellen Punkte Einzug in das Framework finden.

4.2 Aufbau und Struktur des Frameworks

Für den Aufbau und die Struktur des Frameworks gibt es zahlreiche Möglichkeiten, diesen zu realisieren. Wichtig ist, dass es eine übersichtliche Struktur aufweist und auch recht schnell auf Funktionalität getestet werden kann, da die Materie hier noch relativ neu ist und so eventuell auftretende Probleme und Überraschungen rechtzeitig erkannt werden können. Hierfür bieten sich verschiedene Entwicklungs-Strategien aus der Softwaretechnik an, von denen 2 wichtige kurz beschrieben werden:[3]

[2] Debuggen = Fehler im Programmcode suchen/beheben
[3] Vgl. [Hel00], Seite 1055

- **Top-Down Strategie**
  Bei der Top-Down Strategie ist ein genauer Überblick des gesamten Projektes erforderlich. Hier wird detailiert der Aufbau und das Zusammenspiel aller Komponenten im Vorfeld festgelegt und in einem Modell festgehalten. Die einzelnen Komponenten werden dann Stufenweise bis hin zum Programmcode entwickelt. Der große Vorteil dieser Vorgehensweise ist, dass so einzelne Komponenten an das vorher definierte Modell angepasst werden und nicht umgekehrt. Es wird mit Fokus auf das Modell entwickelt! Ein Problem hierbei aber ist, dass das Modell unter Umständen erst recht spät auf Lauffähigkeit getestet werden kann und somit ein gewisses Entwicklungs-Risiko aufweist.

- **Bottom-up Strategie**
  Hier wird genau die entgegengesetzte Strategie angewand. Wichtig ist, dass einzelne Komponenten des Programm frühzeitig lauffähig sind und so getestet werden können. Es werden zuerst elementare Komponenten sehr detailreich umgesetzt, auf diesen dann Komponenten höherer Abstraktions-Ebene aufsetzen. So herrscht zu jeder Zeit ein lauffähiges Programm. Die Gefahr bei dieser Strategie ist aber, dass das Modell sehr schnell komplex werden kann, da das Zusammenwirken aller Komponenten nicht vollständig vorher durchgeplant ist.

Da bereits lauffähige JOGL-Programme geschrieben wurden und der Rahmen der funktionalen Anforderungen gut überschaubar ist, soll das Framework nach einer groben Top-Down Strategie entwickelt werden. Hier überwiegt der Vorteil einer klaren Struktur, und das Riskio, am Ende eine nicht lauffähige Version herauszubekommen, ist aufgrund des Umfangs relativ gering. Grob bedeutet hier, dass nicht jede einzelne Methode im vorraus spezifiert wird, sondern nur die einzelnen Komponenten (Klassen) und deren Zusammenwirken. Es wird sozusagen eine gemischte Strategie aus Top-Down und Bottom-Up angewendet werden. Abbildung 4.1 zeigt den geplanten Aufbau des Frameworks anhand eines UML 2.0 Diagrammes:

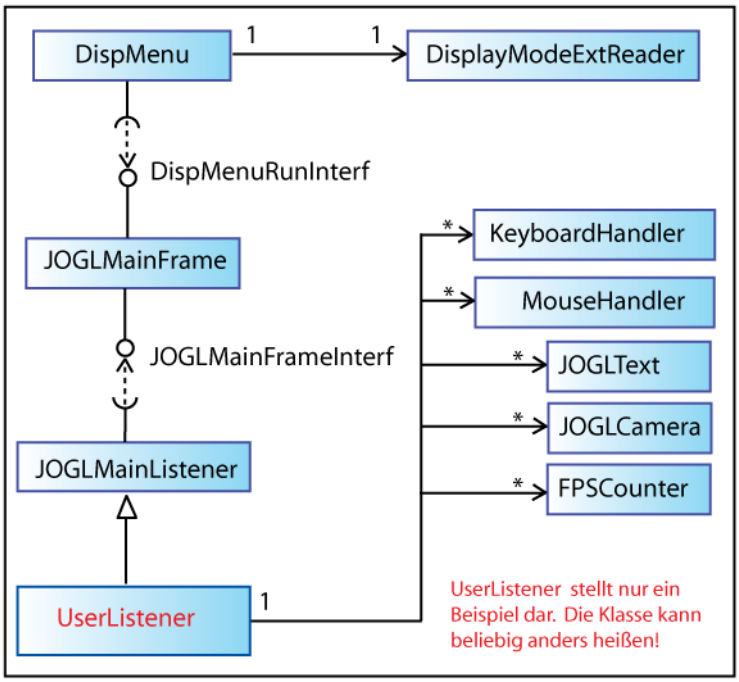


Abbildung 4.1: Klassendiagramm des JOGL Frameworks in UML 2.0

Im Einzelnen sind nun einige der Klassen und deren vorraussichtliche Funktionen beschrieben:

- **DispMenu**
  Diese Klasse wird ein Anzeigemenü darstellen, in dem verschiedene Anzeigekonfigurationen (Auflösung, Farbtiefe, Frequenz, Vollbild-/Fenstermodus) ausgewählt werden können. Wichtige Informationen wie beispielsweise Unterstützungen der aktuellen Grafikkarte sollen in einem Textfeld angezeigt werden können. Über das Anzeigemenü kann dann eine beliebige Anwendung gestartet werden, welche die Schnittstelle `DisMenuRunInterf` implementiert. Die ausgewählte Anzeigekonfiguration im Menü wird dabei über diese Schnittstelle an das aufzurufende Programm weitergegeben. So ist das Anzeigemenü unabhängig von dem weiteren Aufbau des Frameworks. Da nicht jede Grafikkarte und Monitor die gleichen Anzeigekonfigurationen unterstützen, sollten nur die Konfigurationen zur Auswahl stehen dürfen, welche auch kompatibel mit dem aktuellen System sind. Das Herausfinden der kompatiblen Konfigurationen übernimmt die Klasse `DisplayModeExtReader`.

- **JOGLMainFrame**
  Diese Klasse stellt das Hauptfenster dar, indem ein `GLCanvas` und ein `Animator` eingebettet sind. Es wird je nach Benutzereinstellung entweder im Fenstermodus oder im Vollbildmodus lauffähig sein. Auflösung, Farbtiefe und Frequenz können auch vom Benutzer festgelegt werden. Es ist vollkommen unabhängig von dem AnzeigeMenu `Dispmenu`, kann aber über das implementierte Inter-

face `DisMenuRunInterf` jederzeit mit diesem verwendet werden. Dies ist auch empfehlenswert, damit das `JOGLMainFrame` nicht mit falschen Anzeigekonfigurationen, welche vielleicht nicht kompatibel mit dem aktuellen System sind, vom Benutzer versehen werden. Auch Implementiert das `JOGLMainFrame` eine Schnittstelle `JOGLMainFrameInterf`, über dieses auf bestimmte Funktionen des Hauptfensters zugegriffen werden kann.

- **JOGLMainListener**
  Diese Klasse implementiert intern das Interface `GLEventListener` und benutzt die Schnittstelle `JOGLMainFrameInterf`, um mit dem `JOGLMainFrame` kommunizieren zu können. Möchte der Benutzer eine Simulation erstellen, muss er eine eigene Listenerklasse (Rot umrahmt in Abbildung 4.1) programmieren, welche den `JOGLMainListener` erbt. In dieser Klasse wird die gesamte Physik-Simulation implementiert und dann dem `JOGLMainFrame` hinzugefügt. Der Vorteil bei der Vererbung ist hier, dass die nötigen `GLEventListener`-Funktionen bereits schon im `JOGLMainListener` implementiert sind. Die erbende Klasse muss somit nur die Funktionen überschreiben die sie auch wirklich *benötigt*.

- **KeyboardHandler/MouseHandler**
  Diese beiden Klassen sollen alle Funktionalitäten und Unterstützungen bezüglich Benutzerinteraktionen implementieren. Zwar stellt das AWT schon solche Funktionalitäten bereit, aber im späteren Kapitel 5.3 wird erklärt warum diese trotzdem zusätzlich nötig sind.

- **JOGLText**
  Diese Klasse realisiert eine Möglichkeit zur Darstellung von Text auf dem `GLCanvas` mit OpenGL.

- **JOGLCamera**
  Hiermit wird eine vollständig, frei bewegliche 3D-Kamera realisiert werden.

- **FpsCounter**
  `FpsCounter` wird eine Möglichkeit zur Performancemessung bieten. Er misst Schleifendurchgänge pro Sekunde (frames per Second (fps)).

Da in dem geplanten UML-Modell (Abbildung 4.1) die Abhängigkeiten der Klassen jeweils über Interfaces klein gehalten sind, kann das Modell auf die Zukunft gesehen noch ausgebaut und einzelne Komponenten ausgetauscht werden. Folgendes Beispiel zeigt noch einmal die geplante Verwendung des Frameworks in Java-Pseudocode.

```
1 class Physiksimulation extends JOGLMainListener
2 {
3         FPSCounter count  = new FPSCounter();
4         JOGLCamera camera = new JOGLCamera();
```

```
5           ....
6           ....
7  }
8
9  //Pseudocode1: (Wenn Anzeigemenü gewünscht):
10 JoglMainFrame hauptfenster = new JOGLMainFrame();
11 Physiksimulation physikSim = new Physiksimulation(hauptfenster);
12 hauptfenster.addJoglMainListener(physikSim);
13 DispMenu anzeigeMenu = new DispMenu(hauptfenster);
14
15 //Pseudocode2: (Wenn kein Anzeigemenü gewünscht):
16 JoglMainFrame hauptfenster = new JOGLMainFrame();
17 Physiksimulation physikSim = new Physiksimulation(hauptfenster);
18 hauptfenster.addJoglMainListener(physikSim);
19 hauptfenster.runme(Gewuenschter_Anzeigemodus);
```

4.3 Software-Werkzeuge zur Erstellung des Frameworks

Für das Erstellen des Frameworks werden folgende Werkzeuge gewählt:

- **Entwicklungsumgebung JCreator LE 4**
  Als Entwicklungsumgebung zum Programmieren des nötigen Java-Codes wird der JCreator gewählt. Er ist mit seinen knapp 4 Megabyte sehr kompakt und sehr schnell in der Ausführung. Er stellt zwar nicht so ein mächtiges Entwicklungstool wie beispielsweise Eclipse[4] oder Netbeans[5] dar, hat aber hier seine Daseinsberechtigung dadurch, dass er auch auf älteren Rechnersystemen noch mit moderater Geschwindigkeit läuft. Da das Framework aus momentaner Sicht auch auf einigen recht langsamen Rechnern seinen Einsatz finden muss, kommt dort *vorerst* außer dem JCreator kein anderes Entwicklungstool in Frage. Somit ist es vorteilhaft, bereits jetzt schon für die Entwicklung den JCreator zu benutzen und das Framework-Projekt so auf diesen abzustimmen.

- **JDK 6 (Mustang)**
  Vor kurzem hat Sun Microsystems das JDK (Java Development Kit) 6 fertig entwickelt und offiziel freigegeben. Es enthält die nötige Laufzeitumgebung und den Compiler für Java. Das JDK 6 besitzt im Gegensatz zu älteren Versionen viele neue Features und Verbesserungen wie zum Beispiel eine bessere Performance des JNI (Vorteilhaft für JOGL).[6]

[4] Siehe: http://www.eclipse.org/
[5] Siehe: http://www.netbeans.org/
[6] Mehr Details unter [Mic07a]

- **JOGL**
  JOGL wird in seiner aktuellen Version 1.1.0-rc3 verwendet werden.

5 Umsetzung/Programmierung des Frameworks

In diesem Kapitel wird auf die Umsetzung des JOGL-Frameworks in Java eingegangen. Die einzelnen Komponenten/Klassen und dessen Zusammenwirken sollten aus Kapitel 4 bekannt sein. Da das Framework in seinem Gesamtumfang über 1500 Zeilen an Code enthält, werden nur einige wichtige Codestellen exemplarisch aufgezeigt. Zuerst werden die einzelnen Komponenten ausführlich beschrieben und auch aufgetretende Probleme bei deren Implementierung diskutiert. Wie der Benutzer das Framework schließlich verwendet, ist im Kapitel 5.9 zu finden.

5.1 Entwickeln eines Anzeigemenüs

5.1.1 Auswahlmöglichkeit verschiedener Anzeigekonfigurationen

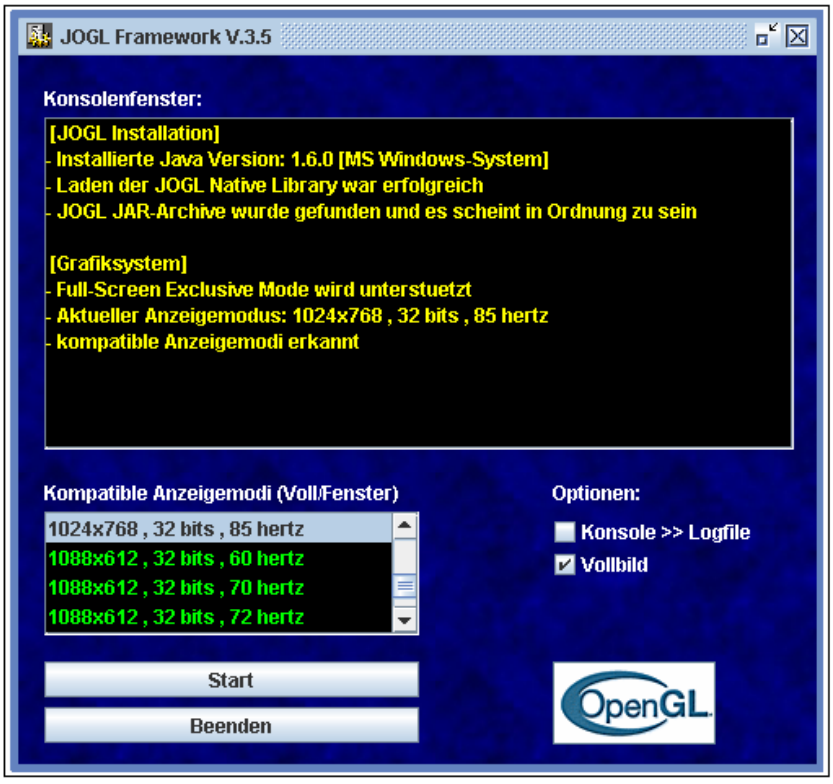


Abbildung 5.1: Das Anzeigemenü des Frameworks

Das Anzeigemenü bietet verschiedene Anzeigekonfigurationen zum Auswählen an, welche mit dem aktuellen PC-System kompatibel sind (Abbildung 5.1). Die ausgewählte Konfiguration kann darauf über die Schnittstelle `DispMenuRunInterf` an eine beliebige andere Klasse, vorraussichtlich natürlich dem `JOGLMainFrame`, weitergegeben werden, welches diese dann verwertet. Das Aufrufen der Schnittstelle geschieht intern in dem dafür vorgesehen Button „Start“ des Anzeigemenüs. Wie in Abbildung 5.1 zu erkennen ist, stellt das Menü einige vorgegebene Anzeigekonfigurationen bereit. Dabei werden zueinander kompatible Auflösungen, Farbtiefen und Frequenzen von der Klasse `DisplayModeExtReader` herausgesucht. Er benutzt intern einen Filter, wobei er beispielsweise Farbtiefen < 16 Bit ignoriert. Sie werden deswegen aussortiert, weil JOGL nur ab einer Farbtiefe von 16 Bit oder größer funktioniert. Auch sehr hohe Auflösungen, welche heutzutage noch von fast keinem System unterstützt werden, sind der Übersicht halber ausgefiltert. Wichtig ist zu erkennen, dass sich in Abbildung 5.1 auf den Vollbildmodus bezogen wird. Dieser ist über die CheckBox auf der rechten Seite aktiviert. Wünscht der Benutzer keinen Vollbildmodus, sondern einen Fenstermodus, ist ein Wechseln in andere Auflösungen nicht möglich und somit deren Auswahlmöglichkeit dann überflüssig. In diesem Fall bietet der `DisplayModeExtReader` automatisch verschiedene, aufsteigende Standard-Fenstergrößen an, welche maximal so groß werden wie die aktuelle Auflösung des Systems. Durch diese dynamische Verfahrensweise des Menüs ist es nicht möglich, eine fehlerhafte Anzeigekonfiguration auszuwählen. Doch es gibt noch einige Sonderfälle die beachtet werden müssen, welche sich aber erst spät nach dem Testen des Menüs auf unterschiedlichen Rechnern und auch Plattformen (Linux SUSE 9.1) herauskristallisierten.

- **Fall1: Es wird kein Vollbildmodus unterstützt**
  In diesem Fall ist das bereitstellen von kompatiblen Anzeigekonfigurationen für den Vollbildmodus unnötig. Da es aber die Möglichkeit gibt, den Vollbildmodus später über einen Fenstermodus zu simulieren, wird zumindest eine Anzeigekonfiguration für den Vollbildmodus angeboten, welche der aktuellen Konfiguration des Systems gleicht.[1]

- **Fall2: Es können keine Anzeigekonfigurationen des Systems ermittelt werden**
  Da in diesem Fall keinerlei Information vorliegt, wird zumindestens probiert, über die Desktop-Größe an die aktuelle Auflösung des Systems heranzukommen. Wenn diese ermittelt werden kann, können hiervon schonmal die einzelnen Anzeigekonfigurationen für den Fenstermodus abgeleitet werden. Für den Vollbildmodus wird wie in Fall1 verfahren.

[1] Für den simulierten Vollbildmodus: Siehe Kapitel 5.2.1, Seite 55

- **Fall3: Das System kann kann einzelne Hertz-Werte nicht ermitteln**
  Java stellt hierfür einen Platzhalter `DisplayMode.REFRESH_RATE_UNKNOWN` zur Verfügung, welcher unbedingt mit berücksichtig werden muss.

- **Fall4: Das System unterstützt mehrere Farbtiefen**
  Die Unterstützung von mehreren Farbtiefen gleichzeitig war bei einem Linux-Testsystem (SUSE 9.1) zu finden. So wurde eine zusätzliche Abfrage nach mehreren Farbtiefen mit `DisplayMode.BIT_DEPTH_MULTI` implementiert.

Da hier das Zusammenspiel des Anzeigemenüs und des `DisplayModeExtReaders` sehr komplex ist und viele Fälle behandelt, wird auf Programmcodeauszüge an dieser Stelle verzichtet. Nützlich ist es aber, wenigstens die wichtigen Java-Befehle kurz zu erläutern, mit denen der `DisplayModeExtReader` an Informationen des System gelangt. Zuerst wird in Java ein `GraphicsDevice` benötigt. Es stellt eine Schnittstelle dar, über die auf verschiedene Elemente, wie zum Beispiel einen Drucker oder den Bildschirm zugegriffen werden kann. Dabei hat jedes einzelne Element einen eigenen `GraphicDevice`. Dies gilt auch, wenn beispielsweise mehrere Drucker oder Bildschirme vorhanden sind. Alle `GraphicDevices` zusammen bilden dann eine sogenannte `GraphicsEnvironment`. Um nun an aktuelle Informationen des Bildschirms zu gelangen (Hier: nur ein Bildschirm vorrausgesetzt!) muss zuerst die `GraphicsEnvironment` ermittelt werden und aus dieser das `GraphicDevice` für den Bildschirm ausgewählt werden:

```
GraphicsDevice dev = GraphicsEnvironment.
                        getLocalGraphicsEnvironment().
                               getDefaultScreenDevice();
```

Mit solch einem `GraphicsDevice` können nun alle kompatiblen Anzeigekonfigurationen des Systems ermittelt werden:

```
//Aktuelle Anzeigekonfiguration
DisplayMode curMode = dev.getDisplayMode();

//Alle kompatiblen Anzeigekonfiguration
DisplayMode[] allModes = dev.getDisplayModes();
```

Die Anzeigekonfigurationen werden in Form eines `DisplayMode`-Objektes zurückgegeben. Auf die einzelnen Werte kann folgendermaßen zugegriffen werden:

```
//Bildschirm-Frequenz
int hertz     = curMode.getRefreshRate()

//Farbtiefe
int bitdepth  = curMode.getBitDepth()
```

```
//Auflösung horizontal in Pixeln
int width     = curMode.getWidth()

//Auflösung vertikal in Pixeln
int height    = curMode.getHeight()
```

Hierbei ist zu beachten, dass für den Hertz- und Farbtiefen-Wert zwei spezielle Werte zurückgegeben werden können. Einmal `DisplayMode.REFRESH_RATE_UNKNOWN`, wenn der Hertz-Wert unbekannt ist und `DisplayMode.BIT_DEPTH_MULTI`, wenn das System mit mehreren Farbtiefen gleichzeitig arbeiten kann. Um schließlich herauszufinden, ob das System den Vollbildmodus unterstützt, reicht eine einfache Abfrage mit `dev.isFullScreenSupported()`.

5.1.2 Ausgabe von wichtigen Informationen in einem Textfeld

Neben dem Bereitstellen einer Auswahl von verschiedenen Anzeigekonfigurationen gibt das Anzeigemenü aber auch einige wichtige Information in einem Textfeld aus. Dabei sind die Informationen unterteilt in 2 Sektionen. Erst werden nützliche Informationen ausgegeben, die für das Arbeiten mit JOGL hilfreich sind. Darunter sind Informationen bezüglich dem Bildschirm/Grafikkarte zu finden. Die Informationen in dem Textfeld können auch bei Bedarf über die Checkbox `Konsole>>Logfile` in eine Datei abgesichert werden. Folgende Informationen werden in dieser Reihenfolge angezeigt:

1. **Ausgabe der aktuellen Java-Version.**
   So kann schnell überprüft werden, ob die Java-Version aktuell ist.

2. **Hinweis, ob die JOGL-Native Bibliothek gefunden worden ist.**
   Diese muss unbedingt im System installiert sein, damit JOGL richtig funktioniert. Sie stellt die Verbindung mittels JNI zu OpenGL her. Diese Textausgabe ist sehr hilfreich, um festzustellen ob JOGL korrekt installiert wurde.

3. **Hinweis, ob das JOGL-Jar Archive vorhanden ist.**
   Hier wird getestet, ob das JOGL-Jar Archive gefunden werden kann und ob dieses auch funktioniert. Dies ist ebenfalls hilfreich, um zu sehen, ob JOGL richtig istalliert wurde.

4. **Hinweis, Ob der Vollbildmodus (Full-Screen Exclusive Mode) unterstütz wird.**
   Nützlich um zu sehen, ob das aktuelle Grafiksystem (Bildschirm/Grafikkarte) neuere Funktionen wie den Full-Screen Exclusive Mode unterstützt.

5. **Ausgabe der aktuellen Auflösung, Farbtiefe und Frequenz.**
   Diese Ausgabe ist sehr hilfreich, da sie einen Richtwert liefert, was das System auf jeden Fall zu Leisten vermag.

6. **Hinweis, ob kompatible Anzeigemodi gefunden werden konnten.**
   Hier kann erkannt werden, ob das Anzeigemenü generell kompatibel mit dem aktuellen System ist. Ein Anzeigemodus umfasst in diesem Kontext die Auflösung, Farbtiefe und die Frequenz des Bildschirms.

5.2 Entwickeln des Hauptfensters

Das Hauptfenster `JOGLMainFrame` ist das wichtigste Element des Frameworks. Während das Anzeigemenü optional genutzt werden kann, ist das Hauptfenster unverzichtbar, da hier die Physiksiumulation mittels des eingebetteten `GLCanvas` dargestellt wird. Zum größten Teil realisiert es folgende wichtige Punkte, welche in den nächsten 3 Unterkapiteln genauer beschrieben werden.

1. **Unterstützung eines Vollbildmodus**
   (Kapitel 5.2.1)

2. **Anbindung an einen JOGLMainListener**
   (Kapitel 5.2.2)

3. **Implementierung der Schnittstelle JOGLMainFrameInterf**
   (Kapitel 5.2.3)

5.2.1 Unterstützung eines Vollbildmodus

Der Aufbau der Klasse `JOGLMainFrame` gleicht in einigen Punkten der Klasse `JoglApp`[2], da diese auch mittels AWT realisiert wird und einen `GLCanvas` und einen `Animator` einbettet. Die größte Erneuerung aber ist, dass das Hauptfenster ein Wechseln in den Vollbildmodus unterstützt. Während das Anzeigemenü `DispMenu` nur versucht, optimale Anzeigekonfigurationen zu finden, ist das Hauptfenster dafür zuständig, solche auch zu verwerten. Die nötige Anzeigekonfiguration erhält das Hauptfenster dabei über das Interface `DispMenuRunInterf`, welches wie folgt aufgebaut ist:

```
runme(DisplayMode dispInitMode, boolean fullScreenMode)
{
}
```

[2]Siehe Kapitel 3.2.3

Dabei probiert das `JoglMainFrame`, unbedingt der gewünschten Anzeigekonfiguration nachzukommen. Ist eine Anzeigekonfiguration fehlerhaft angegeben, wird versucht diese zu korrigieren oder es benutzt automatisch eine alternative, kompatible Anzeigemöglichkeit. Der genaue Ablauf ist in Abbildung 5.2 als UML-Zustandsdiagramm dargestellt. Da der Codeumfang des `JOGLMainFrames` recht groß ist, wird dieser hier

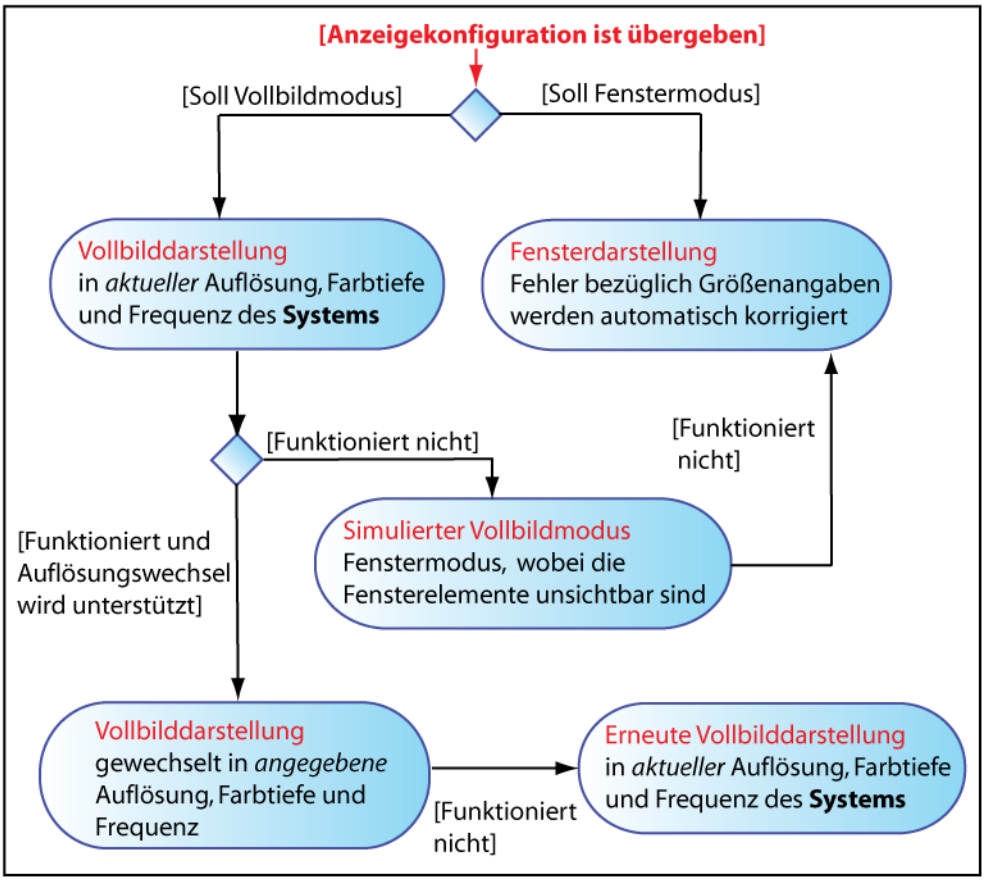


Abbildung 5.2: UML 2.0 Zustandsdiagramm: Vollbild- oder Fensterdarstellung

auch nicht aufgeführt werden. Einzelne Codeauszüge wären auch nicht sinnvoll, da so sehr viele benötigte Zusammenhänge verschleiert würden. An dieser Stelle soll deshalb nur die Verfahrensweise und die nötigen Java-Funktionen beschrieben werden, wie in einen Vollbildmodus gewechselt werden kann und dann in eine andere Auflösung umgeschaltet wird. Zuerst muss wie bereits bekannt, das nötige `GaphicsDevice` für den Bildschirm ermittelt werden. Darauf wird ein `DisplayMode`-Objekt erstellt und über den Konstruktor mit der gewünschten Auflösung (640x480), Farbtiefe (32) und Frequenz (85) versehen.

```
GraphicsDevice dev = GraphicsEnvironment.
                          getLocalGraphicsEnvironment()
                               .getDefaultScreenDevice();
```

```
DisplayMode  dispMode = new DisplayMode (640, 480, 32, 85);
```

Es wird davon ausgegangen, dass sich hier auf ein Hauptfenster bezogen wird, wie es auch im `JOGLMainFrame` der Fall ist. Damit im Vollbildmodus nicht noch die Titelleiste und der Rahmen des Hauptfensters zu sehen sind, müssen diese unsichtbar gemacht werden. Sinngemäß sollte das eigentlich der Vollbildmodus selber realisieren, aber in Java muss dies explizit angegeben werden.

```
this.setUndecorated(true);
```

Darauf kann in den Vollbildmodus gewechselt werden:

```
try
{
   dev.setFullScreenWindow(this);
}
catch(Exception e) {}
```

Wird kein Vollbildmodus unterstützt, entsteht trotzdem eine Vollbilddarstellung. Die Funktion `setFullScreenWindow(this)` maximiert dann im Fenstermodus automatisch das Hauptfenster auf den gesamten Bildschirm. Die Fensterelemente sind hierbei nun unsichtbar, da vorher `this.setUndecorated(true)` benutzt wurde. Dieser Vollbildmodus ist somit kein echter Vollbildmodus, sondern ein simulierter Vollbildmodus. Es ist hier nicht möglich Auflösung, Farbtiefe oder Frequenz zu ändern! Wird aber der Vollbildmodus unterstützt und ist das System generell in der Lage die Auflösung zu wechseln (`isDisplayChangeSupported(...)`), kann dies schließlich mit `setDisplayMode(dispMode)` realisiert werden.

```
if (dev.isFullScreenSupported() && dev.isDisplayChangeSupported())
{
   try
   {
      dev.setDisplayMode(dispMode);
   }
   catch(Exception e) {}
}
```

Abschließend zu diesem Kapitel sollte noch erwähnt werden, dass die Kombination des Anzeigemenüs `DispMenu` und des `JOGLMainFrames` eine sehr robuste Lösung darstellen. Das Anzeigemenü zwingt zu der Auswahl einer kompatiblen Anzeigekonfiguration und das Hauptfenster ist auf alle eventuellen Fehler beim Anzeigen dieser vorbereitet. Nach Fertigstellung der beiden Klassen und Einbindung der Sonderfälle des Anzeigemenüs konnte auf *keinem* Testsystem ein Fehler/Absturz des Frameworks beobachtet werden. Auch Versuche, das `JOGLMainFrame` manuell mittels fehlerhaften Anzeigekonfiguration zum Absturz zu zwingen, schlugen fehl.

5.2.2 Anbindung an einen JOGLMainListener

Das `JOGLMainFrame` hat einen `GLCanvas` sowie auch ein `Animator`-Objekt eingebettet. Dem `GLCanvas` des `JOGLMainFrames` kann aber nun nicht wie in dem einfachen JOGL-Programm in Kapitel 3.2.3 mit `add(...)` ein `GLEventListener` hinzugefügt werden, wie es normalerweise für ein JOGL-Programm üblich ist. Hierfür muss jetzt die neue Methode `addJOGLMainListener(JOGLMainListener joglMainListener)` des `JOGLMainFrames` verwendet werden. Gründe hierfür sind schon in Kapitel 4.2, dem Aufbau des Frameworks, genannt worden. Wiederholend wird nocheinmal darauf aufmerksam gemacht, das nicht eine Instanz des `JOGLMainListener` benutzt werden soll, sondern eine eigene Klasse, welchen diesen erbt! Der `JOGLMainListener` hat selber kaum Funktionalität implementiert:

```
public class JOGLMainListener implements GLEventListener,
                                         KeyListener,
                                         MouseMotionListener,
                                         MouseListener
{
    public JOGLMainFrameInterf mainFrame  = null;

    public JOGLMainListener(JOGLMainFrameInterf mainFrame)
    {
        this.mainFrame  = mainFrame;
    }

    public void init(GLAutoDrawable glDrawable){}

    public void display(GLAutoDrawable glDrawable){}

    public void reshape(GLAutoDrawable glDrawable, int x, int y,
                                             int width, int height){}

    public void displayChanged(GLAutoDrawable glDrawable,
                                         boolean modeChanged,
                                           boolean deviceChanged) {}

    public void mouseEntered(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e){}
    public void mouseDragged(MouseEvent e) {}
    public void mouseMoved(MouseEvent e)   {}
    public void mouseClicked(MouseEvent e) {}
```

```
    public void mouseExited(MouseEvent e)  {}

    public void keyTyped(KeyEvent e){}     {}
    public void keyReleased(KeyEvent e)    {}
    public void keyPressed(KeyEvent e)     {}
}
```

Neben den `GLEventListener`-Funktionen hat dieser auch Keyboard und Mausfunktionen des AWT's vorimplementiert. Sollen später einmal Maus- oder Keyboardfunktionen verwendet werden, müssen die benötigten Funktionen ebenfalls überschrieben werden. Durch die Verwendung dieses Prinzipes kann so der Code der späteren Physiksimulationen etwas übersichtlicher und kleiner gehalten werden. Wie zu erkennen ist, wird im Konstruktor des Listeners die Schnittstelle `JOGLMainFrameInterf` verwendet, über das Zugriffe auf einige Funktionen des `JOGLMainFrames` möglich sind.

5.2.3 Implementierung der Schnittstelle JOGLMainFrameInterf

Damit andere Klassen bestimmte Funktionen des `JOGLMainFrames` nutzen könnnen, wird die Schnittstelle `JOGLMainFrameInterf` implementiert. Die Schnittstelle sieht folgendermaßen aus:

```
public interface JOGLMainFrameInterf
{
    public void    shutdown();
    public Point   getCanvasLocationOnScreen();
    public void    hideMouseCursor();
    public void    showMouseCursor();
}
```

Die erste Methode ist wichtig, um das Hauptfenster von einer anderen Klasse aus herunterfahren zu können. Ein einfacher Abbruch mit `System.exit(1)` ist nicht empfehlenswert, da das `JOGLMainFrame` einen Animator besitzt[3]. Die anderen 3 Methoden sind später wichtig für den Mousehandler (Siehe Kapitel 5.4):

- `getCanvasLocationOnScreen()` ermittelt die aktuelle Position des eingebetteten `GLCanvas`.
- Mit `hideCursor()` kann der Mauscursor bei Bedarf auf unsichtbar geschaltet werden.
- `showMouseCursor()` macht den Mauscursor sichtbar.

[3]Siehe JOGL-Grundlagen: Kapitel 3.2.3

5.3 Entwickeln des KeyboardHandlers

Mit einem KeyboardHandler kann in der Regel auf die Tastatur des PC-Systems zugegriffen werden. Oft besitzt dieser wichtige Funktionen, die erkennen, ob eine bestimmte Taste gedrückt oder losgelassen wurde. Da aber das AWT schon einen fertigen KeyboardHandler anbietet, ist die Frage berechtigt, warum ein zusätzlicher KeyboardHandler überhaupt erstellt werden soll. Um dies zu verstehen, wird zuerst erklärt, wie das Ansprechen der Tastatur mit dem AWT realisiert wird. Dazu ein minimales, vollständig lauffähiges Beispiel:

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class KeyBoardSample extends Frame
5                                      implements KeyListener
6  {
7      public KeyBoardSample()
8      {
9          this.addKeyListener(this);
10         this.setSize(100,100);
11         this.setVisible(true);
12     }
13
14     public static void main(String[] args)
15     {
16         KeyBoardSample sample = new KeyBoardSample();
17     }
18
19     public void keyTyped(KeyEvent e)     {}
20     public void keyReleased(KeyEvent e) {}
21     public void keyPressed(KeyEvent e)
22     {
23         if (e.getKeyCode() == KeyEvent)
24                 System.exit(1);
25     }
26 }
```

Die Klasse `KeyBoardSample` implementiert das nötige Listener-Interface `KeyListener`, um die Tastatur ansprechen zu können. In Zeile 9 wird dieser dann hinzugefügt. Die 3 Keyboard-Methoden unterscheiden sich wie folgt:

- **KeyTyped(...)**
  Diese Methode testet, ob eine bestimmte Taste gedrückt wurde. Sie kann auch Kombinationen von Tasten erkennen, wie Beispielsweise ALT Gr + Q, welche sie dann als @-Zeichen interpretiert. Die Zeichen können mit `getKeyChar()` ermittelt werden, die dann in einer Variablen vom Typ `char` zurückgegeben werden. Wichtig ist zu erwähnen, dass diese Methode „Spezialtasten“ wie beispielsweise die Cursortasten nicht erkennt! Hierfür muss auf die anderen beiden Methoden `keyPressed(...)`/`KeyReleased(...)` zurückgegriffen werden.

- **keyPressed(...)**
  Mit dieser Methode kann getestet werden, wann eine Taste gerade heruntergedrückt wird. Im Gegensatz zu `KeyTyped(...)` unterstützt diese Methode zusätzlich alle Spezialtasten auf dem Keyboard. Darum muss hier die dafür zuständige Methode `getKeyCode(...)` verwendet werden. Der Unterschied zu `getKeyChar()` liegt dabei im Rückgabewert, denn `getKeyCode(...)` gibt einen sogenannten „VirtualKey“ vom Typ `int` zurück. In dem Codebeispiel mit der Klasse `KeyBoardSample` wird abgefragt, ob der VirtualKey `KeyEvent.VK_DOWN` heruntergedrückt wird. Dieser entspricht der Cursor-Taste mit dem Pfeil nach unten. Wird er betätig, beendet sich darauf das Programm.

- **KeyReleased(...)**
  Diese Methode besitzt die gleichen Eigenschaften wie `KeyPressed(...)`, nur dass sie erst beim Loslassen einer Taste aufgerufen wird.

In dem kurzen Beispiel `KeyBoardSample` funktionert das Arbeiten mit dem KeyListener einwandfrei, doch in Verbindung mit JOGL gibt es hier mehrere Probleme/Einschränkungen:

1. **In den KeyListener-Methoden, funktionieren OpenGL-Befehle nicht!**
   Wie in der Benutzeranleitung[4] zu JOGL ausdrücklich erklärt wird, dürfen keine OpenGL-Befehle innerhalb der `Keylistener`-Methoden verwendet werden. Dies liegt daran, dass die `Keylistener`-Methoden in einem anderen Thread (AWT-Event Thread) ausgeführt werden als die OpenGL-Befehle. Mit JOGL besteht zwar die Möglichkeit, OpenGL in einer Multithreading[5]-Umgebung wie dem AWT zu verwenden, OpenGL selbst wurde aber ursprünglich für Singlethreaded[6]-Umgebungen konzipiert. Eigene Tests bestätigen, dass wenn OpenGL-Befehle innerhalb einer der `KeyListener`-Methoden benutzt werden, diese nichts bewirken. Soll zum Beispiel beim Drücken einer Taste ein Dreieck auf dem `GLCanvas` gezeichnet werden, muss der Tastenstatus **global** gehalten sein. Die `display`-Methode des `GlEventlisteners` kann dann den Tastenstatus abfragen und das Dreieck zeichnen. Bei dem Abfragen nach einer einzelnen

[4] Siehe [Dev07c]
[5] Multithreading = Zeitgleiches Abarbeiten mehrerer Threads
[6] Singlethreaded = Abarbeitung nur eines einzelnen Threads aufeinmal möglich

Taste ist die Lösung sicher praktikabel. Doch bei der Behandlung von mehreren Tasten, welche gleichzeitig gedrückt werden, ist dies neben der Entstehung von vielen globalen Variablen ein wenig Aufwand.

2. **Der Status einer Taste kann mehrfach ausgewertet werden**
   Wird der Status einer Taste global festgehalten und dann von der `Display`-Methode ausgewertet, kann es ein Problem geben. Da die `Display`-Methode oft sehr schnell hintereinander aufgerufen wird, manchmal sogar bis zu mehreren hundert mal in der Sekunde, kann es passieren, dass ein globaler Tastenstatus zu oft verwertet wird.

   Fall-Beispiel:
   Mit dem Druck auf die Taste F1 soll ein Viereck gezeichnet werden. Nochmaliges betätigen lässt es wieder verschwinden und ein erneuter Tastendruck wieder erscheinen usw. Diese Doppelfunktionalität auf einer Taste findet man in vielen Programmen. Wird nun zum Beispiel in einer globalen Variable gerade der Status „F1 runtergedrückt“ abgelegt, kann es passieren, dass die `Display`-Methode diesen mehrfach liest und als mehrfaches Drücken der F1-Taste interpretiert, bevor diese losgelassen wird.

Damit der Benutzer des Frameworks hier nicht auf Probleme stößt und schnell und einfach die Tasten des Keyboards abfragen kann, wurde die Klasse `KeyboardHandler` entwickelt. Sie ist streng genommen kein wirklicher KeyboardHandler, sondern setzt auf dem KeyboardHandler des AWT's auf. Da sie etwas komplizierter Aufgebaut ist und mit 2 internen Hashtabellen[7] arbeitet, würde eine vollständige Erklärung zu weit führen. Somit wird hier nur dessen Verwendung erklärt. Der `KeyboardHandler` selbst wird einfach mit `new` ohne Angaben im Konstruktor erstellt

```
KeyboardHandler k = new KeyboardHandler();
```

Als nächstes muss er mit den AWT-Methoden `keyReleased(...)`/`keyPressed(...)` verknüpft werden. `KeyTyped(...)` wird absichtlich nicht unterstützt, da es nicht alle Tasten des Keyboards erkennen kann.

```
public void keyReleased(KeyEvent e)
{
   k.keyReleasedUpdate(e);
}

public void keyPressed(KeyEvent e)
{
   k.keyPressedUpdate(e);
}
```

[7] Siehe [Chr05], Seite 584

Nun kann der `KeyboardHandler` an jeder beliebigen Stelle, vorrausichtlich natürlich in der `display()`-Methode des `GLEventlisters`, verwendet werden. Hier ein Beispielcode, der das Problem von Doppelfunktionalitäten auf einer Taste behebt.

```
boolean schalter = false;

public void display(GLAutoDrawable glDrawable)
{
   if (k.checkIsKeyPressed(KeyEvent.VK_F1))
   {
      schalter = !schalter;
      System.out.println("Schalter ist auf " + schalter);
      k.setManualRelease(KeyEvent.VK_F1);
   }
}
```

Mit `checkIsKeyPressed(KeyEvent.VK_F1)` wird überprüft, ob gerade die F1-Taste gedrückt wird. `setManualRelease(KeyEvent.VK_F1)` lässt die Taste direkt los, auch wenn sie physikalisch vom Benutzer vielleicht noch nicht losgelassen wurde! Im nächsten Aufruf der `display(...)`-Methode ist somit sichergestellt, dass F1 nicht nocheinmal als „gedrückt“ interpretiert wird. Für Tasten, welche permanent gedrückt gehalten werden sollen kann `setManualRelease(KeyEvent.VK_F1)` einfach weggelassen werden. Auch gibt es hier keine Probleme mit unübersichtlichen globalen Variablen, da diese intern im KeyboardHandler verstaut werden.

5.4 Entwickeln des MouseHandlers

Ein MausHandler ist meist dafür zuständig, alle Aktionen und Ereignisse der Maus zu überwachen. Neben der Benachrichtigung, wann eine bestimmte Maustaste gedrückt wird, ist auch das Erkennen der aktuellen Mausposition auf dem Bildschirm wichtig. Hierfür stellt das AWT bereits 2 Schnittstellen zur Verfügung. Eine wertet die Maustasten aus, und die Andere erkennt Mausbewegungen und Mauspositionen auf dem Bildschirm. Es handelt sich um insgesamt 7 Methoden, die zur Übersicht kurz aufgelistet werden:

1. **Interface MouseListener (Zuständig für die Maustasten):**

   - **public void mouseEntered(MouseEvent e)**
     Ein MouseListener bezieht sich immer auf eine bestimmte Komponente des AWT's. Wird der Mouselistener beispielsweise einem `Frame` hinzugefügt, dann bezieht sich der Listener ausschließlich auf dieses `Frame`. Mausaktionen außerhalb des `Frames` werden nicht erkannt und einfach

ignoriert. Die Funktion `MouseEntered` wird aufgerufen, wenn die Maus in den Gültigkeitsbereich des Listeners eintritt. Dies ist der Fall, wenn der Mauszeiger sich auf der dazugehörigen Komponente befindet.

- **public void mouseExited(MouseEvent e)**
  Diese Methode wird aufgerufen, wenn der Mauszeiger die Komponente verlässt, auf die sich der MouseListener bezieht.
- **public void mouseClicked(MouseEvent e)**
  Wird eine beliebige Maustaste geklickt (Heruntergedrückt und Losgelassen), dann wird diese Methode aufgerufen. Mit `MouseEvent.getButton()` kann hier und bei allen anderen Methoden abgefragt werden, um welche Taste es sich dabei handelt.
- **public void mousePressed(MouseEvent e)**
  Wird eine Maustaste heruntergedrückt, wird diese Methode permanent aufgerufen.
- **public void mouseReleased(MouseEvent e)**
  Wird eine Maustaste losgelassen, wird diese Methode einmal aufgerufen.

2. **Interface MouseMotionListener (Zuständig für Mausbewegungen):**

- **public void mouseMoved(MouseEvent e)**
  Diese Methode wird aktiv, wenn die Maus bewegt wird. Sie funktioniert nur, wenn keine Maustaste hierbei gedrückt ist. Die aktuelle Mausposition kann mit `MouseEvent.getPoint()` herausgefunden werden.
- **public void mouseDragged(MouseEvent e)**
  Diese Methode wird aktiv, wenn die Maus bewegt wird und gleichzeitig eine Maustaste heruntergedrückt ist. `MouseEvent.getPoint()` kann hier ebenfalls benutzt werden.

Mit diesen 7 Methoden können schon fast alle Ereignisse der Maus abgefragt werden. In einfachen Programmen reichen diese mehr als aus, aber in Verbindung mit JOGL gibt es wieder die gleichen Einschränkungen/Umstände bezüglich globaler Statusabfragen und Doppelfunktionalität auf den Maustasten wie auch beim AWT-KeyListener.[8] Der neue MouseHandler behebt diese Probleme ebenfalls. Auch wird hier nun nur dessen Anwendung beschrieben, welche sehr der des `KeyboardHandlers` ähnelt und sollte somit keiner weiteren Erklärung bedürfen.

[8] Diese Problematiken sind bereits in Kapitel 5.3 beschreiben

```
//** Initialisierung **
//Auf den boolean-Parameter im Konstruktor
//des Mousehandlers wird im späteren Verlauf
  dieses Kapitels eingangen werden.

MouseHandler m = new MouseHandler(false);
boolean schalter;

//MouseListener-Methode
public void mousePressed(MouseEvent e)
{
   m.mousePressedUpdate(e);
}

//MouseListener-Methode
public void mouseReleased(MouseEvent e)
{
   m.mouseReleasedUpdate(e);
}

public void mouseMoved(MouseEvent e)
{
   m.mouseMovedUpdate(e);
}

//Display()-Methode des GLEventListeners
public void display(GLAutoDrawable glDrawable)
{
   if (m.checkIsLeftButtonDown())
   {
      schalter = !schalter;
      System.out.println("Schalter ist auf " + schalter);
      m.setManualReleaseLeftButton();
   }
}
```

Die aktuelle Mausposition kann zusätzlich mit `m.getWindowMouseXY()` ermittelt werden, welche diese in Form eines `Point`-Objekt zurückgibt.

```
int mouseX = m.getWindowMouseXY().x;
int mouseY = m.getWindowMouseXY().y;
```

Doch es gibt noch eine Maus-Funktionalität die das AWT nicht unterstützt, welche aber oft bei 3D-Simulationen gewünscht wird. Wird der Mauscursor ganz an den linken Rand des Bildschirms/Fensters bewegt, stoppt dieser und kann nicht mehr weiter bewegt werden. Seine X-Position besitzt in diesem Fall dann meistens den Wert 0. Was ist aber, wenn der Benutzer sich beispielsweise mit der Maus frei in einem 3D-Raum bewegen möchte ? Er könnte sich nur in einem begrenzten Umfeld bewegen, da das Mauskoordinatensystem auf einen kleinen Bereich eingeschränkt ist. Wünschenswert wäre ein Koordinatensystem für die Maus, welches unabhängig von einem Bildschirm/Fenster ist. Da später zudem eine frei bewegliche Kamera für das Framework entwickelt wird, ist die Implementierung eines unabhängigen Koordinatensystems sehr sinnvoll. Einige Überlegungen und Recherchen in der AWT API-Dokumentation führten zu dem Ergebnis, dass es hier 2 Möglichkeiten zu dessen Realisierung gibt.

1. **Ermitteln von relativen Mauskoordinaten durch festsetzen des Cursors**
   Die Idee besteht hierin, den Mauscursor automatisch permanent in die Mitte des Bildschirms/Fensters zu verschieben. Bewegt der Benutzter den Mauscursor von der Mitte weg, wird dieser sofort wieder in die Mitte geschoben. Mit dieser Realisierung wird der Mauscursor niemals am Rand des Bildschirms/-Fensters stoppen können, da dieser erst garnicht so weit kommt. Es ist dann möglich, die Differenz zwischen der Position der Mitte und der Position, wo der Benutzer den Mauscursor hinbewegt hat, auszurechnen. Diese Differenz von alter Mauscursorposition zu neuer Position wird auch als relative Mauskoordinate bezeichnet. Anfängliche Implementierungsvesuche dieser Idee wurden aber frühzeitig abgebrochen, da eine Methode[9] des AWT's, mit der die Maus automatisch an eine bestimmte Position gesetzt werden kann, relativ langsam ist. Weil diese bei einer Mausbewegung des Benutzers permanent aufgerufen werden muss, ließ in Testimplementierungen die Ausführungsgeschwindigkeit sehr zu wünschen übrig.

2. **Simulieren eines Weltkkordinatensystems mittels Mausumbruch**
   Diese Idee wurde abgeschaut von einem Computerspiel namens Asteroids[10], wo der Benutzer ein Objekt mit dem Keyboard steuern kann, das auf der entgegengesetzten Seite des Bildschirms herauskommt, wenn es mit diesem in Kontakt kommt. Genau dieses Verfahren kann auch auf den Mauscursor angewandt werden, womit ebenfalls eine unendliche, stetige Bewegung des Mauscursors, wie auch in der ersten Idee, gegeben ist (Siehe Abbildung 5.3). Der Vorteil bei diesem Verfahren ist der, dass die Maus nur sehr selten automatisch neu gesetzt werden muss. Die Koordinaten an dieser Stelle werden hier als Weltkoordinaten

[9] Hier ist die Methode mouseMove(...) der Klasse java.awt.Robot gemeint
[10] Siehe unter http://www.neave.com/games/asteroids/

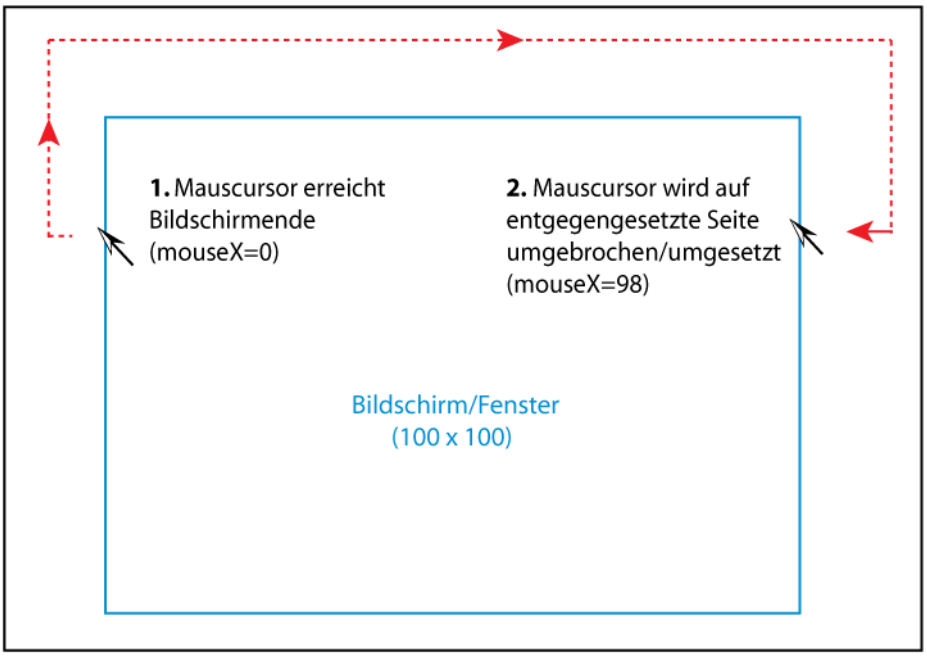


Abbildung 5.3: Mauscursor wird am Bildschirmrand umgebrochen

bezeichnet, da sie nicht beschränkt auf den Bildschirm/Fenster sind, sondern fast beliebig groß gewählt werden können. Testimplementierungen funktionierten hervoragend ohne große Performanceverluste merken zu können.

Folgend ein Codeausschnitt des `Mousehandlers`, welcher die Weltkoordinaten errechnet:

```
1  resetMouse = false;
2
3  if (mx <= 0)
4  {
5    xchange--;  mx = (canvasWidth-1) + mx-1;
6    resetMouse = true;
7  }
8  else
9    if (mx >= canvasWidth-1)
10   {
11     xchange++; mx = mx + 1 - (canvasWidth-1);
12     resetMouse = true;
13   }
14
15 if (my <= 0)
16 {
17   ychange--; my = (canvasHeight-1) + my -1;
```

```
18    resetMouse = true;
19  }
20  else
21    if (my >= canvasHeight-1)
22    {
23      ychange++; my = my +1 - (canvasHeight-1);
24      resetMouse = true;
25    }
26
27  worldCoords.x = ((canvasWidth-2)  * xchange) +
28                                      mx - canvasWidth/2;
29  worldCoords.y = ((canvasHeight-2)  * ychange) +
30                                      my - canvasHeight/2;
31
32  if (resetMouse == true)
33    robot.mouseMove(canvasOffsetX + mx, canvasOffsetY + my);
```

In den Variablen `mx` und `my` sind die aktuellen Mauskoordinaten abgespeichert. Dabei besitzt `mx` den Wertebereich von 0 bis (Breite des Bildschirms/Fensters -1) und `my` von 0 bis (Höhe des Bildschirms/Fensters -1). Die Breite und Höhe des Bildschirms/-Fensters befinden sich in den Variablen `canvasWidth/canvasHeight`. Erreicht nun zum Beispiel die Variable `mx` in Zeile 3 den Wert 0, dann wird dieser in Zeile 5 auf die Breite des Bildschirms/Fensters gesetzt. Die Variable `xchange` zählt hierbei die gesamte Anzahl an Mausumbrüchen. In `resetMouse` wird vermerkt, ob die Maus später umgesetzt werden soll oder nicht. Die Zeilen 27-30 sind schließlich dafür zuständig, die Weltkoordinaten auszurechnen. Dabei wird die Anzahl der Mausümbruche mit der Größe des Bildschirms/Fensters multipliziert. Anschließend wird noch die aktuelle Mausposition addiert und nochmals eine halbe Bildschirmgröße abgezogen, da sich die Anfangsposition des Mauscursors bei Programmstart in der Mitte befindet. In der Letzen Zeile 33 wird mit der AWT-Methode `robot.mouseMove(...)` die Maus umgesetzt, falls ein Umsetzen nötig ist. Deren genaue Funktionsweise kann in der AWT-API nachgelesen werden. `canvasOffsetX` und `canvasOffsetY` sind die Koordinaten der oberen linken Ecke des `GLCanvas`. Im Vollbildmodus haben diese immer den Wert 0, aber im Fenstermodus können sie andere Werte besitzen. Aus den berechneten Weltkoordinaten können über eine andere Methode später zusätzlich auch relative Koordinaten ermittelt werden. Ein Beispiel für die Verwendung des `MouseHandlers` wurde bereits im Bezug auf die Maustasten aufgezeigt. Um mit Weltkoordinaten oder relativen Koordinaten arbeiten zu können, müssen aber noch zusätzliche Methoden des `MouseHandlers` benutzt werden. Sie können über den Konstruktor mit dem Boolean-Wert `true` freigeschaltet werden. Der `MouseHandler` kann so quasi zwischen einer Standard-Variante und eine erweiterten Variante umgestellt

werden. Die Verwendung der erweiterten Variante wird hier nun nicht mehr beschrieben werden, da diese ebenfalls etwas Umfangreich in der Verwendung ist. Es sollte aber darauf hingewiesen werden, dass später 2 fertige JOGLMainListener-Klassen[11] dem Benutzer angeboten werden, wovon eine Klasse[12] bereits den `MouseHandler` in seiner erweiterten Variante implementiert. Ein Einblick in diese gut dokumentierte Klasse wird die vollständige Verwendung des `MousHandlers` transparent machen können. Eine Anmerkung sei hier noch gemacht. Der `MouseHandler` unterstützt über einige Umwege das freie Bewegen der Maus auch in dem Fenstermodus. Diese Feature, realisiert mit Java, wird seltenerer Natur sein, da es bisher noch in keinem anderem Programm beobachtet werden konnte. Selbst bei offiziellen Vorzeigedemos[13] von JGOL, wo diese Unterstützung hätte im Fenstermodus nützlich sein können, wurde auf diese verzichtet.

5.5 Entwickeln der Kamera

In diesem Kapitel wird mit Hilfe der OpenGL-Kamera-Funktion `gluLookAt(...)`, welche aus den Grundlagen von OpenGL bereits bekannt ist, eine dynamische Kamera (`JOGLCamera`) entwickelt.[14] Mit ihr ist es möglich, sich im 3D-Raum, wo die spätere Physiksimulation stattfinden wird, frei umherzubewegen und beliebige Sichtweisen auf interessante Punkte einzunehmen. Diese Möglichkeit des freien Bewegens und Umsehens in 3 Dimensionen ist heutzutage auch in fast jedem modernen Computerspiel zu finden. Diese besitzen dabei meist eine Kamera, welche 3 verschiedene Bewegungsarten unterstützt:

1. **Drehen um die eigenen Achsen**
   Diese Bewegungsart lässt die Kamera um ihre eigene X- und/oder Y-Achse rotieren. Sie verändert hiermit nicht ihre aktuelle Position, sondern in dem Sinne nur ihre Blickrichtung.

2. **Vorwärtsgehen/Rückwärtsgehen**
   Bei dieser Bewegung wird die Kamera entweder vorwärts oder Rückwärts in Blickrichung bewegt.

3. **Seitwärtsgehen und Aufwärts-/Abwärtschweben**
   Bei dem Seitwärtsgehen bewegt sich die Kamera seitwärts relativ zur Blickrichtung. Beim Aufwärts/Abwärtschweben dagegen ist diese dagegen nicht relevant. Hier wird die Kamera entlang der Y-Achse hoch oder runterbewegt.

Mit der Kombination dieser 3 Bewegungsarten kann jede gewünschte Position und Sicht der Kamera herbeigeführt werden. Sogar enge kreisförmige Bewegungen um

[11] Siehe Beschreibungen von Lightweight-/HeavyweightListener in Kapitel 5.8
[12] HeavyweightListener in Kapitel 5.8.1
[13] Zu finden unter [Dev07b]
[14] Beschreibung von gluLookAt(...) in Kapitel 2.3.7

ein Objekt herum sind hiermit möglich. Sie sollen darum von der Klasse `JOGLCamera` implementiert werden.

5.5.1 Drehung um die eigenen Achsen der Kamera

Um eine Drehung um die eigene Achse mit `gluLookAt(...)` zu simulieren, wird sich der Sinus und Cosinus-Funktionen bedient. Zur Errinnerung sei vorher kurz erwähnt, dass `gluLookAt(...)` unter Anderem einen Standort und einen Sichtzentrumspunkt der Kamera verlangt, welche dann eine Sichtlinie bilden. Die Sichtlinie bestimmt die aktuelle Blickrichtung der Kamera. Die Idee hier ist nun, die Lage des Zentrumpunktes, der normalerweise beliebig sein kann, abhängig von einem Drehwinkel und dem Standort zu machen. Zuerst wird ein Einheitskreis definiert auf dem der Zentrumspunkt liegen muss. Der Standort der Kamera befindet sich dabei immer im Mittelpunkt des Kreises. So ist es leicht möglich, die Lage eines benötigten Zentrumspunktes abhängig vom Drehwinkel und dem aktuellen Standort zu errechnen. Abbildung 5.4 zeigt diese Idee noch einmal grafisch, mit dem eine Drehung um die Y-Achse simuliert werden kann:

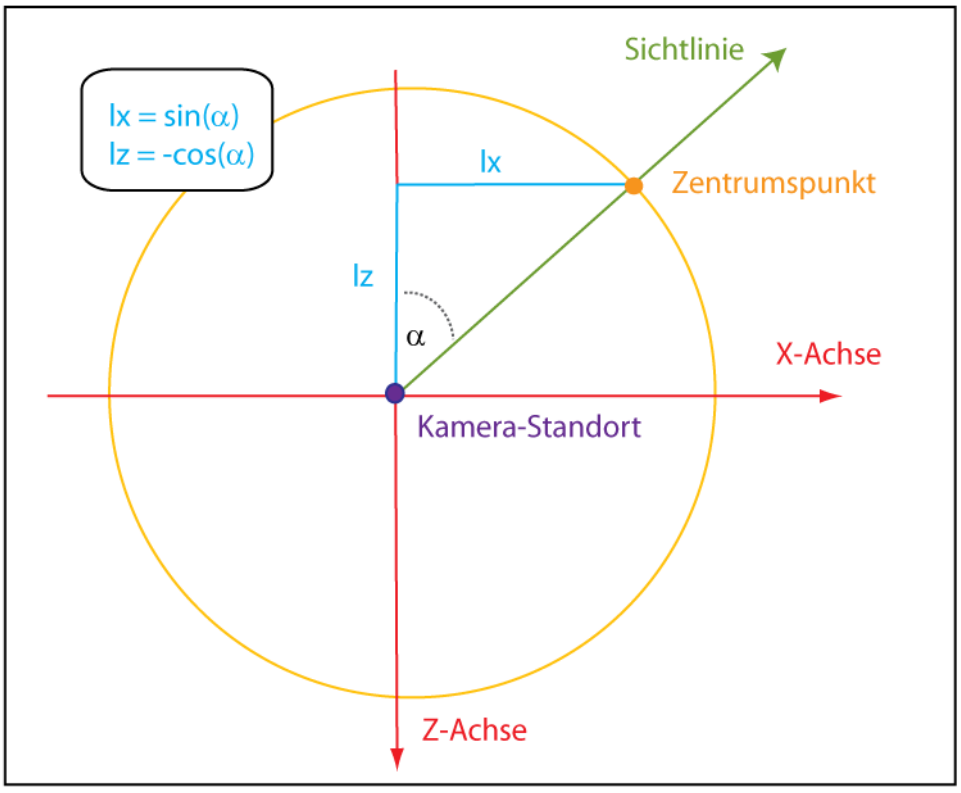


Abbildung 5.4: Drehung der Kamera um die eigene Y-Achse

Zusätzlich folgt ein Java-Codebeispiel dazu:

```
lx = Math.sin(alpha);
ly = 0;
lz = - Math.cos(alpha);

glu.gluLookAt(eyePosX, eyePosY, eyePosZ,
                              eyePosX + lx,
                              eyePosY + ly,
                              eyePosZ + lz,
                            0.0f, 1.0f, 0.0f);
```

Wenn nun zusätzlich auch Bewegungen der Kamera um die X-Achse möglich sein sollen, werden alle 3-Dimensionen des Zentrumspunktes benötigt. Hierfür reicht kein einfacher Einheitskreis mehr aus, sondern es wird eine Kugel mit dem Radius 1 benötigt, auf deren Oberfläche dann der Zentrumspunkt liegen muss. Die Berechnung des Zentrumspunktes dort ist ein wenig umfangreicher und ist in Abbildung 5.5 beschrieben:

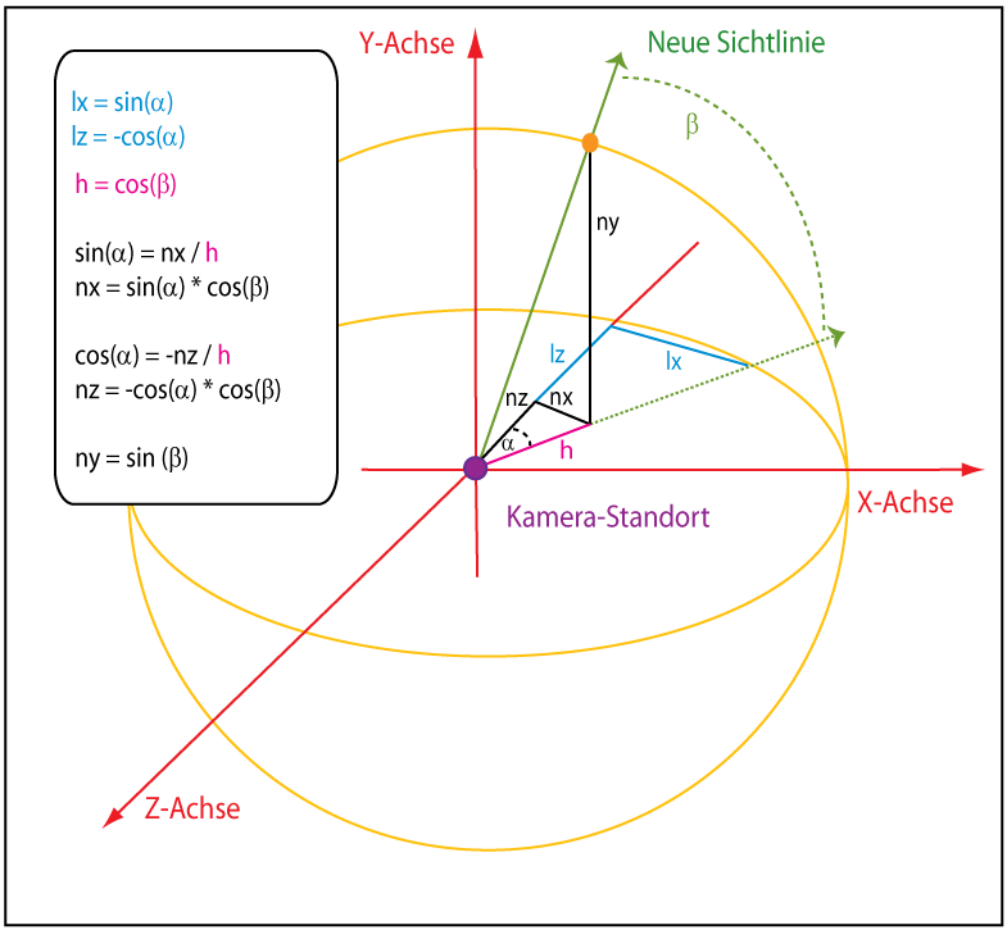


Abbildung 5.5: Drehung der Kamera um die Y- und Z-Achse

Die neue Sichtlinie kann mit Hilfe von `nx`, `ny`, und `nz` herausgefunden werden. Das Berechnen dieser Werte geschieht mit einfachen Sinus/-Cosinus-Funktionen. Es ist ersichtlich, dass nun `nx` und `nz` von den 2 Winkeln α und β abhängig sind, welches zu Folge hat, dass der Zentrumspunkt immer auf der Oberfläche der Kugel bleibt. Genau dieses Verfahren mit einer Kugel, auf deren Oberfläche der Zentrumspunkt wandert, ist in der Klasse `JOGLCamera` realisiert.

5.5.2 Vorwärtsgehen/Rückwärtsgehen

Bei dem Vorwärts- oder Rückwärtsgehen wird sich immer auf die aktuelle Sichtlinie bezogen. Beim Vorwärtsgehen wird dann schrittweise ein bestimmter Teil dieser Sichtlinie entlang gegangen. In welche Richtung gegangen werden muss, kann mithilfe von `nx`, `ny` und `nz` leicht errechnet werden werden.

```
//Vorwärtsgehen
eyePosX += (moveSpeed * nx);
eyePosZ += (moveSpeed * nx);
eyePosY += (moveSpeed * ny);
//Rückwärtsgehen
eyePosX -= (moveSpeed * nx);
eyePosZ -= (moveSpeed * nx);
eyePosY -= (moveSpeed * ny);
```

Mit `moveSpeed` lässt sich die Geschwindigkeit beim vorwärtsgehen/Rückwärtsgehen einstellen. Je größer der Faktor ist, desto größer ist die Schrittweite entlang der Sichtlinie.

5.5.3 Seitwärtsgehen und Aufwärts-/Abwärtsschweben

Das Seitwärtsgehen erfolgt immer seitlich um +90/-90 Grad relativ zur Sichtlinie und funktioniert ähnlich wie das Vorwärtsgehen/Rückwärtsgehen. In der `JOGLCamera`-Klasse ist es wie folgt implementiert:

```
//Seitlich nach links gehen
eyePosX += (moveSpeed * nz);
eyePosZ -= (moveSpeed * nx);
//Seitlich nach rechts gehen
eyePosX -= (moveSpeed * nz);
eyePosZ += (moveSpeed * nx);
```

Hier werden einfach nur die Werte `nz` und `nx` vertauscht. Warum abwechselnd addiert und subtrahiert wird, ist aus folgender Abbildung 5.6 ersichtlich.

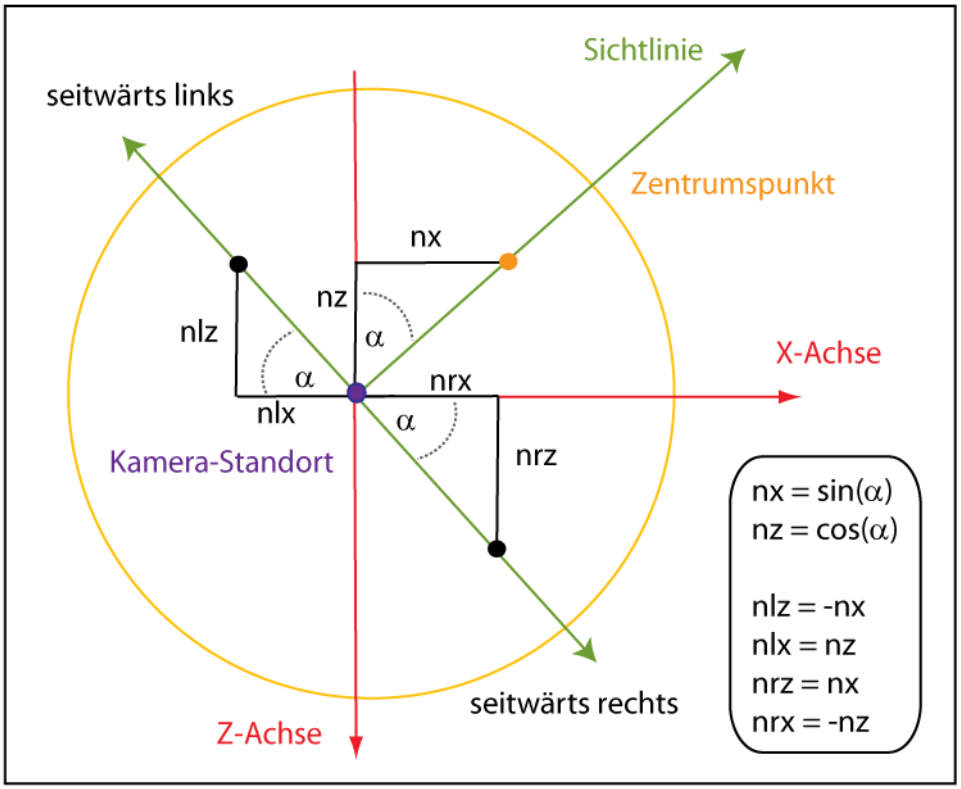


Abbildung 5.6: Seitwärtsgehen relativ zur Sichtlinie

Das Aufwärts- und Abwärtsschweben ist unabhängig von jeglicher Einstellung der Kamera und besitzt auch eine eigene Geschwindigkeit. Sie ist folgendermaßen realisiert:

```
//Aufwärts schweben
eyePosY += (moveSpeedUpDown);
//Abwärts schweben
eyePosY -= (moveSpeedUpDown);
```

5.5.4 Verwendung der Kamera

Bei der Verwendung der `JOGLCamera`-Klasse, wird empfohlen, diese mit der Maus und der Tastatur zugleich zu bedienen. Sehr populär ist dabei eine Maus- und Tastaturbelegung, welche heutzutage in vielen Computerspielen, wie beispielweise „Counter-Strike“[15] seinen Platz gefunden hat. Hierbei wird die Kamera selber über die Tasten `w,a,s,d,e,q` verschoben. Die Blickrichtung (Drehen um eigene Achsen) wird mit der Maus verändert. Andere Belegungen wären bestimmt auch möglich, aber diese hier ist besonders gut auf die gleichzeite Verwendung von Maus und Tastatur abgestimmt, da `w,a,s,d,e,q` am linken Rand der Tastatur liegen und deren Anordung

[15] Sehr bekannter 3D-Shooter: Siehe <http://www.counter-strike.de/>

denen der Pfeiltasten ähnlich ist. Es soll nun kurz an einem Codebeispiel gezeigt werden, wie die Kamera in Verbidung mit dem Framework benutzt werden kann:

```
1  GLU glu          = new GLU();
2  JOGLCamera cam = null;
3
4  public void reshape(GLAutoDrawable glDrawable, int x, int y,
5                                                  int width, int height)
6  {
7      if (cam == null)
8          cam = new JOGLCamera(glu, 0.0f, 1.0f, -1.0f,
9                                                    0.0f, 0.0f);
10 }
11
12 public void display(GLAutoDrawable glDrawable)
13 {
14     Point mp = m.getRelativeMouseXY();
15
16     if (mp.x != 0 || mp.y != 0)
17     {
18         cam.cameraLookV(-mp.y);
19         cam.cameraLookH(mp.x);
20     }
21
22     if (k.checkIsKeyPressed(KeyEvent.VK_W))
23         cam.cameraMoveForward();
24     if (k.checkIsKeyPressed(KeyEvent.VK_S))
25         cam.cameraMoveBackward();
26     if (k.checkIsKeyPressed(KeyEvent.VK_A))
27         cam.cameraMoveSideLeft();
28     if (k.checkIsKeyPressed(KeyEvent.VK_D))
29         cam.cameraMoveSideRight();
30     if (k.checkIsKeyPressed(KeyEvent.VK_E))
31         cam.cameraHoverUp();
32     if (k.checkIsKeyPressed(KeyEvent.VK_Q))
33         cam.cameraHoverDown();
34
35     cam.cameraUpdate(glu);
36 }
```

In Zeile 1 wird ein Objekt erstellt, mit dem Zugriff auf die GLU-Bibliothek möglich ist. Dieses benötigt später die `JOGLCamera`-Klasse, deren Referenz in Zeile 2 definiert wird. In Zeile 8 darauf wird dann erst das `JOGLCamera`-Objekt mit `new` erstellt. Das liegt daran, dass die Funktionalität des `GLU`-Objektes nur innerhalb einer der `GLEventListener`-Methoden gewährleistet werden kann, wie es auch bei der `getGL()`-Methode der Fall ist.[16] Die ersten 3 Parameter `x`,`y`,`z` des Konstruktors von `JOGLCamera` definieren den Standort der Kamera. Der nächste Parameter bestimmt, in welchem Winkel die Kamera nach rechts/links schaut. Der letze Parameter dagegen, in welchem Winkel nach oben/unten geschaut wird. Beide Winkel müssen in rad angegeben werden. In Zeile 18-19 werden darauf relative Mauskoordinaten der Kamera zugeordnet. `cam.cameraLookV(...)` bekommt die Y-Koordinaten und `cam.cameraLookH(...)` die X-Koordinaten übergeben. Die Y-Koordinaten besitzen ein Minuszeichen, da der `MouseHandler` (Hier: `m`) relative Bewegungen des Mauscursors nach oben hin auf dem Bildschirm als negative Werte zurückgibt, die Kamera aber negative Werte als ein Rotieren nach unten auf der Z-Achse interpretiert! Danach werden mittels des `KeyboardHandlers` (Hier: `k`) die entsprechenden Bewegungsmöglichkeiten der Kamera mit der Tastatur assoziert. Mit `cam.cameraUpdate(glu)` muss schließlich die Kamera geupdatet werden, womit die gewünschte Sicht aktualisiert wird.

5.6 Entwickeln der Textausgabe

Einen Text direkt auf dem Zeichenbereich von OpenGL (`GLCanvas`) auszugeben, ist nicht ohne weiteres möglich. Auch können Funktionen aus der `GLU`-Bibliothek diesmal nicht helfen. Doch es gibt noch eine weitere Bibliothek, welche auf OpenGL aufsetzt und mit JOGL mitgeliefert wird. Diese nennt sich `GLUT` und steht für OpenGL Utility Toolkit. Mit `GLUT` ist es möglich, eine betriebssystemunabhängige Anwendung zu schreiben, in welcher dann OpenGL benutzt werden kann. Dabei ist GLUT alleinig für die Verwendung der Programmiersprache C/C++ konzipiert. Es stellt eigene Funktionen bereit, mit denen beispielsweise Fenster erzeugt werden können oder die Maus und Tastatur angesprochen werden. Unter Anderem stellt es zudem noch die nützliche Funktion einer Textausgabe mit OpenGL bereit, welche hier von Interesse ist. Da aber mit Java und JOGL ebenfalls plattformunabhängige Anwendung geschrieben werden können, sollte die Frage aufkommen, warum GLUT dennoch in JOGL enthalten ist. Diese lässt sich dadurch beantworten, dass heutzutage sehr viele C/C++ Programme, welche `GLUT` verwenden, existieren. Um solche Programme gut nach JOGL portieren zu können, wurden einige dieser Funktionen in JOGL implementiert. Sie lassen sich in JOGL über ein `GLUT`-Objekt, welches mit `new GLUT()` erstellt werden kann, aufrufen. Die Klasse `JOGLText`, die nun für die Textausgabe entwickelt wurde, beinhaltet selber nur eine Methode `drawText(...)`, die einen

[16] Siehe Kapitel 3.2.3

beliebigen Text in einer bestimmten Farbe (RGB) und Position (x,y,z) mit der `GLUT`-Methode `glutBitmapString(...)` darstellen kann. Sie wird folgendermaßen angewendet:

```
1  GL gl                = null;
2  GLUT glut            = new GLUT();
3  JOGLText  glText     = new JOGLText();
4
5  public void display(GLAutoDrawable glDrawable)
6  {
7          gl = glDrawable.getGL();
8
9          gl.drawText("HelloWorld!", 255.0f, 0.0f, 0.0f,
10                                    0.0f, 0.0f, -5.0f, gl, glut);
11 }
```

Das Beispiel zeichnet einen roten Text „HelloWorld“ an Position x = 0.0f, y = 0.0f und z = 0.0f. Es ist darauf hinzuweisen, dass die Klasse `JOGLText` nur für kleine Hinweise oder zur Fehlerfindung eingesetzt werden sollte. Es hat sich herausgestellt, dass diese von der Performance her sehr langsam ist und somit nicht für übermäßige Benutzung geeignet ist. Möglich wäre es an dieser Stelle, sich einen eigenen Buchstabensatz aus einzelnen 3D-Objekten zu erstellen und dann eine Methode zu entwickeln, welche diese in einer gewünschten Reihenfolge zeichnet. Da diese Verfahrensweise aber sehr aufwendig ist, wurde auf sie verzichtet. Auch gäbe es noch die Möglichkeit, einen Buchstabensatz über eine Textur zu laden und dann einzelne Texture-Ausschnitte zu verwerten. Auch diese Möglichkeit scheidet aufgrund des Realisierungsaufwandes aus.

5.7 Entwickeln des FpsCounters

Gemäß der Planung des Frameworks, soll dieses auch eine Möglichkeit zur Zeitmessung von zeitkritischen Codestellen bieten. Dabei ist bei dem `JOGLMainListener` die Hauptschleife (`display(...)`) besonders interessant, da sie für eine flüssige Darstellung eine gute Wiederholungsrate und dementsprechend optimierten Code (schnelle Algorithmen usw.) besitzen muss. Der dazu entwickelte `FpsCounter` kann dabei messen, wieviele Schleifendurchgänge und somit Bilder (Frames) die `display()`-Methode in einer Sekunde durchführt. Dabei verwendet der `FpsCounter` die von Java bereitgestellte Zeitmessungs-Methode `System.nanoTime()`. Sie gibt die Zeit in Nanosekunden zurück, die seit einem intern festgelegten Zeitpunk verstrichen ist. So eine Zeit wird auch als „elapsed time“ bezeichnet. Da `System.nanoTime()` selber auch ein wenig Zeit zum Ausführen in Anspruch nemmen wird, sollte für die Zeitmessung diese so wenig wie möglich aufgerufen werden. Die Klasse `FpsCounter` benutzt

folgende Funktion namens `checkFps()` zum Zeitmessen, die nur einmal am Anfang einer Schleife platziert werden muss:

```
1  long now        = 0;
2  long before     = 0;
3  int  frames     = 0;
4  int  fps        = 0;
5
6  public void checkFps()
7  {
8      //FPS messen
9      now = System.nanoTime();
10
11     if (now >= (before + 1000000000))
12     {
13         before = System.nanoTime();
14         fps = frames;
15         frames = 1;
16     }
17     else
18         frames++;
19 }
```

Die Zeitmessungsroutine arbeitet wie folgt. Am Anfang in Zeile 9 wird immer jeweils die aktuelle „elapsed time“ in der Variablen `now` abgelegt. In Zeile 13 wird ebenfalls eine „elapsed time“ namens `before` gemessen, welche allerdings nur jede Sekunde aktualisiert wird. Die `if`-Abfrage testet nun die Differenz dieser beiden Zeiten, indem sie abfragt, ob die aktuell verstrichene Zeit bereits größer oder gleich der vor einer Sekunde gemessenen Zeit + 1 Sekunde (1 Milliarde Nanosekunden) ist. Ist dies nicht der Fall, wird ein Frame (Schleifendurchgang) mit `frame++` hochgezählt. Ist eine Sekunde vergangen, wird die `before`-Variable wieder aktualisiert und die Variable `frame` zurückgesetzt. Sie bekommt nicht den Wert 0, sondern den Wert 1, da dort bereits gerade ein neuer Schleifendurchgang geschieht. Die fps der aktuellen Sekunde werden in der Variablen `fps` abgesichert. Allerdings funktioniert dieser Algorithmus erst ab der 2. Sekunde, hat aber den Vorteil, dass `System.nanoTime()` nur einmal pro Schleifendurchgang ausgeführt wird. Da `System.nanoTime()` immer auch selber ein wenig Zeit zum ausführen braucht, sollte das Messergebnis weniger verfälscht sein, als wenn jeweils immer am Anfang und am Ende einer Schleife `System.nanoTime()` zum Einsatz käme. Es gäbe noch die Möglichkeit, die Zeit, welche `System.nanoTime()` selber verbraucht, mit in die Berechnung einzubeziehen. Versuche hierzu schlugen aber fehl, da die Ausführungszeit von `System.nanoTime()` sehr klein ist und zu sehr

varriert. Auch der Versuch, `System.nanoTime()` mehrere 1000 mal aufzurufen und dann einen Mittelwert zu bilden, funktionierte nicht immer, weil ab und zu einzelne Werte sehr aus dem Ruder liefen. Die Messgenauigkeit des `FpsCounters` sollte aber trotzdem für einfache Geschwindigkeitstests vollkommen ausreichen. Er kann wie folgt verwendet werden:

```
1 FpsCounter fpsCounter = new FpsCounter();
2
3 public void display(GLAutoDrawable glDrawable)
4 {
5     fpsCounter.checkFps();
6
7     System.out.println("Aktuelle fps: " +
8                                 fpsCounter.getActFps());
9 }
```

5.8 Der Heavyweight- und LightweightListener

Das Framework ist so aufgebaut, dass der Benutzer sich eine eigene neue Klasse erstellen muss, welche die Klasse `JOGLMainListener` erbt. In dieser wird die Physiksimulation programmiert und die dafür eventuell nötigen Klassen des Frameworks wie Beispielsweise `JOGLCamera` und `JOGLMousHandler` eingebunden. Zwar nehmen diese dem Benutzer einen erheblichen Programmieraufwand ab, aber deren Einbindung, Initialisierung und Verwendung ist trotzdem noch mit ein wenig Mühe verbunden. Aus diesem Grund bietet das Framework bereits 2 fertige Klassen (Lightweight- und HeavyweightListener) an, mit denen der Benutzer direkt arbeiten kann, ohne erst eine neue Klasse erstellen zu müssen. Mit Hilfe dieser Listener, benötigt der Benutzer nur sehr wenige Zeilen Code, um das Framework an seine Wünsche anzupassen und kann direkt mit der Programmierung der Physiksimulation beginnen. Ein ausführliches Verwendungsbeispiel hierzu ist in Kapitel 5.9 zu finden.

5.8.1 Beschreibung des HeavyweightListeners

Der `HeavyweightListener` beinhaltet bereits alle Komponenten des Frameworks, welche bereits teilweise spezielle Voreinstellungen besitzen:

- FpsCounter (Misst Zeit der display(...)-Methode)
- JOGLText (Gibt Verwendungshinweise aus und zeigt fps an)
- JOGLCamera (Besitzt hier ein mittiges Quadrat zur Orientierungshilfe)
- KeyboardHandler (Einige Tasten bereits vorbelegt)

  - w,a,s,d,e,q für Kamerabewegung
  - F1 um Verwendungshinweise ein/auszublenden
  - Escape zum Beenden des Programmes
- Zwei MouseHandler
  - Rechte Maustaste gedrückt –> Kamerabewegung
  - Linke Maustaste gedrückt –> normale Bewegung des Cursors
- Komplette Initialisierung von OpenGL

Da der `HeavyweightListener` recht viel Programmcode[17] enthält, welchen der Benutzer aber nicht komplett zu verstehen braucht, sind gut gekennzeichnete Einstiegspunkte im Code (Wie auch im `LightweightListener`) mit Kommentarblöcken versehen, welche wie folgt aussehen (Hier: `display(...)`-Methode):

```
1  //-----------------------------------------------------------------
2  // **** Anfang eigener Code fuer Hauptschleife hier     ****
3  //
4  // (Anstatt glLoadIdentity() sollte hier
5  //  gl.popMatrix() benutzt werden um Konflikte mit der
6  //  Kamera zu vermeiden!)
7  //-----------------------------------------------------------------
8
9  //                      - Code -
10
11 //-----------------------------------------------------------------
12 // **** Ende eigener Code fuer Hauptschleife           ****
13 //-----------------------------------------------------------------
```

In Abbildung 5.7 ist der `HeavyweightListener` in seiner „Rohfassung“ zu sehen, wobei der Benutzer noch keine einzige Codezeile hinzugefügt hat. Hier ist bereits ein grafischer 3D-Ursprung vorimplementiert, um dem Benutzer eine bessere Orientierung zu ermöglichen. Dieser kann aber bei belieben an folgender gekennzeichneter Codestelle (In der `display(...)`-Methode) deaktiviert werden:

```
1  //Ursprung zeichnen (Diese Zeile kann bei Bedarf
2  //                                  entfernt werden!)
3  origin.drawOrigin(gl);
```

[17] Kompletter Quellcode hierzu im Anhang in Kapitel A.3.2

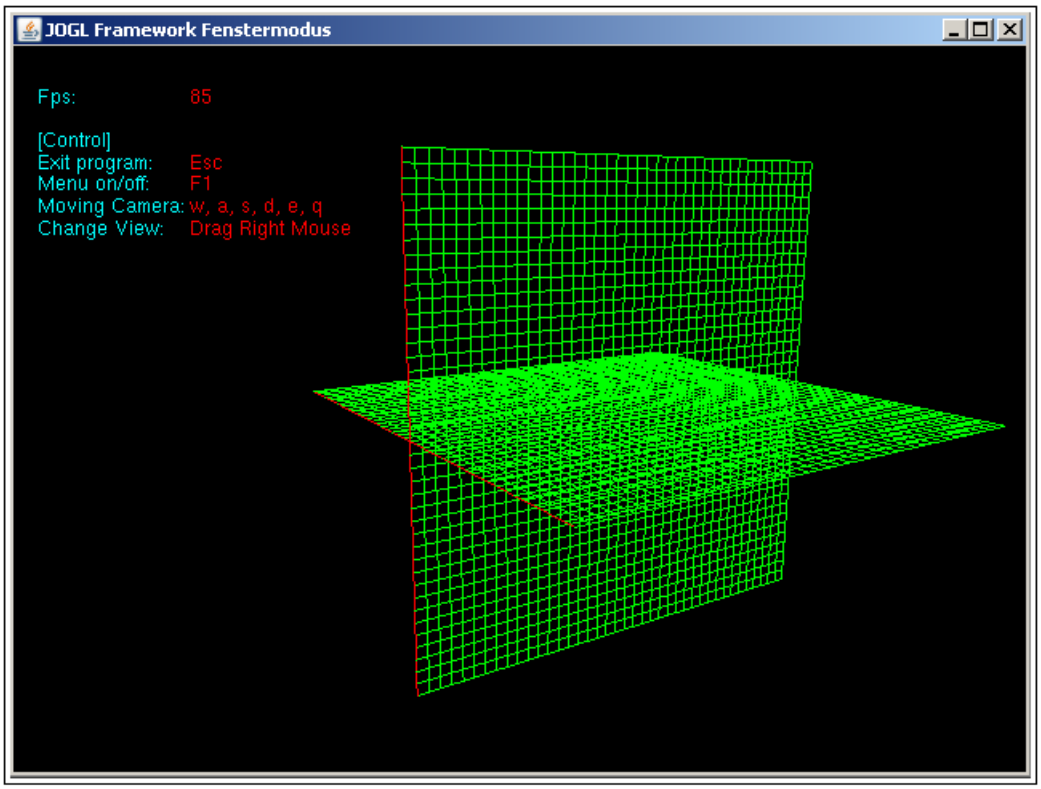


Abbildung 5.7: HeavyweightListener in seiner Rohfassung

5.8.2 Beschreibung des LightweightListeners

Die `LightweighListener`-Klasse ist für erfahrene Benutzer des Frameworks gedacht. Während der `HeavyweightListener` bereits alle möglichen Funktionalitäten des Frameworks ausschöpft und sehr viel vorimplementiert, enthält der `LightweighListener` nur die Dinge, welche in jeder Physiksimulation gebraucht werden.[18] Hier wird der Benutzer nicht behindert durch eventuell störende Vorimplementierungen von Funktionalitäten, die er garnicht benötigt. Werden später doch Komponenten, wie beispielsweise `MouseHandler` oder `JOGLCamera` benötigt, kann deren Verwendung leicht bei dem `HeavyweightListener` abgeschaut werden! Der `LightweightListener` beinhaltet folgende fertige Implementierungen:

- KeyboardHandler (Abbruch des Progammes mit Escape möglich)
- Komplette Initialisierung von OpenGL

In seiner Rohfassung, ohne dass der Benutzer eine Codezeile eingetragen hat, stellt der `LightweighListener` nur ein leeres Fenster dar (Abbildung 5.8).

[18] Kompletter Quellcode hierzu im Anhang in Kapitel A.3.3

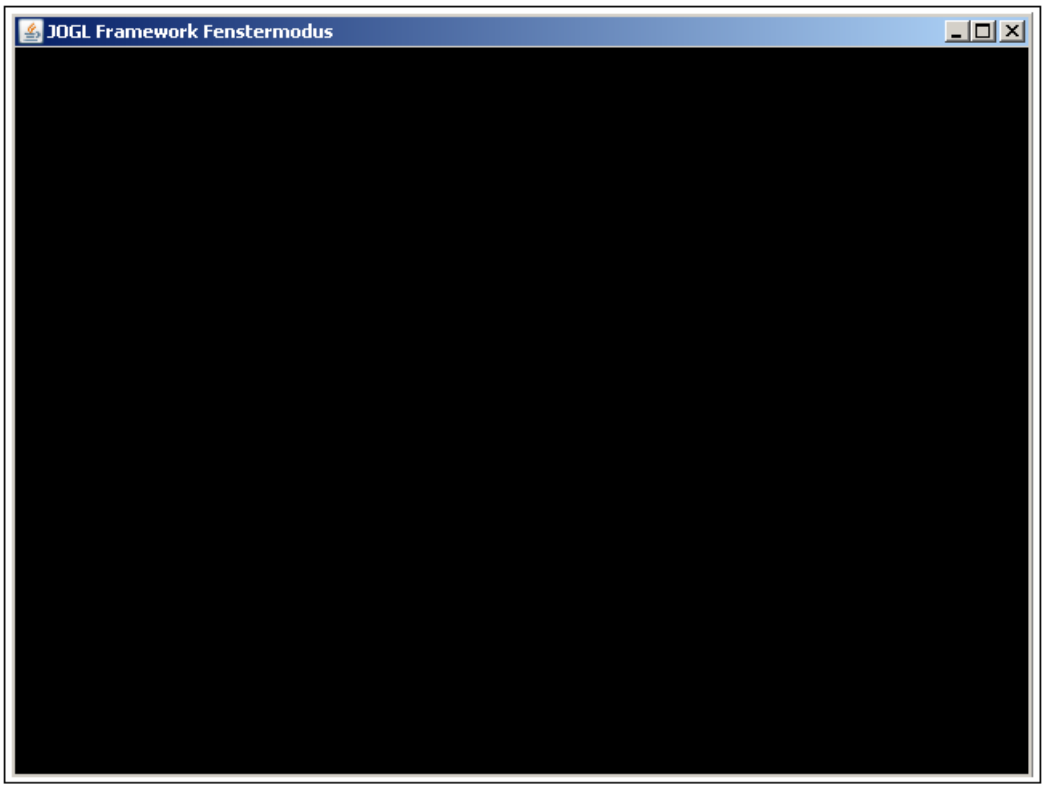


Abbildung 5.8: LightweightListener in seiner Rohfassung

5.9 Verwendung des Frameworks

Die Verwendung des Frameworks mit kompletter Unterstützung aller Funktionalitäten ist dank der beiden Listener aus Kapitel 5.8 recht einfach. Es wird nun anhand eines einfachen Codebeispiels mit dem `HeavyweightListener` gezeigt, wie das Framework später verwendet werden soll. Dabei wird wieder das Beispiel aus Kapitel 2.3.9 und 3.2.3 benutzt, welches ein rotes Dreieck rotieren lässt. Zuerst müssen die entsprechenden OpenGL-Befehle in einen dafür vorgesehen Block des `HeavyweightListeners` (Hier: `display(...)`-Methode) eingetragen werden:

```
1  //-------------------------------------------------------------------
2  // **** Anfang eigener Code fuer Hauptschleife hier     ****
3  //
4  // (Anstatt glLoadIdentity() sollte hier
5  //  gl.popMatrix() benutzt werden um Konflikte mit der
6  //  Kamera zu vermeiden!)
7  //-------------------------------------------------------------------
8  gl.glColor3f(255.0f , 0.0f, 0.0f);
9  gl.glTranslatef(1.0f, 1.0f,-1.0f);
10
```

```
11 //(Die Variable "angle" wurde am Anfang des Listeneres
12 // vorher global definiert!)
13 gl.glRotatef(angle+=1.0f,0.0f, 1.0f,0.0f);
14
15 gl.glBegin(GL.GL_TRIANGLES);
16     gl.glVertex3f( -1.0f, -1.0f, 0.0f);
17     gl.glVertex3f(  1.0f, -1.0f, 0.0f);
18     gl.glVertex3f(  0.0f, 1.0f,  0.0f);
19 gl.glEnd();
20 //--------------------------------------------------------------
21 // **** Ende eigener Code fuer Hauptschleife              ****
22 //--------------------------------------------------------------
```

Danach muss eine Klasse erstellt werden, welche das Framework startet bzw. die einzelnen nötigen Klassen miteinander verknüpft. Diese wird allerdings auch schon vorimplementiert in einer Klasse `RunFrameWork` bereitgestellt![19]

```
1  public class RunFrameWork
2  {
3     public static void main(String[] args)
4     {
5        javax.swing.SwingUtilities.invokeLater(new Runnable()
6        {
7           public void run()
8           {
9              JFrame.setDefaultLookAndFeelDecorated(true);
10             JOGLMainFrame runFrame = new JOGLMainFrame();
11             runFrame.addJOGLMainListener(
12                          new HeavyweightListener(runFrame));
13
14             DispMenu menu = new DispMenu("Rotes Dreieck",
15                                                  runFrame);
16          }
17       });
18    }
19 }
```

[19] Kompletter Quellcode hierzu im Anhang in Kapitel A.3.1

In den Zeilen 2-3 werden die nötigen Klassen des Frameworks eingebunden. In einen separaten Thread wird dann ein neues `JOGLMainFrame` erstellt und diesem eine neue Instanz des `HeavyweightListener` hinzugefügt. Zuletzt wird das `JOGLMainFrame` mit dem Anzeigemenü `DispMenu` verknüpft und darauf automatisch das Anzeigemenü gestartet. Da die Klasse `RunFrameWork` bereits fertig mit dem Framework mitgeliefert wird, muss der Benutzer so *nur* insgesamt 9 Zeilen (inklusive `angle`-Variable) Programmcode schreiben, um ein rotes Dreieck mittels JOGL/OpenGL auf dem Bildschirm rotieren lassen zu können. Um das selbe Beispiel mithilfe des LightweightListeners zu realisieren, wird genauso Verfahren, wie in dem HeavyweightListener. Der einzige Unterschied ist der, dass in der Klasse RunFrameWork hier nun explizit der LightweightListener angegeben werden muss:

```
1 //Anstatt folgender Zeile...
2 runFrame.addJOGLMainListener(
3                                 new HeavyweightListener(runFrame));
4
5 //...muss nun diese dort platziert werden
6 runFrame.addJOGLMainListener(
7                                 new LightweightListener(runFrame));
```

Zur Veranschaulichung wird nun zuletzt nocheinmal der `HeavyweightListener` mitsamt dem roten Dreieck abgebildet:

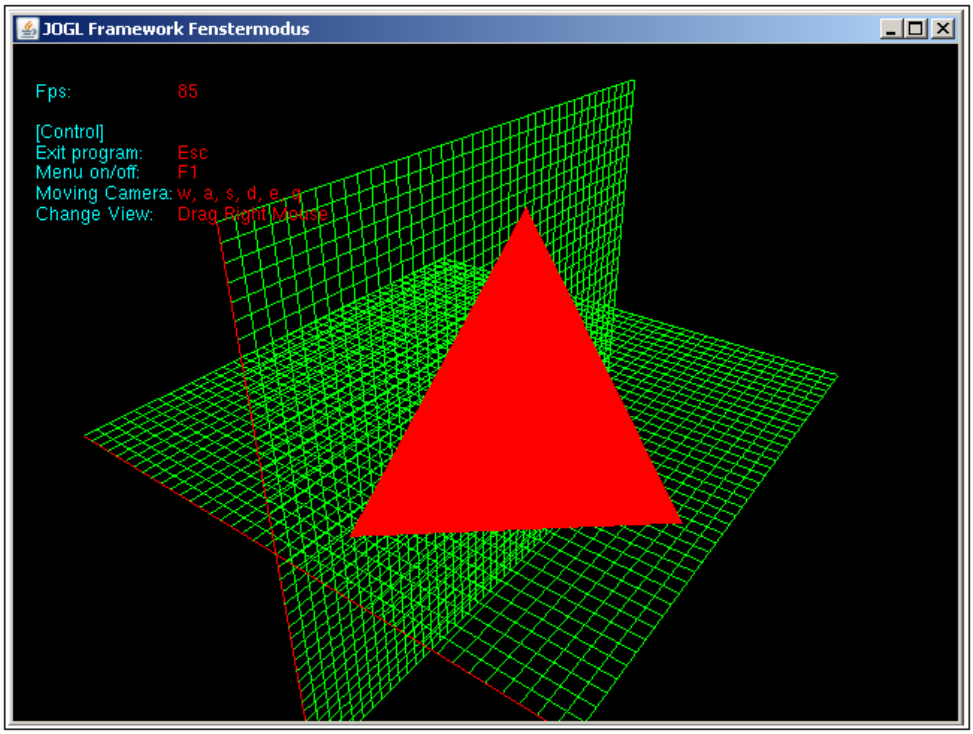


Abbildung 5.9: HeavyweightListener mit einem rotierenden, roten Dreieck

6 Portierung einer C/C++ Physiksimulation in das Framework

6.1 Portierung einer C/C++ Planeten-Physiksimulation

Um zu testen, ob das neue Framework auch wirklich für den Einsatz im echten Betrieb an der Fachhochschule geeignet ist, wurde eine bereits vorhandene Physik-Simulation des alten Frameworks aus dem Wahlpflichtfach „Spiele, Simulation, dynamische Systeme“ in das neue Framework portiert. Die zu portierende Physiksimulation wurde volständig in C/C++ realisiert und simuliert ein Mond-Erde System. Dort wird ein Mond mit 2-3 Schwerpunkten dargestellt, welcher mit einer bestimmten Startgeschwindigkeit um die Erde kreist. Die Simulation zeigt, dass nach einer gewissen „Einpendelzeit“ der Mond für eine Rotation um seine eigene Achse genau so lange braucht, wie für eine gesamte Umrundung der Erde. Hierduch zeigt der Mond der Erde sozusagen immer sein gleiches „Gesicht“. In Abbildung 6.1 ist die alte C/C++ Simulation zu sehen und Abbildung 6.2 zeigt diese im neuen JOGL-Framework.

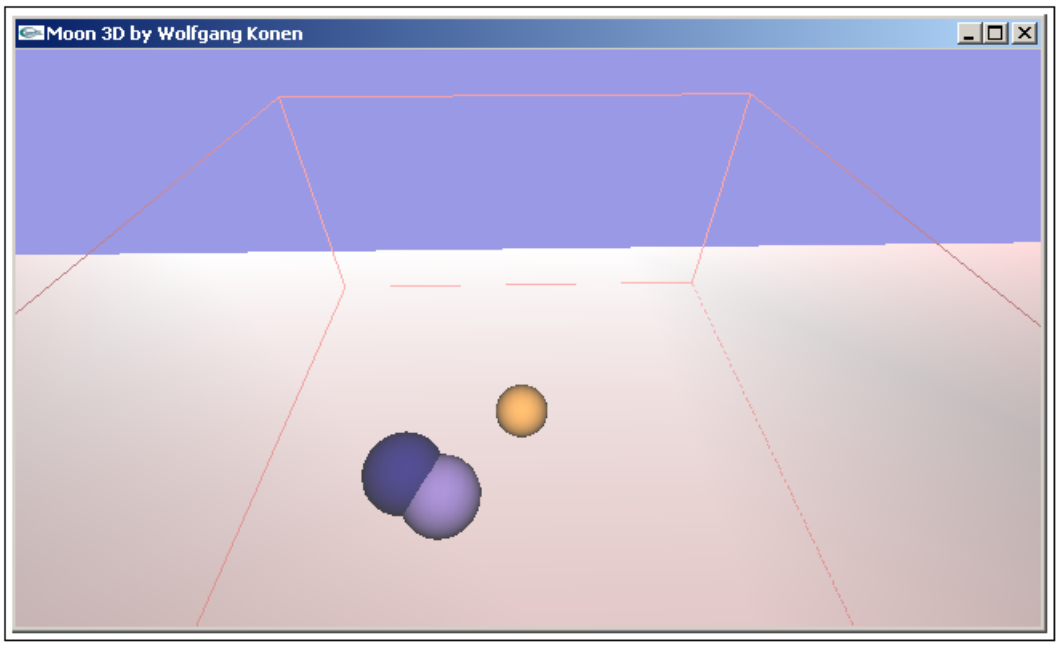


Abbildung 6.1: Altes Framework: C/C++ Planeten-Simulation

Wie zu erkennen ist, sehen beide Simulationen völlig gleich aus. Der einzige Unterschied ist der, dass bei der Java-Version die neue JOGL-Kamera verwendet wird. Die alte Simulation besaß zwar auch eine Kamerafunktion, welche aber nicht ganz so flexibel war und deshalb ausgetauscht wurde. Zusätzlich werden in der Java-Version noch Informationen bezüglich der Kamersteuerung und die aktuellen FPS links am Bildschirmrand angezeigt.

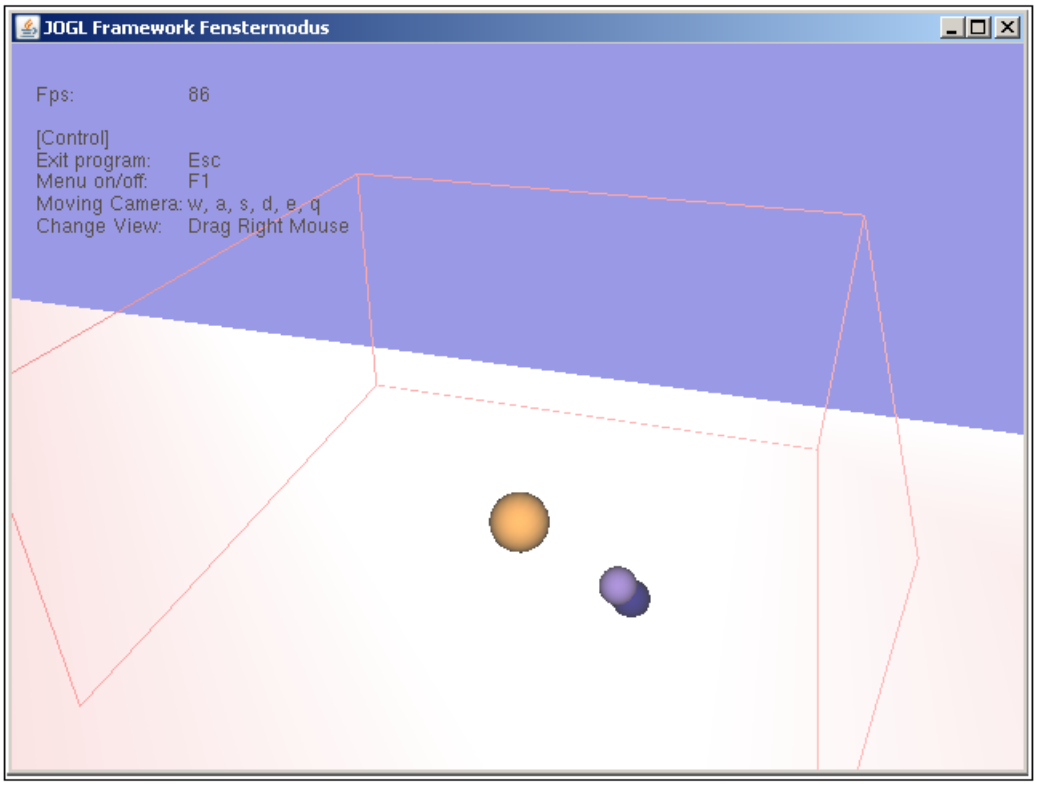


Abbildung 6.2: JOGL Framework: Java Planeten-Simulation

6.2 Zu beachtende Punkte bei einer Portierung von C/C++ nach Java

Damit die C/C++ Planeten-Physiksimulation in das neue Framework portiert werden konnte, mussten erst die für die Portierung kritischen Punkte herrausgesucht werden.

1. **Programmiersprache C/C++ wird verwendet**

2. **Simulation Lässt Interaktion mit Maus und Keyboard zu**

3. **Simulation Verwendet Funktionen der GLU-Bibliothek**

Da mit JOGL eine Verwendung der GLU-Bibliothek möglich ist und das neue Framework mit Maus und Keyboard sehr gut umgehen kann, hätte es theoretisch wenig Hürden bei der Portierung geben sollen. Doch in der Praxis stellte sich heraus, dass es doch ein paar bestimmte Punkte gibt, die *generell* beim Portieren einer C/C++ OpenGL-Anwendung nach Java/JOGL beachtet werden müssen. Diese beziehen sich zum größten Teil auf wichtige Unterschiede der beiden Programmiersprachen C/C++ und Java. Einige für die Portierung relevante Unterschiede sind folgende:

- **Makros**
  C/C++ unterstützt die Verwendung von Makros. Einem Makro kann dabei ein beliebiges Stück Programmcode zugewiesen werden, welches dann anstelle des Programmcodes im Programm verwendet werden kann. Bevor das Programm kompiliert wird, ersetzt ein Präprozesser dann alle diese Makros durch den assoziierten Programmcode. Dadurch können Programme sehr klein und übersichtlich gehalten werden. Ein Makro wird mit `#define` gefolgt von dem Makronamen und dann dem Programmcode erstellt. Es besitzt auch die Möglichkeit einer Parameterübergabe, aber an dieser Stelle soll nur ein einfaches Beispiel gegeben werden:

  ```
  //Makro definieren am Anfang des Programmes
  #define schleife for(int i=0; i< 10 ;i++)

  //Später weiter unten im Programm
  schleife
     printf("%i\n",i);
  ```

  Da Java aber die Verwendung von Makros nicht unterstützt (Java hat keinen Präprozessor), müssen diese bei einer Portierung aufgelöst werden.

- **Pointer**
  Da Java keinen direkten Zugriff auf den Speicher erlaubt, können dort keine Pointer benutzt werden. Sind in dem zu portierenden Programm Pointer vorhanden, muss deren Funktion herausgefunden werden und entsprechende Äquivalente in Java benutzt werden. Hierzu 2 Beispiele:

  ```
  // C/C++ -Programm
  int addiere(int * feld, int fieldSize)
  {
     int summe = 0;
     for (int i=0;i<fieldSize;i++)
        summe+=feld[i];

     return summe;
  ```

```
}

// Mögliche Java-Portierung
int addiere(int[] feld)
{
   int summe = 0;
   for (int i=0;i<feld.length;i++)
      summe+=feld[i];

   return summe;
}



// C/C++ -Programm
int * x = (int *) malloc(3*sizeof(int));
// Mögliche Java-Portierung
int   x = new int[3];
```

- **Structs**
  Besitzt das zu portierende Programm `structs`, müssen diese in Java als Klasse umgesetzt werden. Vorsicht ist beim Arbeiten mit einfachen `struct`-Arrays geboten. Folgendes Beispiel zeigt einen leicht gemachten Fehler beim portieren, der unter Umständen schwer zu finden sein kann.

```
//*** C/C++ Programm ***
struct test
{
   int a;
   int b;
}

test x[2];

x[0] = x[1];

//*** Java Programm ***
class test
{
   int a;
   int b;
}
```

```
test[] x = new test[2];
x[0] = new test();
x[1] = new test();

x[0] = x[1]; // --> Fehler!
```

Der Fehler liegt hierin, das C/C++ eine sogenannte tiefe Kopie macht und Java nur eine flache Kopie. Während C/C++ die einzelnen Werte a,b dem anderen Struct „rüberkopiert“, kopiert Java nur die *Referenzen* der Objekte, wodurch `x[0]` und `x[1]` auf ein und dasselbe Objekt zeigen.

6.3 Besonderheiten von JOGL bei einer Portierung

Wichtig ist auch zu erwähnen, das sich JOGL zwar an die Vorgaben der OpenGL-Spezifikation hält, aber trotzdem bestimmte Parameter ein anderes Format aufweisen. Einige OpenGL-Funktionen müssen somit ein wenig anders parametrisiert werden, als wie beispielsweise in C/C++. Dies liegt daran, dass OpenGL-Funktionen teils einen bestimmten Pointer als Parameter erwarten. Zur Veranschaulichung wird nun eine von den betroffenen Funktionen gezeigt:

```
void glLightfv(GLenum light, GLenum pname, const GLfloat *params )
```

Mit dieser Funktion kann eine Lichtquelle spezifiziert werden. Interessant an ihr ist nur der letze Parameter, welcher als konstanter Pointer vom Typ `GLFloat` übergeben werden muss. Da es diese Form der Übergabe aber in Java nicht gibt, werden von JOGL 2 Ausweichmöglichkeiten bereitgestellt. Einmal kann statt dem Pointer ein `FloatBuffer` oder aber auch ein normales `float`-Array übergeben werden. Beim `float`-Array muss allerdings noch der Start-Index des Arrays mit angegeben werden. Hier ein Portierungs-Beispiel:

```
// *** In C/C++ ***
GLfloat LightTest[]= { 0.5f, 0.5f, 0.5f, 1.0f };
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightTest);

// *** In Java/JOGL ***
float LightDiffuse[] = { 0.9f, 0.7f, 0.7f, 1.0f };
gl.glLightfv(GL.GL_LIGHT0, GL.GL_DIFFUSE,
                           FloatBuffer.wrap(LightDiffuse));

//oder
float LightDiffuse[] = { 0.9f, 0.7f, 0.7f, 1.0f };
gl.glLightfv(GL.GL_LIGHT0, GL.GL_DIFFUSE, LightDiffuse, 0);
```

Bei Beachtung aller der genannten Punkte aus den Kapitel 6.2 und 6.3 sollte es möglich sein, einen Großteil von OpenGL-Anwendungen, welche in C/C++ geschrieben wurden, nach Java/JOGL zu portieren. Für eine Vertiefung und Studium weitere Besonderheiten sei auf die JOGL-API[1] verwiesen, welche sehr ausführlich diesbezüglich beschrieben ist.

[1] Vgl. [Dev07a]

7 Performancetest des Frameworks

In diesem Kapitel wird das neue Framework einem ausführlichen Performancetest unterzogen. Dabei behandelt das Kapitel drei unterschiedliche Performanceuntersuchungen. Zuerst werden in Kapitel 7.1 Tests durchgeführt, welche die generelle Performance des alten und des neuen Frameworks miteinander vergleichen. Da diese Tests sehr auf bestimmte Schwerpunkte zugeschnitten sind und Extremfälle darstellen, werden solche Fälle in der Praxis recht selten vorkommen. Aus diesem Grund kommt in Kapitel 7.2 eine richtige Physiksimulation[1] aus der Praxis zum Einsatz, deren Performance einmal unter dem alten und dann dem neuen Framework untersucht wird. Im letzen Kapitel 7.3 soll dagegen die generelle Einsatzfähigkeit des Frameworks auf verschiedenen Rechnern getestet werden, um eine ungefähre Richtlinie zu erhalten, wieviel Performance die Rechner für ein moderates Arbeiten aufweisen müssen.

7.1 Vergleich des alten und neuen Frameworks

Bevor nun die Geschwindigkeit des alten Frameworks mit dem neuen verglichen wird, sind nocheinmal deren für die Performanceuntersuchung interessanten Punkte aufgelistet:

1. **Alte Framework**
   - **Verwendet C/C++**
     Hierdurch wird schneller, direkt vom Betriebssystem ausführbarer Maschinencode erzeugt.
   - **Ruft Befehle direkt aus der OpenGL-Bibliothek auf**
     Da die OpenGL-Implementierung selber in nativem C implementiert ist, stellt der Aufruf der OpenGL-Funktionen von einem C/C++ Programm aus den kürzesten und schnellsten Weg dar.
2. **Neue Framework**
   - **Verwendet Java**
     Java erzeugt einen unabhängigen Bytecode, welcher durch einen Interpreter(Java Virtual Maschine) ausgeführt wird. Die Java Virtual Maschine

[1] Planeten-Physiksimulation aus Kapitel 6.1

besitzt hierbei einen sogenannten HotSpot-Compiler, der bestimmte zeitkritische Teile eines Programmes *während* der Laufzeit in nativen Maschinencode kompiliert. Somit sollte ein Java Programm „theoretisch“ mit der Geschwindigkeit eines C/C++ Programmes mithalten können.

- **Benutzt das JNI um OpenGL-Befehle aufzurufen**
  Der nötige Umweg über das JNI, um OpenGL-Befehle aufzurufen, dürfte Geschwindigkeitseinbußen mit sich ziehen.

7.1.1 Vorbereitung und Definition der Testfälle

Vorbereitung

Um die Messergebnisse der Testfälle nicht zu beeinträchtigen, wurden alle Hintegrundienste des Betriebssystems heruntergefahren um deren Einwirken zu verhindern. Auch wurde auf Keyboard oder Mausunterstützung in den Testfällen verzichtet. Getestet wurde auf einem PC-System mit einem Pentium 4 3,2 GHz und einer NVIDIA Geforce 6800 Grafikkarte. Dieses ist somit für das Arbeiten mit OpenGL bestens vorbereit

Testfall-Definitionen

Um aussagekräftige Performancetests erhalten zu können, wird ein OpenGL Testprogramm jeweils einmal mit Java/JOGL und einmal mit C/C++ realisiert werden. Zur Zeitmessung in Java wird dabei die Methode `checkFps()` der Klasse `FpsCounter` verwendet werden. In C wird eine äquivalente Zeitmessungsroutine mit der Methode `QueryPerformanceCounter(...)` zum Einsatz kommen. Die Zeitmessung bezieht sich dabei immer auf die Hauptschleife der OpenGL-Programme in der auch die OpenGL-relevanten Funktionen benutzt werden. Ein Test wird dabei 5 mal hintereinander ausgeführt und dann der Mittelwert der gemessenen fps gebildet. So kann vermieden werden, das eventuelle Schwankungen bei einigen Messungen das Ergebnis zu stark verfälschen. Auch sollen die Tests in Java lange genug laufen, um dem Hotspot-Compiler nötige Optimierungen zu ermöglichen. Insgesamt sollen 3 Testfälle erstellt werden:

1. **Aufrufgeschwindigkeit des JNI**
   Hier soll herausgefunden werden, ob das JNI einen Flaschenhals darstellt.

2. **OpenGL-Performance**
   Dieser Test soll Aufschluss darüber geben, wie schnell OpenGL-Befehle von der Grafikkarte verarbeitet/verwertet werden. Theoretisch sollte dies unabhängig von der Programmiersprache sein, da hierfür nur OpenGL und die Grafikkarte selber zuständig sind.

3. **Java vs. C/C++**
   In diesem Testfall wird schließlich überprüft, ob Java wirklich sehr viel langsamer als C/C++ ist, wie sehr häufig die Meinung vertreten wird.

7.1.2 Testfall1: Aufrufgeschwindigkeit des JNI

Dieser Test wurde dafür konzipiert, gezielt die Aufrufgeschwindigkeiten der OpenGL-Funktionen zu testen. Hierbei wurden ca. 66.000 kurzlebige OpenGL-Befehle pro Schleifendurchgang aufgerufen. Mit kurzlebigen Befehlen ist gemeint, dass diese von der GPU[2] der Grafikkarte sehr schnell abgearbeitet werden. Das hat den Sinn, dass die GPU das Testprogramm nicht ausbremst. So sollte schnell erkannt werden, ob das JNI einen Flaschenhals darstellt. In Abbildung 7.1 und 7.2 ist das Testergebnis zu sehen. Deutlich ist zu erkennen, dass das JOGL Program nur etwas ein Drittel

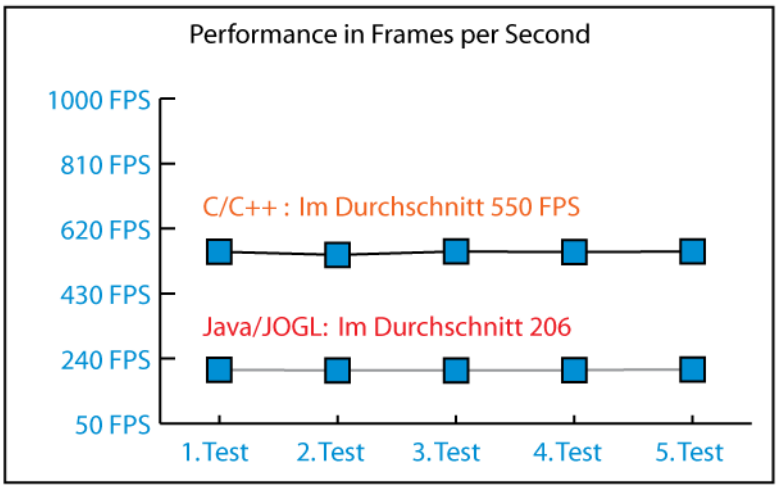


Abbildung 7.1: Aufrufgeschwindigkeiten: Performance in fps

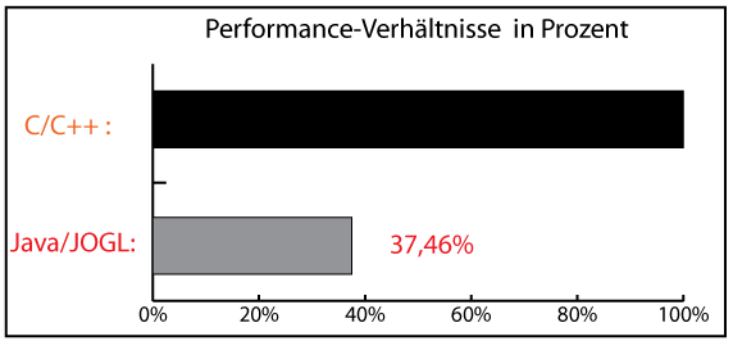


Abbildung 7.2: Aufrufgeschwindigkeiten: Performance-Verhältnis

der Performance aufweisst, wie das dazugehörige C/C++ Programm. Wäre die GPU bei dem JOGL-Programm ausgelastet gewesen, hätte das C/C++-Programm nicht ca. 60% mehr fps aufweisen können. Da hiervon ausgegangen wird, das Java und C/C++ in etwa gleich schnell sind, kann dann so einen gravierenden Unterschied nur das JNI verursacht haben. Die spätere Performanceuntersuchung von Java und C++ selber, wird diese Aussage bekräftigen (Siehe Kapitel 7.1.4).

[2]GPU = Graphics Processing Unit -> Prozessor der Grafikkarte zur Grafikverarbeitung

7.1.3 Testfall2: OpenGL-Performance

Neben der Aufrufgeschwindigkeit der OpenGL-Funktionen ist aber zudem noch die Ausführgeschwindigkeit wichtig. Diese sollte wie bereits erwähnt unabhängig von der Progammiersprache sein. Um sie zu testen, wurden diesmal sehr für die Grafikkarte rechenlastige Operationen ausgeführt. Es wurden mittels DisplayListen[3] 1000 Quadrate mit Alphablendingeffekten[4] direkt in der Grafikkarte verstaut, welche dort zum Zeichnen bereit standen. Das Zeichnen der Würfel konnte dann über einen einzigen OpenGL-Befehl an die Grafikkarte gestartet werden, womit der vermutete Flaschenhals des JNI's sozusagen „ausgehebelt" wird. Ist die Grafikkarte hierbei völlig ausgelastet, sollte die Performance des JNI-Aufrufes nicht mehr relevant sein, da jetzt die Grafikkarte den Flaschenhals darstellt und die Hauptschleife ausbremst. Die Ergebnisse in den Abbildungen 7.3 und 7.3 sind sehr eindeutig, welche die Aussage unterstützen, dass in dem vorherigem Test (Kapitel 7.1.2) das JNI einen Flaschenhals dargestellt hat.

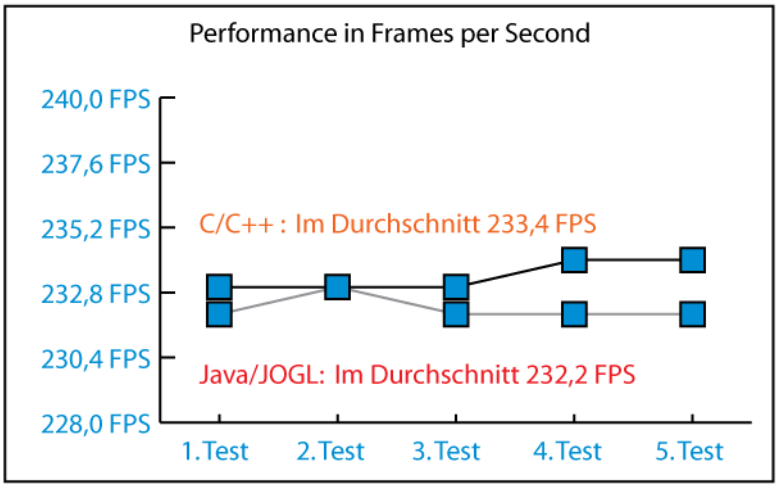


Abbildung 7.3: Ausführungeschwindigkeiten: Performance in fps

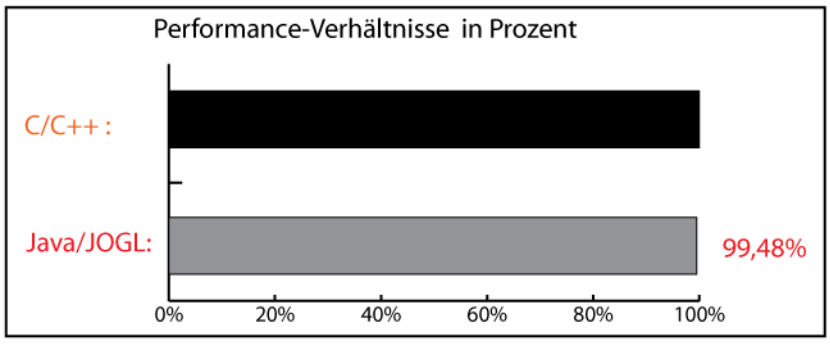


Abbildung 7.4: Ausführungeschwindigkeiten: Performance-Verhältnis

[3]Siehe Kapitel 2.4
[4]Siehe ebenfalls Kapitel 2.4

7.1.4 Testfall3: Java vs. C/C++

Hier wurde die Geschwindigkeit von Java und C/C++ miteinander verglichen. Im Testprogramm wurde ein einfacher Algorithmus mit quadratischer Laufzeit implementiert, um Java und C im zeitlichen Rahmen von ganzen Sekunden rechnen lassen zu können. Das hat den Vorteil, dass die geringen Ausführungszeiten der Messroutinen vernachlässigt werden können. Eingesetzt wurde der Bubblesort-Algorithmus[5], der 30.000 Array-Einträge sortierte. Das Testergebnis in Abbildung 7.5 zeigt das Java etwa um ein Drittel langsamer ist. Warum Java nun soviel langsamer ist, wurde daraufhin genauer untersucht. Es stellte sich schließlich heraus, dass Arrayzugriffe,

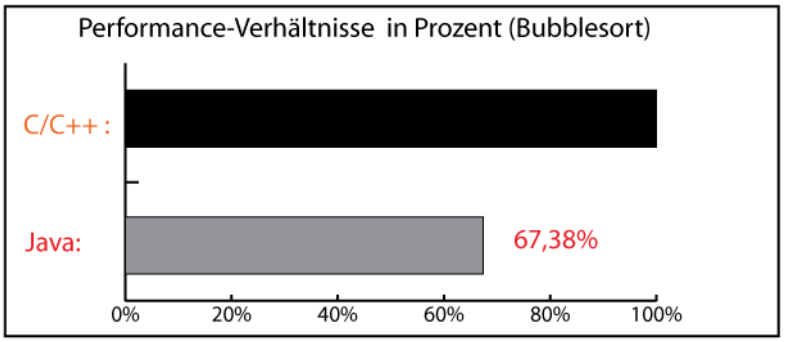


Abbildung 7.5: Bubblesort-Algorithmus: Performance-Verhältnis

welche im Bubblesort zu Genüge getätigt werden, in Java wesentlich langsamer sind als in C/C++. Dies liegt vermutlich daran, dass Java beim Zugriff auf Arrays auch deren Grenzen überprüft. In C/C++ dagegen wird nur mit einem einfachen Pointer gearbeitet, womit es Indizies eines Arrays außerhalb der Grenzen zulässt. Für das Herausfinden dieses Ergebnisses waren 2 vorausgegangene Testfälle verantwortlich. Der eine Testfall beinhaltete eine Schleife in der auf eine `float`-Variable 1 Milliarde mal zugegriffen wird. Der Andere ist auf Arrays spezialisisert und liest/schreibt 5 Millionen Array-Einträge. Wie Abbildung 7.6 verdeutlicht, ist ein Java-Programm

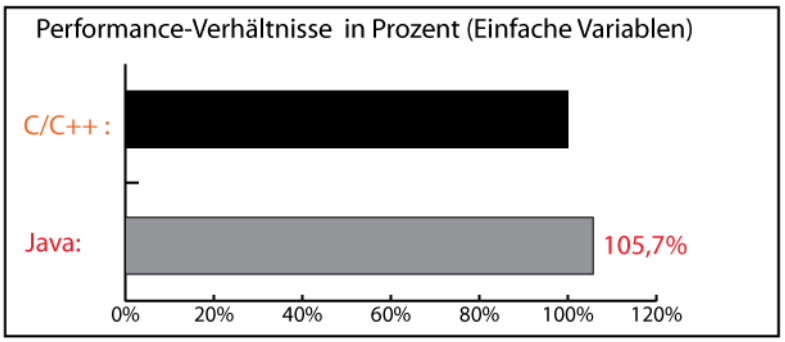


Abbildung 7.6: Einfache float-Variable: Performance-Verhältnis

[5] Einfacher Sortieralgorithmus, der eine Reihe linear angeordneter Elemente nach deren Größe sortiert.

mit einer einfachen Schleife, in der nur eine float-Variable manipuliert wird, signifikant schneller als ein äquivalentes C/C++ Programm. Bei dem gezielten Testen der Performance von Arrayzugriffen büßt aber Java dagegen ganze 37% Performance ein (Vgl. Abbildung 7.7). Hierdurch sollte nun geklärt sein, warum der Bublesort-Algorithmus in etwa den gleichen Anteil an Geschwindigkeitsverlust erlitten hat.

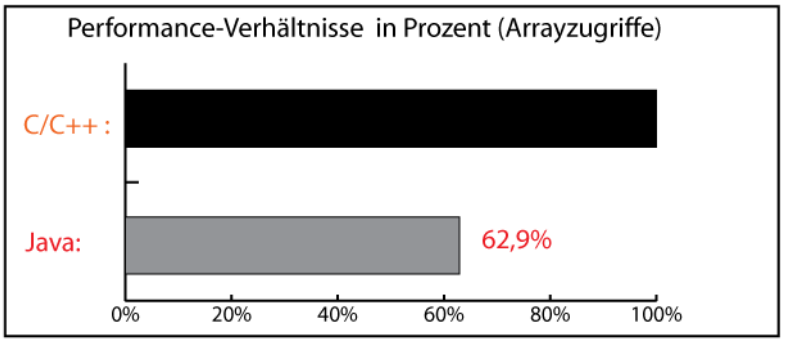


Abbildung 7.7: Arrayzugriff: Performance-Verhältnis

7.1.5 Fazit zu den Tests

Die Testfälle haben gezeigt, dass ein OpenGL-Programm, welches mit JOGL realisiert worden ist, nicht ganz mit einen C/C++ OpenGL-Programm mithalten kann. Zwar stellt das JNI einen großen Flaschenhals dar, lässt aber dennoch eine mehr als nötige Anzahl an fps zu, um sehr flüssige Animationen darstellen zu können. Die Testfälle stellten zudem einen Extremfall dar und werden in dieser Form in richtigen Programmen selten anzutreffen sein. Es sollten sich für ein erfolgreiches Arbeiten mit JOGL folgende Regeln gesetzt werden, wenn viel Performance abverlangt wird:

1. **Regel1: Die Grafikkarte fordern, nicht die CPU**
   Diese Regel gilt nicht nur für Java sonden auch für C/C++-Programme. Wie die Tests gezeigt haben, kann mit vielen kurzlebigen OpenGL-Aufrufen gerade bei Java die CPU schnell in die Knie gezwungen werden. Besteht die Notwendigkeit, viele Objekte zu zeichnen, empfiehlt sich die Verwendung von DisplayLists. Auch sollte jegliche Art von Berechnung, die vorberechenbar ist, nicht in der Hauptschleife ausgeführt werden.

2. **Regel2: So wenig Arrays wie möglich benutzen**
   Wenn möglich, sollte auf eine übermäßige Verwendung von Arrays in Java verzichtet werden. Funktionen, welche mit Arrays arbeiten, sollten stark optimiert sein, so dass sie so wenig Zugriffe wie nur möglich benötigen. Oft gibt es für bestimmte Probleme mehrere Algorithmen zur Lösung. Der Bubblesort-Algorithmus könnte beispielsweise durch den Quicksort-Algorithmus[6] ersetzt werden, welcher in den meisten Fällen weniger Arrayzugriffe benötigt.

[6]Schneller, Rekursiver Sortieralgorithmus, welcher mit Teillisten arbeitet

7.2 Performancetest einer Physiksimulation aus der Praxis

Wie aus dem vorherigen Kapitel 7.1 hervorgeht, hängen einige Faktoren davon ab, wie schnell ein JOGL-Programm im Verhältnis zu einem C/C++ Programm ist. Es wäre voreilig zu behaupten, dass ein JOGL-Programm aufgrund des JNI-Flaschenhalses generell um zwei Drittel langsamer als ein C/C++ Programm ist, da die Testfälle ein Extrem darstellen und außerdem nur ganz gezielt bestimmte Aspekte testen. Aus diesem Grund soll nun eine richtige Physiksimulation aus der Praxis auf deren Performance untersucht werden. Hierfür wird die Planeten-Simulation[7], welche bereits erfolgreich in das neue Framework portiert wurde, eingesetzt. Sie ist sehr gut geeignet für eine Performanceuntersuchung an einem praktischen Programm, da sie viele Berechnung, Arrayzugriffe und JNI-Aufrufe *gemischt* enthält. Weil die Planet-Simulation aber recht wenig Performance verbraucht, wurde sie für diesen Test ein wenig umstrukturiert, so dass die Simulation nun beliebig viele, unabhängig voneinander rotierende Monde um die Erde simulieren kann! Getestet wurde die Performance der Simulation dann mit 10 rotierenden Monden, welche zudem anstatt 2 nun 3 Schwerpunkte besaßen (Abbildung 7.8).

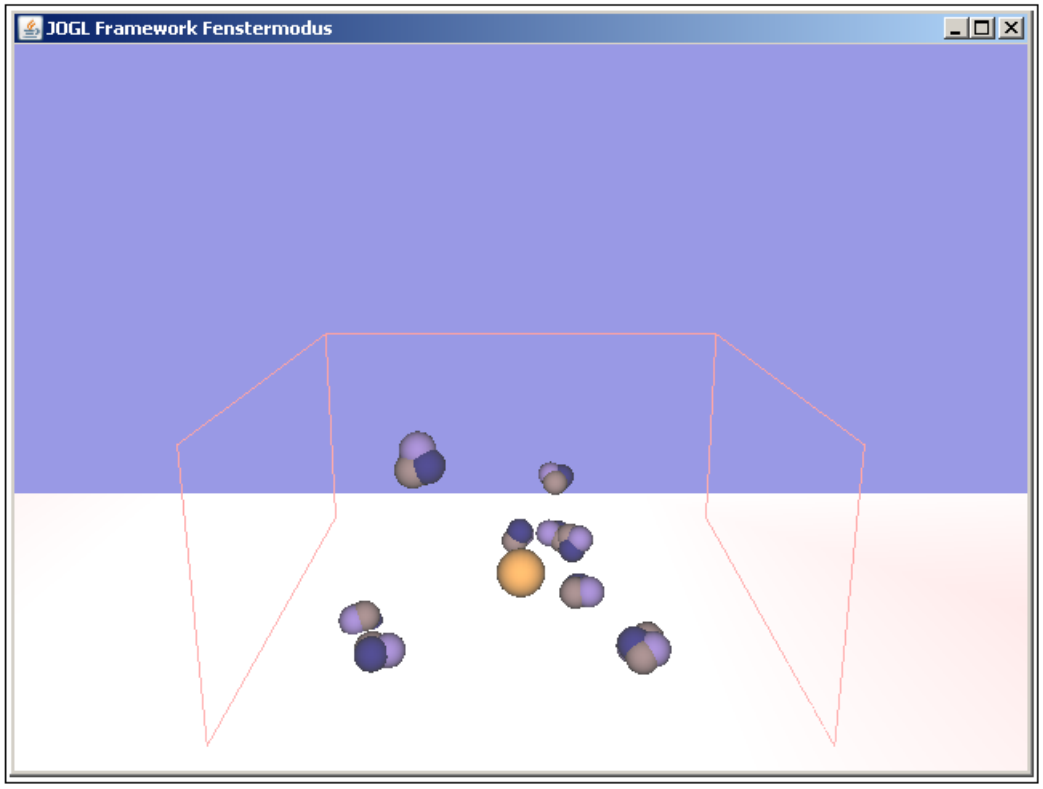


Abbildung 7.8: 10 Monde rotieren um die Erde (JOGL Framework)

[7] Siehe Kapitel 6.1

Die Ergebnisse des Tests vielen dabei recht überraschend zugunsten der C/C++ Planeten-Simulation aus (Abbildung 7.9 und 7.10). Da hier der Performanceunterschied so gravierend ist, wurden beide Simulationen noch einmal genauer untersucht und letzendlich stellte sich heraus, dass hier ein einziger Befehl, welcher aus der GLUT-Bibliothek stammt, für die schlechte Performance von JOGL entscheidend ist. Der Befehl nennt sich `glutSolidSphere(...)` mit dem das Zeichnen von 3D-

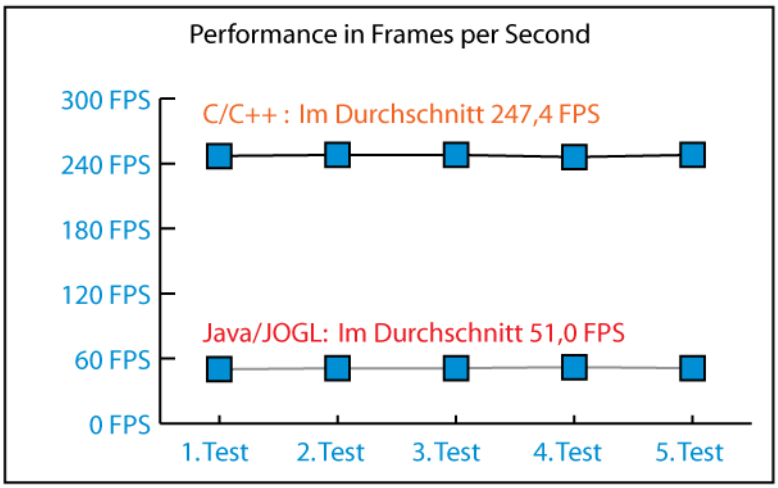


Abbildung 7.9: Planeten-Simulation (10 Monde): Performance in fps

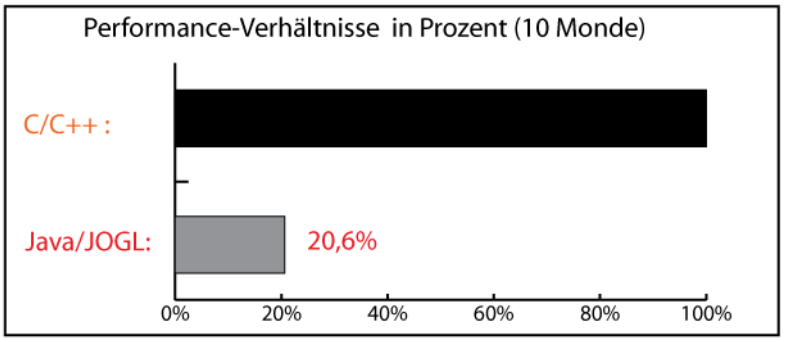


Abbildung 7.10: Planeten-Simulation (10 Monde): Performance-Verhältnis

Kugeln möglich ist. Dieser Befehl ist zwar generell sehr langsam, aber in JOGL weisst dieser eine fast unzumutbare Geschwindigkeit auf. Wird aus dem Test nur diese eine Zeile mit dem Befehl `glutSolidSphere(..)` auskommentiert, entsteht ein ganz anderes Ergebnis wie in Abbildung 7.11 und 7.12 zu sehen ist. Das JOGL-Programm kommt nun deutlich dichter an die Performance des C/C++ Programmes heran. Mit 53% ist das JOGL-Programm nun etwas mehr als halb so schnell. Da in der Planeten-Simulation desweiteren wenig JNI-Aufrufe getätigt werden und überwiegend Arrayzugriffe, stellt dieses Ergebnis einen realistischen Wert dar.

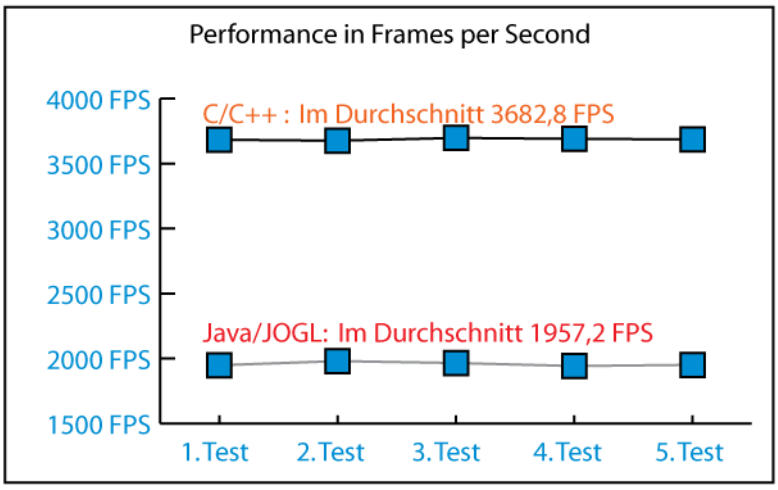


Abbildung 7.11: Planeten-Simulation (10 Monde ohne GLUT): Performance-Verhältnis

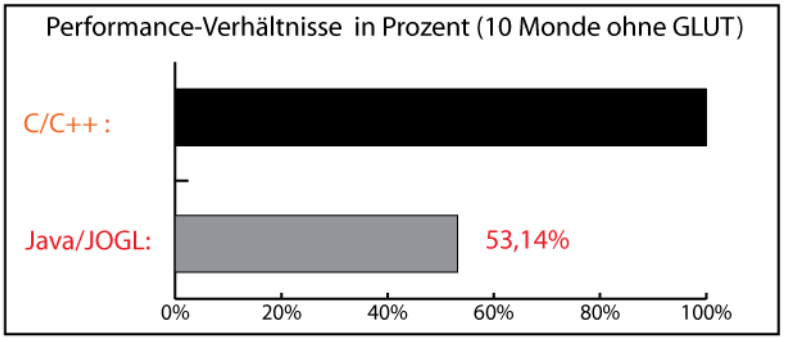


Abbildung 7.12: Planeten-Simulation (10 Monde ohne GLUT): Performance in fps

Ein JOGL Programm, welches gleich viel Zeit für JNI-Aufrufe und Arrayzugriffe verbraucht, sollte theoretisch nach den Tests in Kapitel 7.1 ungefähr halb so schnell sein wie ein C/C++ Programm. Hierzu folgende Rechnung:

Arrayzugriffe in Java: 66% Performance gegenüber einem C/C++ -Programm
JNI-Aufrufe in JAVA: 37% Performance gegenüber einem C/C++ -Programm

Durchschnittliche Performance bei gleichanteiligem **Zeitverbrauch** von Arrayzugriffen und JNI-Aufrufen:

$$\frac{66\% + 37\%}{2} = 51,5\% \tag{7.1}$$

Da ein Arrayzugriff mit Sicherheit schneller als ein JNI-Aufruf ist, in der Planeten-Simulation aber sehr viele davon vorkommen, wird der Zeiverbrauch der Arrayzugriffe wahrscheinlich etwas den Zeitverbrauch für JNI-Aufrufe überwiegen. Das Er-

gebnis von 53% (Abbildung 7.12) würde somit an dieser Stelle gerechtfertigt sein. Konkret bedeutet dies, der Flaschenhals bei der JOGL Planeten-Simulation (*Ohne glutSolidSphere!*) ist ein zusammengesetzter Flaschenhals, wobei dort der größere JNI-Flaschenhals etwas weniger zum Tragen kommt, als der bei den Arrayzugriffen.

7.3 Systemvorraussetzungen des Frameworks

In diesem Kapitel soll herausgefunden werden, auf welchen Rechnern das Framework eingesetzt werden kann und welche Rechner ungeeignet sind. Als Zielrechner des Frameworks dienen vorerst die PC-Systeme des Mathelabors der Fachschule Köln. Hier findet das Wahlpflichtfach „Spiele, Simulation, dynamische Systeme“ statt, für dieses das Framework erstellt wurde. Da für das Framework zusätzlich eine Entwicklungsumgebung benötigt wird, und einige Rechner im Mathelabor weniger performant sind, soll der schnelle Java-Editor *JCreator* eingesetzt werden. Folgende 3 Komponenten der Rechner sind ausschlaggebend für einen reibungslose Arbeit mit dem Framework.

1. **Prozessor**
   Dieser sollte schnell genug sein, um beim Kompilieren von Java-Programmen nicht zu lange warten zu müssen. Auch verlangt OpenGL und die Java Virtual Maschine der CPU einiges ab. Sehr wichtig ist auch, dass die Antwortzeit der GUI des JCreators nicht zu lange dauert, um einen guten Arbeitsfluss und schnelle Interaktionen mit der GUI gewährleisten zu können

2. **Arbeitsspeicher (RAM)**
   Hiervon sollte genug da sein, um vorallem in OpenGL mit genügend Objekten arbeiten zu können. Die Java Virtual Maschine verbraucht nach Tests alleine schon ca. 20 MB RAM, während ein einzelnes Dreieck im Framework rotiert wird. Der Jcreator verbraucht selber dagegen nur 2 MB an Arbeitsspeicher.

3. **Grafikkarte**
   Sie sollte mit aktuellen OpenGL-Implementierungen kompatibel sein, und einen Großteil der OpenGL-Funktionen, wie beispielsweise Licht/Schatten, Alphablending etc., zu unterstützen.

Im Mathelabor gibt es nun 3 verschiedene Typen an Rechnern auf denen das Framework getestet wurde:

1. **TypA**
    - Intel Pentium3 733 MHz
    - 512 MB RAM
    - Ati Grafikkarte mit RV530 pro Chipsatz 512 MB
2. **TypB**
    - Intel Pentium 4 1,8 GHz
    - 512 MB RAM
    - Ati Grafikkarte mit RV530 pro Chipsatz 512 MB
3. **TypC**
    - Intel Celeron 2,67 GHz
    - 512 MB RAM
    - Ati Grafikkarte mit RV530 pro Chipsatz 512 MB

Alle 3 Typen unterscheiden sich hier hauptsächlich nur in der Art und/oder Geschwindigkeit des Prozessors. Die Grafikkarte und der Arbeitsspeicher sind mehr als ausreichend für das Framework. Getestet wurden die PC-Systeme mittels des `HeavyweightListeners` welcher neben dem `FpsCounter`, `JOGLCamera` und `Mouse-/KeyboardHandler` noch einen textuierten, rotierenden 3D-Würfel besaß. Jeder Rechner wurde zudem mit der JDK Version 6 und dem JCreator 4 LE ausgestattet. Es kamen folgende Ergebnisse heraus:

1. **Testergebnis der Rechner vom TypA**

   - **Framework**
     Max. 20 fps Performance im Framework. Es traten sehr starke „Ruckler“ im Bildverlauf auf.
   - **Jcreator/JDK**
     Dieser braucht ca. 10 - 15 Sekunden zum starten. Das Scrollen im Code-Editor ist sehr langsam und das Kompilieren von Java-Programmen dauert oft bis zu 15 Sekunden.

2. **Testergebnis der Rechner vom TypB**

   - **Framework**
     Hier konnten bis zu 640 fps im Framework gemessen werden. Die Darstellung von OpenGL-Grafik war sehr flüssig.
   - **Jcreator/JDK**
     Das Starten des JCreators dauert ca. 3-5 Sekunden. Das Kompilieren geht schnell und ein flüssiges Scrollen im Code-Editor ist möglich.

3. **Testergebnis der Rechner vom TypC**

   - **Framework**
     Auf diesem Typ wurden bis max. 723 fps gemessen, welches mehr als ausreichend ist.
   - **Jcreator/JDK**
     Das Starten des JCreators dauert hier auch ca. 3-5 Sekunden. Das Kompilieren ist sehr schnell und die Antwortzeit der JCreator-GUI hervoragend.

Bei der Installation des Frameworks ist auf keinem der Testrechner ein Fehler aufgetreten. Auch das Framework selber lief sehr stabil und funktionierte einwandfrei auf jedem System. Da vom Typ1 nur noch 3 Rechner im Mathelabor vorhanden sind, und alle anderen, bis auf einen, vom Typ 3 sind, sollte einem Einsatz des JOGL-Frameworks, und somit einer Ersetzung des alten Frameworks, nichts mehr im Wege stehen.

8 Einführung in Real-Time Fluid Dynamics (2D-Rauch)

Da in dieser Diplomarbeit oft der Begriff „Physiksimulation“ gefallen ist, ein konkretes, anschauliches Beispiel hierzu aber noch nicht vorgestellt wurde, soll dies abschließend in diesem Kaptitel passieren. Es gibt viele Arten von Physiksimulationen, welche von einfachen Bewegungen von Objekten, bis hin zu realistischen Bewegungsabläufen von Kleidung oder Stoffen reichen. Ein weiteres recht interessantes Gebiet ist unter anderem auch das Berechnen und Simulieren von sich flüssig bewegenden Strömungen (fluid flows), welche nicht nur im Wasser oder anderen Flüssigkeiten vorkommen, sondern auch in Rauch oder Wolken wiederzufinden sind. Genau zu diesem Thema soll in diesem Kapitel ein Beispiel aufgeführt werden, welches zeigt, wie eine Simulation von diesen flüssigen Strömungen erzeugt werden kann. Als Basis für das Erstellen dieser Simulation dient ein interessanter Artikel[1] von Jos Stam, welcher als Entwickler auch bei dem 3D-Modellierungsprogramm Maya mitwirkt. Er stellt hierbei in seinem Artikel einen Algorithmus zum Thema „Real-Time Fluid Dynamics“ vor, mit dem nicht nur eine bestimmte Art von fluid flows (Hier: 2D-Rauch) simuliert werden könnnen, sondern diese noch dynamisch in Echtzeit (Real-Time) vom Benutzer beeinflussbar sind!

8.1 Entstehung und Schwerpunkte des 2D-Rauchalgorithmus

Der Algorithmus des dynamischen Rauches basiert auf den physikalischen Gleichungen für Flüssigkeitsstömungen (fluid flows), welche auch als die Navier-Stokes Gleichungen bekannt sind. Mit diesen können die meisten flüssigen Strombewegungen, welche in der Natur vorkommen berechnet werden. Nach Jos Stam Aussage ist es aber sehr schwierig, solche Gleichungen aufzulösen, wenn das Primärziel ist, eine sehr exakte physikalische Simulation zu erhalten. Die Komplexität dieser Berechnungen würde schnell viele Rechner überfordern. Somit wurde der Algorithmus mehr mit Fokus auf Stabilität und Geschwindigkeit entwickelt, anstatt auf eine exakte naturgetreue Darstellung. Hierdurch kann dieser Problemlos in Computerspielen eingesetzt werden und ist sogar auf kleinen Endgeräten wie beispielsweise einem Palm Top einsetzbar. Mit Stabilitität des Algorithmus ist gemeint, das sich der Rauch in einem

[1] Siehe [Sta07]

vordefinierten Rahmen bewegt und niemals aus dessen Grenzen ausbrechen kann oder sogar explodiert, wie es in 100% physikalisch korrekten Simulationen der Fall sein kann.

8.2 Funktionsweise des 2D-Rauchalgorithmus

Da dieser Algorithmus recht komplexe Berechnungen beinhaltet, würde eine genaue Erklärung den Rahmen dieser Einführung sprengen. Im Folgenden wird somit nur grob beschrieben, was der Algorithmus macht. Dazu werden erst einige Begrifflichkeiten der Physik geklärt:

- **Geschwindigkeit**
  Die Geschwindigkeit ist eine zurückgelegte Wegstrecke pro Zeit. Sie kann als Geschwindigkeitsvektor dargestellt werden, wobei die Länge des Vektors, den Geschwindigkeitsbetrag darstellt und die Richtung und Orientierung der Geschwindigkeit festgelegt werden.

- **Masse**
  Die Masse ist eine Grundgröße der Physik. Ihre Standardeinheit ist das Kilogramm. Objekte welche verschiedene Massen besizen, reagieren beispielsweise anders auf die Erdanziehungskraft.

- **Dichte**
  Die Dichte ist eine Physikalische Größe, die eine Massenverteilung beschreibt. Sie ist als die Menge einer Masse pro Volumeneinheit definiert.

- **Diffusion**
  Unter Diffusion im engeren Sinne versteht man den Ausgleich von Konzentrationsunterschieden. Im Bezug auf den zu entwickelnden Algorithmus bedeutet hier Diffusion ein Konzentrationsausgleich von unterschiedlichen Dichten, bei denen sich deren Massen gleichverteilen.

- **Kraft**
  Eine Kraft wird in der Physik als „Masse multipliziert mit einer Beschleunigung“ definiert.

Um nun einen natürlich aussehenden, 2-dimensionalen Rauch zu realisieren, bedient sich der Rauchalgorithmus den Navier-Stoke Gleichungen:

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + v\nabla^2\boldsymbol{u} + \boldsymbol{f} \tag{8.1}$$

$$\frac{\partial p}{\partial t} = -(\boldsymbol{u} \cdot \nabla)p + k\nabla^2 p + S \tag{8.2}$$

Die Gleichung 8.1 befindet sich dabei in einer kompakten Vektornotation, welche ein Vektorfeld aus einzelnen Geschwindigkeitsvektoren (u) beschreibt. Das Vektorfeld

kann dabei durch einen Kraft-Vektor (f) beeinflusst werden. Mit Hilfe dieser Formel ist nun möglich, für jeden infinitisemal [2] kleinen, späteren Zeitpunkt (t), die Veränderung der Vektoren im Feld herauszufinden (Abbildung 8.1). Die Idee zur Darstellung

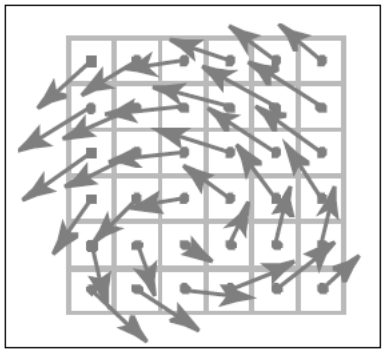


Abbildung 8.1: Geschwindigkeits-Vektorfeld [Sta07], Seite 7

von 2D-Rauch ist nun, diesen in kleinst mögliche Flächeneinheiten, in diesem Fall in Pixel des Bildschirms, einzuteilen. Jede dieser Flächeneinheit ist dabei mit einem dazugehörigen Vektor des Geschwindigkeits-Vektorfeldes assoziiert. Dabei besitzt jede dieser Flächeneinheiten eine bestimmte Dichte, also sozusagen eine Masse pro Pixel, welche zwischen dem Wert 0 und 1 definiert ist. Mit Hilfe von Gleichung 8.2 kann nun ein, über einen bestimmten Zeitraum (t) stattfindender Konzentrationsaustausch der Dichten (ρ) berechnet werden. Diese Diffusion, abhängig vom Geschwindigkeits-Vektorfeld, stellt schließlich eine Illusion von strömenden 2D-Rauch dar (8.2). Um

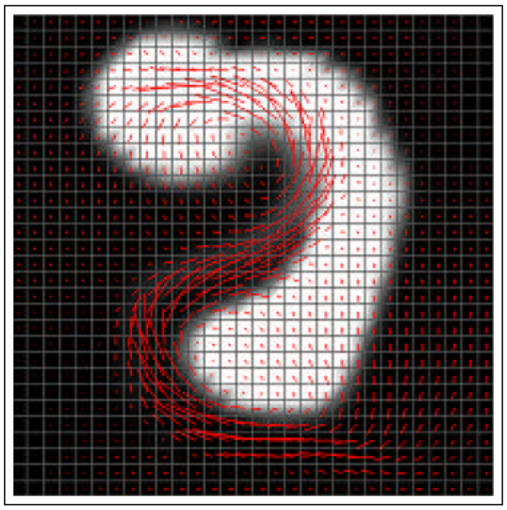


Abbildung 8.2: Dichteverteilung [Sta07], Seite 6

die Diffusion grafisch erkennen zu können, wird dabei die Dichte eines Pixels durch eine Farbe (Werte von 0 bis 1) dargestellt. Eine hohe Dichte bekommt einen hohen Farbanteil an weisser Farbe zugeordnet, eine geringe Dichte nur einen geringen

[2] infinitisemal = unendlich

Anteil. Abbildung 8.3 zeigt hierzu einige Beispiel-Darstellungen des 2D-Rauches:

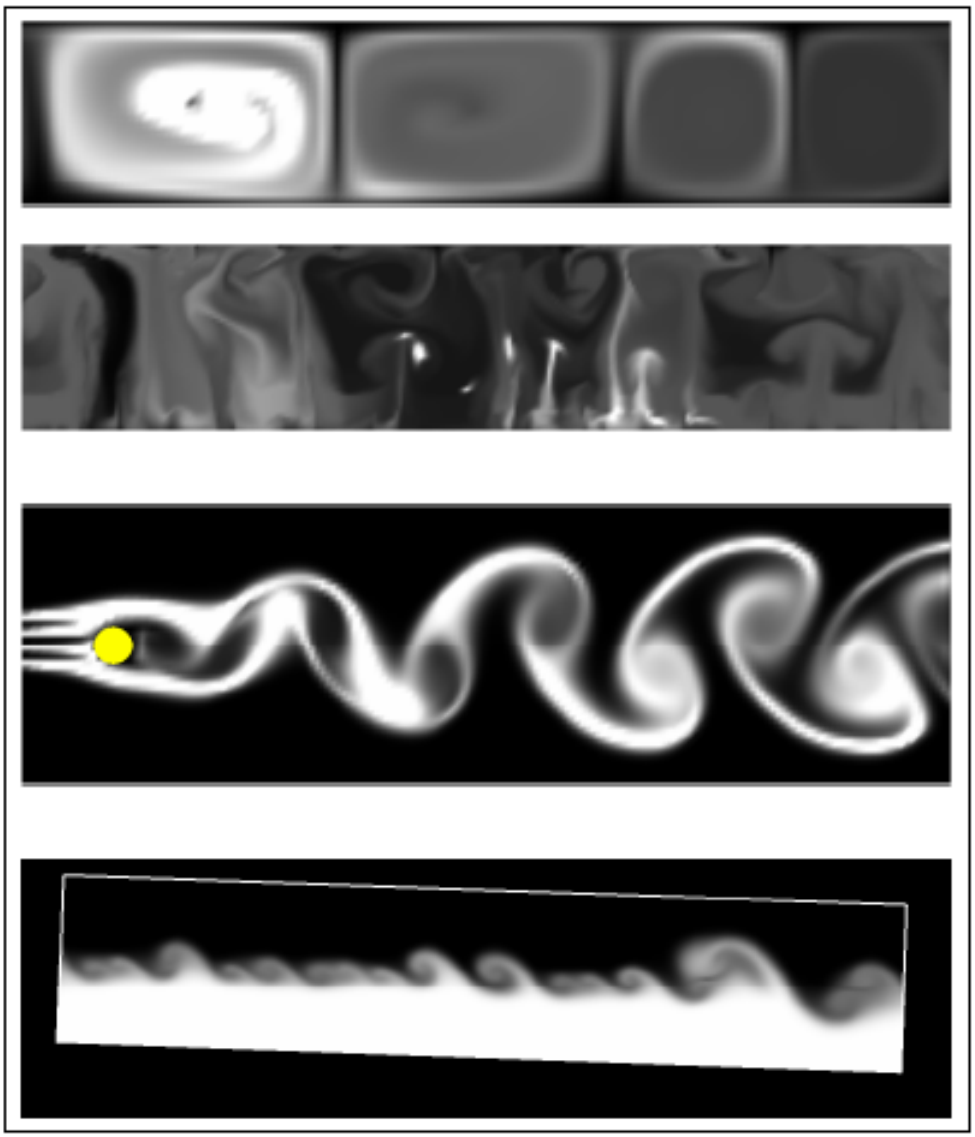


Abbildung 8.3: Beispiel-Darstellungen des 2D-Rauches [Sta07], Seite 17

8.3 Implementierung und Anwendungsbeispiel des 2D-Rauchalgorithmus

Implementierung

Da der 2D-Algorithmus zu dem Artikel von Jos Stam bereits als C-Programm mitgeliefert wurde, musste dieser nur in das neue Framework portiert werden. Weil das C-Programm ebenfalls OpenGL zu Grafikausgabe verwendete, konnte die Portierung mit relativ wenig Schwierigkeiten realisisert werden. Ein wenig umständlich hierbei war nur, dass das C-Programm durch die Verwendung von einigen Makros sehr klein gehalten worden ist und einen Umfang von ungefähr 100 Zeilen aufwies. Diese mussten für die Portierung nach Java alle manuell aufgelöst werden, wodurch sich der Programmcode fast verdoppelte. Auch musste der vorimplementierte OpenGL-Code des für die Simulation verwendeten `LightweightListeners` umgeschrieben werden, weil für den Rauch nur eine reine 2D-Darstellung benötigt wurde.[3]

Anwendungsbeispiel

Zum Abschluss dieses Kapitels werden noch ein paar Screenshots gezeigt werden, welche die 2D-Simulation auf dem neuen Framework in Aktion zeigt. Anfangs ist der Bildschirm der Rauchsimulation schwarz und zeigt keinerlei Aktivität. Der Benutzer muss nun mit der rechten Maustaste an bestimmten Stellen eine Rauchdichte definieren. Diese kann er wie in einem einfachen Zeichenprogramm dort hereinmalen (Abbildung 8.4). Wenn nun die Maus bewegt wird, während die linke Maustaste her-

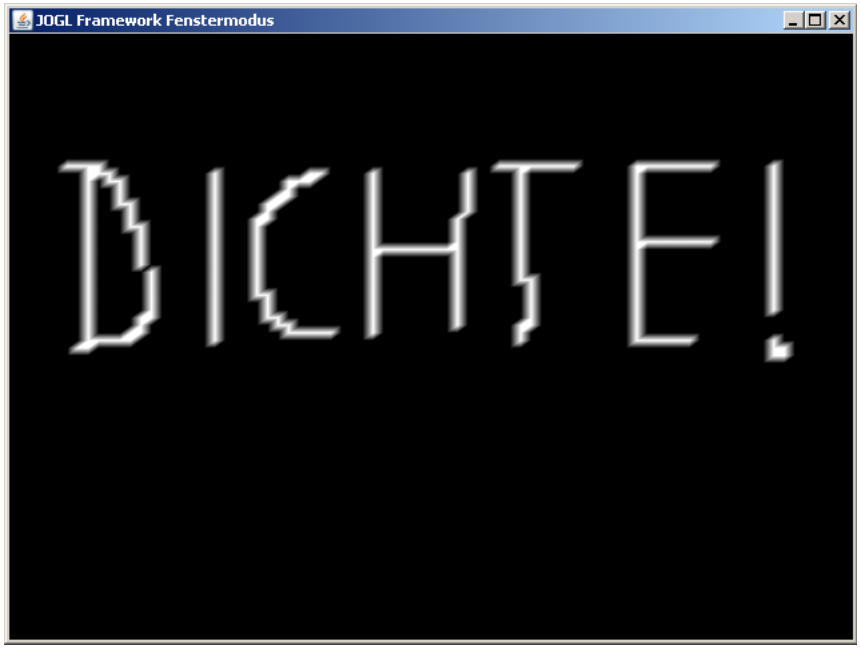


Abbildung 8.4: Setzen der Rauchdichte

[3] 2D-Darstellung von Grafik in OpenGL ist mit der GLU-Funktion gluOrtho2D(...) möglich

untergedrückt ist, wird automatisch eine Geschwindigkeit herbeigeführt (Abbildung 8.5). Genauer gesagt ist die Geschwindigkeit eine Beschleunigung, mit der eine Masse

Abbildung 8.5: Kraft herbeiführen, welche auf den Rauch einwirkt

bewegt wird. Es handelt sich also dort um eine Kraft (= Masse mal Beschleunigung), welche auf den Rauch einwirkt. Mit der Taste `v` kann zwischen verschiedenen Ansichten gewechselt werden. Neben der bisherigen Darstellung des Rauches (Seiner Dichte) ist auch eine Darstellung des aktuellen Geschwindigkeits-Vektorfeldes möglich (Abbildung 8.6).

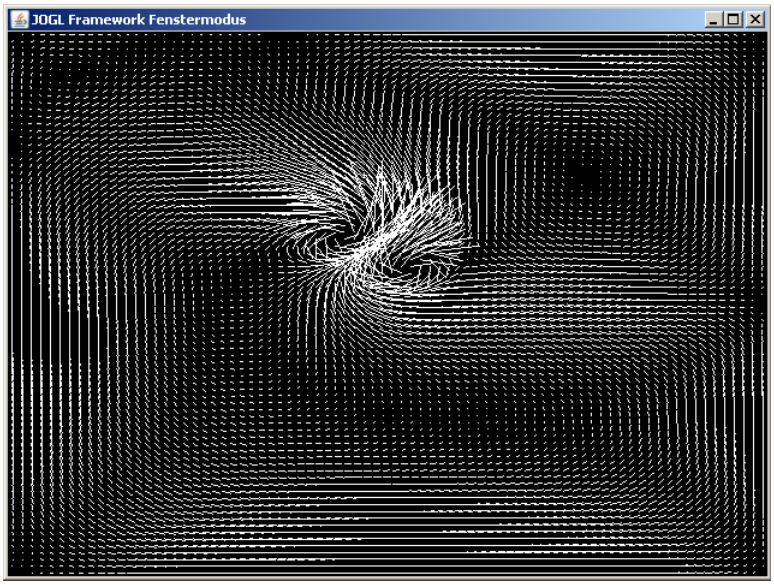


Abbildung 8.6: Umschalten der Ansicht: Geschwindigkeits-Vektorfeld

9 Fazit

Das Ziel, ein JOGL-basiertes Framework zu entwickeln, welches dem späteren Benutzer die Einarbeitungszeit in OpenGL mit JOGL erspart, konnte nach einigen Anlaufschwierigkeiten komplett erreicht werden. Der Umfang des Projektes ist dabei wesentlich größer geworden, als dies vor Beginn der Diplomarbeit geplant gewesen ist. Vor allem eine korrekte Handhabung der Anzeigekonfiguration im Anzeigemenü hat einen großen Teil der zu Verfügung gestanden Entwicklungszeit gekostet. Auch die Erstellung der Performancetests hatte einige Komplikationen mit sich gebracht, weil nicht immer genaue Messergebnisse erlangt werden konnten. Im ganzen gesehen, hat sich aber der Entwicklungsaufwand sehr gelohnt, da es nun möglich ist, im Wahlpflichtfach „Spiele, Simulation, dynamische Systeme“ die physikalischen Simulation mittels Java zu realisieren, ohne auf C/C++-Kenntnisse angewiesen sein zu müssen. Zwar hat sich herausgestellt, dass das neue JOGL-basierte Framework in etwa nur halb so performant ist, wie das bisherige Framework, für einen praktischen Einsatz ist es aber vollkommen ausreichend. Außerdem besitzt das neue Framework eine übersichtliche, objektorientierte Programmstruktur, und die Möglichkeit der Verwendung von umfangreichen neuen Funktionen, wie beispielsweise einer dynamaischen 3D-Kamerasteuerung, lassen die Vorteile bei weitem überwiegen. Diese Diplomarbeit hat gezeigt, dass Java längst nicht mehr nur für Server-Programme oder einfache GUI-Anwendungen geeignet ist. Durch die OpenGL-Bindings ist Java nun in der Lage, sich auch in der Grafik-/Spieleprogrammierung behaupten zu können. Auch wird festgestellt, dass „Game Physics“, welche auf dem neuen Framework simuliert werden sollen, mittlerweile so wichtig und unentbehrlich sind, dass einige neue PC-Systeme (Bsp.: Firma Alienware) sogar standardmäßig mit einem zusätzlichen Physik-Chip[1] ausgestattet werden. In naher Zukunft wird es also neben den gewohnten Hauptkomponenten wie CPU, Grafikkarte und Soundkarte noch eine Physikkarte geben, welche ausschließlich für Physikberechnungen zuständig ist!

[1] Der neue AGEIA PhysX Chip = erster Physik-Prozessor der Welt! <http://www.ageia.com>

Literaturverzeichnis

[Chr05] Christian, Ullenbum: *Java ist auch eine Insel*. 4. Auflage. Galileo Press, 2005

[Com07a] Community: *Delphy OpenGL Wiki*. Version: 2007. <http://wiki.delphigl.com/index.php>, Abruf: 23.02.2007

[Com07b] Community: *Neon Helium Productions*. Version: 2007. <http://nehe.gamedev.net/>, Abruf: 23.02.2007

[Com07c] Community: *Wikipedia -> JOGL*. Version: 2007. <http://en.wikipedia.org/wiki/JOGL>, Abruf: 23.02.2007

[Com07d] Community: *Wikipedia -> OpenGL*. Version: 2007. <http://de.wikipedia.org/wiki/OpenGL>, Abruf: 23.02.2007

[Dev07a] Developer, JOGL: *JOGL API Dokumentation*. Version: 2007. <http://download.java.net/media/jogl/builds/archive/jsr-231-1.1.0-rc2/jogl-1.1.0-rc2-docs.zip>, Abruf: 23.02.2007

[Dev07b] Developer, JOGL: *JOGL Official Demo Source Code*. Version: 2007. <http://download.java.net/media/jogl/builds/archive/jsr-231-1.1.0-rc2/jogl-demos-src.zip>, Abruf: 23.02.2007

[Dev07c] Developer, JOGL: *JOGL Official Users Guide*. Version: 2007. <https://jogl.dev.java.net/unbranded-source/browse/*checkout*/jogl/doc/userguide/index.html>, Abruf: 23.02.2007

[Dev07d] Developer, LWJGL: *LWJGL Homepage*. Version: 2007. <http://lwjgl.org/>, Abruf: 23.02.2007

[Dev07e] Developer, Mesa3D: *Mesa3D Homepage*. Version: 2007. <http://www.mesa3d.org/>, Abruf: 23.02.2007

[Gen07] Gentleware: *Poseidon for UML Homepage* . Version: 2007. <http://www.gentleware.com/products.html>, Abruf: 23.02.2007

[Hel00] Helmut, Balzert: *Lehrbuch der Softwaretechnik*. 2. Auflage. Spektrum, 2000

[Mic07a] Microsystems, Sun: *Java 6 SE - Features and Enhancements*. Version: 2007. <http://java.sun.com/javase/6/webnotes/features.html>, Abruf: 23.02.2007

[Mic07b] Microsystems, Sun: *Java Language Specification - Third Edition*. Version: 2007. <http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf>, Abruf: 13.02.2007

[Mic07c] Microsystems, Sun: *Java Tutorials - Full-Screen Exclusive Mode*. Version: 2007. <http://java.sun.com/docs/books/tutorial/extra/fullscreen/exclusivemode.html>, Abruf: 23.02.2007

[Mic07d] Microsystems, Sun: *Java3D API 1.5*. Version: 2007. <http://java.sun.com/products/java-media/3D/download.html>, Abruf: 23.02.2007

[Mic07e] Microsystems, Sun: *Java3D API Tutorial*. Version: 2007. <http://java.sun.com/developer/onlineTraining/java3d/>, Abruf: 23.02.2007

[Mic07f] Microsystems, Sun: *Java3D Release Notes 1.5*. Version: 2007. <https://j3d-core.dev.java.net/j3d1_5_0/RELEASE-NOTES.html#JoglPipeline>, Abruf: 23.02.2007

[ND97] Neider, Jackie ; Davis, Tom: *OpenGL Redbook 1.1*. Version: 1997. <http://www.gamedev.net/download/redbook.pdf>, Abruf: 23.02.2007

[PR07] Petersen, Daniel ; Russell, Kenneth: *JOGL Javaone 2004 Präsentation*. Version: 2007. <https://jogl.dev.java.net/2125.pdf>, Abruf: 23.02.2007

[RKZ07] Russell, Kenneth ; Kline, Christopher ; Ziemski, Gerard: *JOGL Javaone 2003 Präsentation*. Version: 2007. <https://jogl.dev.java.net/2125.pdf>, Abruf: 23.02.2007

[Ron98] Ron, Fosner: *OpenGL Programming for Windows95 and Windows NT*. 6. Auflage. Addison Wesley, 1998

[Sta07] Stam, Jos: *Real-Time Fluid Dynamics for Games*. Version: 2007. <http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf>, Abruf: 23.02.2007

A Anhang

A.1 Zusammenfassung der Features und Handhabung des Frameworks

Zu besseren Übersicht wird in diesem Kapitel noch einmal kompakt zusammengefasst, was das Framework kann und wie einzelne Funktionalitäten aus Benutzersicht handzuhaben sind.

Das Anzeigemenü
Dieses wird automatisch beim Starten des Frameworks angezeigt. Hier kann eine gewünschte Auflösung im Vollbildmodus oder Fenstermodus gewählt werden. Alle Auflösungen sind kompatibel mit dem aktuell verwendeten PC-System. Wird das Anzeigemenü nicht gewünscht, muss die Klasse `RunFrameWork` in Zeile 46 folgendermaßen modifiziert werden:

```
//Diese Zeile muss geloescht werden...
DispMenu menu = new DispMenu("JOGL Framework V.3.6",runFrame);

//...und durch diese ersetzt werden.
//(Breite, Hoehe, Farbtiefe, Hertz und Vollbild ja/nein)
runFrame.runme(new DisplayMode(1024,768,32,85), true);
```

Features und Handhabung des HeavyweightListeners:

1. **Frei bewegliche 3D-Kamera (JOGLCamera)**
   Die 3D-Kamera lässt eine freie Bewegung im Raum zu. Sie ist fest implementiert und kann über folgende Keyboard Tasten gesteuert werden:

   - w = Kamera vorwärts in Blickrichtung bewegen
   - s = Kamera rückwärts entgegen der Blickrichtung bewegen
   - a = Kamera seitwärts links relativ zur Blickrichtung bewegen
   - d = Kamera seitwärts rechts relativ zur Blickrichtung bewegen
   - e = Kamera aufwärts schweben lassen (Y-Achse)
   - q = Kamera abwärts schweben lassen (Y-Achse)

   Die Blickrichtung dagegen ist mit der Maus beeinflussbar. Um sich umschauen zu können, muss die *rechte* Maustaste gedrückt gehalten werden, während

die Maus bewegt wird. Die Maus kann dabei unendlich weit nach links oder rechts bewegt werden. Dies funktioniert sowohl im Vollbildmodus, als auch im Fenstermodus!

2. **Informationsanzeige**
   Diese Anzeige zeigt Informationen bezüglich der aktuellen fps an und beschreibt die Tasten, welche zum Steuern der Kamera nötig sind.
   - F1 = Aus-/Anschalten der Informationsanzeige

3. **Anzeige eines grafischen 3D-Ursprungs**
   Dieser wird standardmäßig angezeigt. Er kann in der `display(...)`-Methode des HeavyweightListeners in Zeile 184 (`origin.drawOrigin(gl)`) aber auskommentiert werden, wenn er nicht benötigt wird!

4. **Textausgabe auf dem Zeichenbereich (JOGLText)**
   Es kann beliebiger Text auf dem Zeichenbereich, (sehr nützlich zum Debuggen) mit `joglText.drawText(...)` ausgegeben werden.

5. **KeyboardHandler**
   Mit ihm können bequem die Keyboardtasten abgefragt werden. Die Taste „Escape“ ist bereits reserviert, und fährt das Framework herunter. Eine Tastenabfrage kann an jeder Stelle mit `keyboard.checkIsKeyPressed(...)` abgefragt werden. Ein manuelles Loslassen einer Taste (Verhindert Mehrfachauswertung!) ist dagegen mit `keyboard.setManualRelease(...)` möglich.

6. **MouseHandler**
   Der HeavyweightListener besitzt 2 MouseHandler. Einer ist komplett zuständig für die Kamerasteuerung und braucht nicht weiter vom Benutzer beachtet werden. Er ist in der Lage, relative Mauskoordinaten abzufragen. Mit dem anderen MouseHandler `mouseNorm` können Maustasten und Mauspositionen an beliebiger Stelle abgefragt werden.
   - Abfrage der linken Maustasten: mouseNorm.checkIsLeftButtonDown()
   - Abfrage der mittleren Maustasten: mouseNorm.checkIsMiddleButtonDown()
   - Abfrage der rechten Maustasten: mouseNorm.checkIsRightButtonDown()
   - Abfrage der Maus-Position: mouseNorm.getWindowMouseXY()

7. **FPSCounter**
   Die fps werden automatisch in der `display(...)`-Methode des HeavyweightListeners gemessen. Das Messen geschieht durch `fpsCounter.checkFps()` in Zeile 134. Die aktuellen fps können zu jederzeit an einer beliebigen Stelle mit `fpsCounter.getActFps()` erhalten werden.

8. **Vorinitialisierung von OpenGL**
   OpenGL ist komplett mit gängigen Standardeinstellungen voreingestellt. In der Hauptschleife `display(...)` des HeavyweightListeners können direkt OpenGL-Zeichenbefehle verwendet werden. Auch ist bereits die Verwendung von GLU und GLUT über die beiden vorimplementierten Objekte `glu` und `glut` möglich!

Features und Handhabung des LightweightListener:

1. **KeyboardHandler**
   Ein Abfrage von Keyboardtasten ist hiermit möglich. Dies funktioniert genauso wie im HeavyweightListener.

2. **Vorinitialisierung von OpenGL**
   OpenGL ist komplett mit gängigen Standardeinstellungen voreingestellt. Dies sind ebenfalls dieselben Einstellungen wie in dem HeavyweightListener.

Die Syntax und genaue Parametrisierung aller Befehle kann hierbei in der ausführlichen Framework-Dokumentation (Java-Doc Format) auf der beiliegenden CD dieser Diplomarbeit nachgeschaut werden. Sie befindet sich im Ordner „*Framework API Dokumentation*“. Auch ist der Quellcode des Heavyweight- und LightweightListeners sehr hilfreich, welcher sich im Anhang in Kapitel A.3 befindet!

A.2 Installation des Frameworks auf einem PC-System

A.2.1 Generelle Vorbereitungen Windows/Linux

Um das Framework auf einem PC-System zu installieren, müssen generell 2 Vorbereitungen getroffen werden:

1. **Installieren des JDK 6.0 (Mustang)**
   Es empfiehlt sich die neuste Java Version 6.0 zu installieren, welche viele Fehler aus alten Versionen bereinigt hat. Da das Framework hiermit entwickelt wurde, sollte somit keine andere Version des JDK's installiert sein, um sicherzugehen, das alles einwandfrei funktionieren wird. Das JDK 6.0 sowie die dazugehörige Dokumentation können unter *http://java.sun.com/javase/downloads/index.jsp* heruntergeladen werden.

2. **Installieren der neusten Grafikkartentreiber**
   Um zu gewährleisten, dass die aktuellste OpenGL-Implementierung verwendet wird, sollten die neusten Grafikkartetreiber geladen werden. Da aktuelle Gafikkarten heutzutage hauptsächlich nur noch von ATI oder NVIDIA produziert werden, sind meist dafür folgende Links nützlich:
   - *http://ati.de/support/driver.html*
   - *http://www.nvidia.de/page/drivers.html*

A.2.2 Installation des Frameworks/JCreators unter Windows

Da das Framework hauptsächlich Verwendung unter einem MS Windows finden wird, ist dieses auf eine Zusammenarbeit mit dem JCreator zugeschnitten. Der JCreator ist nur für Windows geeignet und kann nicht unter anderen Betriebsystemen wie beispielsweise Linux eingesetzt werden. Um eine Physiksimulation im JCreator mit Hilfe des Frameworks zu programmieren, muss zuvor das Framework und auch JOGL im System installiert werden.

Installation von JOGL

Die aktuellste JOGL Version 1.1 (Version vom: 14. Februar 2007) für Windows kann unter folgender Internetadresse bezogen werden:

http://download.java.net/media/jogl/builds/archive/jsr-231-1.1.0-rc3/jogl-1.1.0-rc3-windows-i586.zip

Diese ZIP-Datei beinhaltet unter anderem den Ordner `lib`, welcher in den Ordner `C:\JOGL` des PC-Systems kopiert werden muss. Ein anderer Pfad oder Name sollte vermieden werden, da die JCreator-Projekte hierauf bereits eingestellt sind! (Abbildung A.1). Im nächsten Schritt müssen 2 Umgebungsvariablen unter Windows gesetzt/verändert werden. An die Umgebungsvariablen gelangt man unter Windows2000/WindowsXP über folgenden Aufruf:

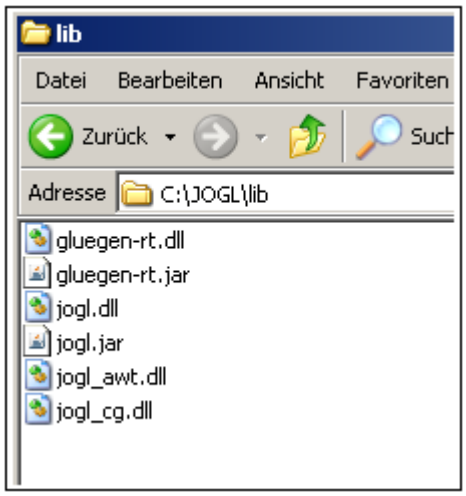


Abbildung A.1: lib-Ordner aus ZIP-Datei entpackt

```
Rechte Maustaste auf den Arbeitplatz->Eigenschaften->
Erweitert->Umgebungsvariablen
```

Unter Windows98 muss die Autoexec.bat im Stammverzeichnis des Systems editiert werden. mit `SET UMGEBUNGSVARIABLENNAME = WERT` wird hier eine Umgebungsvariable gesetzt/verändert. Da die Rechner auf denen JOGL installiert werden soll, fast alle Windows2000/WindowsXP besitzen, wird sich hier desweiteren auch nur auf diese Betriebssysteme bezogen. Zuerst muss die CLASSPATH-Variable verändert werden. Sollte sie noch nicht vorhanden sein, muss diese neu angelegt werden. Sie bekommt 2 JAR-Archive aus `C:\JOGL\lib` zugewiesen, damit diese später von Java gefunden werden können (Abbildung A.2). Als nächstes muss noch der Ordner

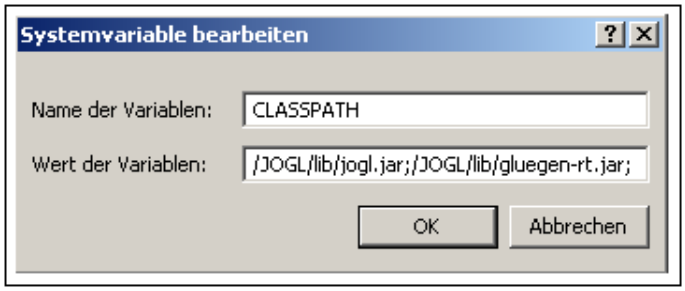


Abbildung A.2: JOGL-JAR-Archive in den CLASSPATH

`C:\JOGL\lib` an die Path-Umgebungsvariable drangehängt werden, damit später die nativen Bibliotheken von JOGL global zur Verfügung stehen (Abbildung A.3).

Abbildung A.3: Path-Variable erweitern

Installation des Frameworks

Das Framework selber besteht nur aus einer einzelnen JAR-Datei, welche sich JOGL-Framework.jar nennt. Sie befindet sich auf der beigefügten CD der Diplomarbeit im Ordner „*Framework Installation\JOGL Framework (JAR)*“[1] und muss nach `C:\JOGL` kopiert werden. Desweiteren muss die JAR-Datei nur noch in den CLASSPATH eingetragen werden (Abbildung A.4).

[1] Es gibt dort 2 jar-Versionen auf der CD. Eine für das JDK 5 und eine hier für das JDK 6

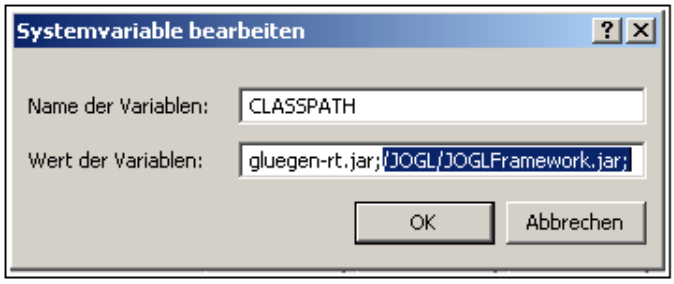


Abbildung A.4: Framework in den CLASSPATH

Installation des JCreators

Die kostenlose Version des JCreators in seiner aktuellen Version 4 LE kann unter folgender Adresse bezogen werden:

http://www.jcreator.com/download.htm

Wenn zuvor das JDK bereits installiert worden ist, muss bei der Installation nichts weiter beachtet werden. Beim ersten Programmstart werden darauf Standardeinstellungen bezüglich des JDK's und eventueller JDK API-Dokumentation abgefragt. Die Standardeinstellungen sollten alle so übernommen werden. Als nächstes muss der JCreator mit dem Framework und mit JOGL verknüpft werden. Auch eine wichtige Java Laufzeit-Einstellung[2] muss hier getätigt werden, damit JOGL auf dem System 100% funktioniert. Um diese Einstellungen nicht alle von Hand machen zu müssen, stehen extra 2 XML-Dateien auf der CD der Diplomarbeit bereit. Sie befinden sich dort im Ordner „*Framework Installation\JCreator 4 LE*“ und müssen nach `C:\Programme\Xinox Software\JCreatorV4LE\Options` kopiert werden. Der JCreator ist hiernach komplett konfiguriert.

Verwenden des Frameworks mit dem JCreator

Um nun eine Physiksimulation mit dem JCreator erstellen zu können, stehen 2 JCreator-Projekte zur Verfügung. Sie sind auf der CD der Diplomarbeit im Ordner „*Framework JCreator Projekt Dateien*“ zu finden.

1. **Projekt: JOGL Framework Lightweight Edition**
   Dieses Projekt beinhaltet den LightweightListener, und stellt nur wenig vorimplementierte Funktionalitäten des Frameworks bereit.

2. **Projekt: JOGL Framework Heavyweight Edition**
   Dieses Projekt beinhaltet den HeavyweightListener, mit dem die Funktionalitäten des Frameworks komplett ausgeschöpft werden können.

[2] -Dsun.java2d.noddraw=true siehe in [Dev07c]

Die Projekte können an eine beliebige Stelle auf das PC-System kopiert werden und sind direkt mithilfe des JCreators ausführbar.

A.2.3 Installation des Frameworks unter Linux

Um das Framework unter Linux verwenden zu können, gibt es einen Installationsweg, der recht einfach zu gehen ist, ohne große Konfigurationen vornehmen zu müssen. Dieser wird nun knapp beschrieben werden. Zuerst muss die JOGL-Version für Linux heruntergeladen werden:

http://download.java.net/media/jogl/builds/archive/jsr-231-1.1.0-rc3/jogl-1.1.0-rc3-linux-i586.zip

In diesem Beispiel wird der *Inhalt* des in der ZIP-Datei befindlichen Ordners `lib` in einen beliebigen **Unter**ordner kopiert. Er wird hier mit `jogl` benannt. Zusätzlich muss in diesen Ordner die `JOGLFramework.jar`[3] kopiert werden. In den Überordner des Ordners `jogl` muss dann der `HeavyweighListener`[4]/`LightweightListener`[5] und die Start-Routine `RunFrameWork`[6] kopiert werden.[7] Per Kommandozeile auf Ebene des Überordners können diese dann folgendermaßen kompiliert und gestartet werden:[8].

Kompilieren der Listener und RunFrameWork:

```
javac␣-cp␣./jogl/jogl.jar:./jogl/JOGLFramework.jar:.␣-d␣.␣*.java
```

Starten des Frameworks/Physiksimulation (RunFrameWork):

```
java␣-Dsun.java2d.noddraw=true␣-Djava.library.path=./jogl/␣-cp
␣./jogl/jogl.jar:./jogl/JOGLFramework.jar:./jogl/gluegen-rt.jar:.
␣RunFrameWork
```

[3] Auf der CD in: „*Framework Installation\JOGL Framework (JAR)*“
[4] Auf der CD in: „*Framework JCreator Projekt Dateien\JOGL Framework Heavyweight Edition*“
[5] Auf der CD in: „*Framework JCreator Projekt Dateien\JOGL Framework Lightweight Edition*“
[6] Auf der CD in: „*Framework JCreator Projekt Dateien\JOGL Framework Lightweight Edition*“
[7] Nur die *einzelnen* Java-Dateien von der CD kopieren, nicht das ganze Projekt!
[8] Achtung: Kommandos beinhalten sichtbare Leerzeichen zum besseren Überblick!

A.3 Quellcodes

Hier werden die Quellcodes der Klassen `Heavyweight-/LightweightListener` und `RunFrameWork` aufgelistet. Die Quellcodes hier besitzen aber nicht immer die Orginal-Zeilennummern, weil sie an das Seitenformat dieser Diplomarbeit angepasst wurden und größere Zeilen teilweise umgebrochen werden mussten!

A.3.1 Quellcode: Klasse RunFrameWork

```
1  import javax.swing.*;              //Anzeigemenue: DispMenu
2  import dispmenu.dispmenu.*;        //Anzeigemenue -> DispMenu
3
4  import framework.jogl.*;           //Fuer: JOGLMainListener,
5                                     //          JOGLMainFrame
6
7  import java.awt.DisplayMode;       //Displaymode-Objekt,
8                                     //falls kein Anzeigemenue
9                                     //gewuenscht.
10 /**
11  * Start-Klasse des Frameworks. Hier wird ein
12  * JOGLMainFrame erstellt und ein DispMenu (Anzeigemenue).
13  * Dem JOGLMainFrame kann mit addJOGLMainListener()
14  * ein beliebiger JOGLMainListener hinzugefuegt werden.
15  * Standardmaessig ist hier bereits ein HeavyweightListener
16  * zugeordnet.
17  */
18 public class RunFrameWork
19 {
20    /**
21     * Haupteinstiegspunkt des Frameworks.
22     *
23     * @param args     Parameteruebergabe an main-Methode().
24     *                 Hat hier aber keine Bedeutung.
25     */
26     public static void main(String[] args)
27     {
28         javax.swing.SwingUtilities.
29                                     invokeLater(new Runnable()
30         {
31
32
33
```

```
34              /**
35               * Seperater Thread in dem die einzelnen
36               * Hauptkomponenten des Frameworks erstellt
37               * und miteinander verknuepft werden.
38               */
39              public void run()
40              {
41                  /** Look&Feel von Swing einstellen*/
42                  JFrame.setDefaultLookAndFeelDecorated(true);
43
44                  /** JOGLMainFrame erstellen*/
45                  JOGLMainFrame runFrame = new JOGLMainFrame();
46
47                  //-----------------------------------------------
48                  // ** Anfang Benutzereinstellungen **
49                  //-----------------------------------------------
50
51                  /** JOGLMainListener zuordnen*/
52                  runFrame.addJOGLMainListener(new
53                                  HeavyweightListener(runFrame));
54
55                  /** Anzeigemenue verwenden (optional!)*/
56                  DispMenu menu = new
57                      DispMenu("JOGL Framework V.3.6",runFrame);
58
59                  //-----------------------------------------------
60                  // ** Ende der Benutzereinstellungen **
61                  //-----------------------------------------------
62              }
63          });
64      }
65  }
```

A.3.2 Quellcode: Klasse HeavyweightListener

```
1  import javax.media.opengl.*;           //Fuer GL-Objekt und
2                                         //GLEventListener
3
4  import javax.media.opengl.glu.*;       //Fuer OpenGL Utility-
5                                         //Library
6
7  import com.sun.opengl.util.*;          //Fuer OpenGL Utility-
8                                         //Toolkit
9
10 import com.sun.opengl.util.texture.*; //Fuer Texturen mit
11                                        //TextureIO
12
13 import java.io.*;                      //Fuer System.out etc.
14
15 import java.nio.*;                     //Neue Java Input/
16                                        //Output Klasse
17
18
19 import java.awt.*;              //Klasse Point etc.
20 import java.awt.event.*;        //AWT-Listener
21
22 import framework.fps.*;         //FpsCounter
23 import framework.controls.*;    //Keyboard-/MouseHandler
24 import framework.jogl.*;        //JOGLCamera, JOGLText
25 import framework.utils.*;       //3D-Ursprung, Fokus-Quadrat
26
27 /**
28 * Vordefinierter JOGLMainListener mit bereits folgenden
29 * vorimplementierten Komponenten des Frameworks:
30 *
31 * GLU, GLUT, KeyboardHandler, 2 mal den MouseHandler,
32 * FpsCounter, JOGLText, JOGLCamera,
33 * JOGLOrigin (3D-Ursprung),
34 * JOGLFocus (Kamera-Orientierungshilfe).
35 *
36 * OpenGL ist bereits mit haeufig verwendeten
37 * Einstellungen vorinitialisiert!
38 */
39
```

```
40 public class HeavyweightListener extends JOGLMainListener
41 {
42     // OpenGL Objekt
43     private GL gl                                = null;
44
45     // OpenGL Utility Library
46     private GLU glu                              = new GLU();
47
48     // OpenGL Utility Toolkit
49     private GLUT glut                            = new GLUT();
50
51     // Framework KeyboardHandler
52     private KeyboardHandler keyboard = new
53                                         KeyboardHandler();
54
55     // Framework MouseHandler fuer Camera
56     //(Relative Koordinaten -> Erweiterte Variante)
57     private MouseHandler mouseCam  = new
58                                           MouseHandler(true);
59
60     // Framework MouseHandler fuer normalen Mauscursor
61     // (normale Variante)
62     private MouseHandler mouseNorm = new
63                                          MouseHandler(false);
64
65     // Framework FpsCounter zum Messen der Frames per second
66     private FpsCounter   fpsCounter     = new FpsCounter();
67
68     // Framework JOGLText zur Unterstuetzung von textausgabe
69     private JOGLText   joglText       = new JOGLText();
70
71     // Framework JOGLCamera
72     private JOGLCamera camera         = null;
73
74     //Framework 3D-Ursprung
75     private JOGLOrigin origin         = new JOGLOrigin();
76
77     //Framework Fokus-Quadrat zur Kamera-Orientierungshilfe
78     private JOGLFocus camFocus        = new JOGLFocus(false);
79
80
```

```
81      //Speichert relative Koordinaten des MouseHandlers
82      private Point relKoord                  = null;
83
84      //Speichert Status, ob Hinweismenue aktiv oder nicht
85      private boolean showInfoMessage         = true;
86
87      /**
88      * Der HeavyweightListener erwartet die Angabe
89      * einer Klasse, welche das Interface JOGLMainFrameInterf
90      * implementiert. Im Normalfall sollte hier eine Instanz
91      * des JOGLMainFrames angegeben werden.
92      *
93      * @param mainFrame  Interface JOGLMainFrameInterf
94      */
95      public HeavyweightListener(JOGLMainFrameInterf mainFrame)
96      {
97          //Weitergabe des JOGLMainFrameInterf an die
98          //OberklasseJOGLMainListener
99          super(mainFrame);
100     }
101
102     //-----------------------------------------------------
103     // **** Eigene globale Methoden/Variablen hier ****
104     //-----------------------------------------------------
105
106     //                    - Code -
107
108     //-----------------------------------------------------
109     // **** Ende eigene globale Methoden/Variablen ****
110     //-----------------------------------------------------
111
112     /** GLEventListener-Methode (Siehe JOGL API)*/
113     public void init(GLAutoDrawable glDrawable)
114     {
115         // AWT Keylistener, MouseListener und
116         // MouseMotionListener registrieren.
117         glDrawable.addKeyListener(this);
118         glDrawable.addMouseListener(this);
119         glDrawable.addMouseMotionListener(this);
120
121
```

```
122         /* Anfang OpenGL Standard-Initialisierungen */
123
124         // OpenGL-Objekt holen
125         gl  = glDrawable.getGL();
126
127         gl.glShadeModel(GL.GL_SMOOTH);
128         gl.glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
129         gl.glClearDepth(1.0f);
130         gl.glEnable(GL.GL_DEPTH_TEST);
131         gl.glDepthFunc(GL.GL_LEQUAL);
132         gl.glHint(GL.GL_PERSPECTIVE_CORRECTION_HINT,
133                                                 GL.GL_NICEST);
134         gl.setSwapInterval(1);
135
136         /* Ende OpenGL Standard-Initialisierungen */
137
138         //------------------------------------------------------
139         // **** Anfang eigene Initialisiserungen hier  ****
140         //------------------------------------------------------
141
142         //                    - Code -
143
144         //------------------------------------------------------
145         // **** Ende eigene Initialisierungen Variablen ****
146         //------------------------------------------------------
147     }
148
149     /** GLEventListener-Methode (Siehe JOGL API)*/
150     public void display(GLAutoDrawable glDrawable)
151     {
152         // Fps messen
153         fpsCounter.checkFps();
154
155         // Wenn, Keyboardtaste "Escape" gedrueckt
156         if (keyboard.checkIsKeyPressed(KeyEvent.VK_ESCAPE))
157         {
158             //Programm herunterfahren
159             mainFrame.shutdown();
160
161             //Loslassen der Escape-Taste simulieren.
162             keyboard.setManualRelease(KeyEvent.VK_ESCAPE);
```

```
163             }
164
165             //w,a,s,d,e,q Ansteuerung der Kamera
166             if (keyboard.checkIsKeyPressed(KeyEvent.VK_W))
167                 camera.cameraMoveForward();
168             if (keyboard.checkIsKeyPressed(KeyEvent.VK_S))
169                 camera.cameraMoveBackward();
170             if (keyboard.checkIsKeyPressed(KeyEvent.VK_A))
171                 camera.cameraMoveSideLeft();
172             if (keyboard.checkIsKeyPressed(KeyEvent.VK_D))
173                 camera.cameraMoveSideRight();
174             if (keyboard.checkIsKeyPressed(KeyEvent.VK_Q))
175                 camera.cameraHoverDown();
176             if (keyboard.checkIsKeyPressed(KeyEvent.VK_E))
177                 camera.cameraHoverUp();
178
179             //Relative Mousekoordinaten ermitteln
180             relKoord = mouseCam.getRelativeMouseXY();
181
182             //Wenn die Maus ueberhaupt bewegt wird
183             if (relKoord.x != 0 || relKoord.y != 0)
184             {
185                 //Camera Vertikale Bewegung berechnen
186                 camera.cameraLookV(-relKoord.y);
187                 //Camera Horizontale Bewegung berechnen
188                 camera.cameraLookH(relKoord.x);
189             }
190
191             /* Anfang OpenGL Standard-Initialisierungen */
192
193             // OpenGL-Objekt holen
194             gl = glDrawable.getGL();
195
196             gl.glClear(GL.GL_COLOR_BUFFER_BIT |
197                                         GL.GL_DEPTH_BUFFER_BIT);
198             gl.glLoadIdentity();
199
200             /* Ende OpenGL Standard-Initialisierungen */
201
202             //Orientierungshilfe zeichnen
203             camFocus.drawFocus(gl);
```

```
204         //Camera aktualisieren
205         camera.cameraUpdate(glu);
206         //Matrix-Stack sichern
207         gl.glPushMatrix();
208
209         //Ursprung zeichnen (Diese Zeile kann bei Bedarf
210         //                   entfernt werden!)
211         origin.drawOrigin(gl);
212
213         //------------------------------------------------------
214         // **** Anfang eigener Code fuer Hauptschleife hier
215         //
216         // (Anstatt glLoadIdentity() sollte hier
217         //  gl.popMatrix() benutzt werden, um Konflikte mit
218         //  der Camera zu vermeiden!)
219         //------------------------------------------------------
220
221         //                   - Code -
222
223         //------------------------------------------------------
224         // **** Ende eigener Code fuer Hauptschleife    ****
225         //------------------------------------------------------
226
227
228         // Wenn, Keyboardtaste "F1" gedrueckt
229         if (keyboard.checkIsKeyPressed(KeyEvent.VK_F1))
230         {
231             //Staus veraendern (Hinweismenue an/aus)
232             showInfoMessage = !showInfoMessage;
233
234             //Loslassen der F1-Taste simulieren.
235             keyboard.setManualRelease(KeyEvent.VK_F1);
236         }
237
238         //Steuerungshinweise anzeigen
239         if (showInfoMessage)
240         {
241             //Matrix zuruecksetzen, damit sich Text nicht
242             //mit Kamera bewegt
243             gl.glLoadIdentity();
244
```

```
245             //Textausgabe
246             joglText.drawText("Fps: ", 0.0f, 1.0f, 1.0f,
247                             -1.1f, 0.70f, -2.0f, gl, glut);
248
249             joglText.drawText(""+fpsCounter.getActFps(),
250                         1.0f, 0.0f, 0.0f, -0.75f, 0.70f,
251                                             -2.0f, gl, glut);
252
253             joglText.drawText("[Control]", 0.0f, 1.0f,
254                         1.0f, -1.1f, 0.6f, -2.0f, gl, glut);
255
256             joglText.drawText("Exit program:", 0.0f, 1.0f,
257                         1.0f, -1.1f, 0.55f, -2.0f, gl, glut);
258
259             joglText.drawText("Esc", 1.0f, 0.0f, 0.0f,
260                             -0.75f, 0.55f, -2.0f, gl, glut);
261
262             joglText.drawText("Menu on/off:", 0.0f, 1.0f,
263                         1.0f, -1.1f, 0.5f, -2.0f, gl, glut);
264
265             joglText.drawText("F1", 1.0f, 0.0f, 0.0f,
266                             -0.75f, 0.5f, -2.0f, gl, glut);
267
268             joglText.drawText("Moving Camera:", 0.0f,
269                 1.0f, 1.0f, -1.1f, 0.45f, -2.0f, gl, glut);
270
271             joglText.drawText("w, a, s, d, e, q", 1.0f,
272                0.0f, 0.0f, -0.75f, 0.45f, -2.0f, gl, glut);
273
274             joglText.drawText("Change View:", 0.0f, 1.0f,
275                         1.0f, -1.1f, 0.4f, -2.0f, gl, glut);
276
277             joglText.drawText("Drag Right Mouse", 1.0f,
278                 0.0f, 1.0f, -0.75f, 0.4f, -2.0f, gl, glut);
279         }
280     }
281
282     /** GLEventListener-Methode (Siehe JOGL API)*/
283     public void reshape(GLAutoDrawable glDrawable,
284                     int x, int y, int width, int height)
285     {
```

```
286         /* Anfang OpenGL Standard-Initialisierungen */
287
288         // OpenGL-Objekt holen
289         gl = glDrawable.getGL();
290
291         if (height <= 0) height = 1;
292
293         float ratio = (float)width / (float)height;
294
295         gl.glViewport(0, 0, width, height);
296         gl.glMatrixMode(GL.GL_PROJECTION);
297         gl.glLoadIdentity();
298         glu.gluPerspective(45.0f, ratio, 0.1, 1000.0);
299         gl.glMatrixMode(GL.GL_MODELVIEW);
300         gl.glLoadIdentity();
301
302         /* Ende OpenGL Standard-Initialisierungen */
303
304         //Dem Mousehandler aktuelle Position des GLCanvas
305         //mitteilen
306         mouseCam.canvasSizeUpdate(width, height);
307
308         //Dem Mousehandler aktuelle Dimensionen des
309         //GLCanvas mitteilen
310         mouseCam.canvasLocationUpdate(
311                 mainFrame.getCanvasLocationOnScreen().x,
312                  mainFrame.getCanvasLocationOnScreen().y);
313
314         //Mauscursor zentrieren
315         mouseCam.centerMouse();
316
317         //Camera Starteinstellungen einstellen
318         if (camera == null)
319         {
320             //Camera aufstellen an bestimmter Position und
321             //Blickrichtung
322             camera = new JOGLCamera(glu, 1.0f, 1.0f,
323                                             5.0f, 0.0f, 0.0f);
324             //Speed --> Bewegen der Kamera
325             camera.setMoveSpeed(0.075f);
326
```

```
327             //Speed --> Schweben der Kamera
328             camera.setHoverSpeed(0.075f);
329         }
330         //-------------------------------------------------------
331         // **** Anfang Eigener Code fuer Aktionen bei
332         //      Groessenaenderung ****
333         //-------------------------------------------------------
334
335         //                    - Code -
336
337         //-------------------------------------------------------
338         // **** Ende eigener Code fuer Aktionen bei
339         //      Groessenaenderung ****
340         //-------------------------------------------------------
341     }
342
343     /** AWT-MouseListener (Siehe Java API)*/
344     public void mouseEntered(MouseEvent e)
345     {
346         //Mousehandler aktuelle Position des GLCanvas
347         //mitteilen
348         mouseCam.canvasLocationUpdate(
349                 mainFrame.getCanvasLocationOnScreen().x,
350                  mainFrame.getCanvasLocationOnScreen().y);
351     }
352
353     /** AWT-MouseListener (Siehe Java API)*/
354     public void mouseReleased(MouseEvent e)
355     {
356         //Beide Mousehandler aktualisieren
357         mouseCam.mouseReleasedUpdate(e);
358         mouseNorm.mouseReleasedUpdate(e);
359
360         //Wenn rechte Maus losgelassen
361         if (!mouseCam.checkIsRightButtonDown())
362         {
363             //Orientierungshilfe ausschalten
364             camFocus.setVisible(false);
365             //Mauscursor wieder anzeigen
366             mainFrame.showMouseCursor();
367         }
```

```
368     }
369
370     /** AWT-MouseListener (Siehe Java API)*/
371     public void mousePressed(MouseEvent e)
372     {
373         //Beide Mousehandler aktualisieren
374         mouseCam.mousePressedUpdate(e);
375         mouseNorm.mousePressedUpdate(e);
376
377         //Wenn rechte Maus gedrueckt dann Mauscursor
378         //unsichtbar
379         if (mouseCam.checkIsRightButtonDown())
380             mainFrame.hideMouseCursor();
381     }
382
383     /** AWT-MouseMotionListener (Siehe Java API)*/
384     public void mouseDragged(MouseEvent e)
385     {
386         //Mousehandler aktualisieren
387         mouseNorm.mouseMovedUpdate(e);
388
389         //Wenn rechte Maus gedrueckt
390         if (mouseCam.checkIsRightButtonDown())
391         {
392             //Orientierungshilfe einschalten
393             camFocus.setVisible(true);
394             //NUR HIER diesen Mousehandler aktualisieren
395             mouseCam.mouseMovedUpdate(e);
396         }
397     }
398
399     /** AWT-MouseMotionListener (Siehe Java API)*/
400     public void mouseMoved(MouseEvent e)
401     {
402          //Mousehandler aktualisieren
403          mouseNorm.mouseMovedUpdate(e);
404     }
405
406     /** AWT-KeyListener (Siehe Java API)*/
407     public void keyReleased(KeyEvent e)
408     {
```

```
409             // KeyboardHandler aktualisieren
410             keyboard.keyReleasedUpdate(e);
411         }
412
413         /** AWT-KeyListener (Siehe Java API)*/
414         public void keyPressed(KeyEvent e)
415         {
416             // KeyboardHandler aktualisieren
417             keyboard.keyPressedUpdate(e);
418         }
419 }
```

A.3.3 Quellcode: Klasse LightweightListener

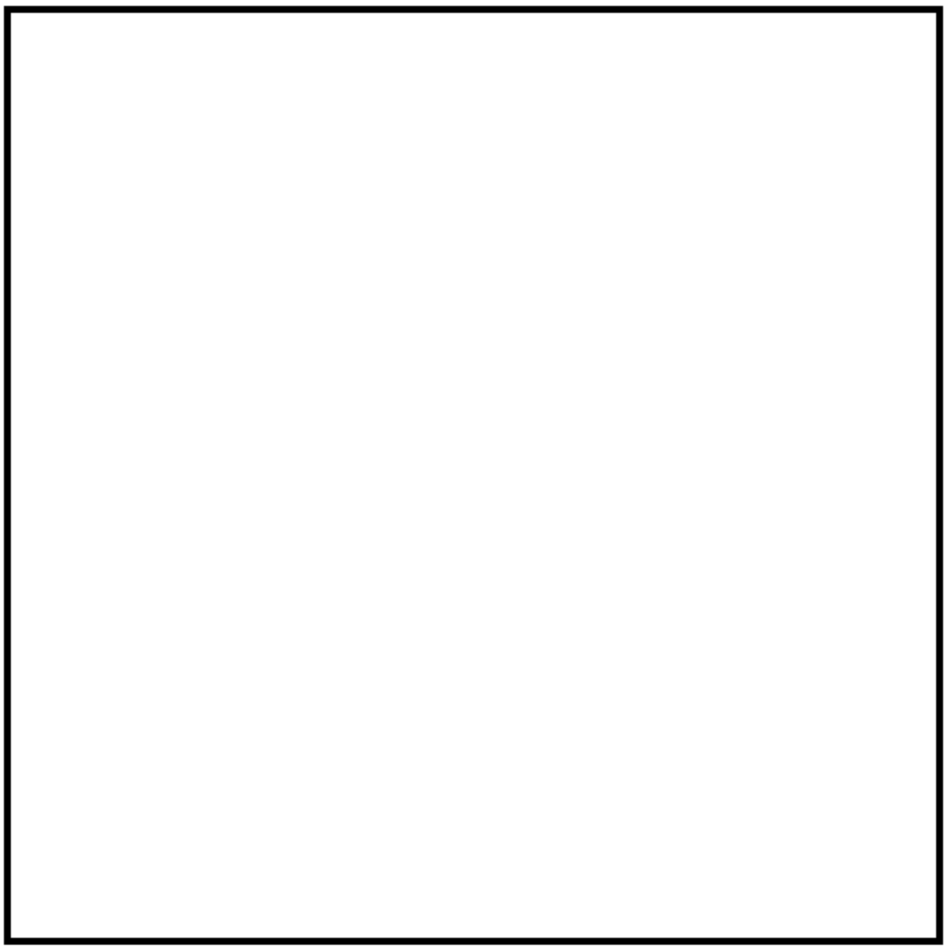
```
1  import javax.media.opengl.*;          //Fuer GL-Objekt und
2                                        //GLEventListener
3
4  import javax.media.opengl.glu.*;      //Fuer OpenGL Utility-
5                                        //Library
6
7  import com.sun.opengl.util.*;         //Fuer OpenGL Utility-
8                                        //Toolkit
9
10 import com.sun.opengl.util.texture.*;  //Fuer Texturen mit
11                                         //TextureIO
12
13 import java.io.*;                    //Fuer System.out etc.
14 import java.nio.*;                   //Neue Java Input/Output
15                                      //Klasse
16
17 import java.awt.*;                   //Fuer Klasse Point etc.
18 import java.awt.event.*;             //Fuer AWT-Listener
19
20 import framework.fps.*;              //FpsCounter
21 import framework.controls.*;         //Keyboard-/MouseHandler
22 import framework.jogl.*;             //JOGLCamera, JOGLText
23
24 /**
25 * Vordefinierter JOGLMainListener mit bereits folgenden
26 * vorimplementierten Komponenten des Frameworks:
27 *
28 * GLU, GLUT, KeyboardHandler.
29 *
30 * OpenGL ist bereits mit haeufig verwendeten
31 * Einstellungen vorinitialisiert!
32 */
33 public class LightweightListener extends JOGLMainListener
34 {
35     // OpenGL Objekt
36     private GL gl                              = null;
37
38     // OpenGL Utility Library
39     private GLU glu                            = new GLU();
```

```
40
41      // OpenGL Utility Toolkit
42      private GLUT glut                                = new GLUT();
43
44      // Framework KeyboardHandler
45      private KeyboardHandler keyboard    = new
46                                            KeyboardHandler();
47
48      /**
49      * Der LeightweightListener erwartet die Angabe
50      * einer Klasse, welche das Interface JOGLMainFrameInterf
51      * implementiert. Im Normalfall sollte hier eine Instanz
52      * des JOGLMainFrames angegeben werden.
53      *
54      * @param mainFrame  Interface JOGLMainFrameInterf
55      */
56      public LightweightListener(JOGLMainFrameInterf mainFrame)
57      {
58          //Weitergabe des JOGLMainFrameInterf an die
59          //Oberklasse JOGLMainListener
60          super(mainFrame);
61      }
62
63      //-----------------------------------------------------------
64      // **** Eigene globale Methoden/Variablen hier ****
65      //-----------------------------------------------------------
66
67      //                    - Code -
68
69      //-----------------------------------------------------------
70      // **** Ende eigene globale Methoden/Variablen ****
71      //-----------------------------------------------------------
72
73      /** GLEventListener-Methode (Siehe JOGL API)*/
74      public void init(GLAutoDrawable glDrawable)
75      {
76          // AWT Keylistener registrieren. Wichtig: Um den
77          // MouseHandler benutzen zu koennen, muessen
78          // MouseListener und MouseMotionlister ebenfalls
79          // registriert werden. Siehe HeavyweightListener!
80           glDrawable.addKeyListener(this);
```

```
81
82          /* Anfang OpenGL Standard-Initialisierungen */
83
84          // OpenGL-Objekt holen
85          gl  = glDrawable.getGL();
86
87          gl.glShadeModel(GL.GL_SMOOTH);
88          gl.glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
89          gl.glClearDepth(1.0f);
90          gl.glEnable(GL.GL_DEPTH_TEST);
91          gl.glDepthFunc(GL.GL_LEQUAL);
92          gl.glHint(GL.GL_PERSPECTIVE_CORRECTION_HINT,
93                                                  GL.GL_NICEST);
94          gl.setSwapInterval(1);
95
96          /* Ende OpenGL Standard-Initialisierungen */
97
98          //------------------------------------------------------
99          // **** Anfang eigene Initialisiserungen hier    ****
100         //------------------------------------------------------
101
102         //                      - Code -
103
104         //------------------------------------------------------
105         // **** Ende eigene Initialisierungen Variablen ****
106         //------------------------------------------------------
107     }
108
109     /** GLEventListener-Methode (Siehe JOGL API)*/
110     public void display(GLAutoDrawable glDrawable)
111     {
112         // Wenn, Keyboardtaste "Escape" gedrueckt
113         if (keyboard.checkIsKeyPressed(KeyEvent.VK_ESCAPE))
114         {
115             //Programm herunterfahren
116             mainFrame.shutdown();
117
118             //Loslassen der Escape-Taste simulieren.
119             keyboard.setManualRelease(KeyEvent.VK_ESCAPE);
120         }
121
```

```
122         /* Anfang OpenGL Standard-Initialisierungen */
123
124         // OpenGL-Objekt holen
125         gl = glDrawable.getGL();
126
127         gl.glClear(GL.GL_COLOR_BUFFER_BIT |
128                                     GL.GL_DEPTH_BUFFER_BIT);
129         gl.glLoadIdentity();
130
131         /* Ende OpenGL Standard-Initialisierungen */
132
133         //------------------------------------------------------
134         // **** Anfang eigener Code fuer Hauptschleife hier
135         //------------------------------------------------------
136
137         //                    - Code -
138
139         //------------------------------------------------------
140         // **** Ende eigener Code fuer Hauptschleife    ****
141         //------------------------------------------------------
142     }
143
144     /** GLEventListener-Methode (Siehe JOGL API)*/
145     public void reshape(GLAutoDrawable glDrawable,
146                         int x, int y, int width, int height)
147     {
148         /* Anfang OpenGL Standard-Initialisierungen */
149
150         // OpenGL-Objekt holen
151         gl = glDrawable.getGL();
152
153         if (height <= 0) height = 1;
154
155         float ratio = (float)width / (float)height;
156
157         gl.glViewport(0, 0, width, height);
158         gl.glMatrixMode(GL.GL_PROJECTION);
159         gl.glLoadIdentity();
160         glu.gluPerspective(45.0f, ratio, 0.1, 1000.0);
161         gl.glMatrixMode(GL.GL_MODELVIEW);
162         gl.glLoadIdentity();
```

```
163
164             /* Ende OpenGL Standard-Initialisierungen */
165
166             //-------------------------------------------------------
167             // **** Anfang Eigener Code fuer Aktionen bei
168             //       Groessenaenderung ****
169             //-------------------------------------------------------
170
171             //                      - Code -
172
173             //-------------------------------------------------------
174             // **** Ende eigener Code fuer Aktionen bei
175             //       Groessenaenderung    ****
176             //-------------------------------------------------------
177         }
178
179         /** AWT-KeyListener (Siehe Java API)*/
180         public void keyReleased(KeyEvent e)
181         {
182             // KeyboardHandler aktualisieren
183             keyboard.keyReleasedUpdate(e);
184         }
185
186         /** AWT-KeyListener (Siehe Java API)*/
187         public void keyPressed(KeyEvent e)
188         {
189             // KeyboardHandler aktualisieren
190             keyboard.keyPressedUpdate(e);
191         }
192     }
```



Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum	Adrian Dietzel