Wolfgang Kowalk

OpenGL™

lernen mit GL_Sourcerer

1 Vorwort

OpenGL ist ein Standard zur Programmierung von 3D-Graphik auf modernen Computersystemen. Heute besitzen fast alle Computer entsprechende Hard- und Software, so dass jeder Besitzer eines Computers, der wenigstens etwas von Programmierung versteht, 3D-Graphik erzeugen kann. Dennoch ist dieses ein anspruchsvolles Thema, da dieses zugleich stets mit dem aktuellen technisches Fortschritt in der Computertechnik verbunden ist.

Dieses Buch soll auf besondere Weise in dieses Thema einführen. Zum einen wollen wir uns von den Beschränkungen der aktuellen Programmiersprachen und Betriebssysteme befreien, so dass wir auf jedem Rechensystem, sei es ein PC oder ein Mac, ein Windows-, Linux- oder Unix-Betriebssystem, unsere Programme ausführen können. Zum anderen wollen wir eine neue Sprache, den GL_Sourcerer verwenden, der die an und für sich recht mühselige Programmierung von OpenGL-Programmen vereinfachen soll, so dass dieses quasi von selbst läuft. Als Programmiersprache wählen wir Java, da dieses eine sehr hohe Verbreitung hat, und auf allen Rechensystemen läuft, ohne große Abhängigkeit von den Betriebssystemen oder der Hardware.

Inhaltsverzeichnis

1 Vorwort....3
2 Einleitung....9
3 Installation....11
3.1 Installation von Java....11
3.1.1 Testen von Java....11
3.2 Installation von OpenGL....11
3.3 Installation von Jogl....11
3.4 Installation des GL_Sourcerer....12
3.4.1 Testen des GL_Sourcerers....12
4 Grundlagen....13
4.1 Ziele des GL_Sourcerers....13
4.2 OpenGL aus Sicht von GL_Sourcerer....13
4.3 OpenGL-Befehle....17
4.3.1 Die Klasse GL....19
4.3.2 Display-Liste....19
4.3.3 Attribut-Werte....21
4.3.4 OpenGL-Primitive....21
4.3.5 Koordinatensystem....23
4.3.6 Raumkoordinaten....25
4.3.7 Transformation und Rotation....25
4.3.8 Attribut-Werte für Primitive in GL_Sourcerer und OpenGL....29
4.3.9 Primitive in GL_Sourcerer und OpenGL....31
Dreieck (Triangles)....33
Vierecke (Quads)....35
Polygon....37
Tessallation....39
Punkte und Linien....39
Beziér-Kurven und -Flächen....41
Schrift....43
4.3.10 Vorder- und Rückseite....43
4.3.11 Geometrie und Farbe....45
Kreis und Ellipse....45
Quader und schiefe Ebene....47
Kugel und Ellipsoid....49
Kegelstumpf (cone)....51
Rohr (pipe)....51
Zahnrad (gear)....51
Heightmap....53
Skybox....53
4.3.12 Objekte aus Standardbibliotheken....55
Quadric....55
GlutObjekte....57
5 Beleuchtung 61
5.1 Lichtmodelle in OpenGL....61
5.1.1 Ambientes Licht....63
5.1.2 Diffuses Licht....63
5.1.3 Speculares Licht....63
5.1.4 Die Lichtformel....65

5.1.5 Lichtdämpfung bei positionellem Licht....65
5.1.6 Direktionelles Licht....67
5.1.7 Scheinwerferkegel (spot)....67
5.1.8 Globales ambientes Licht....69
5.1.9 Emissives Licht....69
5.2 Licht in OpenGL....71
5.2.1 Beleuchtung von Flächen....73
5.3 Hinweise zur Beleuchtung....73
6 Texturen....77
6.1 Texturen auf Flächen....77
6.1.1 Textur-Koordinaten....79
6.1.2 Automatische Berechnung von Textur-Koordinaten....79
6.1.3 Andere Berechnungen von Textur-Koordinaten....81
6.2 Texturen im Raum....83
7 Shader....87
7.1 Die OpenGL-Pipeline....87
7.2 Die Shader-Programmiersprache....87
7.2.1 Beispiel für Shader-Programme....87
7.2.2 Datentypen....91
7.2.3 Konstruktoren....93
7.2.4 Qualifier....95
7.2.5 Operatoren....97
7.2.6 Eingebaut Funktionen....97
Trigonometrische Funktionen....97
Exponentialfunktionen....97
Allgemeine Funktionen....99
Geometrische Funktionen....99
Matrix-Funktionen....99
Vektor-Funktionen....99
Fragment-Funktionen....99
Sampler-Funktionen....101
Noise-Funktionen....101
7.3 Vertex-Shader....101
7.3.1 Die Variable gl_Position....101
7.4 Fragment-Shader....103
7.4.1 Varying-Variablen und interpolierte Werte....103
7.4.2 Eingebaute Varying-Variablen....103
7.4.3 Die Variable gl_FragColor....105
7.4.4 Weitere eingebaute Varying-Variablen....105
7.5 Shader-Programme in GL_Sourcerer....105
7.5.1 Struktur von Shadern in GL_Sourcerer....105
7.5.2 Uniform-Variablen....107
Float-Zahlen, float-Vektoren und Texturen....107
Felder (arrayUniform)....109
Diskrete Werte mittels Tasten (keyUniform)....109
Analoge Werte mittels Maus (mouse)....109
Zeitwerte mittels time....111
7.5.3 Attribute-Variablen....111
7.6 Einfache Shader-Programme....113
7.6.1 Das einfachste Shader-Programm....113
7.6.2 Ändern der Farben in Shadern....115
7.6.3 Farbberechnungen im Vertex-Shader....115

7.6.4 Farbberechnungen im Fragment-Shader....117
7.6.5 Berechnung von Raumkoordinaten im Vertex-Shader....121
7.6.6 Berechnung von Raumkoordinaten im Fragment-Shader....121
7.7 Lichtberechnung mit Shadern....129
7.7.1 Lichtparameter....129
7.7.2 Ambientens Licht....131
7.7.3 Diffuses positionelles Licht....135
7.7.4 Diffuses direktionelles Licht....137
7.7.5 Speculares Licht....139
7.7.6 Spotlight....141
7.7.7 Kombinationen der Lichtquellen....145
7.8 Texturen mit Shadern....145
7.8.1 Einfache Texturen....145
7.8.2 Übergabe von Texturen an Shader....149
7.8.3 Berechnung von Texturen....151
7.8.4 Mehrere Texturen....151
7.9 Spezielle Texturen....151
7.9.1 Bump-Mapping mit Shadern....153

2 Einleitung

OpenGL ist ein Standard zur Programmierung von 3D-Graphik auf modernen Computersystemen. Heute besitzen fast alle Computer entsprechende Hard- und Software, so dass jeder Besitzer eines Computers, der wenigstens etwas von Programmierung versteht, 3D-Graphik erzeugen kann. Dennoch ist dieses ein anspruchsvolles Thema, da dieses zugleich immer mit dem aktuellen technisches Fortschritt in der Computertechnik verbunden ist.

Dieses Buch soll auf besondere Weise in dieses Thema einführen. Zum einen wollen wir uns von den Beschränkungen der aktuellen Programmiersprachen und Betriebssysteme befreien, so dass wir auf jedem Rechensystem, sei es ein PC oder ein Mac, ein Windows-, Linux- oder Unix-Betriebssystem, unsere Programme ausführen können. Zum anderen wollen wir eine neue Sprache, den GL_Sourcerer verwenden, der die an und für sich recht mühselige Programmierung von OpenGL-Programmen vereinfachen soll, so dass dieses quasi von selbst läuft. Als Programmiersprache wählen wir Java, da dieses eine sehr hohe Verbreitung hat, und auf allen Rechensystemen läuft, ohne große Abhängigkeit von den Plattformen oder Betriebssystemen zu zeigen. Darüber hinaus lassen sich mit Java sehr schnelle und sichere Programme schreiben, so dass viele alte Vorurteile gegen Java heute keine Rechtefertigung mehr besitzen.

3 Installation

In diesem Kapitel werden die grundlegenden technischen Installationen von Java, OpenGL, Jogl und GL_Sourcerer beschrieben.

3.1 Installation von Java

Java ist eine moderne, objektorientierte Programmiersprache, die heutzutage ebenso schnell wie andere moderne Sprachen ist, wie beispielsweise C. Sie ist daher auch für Anwendungen mit hoher Leistungsanforderung geeignet.

Um Java zu installieren, ist für das jeweilige System die jeweilige Installationsbeschreibung zu beachten. Wir gehen davon aus, dass Java so installiert wurde, dass ein Java-Archiv durch direkten Aufruf, z.B. durch Doppelklick in Windows- oder Mac-OSX-Betriebssystemen, ausgeführt wird.

3.1.1 Testen von Java

Um Java laufen zu lassen, muss ...

3.2 Installation von OpenGL

OpenGL ist eine Schnittstelle, die vom Hersteller des Computers bzw. Betriebssystem zur Verfügung gestellt wird, um die Graphikkarte des Computers anzusteuern. In den meisten Betriebssystemen ist eine aktuelle Implementierung von OpenGL vorhanden, jedoch ändern sich deren Versionen gelegentlich bzw. werden Upgrades nötig, da OpenGL-Versionen sich ändern oder einfach Fehler beseitigt werden. In der Regel sollte man immer die aktuelle Version von OpenGL verwenden, die entweder von der OpenGL-

3.3 Installation von Jogl

Jogl ist eine Schnittstelle von Java an OpenGL, und darüber hinaus an die Graphikkarte. Durch eine automatische Erzeugung dieser Schnittstelle aus der OpenGL-Sprachbeschreibung wird heute stets eine aktuelle Version der jeweiligen OpenGL-Schnittstelle in Java zur Verfügung gestellt. Diese Schnittstelle ist erforderlich, um den GL-Sourcerer laufen zu lassen. Sollte man sich entscheiden, in

anderen Programmiersprachen zu arbeiten, so ist deren Schnittstelle ebenfalls zur Verfügung zu stellen.

Um Jogl zu installieren,

3.4 Installation des GL_Sourcerer

Der GL_Sourcerer ist eine einfache Hochsprache, in der leicht elementare graphische Objekte erstellt werden können. Gleichzeitig erzeugt GL_Sourcerer die entsprechenden OpenGL-Befehle in der jeweiligen OpenGL-Syntax, so dass die getesteten Programme durch Copy-und-Paste in die eigentliche Anwendung übernommen werden können. Dadurch wird bei der Entwicklung viel Zeit gespart, da man sich auf das Ergebnis konzentrieren kann, weil die vielen syntaktischen Feinheiten von OpenGL automatisch generiert werden.

Um GL_Sourcerer zu installieren, verwende man eine eigene Directory, in welche das Java-Archiv **`glsourcerer.jar`** kopiert wird. Man sollte sich am einfachsten in dieser Directory zwei Ordner anlegen, einen **`data`**-Ordner, in welche Recourcen wie Bilder untergebracht werden können, und einen **`source`**-Ordner, in welchen die GL_Sourcerer-Programm abgelegt werden können. Weitere Ordner, z.B. für Funktionen, können jederzeit bei Bedarf hinzugefügt werden.

GL_Sourcerer erzeugt in der Regel eine Datei, in welche Zustandsinformation abgelegt wird. Wenn diese gelöscht wird, startet das System wieder wie zu Anfang, ansonsten wird beim Starten das zuletzt verwendete Programm geladen und ausgeführt.

3.4.1 Testen des GL_Sourcerers

Um den GL_Sourcerer zu testen, verwende man eine einfache Text-Datei, deren Name am besten standardmäßig die Endung .gls hat, z.B. **`Cube.gls`**. Dieses sollte eine einfache Textdatei sein, z.B. mit WordPad oder TextWrangler erstellt. In diese Datei fügen Sie den folgenden Text ein und speichern diese Datei an einer geeigneten Stelle, z.B. im **`source`**-Ordner Ihrer GL_Sourcerer-Directory (Sie müssen die Datei dazu nicht schließen! Save as ... reicht aus).

4 Grundlagen

4.1 Ziele des GL_Sourcerers

Das Programm *GL_Sourcerer* soll mehreren Zwecke erfüllen.

Zum einen dient *GL_Sourcerer* dazu, schnell und ohne großen 'Overhead' OpenGL-Programme zu erstellen und zu testen. Dadurch soll der Aufwand zum Erstellen von Graphik-Software verringert werden.

Da *GL_Sourcerer* auch verwendet werden kann, um OpenGL-Code zu erzeugen, lassen sich hiermit OpenGL-Programme sehr viel einfacher und effektiver erstellen als von Hand, da es nur wenig Kopierarbeit bedarf, um OpenGL-Sourcecode in eine Applikation – auch in anderen Sprachen wie C oder C++ – einzufügen.

Ein weiterer Zweck ist es, OpenGL im Unterricht zu verwenden, um schnell Beispiele zu erstellen und die Eigenschaften von OpenGL zu demonstrieren. Hierfür hat sich *GL_Sourcerer* bereits vielfach gut bewährt, da schnell Änderungen vorgeführt und anschaulich demonstriert werden können.

Wir beginnen zunächst einmal, uns mit der Schnittstelle eines *GL_Sourcerer*-Programms vertraut zu machen und zeigen dieses konkret an einem Beispiel.

4.2 OpenGL aus Sicht von GL_Sourcerer

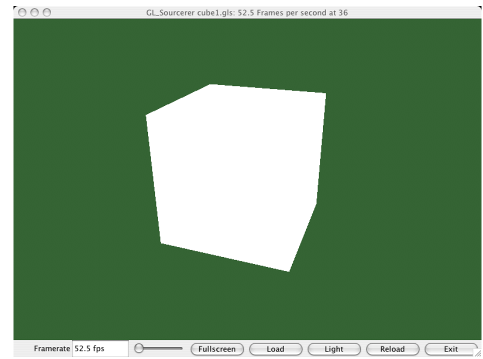
Wir beginnen mit einem sehr einfachen Beispiel.

Erzeugen Sie eine Datei, deren Name am besten standardmäßig die Endung **`.gls`** hat, z.B. **`Cube.gls`**. Dieses sollte eine einfache Textdatei sein, z.B. mit WordPad oder TextWrangler erstellt. In diese Datei fügen Sie den folgenden Text ein und speichern diese Datei an einer geeigneten Stelle, z.B. im **`source`**-Ordner Ihrer GL_Sourcerer-Directory (Sie müssen die Datei dazu nicht schließen! *Save as* ... reicht aus).

```
< Cube
>
```

Wichtig ist, dass die zweite Zeile als einziges Zeichen > enthält. Starten Sie jetzt den *GL_Sourcerer* und laden Sie (**`load`**) die Datei **`Cube.gls`**. Es sollte sich dann ein Bild der nebenstehenden Art ergeben.

Hier haben Sie einen Würfel der Kantenlänge eins erzeugt. Der Würfel ist vollständig weiß, und lässt sich daher nur schlecht als Würfel erkennen. Man kann ihn jedoch drehen, indem man mit der linken Maustaste einmal ins Fenster klickt (um dieses zu aktivieren). Danach kann man durch Drücken der linken Maustaste und Verschieben der Maus die Ansicht um einen Mittelpunkt drehen. Durch Drücken der rechten Maustaste lässt sich das Objekt mittels Verschieben der Maus nach links/rechts bzw. oben/unten verschieben. Wird die mittlere Maustaste gedrückt, so lässt sich das Objekt nach vorne oder hinten verschieben; durch Drehen des Mausrads lässt sich dieses ebenfalls ebenfalls erreichen.

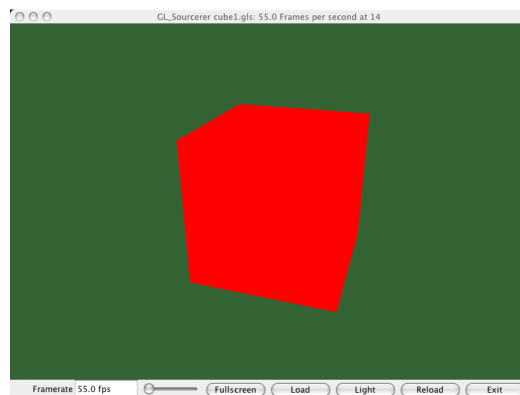


Durch die Bewegung erhält man den Eindruck eines Würfels. Allerdings würde eine unterschiedliche Färbung der Seiten des Würfels dessen Form noch deutlicher werden lassen. Daher fügen wir in dem Befehl **`cube`** noch *Parameter* ein,

```
< Cube
  Color 1 0 0
>
```

Der Bezeichner hinter < (hier **`Cube`**) wird als *Befehlsname* bezeichnet. Der Befehl steht auf der ersten Zeile und hat meist keinen Parameter (es gibt hier einige Ausnahmen). Es sei hier erwähnt, dass GL_Sourcerer in den meisten Fällen nicht zwischen Groß- und Kleinschreibung unterscheidet, so dass die obigen Wörter auch **`cube`**, **`cUbE`** oder **`color`** hätten geschrieben werden können. In der Regel verwenden wir die Schreibung mit großem Anfangsbuchstaben. Werden Dateinamen angegeben oder Shader-Programme, so ist in der Regel auf Groß- und Kleinschreibung zu achten!

Alle folgenden Zeilen enthalten *Parameter* zu diesem Befehl, die den Befehl variieren. In diesem Fall (**`Color`**) haben wir eine Farbe angegeben. Farben werden durch ihren roten, grünen und blauen Farbanteil spezifiziert, in diesem Fall durch den roten Anteil 1, und keinen grünen oder blauen Anteil (jeweils 0). Der Würfel erscheint dann also rot.

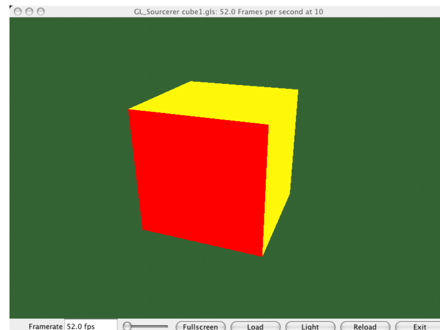


Die Parameter zu einem Befehl werden stets auf eine eigene Zeile geschrieben; sie beginnen mit dem *Parameternamen* und enthalten meistens *Parameterwerte*, hier die Farbanteile. Diese Werte werden immer getrennt durch mindestens ein Leerzeichen hinter den Parameternamen geschrieben.

Die Angabe der gleichen Farbe für alle Flächen löst natürlich nicht das Problem der deutlicheren Erkennbarkeit. Daher müssen wir die Seiten des Würfels unterschiedlich färben. Dieses lässt sich mit dem Befehl

```
< Cube
  ColorFront 1 0 0
  ColorBack 1 1 0
>
```

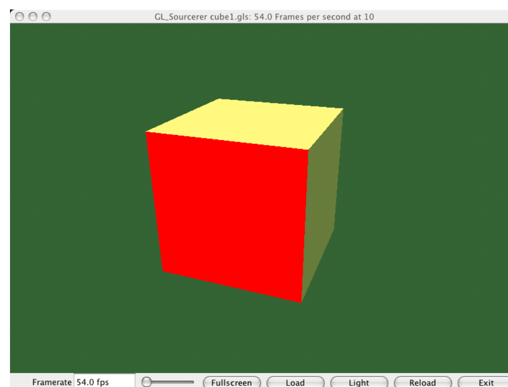
erreichen.



Um jede Fläche mit einer anderen Farbe zu versehen, sind entsprechende Parameter zu setzen:

```
< Cube
  ColorFront 1 0 0
  ColorBack 1 1 0
  ColorUp 1 1 0.5
  ColorDown 1 0 1
  ColorLeft 0 1 1
  ColorRight 0.4 0.5 0.23
>
```

Das Ergebnis sieht dann folgendermaßen aus:



Mit diesen farblichen Änderungen ist das Objekt wesentlich einfacher als Würfel zu erkennen. Auch wenn dem Objekt noch einiges an Natürlichkeit fehlt, bleibt dennoch die Frage, wie dieses Ergebnis letztendlich zustande kommt. Dazu ist die Schnittstelle zu OpenGL zu betrachten.

4.3 OpenGL-Befehle

Der GL_Sourcerer kann die OpenGL-Befehle erzeugen, die für das Anzeigen eines Objekts benötigt werden. Diese Befehle werden in eine externe Datei gespeichert, die jeweils anzugeben ist. Der allgemeine Befehl für die Ausgabe von OpenGL-Befehlen lautet

```
< Output true
  prefix gl. GL.
```

```
    Write // Draw a cube
    file output.txt
  >
```

Um die Daten in der angegebenen Datei `output.txt` zu speichern, muss die Ausgabe wieder beendet werden. Dazu ist der Befehl

```
  < Output false
  >
```

anzugeben. Man beachte, dass die Ausgabe nur zwischen diesen beiden Befehlen erfolgt. Man kann auf diese Weise genau die Objekte, deren OpenGL-Befehle interessieren, spezifizieren.

Die Ausgabe für dieses `Cube`-Beispiel ist relativ lang und soll hier nur in Ausschnitten wiedergegeben werden.

```
// Draw a cube
  gl.glGenLists(1);
  gl.glNewList(1, GL.GL_COMPILE);
    gl.glColor4f(1.0f,0.0f,0.0f, 1.0f);
    gl.glNormal3f(0.0f,0.0f,1.0f);
    gl.glBegin(GL.GL_QUAD_STRIP);
      gl.glVertex3f(-0.5f,0.5f,0.5f);
      gl.glVertex3f(-0.5f,-0.5f,0.5f);
      gl.glVertex3f(0.5f,0.5f,0.5f);
      gl.glVertex3f(0.5f,-0.5f,0.5f);
    gl.glEnd();
    ...
  gl.glEndList();
```

Die hier angegebenen Befehle werden in dieser Form in Java mit der Schnittstelle Jogl verwendet.

4.3.1 Die Klasse GL

`gl` ist Instanz der Klasse `GL`, in welcher OpenGL-Konstante bzw. OpenGL-Schnittstellen-Funktionen definiert sind. Der Parameter

```
prefix gl. GL.
```

in dem `Output`-Befehl legt diese Art der Referenzierung fest. Sollen andere Präfixe verwendet werden, z.B. um die OpenGL-Befehle in C-Programmen zu verwenden, so kann dieses entsprechend eingestellt werden. Da C keine Präfixe verwendet, wird für die Verwendung in C-Programmen der `prefix`-Parameter ohne Argumente angegeben.

```
Prefix
```

Die OpenGL-Befehle haben die folgende Bedeutung.

4.3.2 Display-Liste

Die ersten beiden und der letzte Befehl gehören zusammen und erzeugen eine sogenannte *Display*-Liste. Durch den ersten Befehl `GenLists(1)`[1] wird ein *Handle* erzeugt, über den eine Display-Liste referenziert werden kann; der Handle ist stets eine ganze Zahl; der Parameter gibt die Anzahl an aufeinanderfolgenden Handles an (z.B. 3, mit den Handles 6,7 und 8; es wird 6 zurückgegeben, und der Programmierer weiß dann, dass er 6, 7 und 8 verwenden kann). Das Befehlspaar

1 Wie lassen die immer gleichen Präfixe `gl.gl` bzw. `GL.GL_` im laufenden Text normalerweise weg

```
NewList(1, GL.GL_COMPILE);
...
gl.glEndList();
```

schreibt die dazwischen stehenden OpenGL-Befehle in eine Display-Liste mit dem Handle 1, der als Wert des ersten Funktionsaufrufs bestimmt wurde. Alle zwischen diesen beiden Befehlen stehenden OpenGL-Anweisungen werden gespeichert. Danach können diese durch einen einzigen OpenGL-Befehl

```
gl.glCallList(1);
```

aufgerufen werden. Der Parameter ist natürlich der Handle, hier also 1. Da diese OpenGL-Befehle in der Regel im Speicher der Graphikkarte abgelegt werden und zusätzlich bei der Erzeugung optimiert werden können, stellt dieses eine sehr effiziente Möglichkeit dar, insbesondere aufwändige OpenGL-Programme sowohl Speicherplatz-sparend als auch Laufzeit-effizient zu verwenden. Der GL_Sourcerer erzeugt, soweit wie möglich, immer Display-Listen und startet alle Display-Listen im optimalen Fall durch einen einzigen Befehlsaufruf.

4.3.3 Attribut-Werte

OpenGL betrachtet die Graphikmaschine als Zustandsautomaten. Dies bedeutet, dass zu einem Befehl, der eine graphische Ausgabe bewirkt, sämtliche zugehörigen Parameter vorher einzustellen sind. Der eigentliche Befehlsaufruf ist üblicherweise immer der Befehl `Vertex3f(..)`, den wir im Abschnitt 4.3.6 auf Seite 17 besprechen.

Die Einstellungen des Zustandsautomaten erfolgen durch Befehlsaufrufe. Der Befehl

```
gl.glColor4f( 1.0f, 0.0f, 0.0f, 1.0f);
```

stellt die Farbe ein, in der eine Fläche gezeichnet werden soll. Die ersten drei Parameter sind die Farbanteile für rot, grün und blau, jeweils im Bereich von 0 bis 1. Werden größere oder kleinere Werte angegeben, so werden diese in der Regel abgeschnitten. Der letzte Parameter dieses Befehls gibt die Transparenz an und sollte zunächst immer auf 1 gesetzt werden. Man beachte, dass die Programmiersprachen Wert auf den Typ der jeweiligen Variablen legen, also Float-Zahlen. Sowohl in C als auch in Java muss in jedem Fall dass f hinter einer Gleitpunktzahl-Konstante geschrieben werden, da der Standard diese sonst als Double-Zahlen interpretiert.

Der zweite Attributwert-Befehl lautet

```
gl.glNormal3f(0.0f,0.0f,1.0f);
```

und gibt die Orientierung der Oberfläche, d.h. die Senkrechte zur Oberfläche an. Die drei Zahlenwerte werden als Richtungsvektor interpretiert. In der Regel sollte dieser Vektor die Länge 1 haben. Die Normalen haben nur eine konkrete Bedeutung, wenn Lichtberechnung durchgeführt wird, was in unserem Beispiel bisher noch nicht vorkam. Dieses wird weiter unten betrachtet.

Man sieht hieran, dass der GL_Sourcerer gelegentlich überflüssige Befehle erzeugt. Dieses lässt sich wegen des Zustandsautomaten grundsätzlich nicht immer vermeiden, so dass im folgenden solche Befehle übergangen werden. Sollte der Sourcecode in einer Anwendung eingesetzt werden, so können diese Befehle gelöscht werden.

4.3.4 OpenGL-Primitive

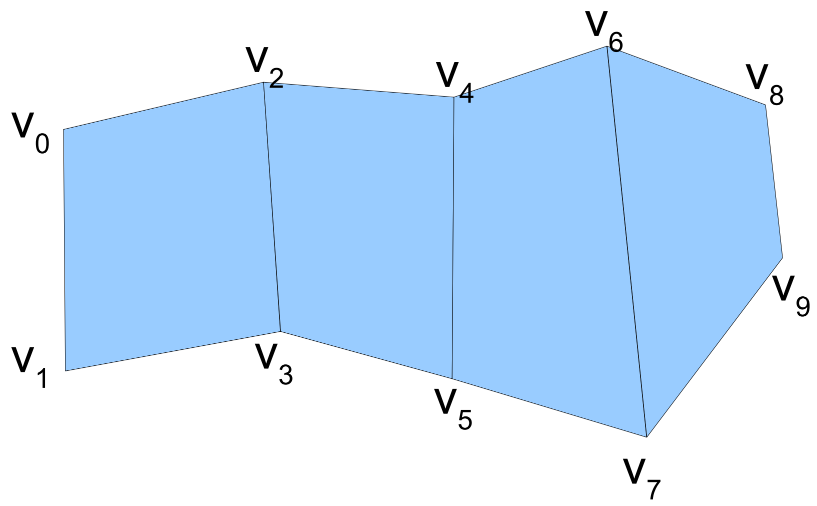
Um Flächen zu zeichnen, müssen die Eckpunkte der Flächen festgelegt werden. Außerdem hat jede Fläche eine Vorder- und eine Rückseite. Diese werden von OpenGL genau spezifiziert, so dass sich der Programmierer damit auskennen muss.

OpenGL erlaubt es, Dreicke, (Triangles), Vierecke (Quads), sowie zusammenhängende Dreiecke bzw. Vierecke zu spezifizieren. Außerdem können noch allgemeine Polygone verwendet werden. Die Reihenfolge, in der die Punkte einer solchen Fläche angegeben werden, legt die Vorderseite fest. Werden bei der Draufsicht auf die Fläche die Punkte entgegen der Uhrzeigerrichtung (*counter-clock wise*) angegeben, so wird diese sichtbare Seite der Fläche als Vorderseite, die andere als Rückseite bezeichnet. OpenGL erlaubt es, diese Definition umzustellen, was aber nur in speziellen Anwendungen (z.B. gespiegelten Objekten) sinnvoll ist.

Der Typ der Fläche wird in OpenGL durch die Befehle

```
gl.glBegin(GL.GL_QUAD_STRIP);
...
gl.glEnd();
```

festgelegt, wobei statt **QUAD_STRIP** eine der Konstanten **QUADS**, **QUAD_STRIP**, **TRIANGLES**, **TRIANGLE_STRIP**, **TRIANGLE_FAN**, **POLGON**, **LINES**, **LINE_STRIP**, **LINE_LOOP** oder **POINTS** stehen kann. Im GL_Sourcerer werden die Befehle **polygon**, **points**, **lines**, **linestrip**, **lineloop**, **quads**, **quadstrip**, **triangles**, **trianglestrip**, **trianglefan** verwendet, mit der gleichen Bedeutung; derartige Befehle werden unten in Abschnitt 4.3.9 auf Seite 21 behandelt.



Im Beispiel **QUAD_STRIP** werden vier oder mehr Punkte zwischen **Begin** und **End** erwartet, welche zu Flächen verbunden werden. Die Reihenfolge der Punkte ist auch hier wichtig, da sie die Vorderseite der Fläche bestimmt. Das Bild zeigt anhand von zehn Punkten, in welcher Reihenfolge die Eckpunkte anzugeben sind, damit die Fläche korrekt gezeichnet wird.

4.3.5 Koordinatensystem

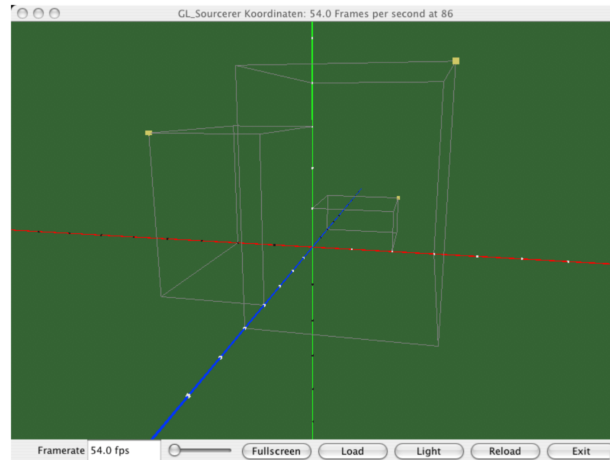
Um die Formen der Flächen zu bestimmen, werden deren Raumkoordinaten angegeben. OpenGL verwendet ein kartesisches Koordinatensystem, deren Koordinaten als x, y und z bezeichnet werden. Bei Koordinatensystemen ist die Orientierung wichtig, wobei OpenGL ein rechtsorientiertes System verwendet.

Um ein Koordinatensystem zu zeichnen kann im GL_Sourcerer der Befehl

```
< Origin 1
  Size 20 0.02 0.02
  Step 2 2 2
  Point -2 3 4
  Point 3 4 5
  Point 2 1 -2
>
```

verwendet werden. Dieser Befehl zeichnet ein Koordinatensystem, wobei der erste Parameter hinter dem Befehlsnamen `Origin` den Abstand zweier Markierungen auf den Achsen angibt, hier also 1.

Mit `Size` werden Länge (20 Einheiten), Breite und Höhe der Koordinaten angegeben. Außerdem können mit `Point` Raumkoordinaten eingezeichnet werden. Der obige Befehl erzeugt das folgende Bild.



In dem Bild sieht man die horizontale x-Achse, die rot gezeichnet ist. Die Punkte auf der positiven x-Achse im Abstand 1 sind weiß markiert, auf der negativen schwarz. Entsprechend sind senkrecht die y-Achse grün und in die Tiefe des Bildes die z-Achse blau gezeichnet (d.h. x=rot, y=grün, z=blau).

Um einen Punkt festzulegen sind die drei Koordinaten des Punkts anzugeben, z.B. `Point 3 4 5` für den Punkt vorne rechts oben: (x=3,y=4,z=5); man schreibt in der Mathematik meistens nur (3,4,5), da die Reihenfolge, in der die Punkte aufgetragen werden, festgelegt ist. Wenn sich ein Leser mit Koordinatensystemen noch nicht gut auskennt, sollte er den Befehl selbst ausführen und die Koordinaten der Punkte variieren. Dann kann er selbst überprüfen, ob die Ergebnisse seinen Vorstellungen entsprechen. Um geometrische Objekte zu erstellen, ist die Kenntnis des kartesischen Koordinatensystems unerlässlich.

4.3.6 Raumkoordinaten

In OpenGL werden die Koordinaten eines Punktes durch den Aufruf der Funktion

```
gl.glVertex3f(-0.5f,0.5f,0.5f);
```

spezifiziert; *vertex* ist das englische Wort für Raumkoordinate. Wird dieser Befehl ausgeführt, so werden die hier spezifizierten Raumkoordinaten sowie sämtliche augenblicklich geltenden Attribut-Werte (Abschnitt 4.3.3) übernommen. Insofern unterscheidet sich der `Vertex`-Befehl in OpenGL von anderen Attribut-Wert-Befehlen wie `Color` oder `Normal`, da ausschließlich der Vertex-Befehl die Raumkoordinaten sowie gleichzeitig sämtliche Attribut-Werte an die Graphik-Maschine übergibt.

4.3.7 Transformation und Rotation

Werden Objekte definiert, so legt man sie meistens in den Ursprung. Allerdings können natürlich nicht sämtliche Objekte an die gleich Stelle gezeichnet werden, weshalb man Funktionen für die Verschiebung, Rotation und Skalierung von Objekten einführt.

Um diese im GL_Sourcerer anzugeben, werden die Parameter

```
translate 4 5 6
rotate 45 x 12 y
scale 0.1 0.2 0.3
```

verwendet. Der erste Parameter **translate** gibt die Verschiebung eines Objekts an, wobei die Richtung der Verschiebung in x-, y- und z-Koordinaten angegeben wird; das Tupel (4,5,6) kann auch als *Verschiebungsvektor* aufgefasst werden. Der zweite Parameter gibt die Rotation an; **45 x** bedeutet, dass um die x-Achse um den Winkel 45° gedreht werden soll (die Winkelangaben erfolgen in Grad). Danach soll noch um 12 Grad um die y-Achse gedreht werden. Die Reihenfolge der Winkelangaben ist hier wichtig, da das Ergebnis von der Reihenfolge abhängt; man könnte auch den ersten und zweiten Parameter vertauschen, also

```
rotate 12 y 45 x
```

schreiben und erhielte dann ein anderes Ergebnis. Der dritte Befehl multipliziert die x-, y-, und z-Koordinaten jedes Vertex mit dem entsprechenden Faktor. In der Regel sollten Skalierungen vermieden werden, da diese verschiedene Nachteile haben.

Die Reihenfolge dieser Operationen ist in OpenGL wichtig, und relativ unübersichtlich. Tatsächlich werden die drei obigen GL_Sourcerer-Parameter in die vier OpenGL-Befehle

```
glTranslate( 4, 5, 6);
glRotate( 12, 0, 1, 0);
glRotate( 45, 1, 0, 0);
glScale ( 0.1, 0.2, 0.3);
```

übersetzt, und (bezüglich ihrer Wirkung) vom letzten zum ersten ausgeführt. Verwendet man den GL_Sourcerer, so werden die Befehle immer in der Reihenfolge: Skalierung, Rotation, Translokation ausgeführt, auch wenn sie innerhalb eines Befehl in anderer Reihenfolge aufgeschrieben wurden. Das gilt natürlich nicht für OpenGL, da man dort die richtige Reihenfolge einhalten muss, wobei sich der Programmierer bewusst sein muss, dass die tatsächliche Wirkung auf die Koordinaten vom letzten zum ersten Transformationsbefehl ist, und nicht wie meistens erwartet umgekehrt.

Um dieses zu verstehen, folgen einige Erläuterungen, die sehr mathematischer Natur sind und daher nicht unbedingt beim ersten Durchlesen verstanden werden müssen.

Im Ursprung definierte Koordinaten werden auch als *Objektkoordinaten* bezeichnet. Werden die Objekte skaliert, rotiert und verschoben, so nennt man die durch diese *Transformation* entstehenden Koordinaten auch *Weltkoordinaten*. Eine weitere Transformation ändert die Koordinaten so, dass sie von der Kamera in einer Standardposition aus gesehen werden können; diese werden dann als *Eye-Koordinaten* bezeichnet (*eye*: engl. für Auge). Eine weitere Transformation bildet schließlich die Koordinaten in einen Einheitswürfel mit der Kantenlänge zwei ab; diese werden als *Clipping-Koordinaten* bezeichnet.

Jede einzelne Transformation kann mittels einer Matrix-Multiplikation beschrieben werden. Die gesamte Transformation kann somit mittels dem Produkt dieser Matrizen, d.h. mit einer einzigen Matrix, beschrieben werden. Das hat den Vorteil, dass die Berechnung relativ einfach mit Hardware dargestellt und ausgeführt werden kann, wenngleich man sich den Nachteil einhandelt, dass statt des üblichen 3-dimensionalen ein 4-dimensionales Koordinatensystem benötigt wird; die vierte Koordinate eines Raumvektors wird als *homogene Koordinate* bezeichnet und in der Regel auf 1 gesetzt.

Die oben erwähnten Transformationen, können somit im Prinzip in einer Matrix zusammengefasst werden, mit der die jeweiligen Koordinaten multipliziert werden. Aus verschiedenen Gründen wird das allerdings nicht gemacht, sondern nur die Transformation von Objekt- in Eye-Koordinaten in einer Matrix – der *ModellView*-Matrix – zusammengefasst, während die Transformation von den

Eye- in die Clipping-Koordinaten durch die *Projection*-Matrix durchgeführt wird. Dieses ist vor allem notwendig, weil die Lichtberechnung in den Eye-Koordinaten durchgeführt wird, so dass die Kenntnis der Eye-Koordinaten für das System notwendig ist. Im zweiten Schritt wird dann die Transformation in die Clipping-Koordinaten durchgeführt, welche zusätzlich die Dreiecke 'abschneide' (*clip*), so dass sie genau in den Einheitswürfel passen und nicht mit einzelnen Ecken überstehen. Diese Koordinaten während zum Schluss durch die *ViewPort*-Transformation in die Gerätekoordinaten des Bildschirms abgebildet.

4.3.8 Attribut-Werte für Primitive in GL_Sourcerer und OpenGL

Betrachten wir noch einmal den erzeugten OpenGL-Code des `cubes` im Zusammenhang.

```
// Draw a cube
  gl.glGenLists(1);
  gl.glNewList(1, GL.GL_COMPILE);
    gl.glColor4f(1.0f,0.0f,0.0f, 1.0f);
    gl.glNormal3f(0.0f,0.0f,1.0f);
    gl.glBegin(GL.GL_QUAD_STRIP);
      gl.glVertex3f(-0.5f,0.5f,0.5f);
      gl.glVertex3f(-0.5f,-0.5f,0.5f);
      gl.glVertex3f(0.5f,0.5f,0.5f);
      gl.glVertex3f(0.5f,-0.5f,0.5f);
    gl.glEnd();
    ...
  gl.glEndList();
```

Der Code erzeugt zunächst einen Handle für eine Display-Liste und öffnet dann diese Display-Liste, um mit den folgenden Funktionsaufrufen die Graphik-Maschine von OpenGL zu initialisieren. Hierzu gehört z.B. das Setzen der Farbe eine Objekts oder das Setzen der Normalen. Danach werden Graphik-Primitive – hier ein `Quad_Strip` – spezifiziert. Das hier erzeugte Graphik-Primitiv wird von OpenGL gezeichnet, d.h. es wird ein Rechteck gezeichnet, dessen Koordinaten entsprechend den angegebenen sind.

Möchte man nur eine Fläche zeichnen, so lässt sich für das letzte Beispiel nur die vordere Fläche zeichnen, was mit dem GL_Sourcerer-Befehl

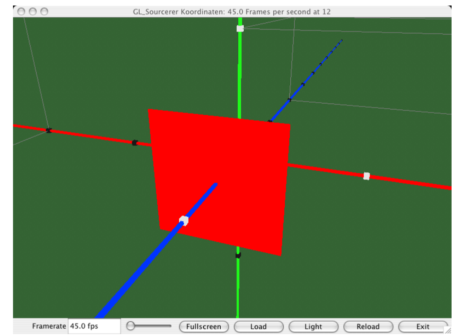
```
< Quadstrip
  vertex -0.5  0.5 0.5
  vertex -0.5 -0.5 0.5
  vertex  0.5  0.5 0.5
  vertex  0.5 -0.5 0.5
  Color 1 0 0
>
```

möglich ist. In diesem Befehl, den wir hinter den Befehl für das Koordinatenkreuz hinzugefügt haben, werden die Raumkoordinaten gesetzt, und dann die Farbe rot gewählt. Das Ergebnis ist jedoch eine weiße Fläche.

Im GL_Sourcerer spielt die Reihenfolge der Parameter in der Regel keine Rolle, da jeder Parameter durch seinen Namen angegeben wird. Allerdings gibt es hierzu (natürlich) Ausnahmen. Einmal wird in den meisten Fälle nur das erste Auftreten eines Parameters berücksichtigt. Wird also z.B. beim `Cube` mehrmals die Farbe durch `Color` gesetzt, so gilt nur die zuerst gesetzte Farbe.

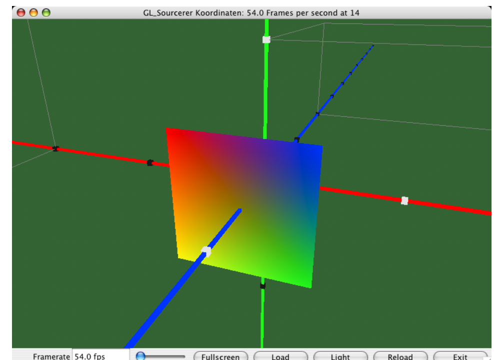
Die zweite Abweichung von der Regel tritt auf, wenn die verschiedenen Parameter jeweils auf andere Komponenten wirken, hier auf die Raumkoordinaten. Ändern wir das obige Beispiel ab zu

```
< Quadstrip
  Color 1 0 0
  vertex -0.5  0.5 0.5
  vertex -0.5 -0.5 0.5
  vertex  0.5  0.5 0.5
  vertex  0.5 -0.5 0.5
>
```



so wird die Fläche tatsächlich rot gezeichnet. Nun kann jedem Vertex eine eigene Farbe zugeordnet werden, da wir behauptet haben, dass genau die bei einem Vertex-Aufruf eingestellten Zustände übernommen werden. Es liegt daher nahe, vor jeden Vertex-Befehl einen eigenen Color-Befehl einzufügen. Man erhält dann beispielsweise für das Primitiv

```
< Quadstrip
  Color 1 0 0
  vertex -0.5  0.5 0.5
  Color 1 1 0
  vertex -0.5 -0.5 0.5
  Color 0 0 1
  vertex  0.5  0.5 0.5
  Color 0 1 0
  vertex  0.5 -0.5 0.5
>
```



einen farblichen Übergang zwischen den einzelnen Punkten.

Sieht man sich noch einmal die wesentlichen Anweisungen an, die OpenGL benötigt, um dieses Bild zu erzeugen, indem man mit dem **output**-Befehl eine Ausgabe erzeugt,

```
// draw a quad
  gl.glGenLists(1);
  gl.glNewList(2, GL.GL_COMPILE);
    gl.glBegin(GL.GL_QUAD_STRIP);
      gl.glColor4f(1.0f,0.0f,0.0f, 1.0f);
      gl.glVertex3f(-0.5f,0.5f,0.5f);
      gl.glColor4f(1.0f,1.0f,0.0f, 1.0f);
      gl.glVertex3f(-0.5f,-0.5f,0.5f);
      gl.glColor4f(0.0f,0.0f,1.0f, 1.0f);
      gl.glVertex3f(0.5f,0.5f,0.5f);
      gl.glColor4f(0.0f,1.0f,0.0f, 1.0f);
      gl.glVertex3f(0.5f,-0.5f,0.5f);
    gl.glEnd();
  gl.glEndList();
```

so erkennt man, dass OpenGL tatsächlich zwischen je zwei **Vertex**-Anweisungen eine **Color**-Anweisung verwendet. Jede Farbe wird entsprechend übernommen und der jeweilige Raumpunkt in dieser Farbe gezeichnet. Zwischen den Eckpunkte werden die Farben interpoliert, d.h. es wird abhängig vom Abstand eines Punktes von dem jeweiligen Eckpunkt ein gewisser Farbanteil dieses Eckpunkts verwendet. Dadurch entsteht ein kontinuierlicher Übergang zwischen den Farben. Zwar lässt sich diese Interpolation abstellen, aber für die meisten Anwendungen ist dieses die sinnvollere Einstellung.

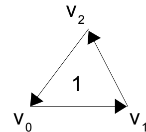
Es ist also wichtig zu verstehen, dass die Attribute je Raumkoordinate übernommen werden. Werden diese zwischen zwei **Vertex**-Anweisungen nicht verändert, so wird der zuletzt eingestellte Wert übernommen. In der Regel gibt es für alle Attribute Standardwerte, z.B. schwarz für Farben. Allerdings werden die Werte seit der letzten Änderung immer übernommen, so dass ohne eine sol-

che Einstellung die Färbung von Objekten davon abhängen könnte, welches Objekt vorher gezeichnet wurde. Das dürfte i.d.R. nicht erwünscht sein, so dass es vernünftig ist, sämtliche Attribute vor deren Verwendung zu setzen, auch wenn die Standardwerte ausreichen würden.

4.3.9 Primitive in GL_Sourcerer und OpenGL

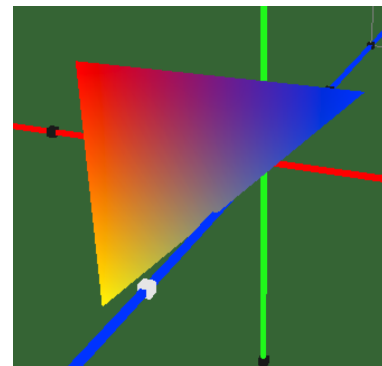
Wie oben bereits angedeutet, gibt es in OpenGL mehrere Flächentypen, die durch ihre Eckpunkte – die Vertices – definiert werden. Im Prinzip werden tatsächlich nur Dreiecke gezeichnet, so dass man eigentlich mit dem Primitiv Dreieck (zumindest für Flächen) auskäme. Zur komfortableren Programmierung und ggf. auch für den effizienteren Ablauf werden jedoch mehrere Primitive bereit gestellt.

Dreieck (Triangles)



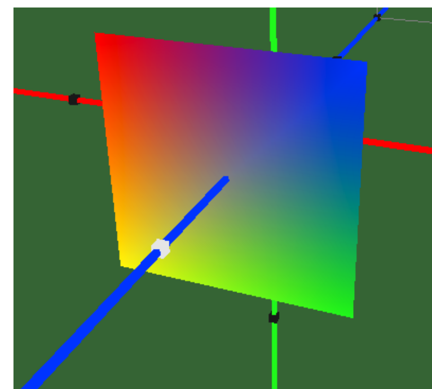
Um ein Dreieck zu zeichnen, sind dessen Eckpunkte (0,1,2) entgegen dem Uhrzeigersinn anzugeben. Der Befehl

```
< Triangles
  Color 1 0 0
  vertex -0.5  0.5  0.5
  Color 1 1 0
  vertex -0.5 -0.5 0.5
  Color 0 0 1
  vertex  0.5  0.5 0.5
>
```

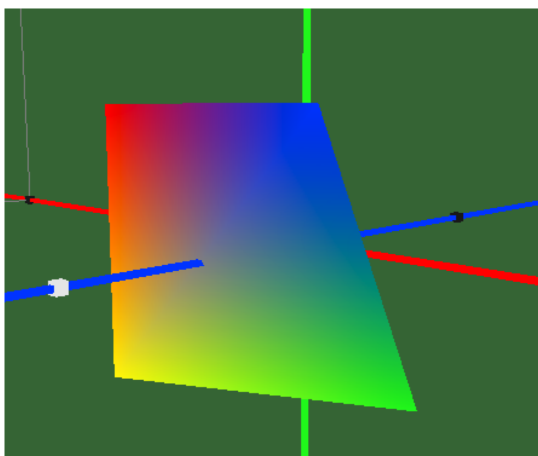


erzeugt somit ein Dreieck. Der aufmerksame Leser hat sicherlich bemerkt, dass das Schlüsselwort **`Triangles`** (**`TRIANGLES`**) im Plural geschrieben wird. Man kann offenbar mehr als ein Dreieck zeichnen. Tatsächlich erwartet OpenGL drei oder ein Vielfaches von drei Punkten, und interpretiert dann jeweils drei aufeinanderfolgende Raumkoordinaten als Eckpunkte eines entsprechenden Dreiecks. Es ist also in OpenGL nicht jedesmal neu **`Begin`** und **`End`** zu schreiben; entsprechend kann auch im GL_Sourcerer mehr als ein Dreieck in einem Befehl festgelegt werden. Es ist auch kein Fehler, wenn die Anzahl der Punkte kein Vielfaches von drei ist; restliche Punkte werden einfach ignoriert.

Da man oft nicht nur Dreiecke, sondern komplexere Flächen erstellt, deren Kanten aneinander liegen, erlaubt OpenGL zu einem Dreieck einen weiteren Punkt zu spefizieren, der mit den letzten beiden Punkten ein neues Dreieck bildet. Der Befehl heißt **`Tri-anglestrip`** (**`TRIANLGE_STRIP`**).

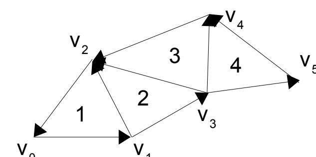


```
< trianglestrip
  Color 1 0 0
  vertex -0.5  0.5  0.5
  Color 1 1 0
  vertex -0.5 -0.5 0.5
```



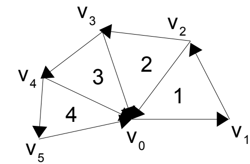
```
  Color 0 0 1
  vertex  0.5
0.5 0.5
  Color 0 1 0
  vertex  0.5
-0.5 0.5
>
```

Man erkennt praktisch keinen Unterschied

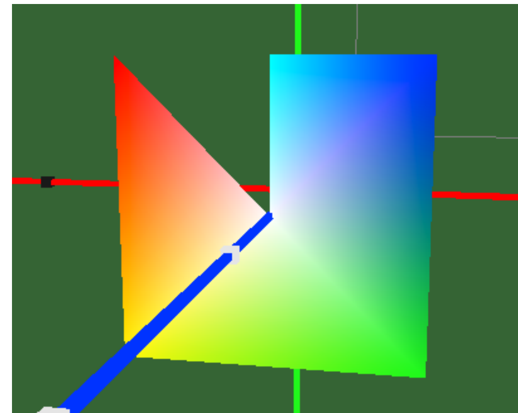


zum Quadstrip; dennoch gibt es einen wichtigen Unterschied, der bei Quads erläutert wird. Hier sei nur erwähnt, dass die vier Punkte nicht in einer Ebene zu liegen brauchen, da jeweils drei Punkte eine Ebene bestimmen, also die ersten drei Punkte (0,1,2) die Ebene des ersten Dreiecks, die nächsten drei (1,2,3) die des zweiten Dreiecks, usw. Setzt man die Tiefe des letzten Punkts zurück auf 0, so erhält man das nebenstehende Bild. Dieses entspricht genau der Definition von OpenGL.

Eine dritte Variante zum Zeichnen von Dreieck definiert zunächst den Mittelpunkt eines 'Fächers' (*fan*) von Dreiecken und dann die Randpunkte, wobei die Reihenfolge der Punkte wieder entgegen dem Uhrzeigersinn zu wählen ist. Mit dem folgenden Befehl




```
< trianglefan
  Color 1 1 1
  vertex 0   0   0.5
  Color 1 0 0
  vertex -0.5  0.5  0.5
  Color 1 1 0
  vertex -0.5 -0.5 0.5
  Color 0 1 0
  vertex  0.5 -0.5 0.5
  Color 0 0 1
  vertex  0.5  0.5 0.5
  Color 0 1 1
  vertex  0.0  0.5 0.5
>
```



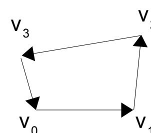
erhält man das nebenstehende Bild. Die Reihenfolge der äußeren Punkte kann man den Farben (rot, gelb, grün, blau, türkis) entnehmen; der Mittelpunkt ist weiß. Offenbar hängt die Anzahl zu verwendenden Punkte nicht nur von der geometrischen Form, sondern auch von den zu zeichnenden Farben ab. Dies ist auch in einem anderen Zusammenhang eine wichtige Technik, die wir später erläutern.

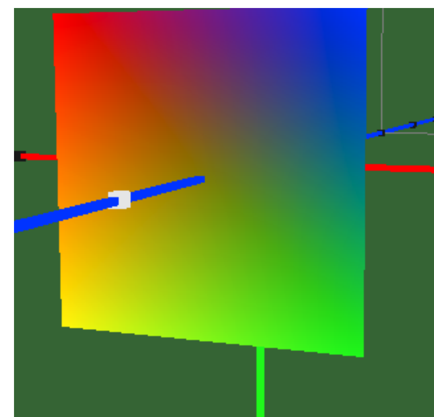
Der Vorteil dieser zusätzlichen Primitive liegt vor allem darin, dass für die gleiche Anzahl von Dreiecken weniger Vertices definiert werden müssen, also insbesondere an die Graphikmaschine übergeben werden; pro Dreieck muss bei Strip oder Fan jeweils nur ein Vertex festgelegt werden, bis auf die beiden initialen Vertices. Allerdings gibt es auch Nachteile, da gleiche Punkte jetzt die Farben auf verschiedenen Flächen definieren, und somit diese nicht mehr unabhängig gewählt werden können. Daher wird diese Technik meistens nicht angewendet, wenn Farben gezeichnet werden sollen.

Vierecke (Quads)

Wie bereits in dem ersten Beispiel gezeigt, werden Vierecke durch die Angabe ihrer Eckpunkte entgegen dem Uhrzeigersinn definiert. Will man ein oder mehr unabhängige Vierecke zeichnen, so kann man dieses durch den Befehl

```
< Quads
  Color 1 0 0
  vertex -0.5  0.5  0.5
  Color 1 1 0
  vertex -0.5 -0.5 0.5
  Color 0 1 0
  vertex  0.5 -0.5 0.5
  Color 0 0 1
  vertex  0.5  0.5 0.5
>
```

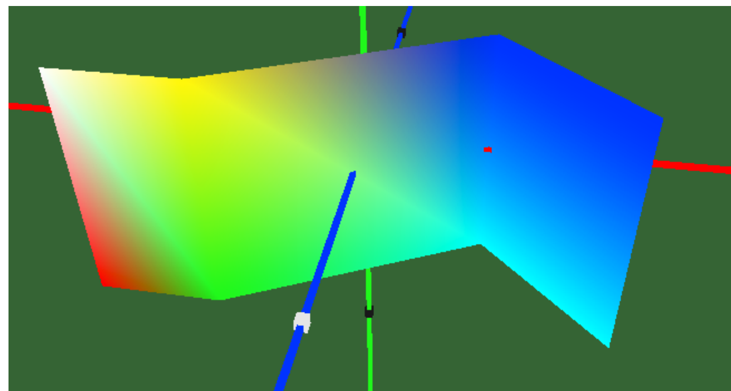




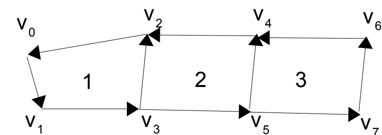
erreichen. Werden vier weitere Punkte angegeben, so wird entsprechend ein weiteres unabhängiges Viereck gezeichnet, usw. Beim Zeichnen von Vierecken (oder Polygonen mit beliebig vielen Ecken), ist es wichtig, dass diese in einer Ebene liegen. OpenGL schreibt dieses ausdrücklich vor, wenngleich die meisten Implementationen auch funktionieren, wenn das nicht garantiert wird. Tatsächlich ist hier von einer Implementierung eine gewisse Toleranz zu erwarten, da beliebige Raumkoordinaten aufgrund von Rundungsfehlern niemals auf genau einer Ebene liegen dürften. Die meisten Implementierungen behandeln **Quads** daher genau wie **Trianglestrips**. OpenGL verlangt, dass beim Quads die Innenwinkel in jedem Eckpunkt kleiner als 180° sind, d.h. das Viereck ist konvex. Andernfalls ist das Ergebnis nicht definiert und kann daher auf verschiedenen Systemen unterschiedlich aussehen.

Sollen mehrere Viereck aneinandergereiht werden, so können wieder die letzten beiden Punkte mit den nächsten beiden verbunden werden. Der Befehl lautet **Quadstrip**.

```
< quadstrip
  Color 1 1 1
  vertex -1     0.5 0.5
  Color 1 0 0
  vertex -1    -0.5 0.5
  Color 1 1 0
  vertex -0.5   0.5 0.5
  Color 0 1 0
  vertex -0.5 -0.5 0.5
  Color 0 0 1
  vertex  0.5   0.5 0
  Color 0 1 1
  vertex  0.5 -0.5 0
  Color 0 0 1
  vertex  1     0.5 0.5
  Color 0 1 1
  vertex  1.0 -0.5 0.5
>
```

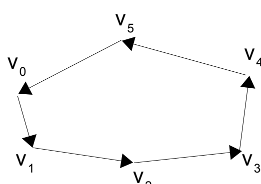


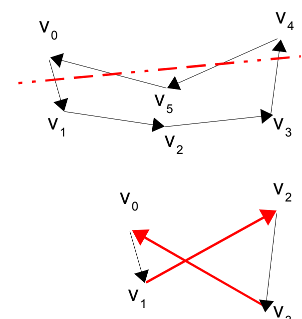
Das Ergebnis zeigt das obenstehende Bild. Man beachte, dass die Reihenfolge beim **Quadstrip** anders ist als beim **Quads**, da Punkt 2 und 3 vertauscht sind. Ähnlich wie beim Trianglestrip wird auch beim Quadstrip die Farbe für zwei Flächen durch einen Vertex definiert, so dass man entsprechend beschränkt in der freien Farbwahl ist. Da ein Viereck aus zwei Dreiecken besteht, sieht man leicht, dass rein numerisch je Dreieck hier wie beim Trianglestrip genau ein Vertex zu übergeben ist, zuzüglich der beiden initialen Vertices. Auch ist die Reihenfolge, in der die Punkte anzugeben sind, die gleiche wie beim Trianglestrip, so dass zwischen beiden Primitiven praktisch kein Unterschied besteht, außer dass Quadstrip nur für eine gerade Anzahl von Vertices definiert ist.



Polygon

Die letzte allgemeine Fläche, die als Primitiv gezeichnet werden kann, ist das Vieleck oder *Polygon*. Die Punkte eines Polygons sind wie beim Dreieck oder einfachem Viereck entgegen dem Uhrzeigersinn zu spezifizieren; der Befehlsname ist Polygon. Auch beim Polygon ist darauf zu achten, dass die Punkte in einer Ebene liegen. Ebenso muss dass Polygon konvex



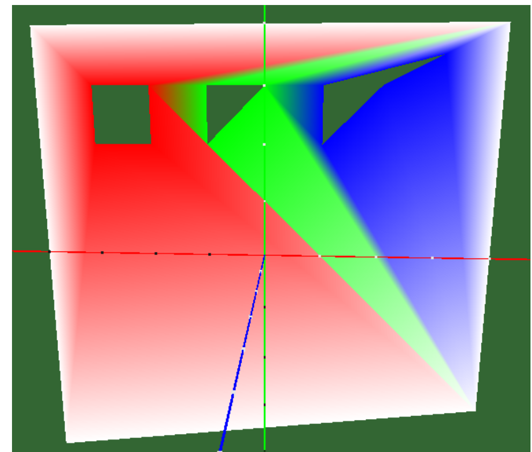


sein, d.h. die Innenwinkel müssen stets kleiner als 180° sein und selbstverständlich dürfen sich die Kanten eines Polygons nicht kreuzen.

Tessallation

Werden komplexere Flächen gezeichnet, die evtl. nicht konvex sind oder sogar Ausschneidungen enthalten, so stellt OpenGL in einer besonderen Funktionssammlung GLU (Graphic Library Utility) eine Methode zur Verfügung, die als *Tessallation* bezeichnet wird und vielleicht mit Mosaikbildung übersetzt werden kann. Das Konzept ist etwas komplexer und soll hier nicht erläutert werden. Das Beispiel

```
< Tessalation
  color 1 1 1
  vertex     -4 4   -4 -4    4 -3    4  4
  endContour
  color 1 0 0
  vertex     -3 3   -3  2   -2  2   -2 3
  endContour
  color 0 1 0
  vertex     -1 3   -1  2    0  3
  endContour
  color 0 0 1
  vertex      1 3    1  2    2  3     3 3.5
  endContour
>
```



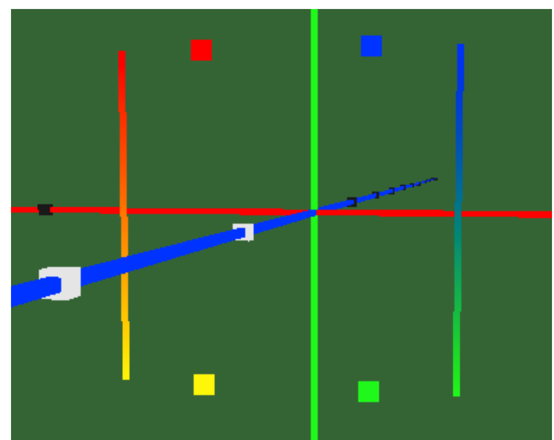
soll das wesentliche Konzept erläutern. Die Punkte der Polygone werden nur in der x-y-Ebene spezifiziert, da alle Vertices in einer Ebene liegen müssen; es sind daher je Punkt auch nur zwei Koordinaten anzugeben. Die erste 'Kontur' gibt den äußeren Rand der zu zeichnenden Fläche an; die anderen Konturen geben innere Ausschnitte der Fläche an. Letztere dürfen weder den äußeren Rand noch sich mit anderen inneren Konturen schneiden. Die einzelnen Konturen werden jeweils durch das Schlüsselwort `endContour` beendet. Pro Vertex-Zeile können ein oder mehrere Vertices definiert werden. Vor den Vertices gesetzte Attribute wie Farben werden entsprechend den Vertices zugeordnet. Das Ergebnis kann jedoch von der Tessallation der Flächen abhängen, was nicht vorhersehbar ist. Deshalb sollte man verschiedene Attribute, wie im Beispiel unterschiedliche Farben, vermeiden.

Punkte und Linien

Man kann auch Punkte bzw. Linien mit OpenGL zeichnen. Allerdings kann deren Größe nur in Pixeln angegeben werden, so dass sich diese beispielsweise bei größerer Entfernung nicht verkleinern. Daher ist der Nutzen dieser Primitive relativ beschränkt. Beispielsweise werden die Koordinaten in unserem Koordinatensystem nicht mit diesen Primitiven gezeichnet.

Die Befehle

```
< Lines
  size 5
  Color 1 0 0
  vertex -0.5  0.5  0.5
  Color 1 1 0
  vertex -0.5 -0.5 0.5
  Color 0 1 0
  vertex  0.5 -0.5 0.5
  Color 0 0 1
  vertex  0.5  0.5 0.5
>
```



```
< Points
  size 15
  Color 1 0 0
  vertex -0.25  0.5  0.5
  Color 1 1 0
  vertex -0.25 -0.5 0.5
  Color 0 1 0
  vertex  0.25 -0.5 0.5
  Color 0 0 1
  vertex  0.25  0.5 0.5
>
```

erzeugen entsprechend das nebenstehende Bild. Punkte und Linien können auch gefärbt werden, wobei unterschiedliche Farbenden der Linien wieder einen interpolierten Übergang bewirken. Wie gesagt wird die Dicke der Linien bzw. Punkte in Pixeln angegeben; die zugehörigen OpenGL-Befehle lauten

```
gl.glLineWidth( 5.0f );
```

bzw.

```
gl.glPointSize( 15.0f );
```

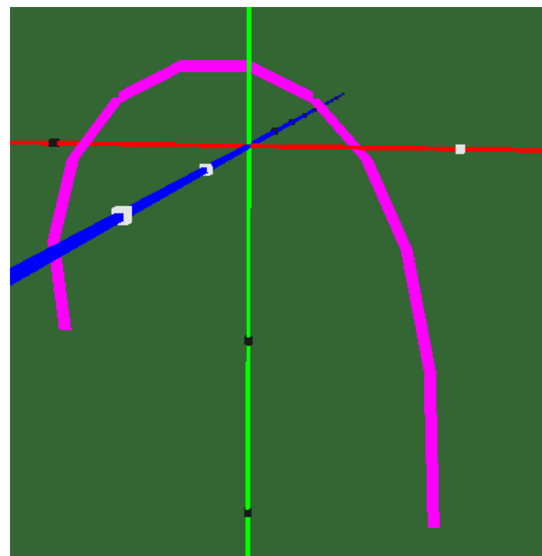
Neben LINES gibt es in OpenGL noch zwei weitere Befehle, nämlich **LINE_STRIP** und **LINE_LOOP**. Wie man sofort richtig vermutet gibt **LINE_STRIP** eine zusammenhängende Folge von Linien an, wobei **LINE_LOOP** zusätzlich den letzten Punkt automatisch mit dem ersten verbindet.

Beziér-Kurven und -Flächen

Neben den Standardkurven unterstützt OpenGL auch das Zeichnen von gekrümmten Kurven bzw. Flächen. Diese werden natürlich stückweise linear zusammengesetzt, aber die Berechnung der Zwischenwerte wird von GLU durchgeführt. Die Technik wird allgemein nach dem französischen Ingenieur Pierre Bézier benannt.

Das Beispiel

```
< Bezier
  Vertex -1 -1  0     -1  1 1
  Vertex  1  1 -1      1 -2 0
  Steps 10
  Width 4 10
  Color 1 0 1
>
```



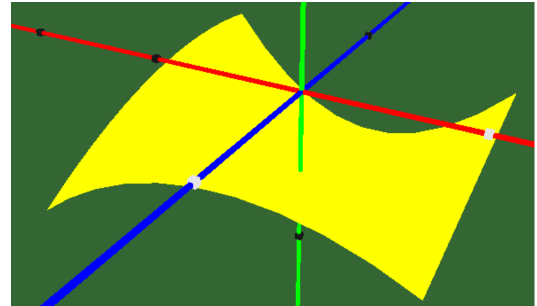
zeichnet die nebenstehende Kürve. In dem **bezier**-Befehl werden die Raumkoordinaten der Kurve als Anfangs- und Endpunkt definiert (der erste und der letzte Vertex). Dazwischen stehende Raumkoordinate sind Kontrollpunkte, welche den Richtungswinkel aus den Endpunkten und dessen Steilheit beschreiben. Insgesamt müssen also vier Vertices oder zwölf Werte angegeben werden, die wieder über mehrere Zeilen definiert werden können.

Der Parameter **Steps** gibt die Anzahl der Abschnitte an, in welche die Kurve aufgeteilt wird. Durch eine größere Zahl lässt sich die Kurve evtl. glatter zeichnen. Der Parameter **Width** gibt die Dicke der Kurve an. Werden zwei Werte angegeben, so ändert sich die Dicke der einzelnen Seg-

mente zwischen dem ersten und letzten Punkte stetig. Ebenso lässt sich eine Farbe angeben. Wird der Parameter `Point` mit Parameterwert `true` angegeben, so werden statt der Linien Punkte gezeichnet.

Statt Linien lassen sich mit dem folgenden Befehl auch Flächen zeichnen.

```
< Bezier 2
  Vertex  -1  0   -1    0  1.3 -1    1 0.2 -1
  Vertex  -1 -0.1 0     0  1.3  0    1 0.2  0
  Vertex  -1  1    1    0 -0.3  1    1 0.2  1
  Steps 10 15
  Width  5 10
>
```



Die Zahl hinter dem Befehlsnamen `Bezier` gibt hier die Dimension an, also mit 2 eine Fläche (es ist auch *surface* erlaubt). Es sind neun Raumkoordinaten anzugeben, wobei die erste und dritte sowie die siebte und neunte die Eckpunkte einer Fläche angeben, während die anderen Raumkoordinaten Kontrollpunkte sind. Mit `Steps` kann die Anzahl der Schritte angegeben werden, wobei diese Zahl in den verschiedenen Richtungen verschieden sein kann. Mit `Width` kann die Dicke von Punkten oder Linien angegeben werden. Ein weiterer Parameter `Points` kann mit den Werten `true` oder `false` steuern, ob die Kontrollpunkte ausgegeben werden sollen. Zusätzlich lässt sich `Line` oder `Point` angeben, wodurch statt der Fläche die Punkte, zwischen denen die Quads gezeichnet werden, bzw. die Linien ausgegeben werden.

Schrift

Um Schrift einzufügen, kann der Befehl

```
< Text  Hallo, wie geht's?
  Color 0.9 0.9 0.1 1
  Position 20 40
  Font SansSerif bold 20
>
< Textrenderer Und noch ein Text!
  Color 0.2 0.9 0.3  1
  Position 20 80
  Font Serif italic 30
>
```

verwendet werden. Nach dem Befehlsnamen (`Text` oder `TextRenderer`) steht der zu schreibende Text. Optional können weitere Parameter gesetzt werden, wie die Schriftfarbe und die Position in Pixeln (links unten ist <0,0>). Der Font ist standardmäßig `SansSerif`, kann aber auf jeden auf dem System bekannten Schrifttyp gesetzt werden, wie `Serif`, `Courier` oder `Helvetica`. Namen, die mit Leerzeichen getrennt sind, müssen in Textanführung (`"Lucida Handwriting"`) gesetzt werden. Statt `bold` kann auch `italic` oder `plain` gesetzt werden. Die letzte ganze Zahl gibt die Schriftgröße in Points an.

4.3.10 Vorder- und Rückseite

Wir haben bereits mehrfach betont, dass OpenGL Vorder- und Rückseite definiert. Dieses ist wichtig, da festgelegt werden kann, ob nur die Vorder-, nur die Rückseiten oder beide gezeichnet werden sollen. Der OpenGL-Standard legt letzteres als impliziten Fall fest, der also immer gilt, wenn nichts geändert wird. Wir gehen davon aus, dass Objekte immer eine gewisse Dicke haben, so dass es

überflüssig ist, die unsichtbaren Rückseiten von Flächen zu zeichnen. Möchte man dieses abstellen, so kann mit dem Parameter

```
cullback false
```

die Rückseite wieder gezeichnet werden. Der Befehl gilt nur für das ganze Primitiv. Das Wort *cull* heißt aussortieren, so dass die Rückseite nicht mehr 'aussortiert' werden. Um dieses in OpenGL zu erreichen, verhindern wir, dass das Zeichnen von irgendwelchen Seiten unterdrückt wird; durch

```
gl.glDisable(GL.GL_CULL_FACE);)
```

wird dieses erreicht. Sollen wieder gewisse Seiten nicht gezeichnet werden, so kann dieses durch

```
gl.glEnable(GL.GL_CULL_FACE);)
```

angeschaltet werden. Die nicht zu zeichnenden Seiten werden durch

```
gl.glCullFace(GL.GL_BACK);
```

ausgewählt, wobei statt `BACK` auch `FRONT` oder `FRONT_AND_BACK` geschrieben werden kann.

Vorder- und Rückseite einer Fläche können unterschiedlich beleuchtet werden, was jedoch erst bei den Beleuchtungsmodellen geschildert werden soll.

4.3.11 Geometrie und Farbe

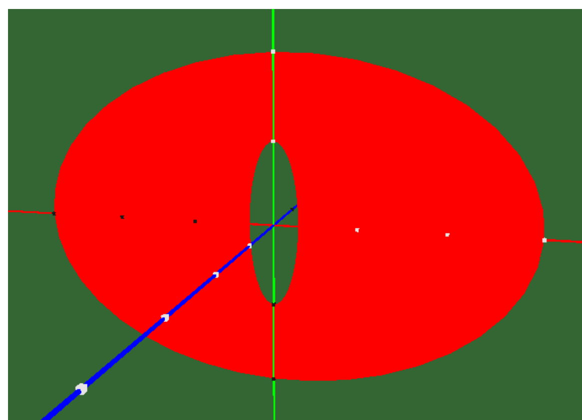
Die bisher eingeführten Konstrukte erlauben es in OpenGL beliebige Geometrien zu erstellen, d.h. beliebige Objekte in beliebigen Farben darzustellen. Einzige Voraussetzung ist es, dass die Objekte durch Dreiecke darstellbar sind, was aber durch beliebige Verfeinerung der Abstände zwischen zwei Raumkoordinaten jederzeit erreicht werden kann.

Wir haben bisher nur Würfel und einseitige Flächen gezeichnet. Natürlich gibt es viele andere geometrische Figuren, die häufiger vorkommen, und die daher einmal definiert werden. OpenGL bietet dafür zwei Möglichkeiten an, die GlutObjekte und die Quadrics. Ehe wir uns diesen zuwenden, betrachten wir Objekte, die der GL_Sourcerer zur Verfügung stellt.

Kreis und Ellipse

Neben dem Rechteck bzw. Quadrat kommen häufig runde Flächen vor, also der Kreis und die Ellipse (mit und ohne Loch). Die GL_Sourcerer-Befehle sind entsprechend kanonisch für die Ellipse

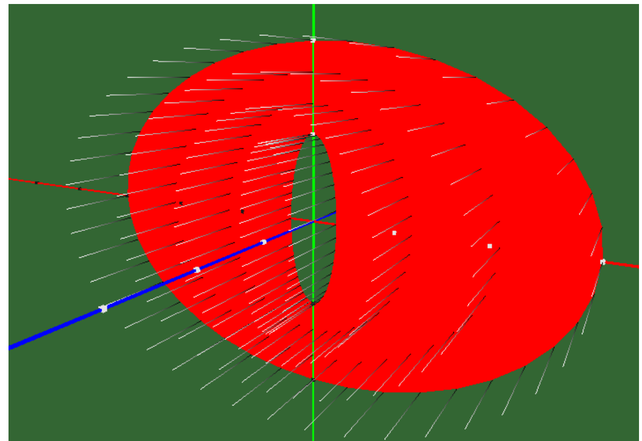
```
< Ellipse
  Size 3 0.3 2 1 40
  Steps 40 3
  Color 1 0 0
>
```



Die Größe (`size`) hat wahlweise bis zu fünf Parameter. Die ersten beiden geben die Radien des Außen- bzw. Innenkreises in x-Richtung, die nächsten beiden in z-Richtung an. Der optionale letzte Parameter gibt die Anzahl der Ecken an, die gezeichnet werden; standardmäßig werden zehn Ecken gezeichnet. Um einen einigermaßen runden Eindruck

zu erhalten, sollte man jedoch ca. 40 Ecken wählen. Das Bild zeigt diese Ellipse mit Loch. Werden die beiden Radien 2 und 1 fortgelassen, so entstünde eine Ellipse ohne Loch.

Der Kreis unterscheidet sich von der Ellipse nur durch das Schlüsselwort **circle** und dem Fortfall der z-Radien.



Der Parameter **Steps <corners> <steps>** kann zum einen zum Setzen der Anzahl der Eckpunkte verwendet werden. Zusätzlich ist es möglich, die Anzahl der Schritte in radialer Richtung zu definieren. Mit Hilfe des Befehls

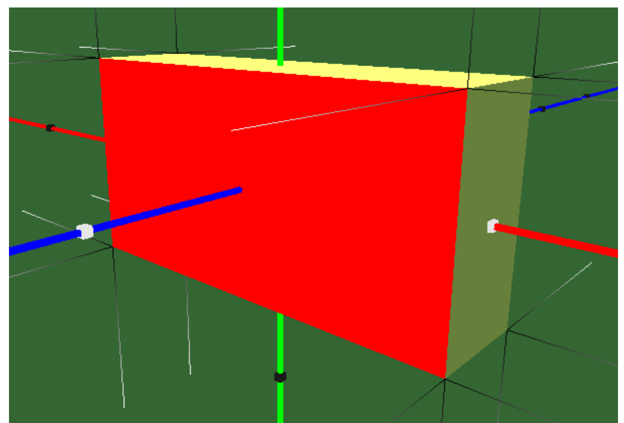
```
< drawnormals true
>
```

lassen sich die Normalen anzeigen, die für jede Raumkoordinate angezeigt werden. Wählt man die Parameterwerte für den Parameter **steps** wie oben angegeben, so erhält man die nebenstehende Ausgabe. Da **steps** auf 3 gesetzt wurde, werden die radialen Achsen in drei Abschnitte unterteilt und dort jeweils die Raumkoordinaten definiert. Die Ellipse wird dann durch drei elliptische Ringe gleicher Breite gezeichnet. Dieses lässt sich auch anhand der Ausgabe der OpenGL-Befehle leicht zeigen.

Der Grund für die Verwendung mehrerer Zwischenschritte ist vor allem die Lichtberechnung, da Licht jeweils pro Vertex berechnet und zwischen Vertices interpoliert wird. Dieses kann zu unschönen Effekten führen, wenn die Vertices zu weit auseinander liegen. Werden jedoch nur einfache Farben verwendet, wie wir das bisher gemacht haben, so reicht jeweils ein Abschnitt je Radius aus.

Quader und schiefe Ebene

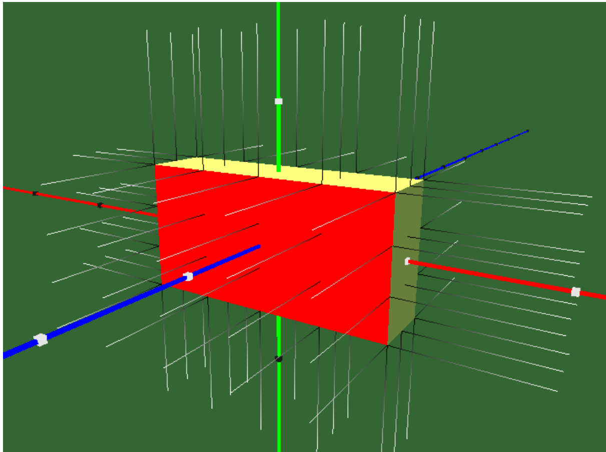
Einfache Körper sind der Würfel oder etwas allgemeiner der Quader sowie die Schiefe Ebene, die ein sechsseitiges Objekt mit nicht notwendig parallelen Seiten darstellt. Den Würfel haben wir bereits als **cube** kennengelernt; wir können jedoch auch einfach einen Quader daraus machen.



```
< Cube
  Size 2 1 0.5
  ColorFront 1 0 0
  ColorBack 1 1 0
  ColorUp 1 1 0.5
  ColorDown 1 0 1
  ColorLeft 0 1 1
  ColorRight 0.4 0.5 0.23
>
```

Wie man sieht, hat sich die Breite auf 2 verdoppelt, während die Höhe 1 geblieben ist; die Tiefe wurde auf 0.5 verkürzt. Da wir auch die Normalen anzeigen sieht man in diesem Fall, dass die Vertices genau in den Eckpunkten definiert wurden. Auch hier lassen sich Anzahl der Punkte je Oberfläche erhöhen, indem man den Parameter

```
Steps stepsX stepsY stepsZ
```

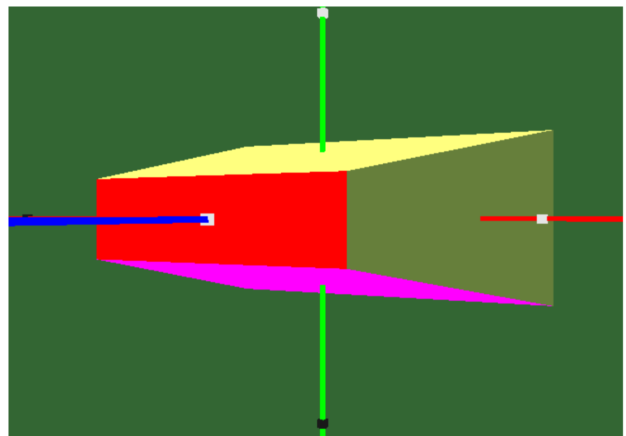


hinzufügt, wobei die Werte jeweils die Anzahl der Abschnitte in die Koordinatenrichtungen angeben (im Beispiels `Steps 4 3 2`).

Eine Schiefe Ebene (*slope*) stellt einen sechsflächigen Körper dar, bei dem die vordere und die hintere Fläche parallele Rechtecke (ggf. unterschiedlicher Größe) sind, und die mit entsprechenden Seitenflächen miteinander verbunden sind; vorderes und hinteres Rechteck können parallel verschoben sein.

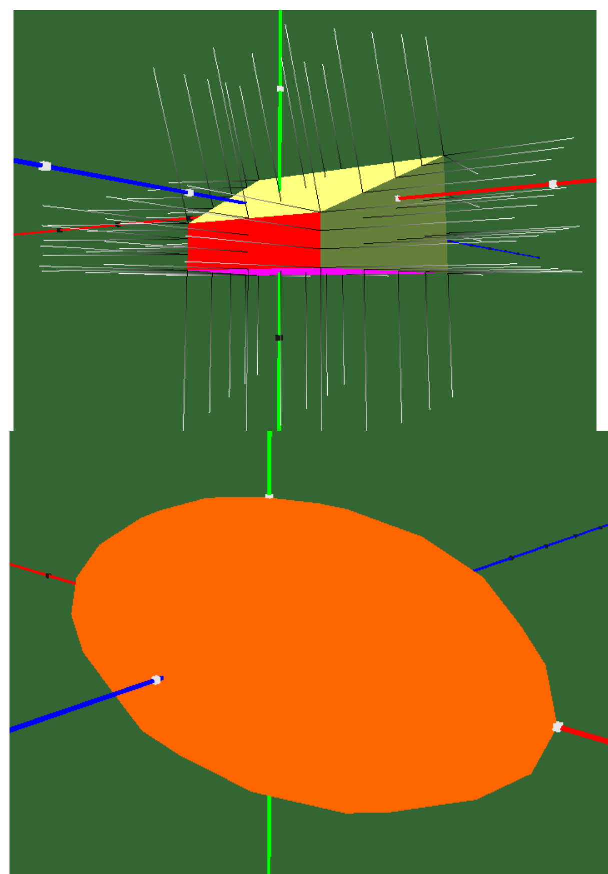
Das folgende Beispiel zeichnet eine Schiefe Ebene mit den entsprechenden Abmessungen Rechteck-vorne-X-Z, Rechteck-hinten-X-Z, Verschiebung-X-Z, Tiefe

```
< slope
  size 1 0.3 2 1 0 0 2
  ColorFront 1 0 0
  ColorBack 1 1 0
  ColorUp 1 1 0.5
  ColorDown 1 0 1
  ColorLeft 0 1 1
  ColorRight 0.4 0.5 0.23
>
```



Die Verschiebung wurde hier auf 0-0 gesetzt, so dass vordere und hintere Flächen um die z-Achse zentriert sind. Setzt man diese Werte auf 0.5 und –0.35, so bilden die rechte Seite mit der hinteren, unteren und vorderen jeweils rechte Winkel, wie man an dem nächsten Bild sieht. In jenem wurden zusätzlich die Normalen gesetzt, die jetzt nicht mehr achsenparallel, sondern entsprechend der Orientierung der Oberflächen senkrecht auf diesen stehen.

Eine Schiefe Ebene lässt sich natürlich vielfach einsetzen, um Objekte oder Räume zu zeichnen, die nicht achsenparallel liegen. Dieses lässt sich zwar auch mit den Grundflächen von OpenGL, also den Quads oder Triangles erreichen, ist dort aber sehr viel umständlicher. Aus diesem Grund sind vorgefertigte Körper in vielen Fällen bequemer. Ein weiterer Vorteil liegt darin, dass die Orientierung der Normalen automatisch berechnet wird, was sonst manuell gemacht werden müsste und häufig sehr unübersichtlich ist.



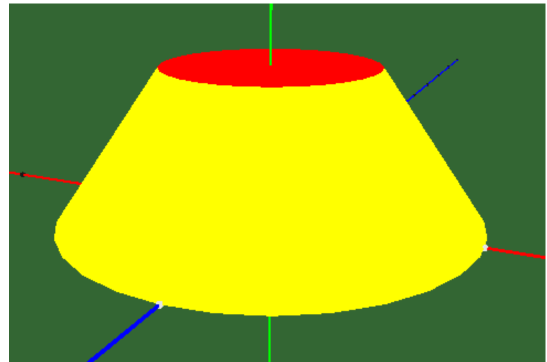
Kugel und Ellipsoid

Weitere häufig verwendete Objekte sind Kugeln bzw. Ellipsoide, welche ebenfalls einfach spezifiert werden können. Der entsprechende Befehl lautet

```
< sphere
  size 2 1 1
  steps 10
  Color 1 0.4 0
>
```

Das Bild zeigt, dass das Ellipsoid mit den Radien x=2, y=z=1 relativ eckig wirkt; in der Regel sollte auch hier die Anzahl der Punkte auf dem Umfang größer als zehn sein. Erfahrungemäß sind ca. dreißig ausreichend (man berechne die Anzahl der benötigten Raumkoordinaten!), kann aber auch mit zwanzig schon brauchbare Ergebnisse erzielen.

Kegelstumpf (cone)



```
< cone
  size 1 2 1.4141 30
  color 1 1 0
  colorUp 1 0 0
  colorDown 1 0 1
>
```

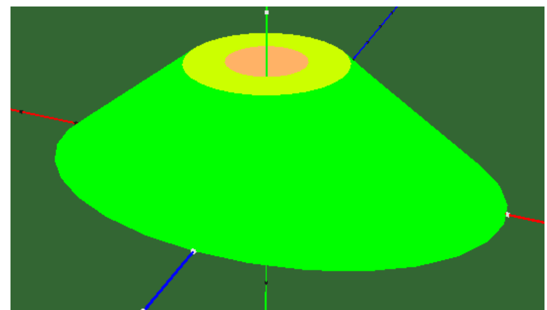
Ein Kegel wird durch die Höhe und die Radien angegeben.

```
size radiusX  [radiusZ]  radiusDownX [radiusDownZ]  height [corners]
```

Werden nur drei Parameterwerte angegeben, so bedeuten diese den unteren und oberen Radius sowie die Höhe des Kegelstumpfes. Bei fünf Parameterwerten werden zusätzlich die Radien in z-Richtung spezifiziert, so dass man dort Ellipsen erhält. Ein weiterer Parameter gibt die Anzahl der Ecken der Kreise bzw. Ellipsen an.

Mit `ColorUp` und `ColorDown` können der oberen und unteren Fläche eigene Farben zugeordnet werden; die äußere Fläche wird durch `Color` spezifiziert.

Rohr (pipe)



```
< Pipe
  Size 1 1 3 2 0.5 0.5 0.5 0.5  1.4141 30
  ColorOutside 0 1 0
  ColorInside 1 0.7 0.4
  ColorUp 0.8 1 0
  ColorDown 0 0 0
>
```

Das Rohr wird analog dem Kegelstumpf durch äußere und zusätzlich durch innere Radien definiert, die jeweils Kreise oder Ellipsen spezifizieren. Die Parameter sind (in einer Zeile!)

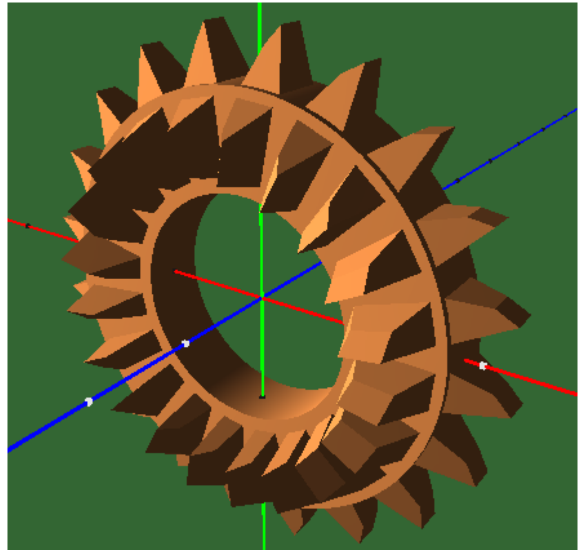
```
Size radiusUp radiusUpz radiusDown radiusDownz
     radiusUpIn radiusUpzIn radiusDownIn radiusDownzIn height [corner]
```

Nur der letzte Parameter ist optional (mit Standardwert 10). Farben können entsprechen gesetzt werden, wobei die Außenseite jetzt mit `ColorOutside`, die Innenseite mit `ColorInside` bezeichnet wird.

Zahnrad (gear)

GL_Sourcerer erlaubt das Zeichnen von Zahnrädern. Im Bild wird ein Zahnrad mit Beleuchtung gezeichnet, da sonst die sehr feinen Strukturen nicht erkennbar wären. Um ein Zahnrad zu zeichnen, muss der Befehl Gear mit den folgenden Parametern angegeben werden.

```
< Gear
  Size 1 2 0.6 0.3
  ToothHeight 0.3 0.5
  Teeth 20
  Side out front
  Sidespertooth 5
  ToothWidth 0.2 0.3 0.1
  ToothDepth 0.1 0.2 0.4 0.15
>
```



In diesem Beispiel wird ein Zahnrad mit dem Innendurchmesser 1 und dem Außendurchmesser 2, der Dicke 0.6 und der Zahnhöhe 0.3 gezeichnet, wobei 20 Zähne dargestellt werden. Die Höhe der Zähne kann an der linken und rechten Kante unterschiedlich sein und wird dann durch `toothHeight` angegeben.

Die Zähne werden standardmäßig an der Außenseite angebracht; wird side gesetzt, so können diese außen (`out`), innen (`in`), an der vorderen Frontseite (`front`) oder an der hinteren Seite (`back`) angebracht werden; es sind mehrere Angaben gleichzeitig in der gleichen Zeile möglich.

Mit dem Parameter *sidesPerTooth* werden die Anzahl der Segmente je Zahn angegeben. Die folgenden Angaben haben Einfluss auf die Form der Zähne und sind jeweils relativ anzusehen, d.h. die Summe sollte nicht größer als 1 werden. *ToothWidth* bestimmt dann die erste Steigung (20%), die Breite der äußeren Kante des Zahns (30%) und die zweite Steigung (10%); der an 1 fehlende Anteil (40%) bestimmt den zahnlosen Teil auf dem Radumfang. *ToothDepth* bestimmt entsprechend den Abstand vom Rand (10%), die erste Steigung (20%), den oben Teil des Zahns (40%) und die zweite Steigung (15%). Der an 1 fehlende Rest bestimmt den Abstand vom anderen Rand.

Heightmap

Um den Untergrund nicht völlig eben zeichnen zu müssen, wird in der Regel eine Höhenkarte verwendet. In diesem Falle wird eine zufällige Höhenkarte erzeugt, wobei für deren Größe unter *steps* die Anzahl der Punkte (*stepX*, *stepZ*) sowie deren Abstand (*deltaX*, *deltaZ*) angegeben werden kann; werden nur zwei Parameter angegeben, so wird eine entsprechende quadratische Fläche angenommen. Die Höhe (`size`) der einzelnen Punkte liegt zwischen der kleinsten und der größten Höhe; der nächste Parameterwert ist optional und gibt die Höhe des Rands an; der nächste ebenfalls optionale Parameter gibt eine Höhe in einem weiteren Abstand an und dient der Berechnung der Normalen am Rand. *borderHeight* (optional) gibt die Höhe des Rands an; *normalHeight* (optional) dient ausschließlich zur Berechnung der normalen am Rand (und wird nicht gezeichnet).

```
< heightmap
   size 0 1 0 0  // <minHeight> <maxHeight> <borderHeight> <normalHeight>
   steps 10 10 1 1 // <stepX> <stepZ> <deltaX> <deltaZ>
   slope 0.5 -0.1 -0.1 0 // <constant> <facX> <facZ> <facXZ>
   translate -5 -0 -5
   diffuse 0.6 0.2 0.2
 >
```

Der optionale Parameter **slope** addiert auf die Höhen der einzelnen Punkte einen Wert nach der folgen Formel; damit kann die schiefe Ebene eine mittlere Steigung in einer Richtung erhalten.

Höhe = constant + facX·X + facZ·Z + facXZ·X·(stepZ–Z) (X = 0..stepX, Z = 0..stepZ)

Skybox

In den Hintergrund der Szenen kann mittels einer Textur (siehe übernächstes Kapitel) eine 'unendlich entfernte' Umgebung dargestellt werden. Man verwendet hierfür in der Regel eine sogenannte Skybox, welche auf der Innenseite eines Würfels sechs Texturen 'aufklebt'. Um eine Skybox zu spezifizieren, müssen sechs solcher Texturen entsprechend dem unter Cubemap gesagten zur Verfügung gestellt werden.

```
< skybox
  size 60
  texture data/hills.jpg
  translate 0 26 0
  replace
>
```

In diesem Beispiel wird eine Skybox mit den Texturen data/hills{_positive_|_negative_}{x|y|z}.jpg gebildet. Ihre Größe beträgt 60·60·60; wird die Größe nicht spezifiziert, so wird standardmäßig **size** 10 verwendet. Damit die Skybox nicht von der Beleuchtung abhängt, verwendet man am besten *replace* oder *decal* für den Texturmodus. In der Regel wird die Skybox um den Mittelpunkt des Koordinatensystems gezeichnet, sie kann allerdings wie alle anderen Objekte transloziert oder rotiert werden; insbesondere die Höhe wird häufig der jeweiligen Situation angepasst.

Normalerweise wird die Skybox der Kameraposition angepasst, so dass der Betrachter den Hintergrund immer in gleicher Entfernung sieht. Dadurch entsteht der Eindruck einer 'unendlich weiten' Entfernung. Das wird in dieser Implementierung allerdings nicht unterstützt, um den Effekt der Skybox besser studieren zu können.

Mit dem Parameter

```
sides front back left right top down
```

können die Seiten, die gezeichnet werden sollen, angegeben werden. Fehlt dieser Parameter, so werden alle Seiten gezeichnet, sonst nur die genannten.

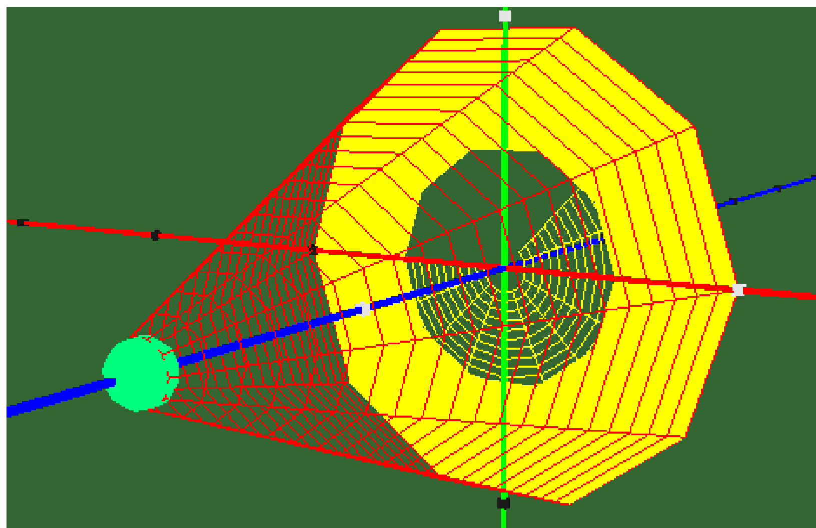
4.3.12 Objekte aus Standardbibliotheken

GLU (Graphic Library Utility) stellt mit den *Quadrics* spezielle komplexere Objekte zur Verfügung. Darüber hinaus werden mit den GLUT-Objekten aus der GLUT-Bibliothek weitere Körper zur Verfügung gestellt, u.a. die berühmte Teekanne.

Quadric

GLU stellt einige einfache Körper und Flächen zur Verfügung, nämlich **Sphere**, ***Cylinder***, **Disk**, **Partialdisk**.

```
< Quadric
  type cylinder line
  size 1 0.1 2
  steps 10 20
```



```
    normal smooth
    color 1 0 0
  >
  < Quadric
    type disk fill
    size 1 0.5
    rotate 90/5 z
    steps 10 20
    normal smooth
    color 1 1 0
  >
  < Quadric
    type partialdisk line
    size 0.1 0.5 45 225
    steps 10 10
    normal smooth
    color 1 1 0
  >
  < Quadric
    type sphere fill
    size 0.1
    steps 10 20
    normal smooth
    translate 0 0 2
    color 0 1 0.5
  >
```

Als **`Type`** kommen die oben genannten Objekte in Frage. Wird hinter diesen ein weiterer Parameter (**`fill`**, **`line`** oder **`silhouette`**) angegeben, so wird das Objekt entweder ganz ausgefüllt, nur das Liniengitter oder die konvexe Hülle gezeichnet; **`fill`** ist der Standardwert.

Bei der Kugel (**`Sphere`**) gibt **`Size`** die Größe ihres Radius an. Der Zylinder (**`Cylinder`**) hat einen unteren sowie einen oberen Radius und eine Höhe. Die Scheibe (**`Disk`**) hat einen inneren und einen äußeren Durchmesser; die partielle Scheibe (**`PartialDisk`**) hat zusätzlich einen Anfangswinkel und eine Winkelgröße, also vier Parameter; die Winkel sind in Grad anzugeben.

Der Parameter **`Steps`** gibt die Anzahl der Zwischenpunkte an, sowohl radial als auch in der Höhe. Ein weiterer Parameter **`Normal`** kann auf **`Smooth`** oder **`Flat`** bzw. **`None`** gesetzt werden; Standard ist **`Smooth`**. Für Gittermodelle kann der Parameter normal auf **`None`** gesetzt werden (**`Flat`** funktioniert nicht wie gedacht).

Damit OpenGL Quadrics erzeugt, muss ein

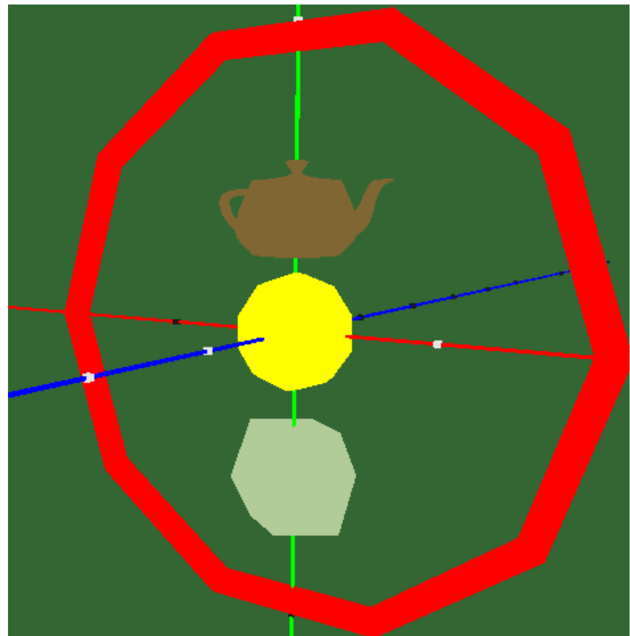
```
GLUquadric qobj = glu.gluNewQuadric();
```

erzeugt werden, welches als Parameter in den jeweiligen Funktionen übergeben wird (siehe OpenGL-Ausgabe-Funktion). Dieses Objekt muss nur einmal erzeugt werden, auch wenn verschiedene Quadrics erzeugt werden.

GlutObjekte

GLUT ist die Graphic Utility Library von OpenGL und stellt einige unterstützende Funktionen zur Verfügung. Wir verwenden hier die Möglichkeit, Objekte zu erzeugen. Die aktuelle Version verwendet die Objekttypen

```
< GlutObject
  Type  Torus
  Size  0.1 2
  Color 1 0 0
>
< GlutObject
  Type  Sphere
  Size  0.4
  Color 1 1 0
>
< GlutObject
  Type  Teapot
  Size  0.4
  Color 0.5 0.4 0.2
  rotate -90 x
  translate 0 0.5 0
>
< GlutObject
  Type  Rhombicdodecahedron
  Color 0.7 0.8 0.6
  translate 0 -1 0
  scale 0.5
>
```



wobei statt **Torus** auch geschrieben werden kann

```
cone, cube, cylinder, dodecahedron, icosahedron, octahedron,
rhombicdodecahedron, sphere, teapot, tetrahedron, torus;
```

deren Form entweder selbsterklärend sind oder durch Ausprobieren verstanden werden. Die Parameter sind

```
Size radiusInnen radiusAussen
Steps sides rings
```

Die Bedeutung der Parameterwerte von **Size** hängt von dem jeweiligen Objekt ab; in manchen Fällen sind sie ohne Bedeutung, da die Objekte keine Parameter annehmen - wie alle '**Hedrons**' - oder nur einen Skalierungsparameter haben – wie **Teapot**, **Sphere** und **Cube**; bei **Torus**, **Cone** und **Cylinder** bedeutet der zweite Parameter den Durchmesser des Rings des Torus bzw. die Höhe des Kegels oder Zylinders. Wird hinter dem Objekttyp **wired** geschrieben, so wird das Drahtgestell der Objekte gezeichnet, z.B.

```
type teapot wired
```

Die Parameter **sides** und **rings** zu **Steps** geben an, mit wie vielen Stufen das Objekt gezeichnet wird, bzw. haben bei den **Hedrons** und dem **Cube** (Würfel) keine Bedeutung.

5 Beleuchtung

Modelle sind nur sichtbar, wenn es Licht gibt. Allerdings ist Licht ein relativ komplexer Aspekt, der in OpenGL standardmäßig durch einen komplexen Satz von Parametern definiert wird.

In den folgenden Bildern wird der Würfel aus dem letzten Kapitel mit und ohne Beleuchtung dargestellt. Man sieht deutlich, dass eine geeignete Beleuchtung, die in diesem Fall richtungs- und entfernungsabhängig ist, ein Objekt sehr viel deutlicher hervorhebt als wenn die Flächen nur einfach gefärbt werden.

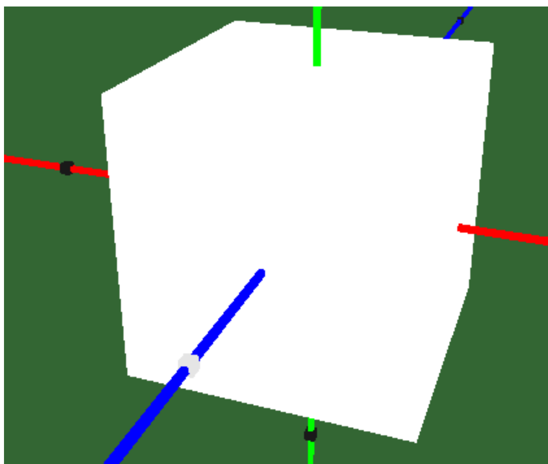


Abbildung 1: Würfel ohne Beleuchtung, alle Flächen gleich hell

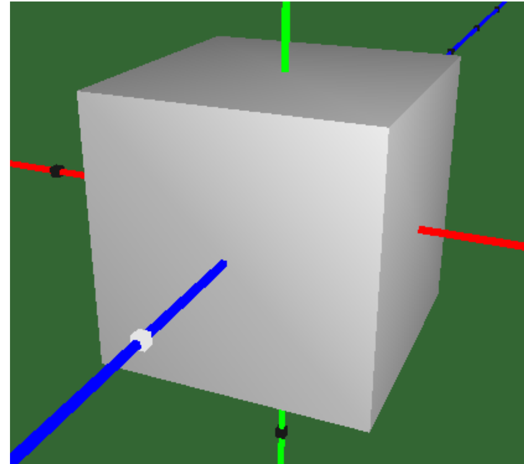


Abbildung 2: Würfel mit Beleuchtung, Intensität richtungs- und entfernungsabhängig

Der Vorteil ist offensichtlich, da man einen sehr viel realistischeren Eindruck von den Objekte erhält. Nachteilig ist der relativ große Aufwand, sowohl an Leistung als auch an Programmierung.

5.1 Lichtmodelle in OpenGL

OpenGL verwendet verschiedene Arten der Beleuchtung, die unabhängig sind; die einzelnen Komponenten werden berechnet und ihre numerischen Werte addiert.

Wenn Licht auf ein Objekt strahlt, so ist zum einen die Farbe des Lichts, zum anderen die Farbe der Objektoberfläche zu berücksichtigen. Die Farbe des Lichts wird in OpenGL durch die Angabe der Farbe der verschiedenen Lichtarten beschrieben; deren Bezeichnung ist **`ambient`**, **`diffuse`** und **`specular`**.

```
< light 0
  position 1 1 1
  ambient 0.2
  diffuse 1 1 1
  specular 0 1 1
>
< cube
  ambient 1
  diffuse 1 1 0.5
  specular 1 0.5 1
>
```

5.1.1 Ambientes Licht

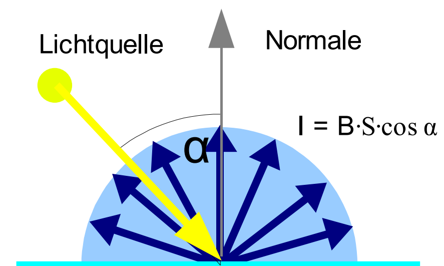
Unter *ambientem* Licht versteht man Licht, welches von einer Lichtquelle auf alle Objekte in der Umgebung scheint, egal aus welcher Richtung dieses Licht kommt; es hängt jedoch von der Entfernung zu dieser Lichtquelle ab. Man stellt sich hier vor, dass das Licht aufgrund von Reflexionen an Objekten in der Umgebung (*ambient* engl. für *umgebend*) von einer Lichtquelle auf alle Oberflächen strahlt, auch solchen, die von der Lichtquelle abgewandt sind.

In dem Beispiel wird für den Parameter *ambient* jeweils nur eine Parameterwert angegeben mit der Bedeutung, dass alle Farbanteile den gleichen Wert erhalten; er entspricht also der Grausstufe zwischen 0 und 1. Natürlich kann auch jede andere Farbe angegeben werden, indem die entsprechenden Farbkomponenten spezifiziert werden.

Um den ambienten Lichtanteil zu berechnen, wird das ambiente Licht der Lichtquelle (r_L,g_L,b_L) mit dem ambienten Material des Objekts (r_M,g_M,b_M) komponentenweise multipliziert:

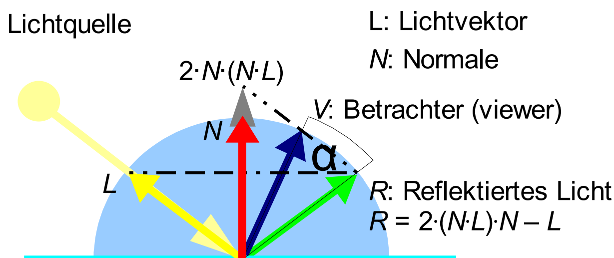
$$(r_L,g_L,b_L)\cdot(r_M,g_M,b_M) = (\, r_M\cdot r_L,\; g_M\cdot g_L,\; b_M\cdot b_L).$$

5.1.2 Diffuses Licht



Zeichnung 1: Lambertsches Gesetzt und diffuses Licht

Unter *diffusem* Licht versteht man Licht, welches abhängig von der Richtung und der Entfernung zur Lichtquelle unterschiedlich hell ist. Man geht hier von einem klassischen Lichtmodell nach Johann Heinrich *Lambert*, 1760, aus, dem *Lambertschen Gesetz*, welches besagt, dass bei einer diffusen Oberfläche das abstrahlende Licht nach allen Seiten gleich hell ist und die Intensität von dem Cosinus des Einfallswinkels zur Senkrechten (der Normalen) abhängt. Um die Intensität des Lichts zu berechnen, wird das Punktprodukt der normalisierten Vektoren in Richtung der Lichtquelle und der Normalenvektor miteinander multipliziert; dieser Wert wird als Dämpfungsfaktor verwendet, um die Intensität des diffusen Lichtanteils an der Objektoberfläche zu bestimmen. Der Farbanteil wird wieder als Produkt der diffusen Lichts mit dem diffusen Material be-



5.1.3 Speculares Licht

Die Intensität des *reflektierenden* (*specular*) Lichts hängt nicht nur vom Einfallswinkel ab, sondern auch von der Richtung des Betrachters ab. Wie vom Spiegel bekannt ist hier der Einfallswinkel gleich dem Ausfallswinkel (gespiegelt an der Normalen), d.h. das reflektierte Licht erscheint am hellsten, wenn der Betrachtungsausfallwinkel gleich dem Lichteinfallswinkel ist. Das nebenstehende Bild berechnet den Aus-

fallvektor R aus der Formel $R = 2\cdot(N\cdot L)\cdot N–L$, da der geklammerte Ausdruck das Punktprodukt zweier Vektoren ist, deren Wert die Projektion des Lichtvektors auf den Normalenvektor ist. Durch Subtraktion des Lichtvektors vom Doppelten dieses verkürzten Normalenvektors erhält man dann geometrisch exakt den Ausfallvektor; alle Vektoren sind Einheitsvektoren, d.h. sie haben die Länge 1. Die Intensität des Lichts aus der Sicht des Betrachters wird dann durch den Cosinus des Winkels zwischen Reflektionsvektor und Betrachtungsvektor bestimmt, also durch

$$R\cdot V = 2\cdot(N\cdot L)\cdot N\cdot V - L\cdot V.$$

Damit ergibt sich eine relativ einfache Formel für die Berechnung des Lichtausfalls. Allerdings wird dieser Wert in der Regel noch mit einer einstellbaren Zahl, dem Specular-Exponenten, potenziert, um ein brauchbares Ergebnis zu erhalten. Die Formel lautet dann

$$I_{\text{spiegelnd}} = S_{\text{light}}\cdot S_{\text{material}}\cdot \max((R\cdot V), 0)^n$$

Wie hier sind auch in den anderen Fällen zwei Faktoren im Spiel, welche die Farbe und deren Intensität bestimmen. Jeder dieser Lichtkomponenten hat eine eigene Farbe, sowohl auf Seiten der Lichtquelle als auch auf Seiten der Objektoberfläche, die als Material bezeichnet wird. Die Materialeigenschaften heißen entsprechend **`ambient`**, **`diffuse`** und **`specular`**, und müssen den jeweiligen Oberflächen der Objekte zugeordnet werden. Geht man dann davon aus, dass z.B. ein rotes Material nur den roten Lichtquellenanteil reflektiert, so wird das komponentenweise Produkt aus den entsprechenden Lichtquellen und Materialien das tatsächliche Licht wiedergeben.

Das hier verwendete Lichtmodell geht auf *Bui-Tuong Phong* aus dem Jahre 1975 zurück. In OpenGL vereinfacht man die Berechnung ein wenig, indem man den reflektierten Vektor aus der halbierten oder normalisierten Summe von Licht und Betrachtungswinkel bestimmt. Dieses ist nicht exakt, reicht aber häufig aus und ist einfacher zu berechnen. Dieses Lichtmodell wird als Blinn-Phong Lichtmodell bezeichnet, da es von Jim Blinn entwickelt wurde.

5.1.4 Die Lichtformel

Die vollständige Formel zur Berechnung des Lichts ist im folgenden abgebildet.

$$\begin{aligned} Light_{vertex} = {} & emission_{material} + ambient_{LightModel}\cdot ambient_{material} + \\ & + \sum_{i=0}^{N-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) (spotlight)_i \cdot [\ ambient_{light_i}\cdot ambient_{material} + \\ & \quad + \ max_0^{l\odot n}\cdot diffuse_{light_i}\cdot diffuse_{material} + \ (max_0^{s\odot n})^{shininess}\cdot specular_{light_i}\cdot specular_{material}\ ]. \end{aligned}$$

Diese Formel enthüllt noch weitere Einflussgrößen auf das Licht an der Oberfläche eines Objekts. Zunächst einmal sieht man, dass es mehr als eine Lichtquelle geben kann, und das reflektierte Licht jeder Lichtquelle addiert wird. Das Licht wird in OpenGL durch 256 Graustufen beschrieben; da es mindestens acht verschiedene Lichtquellen geben muss, bleiben für jede Lichtquelle nicht mehr als 32 Graustufen übrig. Dass sich damit kaum ein differenziertes Lichtmodell entwickeln lässt ist offensichtlich. Deshalb ist es wichtig eher sparsam mit den Lichtquellen umzugehen, da sonst die Szenen schnell zu hell und damit zu blass wirken.

5.1.5 Lichtdämpfung bei positionellem Licht

Darüber hinaus hängt die Lichtintensität von der Entfernung zwischen Lichtquelle und Oberfläche ab. Für diese Berechnung werden für jede Lichtquelle drei Parameter definiert, die als *constant*, *linear* und *quadratic attenuation* bezeichnet werden.

```
< light 0
  position 1 1 1
```

```
    ambient 0.2
    diffuse 1 1 1
    specular 0 1 1
    attenuation 0.5 0.3 0
  >
```

Abhängig von der Entfernung *d* zwischen Lichtquelle und Oberfläche wird die *Dämpfung* (engl. *attenuation*) des Lichtes berechnet. Die Formel lautet

$$\text{attenuation} = \left(\frac{1}{k_c + k_l d + k_q d^2}\right).$$

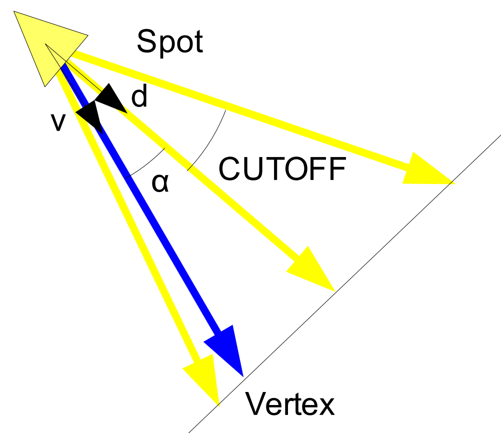
Standardwerte sind hier $k_c = 1$, $k_l = k_q = 0$, was keine Dämpfung bedeutet, da der Attenuation-Faktor dann konstant eins ist. In der Regel sind die Dämpfungparameter relativ klein zu wählen. Das Ergebnis hängt auch von dem absoluten Maßstab des gewählten Modells ab, so dass brauchbare Parameterwerte letzten Endes durch Austesten zu ermitteln sind. Gibt man gewisse Größen vor, z.B. in 1 m Entfernung 90%, in 5 m Entfernung 40% und in 20 m Entfernung 10%, so lässt sich leicht ein lineares Gleichungssystem aufstellen, dessen Lösungen dann als Koeffizienten verwendet werden können. Man beachte, dass die Koeffizienten k_c, k_l und k_q auch negativ sein können; wird dadurch der Nenner sehr klein, so kann auch eine Lichtverstärkung auftreten, was für besondere Effekte interessant sein mag.

5.1.6 Direktionelles Licht

Eine Variante des richtungsabhängigen Lichts ist entfernungsunabhängig. Das *direktionelle* Licht hängt nur von der Richtung ab, aus der das Licht kommt, bzw. allgemein von einem *Richtungsvektor*. Der Entfernungsfaktor wird in diesem Falle nicht berücksichtigt (bzw. er wird tatsächlich berechnet, so dass der Programmierer diesen auf den Standardwert 1,0,0 zu setzen hat). Ansonsten wird die gleiche Rechnung durchgeführt wie beim positionellem Licht, außer dass der Richtungsvektor zur Lichtquelle immer konstant ist. Um dieses im GL_Sourcerer einzustellen, ist der Parameter **`position`** nur durch den Parameter **`direction`** zu ersetzen, dessen Parameterwerte einen Richtungsvektor angeben.

5.1.7 Scheinwerferkegel (spot)

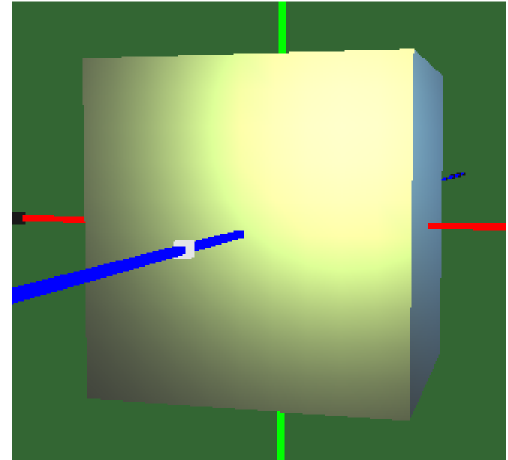
Ein weiterer Faktor innerhalb der Summe soll einen Scheinwerferkegel nachbilden. Es ist einfach, den Vektor von der Lichtquelle zu dem Vertex zu berechnen als Differenz aus den Ortsvektoren des Vertex und der Lichtquelle; ist dann der Vektor der Mittelachse (*spotDirection*) des Scheinwerfers vorgegeben, so lässt sich aus dem Punktprodukt der beiden normalisierten Vektoren der Cosinus des Winkels zwischen diesen beiden Vektoren berechnen. Dieser kann mit dem Cosinus des *SpotCutoff*-Winkels verglichen werden, und wenn der Wert größer ist liegt der Punkt im Licht, sonst im Schatten (der Cosinus fällt monoton).



Der Cosinus des Winkels wird somit zunächst verglichen und der Dämpfungsfaktor für dieses Licht auf null gesetzt, wenn der Vektor außerhalb des Lichtkegels liegt. Ansonsten wird der Cosinuswert verwendet, um die Intensität des Lichts innerhalb des Lichtskegels zu bestimmen. Aus ähnlichen Gründen wie beim specularen Licht wird auch in diesem Fall nicht der Wert des Cosinus direkt verwendet, sondern dieser

Wert mit einem *SpotExponent* potenziert. Die geeignete Größe dieses Werts hängt von den Lichtverhältnissen ab und sollte ausgetestet werden.

```
< light 0
  position 1 0.5 3
  ambient 0.1
  diffuse 1 1 0
  specular 0 0.4 0.5
  spotcutoff 30
  spotexponent 40
  SpotDirection 0 0 -1
>
```



Das nebenstehende Bild zeigt das Ergebnis bei diesen Parametern und geeigneter Einstellung des Würfels: die `SpotDirection` wird nicht mit verändert wenn der Raum gedreht wird, so dass sich diese relativ zur Drehung des Raum ändert. Außerdem ist es noch nötig, die Unterteilung der Flächen des Objekts zu erhöhen (`steps 10`), da OpenGL nur die Vertices beleuchtet und die Lichtwerte der Flächen zwischen den Vertices linear interpoliert werden. Der Leser beobachte selbst, welches Auswirkungen es hat, wenn keine feinere Unterteilung vorgenommen wird.

5.1.8 Globales ambientes Licht

Die Formel für die Lichtberechnung enthält noch zwei weitere Terme, die jedoch nicht von den Lichtquellen abhängen.

```
< LightModel
  ambient 0.2
  localViewer true
  twoSide true
>
```

Der erste Parameter wird als *globales ambientes Licht* bezeichnet und gibt einen Grundlichtanteil an, der auf alle Lichtwerte addiert wird. Dieses soll den Effekt simulieren, dass in jeder Welt immer etwas Licht ist, etwa Sonnenlicht oder Sternen- und Mond-Licht. Diese Komponente kann sehr einfach benutzt werden, um die Grundstimmung einer Szene zu definieren, etwa rötliches Licht bei Sonnenuntergang oder bläuliches Licht in einer strahlenden Umgebung. Die Einstellung erfolgt als LightModel mit dem Parameter ambient.

In diesem Beispiel erhielt ambient nur einen Parameterwert 0.2; dies bedeutet, dass alle drei Farbanteile auf 0.2 gesetzt werden. Werden drei Werte angegeben, so werden die Farbkomponenten unterschiedlich gesetzt. Dieser Wert ist zugleich der Standardwert von OpenGL.

Unter Lichtmodell werden in OpenGL eine Reihe weiterer Parameter gesetzt, die globale Bedeutung haben. Diese Bedeutung soll hier nicht vertieft werden und kann der Sprachbeschreibung entnommen werden.

5.1.9 Emissives Licht

Ein weitere Lichtanteil wird als *emissives Licht* bezeichnet, was mit strahlendem Licht übersetzt werden könnte. Hiermit kann einer Oberfläche unabhängig von einer Lichtquelle ein Lichtanteil zugeordnet werden, z.B. einer großflächigen Lichtquelle wie einem Lampenschirm oder einem Diabild- oder Röntgenbildbetrachter; man beachte, dass die Formel eindeutig sagt, dass das Licht ande-

rer Lichtquellen ggf. auf diese Lichtquelle addiert wird. Das bedeutet dann, dass der Farbton des emissiven Lichts bzw. dessen Intensität verfälscht wird, was häufig nicht erwünscht ist. Daher sollten derartige Objekte ohne Lichtquellen gezeichnet, oder zumindest deren anderen Materialeigenschaften (ambient, diffuse, specular) auf null gesetzt werden.

5.2 Licht in OpenGL

Die Berechnung von Licht muss in OpenGL explizit eingeschaltet werden (`Enable(LIGHTING);`); dieses gilt ebenfalls für jede aktive Lichtquelle (`Enable(LIGHT0);`). Sobald die Lichtberechnung aktiviert ist, werden die Farbwerte, die mit `Color3f(...)` gesetzt wurden, ignoriert; dann gelten für die Objekte nur noch die Materialeigenschaften (**ambient**, **diffuse**, **specular**, **emission**). Die entsprechenden OpenGL-Befehle kann man dem folgenden Programmfragment entnehmen. Da auch das Ein- bzw. Ausschalten der Lichtberechnung Performance kostet, sollte man bei konstanten Parametern diese Befehle in einer Display-Liste ablegen.

Die folgenden Befehle wurden dem *GL_Sourcerer*-Output entnommen.

```
  gl.glEnable(GL.GL_LIGHTING);
  gl.glEnable(GL.GL_LIGHT0);
{ float[] v = {1.0f, 0.5f, 3.0f, 1.0f};
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_POSITION, v); } // auch für direction
{ float[] v = {0.1f, 0.1f, 0.1f, 1.0f};
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_AMBIENT, v); }
{ float[] v = {1.0f, 1.0f, 0.0f, 1.0f};
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_DIFFUSE, v); }
{ float[] v = {0.0f, 0.4f, 0.5f, 1.0f};
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_SPECULAR, v); }
  gl.glLightf(GL.GL_LIGHT0 , GL.GL_CONSTANT_ATTENUATION, 0.8); }
  gl.glLightf(GL.GL_LIGHT0 , GL.GL_LINEAR_ATTENUATION, 0.2); }
  gl.glLightf(GL.GL_LIGHT0 , GL.GL_QUADRATIC_ATTENUATION, 0.0); }
{ float[] v = {0.0f, 0.0f, -1.0f, 1.0f};
  gl.glLightfv(GL.GL_LIGHT0, GL.GL_SPOT_DIRECTION, v); }
  gl.glLightf(GL.GL_LIGHT0, GL.GL_SPOT_EXPONENT, 40.0);
  gl.glLightf(GL.GL_LIGHT0, GL.GL_SPOT_CUTOFF, 30.0);
```

Man vergleiche dieses mit den entsprechenden GL_Sourcerer-Befehlen.

```
< light 0
  position 1 0.5 3
  ambient 0.1
  diffuse 1 1 0
  specular 0 0.4 0.5
  attenuation 0.8 0.2 0
  SpotDirection 0 0 -1
  spotexponent 40
  spotcutoff 30
>
```

Die Übersetzung ist relativ kanonisch, wenngleich die OpenGL-Befehle deutlich länglicher sind. Insbesondere die Notwendigkeit, viele Parameter mit Feldern zu setzen, ist umständlicher als im GL_Sourcerer. Außerdem beachte man, dass die Konstanten in OpenGL häufig Underscores verwenden, was der GL_Sourcerer weitgehend vermeidet.

Einige dieser Befehle müssen in OpenGL in der Regel vor jedem Frame neu ausgeführt werden, da beispielsweise die Lichtberechnung im Eye-Space durchgeführt wird, und daher die Position der Lichtquelle und die Richtung des Spotlights in jedem Frame erneut mit der ModelView-Matrix multipliziert werden müssen. Andere Parameter brauchen nur einmal initialisiert zu werden und können dann – wenn sie zwischen zwei Frames nicht verändert werden – unverändert übernommen werden.

In der Regel wird man wegen dieser komplexen Abhängigkeiten alle Lichtparameter in jedem Frame neu setzten. Dieses gilt auch, weil in einem größeren Level eine Lichtquelle an verschiedenen Stellen verwendet werden kann; die unterschiedlichen Lichtquellen werden nur benötigt, wenn sie gleichzeitig den gleichen Vertex bescheinen sollen. Nach der Berechnung aller Vertices, die von einer oder mehreren Lichtquellen beschienen werden sollen, können die Parameter der Lichtquellen auch wieder zurückgesetzt bzw. für andere Vertices (d.h. für einen anderen Bereich eines Frames) anders parametrisiert werden! Daher ist die maximale Anzahl von acht Lichtquellen eigentlich keine große Beschränkung.

Das direktionelle Licht wird übrigens auch mit der OpenGL-Konstante **`POSITION`** gesetzt, wobei der letzte Parameter des Vektors 0.0 ist; bei positionellem Licht ist er 1.0.

5.2.1 Beleuchtung von Flächen

OpenGL beleuchtet nicht etwa die Flächen, sondern nur die Vertices. Die Farben der Flächen zwischen den Vertices werden linear interpoliert. Dieses vereinfacht die Berechnung, führt aber bei zu großen Abständen zwischen den Vertices zu deutlich sichtbaren, unerwünschten Effekten. Um dieses zu vermeiden müssen in der Regel auch bei großen, ebenen Flächen mehr Vertices als für die Beschreibung der Geometrie nötig eingefügt werden. Das belastet natürlich auch die Performance, ist aber auch häufig umständlich zu programmieren.

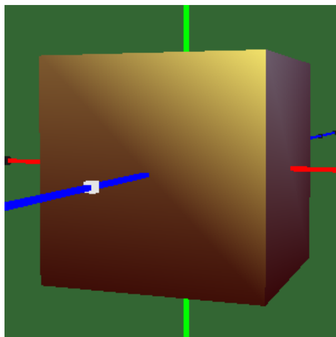


Abbildung 3: Keine Unterteilung

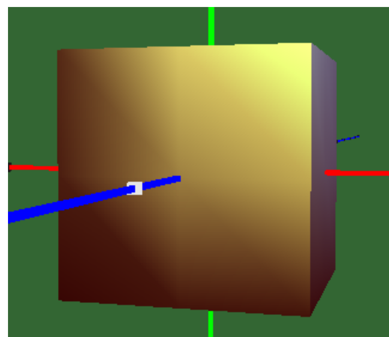


Abbildung 4: Zwei Unterteilungen

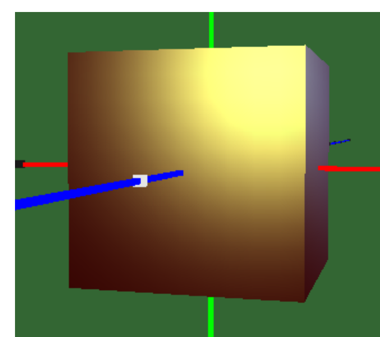


Abbildung 5: Zehn Unterteilungen

Die genannte Möglichkeit, mehr Vertices einzufügen, wird bei modernen Shadern in der Regel dadurch vermieden, dass die Lichtberechnungen je Pixel durchgeführt werden; diese Technik wird auch als *Pixel-Lighting* bezeichnet. Auch diese Technik belastet die GPU, ergibt aber natürlich wesentlich bessere Ergebnisse. Da außerdem nur jene Pixel gezeichnet werden, die innerhalb des Frustums liegen, also sichtbar sind, ist dieses häufig auch bezüglich der Effizienz die bessere Alternative. In diesem Fall ist offenbar bereits bei der Erstellung der geometrischen Daten darauf Rücksicht zu nehmen, welche Art von Shadern verwendet werden.

5.3 Hinweise zur Beleuchtung

Beleuchtung ist ein schwieriges Thema, da es mehrere sich auch widersprechende Anforderungen vereint. Zum einen soll mit Licht eine Stimmung geschaffen werden, die der jeweiligen Szene, Handlung oder Dramaturgie entspricht. Dieses ist ein eher künstlerischer Anspruch, der z.B. beim Film eigene Fachkräfte hervorgebracht hat, die mit aufwändigen Messgeräten und Scheinwerfersystemen komplexe Szenen ausleuchten. So verwendet man etwa in 'Billigproduktionen' wie Fernsehserien meist weniger differenziertes Licht, das also weniger Schatten enthält und somit weniger räumliche Tiefe und detaillierte Strukturen aufweist.

Daneben steht die Technik, die Licht in angemessener Intensität und Farbe an geeigneter Stelle zur Verfügung stellen muss. Dieses Problem gibt es im realen Film genauso wie in Computeranima-

tionen, wobei letztere noch mit dem Problem zu kämpfen haben, dass man nur einen relativ kleinen Lichtbereich zur Verfügung hat, von normalerweise nicht mehr als 256 Grauwerten. Es wird bereits versucht, mit HD-Systemen feinere Graustufenbereiche zu definieren, allerdings hat man mit den üblichen Monitoren ein Medium zur Verfügung, welches meistens gar nicht viel mehr Graustufen darstellen kann.

Bei der Erstellung eines Beleuchtungsmodells muss man zunächst entscheiden, wie viele Lichtquellen man verwenden möchte. Kommt das Licht von zu vielen Seiten, so wirkt die Szene zu wenig konturiert, da man keine beschatteten Gebiete sehen kann, die eigentlich erst räumliche Tiefe vermitteln. Verwendet man zu wenige Lichtquellen, so wirkt das Bild künstlich, da zumindest im Freien immer Licht von fast allen Seiten auf Objekte strahlt, wenn auch in unterschiedlicher Intensität und Färbung.

Im Freien kann man sich die Sonne als den einzigen direktionellen Strahler vorstellen; es ist aber auch möglich, diesige Tage oder Nebel zu emulieren, so dass dann wiederum überhaupt keine Schatten fallen. Auch Färbung ist am hellen Tag meist die 'Standardfarbe' der Objekte, so dass Stimmung nur in der Dämmerung oder in besonderen Umgebungen vorkommt.

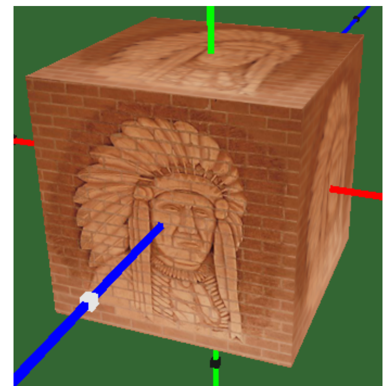
6 Texturen

Um die Oberflächen eines Objekts noch differenzierter zu gestalten, können auf diese Bilder gelegt werden. Die Bezeichnung für diese Bilder in diesem Zusammenhang ist immer *Textur*, was von dem englischen Wort *texture* abgeleitet ist und u.a. die Bedeutung (Oberflächen)-Struktur oder auch Maserung von Holzoberflächen hat.

Um ein Bild auf eine Oberfläche zu legen, muss zum einen ein Bild vorliegen; zum anderen muss eine Projektion der Bildelemente auf die Fläche vorgenommen werden. OpenGL verwendet spezielle Mechanismen zum Laden von Bildern, wobei die Bilder zugleich für einen geeigneten Zugriff vorbereitet werden. Die Projektion der Bildelemente auf die Oberfläche wird mittels Texturkoordinaten durchgeführt, d.h. den Bildern wird ein recht einfaches Koordinatensystem zugeordnet, so dass jeder Punkt (oder *Texel*) eines Bildes beschrieben werden kann, und diese Texel werden den Vertices einer Oberfläche zugeordnet. Diese Zuordnung lässt sich teilweise automatisieren, da die Berechnung von Texturkoordinaten häufig sehr umständlich ist.

6.1 Texturen auf Flächen

Um eine Textur auf einen Würfel zu zeichnen, kann im GL_Sourcerer der Würfel-Spezifikation einfach der Parameter einer Textur hinzugefügt werden.



```
< Cube
  Texture data/bricks2.jpg
>
```

Das Bild erscheint auf allen Seiten des Würfels gleich, wobei jedoch die Orientierung im Prinzip frei gewählt werden kann; hier ist eine konkrete Ausrichtung der einzelnen Bilder willkürlich festgelegt. Der Parameterwert hinter `Texture` ist ein Pfad zu einer Datei, die ein Bild enthalten muss. Im GL_Sourcerer muss dieses ein Name relativ zum Programm (in der Regel dem jar-Archiv) sein. Falls das Betriebssystem zwischen Groß- und Kleinschreibung unterscheidet, ist auch hier darauf zu achten.

Die OpenGL-Befehle nur für die Vorderseite lauten

```
gl.glBegin(GL.GL_QUAD_STRIP);
   gl.glTexCoord2f( 0.0f, 0.0f);
```

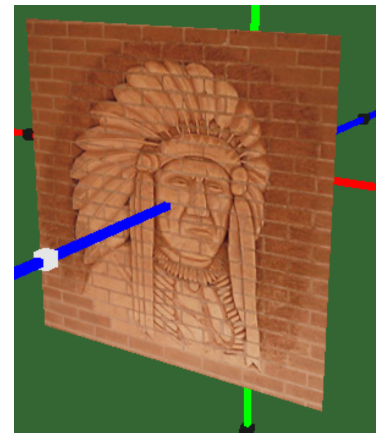
```
      gl.glVertex3f(-0.5f,0.5f,0.5f);
      gl.glTexCoord2f( 0.0f, 1.0f);
      gl.glVertex3f(-0.5f,-0.5f,0.5f);
      gl.glTexCoord2f( 1.0f, 0.0f);
      gl.glVertex3f(0.5f,0.5f,0.5f);
      gl.glTexCoord2f( 1.0f, 1.0f);
      gl.glVertex3f(0.5f,-0.5f,0.5f);
    gl.glEnd();
```

Dieses wurde so dem GL_Sourcerer-Output entnommen.

6.1.1 Textur-Koordinaten

Die OpenGL-Befehle im letzten Beispiel sind identisch jener für ein Viereck mittels `Quad_Strip`. Vor jedem `Vertex`-Befehl ist ein `TexCoord`-Befehl eingefügt, der die jeweiligen Textur-Koordinaten definiert. Die Zuordnung ist also die, dass die Textur-Koordinaten links oben (0,0) dem ersten Vertex links oben zugeordnet werden, die Textur-Koordinaten links unten (0,1) dem zweiten Vertex, usw. Die Fläche zwischen diesen Vertices wird mit den Texeln an den linear interpolierten Textur-Koordinaten aufgefüllt, also z.B. der Vertex (0, 0, 0.5) vorne in der Mitte der Textur-Koordinate (0.5, 0), usw.

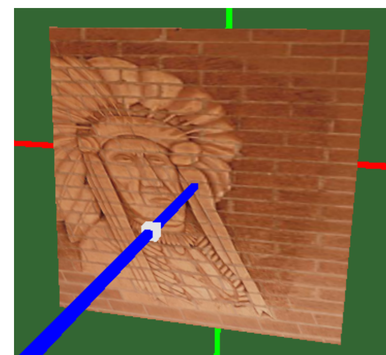
Zeichnen wir nur eine Fläche im GL_Sourcerer



```
< Quads
  texCoord 0.5 0
  vertex -0.5  0.5  0.5
  texCoord 0 1
  vertex -0.5 -0.5 0.5
  texCoord 1 1
  vertex  0.5 -0.5 0.5
  texCoord 1 0
  vertex  0.5  0.5 0.5
  texture data/bricks2.jpg
>
```

so lassen sich die Textur-Koordinaten beliebig variieren, z.B. wenn die erste Textur-Koordinate auf

```
  texCoord 0.5 0
```



gesetzt wird, so erhalten wir entsprechend das nebenstehende Bild. Man beachte, dass jetzt die Mitte der oberen Kante der Textur am linken oberen Vertex erscheint, während die anderen Punkte unverändert sind. Die dazwischenliegenden Punkte werden interpoliert, so dass ein verzerrtes Bild entsteht.

6.1.2 Automatische Berechnung von Textur-Koordinaten

Da die Berechnung der Textur-Koordinaten bei größeren Flächen recht umständlich ist, kann OpenGL die Berechnung automatisch durchführen. Im *GL_Sourcerer* schreiben wir dafür

```
< Quads
  objectLinear s 1 0 0 0.5
  objectLinear t 0 -1 0 0.5
  vertex -0.5  0.5  0.5
```

```
    vertex -0.5 -0.5 0.5
    vertex  0.5 -0.5 0.5
    vertex  0.5  0.5 0.5
    texture data/bricks2.jpg
  >
```

Die Funktion, welche die Textur-Koordinaten berechnet, wird hinter dem Parameternamen `objectLinear` spezifiziert, wobei pro Textur-Koordinate, die in OpenGL die Bezeichnungen *s*, *t*, *r*, *q* haben (entsprechend *x*, *y*, *z*, *w* bei Raumkoordinaten), jeweils eine lineare Funktion anzugeben ist. Die lineare Funktion lautet

Textur-Koordinate = $s \cdot x + t \cdot y + r \cdot z + q \cdot w$.

Für die *s*-Koordinate des letzten Beispiels erhalten wir also die Funktion: x+0.5, für die *t*-Koordinate: $-y$+0.5. Damit kann man auch bei größeren ebenen Flächen, bei denen aus den im letzten Kapitel genannten Gründen deutlich mehr Vertices spezifiziert werden müssen als nur die Geometriedaten, mit einer Anweisung je Textur-Koordinate auskommen.

Die zugehörigen OpenGL-Befehle lauten für die s-Koordinate

```
gl.glEnable(GL.GL_TEXTURE_GEN_S);
{ float[] v = {1.0f, 0.0f, 0.0f, 0.5f}; // definiere lineare Funktion
  gl.glTexGeni(GL.GL_S, GL.GL_TEXTURE_GEN_MODE, GL.GL_OBJECT_LINEAR);
  gl.glTexGenfv(GL.GL_S, GL.GL_OBJECT_PLANE, v,0); }
```

Nach dem ersten dieser Befehle wird die s-Koordinate für jeden folgenden Vertex aus der genannten Formel berechnet. Die explizite Angabe der Texturkoordinaten (`TexCoord2f( 0.0f, 0.0f);`) hat im folgenden für die s-Koordinate keine Bedeutung mehr und wird ignoriert. Analoges gilt für die t-Koordinate. Wird durch (`Disable(GL.GL_TEXTURE_GEN_S);`) die automatische Texturgenerierung wieder ausgeschaltet, so gelten wieder die expliziten Texturkoordinaten.

6.1.3 Andere Berechnungen von Textur-Koordinaten

Neben der Object-Linearen Textur-Koordinaten-Berechnung gibt es noch weitere Möglichkeiten, Textur-Koordinaten automatisch zu berechnen. Die Parameter in *GL_Sourcerer* lauten

```
    eyeLinear s 1 0 0 0.5
    eyeLinear t 0 -1 0 0.5
```

Die *Eye-lineare* Textur-Koordinaten-Berechnung transformiert die Koeffizienten (s,t,r,q) mit der inversen aktuellen Modelview-Matrix. Diese inverse Modelview-Matrix wird jeweils neu berechnet, wenn der OpenGL-Befehl

```
gl.glTexGenfv(GL.GL_S, GL.GL_OBJECT_PLANE, v,0);
```

ausgeführt wird. Ändert sich daher die Modelview-Matrix – d.h. bewegt sich die Kamera – so ändern sich auch die Parameter. Daher kann dieser Befehl i.d.R. nicht in einer Display-Liste verwendet werden, da die Berechnung jeweils von der aktuellen Modelview-Matrix abhängt, also nur bei Erstellung der Display-Liste durchgeführt wird.

Die Textur-Koordinaten-Berechnung mit dem Parameter SPHERE_MAP

```
SphereMap s t
```

berechnet zunächst den Reflektionsvektor *R* (siehe Kapitel 5.1.3 auf Seite 36) aus der Formel

$R = u - 2 \cdot n_f \cdot (n_f \cdot u)$

wobei u der Einheitsvektor in Richtung des Vertex ist und n_f der Normalenvektor, beide in Eye-Koordinaten. Es wird ein Faktor

$$m = 2 \cdot \sqrt{R_x^2 + R_y^2 + (1 + R_z)^2}$$

bestimmt und dann daraus die Texturkoordinaten bestimmt:

$$s = R_x / m + 1/2;$$
$$t = R_y / m + 1/2.$$

Die Texturkoordinaten definieren offenbar einen Punkt auf der Textur, der sich ungefähr aus der dem Reflektionswinkel ergibt, und somit eine Spiegelung auf der Textur vortäuschen kann. Allerdings ist dieser Effekt nicht sehr genau und hängt auch weitgehend nicht von der relativen Position des Betrachters ab. Für einfache Effekte ist dieses aber ausreichend.

Die REFLECTION MAP ist hierfür etwas besser geeignet, da sie ausschließlich den Richtungsvektor R verwendet, um die drei Koordinaten r, s, t zu berechnen.

```
ReflectionMap s t r
```

Die Abbildung dieser drei Koordinaten auf eine spezielle Texture, der *CubeTexture*, wird im nächsten Abschnitt beschrieben.

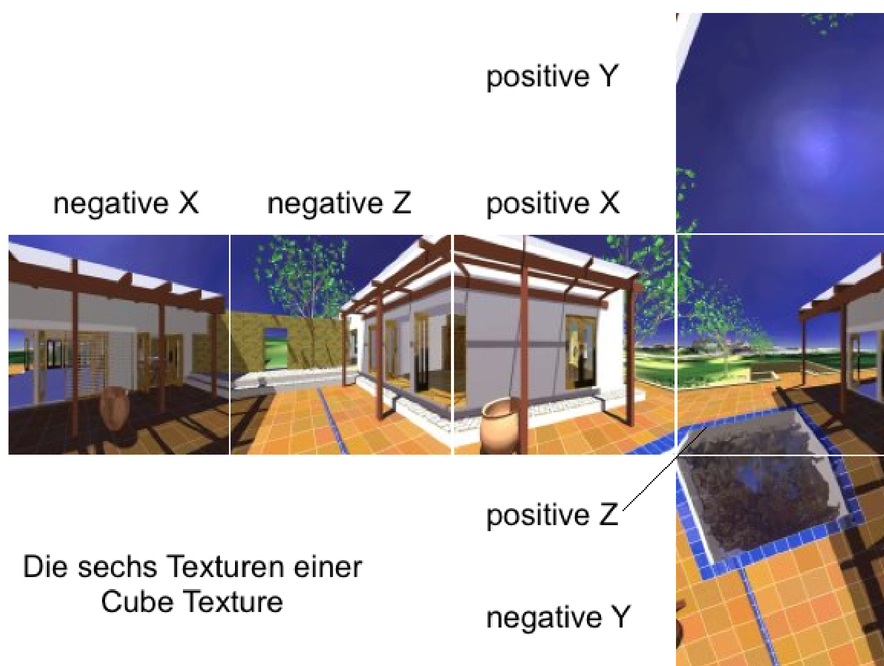
Die letzte Möglichkeit zur automatischen Texturkoordinaten-Berechnung ist die NormalMap:

```
NormalMap s t r
```

Hierbei werden die Normalen n_f im Eyespace berecnnet und deren Komponenten als Texturkoordinaten s, t, r verwendet.

6.2 Texturen im Raum

Texturen können auch für alle sechs Himmelsrichtungen (links, rechts, vorne, hinten, oben, unten) definiert werden, so dass einem Richtungsvektor im Raum eindeutig ein Texelwert zugeordnet werden kann. Dieses wird als *CubeTexture* bezeichnet, da die Texturen als Oberflächen eines Würfels aufgefasst werden können. Um Cube-Texturen zu definieren sind offensichtlich sechs Texturen anzugeben:



Die sechs Texturen einer Cube Texture

Im GL_Sourcerer lässt sich dieses relativ einfach durch den Parameter

```
cubetexture data/desert.png
```

erreichen Dann wird eine entsprechende Cubemap auf diese Oberfläche gezeichnet. Dazu sind in diesem Falle sechs Texturen mit den Namen

```
data/desert_positive_x.png
data/desert_negative_x.png
data/desert_positive_y.png
data/desert_negative_y.png
data/desert_positive_z.png
data/desert_negative_z.png
```

zur Verfügung zu stellen, welche die entsprechenden spiegelnden Bilder enthalten.

Die Texturkoordinaten werden automatisch berechnet, wenn

```
texturemap s t r
```

gesetzt ist. Es wird aus der betragsmäßig größten Koordinate die obige Textur ermittelt (ist z.B. s im Betrag maximal und von positivem Wert, dann wird die Textur **desert_positive_x.png** gewählt, ist s negativ dann wird **desert_negative_x.png** gewählt, usw.), und dann die beiden anderen Koordinaten normalisiert, um aus der gewählten Textur den Texelwert zu bestimmen. Dadurch scheint sich die Welt auf der Oberfläche des Objekts zu spiegeln.

Die entsprechenden OpenGL-Befehle lauten

```
gl.glEnable(GL.GL_TEXTURE_CUBE_MAP);
gl.glTexParameteri(GL.GL_TEXTURE_CUBE_MAP, GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR);
gl.glTexParameteri(GL.GL_TEXTURE_CUBE_MAP, GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR);
gl.glTexParameteri(GL.GL_TEXTURE_CUBE_MAP, GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
gl.glTexParameteri(GL.GL_TEXTURE_CUBE_MAP, GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);
gl.glTexParameteri(GL.GL_TEXTURE_CUBE_MAP, GL.GL_TEXTURE_WRAP_R, GL.GL_REPEAT);
{ int n = Texture.bindCubeTexture( "data/desert.png");
  gl.glBindTexture(GL.GL_TEXTURE_CUBE_MAP, n); }
gl.glBindTexture(GL.GL_TEXTURE_CUBE_MAP, 1);
gl.glEnable(GL.GL_TEXTURE_GEN_S);
gl.glTexGeni(GL.GL_S, GL.GL_TEXTURE_GEN_MODE, GL.GL_REFLECTION_MAP);
gl.glEnable(GL.GL_TEXTURE_GEN_T);
gl.glTexGeni(GL.GL_T, GL.GL_TEXTURE_GEN_MODE, GL.GL_REFLECTION_MAP);
gl.glEnable(GL.GL_TEXTURE_GEN_R);
gl.glTexGeni(GL.GL_R, GL.GL_TEXTURE_GEN_MODE, GL.GL_REFLECTION_MAP);
```

7 OpenGL 3.1

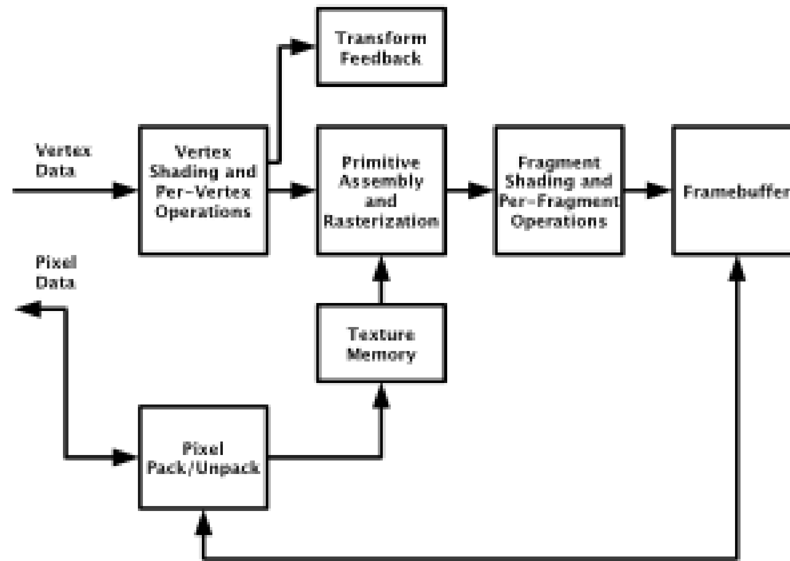
Mit der Einführung von OpenGL in der Version 3.1 haben sich einige Änderungen vollzogen, die insbesondere die Aufwärtskompatibilität betreffen; Anweisungen, die bisher erlaubt waren, sind nicht mehr zulässig bzw. gelten zumindest als veraltet, so dass sie evtl. in einer späteren Version verworfen werden. Dieses gilt nicht nur für einige wenige exotische Funktionen, sondern für vor allem für grundlegende Basisfunktionen, wie sie bis dahin in OpenGL verwendet wurden, z.B. das glBegin(..) .. glEnd() – Paar und die dazwischen zulässigen Befehle, einschließlich `glVertex3f(..)` und die damit zusammenhängenden Befehle wie `glNormal(..)` und `glColor(..)`, sowie das Konzept der DisplayListen. Um OpenGL weiterverwenden zu können, müssen daher viele bisherige Techniken umgestellt werden.

Die grundlegende Motivation für diesen drastischen Einschnitt ist sicherlich die Umstellung auf Shader-Programmierung; weitere Ziele sind die Vereinfachung der Schnittstelle (API) und die Durchsetzung weiterer nur teilweise neuer Konzepte, die in früheren Entwicklungen kaum verwendet wurden. Wer also weiter OpenGL programmieren möchte, muss sich wohl diesen Vorgaben beugen.

7.1 Grundlegende GL-Operationen

Die folgenden Ausführungen fassen die bisherigen Darstellungen zusammen und beschränken sich auf die neuen Schnittstellen in OpenGL.

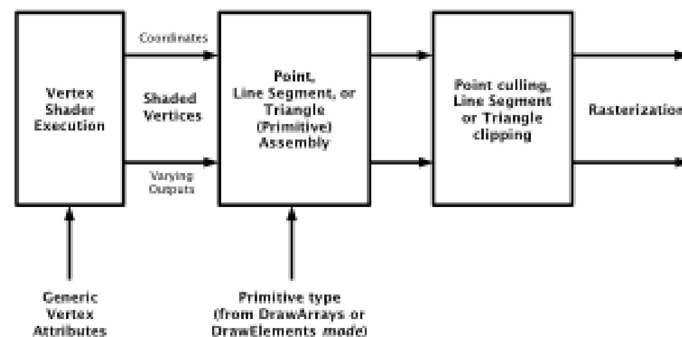
Die grundlegenden Funktionen von OpenGL kann man dem folgenden Bild entnehmen, welches die Reihenfolge – oder Pipeline – beschreibt, in der die Befehle ausgeführt werden. In der ersten Stufe werden die geometrischen Primitive Punkte, Liniensegmente oder Polygone verarbeitet, spezifiziert durch ihre Vertex-Daten an, die neben den Raumkoordinaten weitere Attribute wie Farben, Texturkoordinaten, Normalen usw. umfassen können; ggf. werden hier Transformationen und Beleuchtungsberechnung je Vertex durchgeführt. Diese werden entweder in der Standard-Pipeline oder im Vertex-Shader verarbeitet; das Ergebnis sind Primitive, welche entsprechend dem Ausgaberaum geklippt werden. Der Rasterisierer erzeugt eine Folge von Adressen und Werten im Frame-Buffer, wozu eine zweidimensionale Beschreibung von Punkten, Liniensegmenten oder Polygonen verwendet werden. In der nächsten Stufe, dem Fragment-Shader, wird jedes einzelne Fragment weiter bearbeitet, ehe sie in den Framebuffer geschrieben werden, der das Pixelbild auf dem Bildschirm repräsentiert. Das Schreiben in den Framebuffer hängt ggf von dem Inhalt des Tiefenpuffers, von bereits geschriebenen Werten beim Blending, sowie Maskierung oder andere logischen Operationen auf Fragmenten. Außerdem können die Daten zurückgelesen oder in andere Teile des Framebuffers kopiert werden, nachdem sie evtl. noch de/-kodiert wurden.



7.2 Primitive und Vertices

In OpenGL ab Version 3.1 werden geometrische Objekte nur noch mittels DrawArray(..) oder ähnlichen Befehlen gezeichnet. Es werden nur noch die folgenden sieben geometrischen Objekte unterstützt: Punkte, Liniensegmentfolgen und -schleifen, unterbrochene Liniensegmente, Dreiecksfolgen und -fächer sowie einzelne Dreiecke (*points, line segment strips*, *line segment loops*, *separated line segments*, *triangle strips*, *triangle fans* und *separated triangles*). Quads, Quadstrips und Polygone werden somit nicht mehr unterstützt, was sicherlich nicht besonders dramatisch ist.

Jeder Vertex wird mit mindestens einem generischen Vertexattribut spezifiziert, die jeweils aus bis zu vier Skalarwerten bestehen; Vertexattribute wie Normalen oder Farben können nur auf diese Weise an den Shader übergeben werden. Der Vertex und die generischen Vertexattribute werden im Vertex-Shader verarbeitet und erzeugen neben den homogenen Vertexpositionen (d.h. Vertexkoordinate im Clipping-Space) sowie Ausgaben auf Varying-Variablen.



Das Bild zeigt die Reihenfolge, in welcher Primitive aus einer Folge von Vertices gebildet werden. Abhängig von dem Primitiv-Typ werden aus der Folge von Vertices die Primitiven berechnet und die berechneten Vertex-Koordinaten der Primitive entweder verworfen (bei Punkten) oder ggf. abgeschnitten (geclippt bei Liniensegmenten oder Dreiecken). Beim Clippen können ggf. neue Vertices erzeugt werden, wenn ein Dreieck oder eine Linie den Clipping-Space-Würfel schneidet.

8 Shader

Um OpenGL anzuweisen, gewisse Funktionen auszuführen, müssen verschiedene Parameter gesetzt werden, was teilweise sehr unübersichtlich ist, da diese Parameter mit bestimmten Schlüsselwörtern beschrieben werden, und teilweise die Werte ebenfalls mit Konstanten bezeichnet werden.

Neuere Versionen von OpenGL gestatten es, verschiedene Funktionalitäten durch spezielle Programme, die als Shader bezeichnet werden, explizit zu spezifizieren. Im ersten Abschnitt dieses Kapitels erläutern wir die fest installierten Teile und die frei programmierbaren Teile der OpenGL-Pipeline. Danach werden die grundlegenden Funktionen von Shader-Programmen betrachtet.

8.1 Die OpenGL-Pipeline

Die OpenGL-Pipeline besteht aus folgenden Stufen, die

8.2 Die Shader-Programmiersprache

Die Shader-Programmiersprachen besitzen eine analoge Syntax und Semantik wie C, oder genauer C++, da etwa Überladung von Funktionen erlaubt ist; Syntax und Semantik ähneln insofern auch sehr stark Java. Wir präsentieren hier einen gewissen Ausschnitt aus der Sprache, auch als GLSLang bezeichnet, wobei wir die Version 1.20 zugrunde legen.

8.2.1 Beispiel für Shader-Programme

Eine Programmeinheit wird als Shader bezeichnet. Es gibt zwei grundsätzlich verschiedene Typen von Shadern, die als Vertex- und Fragment-Shader bezeichnet werden. Innerhalb eines Programms können mehrere dieser Shader-Programme gleichzeitig verwendet werden.

Als Beispiel seien ein einfacher *Vertex-Shader* und *Fragment-Shader* aufgelistet.

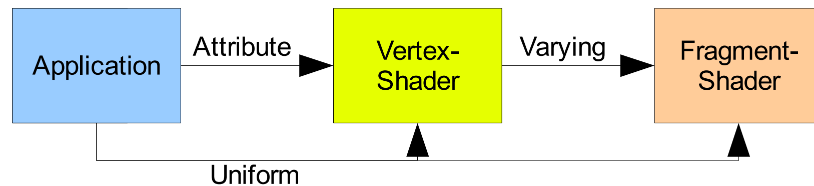
```
void main(void) { // here starts the main program of the Vertex Shader
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex; // assign position
  gl_FrontColor = gl_Color; // assign color
}
void main(void) { // here starts the main program of the Fragment Shader
  gl_FragColor = gl_Color; // assign a color to  gl_FragColor
}
```

Das erste Programm, der Vertex-Shader, verwendet mehrere implizit definierte (oder eingebaute) Variablen, nämlich eine 4x4-Matrix **`gl_ModelViewProjectionMatrix`** und einen Ortsvektor **`gl_Vertex`**, der die Objekt-Koordinaten des jeweiligen Vertex enthält, wie sie im Befehl **`glVertex4f(..)`** in der Anwendung definiert wurden. Der Operator `*` bedeutet hier Matrizenmultiplikation, was die erste Besonderheit dieser Sprache aufzeigt, dass nämlich Vektoren und Matrizen ähnlich wie in der Mathematik üblich mit eingebauten Befehlen verknüpft werden können.

Der erste Befehl des Vertex-Shaders weist der eingebauten *Varying-Variablen* **`gl_Position`** den Ortsvektor des Vertex in Clipping-Koordinaten zu, welcher aus dem Produkt auf der rechten Seite der Wertzuweisung berechnet wird. Der zweite Befehl des Vertex-Shaders weist der eingebauten Varying-Variablen **`gl_FrontColor`** den Wert der eingebauten *Attribut-Variablen* **`gl_Color`** zu, welche den in der Anwendung spezifizierten Wert der Funktion **`glColor4f(..)`** enthält.

Die hier gezeigten Befehle fassen zugleich die Hauptaufgabe des Vertex-Shaders zusammen, nämlich die Position eines Vertex in Clipping-Koordinaten anzugeben, sowie die Farbe des jeweiligen Vertex zu spezifizieren. Natürlich lassen sich sowohl die Position als auch die Farben vielfältig manipulieren, z.B. für die Lichtberechnung oder auch für eine Veränderung der Raumkoordinaten; aber die prinzipiellen Aufgaben des Vertex-Shaders sollten mit diesem Beispiel erst einmal verständlich sein. Eine weitere Aufgabe des Vertex-Shaders ist es, Varying-Variablen zu spezifizieren, was später genauer erläutert wird. Diese unterschiedlichen Typen von Variablen haben sehr verschiedene Aufgaben und werden daher unterschiedliche bezeichnet. Dem folgenden Bild, welches später genauer erläutert wird, kann man die Aufgaben dieser Variablen entnehmen.

Vertex-Shader
Setzt Position eines Vertex in Clipping-Koordinaten
Setzt Farbe des Vertex
Setzt Varying-Variablen für Fragment-Shader

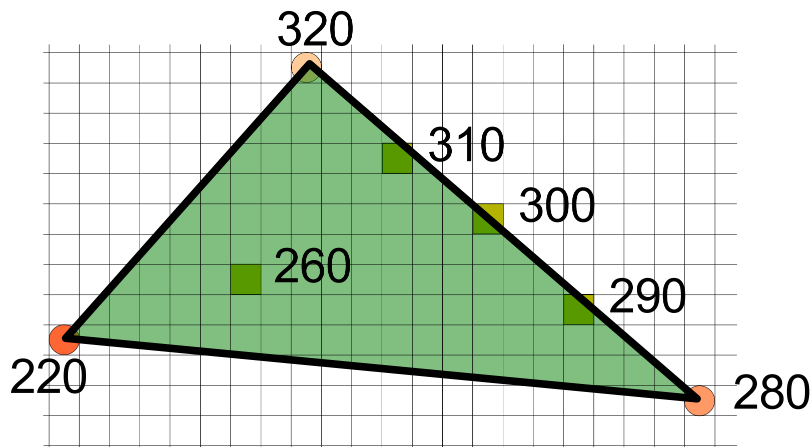


Die Aufgabe des Fragment-Shaders besteht nun darin, die Farbe eines Fragments – d.h. eines Pixels auf dem Bildschirm – festzulegen, d.h. jeder Durchlauf durch den Fragment-Shader berechnet die Farbe genau eines Fragments. Dazu ist das Zusammenspiel von Vertex- und Fragment-Shader zu verstehen. Der Vertex-Shader bearbeitet die Vertices, also die Eckpunkte von Dreiecken. Die Flächen dieser Dreiecke müssen dann mit Farben ausgemalt werden, wozu der Fragment-Shader die jeweilige Position der Pixel eines Dreiecks bestimmt. Dieser Vorgang wird als *Interpolation* bezeichnet, d.h. es werden zwischen den Eckpunkten des Dreiecks die Raumkoordinaten und natürlich auch die Bildschirmkoordinaten berechnet, an denen ein Fragment zu zeichnen ist.

Aus den Raumkoordinaten eines Fragments wird zusätzlich berechnet, welche Werte, z.B. Farben, dieses Fragment haben soll. Dazu wird aus den Abständen zu den jeweiligen Vertices (Eckpunten des Dreiecks) ein Gewicht bestimmt, mit dem die Werte (Farben usw.) in den einzelnen Eckpunten zu mitteln sind. Solche Werte werden mittels *Varying-Variablen* von dem Vertex-Shader an den Fragment-Shader übergeben. Beispielsweise wird ein Fragment genau in der Mitte zwischen den drei Vertices eines Dreiecks von jedem Vertex den gleichen Farbanteil (nämlich ein Drittel) erhalten. Dieses gilt nun für alle Varying-Variablen, also außer für die Raumkoordinaten und die Farben auch z.B. für Richtungsvektoren, wie den Normalen, oder vom Programmierer definierten Varying-Variablen.

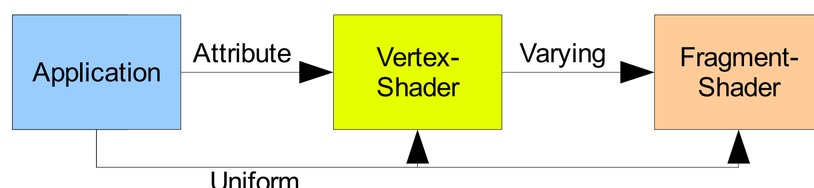
Fragment-Shader
Berechnet Farbe des Fragments
Setzt Farbe des Fragments

In dem folgenden Bild habe in dem oberen Vertex irgendeine Varying-Variable den Wert 320, im unteren rechten Vertex den Wert 280. In der Mitte zwischen diesen beiden Vertices hat denn die jeweilige Varying-Variable den Wert (320+280)/2 = 300. Analoges gilt für jedes Pixel auf dem Dreieck, so dass die Werte der Varying-Variablen über die gesamte Fläche linear interpoliert werden. Man beachte, dass die Varying-Variablen automatisch den interpolierten Wert erhalten; der Fragment-Shader arbeitet nur mit interpolierten Werten der Varying-Variablen, so dass der tatsächlich im Vertex-Shader gesetzte Wert einer Varying-Variable so niemals im Fragment-Shader ankommt.



Zeichnung 2: Interpolierte Werte im Fragment-Shader

Die Kommunikation zwischen dem Vertex- und dem Fragment-Shader geschieht über die Varying-Variablen, die entsprechend interpoliert werden. Die Kommunikation von der Anwendung zu Vertex- oder Fragment-Shader geschieht über *Uniform-Variablen*. Als dritte Art von Variablen gibt es *Vertex-Attribut-Variablen*. Diese werden je Vertex definiert und bestehen z.B. aus den Werten der Funktionsaufrufe **`glVertex4f(..)`**, **`glColor4f(..)`** oder **`glFogCoord4f(..)`**.



Die Vertex-Attribut-Variablen müssen in den Anwendungen mit einem Wert versehen werden und stehen dann sowohl im Vertex- als auch im Fragment-Shader zum Lesen bereit; sie dürfen in der Regel im Shader nicht beschrieben werden. Darüber hinaus lassen sich auch in der Anwendung eigene Vertex-Attribut-Variablen definieren, deren Anzahl in der Regel allerdings beschränkt ist. Die anwendungsdefinierten Vertex-Attribut-Variablen übergeben ihre Werte genau wie die anderen Vertex-Attribut-Variablen genau dann an den Vertex-Shader, wenn die Funktion **`glVertex4f(..)`** aufgerufen wird; ihre Werte sind daher jeweils vorher zu setzen, wie auch die Werte der anderen Vertex-Attribut-Variablen, also z.B. **`glColor4f(..)`** usw.

8.2.2 Datentypen

Die Sprache GLSL (Graphik Library Shading Language) enthält eine Reihe von Datentypen, wie sie auch aus Programmiersprachen wie Java oder C++ bekannt sind. Dazu gehören ganze Zahlen (**`int`**), Gleitpunktzahlen (**`float`**), Vektoren unterschiedlicher Länge (**`vec2`**, **`vec3`**, **`vec4`**), oder Matrizen unterschiedlicher Größe (**`mat2`**, **`mat3`**, **`mat4`**). Es gibt auch logische Werte wie in Java (**`bool`**) und Datensätze (**`struct`**) wie aus C bekannt. Ganze und Gleitpunktzahlen haben alle ein festes Format (das allerdings implementierungsabhängig sein kann). Zahlenkonstante werden wie in C und Java geschrieben: 1, 3, 1002; 1.0, 3.1415, 1.5E9. Oktalzahlen werden mit führender Null,

Hexadezimalzahlen mit führender 0x (Null, x) geschrieben: 045, 0x12d3. Eine Typanpassung wird nicht durchgeführt, d.h. Gleitpunktvariablen können nur Gleitpunktkonstante verwenden.

Vektoren basieren auf Gleitpunktzahlen. Sollen Vektoren mit ganzzahligen Elementen (**ivec2**, **ivec3**, **ivec4**) oder mit logischen Elementen (**bvec2**, **bvec3**, **bvec4**) gebildet werden, so muss der Typ entsprechend gekennzeichnet werden. Der Zugriff auf die Elemente von Vektoren ist gut durchdacht und sehr komfortabel:

```
vec4 vektor;
vektor = vec4(1.5, 2.0, 3.0, 1.0);
if(vektor.x  == 1.0) ...
if(vektor.xy  == vec2(1.0, 2.9)) ...
vec2 vek2 = vektor.zw;        // == (3.0, 1.0)
vec3 vek3 = vektor.xyz;       // == (1.5, 2.0, 3.0)
vec3 vek4 = vec3(vek2,0.3);   // == (3.0, 1.0, 0.3)
vek3.xy = vek2;               // == (3.0, 1.0, 3.0,
vek2 = vek2.yx;               // == (1.0, 3.0) vertausche Komponenten
```

Der angehängte Selektor **.xyzw** wählt entsprechend die erste bis vierte Komponente aus. Dabei dürfen auf der rechten Seite gleiche Indizes vorkommen, z.B. **vektor.yyx**, oder die Indizes in vertauschter Reihenfolge stehen. Auf der linken Seite müssen alle Indizes verschieden sein, die Reihenfolge ist jedoch beliebig. Statt **.xyzw** kann auch **.rgba** oder **.stpq** geschrieben werden, was semantisch äquivalent ist, da die Sprache nur Vektoren, nicht jedoch Farben, Ortsvektoren oder Texturkoordinaten unterscheidet (man beachte **p** statt **r**, da **r** auch in Farben verwendet wird). Einzelne Komponenten können auch mit der klassischen Index-Notation **vek[2]** selektiert werden, wobei **vek[0]** die erste Komponente auswählt. Bei der Selektion und Komposition von Vektoren muss darauf geachtet werden, dass die Vektoren links und rechts die gleiche Länge haben.

Neben Vektoren können auch Felder beliebiger Länge gebildet werden:

```
float feld[10]; feld[0] = 1.0; feld[5] = feld[0];
```

Vektoren können addiert und multipliziert werden, wobei komponentenweise Addition und Multiplikation gemeint ist. Um das Punktprodukt zweier gleichlanger Vektoren zu bilden, ist die Funktion **dot()** zu verwenden, für das Kreuzprodukt **cross()**.

Matrizen gibt es nur mit Gleitpunktzahlen.

```
mat4 matrix;
matrix[0] = vec4(1.5, 2.0, 3.0, 0.0);
matrix[1] = vec4(1.5, 2.0, 3.0, 0.0);
matrix[2] = vec4(1.5, 2.0, 3.0, 0.0);
matrix[3] = vec4(1.5, 2.0, 3.0, 0.0);
vek2 = vek2.yx;               // == (1.0, 3.0) vertausche Komponenten
```

Die Indizierung einer Matrix wählt eine Spalte (*column*) der Matrix aus. Matrizen können auch eine andere Zeilen- als Spaltenzahl haben. Die Typangaben lauten dann **mat2x3**, **mat4x3**, usw. wobei die erste Zahl die Anzahl der Spalten, die zweite die Anzahl der Zeilen angibt.

8.2.3 Konstruktoren

Ein Konstruktor erzeugt ein Objekt, wie einen Vektor oder ein Feld. Es gibt keine Cast-Operatoren wie in Pointersprachen wie C; stattdessen sind Konstruktoren zu verwenden, die jeweils ein neues Objekt erzeugen.

Der Konstruktor besteht aus dem Typnamen und der Liste der Anfangsdaten in runden Klammern, wie sie in den letzten Beispielen verwendet wurden. Für Matrizen gelten seit Version 2.0 der GLSlang einige erweiterte Regeln; für Matrizen ist die Anordnung der Daten major-column, d.h. die ersten n Werte bilden die erste Spalte (column) der Matrix **mat*n***.

```
mat2(vec2, vec2);             // one column per argument
mat3(vec3, vec3, vec3);       // one column per argument
mat4(vec4, vec4, vec4, vec4); // one column per argument
mat3x2(vec2, vec2, vec2);     // one column per argument
mat2(float, float,            // first column
     float, float);           // second column
mat3(float, float, float,     // first column
     float, float, float,     // second column
     float, float, float);    // third column
mat4(float, float, float, float,  // first column
     float, float, float, float,  // second column
     float, float, float, float,  // third column
     float, float, float, float); // fourth column
mat2x3(vec2, float,               // first column
       vec2, float);              // second column
mat3x3(mat4x4);  // takes the upper-left 3x3 of the mat4x4
mat2x2(mat4x2);  // takes the upper-left 2x2 of the mat4x4, last row is 0,0
mat4x4(mat3x3);  // puts the mat3x3 in the upper-left, sets the lower right
                 //    component to 1, and the rest to 0
```

8.2.4 Qualifier

Datentypen können bei der Deklaration einen Qualifier erhalten. Diese sind **const**, **attribute**, **uniform**, **varying** und **centroids varying**.

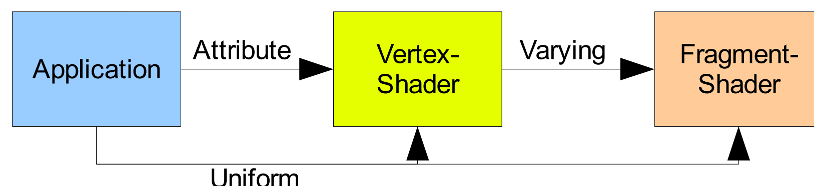
Mit **const** kann eine Variable zu einer Konstanten werden, darf also nach der Initialisierung nicht mehr verändert werden.

```
const float PI = 3.14159265;
```

Der Qualifier **attribute** spezifiziert eine Variable, die pro Vertex einen anderen Wert erhalten kann. Eingebaute Variablen dieses Typs sind **gl_Vertex** und **gl_Color**, welche die Werte der OpenGL-Funktionen **glVertex4f(..)** und **glColor4f(..)** übernehmen (und die für jeden Vertex andere Werte übergeben können).

```
attribute float diffuseColor;
```

Die Anwendung kann eine beschränkte Anzahl von Vertex-Attribute-Variablen selbst definieren, um z.B. diffuse Farben pro Vertex zu spezifizieren. Vertex-Attribute-Variablen dürfen nur im Vertex-Shader verwendet werden.



Zeichnung 3: Qualifizierte Variablen

Der Qualifier **varying** spezifiziert eine Variable, die pro *Fragment* einen anderen Wert besitzen kann. Sie muss im Vertex-Shader einen Wert erhalten und kann im Fragment-Shader nur gelesen

werden. Der Wert im Fragment-Shader ist der interpolierte Wert der jeweiligen Vertices, welche das zu zeichnende Dreieck bestimmen. Um interpolieren zu können, müssen die Werte Gleitpunktzahlen sein, auch bei Vektoren und Matrizen. Structures können nicht als **`varying`** qualifiziert werden.

```
varying float distance;
```

Der Qualifier **`uniform`** spezifiziert eine globale Variable, die pro *Primitive* (also Polygon, Quads usw.) einen anderen Wert erhalten kann. Eingebaute Variablen dieser Art sind z.B. **`gl_ModelViewMatrix`**, **`gl_NormalMatrix`**, **`gl_LightSource[]`**, usw. Diese Liste ist relativ lang und kann dem Standard oder der Kurzreferenz entnommen werden.

```
uniform float lightPosition;
```

8.2.5 Operatoren

Die Multiplikation von Vektoren mit Matrizen oder von Matrizen untereinander werden als lineare Multiplikation im mathematischen Sinne interpretiert. Ansonsten können Matrizen und Vektoren gleicher Größe miteinander addiert und multipliziert, subtrahiert und dividiert werden, wobei wieder komponentenweise Operation gemeint ist. Analoges gilt für die meisten anderen arithmetischen Operatoren wie ++ (inkrementieren), oder – – (dekrementieren). Außerdem sind diese Operationen mit einem Skalar sowie einem Vektor, Matrix oder allgemeinem Feld erlaubt; die Bedeutung ist die komponentenweise Addition, Multiplikation, usw.

```
matrix0 = matrix1 + matrix2; // komponentenweise Addition
matrix0 += matrix2; // komponentenweise Addition
matrix0 = matrix1 * matrix2; // lineare Multiplikation (Matrizenprodukt)
matrix0 *= matrix1; // lineare Multiplikation (Matrizenprodukt)
matrix0 = matrix1 / matrix2; // komponentenweise Division
matrix0 = matrix1 / 100.0; // komponentenweise Division durch Skalar 100.0
vektor0 = vektor1 + vektor2; // komponentenweise Addition
vektor0 = vektor1 + 100.0; // komponentenweise Addition mit Skalar 100.0
matrix0 = matrix1 * vektor2; // lineare Multiplikation (Matrizenprodukt)
```

8.2.6 Eingebaute Funktionen

GLSL bietet einige eingebaute Funktionen an, die das Programmieren erleichtern sollen und eine bessere Dokumentation ermöglichen.

Trigonometrische Funktionen

Die folgenden trigonometrischen Funktionen arbeiten auf Floats oder Vektoren und sind jeweils komponentenweise gemeint; die Ergebnisse sind entsprechend auch Floats oder Vektoren.

`radians`(...) (von Grad nach Bogenmaß), **`degree`** (...) (von Bogenmaß nach Grad),

`sin`(...), **`cos`**(...), **`tan`**(...), **`asin`**(...), **`acos`**(...), **`atan`**(...), **`atan`**(..., ...).

Der Arcus-Tangens kann mit einem oder zwei Parametern (jeweils Float oder Vektor) verwendet werden mit der gleichen Funktionalität wie aus Java bekannt.

Exponentialfunktionen

Die folgenden reellen Funktionen arbeiten auf Floats oder Vektoren und sind jeweils komponentenweise gemeint; die Ergebnisse sind entsprechend auch Floats oder Vektoren.

pow(...,...), **exp**(...), **log**(...), **exp2**(...), **log2**(...), **sqrt**(...), **inversesqrt**(...).

Die Basis ist ***e*** bzw. 2 bei **exp2**, **log2**.

Allgemeine Funktionen

Die folgenden allgemeinen Funktionen (...) arbeiten auf Floats oder Vektoren und sind jeweils komponentenweise gemeint; die Ergebnisse sind entsprechend auch Floats oder Vektoren. Ist nur Float angegeben, so bezieht er sich auf alle Komponenten eines entsprechenden Vektors.

abs(...), **sign**(...), **floor**(...), **ceil**(...), **fract**(...), **mod**(...,**float**), **mod**(...,...),

min(...,**float**), **min**(...,...), **max**(...,**float**), **max**(...,...), **clamp**(...,**float**,**float**), **clamp**(...,...,...),

mix(...,...,**float**), **mix**(...,...,...), **step**(...,...), **smoothstep**(...,...,...).

Die Funktionen sind analog den entsprechenden Java-Funktionen (und den C-Funktionen, soweit sie standardisiert sind). **clamp** beschränkt den Wertebereich zwischen unteren und oberen Grenze; **mix** wichtet die ersten beiden Werte mit den Faktoren (1-*a*) und *a*, wobei *a* der dritte Werte ist. **step**(*a*,*x*) liefert 0 für $x<a$ und sonst 1. **smoothstep**(*a*,*b*,*x*) liefert 0 für $x<a$, 1 für $x>b$, und dazwischen einen interpolierten Wert.

Geometrische Funktionen

Die folgenden geometrischen Funktionen (...) arbeiten auf Floats oder Vektoren und sind jeweils komponentenweise gemeint; die Ergebnisse sind entsprechend auch Floats oder Vektoren.

length(...), **distance**(...,...), **dot**(...,...), **cross**(**vec3**,**vec3**), **normalize**(...), **ftransform**(),

faceforward(...,...,...), **reflect**(...,...), **refract**(...,...,**float**).

Die Funktionen sind weitgehend selbsterklärend. **ftransform**() kann nur im Vertex-Shader verwendet werden und erzeugt die Raumkoordinate im Clip-Space. **reflect**() und **refract**(...,...,**float**) können zur Berechnung von reflektiertem bzw.. gebeugtem Licht verwendet werden.

Matrix-Funktionen

Die Matrix-Funktionen mit dem '*'-Operator berechnen das lineare Produkt zweier Matrizen, bzw. eines Vektors mit einer Matrix. Um zwei Matrizen komponentenweise zu multiplizieren, kann die Funktion **matrixCompMult**(...,...) verwendet werden.

Vektor-Funktionen

Die folgenden Funktionen dienen dem komponentenweisen Vergleich von Vektorelementen; das Ergebnis ist ein boolscher Vektor mit entsprechend vielen Komponenten.

LessThan(...,...), **lessThanEqual**(...,...), **greaterThan**(...,...), **greaterThanEqual**(...,...),

equal(...,...), **notEqual**(...,...), **any**(...), **all**(...), **not**(...).

Die Funktionen **any**(**bvec**...) und **all**(**bvec**...) überprüfen, ob ein (logisches Oder) oder alle Komponenten (logisches Und) wahr sind. **not**(**bvec**...) negiert den Inhalt des Argument-Vektors.

Fragment-Funktionen

Fragment-Funktionen dienen der Berechnung von Steigungen und können nur im Fragment-Shader verwendet werden.

dFdx(...), **dFdy**(...), **fwidth**(...),

Die Funktionen sind implementierungsabhängig und bilden die Ableitungen eines Samplers in x- bzw. y-Richtung bzw. die Summe der Absolutbeträge der Ableitungen.

Sampler-Funktionen

Sampler-Funktionen dienen dem Zugriff auf eine Textur. OpenGL überlädt die Funktionen, so dass für einen Zugriff **texture**(SamplerName, TexturKoordinate) ausreicht. Frühere Versionen verwendeten **texture1D**(), **texture2D**() usw., die mittlerweile als veraltet gelten. Allerdings gibt es verschiedene weitere Varianten, die hier aber nicht im Detail erläutert werden. Genaueres entnehme man dem Standard (8.7 Texture Lookup Functions).

textureSize(...), **textureProj**(...), **textureLod**(...), **textureOffset**(...),

textureProjOffset(...), **textureLodOffset**(...), **textureProjLodOffset**(...),

texelFetch(...), **texelFetchOffset(...)**, **textureGrad**(...),

textureGradOffset(...), **textureProjGrad**(...), **textureProjGradOffset**(...).

Noise-Funktionen

Zufalls-Zahlen werden mit den noise{1234}(genType)-Funktionen erzeugt. Als Parameter sind Floats oder Vektoren erlaubt. Die Ergebnisse sind entsprechend **float**, **vec2**, **vec3** und **vec4**.

noise1(...), **noise2**(...), **noise3**(...), **noise4**(...).

8.3 Vertex-Shader

Sobald ein Vertex in der Applikation definiert wird, werden seine Attribute an den Vertex-Shader geschickt. Attribute sind die bekannten Größen wie u.a. Color, Material, Normal, Texturkoordinaten usw. sowie nicht zuletzt die Raumkoordinaten, die im glVertex-Befehl als Parameter anzugeben sind; bis auf die Raumkoordinaten werden sämtliche Attribute im Zustandsautomaten von OpenGL spezifiziert.

8.3.1 Die Variable gl_Position

Die wichtigste Aufgabe des Vertex-Shaders ist die Angabe der Raumkoordinate des Vertex im *Clippingspace*. Dazu ist als eingebaute Uniform-Variable *gl_Vertex* zu verwenden, welche jedoch die Raumkoordinate im *Objectspace* angibt. Dieser Wert ist daher mit der *Modelview*-Matrix und der *Projection*-Matrix zu multiplizieren, was im Vertex-Shader entweder durch den Befehl

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

oder durch den Befehl

```
gl_Position = ftransform();
```

durchgeführt werden kann. Die zweite Variante ist in der Regel zu bevorzugen, da sie u.U. schneller und ggf. numerisch genauer ist. Tatsächlich schreibt die Shader-Sprache vor, dass der eingebauten Varying-Variablen *gl_Position* ein Wert zuzuweisen ist; andernfalls kann der Compiler eine Fehlermeldung ausgeben. Das ist natürlich sinnvoll, da mindestens die Raumkoordinaten eines zu zeichnenden Dreiecks anzugeben sind. Alle anderen Größen, insbesondere die Farben, können implizit (z.B. schwarz) definiert werden.

8.4 Fragment-Shader

Die Dreiecke, welche drei Vertices spezifizieren, müssen mit Farbwerten – entweder mit identischen Farben, mit Texturen oder durch Lichtberechnung oder sonstwie verschieden modulierten Farbanteilen – versehen werden. Diese Berechnung geschieht im Fragment-Shader. Der Fragment-Shader verwendet die gleiche Programmiersprache wie der Vertex-Shader, hat allerdings Zugriff auf eine Reihe spezieller Variablen, die nur hier definiert sind.

8.4.1 Varying-Variablen und interpolierte Werte

Das wichtigste Konzept eines Fragment-Shaders ist die Interpolation von Varying-Werten. In der Regel werden die drei Eckpunkte eines Dreiecks im Vertex-Shader berechnet. Die Aufgabe des Fragment-Shaders ist es, dieses Dreieck 'auszumalen', d.h. je sichtbarem Pixel auf den Bildschirm einen Farbpunkt zu zeichnen, der innerhalb des Dreiecks liegt, und der die entsprechende Farbe hat, welche im Vertex-Shader spezifiziert wurde. Dazu werden die Farben und alle anderen Varying-Werte, die vom Vertex-Shader übergeben werden, interpoliert entsprechend dem Abstand zu den Eckpunkten der jeweiligen Vertices. Liegt beispielsweise ein Fragment gerade auf einem Eckpunkt, so wird der Farbwert des Eckpunkts verwendet; liegt er genau zwischen allen drei Eckpunkten, so wird der gemittelte Farbwert von allen drei Eckpunkten genommen. Dieses geschieht automatisch und kann auch nicht abgestellt werden. Andere Werte wie Uniform-Variablen, die von der Anwendung programmiert werden, werden nicht interpoliert.

Die Werte der Varying-Variablen werden daher als *interpolierte Werte* bezeichnet, da sie tatsächlich in der Regel (bis auf ggf. die Eckpunkte) niemals direkt vom Vertex-Shader erzeugt wurden. Sollen sie unverändert übergeben werden, so muss entweder jeder Vertex des Dreiecks den gleichen Werte erzeugen, oder die Variablen werden als Uniform-Variablen übergeben.

Die Varying-Variablen müssen von dem Vertex-Shader gesetzt werden, wenn sie im Fragment-Shader verwendet werden; andernfalls wird eine Fehlermeldung erzeugt und das Programm läuft nicht.

8.4.2 Eingebaute Varying-Variablen

Die folgenden Varying-Variablen sind fest eingebaut, d.h. in jedem Fragment-Shader verfügbar und dürfen auch nicht deklariert werden.

```
varying vec4 gl_Color;
varying vec4 gl_SecondaryColor;
varying vec4 gl_TexCoord[];
varying float gl_FogFragCoord;
```

Die ersten beiden Variablen werden interpoliert aus dem Vertex-Shader übernommen. Die dritte Variable kann die Werte der entsprechenden Texturkoordinaten enthalten, wenn sie im Vertex-Shader gesetzt wurden. Ist das nicht der Fall, so enthält **gl_TexCoord[n]** den Wert der entsprechenden Texturkoordinate der n-ten Textureinheit. Die letzte Variable enthält die Nebelkoordinate, wobei im **GL_FRAGMENT_DEPTH**-Modus die z-Koordinate des Fragments im Eye-Modus übergeben wird, und im **GL_FOG_COORDINATE**-Modus die interpolierte Nebelkoordinate.

Die eingebaute Variable **gl_FragCoord** gibt die Fensterkoordinaten des Fragments an; die z-Komponente enthält dabei den Wert des Tiefenpuffers; die vierte Komponente enthält den Wert 1/w.

Die eingebaute boolsche Variable **gl_FrontFacing** ist wahr, wenn der jeweilige Vertex-Shader **gl_FrontColor** bzw. **gl_FrontSecondaryColor** gesetzt wurde; wurde **gl_BackColor** bzw. **gl_BackSecondaryColor** gesetzt, so hat die Variable **gl_FrontFacing** den Wert falsch. Beide Variablen dürfen nur gelesen werden.

8.4.3 Die Variable gl_FragColor

Die wichtigste Aufgabe eines Fragment-Shaders ist das Festlegen der Farbe eines Fragments, d.h. eines Pixels. Dazu ist die Variable **gl_FragColor** in einem Fragment-Shader mindestens einmal beschrieben wird. Geschieht das nicht, so ist die Farbe des Fragments undefiniert. Die eingebaute Varying-Variable **gl_Color** übernimmt in der Regel vom Vertex-Shader die interpolierte Farbe, so dass der einfachste Fragment-Shader ist:

```
void main(void) { // here starts the main program
  gl_FragColor = gl_Color; // assign a color to  gl_FragColor
}
```

Man erinnere sich, dass **gl_Color** eine interpolierte Variable ist; deren Wert ist also der gemittelte Wert der möglicherweise unterschiedlichen Farbwerte der drei Eckpunkte eines Dreiecks entsprechend ihres Abstand; daher wird in der Regel eine andere Farbe als die in einem Vertex-Shader gesetzte gezeichnet. Die Varying-Variable, deren Wert im Vertex-Shader gesetzt werden muss, lautet in der Regel **gl_FrontColor** (es kann auch **gl_BackColor** gesetzt werden).

8.4.4 Weitere eingebaute Varying-Variablen

Neben **gl_FragColor** können die eingebauten Variablen **gl_FragData** und **gl_FragDepth** geschrieben werden; außerdem ist es möglich, keine dieser Variablen zu beschreiben, indem z.B. der Fragment-Shader durch den eingebauten Befehl **discard;** vorzeitig beendet wird. In diesem Fall wird kein Wert geschrieben, also der aktuelle Wert des Fragments unverändert gelassen.

Wird **gl_FragDepth** beschrieben, so wird dieses als der Tiefenwert in den Tiefenpuffer übernommen (nachdem der übliche Tiefenpuffertest durchgeführt wurde). Wird dieser Wert nicht geschrieben, so wird der Wert der z-Koordinate als Tiefenwert genommen.

Die Varying-Variable **gl_FragData** erlaubt das Schreiben von Daten in einen externen Puffer.

8.5 Shader-Programme in GL_Sourcerer

Im GL_Sourcerer können Shader-Programme eingefügt werden. Es ist vorgesehen, genau einen Vertex-Shader und einen Fragment-Shader für den Ablauf zur Verfügung zu stellen.

8.5.1 Struktur von Shadern in GL_Sourcerer

Es werden zwei Arten von Shadern definiert:

```
< vertexshader myShader
  startProgram
  void main(void) {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_FrontColor = gl_Color;
  }
>
< fragmentshader myShader
  startProgram
  void main(void) {
    gl_FragColor = gl_Color;
  }
>
```

Der Befehl *vertexshader* bzw. *fragmentshader* kann von einem Namen gefolgt werden, so dass mehrere verschiedene Shader für eine Szene definiert werden können. Nach dem Parameter *start-*

Program folgen dann bis zum Ende des Befehls die Shader-Programme. Da die Shader-Programme direkt übernommen werden, ist in diesen auf korrekte Syntax zu achten und Groß- und Kleinschreibung zu unterscheiden.

Um einen Shader für das Zeichnen eines Objekts zu aktivieren, muss der Parameter shader mit dem Shader-Namen als Parameterwert in dem Objekt definiert werden, z.B.

```
shader myShader
```

Es wird dann nur dieses Objekt mit diesem Shader gezeichnet. Sollen alle nicht mit diesem Parameter gekennzeichneten Objekte mit einem allgemeinen Shader gezeichnet werden, kann ein Shader mit dem Namen *Standard* definiert werden. Objekte, die keinen Shader-Befehl enthalten, werden mit der Fixed-Pipeline gezeichnet, wenn es keinen Shader mit dem Namen *Standard* gibt.

8.5.2 Uniform-Variablen

Float-Zahlen, float-Vektoren und Texturen

Um Uniformvariablen, welche Zahlenwerte, Vektoren oder Texturen repräsentieren, in Shadern zu verwenden, müssen diese vor *startProgram* deklariert werden:

```
< fragmentshader myShader
    texture0 alpha
    texture1 beta
    uniform newColor
    uniform lightPosition
  startProgram
    uniform sampler2d alpha, beta;
    uniform vec4 newColor;
    uniform vec4 lightPosition;
    void main(void) {
      ....
      gl_FragColor = gl_Color + newColor;
  }
>
```

Die Texturen müssen wie in OpenGL in den Objekt-Beschreibungen den jeweiligen Textur-Einheiten (*texture units*) zugeordnet werden. Die anderen Zahlen werden innerhalb eines Objekts, welches den gleichen Shader (hier *myShader*) verwendet, in das Programm über den speziellen Befehl

```
shader myShader
uniform newColor 1 1 0 1
```

eingegeben; es können nur float-Zahlen bzw. float-Vektoren übergeben werden, die während des Programmlaufs konstant sind. Eine Ausnahme bildet die Standardvariable *lightPosition*, welche nach der Auswahl des Buttons *Light* (oder) eine interaktive Werteübergabe erlaubt. *lightPosition* darf nicht in den Objekten definiert werden! Andere Möglichkeiten, um variable Werte zu übergeben werden in den nächsten Abschnitten beschrieben.

Man beachte, dass in diesem Falle Groß- und Kleinschreibung relevant sind, da die Shader dieses verlangen. Außerdem muss die Anzahl der Vektorelemente übereinstimmen, im letzten Beispiel 4, da im Shader-Programm *vec4 newColor* deklariert wurde. Ist das nicht der Fall, kann es zu fehlerhaftem Verhalten der Shader während der Laufzeit kommen.

Felder (arrayUniform)

Statt einer Zahl oder eines Vektors lassen sich auch Felder von float-Zahlen oder Vektoren definieren und Werte übergeben:

```
    uniform field
    startProgram
    uniform vec3 field[5];
  void main(void) { ...
```

In dem jeweiligen Objekt muss dieses Feld mit Anfangswerten versehen werden.

```
arrayUniform 5 field  0 0 1  0.5 0 0.75  0.5 0 0.5  0.75 0 0.25   1 0 0
```

Die Anzahl der Parameter muss der Vektorgröße mal der Dimension des Felds entsprechen, im letzten Beispiel also 3 mal 5, da die Vektoren die Größe 3 haben (*vec3*), und da das Feld 5 Elemente enthält, was mit der Zahl hinter *arrayUniform* spezifiziert wird.

Diskrete Werte mittels Tasten (keyUniform)

Soll ein Eingabewert für einen Shader geändert werden, ohne dass die Szene neu geladen werden muss, so kann dieses über eine Tastensteuerung erreicht werden.

```
keyUniform <key> <keyName> <startValue> <increment> <decrement>
```

key kennzeichnet die Taste, z.B. C oder das '+'-Zeichen. *keyName* gibt einen Namen an, wie er im Shader verwendet werden soll. *startValue* ist der Anfangswert der Variablen, welcher um das *Inkrement* bei Drücken der Taste C mit Umschaltung (oder bei fester Umschaltung) erhöht wird, und um das *Dekrement* bei Drücken der Taste C ohne Umschaltung erniedrigt wird. Z.B. erhält man bei

```
<fragmentShader keyTest
    uniform  keyC
   startProgram
    uniform float keyC;
    void main(void) {    ...    }
...
< cube
shader keyTest
keyUniform    c    keyC    0   1    0.5
>
```

die Möglichkeit, den Wert 0 auf der Variablen *keyC* zu übergeben, und diesen Wert während des Programmlaufs durch die Taste 'C' um 1 zu erhöhen, bzw. durch 'c' um 0.5 zu erniedrigen.

Ist das Zeichen kein Buchstabe, so wird ein Toggle-Wert angegeben, der abwechselnd die Werte 0 und 1 annimmt und z.B. zum Ein- und Ausschalten eines Features benutzt werden kann. Die übergebenen Werte und die Variablen sind immer float-Werte.

Analoge Werte mittels Maus (mouse)

Um einen analogen Wert einzugeben, kann die Maus verwendet werden. Die Uniform-Variable heißt *mouse* und ist vom Typ *vec2*. Wird die Strg (cntrl)-Taste gedrückt gehalten, so wird die Position des Mauszeigers auf die *mouse*-Variable gegeben, wobei der Punkt im Fenster links oben dem Wert (0.0,0.0) entspricht, der Punkt rechts unten (1.0,1.0). Durch die Notation *mouse.x* bzw. *mouse.y* können die Werte getrennt gelesen werden.

```
<fragmentShader keyTest
    uniform  mouse
   startProgram
    uniform vec2 mouse;
    void main(void) {
          ...  mouse.x ...
          ...  mouse.y ...}
```

Zeitwerte mittels time

Um die aktuelle Zeit seit Start des Programms in einen Shader zu übergeben, kann die Uniform-Variable *time* verwendet werden. Sie ist nur im Shader, in dem sie verwendet werden soll, zu deklarieren, nicht im zu zeichnenden Objekt, da sie für alle Objekte gleichermaßen gilt.

```
<fragmentShader timeTest
    uniform  time
    startProgram
    uniform float time;
    void main(void) { if(fract(time)<0.5) ... }
```

8.5.3 Attribute-Variablen

Um Daten pro Vertex an einen Vertex-Shader zu übergeben, können Attribute-Variablen verwendet werden. Diese werden analog den Attributen zu Vertices genau dann übergeben, wenn der `Vertex`-Befehl aufgerufen wird. OpenGL kennt eine Reihe von eingebauten Attribute-Variablen, die somit nicht explizit behandelt werden müssen.

```
attribute vec4 gl_Color;
attribute vec4 gl_SecondaryColor;
attribute vec4 gl_Normal;
attribute vec4 gl_Vertex;
attribute vec4 gl_MultiTexCoord0..(gl_MaxTextureCoords);
attribute vec4 gl_FogCoord;
```

Neben diesen Werten kann eine beschränkte Anzahl von Werten an den Vertex-Shader übergeben werden. Diese Werte werden im Shader durch

```
attribute vec4 meineVariable;
```

definiert. Im GL_Sourcerer ist eine solche durch

```
attribute meineVariable 0 3 4 5
```

vor dem jeweiligen Vertex-Befehl in einem Vertex-basierten Objekt anzugeben (Quads, Polygons, etc.).

```
attribute vec3 diffuse;
void main(void) {
  gl_Position = ftransform();
  gl_FrontColor = diffuse;
}
...
normal 0 0 1
attribute diffuse 0.2 0.3 0.9  // setze Attribut-Wert
vertex   -1 1 0 // übergebe Raumkoordinate und Attribut-Wert
```

8.6 Einfache Shader-Programme

Als einführende Beispiele werden jetzt einige sehr einfache Shader-Programme vorgestellt. Diese dienen der Einübung in die Programmierung und dem Kennenlernen der syntaktischen Feinheiten dieser Programmiersprachen.

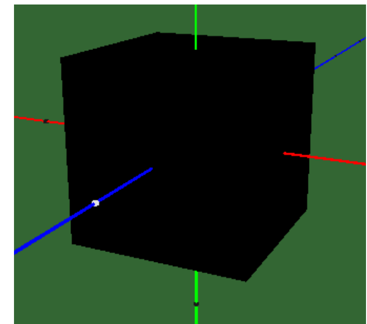
8.6.1 Das einfachste Shader-Programm

Shader-Programme unterteilen sich in Vertex- und Fragment-Shader. Die ersten berechnen ausschließlich die Raumkoordinaten (Vertices) der jeweiligen Objekte. Diese können beliebig manipuliert werden, wenngleich in der Mehrheit der Fälle die Raumkoordinaten durch die Anwendung (im Befehl `glVertex4f(..)`) definiert werden. Das einfachste Programm, welches beide Shader verwendet, ist folgendes

```
void main(void) { // here starts the main program of the Vertex Shader
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex; // assign position
}
void main(void) { // here starts the main program of the Fragment Shader
  gl_FragColor = gl_Color; // assign a color to  gl_FragColor
}
```

Um dieses im GL_Sourcerer laufen zu lassen, ist ein einfaches Objekt zu definieren, z.B. ein Würfel, und mit diesem Shader-Programm laufen zu lassen.

```
< vertexShader simpleShader
  startProgram
  void main(void) { // here starts main program
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  }
>
< fragmentShader simpleShader
  startProgram
     void main(void) { // here starts main program
     gl_FragColor = gl_Color ;
  }
>
< cube
  shader simpleShader
>
```



Das Ergebnis ist ein schwarzer Einheitswürfel. Man teste dieses auch mit anderen Objekten, z.B. GLUT-Objekten. In jedem Fall ist das Ergebnis einfach ein schwarzes Objekt.

Die Funktion des Shaders legt dieses Resultat auch nahe. Als erstes werden die Vertex-Koordinaten im Clippingspace berechnet und an die eingebaute Varying-Variable `gl_Position` übergeben; diese Wertzuweisung an `gl_Position` ist notwendig, da sonst bereits der Compiler eine Übersetzung ablehnt. Eine Farbe wird nicht gesetzt, so dass ein Standardwert – hier schwarz (RGBA=0,0,0,1) – verwendet wird. Im Fragment-Shader wird lediglich die Farbe auf die Fragment-Farbe gesetzt. Soll ein Fragment gezeichnet werden, so berechnet der Fragmentshader zunächst die Raumkoordinaten der jeweiligen Dreiecksfläche und führt für jedes Fragment, welches in dieser Fläche liegt, das Fragment-Shader-Programm aus, d.h. setzt die Farbe auf schwarz. Den Rest erledigt OpenGL, indem es das Fragment abhängig von der Tiefenfunktion zeichnet, den Tiefenpuffer ggf. umsetzt, usw.

8.6.2 Ändern der Farben in Shadern

Möchte man ein Objekt in anderer Farbe zeichnen, so kann man die Farbe im Fragment-Shader ändern. Man ersetzt einfach den vorhandenen Befehl durch

```
gl_FragColor = vec4( 1.0, 1.0, 0.0, 1.0) ; // gelb
```

Alternativ könnte man aber auch die Farbe im Vertex-Shader ändern. Dazu fügt man den Befehl

```
gl_FrontColor = vec4( 0.0, 1.0, 1.0, 1.0) ; // cyan
```

hinter dem ersten Befehl ein (oder davor – die Reihenfolge, in der die Variablen gesetzt werden, ist in diesem Fall natürlich egal). In jedem Fall zeigt der Würfel die neue Farbe an, so dass hier die Möglichkeit aufgezeigt ist, die Farben von Fragmenten beliebig zu ändern.

In der Regel soll die Anwendung bestimmen, welche Farbe der jeweilige Vertex hat. Durch den OpenGL-Befehl `glColor4f(..)` wird in der Anwendung eine Farbe gesetzt; dieses wird als *Attribut-Variable* bezeichnet, da diese Farbe je Vertex gesetzt werden kann. Im Objekt ist im *GL-Sourcerer* entsprechend die Zeile

```
color 1 1 0  // gl.glColor4f(1.0f,1.0f,0.0f, 1.0f);
```

einzufügen. Dieser Wert lässt sich im Vertex-Shader durch die Variable `gl_Color` auslesen.

```
gl_FrontColor = gl_Color; // (1.0f,1.0f,0.0f, 1.0f)
```

Die auf `gl_FrontColor` gesetzte Farbe kommt im Fragment-Shader auf der Varying-Variablen `gl_Color` an. Dort ist diese entsprechend an die Variable `gl_FragColor` weiterzureichen. Allerdings könnte sie auch dort noch verändert werden, z.B. durch

```
gl_FragColor = gl_Color + vec4( 0.0, 0.0, 0.5, 0.0);
```

Hier wird also der Blauanteil der Farbe um 0.5 erhöht. Sollte der Wert größer als 1.0 werden, so schneidet OpenGL dieses automatisch auf 1.0 ab (*clipping*). Man erinnere sich, dass die Vektoraddition komponentenweise durchgeführt wird. Stattdessen könnte man auch einen Faktor wählen, mit dem die Farben verändert werden, z.B. erhält man mit

```
gl_FragColor = gl_Color * 0.5;
```

eine Dämpfung des Lichts auf die Hälfte; hier wird jede Komponente des Vektors `gl_Color` mit dem Faktor `0.5` multipliziert (auch der Alpha-Kanal!). Tatsächlich können für die Zuweisung an `gl_FragColor` beliebige korrekte Ausdrücke, die als Ergebnis einen Float-Vektor der Länge 4 haben, verwendet werden, z.B.

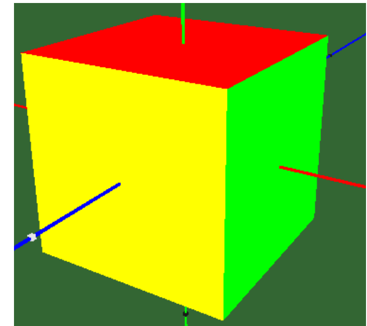
```
gl_FragColor = (gl_Color + vec4( 0.0, 0.0, 0.5, 0.0) ) * 0.5;
```

und andere.

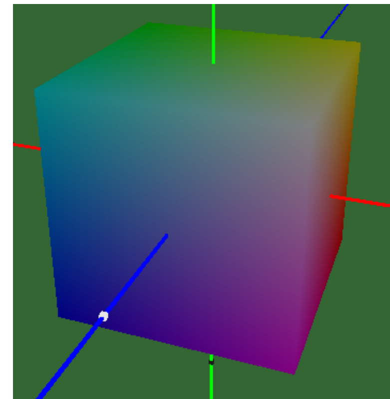
8.6.3 Farbberechnungen im Vertex-Shader

Bisher war das Ergebnis immer ein einfarbiger Würfel, was nicht sehr plastisch aussieht. Will man pro Fläche eine andere Farbe setzen, so kann man dieses im GL_Sourcerer mit den entsprechenden Parametern ColorFront, ColorBack usw. einfach erreichen. Es werden dann für die jeweiligen Vertices die Farben übernommen und die Flächen entsprechend gesetzt; wir gehen von dem folgenden Programm aus:

```
< vertexShader simpleShader
  startProgram
  void main(void) { // here starts main program
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_FrontColor = gl_Color;
  }
>
< fragmentShader simpleShader
  startProgram
     void main(void) { // here starts main program
     gl_FragColor = gl_Color ;
  }
>
< cube
  ColorFront 1 1 0
  ColorUp 1 0 0
  ColorRight 0 1 0
  shader simpleShader
>
```



Das Ergebnis ist ein farbiger Einheitswürfel. Man kann die Farben aber auch von anderen Größen abhängig machen, z.B. von den Werten der Ortskoordinaten, d.h. wir ersetzen den zweiten Befehl im Vertex-Shader durch



```
gl_FrontColor = gl_Vertex;
```

Das Ergebnis ist das nebenstehende Bild, also ein durchaus farbenfroher Würfel. Die Raumkoordinaten des Würfels werden somit als Farbwerte interpretiert; es können zwar auch negative Werte vorkommen, die aber grundsätzlich von OpenGL im Bereich [0,1] geclippt werden. Mit der Funktion `clamp` können die Werte explizit entsprechend abgeschnitten werden.

```
gl_FrontColor = clamp( gl_Vertex*2.0 , 0.0, 1.0 );
```

Eine andere Möglichkeit dieses Ergebnis zu erhalten, sind die Funktionen `min` und `max`. Diese sind für gleiche Vektortypen oder für Vektoren und Float-Zahlen als zweiten Parameter definiert.

```
gl_FrontColor = max( min( gl_Vertex*2.0 , 1.0), 0.0 );
```

Der Vergleich mit der oberen bzw. unteren Grenze erfolgt wieder komponentenweise. Wie bereits gesagt, werden Farben jedoch immer automatisch auf den zulässigen Bereich eingeschränkt.

Statt der Ortskoordinaten können beliebige andere Vektoren verwendet werden, z.B. Richtungsvektoren wie die Normalen.

```
gl_FrontColor=vec4(abs(gl_Normal.x),abs(gl_Normal.y),abs(gl_Normal.z), 1.0);
```

In diesem Fall hängt die Färbung von der Orientierung der Oberflächen ab, wenn die Normalen richtig gesetzt sind, was für einige Anwendungen recht nützlich sein kann. Insgesamt sollen diese Beispiele zeigen, dass die Farben im Shader nahezu beliebig gesetzt werden können.

8.6.4 Farbberechnungen im Fragment-Shader

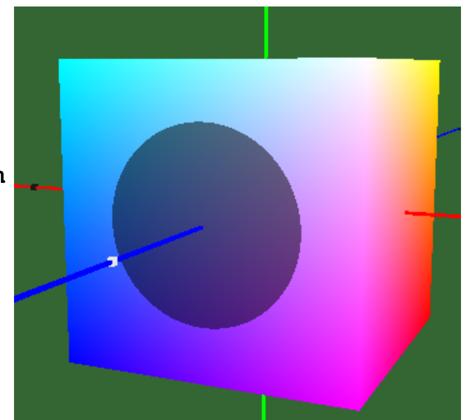
Bisher war die Färbung auf die Eckpunkte, also die Vertices, bezogen. Das führt u.U. zu Artefakten, insbesondere bei der Lichtberechnung, wenn die Beleuchtung der Eckpunkte über eine Fläche interpoliert wird. Man kann Farben aber auch im Fragment-Shader berechnen, indem pro Fragment eine

eigene Farbberechnung durchgeführt wird. Dadurch kann in vielen Fällen die Färbung einer Oberfläche sehr viel gleichmäßiger durchgeführt werden.

Um im Fragment-Shader Farben je Pixel zu berechnen, muss in der Regel im Fragment-Shader bekannt sein, an welcher Stelle der zu zeichnenden Oberfläche sich das zu zeichnende Fragment befindet. Dieses Problem lässt sich auf verschiedene Weisen lösen. Die einfachste Möglichkeit ist es, Texturkoordinaten zu verwenden, da diese in der Regel linear interpoliert werden, abhängig von den Raumkoordinaten. Eine andere Möglichkeit ist es, aus den Vertices die jeweiligen Ortskoordinaten der Objekte zu bestimmen. Die erste Möglichkeit wird später bei Texturen und anderen Techniken, z.B. Parallax-Mapping besprochen. Die zweite Methode soll hier kurz angedeutet werden.

Der Vertex-Shader kennt die Raumkoordinaten der jeweiligen Eckpunkte; diese können an den Fragment-Shader übergeben werden, wo sie interpoliert ankommen. Soll beispielsweise die Vorderseite des Würfels, die dadurch gekennzeichnet ist, dass $z = 0.5$, besonders behandelt werden, so kann dieses durch eine entsprechende Abfrage durchgeführt werden.

```
< vertexShader bigDot
  startProgram
  varying vec3 location;
  void main(void) {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_FrontColor = clamp( gl_Vertex*2.0 , 0.0, 1.0 );
    location = vec3(gl_Vertex);
  }
>
< fragmentShader bigDot
  startProgram
  varying vec3 location;
  void main(void) { // here starts the main program
    if(location.z>=0.499) {
      float x = location.x, y = location.y;
      if(x*x+y*y<0.1)
           gl_FragColor = gl_Color*0.5;
      else gl_FragColor = gl_Color;
    } else gl_FragColor = gl_Color;
  }
 >
```



Die Varying-Variable **location** enthält im Fragment-Shader die interpolierten Ortskoordinaten. Da der z-Wert für die Vorderseite konstant 0.5 ist, kann der Fall dass die Vorderseite gezeichnet wird, entsprechend abgefragt werden; hier muss wegen der Rundungsfehler eine kleine Toleranz zugelassen werden. Durch die Bedingung $x^2+y^2 < 0.1$ kann dann im entsprechenden Abstand vom Mittelpunkt eine andere Färbung – hier eine Dämpfung auf die Hälfte – angegeben werden. Man kann diese Bedingung variieren, z.B. $|x|+|y| < 0.3$, oder den Dämpfungsfaktor entfernungsabhängig gestalten. Dieses lässt sich für alle sechs Seiten durch entsprechende Fallunterscheidungen durchführen. Z.B. ließe sich auf die Weise ein Würfel (mit Augen 1 bis 6) ausschließlich im Fragment-Shader berechnen.

Der große Vorteil der Berechnung von Pixeln im Fragment-Shader ist die deutlich verbesserte Qualität; jeder Pixelwert kann genau bestimmt und berechnet werden. Nachteilig für die Berechnung im Fragment-Shader ist die schlechtere Performance, da jetzt für jedes Fragment ein etwas aufwändigeres Programm durchlaufen werden muss. Allerdings relativiert sich dieser Aufwand, da dieses tatsächlich nur je sichtbarem Pixel durchgeführt wird, und nicht wie im Vertex-Shader immer für jeden Vertex berechnet wird, auch wenn dieser gar nicht zu sehen ist. Moderne Systeme können diesen zusätzlichen Aufwand jedoch meistern, so dass wegen der höheren Qualität heute meistens per-Fragment-Lighting berechnet wird.

8.6.5 Berechnung von Raumkoordinaten im Vertex-Shader

Da Shader eigene Programmiersprachen darstellen, sind alle berechneten Werte völlig frei veränderbar. So lassen sich auch die Raumkoordinaten im Vertex-Shader verändern. Als Beispiel einer konstanten Änderung aller Vertices könnte beispielsweise durch die Befehle

```
vec4 vertex = vec4( 0.0, 0.0, 0.0, 1.0);
vertex.xyz = gl_Vertex.xyz + 0.5;
gl_Position = gl_ModelViewProjectionMatrix * vertex ;
```

eine Verschiebung um den Vektor `(0.5,0.5,0.5)` erreicht werden; die Operation in der zweiten Zeile bedeutet, dass die ersten drei Komponenten von **`vertex`** jeweils um 0.5 erhöht werden; die vierte Komponente muss natürlich den Wert 1 behalten. Würde das nicht geschehen, so erhielte man eine (i.d.R. ungewollte) 'Normalisierung' der Koordinaten; der Leser sollte das selbst ausprobieren!

Wir können mit dem GL_Sourcerer einfach dynamisch geänderte Werte in den Shader eingeben, z.B. über die Maus; siehe hierzu 8.5.2 auf Seite 61. Die zweite Zeile ist dann zu ersetzen durch

```
vertex.xyz = gl_Vertex.xyz + mouse.x;
```

Jetzt lässt sich mit der Taste <Strg> und der horizontalen Mausposition der Würfel verschieben. Wird in der letzten Zeile dass '+' durch '*' ersetzt, so wird der Würfel entsprechend skaliert. Alternativ ließe sich auch ein Vektor mit den x,y-Koordinaten der Maus verwenden, z.B.

```
vertex.xyz = gl_Vertex.xyz + vec3(mouse.x,mouse.y,0.0);
```

Der Vertex-Shader kann nicht die Anzahl der Vertices verändern, nur deren Werte, also die Raumkoordinaten. Daher können geometrische Formen beliebig verändert werden, solange die Anzahl der 'Eckpunkte' unverändert bleibt. Aus dem Würfel lässt sich beispielsweise eine Pyramide mittels

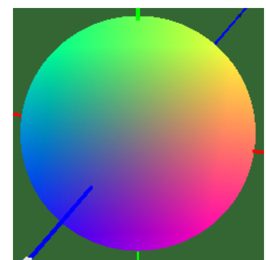
```
vec4 vertex = gl_Vertex;
if(vertex.y > 0.0)
   vertex.xyz = vec3(vertex.x*mouse.x*2.0,vertex.y,vertex.z*mouse.x*2.0);
gl_Position = gl_ModelViewProjectionMatrix * vertex;
```

erzeugen. Man beachte, dass in diesem Fall andere Größen wie Normalen nicht automatisch verändert werden, so dass diese ggf. zu korrigieren sind.

8.6.6 Berechnung von Raumkoordinaten im Fragment-Shader

Überraschenderweise lassen sich im Fragment-Shader sehr wohl eigene Konturen erzeugen, die deutlich mehr Vertices berücksichtigen können als an den Vertex-Shader übergeben wurden. Im folgenden wird eine ideale Kugel erzeugt, deren Kontur pro Fragment der einer Kugel folgt.

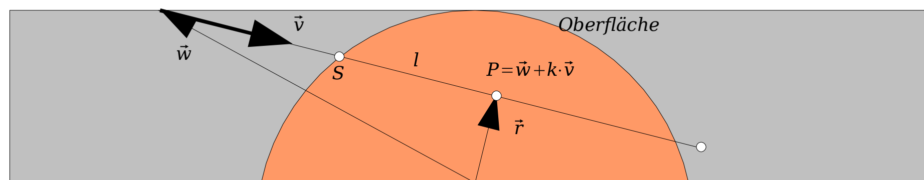
Wir wollen eine volle Kugel zeichnen; der Vertex-Shader übergibt dazu einen Würfel, innerhalb welchem die Kugel-Koordinaten berechnet werden. Das folgende Programm leistet dieses.



```
< VertexShader colorSphere
  startProgram
    varying vec3 eyeDir;
    varying vec3 fragmentCoord;
    void main(void) {
      gl_Position = ftransform();
      fragmentCoord = gl_Vertex.xyz;
      eyeDir = vec3(gl_ModelViewMatrix * gl_Vertex) * gl_NormalMatrix;
```

```
    }
  >
  < FragmentShader colorSphere
    startProgram
      varying vec3 eyeDir;
      varying vec3 fragmentCoord;
      void main(void) { // here starts the main program
        float sphereRadius = 0.5;  // radius of sphere
        vec3 v = normalize(eyeDir);
        vec3 r = fragmentCoord - v * dot(v,fragmentCoord);
        if(length(r) > sphereRadius) discard; // not hit of view to sphere
        gl_FragColor = vec4(0.5 + 1.4*r.xyz,1.0); // draw any color
      }
  >
  < cube
    shader colorSphere
  >
```

Wenn ein Fragment gezeichnet werden soll, so kann statt des Fragments an der Oberfläche eines Rechtecks mit der jeweiligen Raumkoordinate ein Fragment an einer beliebigen anderen Stelle im Raum berechnet werden, wenn der Betrachtungsvektor bekannt ist.



Sei also $\vec{v}$ der normalisierte Richtungsvektor von der Kamera (d.h. dem Ursprung des Eye-Koordinatensystems) auf die Objektkoordinate $\vec{w}$. Wir suchen einen Faktor k gesucht, so dass $\vec{r}=\vec{w}+k\cdot\vec{v}$ der senkrechte Vektor vom Mittelpunkt der Kugel auf den Betrachtungsvektor $\vec{v}$ ist.

$$\vec{v}\circ\vec{r}=\vec{v}\circ(\vec{w}+k\cdot\vec{v})=0\,,$$
$$k=\frac{-\vec{v}\circ\vec{w}}{\vec{v}\circ\vec{v}}.$$

Ist die Länge des Vektors $\vec{r}$ kleiner als der Radius der Kugel, so schneidet der Betrachtungsvektor die Kugeloberfläche, also ist diese im Fragment zu sehen. Den exakten Schnittpunkt S mit der Kugeloberfläche kann man durch die Länge l der halben Sehne in der Kugel bestimmen. Durch irgendein Verfahren wird dann die konstante oder variable Färbung der Kugeloberfläche berechnet und gerendert.

In dem Beispiel wird die Berechnung in Objektkoordinaten durchgeführt, d.h. der Mittelpunkt der Kugel liegt im Urspung. Daher muss auch der Betrachtungsvektor $\vec{v}$ in Objektkoordinaten vorliegen. Das Programm bestimmt zunächst auf **`fragmentCoord`** die Koordinaten der Würfeloberfläche; da diese interpoliert an den Fragment-Shader übergeben werden, erhält die entsprechende Variable **`fragmentCoord`** im Fragment-Shader den Wert der Raumkoordinate der Würfeloberfläche in Objektkoordinaten. Um auch den Betrachtungsvektor $\vec{v}$ in Objektkoordinaten zu erhalten, wird hier die Raumkoordinate in Eye-Koordinaten (**`gl_ModelViewMatrix*gl_Vertex`**) mit der Normalenmatrix **`gl_NormalMatrix`** multipliziert. Die Normalenmatrix ist definiert als die inverse transponierte Matrix der oberen linken 3x3-Modelviewmatrix. Um daher die Rücktransformation aus den Eye-Koordinaten in die Objektkoordinaten durchzuführen, sind die Raumkoordinaten mit der Normalenmatrix zu multiplizieren. Die Reihenfolge der Operanden ergibt sich daraus, dass die Multiplikation eines (Zeilen)-Vektors mit der Matrix gleich dem transponierten Vektor der Multiplikation der Matrix mit dem (Spalten)-Vektor darstellt.

$$(\hat{m}\times\vec{r})^T=\vec{r}^T\times\hat{m}^T.$$

Man beachte, dass in dieser Sprache nicht zwischen Spalten- und Zeilenvektoren unterschieden wird. Alternativ ließen sich die Berechnungen auch in Eye-Koordinaten durchführen.

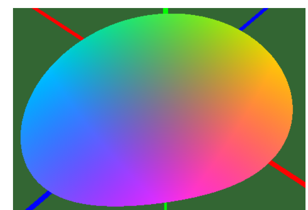
```
< VertexShader colorSphere
  startProgram
    varying vec3 null;
    varying vec3 frag;
    void main(void) {
      gl_Position = ftransform();
      null = gl_ModelViewMatrix[3].xyz;
      frag = (gl_ModelViewMatrix * gl_Vertex).xyz;
    }
>
< FragmentShader colorSphere
  startProgram
    varying vec3 null;
    varying vec3 frag;
    void main(void) { // here starts the main program
      float sphereRadius = 0.5;  // radius of sphere
      vec3 v = normalize(frag);
      float k = dot(null,v)/dot(v,v);
      vec3 r = k*v - null;
      float len = length(r);
      if(len > sphereRadius) discard; // no hit of view to sphere
      gl_FragColor =  vec4(0.5+ 2.0*r.xyz,1.0); // draw any color
    }
>
```

In diesem Beispiel wird der Ursprung der Objektkoordinaten in die Eye-Koordinaten transformiert, ebenso wie die Raumkoordinaten, welche gleich dem Betrachtungsvektor sind. Der Abstandsvektor $\vec{r}$ vom Nullpunkt $\vec{n}$ in Objektkoordinaten errechnet sich dann aus der Differenz

$$\vec{r}=k\cdot\vec{v}-\vec{n},$$
$$\vec{r}\circ\vec{v}=(k\cdot\vec{v}-\vec{n})\circ\vec{v}=0,$$
$$k=\frac{\vec{n}\circ\vec{v}}{\vec{v}\circ\vec{v}}.$$

Die Länge dieses Vektors $\vec{r}$ kann wieder verwendet werden um zu testen, ob der Abstand kleiner ist als der Kugeldurchmesser, die Lichtstrahl somit die Kugeloberfläche schneidet. Der einzige Unterschied zur ersten Version ist, dass der Vektor $\vec{r}$ nicht mehr in Objektkoordinaten angegeben wird, sondern in Eye-Koordinaten, und somit die Färbung der Oberfläche nicht mehr von dem Kamerastandpunkt abhängt.

Einfache Variationen zu diesem Beispiel ergeben sich u.a. durch Änderungen des Längenvektors. Für ein Ellipsoid kann man z.B. die folgende Länge nehmen:



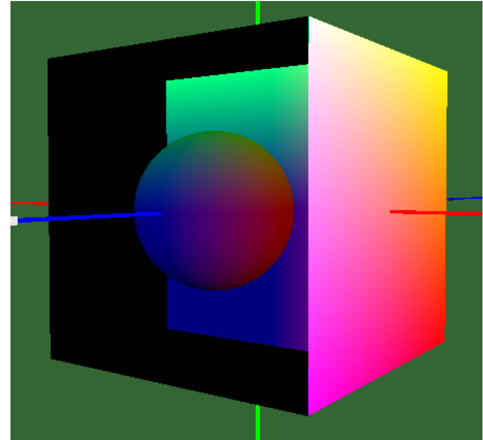
```
length = sqrt(r.x*r.x*1.5 + r.y*r.y*5.0 + r.z*r.z*1.0);
```

Das Ergebnis ist eine langgezogene Linse. Wird der Koeffizient von *y* auf null gesetzt, so wird nur der Abstand von der Y-Achse gemessen, als im Prinzip ein Rohr definiert. Wegen der Ränder am Würfelrand sieht das Ergebnis jedoch etwas anders aus. Im nächsten Beispiel wird nur beim Blick auf eine Seite des Würfels eine kleinere Kugel gezeichnet, die anderen Seiten bleiben unverändert die Seiten des Würfels:

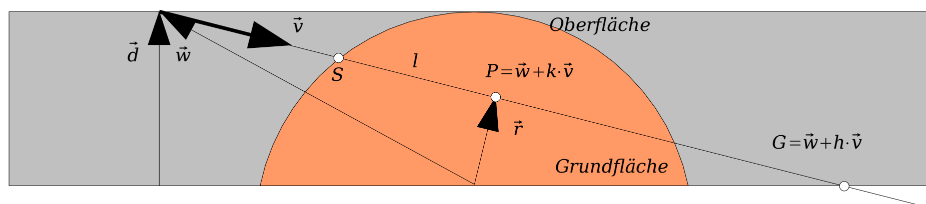
```
< VertexShader parallaxMapping
  uniform tangent
  startProgram
    varying vec3 eyeDir;
```

```
    varying vec3 location;
    uniform vec3 tangent;

    void main(void) {
      gl_Position = ftransform();
      gl_FrontColor = clamp( gl_Vertex*2.0 , 0.0, 1.0 );
      location = gl_Vertex.xyz;
      eyeDir = vec3(gl_ModelViewMatrix * gl_Vertex) * gl_NormalMatrix;
    }
>
< FragmentShader parallaxMapping
    uniform mouse
  startProgram
    varying vec3 eyeDir;
    varying vec3 location;
    uniform vec2 mouse;
    void main(void) { // in program
      float dif = mouse.x;
      float radius = mouse.y/2.0;
      float hoehe = 0.5 - radius;
      vec3 origin = vec3(0.0, 0.0, hoehe);
      if(location.z>=0.499) {
        vec3 v = normalize(eyeDir);
        vec3 w = location - origin;
        float k = -dot(v,w);
        vec3 r = w + v * k;
        float rLen = length(r);
        float l = sqrt(radius*radius - rLen*rLen);
        vec3 rr = w + v * (k-l);
        if(rLen <= radius && rr.z > radius-dif){
          gl_FragColor = vec4(2.0*(rr).xyz,1.0);
        } else {
          float h = - dif*dif/dot(v,vec3(0.0,0.0,dif) );
          vec3 vh = w + v * h;
          if(-0.5 <= vh.x && vh.x <= 0.5 && -0.5 <= vh.y && vh.y <= 0.5)
            gl_FragColor = vec4(2.0*(vh).xy,0.5,1.0);
        }
      } else
      gl_FragColor = gl_Color;
    }
>
```



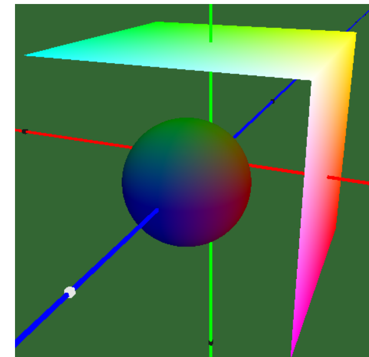
Wenn ein Fragment gezeichnet werden soll, so kann zunächst wie im ersten Beispiel getestet werden, ob die Länge des Vektors $\vec{r}$ kleiner als der Radius der Kugel ist; dann schneidet der Betrachtungsvektor die Kugeloberfläche, also ist diese im Fragment zu sehen. Der Schnittpunkt S mit der Kugeloberfläche kann durch die Länge l der halben Sehne in der Kugel berechnet werden. Durch irgendein Verfahren wird dann die konstante oder variable Färbung der Kugeloberfläche berechnet und gerendert.



Schneidet der Betrachtungsvektor die Kugeloberfläche nicht, so wird der Schnittpunkt G mit der Grundfläche berechnet: $\vec{w}+h\cdot\vec{v}$. Dazu wird der Tiefenvektor $\vec{d}$ unterhalb der Oberfläche des Würfels zur Grundfläche verwendet.

$$\vec{d} \circ (\vec{d} + h \cdot \vec{v}) = 0,$$
$$h = \frac{-\vec{d} \circ \vec{d}}{\vec{v} \circ \vec{d}}.$$

Auch hier kann die Färbung der Grundfläche nach beliebigen Kriterien berechnet werden.



Details entnehme man dem Programm. Es wurde keine Beleuchtung verwendet. Da jedoch sowohl die Raumkoordinate jedes gezeichneten Pixels als ggf. auch die Normale bestimmt werden kann (z.B. bei der Kugel zeigt der Abstandsvektor $\vec{r}$ in die gleiche Richtung wie die Normale) lässt sich dieses Verfahren auch bei Lichtberechnung durchführen.

Einige Varianten dieses Programms ergeben zum Teil überraschend überzeugende Ergebnisse. Wird z.B. der `else`-Zweig durch `discard` abgebrochen, so blickt man in den Würfel, in welchem man nur die Kugel sieht; die Grundfläche wird durch den Hintergrund dargestellt.

In unseren Beispielen werden die Objekte stets in Einheitsobjektkoordinaten definiert, d.h. der Würfel wird ohne Drehung im Ursprung gezeichnet. In diesem Fall reicht die in den Beispielen verwendete Transformation aus, um Richtungsvektoren, die in Eye-Koordinaten gegeben sind, in Objektkoordinaten zu transformieren.

8.7 Lichtberechnung mit Shadern

Nachdem die wesentlichen Konzepte der Shader-Programmierung bekannt sind, stellen wie uns als erstes die Aufgabe, bekannte Funktionen von Shadern nachzubilden, so dass mindestens die bekannten OpenGL-Fixed-Pipeline-Funktionen implementiert werden können. Die wichtigste Aufgabe für realistische Szenen ist sicherlich die Lichtberechnung, so dass wir diese zuerst behandeln.

8.7.1 Lichtparameter

Um die Parameter einer Lichtquelle im Shader zu verwenden, kann die eingebaute Uniform-Variable *gl_LightSource*[0..7] verwendet werden, welche die Parameter der entsprechenden Lichtquelle angibt. Die Namen sind in der Regel die gleichen wie die Parameter in den OpenGL-Befehlen, bzw. im GL_Sourcerer. Für die Lichtquelle 0 erhalten wir beispielsweise

```
gl_LightSource[0].ambient / ...diffuse / ...specular / ...position
/ ...spotDirection / ...spotExponent / ...spotCutOff
/ ...constantAttenuation / ...linearAttenuation / ...quadraticAttenuation
/ ...halfVector / ...spotCosCutOff
```

Die letzten beiden Parameter sind abgeleitete Größen, die für spezielle Berechnungen benötigt werden. Der *halfVector* wird beim Blinn-Phong-Lichtmodell verwendet; der *spotCosCutOff* ist der Cosinus des Winkels *spotCutOff* und wird bei der Berechnung des Spotlight benötigt.

Für das globale Umgebungslicht aus dem Lichtmodell kann die Variable

```
gl_LightModel.ambient
```

verwendet werden. Den beschienenen Objekten sind Materialeigenschaften zuzuordnen, auf welche durch die eingebaute Uniform-Variable vom Typ *gl_MaterialParameters* zugegriffen werden kann.

```
gl_FrontMaterial.emission /...ambient /...diffuse /...specular/...shininess
gl_BackMaterial.emission  /...ambient /...diffuse /...specular/...shininess
```

Zur effizienteren Berechnung stellt die Shader-Sprache einige vorher berechnete Produkte bereit.

```
gl_FrontLightModelProduct.sceneColor // ambEnv*ambMat + emissiveMat
gl_BackLightModelProduct.sceneColor
```

ergibt das Produkt aus ambienten Umgebungslicht und Materialfarbe plus dem emissivem Licht.

```
gl_FrontLightProduct[0].ambient / ...diffuse / ... specular
gl_BackLightProduct[0].ambient  / ...diffuse / ... specular
```

ergibt das Produkt aus jeweiliger Materialfarbe und Lichtfarbe.

8.7.2 Ambientes Licht

Das ambiente Licht ist i.allg. nicht von der Orientierung der Oberfläche zur Lichtquelle abhängig, und somit am einfachsten zu berechnen; es hängt aber u.U. von der Entfernung einer Lichtquelle ab. Für das globale Umgebungslicht aus dem Lichtmodell kann die Uniform-Variable

```
vec4 gl_LightModel.ambient
```

verwendet werden. Dieses Licht ist für alle Oberflächen gleich und hängt von dem ambienten Licht der Oberflächen ab. Man wird also im *GL_Sourcerer* schreiben

```
< vertexShader texture
  startProgram
    void main(void) {
      gl_Position = ftransform();
    }
>
< fragmentShader texture
  startProgram
    void main(void) {
      gl_FragColor = gl_LightModel.ambient ;
    }
>
< lightmodel
  ambient 0.4 0.4 0.2 1
>
```

Die Farbe kann in der Applikation im Lichtmodell gesetzt werden. Wird auch dem Material eine Farbe zugeordnet, so sollte die Zuweisung im Fragment-Shader lauten

```
gl_FragColor = gl_LightModel.ambient*gl_FrontMaterial.ambient;
```

bzw.

```
gl_FragColor = gl_FrontLightModelProduct.sceneColor;
```

Im letztem Fall würde noch ggf. emissives Licht, welches im Material definiert ist, aufaddiert werden. Ist eine Lichtquelle definiert, so muss deren ambientes Licht mit dem Materiallicht moduliert werden, so dass wir für die Lichtquelle 0 erhielten

```
gl_FragColor = gl_FrontLightSource[0].ambient*gl_FrontMaterial.ambient;
```

bzw. mit dem Standardprodukt

```
gl_FragColor = gl_FrontLightProduct[0].ambient;
```

Es sei daran erinnert, dass die Lichtformel aus Abschnitt 5.1.4 auf Seite 37

$$Light_{vertex} = emission_{material} + ambient_{LightModel} \cdot ambient_{material} + \\ + \sum_{i=0}^{N-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) (spotlight)_i \cdot [\, ambient_{light_i} \cdot ambient_{material} + \\ + \; max_0^{1 \odot n} \cdot diffuse_{light_i} \cdot diffuse_{material} + \; (max_0^{s \odot n})^{shininess} \cdot specular_{light_i} \cdot specular_{material} \,].$$

vorschreibt, dass die Summe aus dem globalen Umgebungslicht und den anderen Lichtkomponenten zu bilden ist; um dieses exakt nachzubilden ist die Lichtfarbe allein für den ambienten (und emittierenden) Lichtanteil also aus der Summe

```
gl_FragColor = gl_FrontLightProduct[0].ambient +
                                   gl_FrontLightModelProduct.sceneColor;
```

zu bilden, wobei ggf. noch andere Lichtquellen zu addieren sind. Es bleibt natürlich dem Programmierer überlassen zu entscheiden ob er dieses so wünscht. Allerdings müssen auch noch die anderen Lichtkomponenten nach der obigen Lichtformel berücksichtigt werden, wobei wieder zu empfehlen ist eher sparsam mit Licht umzugehen.

Damit sind die Möglichkeiten des ambienten Lichts in OpenGL mit der Shader-Sprache vollständig beschreibbar. Zusätzlich können auch Erweiterungen hierzu entwickelt werden, die nicht in der OpenGL-Standardpipeline vorgesehen sind. Beispielsweise könnte die Färbung des Lichts abhängig vom Betrachter verändert werden, z.B. entfernte Objekte – ähnlich wie bei Nebel – mit geringerer Lichtstärke oder mit entsprechender Farbverfälschung gezeichnet werden, weil entfernte Objekte im Dunst andere Färbungen haben als in der Nähe.

```
distance = length((gl_ModelViewMatrix*gl_Vertex).xyz)/10.0; // Vertex-Shader
gl_FragColor = gl_FrontLightModelProduct.sceneColor/distance; // Fragment-Sh.
```

Abhängig von der Entfernung zum Betrachter oder der Lichtquelle könnte auch die Intensität des ambienten Lichts verringert werden; diffuses Licht ist in OpenGL zwar auch entfernungsabhängig, aber in erster Linie wird bei diesem die Orientierung der Oberfläche zur Lichtquelle bestimmt, was nicht abgestellt werden kann. Ambientes Licht kann in Shadern daher einfach mit einem Dämpfungsfaktor multipliziert werden, um Entfernungsabhängigkeit zu simulieren.

```
distance = length((gl_ModelViewMatrix*gl_Vertex).xyz)/10.0; // Vertex-Shader
gl_FragColor = gl_FrontLightModelProduct.sceneColor/distance; // Fragment-Sh.
```

Eine andere Möglichkeit ambientes Licht zu variieren ist die Verwendung von zusätzlichen Färbungen, die von der Umgebung abhängen, z.B. bläuliches Licht vom Himmel oder grünliches von einer Grasfläche. Die Umgebung ist dabei durch die Orientierung der Oberflächen bestimmt, also durch die Normalen. Abhängig von diesen wird ein entsprechendes Bodenlicht oder Himmelslicht addiert bzw. moduliert.

```
float a = 0.5+0.5*dot(gl_Normal,vec3(0.0,1.0,0.0));
gl_FrontColor = mix( groundColor, skyColor ,a);
```

Die Variable **a** nimmt einen Wert an, der ungefähr der Orientierung der jeweiligen Seite entspricht. Die Färbung wird durch die Wichtung der beiden Farben des Himmels bzw. des Bodens erreicht. Natürlich soll dieses nur eine feinere Schattierung liefern und nicht die vollständige Färbung bedeuten, so dass diese Werte noch mit den Texturwerten moduliert werden müssen. Die Normale muss dabei in Weltkoordinaten angegeben werden, da sie die Orientierung der Oberfläche zur Umgebung angeben soll und nicht zum Betrachter. Diese Normalen sind üblicherweise nicht gegeben, so dass sie entweder vom Shader berechnet oder besser beim Erstellen des Modells bestimmt und dann dem Shader übergeben werden.

Weitere Möglichkeiten bestehen darin, statt zwei Farben mehrere zu verwenden, wenn beispielsweise die Beleuchtung auch von der Seite kommt, durch ein Fenster oder eine andere Lichtquelle. Hier kann man allgemeiner eine texturbasierte Umgebungsbeleuchtung verwenden, welche die Aufhellung eines Objekts von einer speziellen Umgebungstextur abhängig macht. Diese sollte als diffuses Licht nicht direkt die Umgebungstextur sein, z.B. aus einer Environment-Map, sondern nur die entsprechendes Helligkeitswerte enthalten, somit deutlich flächiger gefärbt sein. Der Fragment-Shader könnte dann etwa folgendermaßen aussehen:

```
void main(void) {
     vec3 light = vec3(gl_LightSource[0].position)-eye;
     vec3 dir = normalize(light);
     gl_FragColor = (dot(nrm,dir)  / (1.0+ 0.3*length(light))
                    + gl_Color +  textureCube(hemisphere, normal))
                   * texture2D(colorMap, st );
   }
```

Hier ist **nrm** die Normale im Eye-Space; **normal** ist die Normale im World-Space. Die Environment-Textur **hemisphere** enthält die Beleuchtung durch die Umgebung, in entsprechender Vergröberung. Weitere Informationen zur texturbasierten Umgebungsbeleuchtung findet man in der Literatur.

8.7.3 Diffuses positionelles Licht

Diffuses, positionelles Licht hängt einerseits von der Entfernung der Lichtquelle zu einem Vertex ab, andererseits aber auch von der Orientierung der Oberfläche – d.h. der Normalen des jeweiligen Vertex – zur Lichtquelle. Das einfache Lambert-Lichtmodell bestimmt aus dem Cosinus der Winkel von Normale und Richtung zur Lichtquelle die Intensität der Beleuchtung. Zusätzlich kann die Intensität durch die Entfernung mit quadratischem, linearem und konstanten Faktor gedämpft werden.

Soll dieses im Shader beschrieben werden, so müssen die beiden Dämpfungsfaktoren berechnet und deren Produkt als Dämpfung an den Fragment-Shader übergeben werden.

```
< vertexShader testDiffuse
  startProgram
    varying float attenuation;
    void main(void) {
      gl_Position = ftransform();
      vec3 eye = (gl_ModelViewMatrix * gl_Vertex).xyz;
      vec3 nrm = gl_NormalMatrix * gl_Normal;
      float cos = clamp(dot(normalize(gl_LightSource[0].position.xyz-eye)
                       ,normalize(nrm)),0.0,1.0);
      float len = length(gl_LightSource[0].position.xyz - eye);
      attenuation = cos / (
      gl_LightSource[0].constantAttenuation +
```

```
        gl_LightSource[0].linearAttenuation*len +
        gl_LightSource[0].quadraticAttenuation*len*len);
      }
  >
  < fragmentShader testDiffuse
    startProgram
      varying float attenuation;
      void main(void) {
        gl_FragColor = gl_FrontLightProduct[0].ambient
           + attenuation * gl_LightSource[0].diffuse*gl_FrontMaterial.diffuse;
      }
   >
```

Die einzige Programmzeile im Fragment-Shader kann auch durch

```
  gl_FragColor = gl_FrontLightProduct[0].ambient
               + attenuation * gl_FrontLightProduct[0].diffuse;
```

ersetzt werden. Testet man dieses aus, so erhält man offenbar das gleiche Lichtverhalten, wie man es von der Standardpipeline von OpenGL kennt. Das ist insofern unbefriedigend, als damit die Helligkeit unnatürlich von den Eckpunkten abhängt. Mit etwas mehr Aufwand lässt sich jedoch auch ein Pixel-Lighting berechnen, bei denen dieses nicht mehr der Fall ist.

Programmiertechnisch ist der Aufwand minimal, da lediglich die Berechnung von Entfernung und dem Cosinus des Winkels zwischen Normalen und Richtung zur Lichtquelle im Fragment-Shader durchgeführt werden muss, wobei die eye-Koordinaten des Vertex und die Normalen als varying-Parameter übergeben werden. Dadurch werden im Fragment-Shader die interpolierten Werte zur Berechnung der korrekten Entfernung bzw. des korrekten Cosinus des Winkels zur Lichtquelle benutzt. Das Programm könnte also folgendermaßen aussehen

```
  < vertexShader testDiffusePixelLighting
    startProgram
      varying vec3 eye, nrm;
      void main(void) {
        gl_Position = ftransform();
        eye = (gl_ModelViewMatrix * gl_Vertex).xyz;
        nrm = gl_NormalMatrix * gl_Normal;
      }
  >
  < fragmentShader testDiffusePixelLighting
    startProgram
      varying vec3 eye, nrm;
      void main(void) {
        float cos = clamp(dot(normalize(gl_LightSource[0].position.xyz-eye)
                          ,normalize(nrm)),0.0,1.0);
        float len = length(gl_LightSource[0].position.xyz - eye);
        float attenuation = cos /
                            ( gl_LightSource[0].constantAttenuation +
                              gl_LightSource[0].linearAttenuation*len +
                              gl_LightSource[0].quadraticAttenuation*len*len);
        gl_FragColor = gl_FrontLightProduct[0].ambient
                     + attenuation * gl_FrontLightProduct[0].diffuse;
      }
   >
```

Das Ergebnis ist eine wesentlich bessere Lichtverteilung über größere Flächen als mit der ersten Version, dem Vertex-Lighting. Zwar ist die Framerate messbar schlechter, aber dennoch immer hoch genug, dass diese Beleuchtungstechnik mittlerweile Stand der Technik ist.

8.7.4 Diffuses direktionelles Licht

Diffuses, direktionelles Licht hängt ausschließlich von der Orientierung der Oberfläche – d.h. der Normalen des jeweiligen Vertex – zur Einfallsrichtung des Lichts ab, die für alle Vertices gleich ist. Das einfache Lambert-Lichtmodell bestimmt aus dem Cosinus der Winkel von Normale und Richtung zur Lichtquelle die Intensität der Beleuchtung.

Soll dieses im Shader beschrieben werden, so müssen der Dämpfungsfaktor berechnet und an den Fragment-Shader übergeben werden.

```
< vertexShader directionalDiffuse
  startProgram
    varying float attenuation;
    void main(void) {
      gl_Position = ftransform();
      vec3 nrm = normalize(gl_NormalMatrix * gl_Normal);
      vec3 pos = normalize(vec3(gl_LightSource[0].position));
      attenuation = max(0.0, dot(pos,nrm));
    }
>
< fragmentShader directionalDiffuse
  startProgram
    varying float attenuation;
    void main(void) {
      gl_FragColor = gl_FrontLightProduct[0].ambient +
         attenuation * gl_FrontLightProduct[0].diffuse;
    }
 >
```

Da die Lichtverteilung in allen Punkten die gleiche ist, gibt es jetzt keine Fehler bei der Interpolation.

8.7.5 Speculares Licht

Speculares Licht hängt von der Orientierung der Oberfläche zur Einfallsrichtung des Lichts und der Blickrichtung des Betrachters ab, sowie ggf. von der Entfernung. Nach dem Blinn-Phong-Modell muss der Mittelwert aus der Summe der normalisierten Richtungsvektoren zum Betrachter und zur Lichtquelle mit der Normalen multipliziert werden.

Soll dieses im Shader beschrieben werden, so müssen die entsprechenden Vektoren berechnet, deren normalisiertes Mittel gebildet und dann das Punktprodukt mit dem Normalen-Vektor berechnet werden. Dieser Wert ist noch mit der Shininess zu potenzieren.

```
< vertexShader testSpecular
  startProgram
    varying vec4 specular;
    void main(void) {
      gl_Position = ftransform();
      vec3 nrm = normalize(gl_NormalMatrix * gl_Normal);
      vec3 eye = (gl_ModelViewMatrix * gl_Vertex).xyz;
      vec3 dir = normalize(vec3(gl_LightSource[0].position)-eye);
      vec3 hlf = normalize(dir+normalize(-eye));
      float att = max(0.0, dot(hlf,nrm));
      float power = pow(att, gl_FrontMaterial.shininess);
      specular = power*gl_FrontMaterial.specular;
    }
>
< fragmentShader testDiffuse
  startProgram
```

```
    varying vec4 specular;
    void main(void) {
      gl_FragColor = specular;
    }
>
```

Da dieses eine häufige Aufgabe ist, stellt OpenGL dafür einen abgeleiteten halfVector zur Verfügung, so dass sich der Vertex-Shader etwas vereinfacht.

```
< vertexShader testSpecular
  startProgram
    varying vec4 specular;
    void main(void) {
      gl_Position = ftransform();
      vec3 nrm = normalize(gl_NormalMatrix * gl_Normal);
      vec3 hlf = vec3(gl_LightSource[0].halfVector);
      float att = max(0.0, dot(hlf,nrm));
      float power = pow(att, gl_FrontMaterial.shininess);
      specular = power*gl_FrontMaterial.specular;
    }
>
```

Wie beim diffusen positionellen Licht ergeben sich auch hier bei größeren Abständen der Vertices unangenehme Artefakte. Um dieses zu vermeiden, müssen auch hier wieder die Berechnungen im Fragment-Shader per Pixel durchgeführt werden.

```
< vertexShader testSpecular
  startProgram
    varying vec3 nrm, eye;
    void main(void) {
      gl_Position = ftransform();
      nrm = normalize(gl_NormalMatrix * gl_Normal);
      eye = (gl_ModelViewMatrix * gl_Vertex).xyz;
    }
>
< fragmentShader testSpecular
  startProgram
    varying vec3 nrm, eye;
    void main(void) {
      vec3 dir = normalize(vec3(gl_LightSource[0].position)-eye);
      vec3 hlf = normalize(dir+normalize(-eye));
      float att = max(0.0, dot(hlf,nrm));
      float power = pow(att, gl_FrontMaterial.shininess);
      gl_FragColor = power*gl_FrontMaterial.specular;
    }
 >
```

Das Ergebnis ist wieder ein per-Pixel-lighting, welches zwar aufwändiger zu berechnen, aber qualitativ wesentlich besser ist.

Ein etwas genaueres Modell verwendet eine etwas aufwändigere Berechnung für den halfVector nach Phong, die auch etwas andere Ergebnisse liefert.

```
vec3 hlf = 2.0*dot(nrm,dir)*nrm-normalize(-eye); //Phong
```

Um diese Berechnung möglichst gut durch OpenGL zu unterstützen, wurde eine spezielle Funktion *reflect* eingeführt, mit der die Berechnung des Reflektionsvektors effizienter durchgeführt werden kann. Der Fragment-Shader muss dann folgendermaßen aussehen

```
< fragmentShader testDiffuse
  startProgram
    varying vec3 nrm, eye;
    void main(void) {
      vec3 dir = normalize(vec3(gl_LightSource[0].position)-eye);
      vec3 hlf = reflect(dir, nrm);
      float att = max(0.0, dot(hlf,normalize(eye)));
      float power = pow(att,gl_FrontMaterial.shininess);
      gl_FragColor = power*gl_FrontMaterial.specular;
    }
 >
```

Auch speculares Licht kann mit dem Entfernungsfaktor gedämpft werden, was hier nicht im Detail implementiert werden soll.

8.7.6 Spotlight

Ein Scheinwerfer (spot) ist ein positionelles Licht, welches nur in eine bestimmte Richtung mit einem vorgebbaren Winkel strahlt; positionelles Licht kann auch als Spotlight mit einem Vollkreis als Öffnungswinkel angesehen werden. Es arbeitet daher ähnlich wie das positionelle Licht mit der zusätzlich Maßgabe, dass Licht nur dann entsteht, wenn sich der Zielpunkt innerhalb eines Winkels von einer vorgegebenen Achse befindet. Der Winkel (*spotCutOff*) zwischen dieser Achse (*spotDirection*) kann durch das Punktprodukt und dem Vergleich mit dem Cosinus (der monoton fällt) einfach durchgeführt werden. Wie bei den anderen Lichtmodellen ist es auch hier für ein qualitativ hochwertiges Ergebnis nötig, die Berechnung direkt im Fragment-Shader durchzuführen, was hier direkt gemacht wird, da die anderen Ergebnisse völlig unbefriedigend sind.

```
< vertexShader testSpot
  startProgram
  varying vec3 eye, nrm, pos, sDir;
  void main(void) {
    gl_Position = ftransform();
    eye = (gl_ModelViewMatrix * gl_Vertex).xyz;
    nrm = gl_NormalMatrix * gl_Normal;
    pos = gl_LightSource[0].position.xyz;
    sDir = normalize(gl_LightSource[0].spotDirection.xyz);
  }
>
< fragmentShader testSpot
  startProgram
    varying vec3 eye, nrm, pos, sDir;
    void main(void) {
      vec3 dir = normalize(pos-eye);
      float attenuation = 0.0;
      if(dot(-dir,sDir)>gl_LightSource[0].spotCosCutoff)
        attenuation = max(0.0,dot(dir,normalize(nrm)));
      gl_FragColor = gl_FrontLightProduct[0].ambient
         + attenuation * gl_FrontLightProduct[0].diffuse;
    }
>
```

Die interpolierte Normale und Eye-Coordinate werden im Shader mit der Position der Lichtquelle zum Richtungsvektor zur Lichtquelle und zum diffusen Lichtanteil nach Lambert verrechnet. Der Lichtdämpfungsfaktor *attenuation* wird zunächst auf null gesetzt. Ist dann der Cosinus des Winkels zwischen der Richtung zur Lichtquelle *dir* und der Ausrichtung des Spotlights, die als eingebaute Uniform-Variable *gl_LightSource[0].spotDirection* übergeben wird, größer als der Cosinus des Öffnungswinkels des Spotlights, so wird die Attenuation auf den diffusen Lichtanteil nach Lambert ge-

setzt. Damit erhalten wir als Summe aus ambientem und diffusem Licht einen gleichmäßig ausgeleuchteten Lichtfleck.

Koordinaten der Lichtquelle werden in der Regel in Eye-Koordinaten angegeben, was aber nicht für die *SpotDirection* zutrifft; daher ändert sich die relative Richtung der *SpotDirection*, wenn sich die Kamera ändert. Um dieses zu vermeiden, ist die *SpotDirection* entsprechend zu rotieren, was mit der *NormalMatrix* geschehen kann, so das im Vertex-Shader geschrieben werden muss

```
sDir = normalize(gl_NormalMatrix * gl_LightSource[0].spotDirection.xyz);
```

Es können weitere Varianten hierzu implementiert werden. Zum einen kann die Lichtstärke durch geeignete entfernungsabhängige Dämpfung variiert werden; dieses ist wie in den anderen Beispielen realisierbar. Zum anderen kann die Lichtstärke radial bezogen auf die Mittelachse des Spotlights variiert werden. Der Standard verwendet hier eine Potenzfunktion, z.B.

```
attenuation = cos*pow(dot(-dir,sDir),gl_LightSource[0].spotExponent) ;
```

Eine andere Möglichkeit besteht darin, die Cosinus-Funktion auszunutzen, und so die Lichtstärke zum Rand zu dämpfen, was einen ähnlichen Effekt ergibt wie die Potenzierung.

```
attenuation = cos*max(0.0,dot(-dir,sDir)-cosCut)/ (1.0-cosCut);
```

Darüber hinaus sind natürlich weitere Varianten denkbar, z.B. dass der Lichtkegel zur Mitte hin (statt zum Rand) abnimmt

```
attenuation = cos*max(0.0,1.0 - dot(-dir,sDir))/ (1.0-cosCut);
```

oder dass der Lichtkegel sich farblich zum Rand verändert, was auch interessante Effekte ergeben kann. Dieses sind alles Erweiterungen, die mit Standard-Pipeline nicht möglich sind.

8.7.7 Kombinationen der Lichtquellen

Der OpenGL-Standard sieht vor, dass die verschiedenen Lichtquellen unabhängig voneinander berechnet werden und deren Farbwerte addiert werden. Das ergibt aufgrund des relativ kleinen Wertebereichs von Licht in OpenGL (höchstens 256 Graustufen, im Vergleich zum realen Licht von düsterem Mondlicht bis zum grellen Sonnenlicht ein winziger Bereich) häufig zu helles und damit zu blasses Licht. Bei stimmungsvollem Licht ist weniger eher mehr, so dass Lichtquellen in Maßen und mit Geschmack eingesetzt werden sollten.

Die hier vorgestellten Licht-Modelle können einfach in Methoden zusammengefasst und parametrisiert berechnet werden, je nachdem, welche und wie viele Lichtquellen verwendet werden sollen. Es ist auch möglich, Lichtquellen, die zu weit von einem Objekt entfernt sind beim Zeichnen dieses Objekts ganz wegzulassen, was Rechenzeit sparen kann. Darüber hinaus sollte man sich überlegen, ob nicht häufiger das Konzept des emissiven Lichts, welches ohne Lichtquelle ein Objekt strahlen lässt, eingesetzt werden kann, da es interessante Effekte ohne große Rechenzeitanforderung erzeugen kann.

Außerdem sollte untersucht werden, ob die einfache additive Lichtberechnung nicht zu grob ist. Da das menschliche Auge (wie die meisten Sinnesorgane) ein eher logarithmisches Auflösungsvermögen hat, ist die additive Lichtberechnung wahrscheinlich unrealistisch. Allerdings sind Forschungen zu diesem interessanten Thema noch nicht durchgeführt worden, so dass gegenwärtig Lichtwerte ausschließlich additiv berechnet werden.

Lichtberechnung ist ein schwieriges Problem der Graphik, da es ohne Licht bekanntlich gar keine Graphik gibt, aber mit zu viel Licht die Graphik häufig nicht gewünschte Wirkung hat. Hier

müssen natürlich auch künstlerische Gesichtspunkte berücksichtigt werden, die jedoch nicht Thema dieses Berichts sind.

8.8 Texturen mit Shadern

Neben der Berechnung von Lichtreflektionen auf Oberflächen werden mittels Texturen Oberflächenstrukturen beschrieben. In diesem Abschnitt werden die grundlegenden Konzepte der Anwendung von Texturen mit OpenGL-Shadern beschrieben.

Texturen sind eigentlich 'Bilder', die auf eine Oberfläche geklebt werden. Etwas drastisch könnte man die Oberflächen in OpenGL-Programmen als Tapeten bezeichnen, die statt der Wände zwischen den Vertices aufgespannt werden; werden nur einfarbige Tapeten verwendet, so handelt es sich um das einfache Farbmodell von OpenGL.

8.8.1 Einfache Texturen

Wird nur eine Textur verwendet, so kann diese geladen werden und dann können abhängig von den Textur-Koordinaten die einzelnen Texel-Werte ausgelesen werden. Ein sehr einfaches Beispielprogramm dazu könnte folgendermaßen aussehen.

```
< vertexShader texture
  startProgram
    void main(void) {
      gl_Position = ftransform();
      gl_TexCoord[0] = gl_MultiTexCoord0;
    }
>
< fragmentShader texture
  texture0 colorMap
  startProgram
    uniform sampler2D colorMap;
    void main (void) {
      vec4 textureColor = texture2D( colorMap, gl_TexCoord[0].st);
      gl_FragColor = textureColor;
    }
>
```

Texturen werden in OpenGL-Shadern durch die Deklaration

```
uniform sampler2D colorMap;
```

eingebunden. Das Schlüsselwort *sampler2D* kennzeichnet die Art der zu ladenden Daten, hier zweidimensionale Texturen. Analog sind ein- oder dreidimensionale Texturen sowie Cube-Texturen und Tiefentexturen zur Schattenberechnung möglich:

```
sampler1D,  sampler2D,  sampler3D,  textureCube, shadow1D, shadow2D.
```

Varianten zu diesen Typangaben erlauben die automatische Berechnung der Division durch die q-Komponente (Proj) sowie eine Level-of-Detail-Selektion (Lod).

Um Textur-Koordinaten zu berechnen, werden den Vertices die Textur-Koordinaten zugeordnet, die in den Vertex-Shadern durch *gl_MultiTexCoord0* zugegriffen werden können. Dieses sind Vektoren der Länge 4, wobei die letzte Zahl durch Werte von 0 bis zur maximalen Anzahl von Textur-Einheiten minus 1 (*gl_MaxTextureUnits*) ersetzt werden kann.

Eine eingebaute *varying*-Variable *gl_TexCoord*[..] erlaubt es, diese als interpolierte Werte an die Fragment-Shader zu übergeben; sie können aber auch über ein normales vierdimensionales *varying*-

Feld übergeben werden, bzw. – da nur zwei Koordinaten benötigt werden, auf die entsprechenden beiden Koordinaten beschränkt werden:

```
< vertexShader texture
  startProgram
    varying vec2 st;
    void main(void) {
      gl_Position = ftransform();
      st = gl_MultiTexCoord0.st;
    }
>
< fragmentShader texture
  texture0 colorMap
  startProgram
    varying vec2 st;
    uniform sampler2D colorMap;
    void main (void) {
      vec4 textureColor = texture2D(colorMap, st);
      gl_FragColor = textureColor;
    }
>
```

Natürlich können die Textur-Koordinaten im Fragment-Shader neu berechnet werden; z.B. ließe sich im letzten Programm

```
vec4 textureColor = texture2D(colorMap, st/2.0);
```

schreiben, was die Textur in beide Richtungen um den Faktor 2 dehnt. Soll die Dehnung oder Stauchung in den verschiedenen Richtungen unterschiedlich sein, so lässt sich dieses beispielsweise durch den Befehl

```
vec4 textureColor = texture2D(colorMap, st*vec2(0.5, 2.0) );
```

realisieren. Natürlich lässt sich auch eine Verschiebung der Textur-Koordinaten durch

```
vec4 textureColor = texture2D(colorMap, st*vec2(0.5, 2.0)+vec2( 0.2, 0.5) );
```

erreichen. Analog können die Koordinaten auch sonst beliebig manipuliert werden, z.B. vertauscht werden

```
vec4 textureColor = texture2D(colorMap, st.ts*vec2(0.5,2.0)+vec2(0.2,0.5) );
```

oder es könnte eine Koordinate konstant gehalten werden.

```
vec4 textureColor = texture2D(colorMap, st.ts*vec2(0.5,0.0)+vec2(0.2,0.5) );
```

Als weitere Variation lässt sich z.B. jeder Pixel aus mehreren verschiedenen Texeln berechnen.

```
vec4 textureColor = vec4(0.0);
for(float i=-1.0;i<=1.0;i++)
for(float j=-1.0;j<=1.0;j++)
textureColor = textureColor +
texture2D(colorMap, st*vec2(1.0+i*0.01,-1.0-j*0.01) );
gl_FragColor = textureColor / 9.0;
```

Schließlich sei noch erwähnt, dass der Fragment-Shader auch unterbrochen werden kann; es wird dann kein Pixel gezeichnet, die Fläche wird also 'löcherig'. Mit dem Befehl

```
if(textureColor.r <= 0.6) discard;
```

werden alle die Pixel nicht gesetzt, bei denen die zu schreibende rote Komponente kleiner oder gleich 0.6 ist. Statt die Pixel nicht zusetzen, kann auch eine andere Farbe, oder z.B. ein anderes Beleuchtungsmodell verwendet werden, so dass unterschiedliche Flächen mit unterschiedlichem Reflexionsverhalten nachgebildet werden können.

Wie man sieht, lässt sich auf die unterschiedlichste Weise auf Texturen zugreifen. Die Flexibilität ist beliebig groß, ohne dass die Effizienz unnötig eingeschränkt wird.

8.8.2 Übergabe von Texturen an Shader

Texturen werden wie bei der Standard-Pipeline üblich an die Textur-Einheiten gebunden; die Shader erhalten die Texturen dann durch die Verknüpfung mit der jeweiligen Textureinheit. Textureinheiten werden als Uniform-Variablen von der Anwendung an die Shader gegeben. Sie werden für den Sampler *uniform sampler2D colorMap* durch den Befehl

```
int uniformLoc = gl.glGetUniformLocationARB(myProgram, "colorMap");
if (uniformLoc != -1) gl.glUniform1iARB(uniformLoc, textureUnit);
```

an den Shader gebunden. Dabei ist *textureUnit* die Nummer der Textureinheit (d.h. die Zahlen 0, 1, ...), an welche die jeweilige Textur gebunden wird. Da mehrere Textureinheiten vorhanden sein müssen (mindestens 2, meistens 8 oder mehr), können auch mehrere Texturen gleichzeitig an einen OpenGL-Shader übergeben werden.

Texturen können von Vertex- und Fragment-Shadern gelesen werden, wenngleich nur Fragment-Shader Mipmapping erlauben, da die level of detail-Berechnung (lod) nur für varying-Variablen durchgeführt wird.

8.8.3 Berechnung von Texturen

Statt die Texturen aus Bildern zu entnehmen, können die Farbwerte auch berechnet werden, wozu allerdings im Shader die Position des Pixels bekannt sein muss. Diese lässt sich einfach über die Texturkoordinaten bestimmen, die also auch dann übergeben werden müssen, wenn gar keine Textur gebunden wird. Aus den interpolierten Texturkoordinaten können dann die jeweiligen Werte berechnet werden.

```
vec2 c = Dichte * gl_TexCoord[0].st;
vec2 p = fract(c) - vec2(0.5);
gl_FragColor = KaroFarbe;
if( abs(p.s)+abs(p.t) >= KaraGroesse ) gl_FragColor = OberflaechenFarbe;
```

Die Farbe des Karos wird durch die **`Oberflächenfarbe`** überschrieben, wenn der Abstand der Summe der s- und t-Koordinaten von der Mitte größer ist als die vorgegebene **`KaraGroesse`** ist. Die **`Dichte`** gibt an, wie viele Karos auf jede Einheit von der Texturkoordinaten gezeichnet werden.

Neben festen geometrischen Mustern lassen sich auch komplexere Muster erzeugen, z.B. durch trigonometrische Funktionen, oder durch Zufallszahlen. Die Shader-Sprache stellt auch sogenannte Noise-Funktionen zur Verfügung, die Speudozufallswerte erzeugen und ebenfalls für verschiedene Muster eingesetzt werden können.

Weitere Techniken können ebenfalls mit Shadern direkt implementiert werden, z.B. Bump-Mapping, welches auch mit der Fixed-Pipeline realisiert werden kann, oder Parallas-Mapping, welches nur noch mit Shadern realisiert werden kann. Dieses wird unten beschrieben.

8.8.4 Mehrere Texturen

Wie bei der Standard-Pipeline können auch mit Shadern mehrere Texturen gleichzeitig verarbeitet werden, in der Regel deutlich mehr als mit der Fixed-Pipeline, z.B. 8 oder 16. GLSL lässt sich dieses durch Konstante wie *GL.GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS* abfragen. Die Verarbeitung der Texturen besteht dann einfach darin, dass die jeweiligen Texturwerte im Fragment-Shader miteinander verknüpft werden. Besonders interessant sind spezielle Anwendungen wie Bump-Mapping oder Schattenberechnung.

8.9 Spezielle Texturen

Wie bei der Standard-Pipeline können auch mit Shadern mehrere Texturen gleichzeitig verarbeitet werden, in der Regel deutlich mehr als mit der Fixed-Pipeline, z.B. 8 oder 16. GLSL lässt sich dieses durch Konstante wie *GL.GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS* abfragen. Die Verarbeitung der Texturen besteht dann einfach darin, dass die jeweiligen Texturwerte im Fragment-Shader miteinander verknüpft werden. Besonders interessant sind spezielle Anwendungen wie Bump-Mapping oder Schattenberechnung.

8.9.1 Bump-Mapping mit Shadern

Diese Technik legt die Information über die Oberflächenstruktur in einer eigenen Textur ab. Da die Oberflächenstruktur durch ihre Senkrechten, also durch die Normalen, angegeben wird, spricht man auch von einer Normal-Map, bzw. nennt die Technik auch Normal-Mapping. Wir werden die Textur hier als Normal-Map bezeichnen, aber weiter von Bump-Mapping sprechen (bump: engl. Beule).

Die Bedeutung der Werte in der Normal-Map sind die Normalen je Texel, d.h. für jeden Farbwert wird eine eigene Normale definiert. Wenn dann das Licht abhängig von den Normalen berechnet wird, wie beim diffusen und specularen Licht, erscheint ein Punkt an der Oberfläche heller oder dunkler, je nachdem ob die Oberfläche zur Lichtquelle hin orientiert ist oder von dieser weg. Dadurch entsteht durch einen

Als Beispiel für einen entsprechenden Shader verwenden wir den folgenden

```
< vertexShader bumpMapping
  startProgram
  varying vec3 eye;
  varying vec3 pos;
  void main(void) {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;   /**/
    eye = (gl_ModelViewMatrix * gl_Vertex).xyz;
    pos = gl_LightSource[0].position.xyz;
    gl_TexCoord[0] = gl_MultiTexCoord0;
  }
>
< fragmentShader bumpMapping
  texture0 RenderTexture
  texture1 NormalTexture
  uniform alpha
  uniform mouse
  startProgram
  uniform sampler2D RenderTexture,NormalTexture;
  varying vec3 eye;
  varying vec3 pos;
  uniform float alpha ;
  uniform vec2 mouse;
  void main(void) { // here starts the main program
    vec3 normalMap = vec3(texture2D(NormalTexture, gl_TexCoord[0].st));
//  Expand the normalMap into a normalized signed vector
```

```
    normalMap = normalMap * 2.0 - 1.0;
    normalMap = normalize(gl_NormalMatrix * normalMap);
//  Get the color of the texture
    vec4 DecalCol = vec4(texture2D(RenderTexture,gl_TexCoord[0].st).rgb,1.0);
//  Find the dot product between light direction and normal
    vec3 dir = normalize(pos.xyz - eye.xyz);
    float len = length(pos.xyz - eye.xyz); // distance to lightsource
    float NdotL = max(dot( normalMap, dir), 0.0);
    vec3 eyeN = normalize(eye.xyz);
    vec4 amb  = gl_LightSource[0].ambient * gl_FrontMaterial.ambient;
    vec4 dfu  = gl_LightSource[0].specular * gl_FrontMaterial.diffuse;
    vec4 spec = gl_LightSource[0].specular * gl_FrontMaterial.specular;
    float specularIntensity =
      pow(max(0.0, dot(reflect(dir, normalMap), eyeN)),
          gl_FrontMaterial.shininess );
    float att = 1.0/(1.0+0.1*len);
    gl_FragColor = amb*DecalCol
           + att*(NdotL*dfu*DecalCol* mouse.y
                 + spec*specularIntensity*DecalCol*mouse.x)
                 *vec4(1.0, 1.0, len*0.4, 1.0);
  }
>
```

Diese Implementierung enthält einige Besonderheiten, auf die wir später eingehen. Zunächst werden die üblichen Werte berechnet. *normalMap* enthält die Normale in kodierter Form, welche entsprechend umgerechnet werden muss, um den Vektor zu erhalten, der Senkrecht zur Oberfläche zeigt. Die Transformation dieses Vektors mit der *NormalMatrix* wird gemacht, da sämtliche Flächen im Ursprung in der x-y-Ebene gezeichnet wurden und durch entsprechende Transformationen in die jeweiligen Weltkoordinaten gebracht wurden. Die nachfolgende View-Transformation bringt diese Flächen dann in die Eye-Koordinaten. Damit enthält die *NormalMatrix* die Transformation der Normalen in die Eye-Koordinaten. Mit der gleichen Transformation lassen sich dann die Oberflächen-Normalen in die Eye-Koordinaten transformieren.

Als nächstes werden die üblichen Berechnungen für ambientes, diffuses und spekulares Licht durchgeführt, wobei jetzt die Bump-Map-Normale verwendet wird, da wir ja die Reflexion des Lichts an diesen Oberflächenstrukturen orientieren wollen, d.h. die Berechnung erfolgt Pixel-genau, mit der Ausnahme, dass wir nicht die Vertex-Normalen verwenden, sondern die Bump-Map-Normalen. Die in der letzten Zeile einfügte Farbänderung sorgt für eine entfernungsabhängige Farbänderung, die für die Bump-Funktionalität nicht notwendig wäre.