TECHNISCHE UNIVERSITÄT DRESDEN

Grafikprogrammierung mit OpenGL

Proseminar Computergrafik

Sommersemester 2009

Dozent: Wilfried Mascolus
Lehrstuhl für Computergraphik und Visualisierung
Fakultät Informatik

Sebastian Hubl
Matrikelnummer: 3008011
s5537767@mail.inf.tu-dresden.de

Inhaltsverzeichnis

1 Abstract 4

2 Grundlagen von OpenGL 5

2.1 Was ist OpenGL? 5
2.2 Merkmale von OpenGL 5
2.3 Bibliotheken 6
2.4 Zustandsautomat 6
2.5 Konstanten, Datentypen, Funktionen, Konventionen 7

3 Primitive 8

3.1 Punkte 9
3.2 Linien 9
3.3 Dreiecke 9
3.4 Vierecke und Polygone 10

4 Farbe, Licht und Material 12

4.1 Materialdefinitionsmodi 12
4.2 Der glColor – Materialmodus 12
4.3 Transparenz 12
4.4 Der glMaterial – Materialmodus 13
4.4.1 Lichtquellen 13
4.4.1.1 Lichtquelleneigenschaften 14
4.4.1.2 Lichtquellentypen 15
4.4.2 Material 16

5 Viewing Pipeline 18

5.1 Die Arbeit mit Matrizen 18
5.1.1 Modelview- und Projection-Matrix 19
5.1.2 Hilfsfunktionen 19
5.1.3 Display Lists 20
5.1.4 Push und Pop 21
5.2 Modelltransformation 21
5.3 Ansichtstransformation 22
5.4 Projektionsarten 23
5.5 Viewport 24

6 Quellenangabe 26

Abbildungsverzeichnis

2.1 Grafikprimitive in OpenGL 11
4.2 Lichtquellentypen 16
5.1 Viewing Pipeline 19
5.2 Orthogonale und Perspektivische Projektionen 24
5.3 unverzerrter und verzerrter Viewport 25

1 Abstract

Der Vortrag „Grafikprogrammierung mit OpenGL“ führt den Hörer in die Grundlagen der Programmierung von zwei- und dreidimensionalen Grafiken mittels des offenen Standards OpenGL ein. Schwerpunkte bilden neben der grundlegenden Vorstellung von OpenGL, das Zeichnen von Primitiven, die farbliche Darstellung und das Zusammenspiel von Licht und Material in Grafikszenen sowie die Viewing Pipeline.

Ziel des Vortrags ist es, den Hörer mit den Grundlagen der Grafikprogrammierung unter OpenGL vertraut zu machen und ihm wichtige Techniken zur Realisierung eigener Projekte aufzuzeigen. Aufgrund der Komplexität des Themas muss ein Grundverständnis der allgemeinen Grafikprogrammierung sowie Grundkenntnisse der Mathematik und Physik unterstellt werden.

2 Grundlagen von OpenGL

2.1 Was ist OpenGL?

OpenGL steht für „Open Graphics Library“ und bezeichnet einen plattform- und programmiersprachenunabhängigen Standard zur Programmierung grafikfähiger Hardware. Der OpenGL-Standard beschreibt etwa 250 verschiedene Befehle, mit denen sich zwei- und dreidimensionale Grafiken erzeugen lassen.

Er entstand 1992 aus dem von der Firma SGI, Silicon Graphics Incorporated, entwickelten IRIS GL und wird durch das ARB („Architecture Review Board“), einem Zusammenschluss mehrerer Firmen wie Intel, AMD oder NVIDIA, festgelegt und überwacht. Die Firma Microsoft, eines der Gründungsmitglieder des ARB, verlies dieses 2003. Ende Juli 2006 übernahm die Khronos Group, ein 2000 gegründetes Industriekonsortium, die Weiterentwicklung des Grafikstandards.

Die aktuellste Version ist 3.1, genannt „Longs Peak Reloaded“, die im März 2009 erschienen ist.

2.2 Merkmale von OpenGL

Um OpenGL näher zu verstehen, ist es zunächst nötig, auf dessen Merkmale näher einzugehen.

Das wichtigste Merkmal ist seine Plattform- und Programmiersprachenunabhängigkeit. Betriebssysteme wie Windows von Microsoft oder die Unix-Systeme bringen von Haus aus die nötigen Systembibliotheken zur Ausführung von OpenGL-Funktionen mit. Aufgrund, dass OpenGL keine direkte Implementierung von Funktionen darstellt sondern nur einen Standard für diese spezifiziert, ist es folglich auch nicht von einer Programmiersprache abhängig. Allerdings müssen die entsprechenden Bibliotheken für die einzelnen Programmiersprachen zur Verfügung gestellt werden. Es gibt Implementierungen für C, C#, Java und viele andere.

Ein weiteres wichtiges Merkmal ist, das OpenGL als Zustandsautomat entworfen wurde. Damit lassen sich verschiedene Zustände, wie die aktuelle Farbe oder die Transparenz aktivieren und durch Funktionen ihre Auswirkung spezifizieren.

Desweiteren zeichnet OpenGL eine Client-Server-Architektur aus und die Fähigkeit, OpenGL-Befehle über ein Netzwerk auf anderen Rechnern bearbeiten bzw. ausführen zu lassen. Es stellt zudem eine hohe Speicheranforderung an die

Hardware. Grafikkarten, welche die Darstellung von OpenGL-Grafiken ermöglichen sollen, müssen mindestens vier verschiedene Pufferspeicher implementieren. Diese wären zum einen der Bildspeicher (Frame Buffer), welcher die Darstellung der Grafikszene speichert, ein Tiefenspeicher (Depth Buffer) zur Tiefenberechnung sowie ein Maskenspeicher (Stencil Buffer) und ein Akkumulationsspeicher. OpenGL unterstützt zudem den sogenannten Doublebuffer-Modus. Verfügt die Hardware einer Grafikhardware über wenigstens zwei Bildspeicher, wobei OpenGL sogar bis zu vier unterstützt, kann einer der beiden, der sogenannte Backbuffer, benutzt werden, um die neue Szene vorzuzeichnen. Der andere Speicher (sog. Frontbuffer) beinhaltet die aktuelle Szene, die auf dem Bildschirm angezeigt wird. Beide Puffer können über eine Systembefehl getauscht werden, sodass nun die im Backbuffer vorgezeichnete Szene zur Anzeige gebracht wird. Der Inhalt des Frontbuffers wird gelöscht und kann nun selber als Backbuffer genutzt werden. Doublebuffering ist sehr wichtig, wenn fließende und performante Bildübergänge nötig sind, wie bei der Animations- und Spieleprogrammierung.

2.3 Bibliotheken

Es gibt zwei Bibliotheken, die die OpenGL-Funktionen zur Verfügung stellen. Zum einen umfasst die OpenGL Library (*Abkürzung: GL*) ca. 200 grundlegende Befehle, mit denen sich unter anderem einfache Primitive, wie Linien oder Dreiecke zeichnen lassen. Zum anderen existiert die OpenGL Utility Library (*GLU*) , die knapp 50 Befehle zur Verfügung stellt, welche jeweils mehrere elementare Befehle der OpenGL Library zu komplexeren Operationen zusammenfasst. So lassen sich zum Beispiel NURBS zeichnen oder auch Kamerapositionen einstellen. Eine weitere, aber nicht direkt du OpenGL gehörende Bibliothek ist das OpenGL Utility Toolkit (*GLUT*), welches auf den beiden genannten Bibliotheken aufsetzt und zudem eine Anbindung an das jeweilige Betriebssystem herstellt. Mit GLUT lassen sich Fenster erstellen, in denen OpenGL-Grafiken angezeigt werden. Zudem kann man auf Tastatur- oder Mausereignisse reagieren und so mit der Szene interagieren.

2.4 Zustandsautomat

Wie anfangs bereits erwähnt wurde, hat man OpenGL als Zustandsautomat entwickelt, wobei sich das Verhalten von OpenGL über knapp 250 Zustandsvariablen kontrollieren und steuern lässt. Beispiele für solche Variablen sind zum Beispiel die aktuelle Zeichenfarbe oder das Ein- und Ausblenden von Vorder- oder Rückseiten von

Flächenprimitiven. Jede Variable bzw. jedes Attribut ist zu Beginn mit einem Standardwert initialisiert. Dieser bleibt solange gesetzt, bis er geändert wird.

Zustände lassen sich über die OpenGL-Befehle

```
void glEnable(GLenum cap)
void glDisable (GLenum cap)
```

aktivieren bzw. deaktivieren. In den meisten Fällen reicht es aber nicht, einen Zustand nur zu setzen, sondern man benötigt weitere Funktionen oder andere aktive Zustände, damit ein gewünschtes Ergebnis erzielt werden kann.

2.5 Konstanten, Datentypen, Funktionen, Konventionen

Alle Datentypen, Konstanten und Funktionen folgen bestimmten Richtlinien in ihrer Benennung.

Konstanten werden prinzipiell groß geschrieben und beginnen immer mit einem voranstehenden „GL“ gefolgt von einem Unterstrich und dem Namen der Konstante. Einige Beispiele für Konstanten sind: `GL_MODELVIEW,GL_TRUE, GL_FALSE.`

Datentypen beginnen ebenfalls mit GL gefolgt von dem Namen, z.B. GLbyte, GLshort. Tabelle 2.1 zeigt eine Übersicht der verschiedenen Datentypen, die OpenGL verwendet. In vielen Programmiersprachen existieren Äquivalente zu den Datentypen (in der Übersicht ist eine Spalte mit dem jeweiligen C-Äquivalent angegeben.

Zuletzt sei noch die OpenGL-Syntax zur Beschreibung von Funktionen zu erwähnen. Ein Beispiel, an dem der Aufbau beschrieben werden soll, wäre zum Beispiel die Funktion zum Setzen des aktuellen Farbwertes:

```
gl Color 3 f v (Parameter)
```

Funktionen in OpenGL beginnen immer mit dem Namen der Bibliothek in Kleinbuchstaben, die diese zur Verfügung stellt. In oben stehender Funktion bedeutet `gl`, dass die Funktion aus der OpenGL Library (GL) stammt. Danach folgt der Name der Funktion. Der Zahlenwert legt fest, wie viele Parameter der Funktion übergeben werden und `f` bezeichnet den Datentyp der Parameter. Hier stünde `f` für einen GLfloat.Tabelle 2.1 stellt in der ersten Spalte die entsprechenden Kürzel dar. Zuletzt sei noch auf das `v` eingegangen. Damit lässt sich angeben, dass man die Paramater nicht explizit übergeben möchte, sondern in einem Vektordatentyp, wie einem Array. Dieses ist optional. Nachfolgend sind einige Beispiele für Funktionen angegeben:

(i) Übergabe von 3 Zahlen:

```
glColor3f(1.0f,1.0f,1.0f);
glColor3d(1.0,1.0,1.0);
```

(ii) Übergabe eines Vektors (z.B. als Array):

```
GLfloat v[3] = {1.0f,0.8f,1.0f};
glColor3fv(v);
```

Kürzel	Datentyp	OpenGL-Bezeichner	C-Äquivalent
b	8-bit Integer	GLbyte	signed char
s	16-bit Integer	GLshort	short
i	32-bit Integer	GLint GLsizei	int
f	32-bit Floating-Point	GLfloat GLclampf	float
d	64-bit Floating-Point	GLdouble GLclampd	double
ub	8-bit unsigned Integer	GLubyte GLboolean	unsigned char
us	16-bit unsigned Integer	GLushort	unsigned short
ui	32-bit unsigned Integer	GLuint GLenum GLbitfield	unsigned int

Tabelle 2.1: Datentypen von OpenGL

3 Primitive

Primitive sind in OpenGL die einfachen Grafikelemente Punkt, Linie und Dreiecke. Vierecke und Polygone gehören ebenfalls noch dazu, sind allerdings bereits erweiterte Flächenprimitive, weil sich diese aus zwei oder mehreren Dreiecken zusammengesetzen lassen. Die Eckpunkte eines Primitives werden über Vertices festgelegt und bestimmen damit auch dessen Lage im Raum.

Die Definition eines Primitives erfolgt in OpenGL immer innerhalb eines Primitiv-Definitionsbereiches:

```
glBegin(GLenum mode);
      // Vertex-Liste
      glVertex3f(1.0f,1.0f,1.0f);
      // ...
glEnd();
```

Dieser Bereich wird von den Befehlen `glBegin()` und `glEnd()` begrenzt, innerhalb derer dann einer oder mehrere Vertices mit ihren x-, y- und z-Koordinaten über den Befehl `glVertex3f()` festgelegt werden. Welches Primitiv sich aus dieser Vertexliste ergibt, bestimmt die Konstante *`mode`*. Die Werte, die *`mode`* annehmen kann, werden in den folgenden Abschnitten näher erläutert.

3.1 Punkte

Um Punkte zu zeichnen, belegt man den Parameter *mode* mit der Konstante GL_POINTS, die festlegt, dass jeder definierte Vertex innerhalb des Definitionsbereiches einen Punkt beschreibt. Zusätzlich kann man über die Funktion `glPointSize(GLfloat size)` die Größe des zu zeichnenden Punktes bestimmen.

3.2 Linien

OpenGL bietet drei verschiedene Arten, Linien zu zeichnen. Belegt man *`mode`* mit der Konstante GL_LINES, definieren immer zwei aufeinanderfolgende Vertices eine Linie, wobei jeder Vertex nur zu einer Linie gehören kann. GL_LINE_STRIP stellt die Linienzugvariante dar. Auch hier bilden zwei aufeinanderfolgende Vertices eine Linie mit dem Unterschied, das dabei ein fortlaufender Linienzug entsteht, wobei jeder Vertex außer dem ersten und dem letzten zu zwei Linien gehört. Die letzte Variante, GL_LINE_LOOP, ist ähnlich der Linienzugvariante. Hierbei bildet allerdings der letzte Vertex mit dem ersten ebenfalls eine Linie, sodass ein geschlossener Linienzug entsteht.

3.3 Dreiecke

Dreiecke sind das elementarste Flächenprimitiv, da, wie zu Beginn des Abschnitts schon einmal beschrieben, sich andere Flächen aus Dreiecken zusammensetzen lassen. Bei der Definition ist es wichtig, auf die Reihenfolge der Vertices zu achten. OpenGL arbeitet standardmäßig in einem rechtshändigen Drehsinn. Wenn man die

Vertices in einer Reihenfolge gegen den Uhrzeigersinn definiert, dann steht die positive Flächennormale dem Beobachter zugewandt senkrecht auf der Dreiecksfläche. Über diese Normalen wird die Vorder- und Rückseite eines Dreiecks bestimmt - die positive steht senkrecht auf der Vorderseite, die negative senkrecht auf der Rückseite.

OpenGL bietet für das Zeichnen von Dreiecken ebenfalls drei Arten an. Zum einen ist das GL_TRIANGLES. Hier definieren immer drei aufeinanderfolgende Vertices ein Dreieck, wobei jeder Vertex zu genau einem Dreieck gehört. Die zweite Möglichkeit ist GL_TRIANGLE_STRIP. Bei dieser Variante entsprechen immer die letzten beiden Vertices einer Dreiecksdefinition den ersten zwei der folgenden. Die letzte Variante ist GL_TRIANGLE_FAN. Hierbei haben alle Dreiecke den zuerst definierten Vertex als gemeinsamen Punkt.

Am Ende dieses Abschnittes sollen noch einige Funktionen vorgestellt werden, die mit Dreiecksprimitiven in Zusammenhang stehen. Mittels der Funktion

```
glNormal_(x,y,z);
```

kann man die Richtung der Flächennormalen per Hand setzen. Sie muss immer vor einer Dreiecksdefinition innerhalb des Definitionsbereiches stehen. Die Funktion

```
glEnable(GL_NORMALIZE)
```

ist wichtig für die Lichtberechnung. Dadurch wird die Normalisierung aller durch `glNormal()` gesetzten Flächennormalen eingeschaltet.

In OpenGL ist es auch möglich, Vorder- oder Rückseiten oder gar beide auszublenden, was sehr nützlich bei Transparenzeffekten ist. Die dazu gehörende OpenGL-Direktive lautet:

```
glEnable(GL_CULL_FACE);     // Ausblenden aktivieren
glCullFace(GLenum mode)     // betreffende Seiten bestimmen
```

Der Parameter *mode* kann hier die folgenden Werte annehmen:

GL_FRONT:	blendet alle Vorderseiten aus
GL_BACK:	blendet alle Rückseiten aus
GL_FRONT_AND_BACK:	blendet sowohl Vorder- als auch Rückseiten aus

3.4 Vierecke und Polygone

OpenGL bietet auch das direkte Zeichnen von Vierecken und Polygonen an.
Es gibt zwei Möglichkeiten Vierecke zu zeichnen, zum einen über GL_QUADS und zum anderen GL_QUAD_STRIP. GL_QUADS ist wieder die normale Primitivdefinition für

Vierecke. In diesem Fall bilden vier aufeinanderfolgende Vertices ein Viereck. Bei der zweiten Variante mit GL_QUAD_STRIP bilden, ähnlich der GL_TRIANGLE_STRIP–Variante bei Dreiecken, die letzten beiden Vertices eines Vierecks die ersten beiden des nächsten.

Bei dem Zeichnen von Polygonen ist zu beachten, dass jeder Vertex zu genau einem Polygon gehört und das Zeichnen mehrerer Polygone innerhalb eines Definitionsbereiches nicht möglich ist.

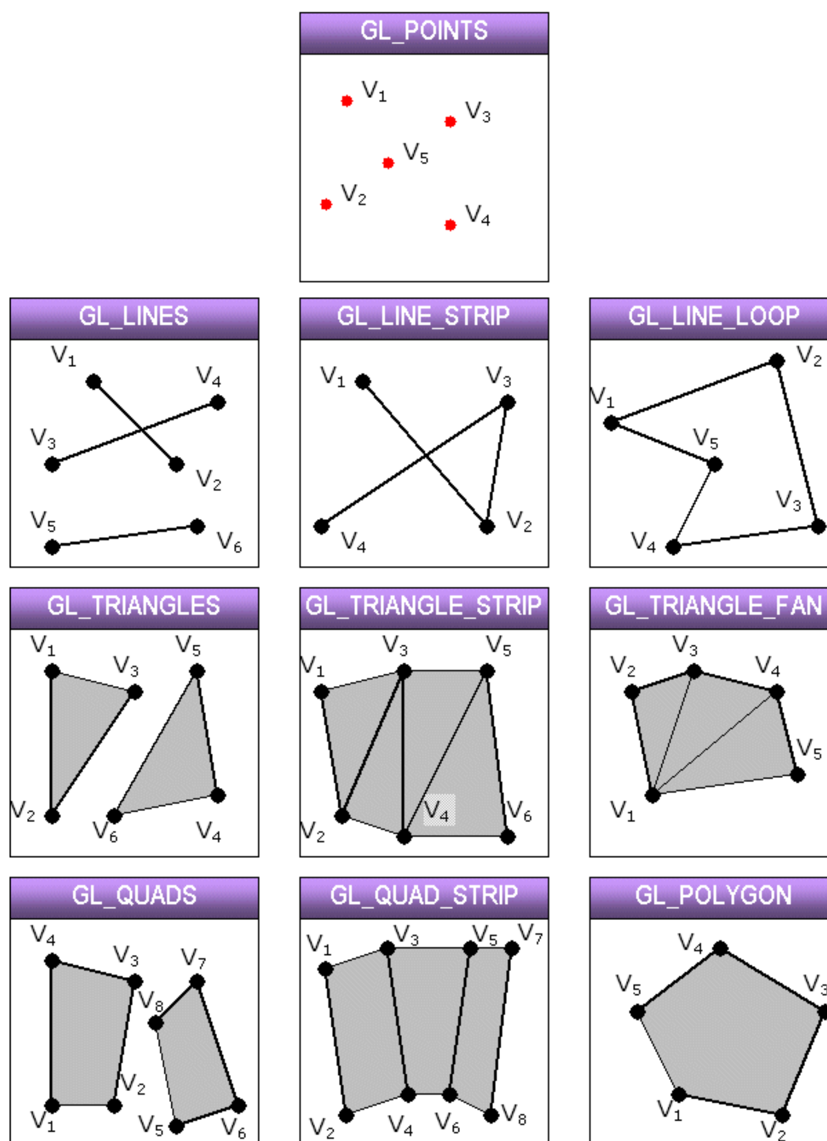

Abbildung 3.1: Grafikprimitive in OpenGL

4 Farbe, Licht und Material

4.1 Materialdefinitionsmodi

Nach dem Zeichnen von Primitiven soll es nun darum gehen, wie man diese farblich gestaltet. Dazu bietet OpenGL zwei verschiedene Modi an: den primitiven gl-Color-Materialmodus und den komplexeren gl-Material-Materialmodus.

4.2 Der gl-Color-Materialmodus

Nutzt man den einfachen Farbmodus um seine Grafikszene farblich zu gestalten, muss man einfach nur den aktuellen Farbwert in OpenGL auf die gewünschte Farbe setzen. Die Vertices, die danach gezeichnet werden, erhalten diesen Farbwert als Eigenschaft. Über die Farbwerte der Vertices eines Primitives definiert sich schließlich dessen Farbe.

Um diesen Farbmodus nutzen zu können, muss man ihn zunächst mit der Direktive

```
glEnable(GL_COLOR_MATERIAL)
```

aktivieren, da diese standardmäßig ausgeschalten ist. Danach kann über den Befehl

```
glColor*(
     GLfloat red, GLfloat green,
     GLfloat blue [, GLfloat alpha]
)
```

der Farbwert gesetzt werden. Die Anzahl der Parameter, die dieser Funktion übergeben werden, variiert zwischen drei und vier. Die ersten drei legen die RGB-Farbwerte in einem Wertebereich von 0.0 bis 1.0 fest, wobei 1.0 für vollen Farbanteil steht und 0.0 für keinen. Optional kann mit dem vierten Parameter der Alpha-Wert angegeben werden, der unter anderem für die Transparenz eines Objektes genutzt wird. Der nächste Abschnitt wird sich näher mit dem Thema Transparenz beschäftigen.

4.3 Transparenz

Wie im letzten Abschnitt bereits angedeutet, kann der Alpha-Wert genutzt werden, um die Transparenz eines Objektes anzugeben. Volle Transparenz erreicht man mit

einem Wert für *alpha* von 0.0 , wobei das Objekt dann komplett durchschimmert. Im Gegensatz dazu bewirkt 1.0 keinen Transparenzeffekt. Innerhalb dieses Bereiches lassen sich Transparenzabstufungen einstellen, 0.6 bedeutet zum Beispiel eine Transparenz von 60%.

Damit der Effekt sichtbar wird, muss dieser mit

```
glEnable(GL_BLEND)
```

aktiviert und mit der Funktion

```
glBlendFunc(
     GL_SRC_ALPHA,
     GL_ONE_MINUS_SRC_ALPHA
)
```

festgelegt werden, wie sich der Effekt auswirkt.

Damit Objekte, die hinter einem transparenten Objekt liegen, sichtbar werden, müssen diese bereits vollständig gezeichnet worden sein.

4.4 Der gl-Material-Materialmodus

Der gl-Material-Materialmodus simuliert die natürliche Farbentstehung durch das Zusammenspiel von Licht und den Materialen der Objekte einer Szene.

Er ist das Gegenstück zum glColor-Materialmodus, und muss deshalb, falls zuvor der gl-Color-Modus eingestellt wurde, über

```
glDisable(GL_COLOR_MATERIAL)
```

aktiviert werden. Weiterhin müssen die Lichteffekte mit

```
glEnable(GL_LIGHTING)
```

aktiviert werden, weil diese standardmäßig ausgeschalten sind.

4.4.1 Lichtquellen

Für die direkte und indirekte Beleuchtung der Szene sorgen Lichtquellen, von denen mindestens acht von OpenGL unterstützt werden. Diese werden über die OpenGL-Routine

`glEnable(GL_LIGHT`i`)` | $0 \leq i \leq 7$

jeweils eingeschaltet, wobei *i* eine Lichtquelle bezeichnet.

4.4.1.1 Lichtquelleneigenschaften

Bei der Definition von Lichtquellen bedient man sich so genannter Lichtmodelle. Jeder Lichtquelle können mehrere Eigenschaften zuweisen, die in Tabelle 4.1 dargestellt sind.

Eigenschaft	**OpenGL-Konstante** ***Wertebereich***	**Beschreibung**
Ambienter Anteil	GL_AMBIENT *RGBA (0.0 .. 1.0)*	Grundhelligkeit
Diffuser Anteil	GL_DIFFUSE *RGBA (0.0 .. 1.0)*	Streuung
Spiegelnder Anteil	GL_SPECULAR *RGBA (0.0 .. 1.0)*	Spiegelung
Leuchtposition	GL_POSITION *(x,y,z,pos)* *pos=1.0: fest; pos=0.0: frei*	Position
Leuchtrichtung	GL_SPOT_DIRECTION *-1.0 .. 1.0 für (x,y,z)*	Wirkrichtung
Leuchtkegel	GL_SPOT_CUTOFF *0.0 .. 90.0* *[Ausnahme: 180.0]*	halber Öffnungswinkel
Leuchtintensität	GL_SPOT_EXPONENT *0.0 .. +128.0*	Leuchtdichtenexponent zum Lichtkegel
Leuchtkraftabschwächung	GL_CONSTANT_ATTANUATION GL_LINEAR_ATTENUATION GL_QUADRATIC_ATTENUATION	Änderung der Leuchtintensität mit der Entfernung

Tabelle 4.1: Lichtquelleneigenschaften

Um nun bestimmte Eigenschaften einer Lichtquelle zu setzen, bedient man sich der Funktion

```
glLight*(
   GLenum light,
   GLenum pname,
   const TYPE param
);
```

```
light:     Name der Lichtquelle
pname:     OpenGL-Konstante der Lichtquelle
param:     Werte der Eigenschaft
```

4.4.1.2 Lichtquellentypen

OpenGL bietet insgesamt vier verschiedene Lichtquellentypen an, die sich nach ihrem Strahlenverlauf und ihrer Position im Raum unterscheiden lassen. Das sind der Spotstrahler, der Punktstrahler, der Ambient- und der Parallelstrahler. Der Typ einer Lichtquelle wird über die Lichtquelleneigenschaften festgelegt.

Spotstrahler

Der Spotstrahler hat einen gerichteten Strahlenverlauf und eine feste Position im Raum. Sein Hauptmerkmal ist ein gerichteter Lichtkegel. Ein Spotstrahler wird wie folgt definiert:

```
glLightf(GL_LIGHTi, GL_POSITION, {x,y,z,1.0});
glLightf(GL_LIGHTi, GL_SPOT_DIRECTION, {x,y,z});
glLightf(GL_LIGHTi, GL_SPOT_CUTOFF, 0.0F..90.0F);
```

Die Konstante GL_POSITION legt die Position des Strahlers im Raum fest. Der letzte Parameter spielt dabei eine wichtige Rolle. Der Wert 1.0 bedeutet, dass es sich um eine Lichtquelle mit fester Position handelt, 0.0 würde eine Lichtquelle ohne feste Position beschreiben. Mittels GL_SPOT_DIRECTION wird die Position im Raum angegeben, auf die der Spot strahlt. Aus der Differenz des Ortsvektors der Zielposition und dem Ortsvektor der Position des Strahlers ergibt sich der Richtungsvektor der Strahlen. GL_SPOT_CUTOFF bezeichnet den halben Öffnungswinkel des Lichtkegels des Strahlers.

Punkstrahler

Der Punktstrahler ist eine Sonderform des Spotstrahlers, mit dem Unterschied, das der Wert für GL_SPOT_CUTOFF auf 180.0F gesetzt. Das bedeutet, dass er keinen gerichteten Strahlenverlauf hat, sondern in alle Richtungen Licht ausstrahlt. Man kann ihn mit einer herkömmlichen Glühbirne vergleichen.

Parallelstrahler

Der Parallelstrahler ist ein reiner Richtungsstrahler ohne feste Position. Das bedeutet, dass der vierte Parameter der Eigenschaft GL_POSITION auf 0.0 gesetzt wird. Die einzelnen Lichtstrahlen treffen stets im gleichen Winkel auf den Primitivflächen auf. Die Richtung seiner Strahlen wird hierbei über Eye-Koordinaten festgelegt, da es im Gegensatz zum Spotstrahler nicht möglich ist, den Richtungsvektor über die Position

und das Ziel zu bestimmen. Eye-Koordinaten bedeuten, das die Strahlrichtung sich aus dem entgegengesetzten Ortsvektor des Parallelstrahlers ergeben.

Ambientstrahler

Der Ambientstrahler ist richtungs- und positionslos. Seine Lichtstrahlen leuchten die Szene gleichmäßig aus allen Richtungen aus. Er ist zu vergleichen mit dem Umgebungslicht auf der Erde.

OpenGL besitzt standardmäßig einen globalen Ambientstrahler, der die Szene bereits mit Aktivierung der Lichteffekte gleichmäßig ausleuchtet, weshalb es in den meisten Fällen nicht nötig ist, eine eigene ambiente Lichtquelle zu definieren. Aus diesem Grund haben alle manuell definierten Lichtquellen standardmäßig keine ambienten Lichtquellenanteile.

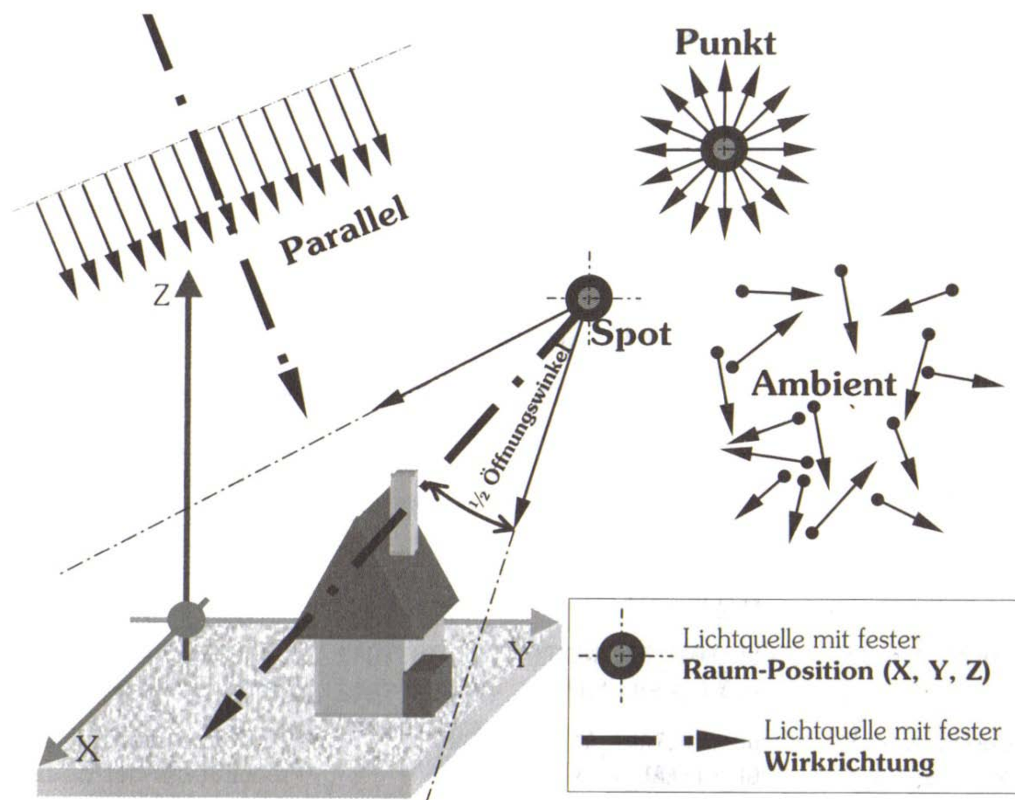

Abbildung 4.2: Lichtquellentypen

4.4.2 Material

Ähnlich den Lichtquellen lassen sich auch Objekte mit komplexen Materialeigenschaften versehen. Diese lassen sich grob einordnen in:

- Farbeigenschaften
- Reflexionseigenschaften
- Emissionseigenschaften

Die Farbeigenschaften bestimmen, welche Farbeanteile des Lichts, mit dem die Objekte bestrahlt werden, reflektiert oder absorbiert werden. Wird ein Körper zum Beispiel mit weißem Licht bestrahlt und hat die Eigenschaft, nur die Rot- und Grünanteile des Lichts zu reflektieren, erscheint der Körper in gelber Farbe. Die Reflexionseigenschaften ambient, diffus oder spiegelnd bestimmen, wie das Licht reflektiert wird. Die Emissionseigenschaften geben einem Körper die Fähigkeit, selbst zu leuchten. Tabelle 4.2 zeigt alle Eigenschaften in einer Übersicht:

Materialeigenschaft	**OpenGL-Konstante** ***Wertebereich***	**Beschreibung**
Ambienter Anteil	GL_AMBIENT *RGBA (0.0 .. 1.0)*	Grundhelligkeit
Diffuser Anteil	GL_DIFFUSE *RGBA (0.0 .. 1.0)*	Lichtbrechung
Spiegelnder Anteil	GL_SPECULAR *RGBA (0.0 .. 1.0)*	Spiegelfähigkeit
Ambienter und diffuser Anteil	GL_AMBIENT_AND_DIFFUSE *RGBA (0.0 .. 1.0)*	Kombinationseigenschaft (ambient und diffus)
Materialemission	GL_EMISSION *RGBA (0.0 .. 1.0)*	passive Strahlereigenschaft
Materialexponent	GL_SHININESS *0.0 .. 128.0*	Spiegelexponent

Tabelle 4.3: Materialeigenschaften

Materialien lassen sich für jedes Flächenprimitv festlegen, selbst ist es möglich, für Vorder- und Rückseiten unterschiedlichen Materialeigenschaften festzulegen. Dies geschieht über die folgende Funktion:

```
glMaterial_[v](
   GLenum face,
   GLenmu pname,
   GLfloat param
);
```

face: legt fest, welche Seite mit dem Material belegt wird (GL_FRONT (Vorderseite),
GL_BACK (Rückseite), GL_FRONT_AND_BACK (beide))

pname: nimmt die OpenGL-Konstante der entsprechenden Eigenschaft auf

param: mit Ausnahme von GL_SHININESS (Wertebereich: 0.0..128.0) nimmt *param* die RGBA-Werte auf

Eine Eigenschaft, auf die bisher noch nicht näher eingegangen wurde, soll noch kurz beleuchtet werden: Mit GL_SHININESS kann man die Oberflächenstruktur simulieren. Je höher man den Wert wählt, desto härter bzw. glatter wirkt das Material.

5 Viewing Pipeline

Nachdem sich die letzten Abschnitte damit beschäftigten, wie man mit OpenGL einfache Primitive zeichnet und sie mit einfachen oder komplexeren Farbeigenschaften belegt, soll der folgende Abschnitt aufzeigen, wie die eigentliche OpenGL-Szene zur Anzeige auf dem Bildschirm gebracht wird. Die einzelnen Schritte, von der Platzierung der einzelnen Elemente über die Betrachtungswinkel bis zur Ausgabe auf dem Bildschirm werden in der OpenGL-Viewing-Pipeline zusammengefasst (siehe auch *Abbildung 5.1*).

5.1 Die Arbeit mit Matrizen

Matrizen spielen in der Grafikprogrammierung mit OpenGL eine große Rolle. Sie speichern aktuelle Szenenzustände, den Blick auf die Szene und verschiedene Projektionen.

OpenGL verwendet drei verschiedene Matrizenstapel zur Realisierung der Viewing-Pipeline: Die Modelview-Matrix, die Projection-Matrix und die Texture-Matrix. Im Weiteren sollen die ersten beiden näher betrachtet werden. Zur dritten, der Texture-Matrix, soll nur kurz gesagt sein, das sie, wie der Name vielleicht schon vermuten lässt, zur Darstellung von Texturen dient und Informationen darüber speichert.

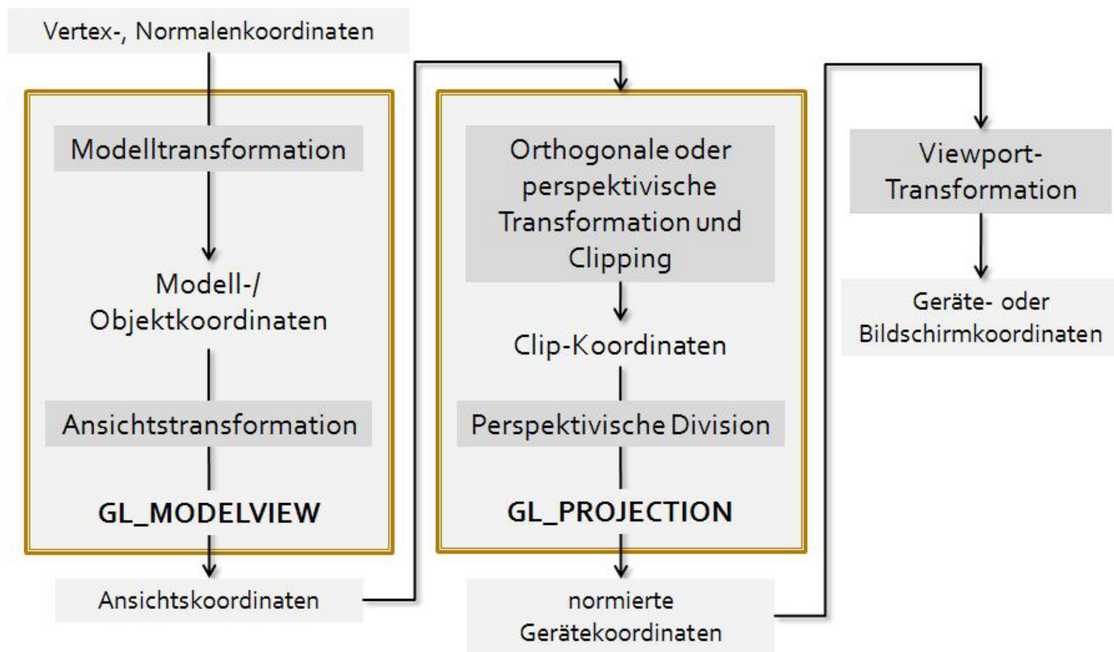

Abbildung 5.1: Viewing Pipeline

5.1.1 Modelview- und Projection-Matrix

Die Modelview-Matrix, in OpenGL angesprochen über die Konstante GL_MODELVIEW, hat zwei Aufgaben. Zum einen speichert sie Modelltransformationen, wie Translation oder Skalierung, auf den Primitiven, zum anderen den Blickwinkel des Betrachters auf die Szene. Beide sind allerdings als identisch anzusehen. Zum Beispiel hat eine Verschiebung aller Szenenobjekte um zwei Einheiten nach links denselben Effekt, als wenn man die Kamera, also die Position und den Blickwinkel des Betrachters auf die Szene, um zwei Einheiten nach rechts verschieben würde.

Die Projection-Matrix, GL_PROJECTION, dient dazu, Informationen über die Projektion der Szene zu speichern. In Abschnitt 5.x folgen nähere Erläuterungen zu Projektionen in OpenGL.

5.1.2 Hilfsfunktionen

Die Arbeit mit diesen Matrizen wird in OpenGL durch verschiedene Funktionen erleichtert und optimiert. Möchte man auf einer Matrix bestimmte Funktionen ausführen, muss diese zunächst erst aktiviert werden bzw. geladen werden. Dies geschieht über den Befehl

```
glMatrixMode(GLenum mode)
```

Möchte man zum Beispiel ein Rechteck verschieben, so muss man mit

```
glMatrixMode(GL_MODELVIEW)
```

zunächst die Modelview-Matrix in den Speicher laden und kann die Transformationen auf dem Rechteck durchführen.

Mit der Funktion

```
glLoadIdentity()
```

überschreibt man die aktuell geladene Matrix mit der Einheitsmatrix, wodurch man alle vorher getätigten Transformation auf der aktuellen Matrix löscht.

Weitere wichtige OpenGL-Funktionen sind

```
glLoadMatrix(const TYPE *M)
```

mit der man die aktuelle Matrix im Speicher mit einer eigens erstellte Matrix M überschreibt, und

```
glMultMatrix(M),
```

die eine Matrix M mit der aktuellen Matrix multipliziert.

5.1.3 Display Lists

Öfter benötigte Transformationen und Primitivdefinitionen, selbst ganze Szenen, lassen sich in sogenannten Display Listen speichern. Sie bringen performante Vorteile mit sich, weil diese Listen nach ihrer Erstellung direkt in den Speicher der Grafikkarte geladen werden und dort mit voller Verarbeitungsgeschwindigkeit ausgeführt werden können. Listen, die einmal geladen wurden, sind später immer Verfügbar. Ein weiterer Vorteil liegt darin, dass sie auch eine Verbesserung zum Verständnis der Struktur von OpenGL-Szenen mit sich bringen.

Eine solche Display List lässt sich wie folgt erzeugen und laden:

```
glNewList(GLint name, GLenum mode);
     // Primitive definieren, Transformationen
glEndList();
. . .
```

Eine Display List wird ähnlich zu den Primitivdefinitionen innerhalb der Methoden `glNewList()` und `glEndList()` definiert. Der erste Parameter der glNewList-

Funktion speichert einen eindeutigen Identifikator für die Liste, anhand dessen die Liste später geladen werden kann. Dieser kann manuell vergeben werden oder aber man lässt sich durch OpenGL mittels der Funktion

```
GLint name = glGenLists(1);
```

einen Identifikator ermitteln und zurückgeben. Über den zweiten Parameter, *mode*, kann man einstellen, ob die Liste vorkompiliert und sofort zur Ausführung gebracht (GL_COMPILE_AND_EXECUTE) oder nur kompiliert und im Speicher gehalten werden soll (GL_COMPILE). Eine Liste kann jederzeit über

```
glCallList(GLint name);
```

ausgeführt werden.

5.1.4 Push und Pop

Zu Beginn dieses Kapitels wurde der Begriff Matrizenstapel in Verbindung mit den Matrizen schon einmal kurz erwähnt. In Wirklichkeit kann es in einem OpenGL-System nicht nur eine Modelview- bzw. Projection-Matrix geben, sondern jeweils einen ganzen Stapel. Zur Verwaltung von Matrizenstapeln dient ein Stack, auf den man die aktuelle Matrix mit der Funktion `glPushMatrix()` legen kann und später bei Bedarf diese wieder vom Stack mit der Funktion `glPopMatrix()` holen kann. So ist es möglich, Zustände von Szenen temporär zu speichern und diese bei Bedarf später wieder her zu stellen.

5.2 Modelltransformation

Unter dem Begriff Modelltransformationen fasst man alle Operationen zusammen, die die Lage und Erscheinung von Primitiven in der OpenGL-Szene verändern. Zu diesen Operationen gehören, die Translation, Skalierung, Spiegelung, Rotation und Scherung. Für Modelltransformationen muss GL_MODELVIEW die aktuelle Matrix sein.

Translation

Die Translation angewendet auf ein Primitiv verschiebt dessen Vertices um den (Translations-) Vektor (x,y,z). Die dazugehörige OpenGL-Direktive lautet

```
glTranslate3*(x,y,z)
```

Skalierung

Mittels der Skalierung ändern alle begrenzenden Vertices eines Objektes ihre Abstände bezüglich des Koordinatenursprungs

```
glScale3*(x,y,z)
```

Spiegelung

Die Spiegelung ist ein Spezialfall der Skalierung mit negativen Skalierungsfaktoren. Möchte man zum Beispiel ein Objekt an der yz-Ebene spiegeln, übergibt man der Funktion `glScale` einen negativen x-Wert.

Rotation

Die Rotation lässt ein Objekt um eine oder mehrere Koordinatenachsen drehen. Mit dem Parameter *grad* von

```
glRotate*(grad,x,y,z)
```

lässt sich zudem der Drehwinkel einstellen. Das Drehen eines Objektes 30° um die x-Achse würde wie folgt erreicht werden: glRotate3d(30.0,1.0,0.0,0.0)

Scherung

Die letzte mögliche Modelltransformation ist die Scherung. Durch sie wird ein Punkt parallel zu den Ebenen xy, yz oder xz verschoben. Dazu wird jeder Vertex mit der Scherungsmatrix

$$\begin{bmatrix} 1 & s_1 & s_2 & 0 \\ s_3 & 1 & s_4 & 0 \\ s_5 & s_6 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad | \; s_1 \ldots s_6\text{: Scherungsfaktoren}$$

multipliziert.

Die einzelnen Transformationen können innerhalb und außerhalb des Primitivdefinitionsbereiches erfolgen. Legt man mehrere Transformationen hintereinander fest, muss man beachten, dass diese in umgekehrter Reihenfolge ausgeführt werden.

5.3 Ansichtstransformation

Ansichtstransformationen unter OpenGL legen die Position, die Blickrichtung und den Blickwinkel des Betrachters fest. Die standardmäßig eingestellte Blickrichtung ist in

Richtung der negativen z-Achse.
Um die Kamera positionieren zu können, muss zunächst sichergestellt sein, dass GL_MODELVIEW aktiv ist. Danach kann man mit der Funktion

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

)
```

eyeX, eyeY, eyeZ:	Position des Betrachters
centerX, centerY, centerZ:	Position des Referenzpunktes, auf den geblickt wird
upX, upY, upZ:	Up-Vektor, steht senkrecht auf der Kamera; Drehwinkel

die Kamera positionieren und ausrichten.

5.4 Projektionsarten

Es gibt unter OpenGL zwei verschiedene Projektionsarten, die orthogonale und die perspektivische Projektion. Die Projektion nutzt ein Sichtvolumen, das alle Objekte, die sich innerhalb dieses Bereiches befinden, darstellt, die übrigen Objekte außerhalb werden abgeschnitten und nicht auf dem Bildschirm dargestellt. Dieses Verfahren nennt man Clipping. Das Sichtvolumen wird von sechs Clipping-Planes begrenzt.

Die orthogonale Projektion hat keine Tiefenwirkung, dass bedeutet das Objekte, die weiter hinten liegen entsprechend ihrer definierten Größe abgebildet werden. Im Gegensatz dazu, werden bei der perspektivischen Projektion, die Tiefenwirkung verwendet, diese Objekte im Hintergrund kleiner dargestellt.

Zum Einstellen der Projektion stellt OpenGL zwei Funktionen bereit, die sich in ihrem Aufbau aber nicht unterscheiden.

Die Funktion

```
glOrtho(
     GLdouble left, GLdouble right,
     GLdouble bottom, GLdouble top,
     GLdouble near, GLdouble far
)
```

aktiviert die orthogonale Projektion. Die Parameter stellen entsprechend ihres Namens die Position der Clipping-Planes und bezug auf den Koordinatenursprung dar.

Analog dazu lautet die Funktion für die perspektivische Projektion

```
glFrustum(
     GLdouble left, GLdouble right,
     GLdouble bottom, GLdouble top,
     GLdouble near, GLdouble far
)
```

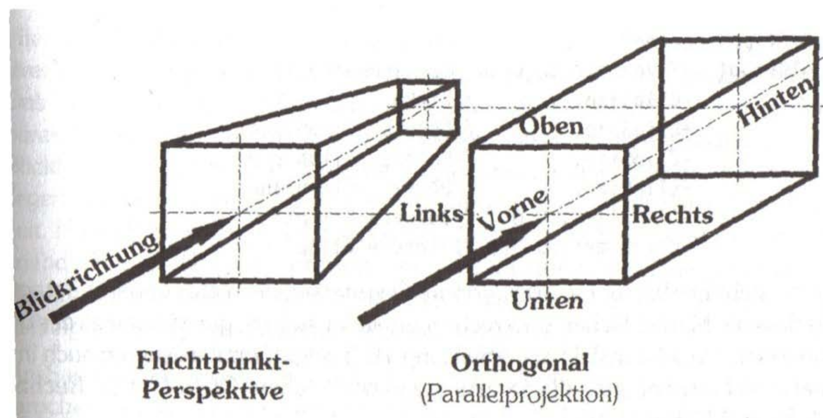

Abbildung 5.2: Orthogonale und Perspektivische Projektion

5.5 Viewport

Die letzte Station der Viewing-Pipeline ist der Viewport. Als Viewport bezeichnet man den Teil eines Fensters des Betriebssystems, der die OpenGL-Szene darstellt, was maximal der gesamte Bildschirm sein kann. Das bedeutet auch, das der Viewport nicht das gesamte Fenster ausfüllen muss.

Mit dem Befehl

```
glViewport(
     (GLint) x, (GLint) y,
     (GLsizei) width, (GLsizei) height
)
```

werden die durch die perspektivische bzw. orthogonale Projektion erzeugten normierten Gerätekoordinaten in Bildschirmkoordinaten transformiert. Die ersten beiden Parameter stellen dabei die Koordinatenwerte der unteren linken Ecke des

Client-Bereiches, also des Fensters, dar. Über die letzten beiden Parameter werden die Breite und die Höhe des Viewports festgelegt.

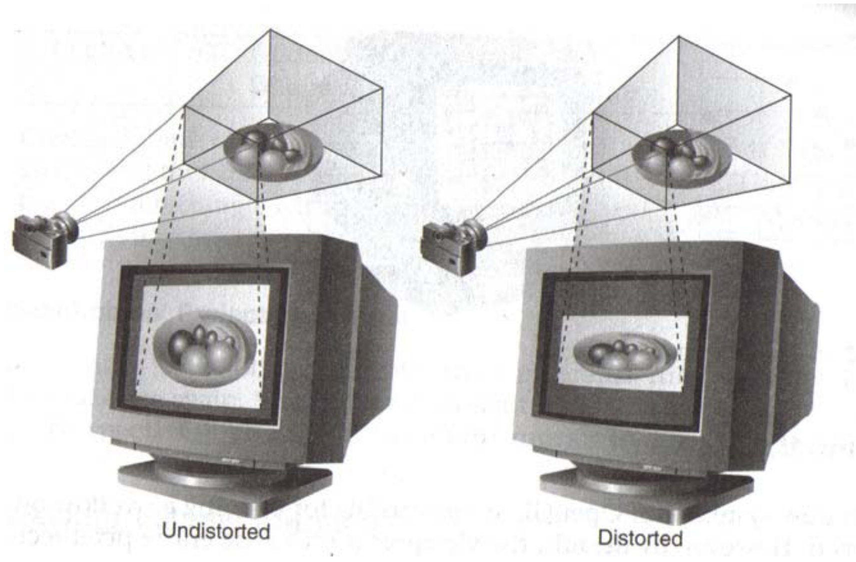

Abbildung 5.3: unverzerrter und verzerrter Viewport

6 Quellenangabe

Literatur

Burggraf: *OpenGL – Der einfache Einstieg in die Schnittstellenprogrammierung*
Markt+Technik Verlag

Shreiner, Woo, Neider, Davis: *OpenGL Programming Guide (6th Ed.)*
Addison-Wesley

Orlamünder, Mascolus: *Computergrafik und OpenGL*
Fachbuchverlag Leipzig

Abbildungen

2.1 Grafikprimitive in OpenGL:
http://graphics.cs.uni-sb.de/Courses/ws9900/cg-seminar/Ausarbeitung/Philipp.Walter/images/primitive.gif

4.2 Lichtquellentypen:
OpenGL – Der einfache Einstieg in die Schnittstellenprogrammierung (s. Literatur)

5.1 Viewing Pipeline:
nach Vorlage von *Computergrafik und OpenGL (s. Literatur)*

5.2 Orthogonale und Perspektivische Projektionen:
OpenGL Programming Guide (s. Literatur)

5.3 unverzerrter und verzerrter Viewport:
OpenGL Programming Guide (s. Literatur)