Technische Universität Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnik

Dresden, August 2003

Lehrheft

Computergrafik

Grafikprogrammierung mit OpenGL

Herausgabe: K. Hoedt
W. Mascolus

Grafikprogrammierung mit OpenGL

VORWORT

Der hier vorliegende Lehrbrief soll den Studenten der Informatik der TU Dresden und natürlich auch anderer Institutionen einen kurzen Einblick in das Grafiksystem OpenGL bieten. Dieses Schriftstück stellt nicht den Anspruch auf Vollständigkeit will aber die wesentlichen Aspekte der OpenGL Programmierung behandelt wissen. Da die Autoren sich selbst noch im Studium der Informatik befinden, soll hier nicht der Anspruch auf 100%ige Richtigkeit gestellt werden. Für Anfragen zu Problemen stehen die Autoren unter den angeführten e-mail-Adressen zur Verfügung. Im folgenden sollen die Grundaspekte der OpenGL aber auch vertiefende Programmiertechniken behandelt werden. Die Autoren sind davon überzeugt, daß OpenGL in naher Zukunft eine dominierende Rolle in der 3D-Programmierung spielen wird. Indizien hierfür sind u.a. die Implementierung von OpenGL Treibern von verschiedensten Firmen wie z.B. 3DFX. Ein weiterer wichtiger Punkt für die Argumentation hin zu OpenGL ist die Plattformunabhängigkeit und die (relative) Hardwareunabhängigkeit von vorhandener PC-Hardware. Nicht zuletzt die Tatsache, daß id-Software sich bei der Implementierung des Kult-Computerspiels **Quake** für OpenGL entschieden hat, sollte ein deutliches Anzeichen für die Leistungsfähigkeit von OpenGL sein.

Genug der Vorrede, die Autoren wünschen viel Spaß bei der Lektüre des vorliegenden Lehrbriefes.

Stefan Rippert schelm@inf.tu-dresden.de

Tobias Pietzsch tp8@inf.tu-dresden.de

1 HISTORISCHES

Die U.S. amerikanische Firma **Silicon Graphics** (SGI) entwickelte in den 80er Jahren Workstations, die mit spezieller skalierbarer Grafikhardware ausgestattet waren. Diese Spezialhardware übernimmt je nach Ausbaustufe die Grafikdarstellung und entlastet damit den Prozessor. In der Mitte der 90er Jahre kam dann auch eine solche Hardware im PC-Bereich auf den Markt. Führend in diesem Segment ist die Firma 3DFX.
Zur Programmierung der SGI Workstations wurde die IRIS GL entwickelt (GL steht hier für Graphics Library). OpenGL wird weitgehend als der Nachfolger der IRIS GL betrachtet, obwohl das so nicht ganz korrekt ist. OpenGL wurde nämlich von einem Konsortium, dem **ARB** (Architectural Review Board) entwickelt, stützt sich aber in seiner Funktionalität und auf die IRIS GL. Dem ARB gehören zur Zeit IBM, DEC, Intel, SGI (und Microsoft) an. Die Gruppe trifft sich vierteljährlich. Auf diesen Treffen werden Erweiterungen der Spezifikationen diskutiert, eingegangene Anwenderreports bearbeitet und die manual pages verbessert. Nicht zuletzt durch die Integration von OpenGL in das Betriebssytem Windows 9x stehen die Zeichen recht gut dafür, daß OpenGL ein Standard für 3D - Programmierung im PC Bereich werden könnte.

2 DAS MINIMALE OPENGL - PROGRAMM

Ziel dieses Kapitels ist in Kurzform zu zeigen, welche Schritte unter Windows 9x/NT nötig sind, um OpenGL benutzen zu können. Darstellungen finden in OpenGL immer in einem *rendering context* statt. Es ist also nötig, eine Verbindung zwischen einem Fenster (*device context*) von Windows und einem OpenGL *rendering context **(RC)*** herzustellen, damit OpenGL Anweisungen irgendwelche sichtbaren Resultate hervorrufen.

Im folgenden wird anhand eines Beispieles Schritt für Schritt erklärt, wie man diese Anbindung unter **VisualC++ 4.2** bzw. **VisualC++ 5.0** und **MFC** realisiert.

2.1 Projekt zu VisualC++ 4.2

2.1.1 Neues Projekt erstellen

Zunächst muß ein neuer **VisualC** Workspace erstellt werden:

- **New** aus dem Menü **File** wählen.
- **Project Workspace** wählen. **OK** klicken.
- Im **New Project Workspace** Dialog **AppWizard(exe)** wählen, als Projektnamen ***MinGL*** eingeben und **Create** klicken.
- Auf der nächsten Seite **Single Document** auswählen und **Next** klicken.
- Auf der nächsten Seite **Next** klicken.
- Auf der nächsten Seite **Next** klicken.
- Auf der nächsten Seite alles deaktivieren außer **3D controls**. **Finish** klicken.

Der AppWizard erstellt jetzt das Gerüst für eine MFC Anwendung, das nun nur noch mit der gewünschten Funktionalität versehen werden muß.

2.1.2 opengl32.dll verfügbar machen

Im **FileView** das Headerfile *MinGLView.h* öffnen und als erste Zeile einfügen:

```
#include <gl\gl.h>
```

Settings... im Menü **Build** wählen.
Im **Link**-Tab bei **Object/library modules:** *opengl32.lib* hinzufügen.

2.1.3 rendering context zur Verfügung stellen

Die *CMinGLView* Klasse, die für das Zeichnen ins Fenster verantwortlich ist, braucht erst mal eine Variable, um das Handle des *rendering context* speichern zu können. Dazu im **ClassView** auf *CMinGLView* rechtsklicken und im Kontextmenü **Add Variable** wählen und eine protected Variable *m_hRC* vom Typ *HGLRC* hinzufügen.

Die Anbindung des ***RC*** ans Fenster erfolgt am zweckmäßigsten in der *OnCreate()* Methode:

- **Strg + w** drücken. Der **MFC ClassWizard** wird angezeigt.
- Bei **Class name** CMinGLView wählen.
- Bei **Messages** *WM_CREATE* wählen. **Add Function** klicken. Damit wurde *CMinGLView::OnCreate()* erstellt.
- **OK** klicken.

CMinGLView::OnCreate() wurde jetzt erstellt und wird wie folgt erweitert:

```
int CMinGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
    return -1;

    // TODO: Add your specialized creation code here
    PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),     // Structure size.
      1,                                 // Structure version number.
      PFD_DRAW_TO_WINDOW |               // Property flags.
      PFD_SUPPORT_OPENGL,
      PFD_TYPE_RGBA,
      24,                                // 24-bit color.
      0, 0, 0, 0, 0, 0,                  // Not concerned with these.
      0, 0, 0, 0, 0, 0, 0,               // No alpha or accum buffer.
      32,                                // 32-bit depth buffer.
      0, 0,                              // No stencil or aux buffer.
      PFD_MAIN_PLANE,                    // Main layer type.
      0,                                 // Reserved.
      0, 0, 0                            // Unsupported.
    };

    CClientDC clientDC(this);

    int pixelFormat = ChoosePixelFormat(clientDC.m_hDC, &pfd);
    BOOL success = SetPixelFormat(clientDC.m_hDC, pixelFormat, &pfd);
    m_hRC = wglCreateContext(clientDC.m_hDC);

    return 0;
}
```

Die Funktion *wglCreateContext()* erzeugt einen OpenGL **R**endering **C**ontext, passend zum DC des Fensters. Der RC übernimmt das Pixelformat des DC, welches deshalb logischerweise vor dem Aufruf von *wglCreateContext()* gesetzt wird. Ein Pixelformat spezifiziert verschiedene Eigenschaften der OpenGL-Darstellung, zum Beispiel:

- ob der Pixelbuffer single- oder double-buffered ist.
- ob die Pixeldaten in RGBA oder color-index form sind.
- die Anzahl der Bits für die Farbinformation.
- die Anzahl der Bits für den Z-Buffer.

Genaue Erläuterungen zur *PIXELFORMATDESCRIPTOR* - Struktur und zu den Funktionen *ChoosePixelFormat* und *SetPixelFormat* sind in der Visual C++ 5.0 Hilfe zu finden.
Der RC sollte bei Beendigung des Programms gelöscht werden. Dazu fügt man in *CMinGLView* noch einen Handler für die Windowsmessage ***WM_DESTROY*** ein (mit dem Class Wizard. Vorgehensweise analog zu *OnCreate*). In *CMinGLView::OnDestroy()* wird folgendes eingefügt:

```
void CMinGLView::OnDestroy()
{
      CView::OnDestroy();

      // TODO: Add your message handler code here
      wglDeleteContext(m_hRC);
}
```

2.1.4 rendering context aktivieren

Bevor OpenGL-Darstellungen ausgeführt werden können, muß jetzt nur noch der RC zum aktuellen Kontext gemacht werden. Die Funktion *wglMakeCurrent()* bindet den RC an einen DC des Fensters (das muß nicht derselbe sein, mit dem der RC kreiert wurde, nur das PixelFormat muß übereinstimmen). Wird als RC *NULL* angegeben, so wird der aktuelle Kontext "entbunden". Der Wert von DC spielt dann keine Rolle. Die Methode *CMinGLView::OnDraw()* wird also erweitert:

```
void CMinGLView::OnDraw(CDC* pDC)
{
CMinGLDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);

// TODO: add draw code for native data here
wglMakeCurrent(pDC->m_hDC, m_hRC);

// diese Methode führt die GL-Aufrufe aus
DrawWithGL();
      wglMakeCurrent(pDC->m_hDC, NULL);
}
```

2.1.5 OpenGL-Aufrufe zum Zeichnen

Jetzt kann endlich gezeichnet werden. Der Übersicht halber werden die GL-Aufrufe in einer separaten Methode untergebracht, nämlich *CMinGLView::DrawWithGL()*.
Mit **Add Function** aus dem Kontextmenü zu *CMinGLView* (rechts auf *CMinGLView* klicken) fügt man diese Methode ein (typ: *void*).

Beispiel für *DrawWithGL()*:

```
void CMinGLView::DrawWithGL()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0f, 1.0f, 1.0f);

    glBegin(GL_LINES);
        glVertex2f (0.25f, 0.25f);
        glVertex2f (0.75f, 0.25f);
    glEnd();

    glFlush();
}
```

Kurze Erklärung:
glClearColor() setzt die Löschfarbe (RGBA).
glClear() löscht ein Buffer, in diesem Fall den Bildspeicher.
glColor3f() setzt die Zeichenfarbe (RGB).
*glBegin(**GL_LINES**)...glEnd()* teilt OpenGL mit, daß die dazwischen stehenden Punkte (*glVertex2f()*) als Linien gezeichnet werden sollen.
glFlush() bewirkt die Ausführung der vorangegangenen Befehle "in endlicher Zeit".

2.1.6 Und fertig !

Aus dem **Build** Menü **Rebuild All** wählen und das war's im Prinzip schon. Mit **Execute MinGL.exe** aus dem **Build** Menü kann das Programm gestartet werden. Zum

Herumexperimentieren mit den OpenGL-Anweisungen der nächsten Kapitel genügt es zunächst, nur an der Methode *DrawWithGL()* Änderungen vorzunehmen.

2.2 Projekt zu VisualC++ 5.0

2.2.1 Neues Projekt erstellen

Zunächst muß ein neuer **VisualC** Workspace erstellt werden:
New aus dem Menü **File** wählen.
Im **New** Dialog **AppWizard(exe)** wählen, einen Projektnamen (z.B. *MinGL*) eingeben und **OK** klicken.
Auf der nächsten Seite **Single Document** auswählen und **Next** klicken.
Auf der nächsten Seite **Next** klicken.
Auf der nächsten Seite **ActiveX controls** deaktivieren und **Next** klicken.
Auf der nächsten Seite alles deaktivieren außer **3D controls**. **Finish** klicken.
Der AppWizard bastelt jetzt das Gerüst für eine MFC Anwendung, das nun nur noch mit der gewünschten Funktionalität versehen werden muß.

2.2.2 opengl32.dll verfügbar machen

Im **FileView** das Headerfile *MinGLView.h* öffnen und als erste Zeile einfügen:

```
#include <gl\gl.h>
```

Settings... im Menü **Project** wählen.
Im **Link**-Tab bei **Object/library modules:** *opengl32.lib* hinzufügen.

2.2.3 rendering context zur Verfügung stellen

Die *CMinGLView* Klasse, die für das Zeichnen ins Fenster verantwortlich ist, braucht erst mal eine Variable, um das Handle des *rendering context* speichern zu können. Dazu im **ClassView** auf *CMinGLView* rechtsklicken und im Kontextmenü **Add Member Variable** wählen und eine protected Variable *m_hRC* vom Typ *HGLRC* hinzufügen.
Die Anbindung des ***RC*** ans Fenster erfolgt am zweckmäßigsten in der *OnCreate()* Methode. Dazu wählt man im Kontextmenü zu *CMinGLView* **Add Windows Message Handler**, und fügt einen Handler für die message *WM_CREATE* hinzu. *CMinGLView::OnCreate()* wurde jetzt erstellt und wird wie folgt erweitert:

```
int CMinGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
    return -1;

    // TODO: Add your specialized creation code here
    PIXELFORMATDESCRIPTOR pfd =
    {
      sizeof(PIXELFORMATDESCRIPTOR),      // Structure size.
      1,                                  // Structure version number.
      PFD_DRAW_TO_WINDOW |                // Property flags.
      PFD_SUPPORT_OPENGL,
      PFD_TYPE_RGBA,
      24,                                 // 24-bit color.
      0, 0, 0, 0, 0, 0,                   // Not concerned with these.
      0, 0, 0, 0, 0, 0, 0,                // No alpha or accum buffer.
      32,                                 // 32-bit depth buffer.
```

```
      0, 0,                              // No stencil or aux buffer.
      PFD_MAIN_PLANE,                    // Main layer type.
      0,                                 // Reserved.
      0, 0, 0                            // Unsupported.
   };

   CClientDC clientDC(this);

   int pixelFormat = ChoosePixelFormat(clientDC.m_hDC, &pfd);
   BOOL success = SetPixelFormat(clientDC.m_hDC, pixelFormat, &pfd);
   m_hRC = wglCreateContext(clientDC.m_hDC);

   return 0;
}
```

Die Funktion *wglCreateContext()* erzeugt einen OpenGL **R**endering **C**ontext, passend zum DC des Fensters. Der RC übernimmt das Pixelformat des DC, welches deshalb logischerweise vor dem Aufruf von *wglCreateContext()* gesetzt wird. Ein Pixelformat spezifiziert verschiedene Eigenschaften der OpenGL-Darstellung, zum Beispiel:

- ob der Pixelbuffer single- oder double-buffered ist.
- ob die Pixeldaten in RGBA oder color-index form sind.
- die Anzahl der Bits für die Farbinformation.
- die Anzahl der Bits für den Z-Buffer.

Genaue Erläuterungen zur *PIXELFORMATDESCRIPTOR* - Struktur und zu den Funktionen *ChoosePixelFormat* und *SetPixelFormat* sind in der Visual C++ 5.0 Hilfe zu finden.
Der RC sollte bei Beendigung des Programms gelöscht werden. Dazu fügt man in *CMinGLView* noch einen Handler für die Windowsmessage ***WM_DESTROY*** ein. In *CMinGLView::OnDestroy()* wird folgendes eingefügt:

```
void CMinGLView::OnDestroy()
{
      CView::OnDestroy();

      // TODO: Add your message handler code here
      wglDeleteContext(m_hRC);
}
```

2.2.4 rendering context aktivieren

Bevor OpenGL-Darstellungen ausgeführt werden können, muß jetzt nur noch der RC zum aktuellen Kontext gemacht werden. Die Funktion *wglMakeCurrent()* bindet den RC an einen DC des Fensters (das muß nicht derselbe sein, mit dem der RC kreiert wurde, nur das PixelFormat muß übereinstimmen). Wird als RC *NULL* angegeben, so wird der aktuelle Kontext "entbunden". Der Wert von DC spielt dann keine Rolle. Die Methode *CMinGLView::OnDraw()* wird also erweitert:

```
void CMinGLView::OnDraw(CDC* pDC)
{
CMinGLDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);

// TODO: add draw code for native data here
wglMakeCurrent(pDC->m_hDC, m_hRC);

// diese Methode führt die GL-Aufrufe aus
DrawWithGL();
```

```
      wglMakeCurrent(pDC->m_hDC, NULL);
}
```

2.2.5 OpenGL-Aufrufe zum Zeichnen

Jetzt kann endlich gezeichnet werden. Der Übersicht halber werden die GL-Aufrufe in einer separaten Methode untergebracht, nämlich *CMinGLView::DrawWithGL()*.
Mit **Add Member Function** aus dem Kontextmenü zu *CMinGLView* (rechts auf *CMinGLView* klicken) fügt man diese Methode ein (typ: *void*).

Beispiel für *DrawWithGL()*:

```
void CMinGLView::DrawWithGL()
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0f, 1.0f, 1.0f);

    glBegin(GL_LINES);
        glVertex2f (0.25f, 0.25f);
        glVertex2f (0.75f, 0.25f);
    glEnd();

    glFlush();
}
```

Kurze Erklärung:
glClearColor() setzt die Löschfarbe (RGBA).
glClear() löscht ein Buffer, in diesem Fall den Bildspeicher.
glColor3f() setzt die Zeichenfarbe (RGB).
*glBegin(**GL_LINES**)...glEnd()* teilt OpenGL mit, daß die dazwischen stehenden Punkte (*glVertex2f()*) als Linien gezeichnet werden sollen.
glFlush() bewirkt die Ausführung der vorangegangenen Befehle "in endlicher Zeit".

2.2.6 Und fertig !

Build All anklicken und das war's im Prinzip schon. Zum Herumexperimentieren mit den OpenGL-Anweisungen der nächsten Kapitel genügt es zunächst, nur an der Methode *DrawWithGL()* Änderungen vorzunehmen.

3 DIE NOMENKLATUR VON OPENGL

OpenGL zeichnet sich wie andere Bibliotheken durch eine strenge Namensgebung für die Funktionen, Datentypen, Konstanten und Variablen aus. Weiterhin werden Befehle durch eine bestimmte Nomenklatur zu Befehlsgruppen zusammengefaßt. In dem jetzt folgenden Teil werden diese Vereinbarungen am Beispiel der C-Schnittstelle vorgestellt.

3.1 Konstanten

Die Konstanten werden in OpenGL durchweg groß geschrieben. Sie beginnen immer „**GL_**“ und zum Abtrennen einzelner Wortteile dient der „**_**“- Unterstrich. Bei den Konstanten muß zwischen verschiedenen Konstanten-Typen unterschieden werden. So sind z.B. die Konstanten ***GL_TRUE*** und ***GL_FALSE*** vom Typ *GLBoolean*, ***GL_FRONT*** und ***GL_BACK*** vom Typ *GLEnum*.

3.2 Datentypen

Streng werden die Regeln für die Nomenklatur auch bei den Datentypen der OpenGL eingehalten. Da für OpenGL schon jetzt bei weitem nicht nur die C-Schnittstelle zur Verfügung steht, ist eine strenge Nomenklatur im Sinne der Portabilität durchaus sinnvoll. Durch diese strengen Regeln, die z.B. auch jede Prozessorplattform unterstützen muß, wurden natürlich auch Standards für die Rechengenauigkeit gesetzt. So sind die Farbwerte in OpenGL intern immer Fließkommazahlen.

In der jetzt folgenden Tabelle werden die Datentypen, ihre Entsprechungen in Ansi C, Borland Pascal, OpenGL und die für sie typischen OpenGL-Kürzel aufgezeigt:

Datentyp	***ANSI C***	***Borland Pascal***	***OpenGL***	***OpenGL-Kürzel***
8 Bit Integer mit Vz.	*signed char*	*Shortint*	*GLbyte*	*b*
16 Bit Integer mit Vz.	*short*	*Integer*	*GLshort*	*s*
32 Bit Integer mit Vz.	*int*	*Longint*	*GLint* *GLsizei*	*i*
8 Bit Integer ohne Vz.	*unsigned char*	*Byte*	*GLubyte* *GLboolean*	*ub*
16 Bit Integer ohne Vz.	*unsigned short*	*Word*	*GLushort*	*us*
32 Bit Integer ohne Vz.	*unsigned int*	-	*GLuint* *GLenum* *GLbitfield*	*ui*
32 Bit Fließkomma	*float*	*Single*	*GLfloat* *GLclampf*	*f*
32 Bit Integer mit Vz.	*double*	*Double*	*GLdouble* *GLclampd*	*d*

Tabelle: die Datentypen der OpenGL

Es fällt auf, daß in OpenGL mit „**GL**“ beginnen und klein geschrieben fortgesetzt. Auch das ist ein Teil der Nomenklatur derselben. Der Autor ist der Meinung, daß die Datentypen keiner näheren Erläuterung bedürfen, mit der Ausnahme der Typen *GLclampf* und *GLclampd*. Das englische Wort „clamp“ ist gemeinsamer Bestandteil dieser beiden Typen. „clamp“ bedeutet hier, daß diese Datentypen in ihrem Wertebereich beschränkt sind. Meistens bewegen sich die gültigen Werte für diese Typen zwischen 0.0 und 1.0. Werte, die größer als 1.0 sind werden auf 1.0 und Werte, die kleiner als 0.0 auf 0.0 gemappt.

3.3 Die verschiedenen Varianten von GL-Befehlen

Genau wie bei den Konstanten und Datentypen beginnen in OpenGL auch alle Prozedurnamen mit dem Kürzel „gl“, diesmal aber klein geschrieben. Um diese ursprünglichen OpenGL Befehle von denen anderer Bibliotheken unterscheiden zu können, beginnen diese mit anderen Kürzeln. Prozeduren der GLU beginnen mit „glu“, die der GLX mit „glx“ und die der WGL eben mit „wgl“.

Viele OpenGL Aufrufe können in verschiedenen Varianten aufgerufen werden. So können einerseits die Eingabeparameter differieren, zum anderen können die Parameter in vektorieller oder nicht vektorieller Form übergeben werden. Im allgemeinen gilt die folgende Darstellung für die OpenGL Kommandos:

Rückgabetyp glSinnvollerName{*b s i f d ub us ui*}{*v*} (*GLTyp bla, ...*). Das bedeutet nun, daß das erste Anhängsel im Prozedurnamen den Typ der Übergabeparameter kennzeichnet und das „*v*“, ob die Parameter in Vektorform übergeben werden oder auch nicht. Diese etwas verwirrend erscheinende Namensgebung soll nun am Beispiel von *glLight()* verdeutlicht werden:

void glLight{i f}{ v }(Glenum light, Glenmum name,TYPE parameter)

Der Parameter, auf den sich die Ausführungen beziehen, wird durch *TYPE* gekennzeichnet. Demnach kann also dieser Parameter im Befehl glLight() in folgenden Varianten auftreten:

als GLint in der nicht–vektoriellen Form (i)

als const GLint* in vektorieller Form (i v)

als GLfloat in der nicht-vektoriellen Form (f)

als const GLfloat* in vektorieller Form (f v)

Damit entstehen folgende vier Aufrufvarianten von glLight():

void glLighti(...,GLint lala);

void glLightf(...,GLfloat lala);

void glLightiv(...,const GLint lala);*

void glLightfv(...,const GLfloat lala);*

Welche Aufrufform nun im speziellen Fall benutzt wird, hängt von verschiedenen Überlegungen ab: Wird die nicht-vektorielle Aufrufform benutzt, ist es nicht immer möglich, alle gewünschten Parameter an die jeweilige Prozedur zu übergeben. Wenn nun Parameter direkt als Vektor verwaltet werden, dann liegt der vektorielle Aufruf der Prozedur nahe. Es ist grundsätzlich immer dem Benutzer von OpenGL vorbehalten, die jeweilig für ihn passende Aufrufform der GL-Kommandos zu wählen.

Damit wären im Prinzip alle grundsätzlichen Aspekte der Nomenklatur für diese Grafik - Bibliothek genannt. Sicherheit im Umgang mit Variablen und Prozeduren wird natürlich wie immer durch die Praxis erlangt.

4 GRAPHISCHE PRIMITIVE

Graphische Primitive sind Ausgabeelemente in der Computergrafik. Jedes grafische Objekt wird durch Punkte charakterisiert. Ein grafisches Primitiv wird in OpenGL wie folgt definiert:

```
glBegin (primitiv-name)
    glVertex*
    ...
    glVertex*
glEnd()
```

Dies ist die allgemeingültige Vereinbarung für Primitive in OpenGL. Wie die in *glBegin()* und *glEnd()* geklammerten Punkte interpretiert werden, hängt vom Parameter *primitiv-name* ab. Jetzt kommen wir zur Programmierung der OpenGL – Ausgabeelemente:

```
glBegin (GL_LINES)
    glVertex2f(...) // 1. Punkt
    ...
    glVertex2f(...) // n. Punkt
glEnd()
```

Diese Vertices, die mit *glVertex2f()* definiert werden, liegen in einer Ebene bei z=0. Indikator hierfür ist die „2“ im Routinennamen.

4.1 Punkte

Wenn wir nun den Parameter *primitiv-name* durch ***GL_POINTS*** ersetzen, dann wird jedes Vertex als ein Punkt interpretiert. Wir setzen also mit unserer oberen Schleife n Punkte im 2-dimensionalen Raum. Hierbei kann nun der Durchmesser der Einzelpunkte durch

```
glPointSize( GLfloat size );
```

spezifiziert. Die Größe *size* ist hierbei als Größe in Pixeln zu verstehen. Der voreingestellte Wert hierfür ist 1.0f. Für die Größe gilt *size*>0.0f. Die Minimalgröße und die Maximalgröße sind implementierungsabhängig und können durch *glGetFloatv(GLenum pname, GLfloat * params)* erfragt werden. Als Parameter für *pname* ist hier ***GL_POINT_SIZE_RANGE*** zu übergeben. Der erste Wert in der Liste ist das Minimum und der zweite ist das Maximum. Die Darstellung der Punkte ist abhängig davon, ob Antialiasing eingeschaltet ist, oder eben nicht. Die Punktgröße ist in Stufen einstellbar. Der entsprechende Inkrementierungswert ist mit *glGet*-Funktion und dem Parameter ***GL_POINT_SIZE_GRANULARITY*** erfragbar. Doch diese Werte sind wie schon erwähnt implementierungsabhängig, aber eine Punktgröße von *1.0f* muß unterstützt werden.

4.2 Linien

Wie schon hinreichend bekannt, wird als Linie eine Verbindung zwischen zwei Punkten gezeichnet. Und eben diese zwei Punkte werden bei Angabe des Parameters ***GL_LINES*** miteinander verbunden. Wird also dieser Parameter angegeben, dann verbindet die OpenGL die Vertices [1-2,3-4,5-6,...] zwischen den Befehlen

glBegin()

und

glEnd().

Wird zwischen den Befehlen *glBegin()* und *glEnd()* eine ungerade Anzahl an Vertices übergeben, wird der letzte ignoriert.

Wenn nun ein Linienzug erzeugt werden soll, ist für *primitiv-name* ***GL_LINE_STRIP*** einzusetzen. Das Ergebnis ist, wie nicht anders zu erwarten, ein Linienzug, der die Vertices in der angegebenen Reihenfolge verbindet.

Als letzte Möglichkeit, eine(n) Linie(nzug) mit OpenGL zu erzeugen, wird ***GL_LINE_LOOP*** durch die Bibliothek angeboten. Dieser Parameter ist fast identisch mit ***GL_LINE_LINE_STRIP*** nur mit dem Unterschied, daß der letzte Punkt mit dem ersten verbunden wird.

Nun besteht natürlich auch analog zu *glPointSize()* die Möglichkeit die Liniendicke zu ändern. Der Befehl hierfür lautet *glLineWidth(GLfloat width).* Die Einheit für *width* ist (vergleiche auch *glPointSize()*)das Pixel. Auch hier muß *width>0.0f* gelten. Die Voreinstellung für *width* ist *1.0f*. Die maximale Linienstärke kann hier mit dem Parameter ***GL_LINE_WIDTH_RANGE*** erfragt werden. Linien müssen natürlich nicht unbedingt monoton dargestellt werden. Sie können auch mit Mustern zur Anzeige gebracht werden. Um solche Linien darzustellen bedarf es der Befehlssequenz:

*glEnable (**GL_LINE_STRIPPLE**);*

glLineStipple(GLint faktor, GLushort muster);

Mit *glEnable (**GL_LINE_STRIPPLE**);* wird zunächst die Musterdarstellung für Linien ermöglicht und mit dem zweiten Befehl wird das Muster definiert. *muster* wird als Muster interpretiert. *muster* ist vom Typ *GLushort* und damit 16 Bit lang und in eben diesen 16 Bit bedeutet eine binäre „0“ einen Zwischenraum und eine „1“ einen gesetzten Pixel. Der Parameter *faktor* gibt an, wie stark das Muster *muster* in die Länge gezogen wird.

4.3 Polygone

Ein Polygon ist ein durch die an OpenGL übergebenen Vertices definiertes n-Eck. Das visuelle Ergebnis ähnelt sehr dem Primitiv ***GL_LINE_LOOP***, allerdings ist dieses n-Eck gefüllt. In OpenGL gelten für Polygone folgende Einschränkungen:

- Polygone müssen konvex sein,
- die Kanten der Polygone dürfen sich nicht überschneiden,
- Polygone müssen planar sein.

Warum gelten nun diese Konventionen ? Der einzige Grund hierfür ist die bessere Performance, die mit solchen konvexen Polygonen erreicht werden kann. 3D-Beschleunigerkarten zerlegen intern die Polygone in Dreiecke, und diese Zerlegung ist am besten mit konvexen Polygonen realisierbar. Ist ein Polygon nun doch konkav, so ist das Ergebnis der Darstellung nicht vorhersehbar.

Das Darstellen von Polygonen ist in drei verschiedenen Modi möglich:

GL_POINT es werden nur die Eckpunkte gezeichnet

GL_LINE es werden nur die Kanten gezeichnet

GL_FILL das Polygon wird gefüllt dargestellt

Es darf an dieser Stelle nicht vergessen werden, daß wir uns im 3-dimensionalem Raum befinden. Daher hat jedes Polygon eine Vorder- und eine Rückseite. Der Darstellungsmodus der betreffenden Seite des Polygons kann mit der Funktion *glPolygonMode(GLenum face, GLenum mode)* geändert werden. Hierbei erwartet die Funktion für *face* entweder ***GL_FRONT*** oder ***GL_BACK*** oder eben ***GL_FRONT_AND_BACK***. Für *mode* sind die oben genannten Konstanten möglich. Ein Polygon kann also durchaus von vorn anders aussehen als von hinten.

Zum Festlegen von Füllmustern von Polygonen bietet die OpenGL die Funktionen:

*glPolygonStipple(GLubyte *maske);*

und

glEnable(GLenum was_denn_nun_eigentlich);

Analog zur Definition von Linientypen ist hier für *was_denn_nun_eigentlich* ***GL_POLYGON_STIPPLE*** einzusetzen. Der Parameter *maske* ist diesmal ein Zeiger auf ein Feld von GLubyte. Durch die in diesem Feld kodierten binären Zahlen wird jetzt das Füllmuster definiert.

4.4 Komplexere geometrische Primitive in OpenGL

Einzelne gefüllte Vierecke werden durch das Primitiv ***GL_QUADS*** erzeugt. So werden hier, ähnlich zum Vorgehen bei ***GL_LINES*** die vier aufeinanderfolgenden Punkte (Vertices) zu einem Viereck verbunden, die mit *glBegin()* und *glEnd()* geklammert wurden.

Etwas anders läuft das bei der Darstellung von Streifen, die aus Vierecken zusammengesetzt sind. Hier werden die ersten vier Vertices zu einem Viereck verbunden und die jeweils nachfolgenden zwei Vertices werden an die vorhergehenden zwei Punkte „angebunden“. Dadurch entsteht ein Viereckzug. Es ist darauf zu achten, daß immer die nachfolgenden zwei Punkte „über Kreuz“ anzugeben sind. Was passieren würde, wenn die Punkte in anderer Reihenfolge angegeben würden, kann sich der Leser sicher selbst ausmalen. Um diese Form von Primitiven zu erzeugen, muß die Konstante ***GL_QUAD_STRIP*** benutzt werden.

Nach der vorangegangenen Performance-Argumentation hin zu Dreiecken , ist natürlich auch an die Implementation dieser in der OpenGL gedacht worden. Für die Implementierung der Dreiecke gibt es in OpenGL drei verschiedene Möglichkeiten:

GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

Durch das Primitiv ***GL_TRIANGLES*** werden fortlaufend drei Punkte zwischen *glBegin()* und *glEnd()* „hergenommen“ und verbunden. Überschüssige Punkte werden hierbei ignoriert.

Im Gegensatz dazu werden bei Angabe von ***GL_TRIANGLE_STRIP*** alle Punkte in der Liste zwischen *glBegin()* und *glEnd()* beachtet und zwar derart, daß die ersten drei Punkte mit Linien verbunden werden und dann fortlaufend jeweils die beiden letzten Punkte des Vorgängerdreiecks und der nächste Punkt usw.

Wird nun als Parameter ***GL_TRIANGLE_FAN*** gewählt, dann dient der erste Punkt als gemeinsamer Anfangspunkt für alle folgenden Dreiecke. Das erste Dreieck wird aus den drei ersten Punkten gebildet, jedes weitere Dreieck besteht aus dem gemeinsamen Eckpunkt, dem nächsten Punkt der Liste und dem letzten verwendeten Punkt der Liste. Das Ergebnis hiervon ähnelt einem Fächer.

Aber wozu gibt es nun solche komplexen Primitive ??? Die erste Antwort hierfür liegt auf der Hand. Zeichnet man z.B. drei zusammenhängende Dreiecke, so sind dafür neun Eckpunkte notwendig, bei ***GL_TRIANGLE_STRIP*** nur fünf. Wenn man jetzt noch bedenkt, daß zu jedem Vertex noch Farbwert, Normalenvektor, Materialspezifikation und Texturkoordinaten gehören dürfen, sollte die Ersparnis von Speicherplatz und die Verringerung der Rechenzeit dem Programmierer auffallen.

Damit sollten die grafischen Primitive erst einmal hinreichend behandelt sein. Weiterführend zu diesem Kapitel verweisen die Autoren auf die im Anhang genannten Literaturquellen.

5 LANGSAM – LANGSAMER - PRIMITIVE HÖHERER ORDNUNG

Um mit Primitiven höherer Ordnung in OpenGL zu arbeiten gibt es mehrere Möglichkeiten. Zum einen können diese selbst erstellt werden, man denke z.B. an Displaylisten (siehe Kapitel 6). Zum zweiten bietet die Utility-Library, die **GLU**, hierfür einiges an. Die dritte Möglichkeit komplexere Primitive darzustellen ist die **GLAUX.** Wie werden diese aber für ein „normales“ OpenGL-Programm unter WindowsXX verfügbar ? Zunächst sollte in der MinOpenGLView.cpp die Anweisung

include <gl/glu.h>

bzw.

include <gl/glaux.h>

eingefügt werden. Zum zweiten ist es dringend erforderlich, Visual C++ die **GLU32.DLL** und die **GLAUX.LIB** verfügbar zu machen. Dafür müssen die **GLU32.LIB** und die **GLAUX.LIB** unter **Settings...** im Menü **Project** auf dem **Link**-Tab bei **Object/library modules** eingefügt werden. Jetzt können wir über den Funktionsumfang der beiden Bibliotheken verfügen.

5.1 Primitive der GLU

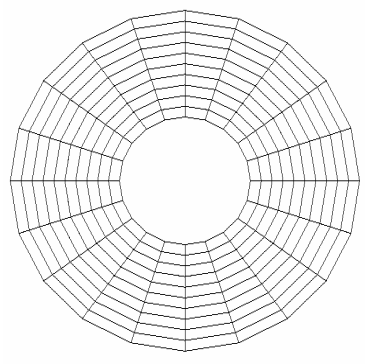
Fest vordefinierte geometrische Objekte heißen in der **GLU** *Quadrics*. Man definiert sich ein solches durch die Funktion:

GLUquadricObj gluNewQuadric();*

Diese liefert einen Zeiger auf ein Quadrics-Objekt zurück. Der Zeiger dient als Referenz für einen folgenden Aufruf. Nun kommen wir zu den Komplexen im einzelnen:

- *gluDisk(*
  GLUquadricObj qobj,*
  GLdouble innerRadius,
  GLdouble outerRadius,
  GLint slices,
  GLint loops
  );

  gluDisk() definiert eine Scheibe, dem Innerradius *innerRadius*, dem Außenradius *outerRadius*, den „Tortenstücken“ *slices* und *loops* Ringen.

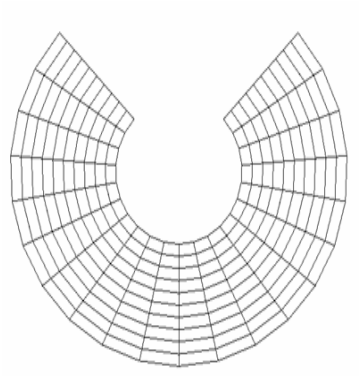


InnerRadius = 0.3
OuterRadius = 0.8
slices = 20
loops = 10

gluDisk()

- *gluPartialDisk(*
  GLUquadricObj qobj,*
  GLdouble innerRadius,
  GLdouble outerRadius,
  GLint slices,
  GLint loops,
  GLdouble startAngle,
  GLdouble sweepAngle
  );

  gluPartialDisk() definiert ein Kreissegment analog zu *gluDisk()* , startAngle definiert den Startwinkel und sweepAngle legt fest, wieviel Grad bis zum Ende in Uhrzeigerrichtung zu durchlaufen sind.



Slice = 20
loops = 10
startAngle = 45
sweepAngle = 270

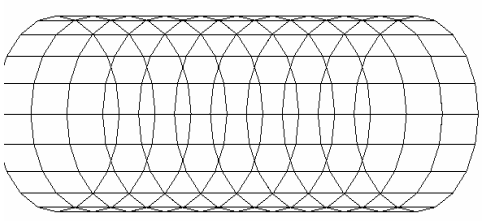
gluPartialDisk()

- *gluCylinder(*
  GLUquadricObj qobj,*
  GLdouble baseRadius,
  GLdouble topRadius,
  GLdouble height,
  GLint slices,
  GLint stacks
  );

  gluCylinder() definiert einen Zylinder mit dem Anfangsradius *baseRadius* und dem Endradius *topRadius*. Seine Höhe beträgt *height*, die Anzahl seiner Scheiben *slice* und die Anzahl der Segmente *stacks*.

BaseRadius = 0.3
topRadius = 0.3
height = 1.0
slice = 20
stacks = 10

gluCylinder()

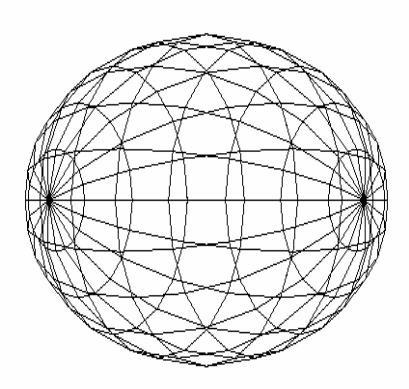


- *gluSphere(*
  GLUquadricObj qobj,*
  GLdouble radius,
  GLint slices,
  GLint stacks
  );

  gluSphere() definiert eine Kugel mit dem Radius *radius*, *slice* Scheiben und *stacks* Segmenten.

radius = 0.5
slice = 20
stacks = 10

gluSphere()



5.2 Angabe zusätzlicher Parameter bei GLU - Primitiven

Neben der Definition von Primitiven ist es auch möglich verschiedene Parameter dieser Objekte automatisch erzeugen zu lassen. Dazu stellt die *GLU* folgende Funktionen zur Verfügung:

void gluQuadricDrawStyle(
GLUquadricObj qobj,*
GLenum drawStyle
);

Mit Hilfe dieser Funktion legt man für ein Quadric *qobj* fest, wie es dargestellt wird. Der Parameter *drawStyle* gibt dabei die Darstellungsform an. Die folgenden symbolischen Konstanten sind möglich:

Symbolische Konstante	***Bedeutung***
GLU_FILL	Das Quadric wird aus gefüllten Polygonen zusammengesetzt. Die Polygone werden dabei entgegen dem Uhrzeigersinn gezeichnet (unter Berücksichtigung der Normalenvektoren).
GLU_LINE	Das Quadric wird als Drahtgittermodell dargestellt.

GLU_SILHOUETTE	Das Quadric wird als Drahtgittermodell dargestellt. Dabei werden die Begrenzungslinien zwischen koplanaren Flächen *nicht* gezeichnet.
GLU_POINT	Nur die Eckpunkte der Quadricflächen werden gezeichnet.

Beispiel:

```
glClearColor(0,0,0,0);
glClear(GL_COLOR_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);

GLUquadricObj* disk = gluNewQuadric();

glTranslatef(-0.5,-0.5,0);
gluQuadricDrawStyle(disk,GLU_FILL);
gluDisk(disk,0.2,0.5,30,7);

glTranslatef(0,1,0);
gluQuadricDrawStyle(disk,GLU_LINE);
gluDisk(disk,0.2,0.5,30,7);

glTranslatef(1,0,0);
gluQuadricDrawStyle(disk,GLU_SILHOUETTE);
gluDisk(disk,0.2,0.5,30,7);

glTranslatef(0,-1,0);
gluQuadricDrawStyle(disk,GLU_POINT);
gluDisk(disk,0.2,0.5,30,7);

glFlush();
```

Es wird eine *gluDisk* in allen Spielarten der GLU gezeichnet.

Die GLU ermöglicht es auch, zu den Flächen eines *Quadric* Normalenvektoren generieren zu lassen, was für die Beleuchtungsrechnung nicht ganz unwichtig ist. Die entsprechende Funktion ist

void gluQuadricNormals(

GLUquadricObj qobj,*

GLenum normals

);

Der Parameter *qobj* spezifiziert das Quadric, auf das sich die Einstellungen beziehen. Für *normals* sind folgende symbolische Konstanten möglich

Symbolische Konstante	***Bedeutung***
GLU_NONE	Es werden keine Normalenvektoren generiert.
GLU_FLAT	Pro Fläche des Quadric wird ein Normalenvektor generiert.
GLU_SMOOTH	Pro Vertex des Quadric wird ein Normalenvektor generiert.

Mit

```
void gluQuadricOrientation(
    GLUquadricObj * qobj,
    GLenum orientation
    );
```

wird die Orientierung der Flächen des *Quadric* spezifiziert, das heißt, ob die Normalenvektoren nach innen oder außen zeigen. *qobj* gibt an für welches *Quadric* die Einstellungen gemacht werden. Für *orientation* sind folgende symbolische Konstanten möglich:

Symbolische Konstante	***Bedeutung***
GLU_OUTSIDE	Die Flächen werden mit nach außen zeigenden Normalenvektoren generiert. Dies ist voreingestellt.
GLU_INSIDE	Die Flächen werden mit nach innen zeigenden Normalenvektor generiert.

Die automatische Generierung von Texturkoordinaten wird mit

```
void gluQuadricTexture(
    GLUquadricObj* qobj,
    GLboolean textureCoords
    );
```

beeinflußt. *qobj* gibt an, für welches *Quadric* die Einstellungen gemacht werden. *textureCoords* kann die Werte ***GL_TRUE*** oder ***GL_FALSE*** annehmen. Die Art der Generierung von Texturkoordinaten hängt vom Typ des gezeichneten *Quadric* ab.

Beispiel:

```
void CMinOpenGLView::DrawWithGL(HDC hdc)
{
      glEnable(GL_CULL_FACE);
      glClearColor(0,0,0,0);
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(1,1,1);

      make_texture();
      glEnable(GL_TEXTURE_2D);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
      glTexImage2D(GL_TEXTURE_2D, 0, 3, 256, 256, 0,
                        GL_RGB, GL_UNSIGNED_BYTE, RGBTexture);
      GLUquadricObj* kugel = gluNewQuadric();
      gluQuadricTexture(kugel,GL_TRUE);
      glMatrixMode(GL_MODELVIEW);
      for (int i=0;i<360;i++){
            glRotatef(3,0,1,0);
            gluSphere(kugel,0.8,100,100);
            glFlush();
            SwapBuffers(hdc);
      }
}
```

Texturen werden in einem späteren Kapitel ausführlich behandelt.

5.3 Primitive der GLAUX

Die Glaux bietet folgende „Grundkörper“:

- **Kugel**

  auxWireSphere(
  GLdouble radius
  );

  auxSolidSphere(
  GLdouble radius
  );

- **Würfel**

  auxWireCube(
  GLdouble size
  );

  auxSolidCube(
  GLdouble size
  );

- **Quader**

  auxWireBox(
  GLdouble width,
  GLdouble height,

```
        GLdouble depth
        );

    auxSolidBox(
        GLdouble width,
        GLdouble height,
        GLdouble depth
        );
```

- **Torus**

```
    auxWireTorus(
        GLdouble innerRadius,
        GLdouble outerRadius
        );

    auxSolidTorus(
        GLdouble innerRadius,
        GLdouble outerRadius
        );
```

- **Zylinder**

```
    auxWireCylinder(
        GLdouble radius,
        GLdouble height
        );

    auxSolidCylinder(
        GLdouble radius,
        GLdouble height
        );
```

- **Ikosaeder**

```
    auxWireIcosahedron(
        GLdouble radius
        );

    auxSolidIcosahedron(
        GLdouble radius
        );
```

- **Oktaeder**

```
    auxWireOctahedron(
        GLdouble radius
        );

    auxSolidOctahedron(
        GLdouble radius
        );
```

- **Tetraeder**

```
    auxWireTetrahedron(
```

```
        GLdouble radius
        );
auxSolidTetrahedron(
        GLdouble radius
        );
```

- **Dodekaeder**

```
auxDodecahedron(
        GLdouble radius
        );

auxDodecahedron(
        GLdouble radius
        );
```

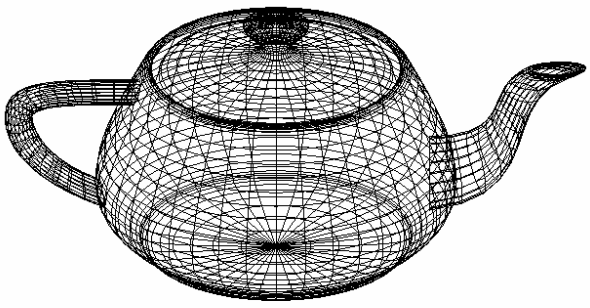
- **Kegel**

```
auxWireCone(
        GLdouble radius,
        GLdouble height
        );

auxSolidCone(
        GLdouble radius,
        GLdouble height
        );
```

- **„Teapot“**

```
auxWireTeapot(
        GLdouble size
        );

auxSolidTeapot(
        GLdouble size
        );
```

Die Formen *auxWireX* und *auxSolidX* erzeugen jeweils den gleichen Körper, wobei *auxWireX* diesen als Drahtgitter darstellt, und *auxSolidX* als soliden Körper. Diese Körper sind teilweise sehr komplex und bestehen aus vielen Flächen. So sieht z.B. eine Teekanne aus:



Kanne!

Das soll alles gewesen sein zu den komplexeren Primitiven der Erweiterungsbibliotheken. Die Parameter der **GLAUX** Funktionen sollten selbstreferenziell sein. Der geneigte Leser möge sich ein bißchen an diesen Primitiven probieren, um im Umgang mit denselben Erfahrung zu gewinnen.

6 DARSTELLUNGSLISTEN DER OPENGL

Ein wichtiges Problem bei grafischen Systemen ist die Wiederverwendbarkeit von komplexeren grafischen Objekten. Man stelle sich vor, man möchte ein Objekt definieren und später benutzen, daß so in dieser Form des öfteren in der gerenderten Szene auftritt. Mit dem bisherigen Wissen würde man dieses Objekt wahrscheinlich an allen Stellen, an denen es auftritt, wieder neu definieren, etwa so:

```
glBegin(GLenum mode)
    ...
glEnd()
```

Daß dieser Stil der Grafikprogrammierung auf Dauer umständlich und Speicherplatz verschwendend ist, liegt auf der Hand. Ein Mittel zur vorläufigen Speicherung von geometrischen Primitiven oder Zustandsänderungen bieten Darstellungslisten oder ***display lists***. Eine Darstellungsliste ist eine Gruppe von OpenGL – Kommandos. Wenn eine Darstellungsliste zur Ausführung kommt, dann werden die Befehle in der Reihenfolge ausgeführt, in der sie in der Liste gespeichert wurden. Die grundsätzliche Definition einer solchen *display list* sieht wie folgt aus:

```
glNewList( GLuint list, GLenum mode );
    ... // grafische Verrenkungen
glEndList();
```

Für den Parameter *list* wird hier ein „Name“ erwartet. Da es sich hier um einen *unsigned int*, also einen positiven Integer-Wert handelt, kann an dieser Stelle nur eine Konstante erwartet werden. Hierfür sollte zunächst eine Konstante mittels der Compilerdirektive ***#define*** definiert werden, beispielsweise

```
#define BLA_BEZEICHNUNG 2;
```

Nun wird der Parameter ***BLA_BEZEICHNUNG*** für *list* eingesetzt und unsere Liste hat einen „Namen“. Wir werden später noch sehen, dass für den Namen der Displayliste die Zahlentypen recht flexibel sind. Der zweite Parameter *mode* erwartet eine der beiden Konstanten, ***GL_COMPILE*** oder ***GL_COMPILE_AND_EXECUTE***. Hierbei bedeutetet die erste Konstante ***GL_COMPILE***, dass die Kommandos in der Liste nur compiliert, also zusammengestellt, aber nicht ausgeführt werden und ***GL_COMPILE_AND_EXECUTE*** besagt, dass die Kommandos zusammengestellt und zur Ausführung gebracht werden. Ein Beispiel hierfür ist die nachfolgende Definition eines Würfels:

```
#define CUBE 42

...

glNewList(CUBE, GL_COMPILE);
```

```
      glColor3f(0.0, 1.0, 0.0);
      // Draw the sides of the cube
      glBegin(GL_QUAD_STRIP);
            glVertex3d(3, 3, -3);
            glVertex3d(3, -3, -3);
            glVertex3d(-3, 3, -3);
            glVertex3d(-3, -3, -3);
            glVertex3d(-3, 3, 3);
            glVertex3d(-3, -3, 3);
            glVertex3d(3, 3, 3);
            glVertex3d(3, -3, 3);
            glVertex3d(3, 3, -3);
            glVertex3d(3, -3, -3);
      glEnd();

      glColor3f(0.0, 0.0, 1.0);
      // Draw the top and bottom of the cube
      glBegin(GL_QUADS);
            glVertex3d(-3, -3, -3);
            glVertex3d(3, -3, -3);
            glVertex3d(3, -3, 3);
            glVertex3d(-3, -3, 3);
            glVertex3d(-3, 3, -3);
            glVertex3d(3, 3, -3);
            glVertex3d(3, 3, 3);
            glVertex3d(-3, 3, 3);
      glEnd();
glEndList();
```

Wie an diesem Beispiel deutlich wird, kann eine Listendefinition natürlich auch mehrere *glBegin() – glEnd()* – Blöcke enthalten. Im Beispiel wird zunächst die Mantelfläche des Würfels mittels ***GL_QUAD_STRIP*** beschrieben, dann folgt die Definition der Ober – und Unterflächen. Diese Liste wurde mit dem ***GL_COMPILE*** – Parameter zusammengestellt, was bedeutet, daß ***CUBE*** der weiteren Verwendung harrt.

Wenn nun umgangen werden soll, daß die Konstanten für list in

glNewList(

GLuint list,

GLenum mode

);

aufs Geradewohl geraten werden müssen, sollte man sich folgender Funktion bedienen:

GLuint glGenLists(

GLsizei range

);

Diese Funktion liefert einen nicht vorzeichenbehafteten Integer *n* zurück und zwar derart, daß eine Liste von Displaylisten erstellt wurde, die mit dem Wert *n* beginnt und beim Wert *n+range-1* endet. Sie bietet eine komfortable Möglichkeit, Displaylisten zu kreieren ohne Gefahr zu laufen, dass sich Konstanten überschneiden könnten. Sollte ein Fehler bei der Erstellung der Listen auftreten, dann liefert diese Funktion 0 zurück. Demnach ist ein Rückgabewert ≠ 0 ein Indikator dafür, daß diese Listen erzeugt wurden.

Soll nun ein potentieller Listenkandidat auf seine Identität überprüft werden, so bedient man sich der Funktion

GLboolean glIsList(
GLuint list
);

wobei für *list* der zu überprüfende Platzhalter eingetragen werden muß. Ist der für *list* angegebene Wert eine Liste, dann liefert diese Funktion ***GL_TRUE*** zurück, andernfalls ***GL_FALSE***.

Wie schon eingangs erwähnt, lassen sich mit *glNewList()* Displaylisten kreieren, die nicht unbedingt sofort zur Ausführung gelangen müssen. Soll nun eine solche Liste ausgeführt werden, dann steht hierfür u.a. die Funktion

glCallList(
GLuint list
);

zur Verfügung. Diese Funktion ist vom Typ *void*, hat also keinen Rückgabewert, wofür auch??? *glCallList()* erzwingt die sofortige Ausführung der in *list* übergebenen Liste. Ist diese Liste ungültig, so wird dieses Kommando ignoriert.

Diese Funktion *glCallList()* kann allerdings auch innerhalb einer Displaylistendefintion auftauchen. Um nun eine potentielle Endlosrekursion zu verhindern, wird in der OpenGL ein Limit für die Rekursionsschritte gesetzt. Dieses *nesting* – Limit ist implementationsabhängig, beträgt aber mindestens 64. Dieser Wert kann mit

*glGet(**GL_MAX_LIST_NESTING**)*

erfragt werden.

Der OpenGL – Status wird innerhalb eines Displaylisten – Aufrufs nicht gesichert, was für den Programmierer bedeutet, daß der Status der OpenGL mittels

glPushAttrib(),
glPopAttrib(),
glPushMatrix() und
glPopMatrix()

gesichert werden muß (diese Funktionen werden zu einem späteren Zeitpunkt besprochen).

Nun gibt es analog zu *glCallList()* natürlich auch die Möglichkeit, mehrere Displaylisten auszuführen. Das OpenGL – Kommando hierfür ist

```
glCallLists(
    GLsizei n,
    GLenum type,
    const GLvoid* lists
    );
```

Hierbei dient *n* als Platzhalter für die Anzahl der darzustellenden Listen, *type* ist Indikator für den Typ der im *lists* – Array übergebenen Displaynummernwerte. Für *type* stehen folgende Konstanten zur Verfügung:

Konstante	***Bedeutung***
GL_BYTE	Die in dem *lists* – Array übergebenen Werte werden als *signed Bytes* behandelt [-128,...,127].
GL_UNSIGNED_BYTE	Die in dem *lists* – Array übergebenen Werte werden als *unsigned Bytes* behandelt [0,...,255].
GL_SHORT	Die in dem *lists* – Array übergebenen Werte werden als *2-Byte signed Integer* behandelt [-32768,...,32767].
GL_UNSIGNED_SHORT	Die in dem *lists* – Array übergebenen Werte werden als *2-Byte unsigned Integer* behandelt [0,...,65535].
GL_INT	Die in dem *lists* – Array übergebenen Werte werden als *4-Byte signed Integer* behandelt.
Konstante	***Bedeutung***
GL_UNSIGNED_INT	Die in dem *lists* – Array übergebenen Werte werden als *4-Byte unsigned Integer* behandelt.
GL_FLOAT	Die in dem *lists* – Array übergebenen Werte werden als *4-Byte Fließkommazahlen* behandelt.

GL_2_BYTES	Die in dem *lists* – Array übergebenen Werte werden als *2-Byte unsigned Integer* – Paar behandelt. Jedes Paar repräsentiert einen Listennamen, der Wert des Listennamens ergibt sich aus 256 mal dem Wert des ersten *unsigned Bytes* plus dem *unsigned* Wert des zweiten Bytes. *Name = 256*Byte1st+Byte2nd;*
GL_3_BYTES	Die in dem lists – Array übergebenen Werte werden als *3-Byte unsigned Integer* – Tripel behandelt. Jedes Tripel repräsentiert einen Listennamen, der Wert des Listennamens ergibt sich aus 65535 mal dem Wert des ersten *unsigned Bytes* plus 256 mal dem Wert des zweiten *unsigned Bytes* plus dem *unsigned Wert* des dritten Bytes. *Name = 65535*Byte1st+256*Byte2nd+Byte3rd;*
GL_4_BYTES	Die in dem lists – Array übergebenen Werte werden als *4-Byte unsigned Integer* – quad –Tupel behandelt. Jedes quad - Tupel repräsentiert einen Listennamen, der Wert des Listennamens ergibt sich aus 16777216 mal dem Wert des ersten *unsigned Byte* plus 65535 mal dem Wert des zweiten *unsigned Bytes* plus 256 mal dem Wert des dritten *unsigned Bytes* plus dem *unsigned* Wert des vierten Bytes. *Name=16777216*Byte1st+65535*Byte2nd+256*Byte3rd+Byte 4th;*

Schließlich gilt es noch, *lists* zu spezifizieren. *lists* ist ein auf *Glvoid** gecasteter Zeiger auf das Array mit den für die Listennamen relevanten Einträgen. Hier ist ein *GLvoid* – Zeiger notwendig, weil die Art der Einträge in dem Array für OpenGL noch nicht vorhersehbar ist.

Da es nun die Möglichkeit zur Erzeugung von Displaylisten gibt, sollte es auch die Möglichkeit zum Löschen von bereits existierenden Darstellungslisten geben. Hierfür stellt die OpenGL die Funktion:

```
glDeleteLists(
        GLuint list,
        GLsizei range
        );
```

zur Verfügung. Hierbei gibt *list* den Integer – Wert der ersten zu löschenden Displayliste an. Es werden alle Displaylisten *d* im Bereich von *list <= d <= list + range –1* gelöscht Eine einzelne Displayliste kann durch *range = 0* gelöscht werden.

7 DIE DRITTE DIMENSION

Wir kennen jetzt im Prinzip schon alle OpenGL-Anweisungen, mit denen Objekte erzeugt werden können. Durch einfaches hinzufügen einer z-Koordinate bei der Definition der Vertices, ist es möglich Polygone im Raum zu definieren.

```
void glVertex3f(
        GLfloat x,
        GLfloat y,
        GLfloat z
        );
```

Und durch Zusammenfassen von Polygonen zu komplexeren Objekten können beliebig komplizierte Körper "zusammengebaut" werden.

7.1 Transformationen

Um aus verschiedenen Objekten eine Szene zu erstellen, bedient man sich verschiedener **Transformationen**, welche die Wahl eines Betrachterstandpunktes, die Festlegung einer Abbildungsvorschrift (Perspektive) sowie die Plazierung von Objekten im Raum realisieren. Die Transormationsprozesse, die bis zur fertigen Szene durchlaufen werden, können am besten durch die "Kameraanalogie" beschrieben werden. Beim Erstellen einer OpenGL-Szene gehen wir vor wie bei der Aufnahme eines Fotos:

1. Objekte in der Szene plazieren. Bei OpenGL werden die Objekte mit Hilfe von *Modelltransformationen* im Raum positioniert, skaliert bzw. gedreht.
2. Kameraposition festlegen. Der Betrachterstandpunkt wird durch die *Ansichtstransformation* festgelegt.
3. Wahl eines Objektives (z.B. Weitwinkelaufnahme), Festlegung der Brennweite. Der Strahlengang der Projektionsstrahlen wird mit der *Projektionstransformation* beeinflußt.
4. Abmaße der Fotografie bestimmen, Foto entwickeln. Die Gröβe und Position des Viewports auf dem Bildschirm wird durch die *Viewporttransformation* festgelegt.

Alle Transformationen werden mit Hilfe von Matrizen realisiert. Transformationsmatrizen sind grundsätzlich 4x4–Matrizen. Prinzipiell wird für die Anwendung einer Transformation auf ein Objekt jeder Definitionspunkt dieses Objekts mit der Transformationsmatrix multipliziert:

Vorraussetzung dafür ist das Hinzufügen einer vierten Komponente zu den Definitionspunkten,

$$P' = M * P$$

der *homogenen Komponente w*. Meist wird als Wert für *w* 1 gewählt. Die Umwandlung von kartesischen in homogene Koordinaten erfolgt durch Multiplikation jeder Komponente mit *w*:

$$P_k = (x, y, z)$$
$$P_h = (x * w, y * w, z * w, w) \qquad , w \neq 0$$

Die Umwandlung von homogenen in kartesische Koordinaten erfolgt analog, mittels Division durch w. Richtungsvektoren, die ja als Differenz zwischen zwischen zwei Ortsvektoren dargestellt werden können, haben w=0 als homogene Koordinate, da alle Ortsvektoren mit w=1 definiert werden.

Die Transformation mittels Matrizen bietet den Vorteil, daß verschiedene Transformationen zu einer Gesamttransformationsmatrix zusammengefaßt werden können:

$$P' = M_{rotation} * M_{translation} * M_{skalierung} * P$$

$$M = M_{rotation} * M_{translation} * M_{skalierung}$$

$$P' = M * P$$

Dies führt, speziell bei Szenen mit sehr vielen Vertices, zu einem enormen Effektivitätsgewinn.

7.2 Die Matrizenstacks

Wie wir noch sehen werden, ist es oft nötig eine Vielzahl unterschiedlicher Matrizen für Modell- und Projektionstransformationen zu verwenden. Besonders beim Aufbau von hierarchischen Modellen wird dies deutlich. Hierbei definiert man komplexe Modelle aus Submodellen in eigenen lokalen Koordinatensystemen. Da es notwendig ist, sich die Transformationen auf die übergeordneten Koordinatensysteme zu merken, wurden *Matrizenstacks* eingeführt. In der OpenGL sind drei Matrizenstacks implementiert:

- der Modellierungs-Matrizenstack *(model view matrix stack),*
- der Projektions-Matrizenstack *(projection matrix stack)* und
- der Textur-Matrizenstack *(texture matrix stack).*

Der Textur-Matrizenstack wird im Rahmen dieses Lehrbriefes nicht behandelt. Das oberste Element auf dem Matrixstack ist die ***aktuelle Matrix***. Wenn ein Punkt gezeichnet wird, werden seine Koordinaten durch die aktuelle Modellierungsmatrix und durch die aktuelle Projektionsmatrix transformiert. Mit Hilfe der Funktion

```
glMatrixMode(
        GLenum mode
);
```

wird der aktive Matrixstack ausgewählt, auf den sich dann nachfolgende Operationen auswirken. **mode** kann die Werte *GL_MODELVIEW*, *GL_PROJECTION* oder *GL_TEXTURE* annehmen. Die Tiefe des Modellierungs-Matrixstacks ist mindestens 32, die Tiefe des Projektions-Matrixstacks mindestens 2.

glPushMatrix();

schiebt die Matrizen auf dem Stack um eins nach unten. Als oberstes Element (aktuelle Matrix), wird eine Kopie der alten aktuellen Matrix auf auf den Stack gelegt.

glPopMatrix();

schiebt alle Matrizen des Stacks um eins nach oben. Die aktuelle Matrix wird durch die auf dem Stack darunterliegende Matrix ersetzt.

glLoadIdentity();

ersetzt die aktuelle Matrix durch die Einheitsmatrix.

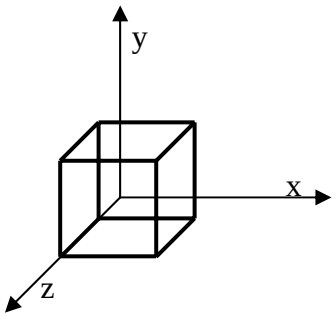
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

7.3 Die Modelltransformationen

Als Modelltransformationen werden bezeichnet:

- Skalierung von Objekten,
- Rotation von Objekten sowie
- Verschiebung von Objekten.

Um die Wirkung der Transformationen optisch nachvollziehen zu können, zeichnen wir zunächst einen Würfel aus sechs verschiedenfarbigen Polygonen, so daß der Mittelpunkt des Würfels im Koordinatenursprung liegt:



```
//Darstellungsliste für Würfel definieren
GLuint wuerfel = glGenLists(1);
glNewList(wuerfel,GL_COMPILE);
      //Unterseite  GRAU
      glBegin(GL_POLYGON);
            glColor3f(0.5f, 0.5f, 0.5f);
            glVertex3f(-0.3,  -0.3, 0.3);
            glVertex3f(0.3,   -0.3, 0.3);
            glVertex3f(0.3,   -0.3, -0.3);
            glVertex3f(-0.3,  -0.3, -0.3);
      glEnd();
      //Oberseite  WEISS
      glBegin(GL_POLYGON);
            glColor3f(1.0f, 1.0f, 1.0f);
            glVertex3f(-0.3,  0.3,  0.3);
            glVertex3f(-0.3,  0.3,  -0.3);
            glVertex3f(0.3,   0.3,  -0.3);
            glVertex3f(0.3,   0.3,  0.3);
      glEnd();
      //Vorderseite  ROT
      glBegin(GL_POLYGON);
            glColor3f(1.0f, 0.0f, 0.0f);
            glVertex3f(-0.3,  -0.3, 0.3);
            glVertex3f(-0.3,  0.3,  0.3);
            glVertex3f(0.3,   0.3,  0.3);
            glVertex3f(0.3,   -0.3, 0.3);
      glEnd();
      //Hinterseite  BLAU
      glBegin(GL_POLYGON);
            glColor3f(0.0f, 0.0f, 1.0f);
            glVertex3f(0.3,   -0.3, -0.3);
            glVertex3f(0.3,   0.3,  -0.3);
            glVertex3f(-0.3,  0.3,  -0.3);
            glVertex3f(-0.3,  -0.3, -0.3);
      glEnd();
      //linke Seite GRÜN
      glBegin(GL_POLYGON);
            glColor3f(0.0f, 1.0f, 0.0f);
            glVertex3f(-0.3,  -0.3, -0.3);
            glVertex3f(-0.3,  0.3,  -0.3);
            glVertex3f(-0.3,  0.3,  0.3);
            glVertex3f(-0.3,  -0.3, 0.3);
      glEnd();
      //rechte Seite GELB
      glBegin(GL_POLYGON);
            glColor3f(1.0f, 1.0f, 0.0f);
            glVertex3f(0.3,   -0.3, 0.3);
            glVertex3f(0.3,   0.3,  0.3);
            glVertex3f(0.3,   0.3,  -0.3);
            glVertex3f(0.3,   -0.3, -0.3);
glEnd();
glEndList();

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
glClear(GL_DEPTH_BUFFER_BIT);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
glFrustum(-1,1,-0.75,0.75,10,15);
glTranslatef(0,0,-12.5);

glCallList(wuerfel);
glFlush();
```

Der Würfel wird in einer Displaylist definiert, und diese mit *glCallList* zur Ausführung gebracht. Das Programmsegment

*glMatrixMode(**GL_PROJECTION**);*
glLoadIdentity();
glFrustum(-1,1,-0.75,0.75,10,15);
glTranslatef(0,0,-12.5);

definiert die Kameraposition und -perspektive. Nähere Erläuterungen dazu folgen an späterer Stelle. Zunächst genügt es zu wissen, daß die Kamera auf der z-Achse bei 12.5 liegt und in Richtung negative z-Achse gerichtet ist. Mit

*glEnable(**GL_DEPTH_TEST**);*
*glClear(**GL_DEPTH_BUFFER_BIT**);*

wird Z-Buffering zur Verdeckungsberechnung aktiviert, sowie der Z-Buffer gelöscht. Damit ist gesichert, das Flächen, die weiter vorn (auf der positiven z-Achse) liegen nicht durch Flächen verdeckt werden, die hinter ihnen liegen. Wird Z-Buffering disabled, *glDisable(**GL_DEPTH_TEST**)*, verdeckt die Rückseite des Würfels die Vorderseite, da OpenGL dann die Polygone in der gleichen Reihenfolge zeichnet, wie sie in der Displaylist definiert wurden.

Anhand dieses Würfels soll im folgenden die Wirkung der einzelnen Modelltransformationen gezeigt werden.

7.3.1 Translation

Um Objekte (bzw. ihre Definitionspunkte) im Raum zu verschieben, wird die Funktion *glTranslate* benutzt.

void glTranslated(
GLdouble x,
GLdouble y,
GLdouble z
);

void glTranslatef(
GLfloat x,

GLfloat y,

GLfloat z

);

glTranslate verschiebt den Koordinatenursprung an den Punkt (x,y,z), indem es die aktuelle Matrix mit einer Translationsmatrix multipliziert (M := M*T) :

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In der Praxis heißt das, daß alle Objekte, die nach dem Aufruf von *glTranslate* gezeichnet werden, um (x,y,z) verschoben werden.

Hinweis:

glTranslate darf nicht zwischen *glBegin()* und *glEnd()* aufgerufen werden, sonst wird die Fehlermeldung ***GL_INVALID_OPERATION*** erzeugt.

Beispiel:

```
glCallList(wuerfel);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glTranslatef(0.8,  0,  2);
glCallList(wuerfel);
glPopMatrix();
```

Der oben definierte Würfel wird zuerst im Ursprung gezeichnet. Dann wird die aktuelle Modelview-Matrix mit *glTranslatef* manipuliert, so daß der Ursprung um 0.8 in x-Richtung (nach rechts) und um 2 in z-Richtung (nach vorn) verschoben wird. Danach wird der Würfel nochmals gezeichnet, jetzt im neuen Ursprung, und somit verschoben.

7.3.2 Rotation

Um Objekte um einen bestimmten Winkel zu drehen, benutzt man die Funktion *glRotate*.

void glRotated(

GLdouble angle,

GLdouble x,

```
        GLdouble y,
        GLdouble z
        );

    void glRotatef(
        GLfloat angle,
        GLfloat x,
        GLfloat y,
        GLfloat z
        );
```

glRotate dreht das Koordinatensystem (und damit alle Objekte, die nach *glRotate* gezeichnet werden) um *angle* Grad gegen den Uhrzeigersinn, und zwar um die Achse, die vom Koordinatenursprung durch den Punkt *(x,y,z)* verläuft (gegen den Uhrzeigersinn, wenn man von der positiven Seite der Achse auf den Ursprung sieht). Der Koordinatenursprung ist dabei immer das Rotationszentrum.

glRotate multipliziert die aktuelle Matrix mit einer Rotationsmatrix (M := M*R).

Die Matrizen für die Rotation setzen sich folgendermaßen zusammen:

Rotation um die x-Achse:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um die y-Achse:

$$\begin{pmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um die z-Achse:

$$\begin{pmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Komplexe Rotationen werden auf einzelne Achsenrotationen zurückgeführt, und diese dann zu einer Gesamtrotationsmatrix multipliziert. *glRotate* übernimmt diese Arbeit, deshalb kann die Rotationsachse frei gewählt werden.

Hinweis:

glRotate darf nicht zwischen *glBegin()* und *glEnd()* aufgerufen werden, sonst wird die Fehlermeldung ***GL_INVALID_OPERATION*** erzeugt.

Beispiel:

```
glCallList(wuerfel);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glTranslatef(0.8,  0,  2);
glRotatef(45.0,  1,  1,  1);
glCallList(wuerfel);
glPopMatrix();
```

Der Würfel wird einmal im Ursprung gezeichnet, und einmal um 45 Grad um die Achse (1;1;1) gedreht bei (0.8;0;2).

7.3.3 Skalierung

Mittels einer Skalierung, kann man die Ausdehnung eines Objektes in allen drei Achsen manipulieren. Die OpenGL Funktion hierfür ist *glScale*.

```
void glScaled(
        GLdouble x,
        GLdouble y,
        GLdouble z
        );

void glScalef(
        GLfloat x,
        GLfloat y,
        GLfloat z
        );
```

x, *y* und *z* sind Skalierungsfaktoren zu den jeweiligen Achsen. Wenn negative Skalierungsfaktoren gewählt werden, so ist es möglich, Objekte zu spiegeln. Zum Beispiel *x = 1*, *y = -1* und *z = 1*

bewirken eine Spiegelung an der x-z-Ebene. In jedem Fall führen Skalierungsfaktoren, die betragsmäßig kleiner als eins sind zu einer Stauchung, und solche, die betragsmäßig größer als eins sind zu einer Streckung. *glScale* multipliziert die aktuelle Matrix mit einer Skalierungsmatrix (M := M*S).

Die Transformationsmatrix für die Skalierung lautet:

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Hinweis:

Wenn ein Objekt außerhalb des Ursprungs liegt, kann die Skalierung auch eine gewisse Translation mit sich bringen (bei Betrachtung einzelner Punkte ist dies ohnehin der Fall).

Hinweis:

glScale darf nicht zwischen *glBegin()* und *glEnd()* aufgerufen werden, sonst wird die Fehlermeldung ***GL_INVALID_OPERATION*** erzeugt.

Beispiel:

```
glCallList(wuerfel);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glTranslatef(0.8,  0,  0);
glScalef(1,  2,  -1);
glCallList(wuerfel);
glPopMatrix();
```

Der erste Würfel wird wie immer im Ursprung gezeichnet, der zweite bei (0.8; 0; 0). Der zweite Würfel wird mit *glScale* in y-Richtung gestreckt, und an der x-y-Achse gespiegelt (die blaue Seite zeigt nach vorn).

7.3.4 Kombination von Transformationen

Die Verkettung von Transformationen, erfolgt einfach durch die Multiplikation der einzelnen Transformationsmatrizen. Die Zeilen

```
glTranslatef(0.8,  0,  0);
glScalef(1,  2,  -1);
```

aus bewirken beispielsweise folgendes:

$$M_{akt} := M_{akt} * M_{trans}$$

$$M_{akt} := M_{akt} * M_{skal}$$

bzw.

$$M_{akt} := M_{akt} * M_{trans} * M_{skal}$$

Es ist zu beachten, daß die Transformation die am Ende steht zuerst ausgeführt wird (bedingt durch die Reihenfolge der Faktoren bei der Matrixmultiplikation). Bei obigem Beispiel heißt das, daß zuerst skaliert und dann verschoben wird. Es ist wichtig, sich bei zusammengesetzten Transformationen stets die Reihenfolge vor Augen zu halten (Wenn man sich um 180° dreht und 3 Schritte vorwärts geht, führt das zu einem anderen Ergebnis, als wenn man 3 Schritte vorwärts geht und sich um 180° dreht).

Beispiel: Rotation bezüglich eines Festpunktes

Angenommen, ein Würfel, der beliebig im Raum liegt, soll um seinen Schwerpunkt gedreht werden. Fällt der Schwerpunkt mit dem Koordinatenursprung zusammen, muß nur eine Rotation durchgeführt werden; die Rotationsmatrix bezieht sich ja auf den Ursprung.

Befindet sich der Schwerpunkt jedoch nicht im Ursprung (sondern im Punkt (x_f, y_f, z_f)), müssen mehrere Transformationen vorgenommen werden:

1. Fixpunkt der Rotation in den Koordinatenursprung verschieben

   glTranslatef(-x_f, -y_f, -z_f);

2. Rotation des Objektes um Ursprung

   glRotate();

3. Rückverschiebung des Fixpunktes

   glTranslatef(x_f, y_f, z_f);

Die zuerst auszuführende Transformation, muß beim Zusammensetzen der

$$M_{ges} = M_{trans\_rück} * M_{rotation} * M_{trans\_hin}$$

Gesamttransformationsmatrix als letzes Element stehen.

Das heißt, das Programmfragment, welches diese Fixpunktrotation realisiert, müßte so aussehen:

glTranslatef(x_f, y_f, z_f);

glRotate();

glTranslatef($-x_f$, $-y_f$, $-z_f$);

Will man ein Objekt bezüglich eines Fixpunktes skalieren, geht man analog vor.

7.4 Die Projektionstransformationen

Die folgenden Erklärungen sollen zeigen, wie der Betrachterstandpunkt und die Blickrichtung festgelegt werden und wie Parallelprojektion und Perspektivprojektion realisiert werden. Alle im folgenden eingeführten Funktionen realisieren Projektionstransformation. Also als erstes zum Projektions-Matrixstack wechseln:

*glMatrixMode(**GL_PROJECTION**);*

7.4.1 Ansichtstransformationen

Um die Szene aufnehmen zu können muß zunächst eine Kameraposition sowie die Blickrichtung festgelegt werden. Standardmäßig liegt der Betrachterstandpunkt im Ursprung des Modellkoordinatensystems und die Blickrichtung zeigt entlang der negativen z-Achse. Veränderungen lassen sich natürlich mit Hilfe von *glTranslate* und *glRotate* vornehmen (siehe Beispiel S.39). Dabei ist zu beachten, daß man nicht die Kamera bewegt, sondern das Koordinatensystem. Das heißt um den Punkt (x,y,z) zum Betrachterstandpunkt zu machen, ist der Befehl *glTranslate(-x,-y,-z)* nötig. Analog gilt dies für Rotation.

In der OpenGL Utility Library (GLU) ist glücklicherweise eine Funktion implementiert, die diese etwas verwirrende Aufgabe vereinfacht.

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

Die Parameter *eyeX*, *eyeY* und *eyeZ* legen den Standpunkt des Betrachters fest. *CenterX*, *centerY* und *centerZ* bezeichnen den Punkt, auf den die "Kamera" gerichtet ist, den ***"view reference point"***. *UpX*, *upY* und *upZ* bilden den sogenannten ***"view up vector"***, daß heißt einen Vektor, dessen

Richtung im "entwickelten Bild" nach oben zeigt. Mit Hilfe dieses Vektors kann also die Kamera z.B. auf den Kopf gestellt werden.

Im Prinzip passiert durch *gluLookAt()* nichts anderes, als daß das Koordinatensystem der Szene so transformiert wird, daß die Kamera im Ursprung liegt und in Richtung negative z-Achse schaut. Das so entstandene transformierte Koordinatensystem nennt man Kamerakoordinatensystem.

Hinweis:

Um Funktionen der GLU benutzen zu können, muß der entsprechende Header eingebunden werden:

```
#include <gl/glu.h>
```

Außerdem muß die glu32.lib zum Projekt hinzugelinkt werden. Dazu wählt man den Menüpunkt **Project → Settings** und fügt im erscheinenden Dialog im **Link**-Tab bei **Object/Library Modules** ***glu32.lib*** hinzu.

7.4.2 Parallelprojektion

Bei der Parallelprojektion treffen alle Sehstrahlen senkrecht auf die Projektionsfläche auf. Alle Sehstrahlen sind parallel, das heißt der Betrachterstandpunkt wird im Unendlichen angenommen.

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

glOrtho legt das Viewingvolumen fest, daß heißt ein Quader wird definiert. *left* und *right* bezeichnen die X-Koordinaten von linker und rechter Seite des Quaders. Die linke und die rechte Seite liegen parallel zur Y-Z-Ebene. *top* und *bottom* bezeichnen die Y-Koordinaten von Ober- und Unterseite, die parallel zur X-Z-Ebene liegen. *near* und *far* bezeichnen analog die Z-Koordinaten von Vorder und Rückseite. Alle Koordinaten beziehen sich auf das Kamerakoordinatensystem. Bei *near* und *far* sind negiert anzugeben (die Kamera blickt in Richtung negativer z-Achse). Der wert für *near* ist immer kleiner als der für *far.*

Alle Objekte die innerhalb des durch *glOrtho* definierten Quaders liegen werden dargestellt, alle die außerhalb liegen nicht. Die Seiten des Quader sind Clipping-Ebenen, an denen die Objekte "zerschnitten" werden, sollten sie zum Teil außerhalb liegen.

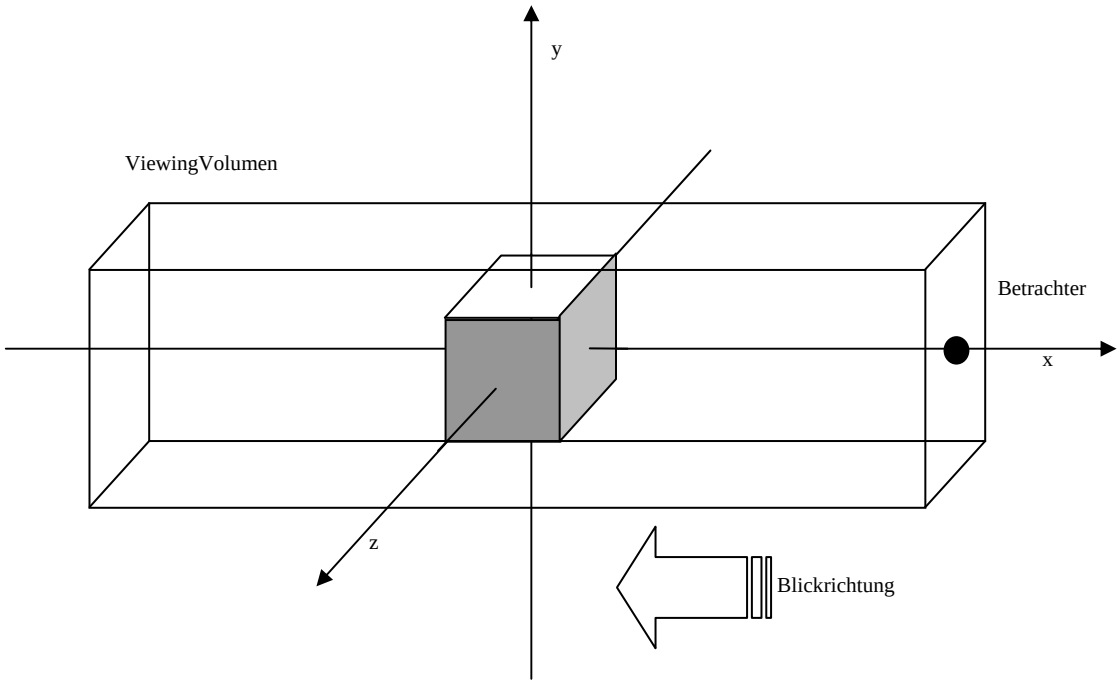
Beispiel:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
glOrtho(1,  -1,  1,  -1,  0,  10);
gluLookAt(5, 0, 0,   0, 0, 0,   0, 1, 0);
glCallList(wuerfel);
glFlush();
```

Der Betrachterstandpunkt liegt bei (5; 0; 0), also auf der positiven x-Achse. Der *view reference point* ist der Ursprung, die Kamera ist also entlang der negativen x-Achse gerichtet. der *view up vector* (0; 1; 0), parallel zur y-Achse zeigt auch im Modellkoordinatensystem nach oben.

Mit *glOrtho(1, -1, 1, -1, 0, 10)* wird die Ausdehnung des Viewingvolumens festgelegt: Linke, rechte, obere und untere Clipping-Ebene liegen jeweils im Abstand 1 zum Ursprung, die vordere liegt bei 0, also direkt auf dem Betrachterstandpunkt, die hintere im Abstand 10 zum Ursprung. Bezogen auf das Modellkoordinatesystem sieht das ungefähr so aus:

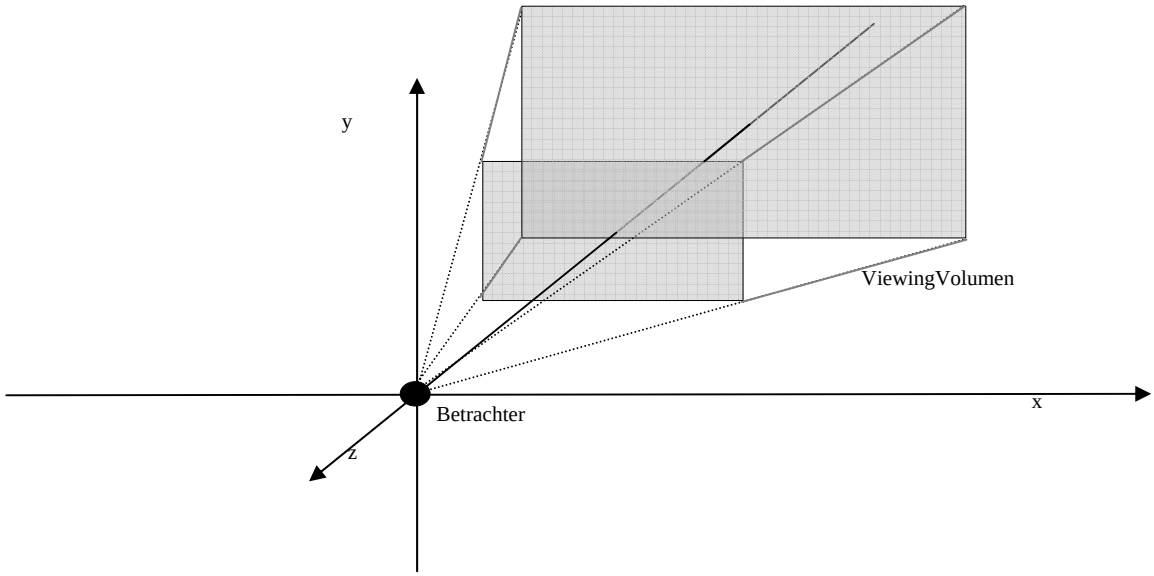


7.4.3 Perspektivprojektion

Die Perspektivprojektion entspricht eher als die Parallelprojektion der gewohnten Wahrnehmung (menschliches Auge, Kamera): weiter entfernte Objekte erscheinen kleiner als naheliegende.

```
void glFrustum(
        GLdouble left,
        GLdouble right,
        GLdouble bottom,
        GLdouble top,
        GLdouble znear,
        GLdouble zfar
        );
```

glFrustum() funktioniert ähnlich wie *glOrtho*. Allerdings wird diesmal kein Quader als Viewing-Volumen beschrieben, sondern ein Pyramidenstumpf, dessen "virtuelle Spitze" der Ursprung (und damit der Betrachter) ist:



Die Parameter *znear* und *zfar* geben wieder die Entfernung der vorderen bzw. hinteren Clipping-Ebene vom Betrachterstandpunkt in Richtung negative z-Achse an. Mit *left*, *right*, *bottom* und *top* wird die Ausdehnung der Vorderfront des Pyramidenstumpfes angegeben. Diese Vorderfront fällt mit der Projektionsfläche zusammen, das heißt die Punkte *(left, bottom, znear)* und *(right, top, znear)* werden in untere linke Ecke und die obere rechte Ecke des Ausgabefensters projiziert. (Die Koordinaten beziehen sich natürlich auf das Kamerakoordinatensystem, mit Betrachterstandpunkt im Ursprung.)

Wie gehabt, werden alle Objekte, die innerhalb des Viewing-Volumens liegen dargestellt, alle die außerhalb liegen nicht.

Beispiel:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum( -1,  1, -0.75,  0.75,  1.5,  3 );
gluLookAt( 1, 1, 1,  0, 0, 0,  -1, 1, -1 );
glCallList(wuerfel);
glFlush();
```

Der Betrachterstandpunkt liegt bei (1; 1; 1). Der *view reference point* ist (0; 0; 0), d.h. die Kamera ist auf den Ursprung gerichtet. Der *view up vector* ist (-1; 1; -1).

Mit *glFrustum(-1, 1, -0.75, 0.75, 1.5, 3)* wird die Ausdehnung des Viewingvolumens festgelegt. Als *znear* wurde 1.5 gewählt, was den interessanten Effekt hat, daß eine "Spitze" des

Würfels aus dem Viewing-Volumen herausragt. Diese Spitze wird abgeschnitten, so daß man in den Würfel hineinsehen kann.

Auch die GLU bietet eine Funktion zur Festlegung der Perspektive an:

```
void gluPerspective(
        GLdouble fovy,
        GLdouble aspect,
        GLdouble zNear,
        GLdouble zFar
        );
```

gluPerspective() macht genau das gleiche wie *glFrustum*, es definiert einen Pyramidenstumpf. Allerdings benutzt es andere Parameter zu dessen Beschreibung. *fovy* ist der Öffnungswinkel des Blickfeldes in y-Richtung. *aspect* ist das Verhältnis von x- zu y-Ausdehnung des Blickfeldes. *aspect=2* würde also heißen, daß das Blickfeld zweimal so breit ist wie hoch. *znear* und *zfar* entsprechen den gleichnamigen Parametern von *glFrustum().*

Ob *gluPerspective()* oder *glFrustum()* als anschaulicher empfunden wird, ist Geschmackssache.

7.4.4 Wichtiger Hinweis zu Projektionstransformationen

Man sollte sich ständig vor Augen halten, daß die GL-Funktionen zur Festlegung des Betrachterstandpunktes und der Perspektive grundsätzlich Transformationsmatrizen erzeugen, die mit der aktuellen Matrix verknüpft werden ($M_{akt}= M_{akt}*M_{transform}$). Das heißt, es ist **nicht** egal, in welcher Reihenfolge die Funktionen aufgerufen werden. Die Perspektivtransformationen werden als letztes ausgeführt (das heißt, die entsprechenden Funktionen müssen als erstes aufgerufen werden).

Zum Beispiel wäre **richtig**

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1,1,-0.75,0.75,1.5,3);
gluLookAt(1,1,1, 0,0,0, -1,1,-1);
```

Folgendes Fragment führt **nicht** zum selben Ergebnis:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluLookAt(1,1,1, 0,0,0, -1,1,-1);
glFrustum(-1,1,-0.75,0.75,1.5,3);
```

Ebensowenig sollte man erwarten, daß bei

```
gluLookAt(1,1,1, 0,0,0, -1,1,-1);
gluLookAt(3,1,3, 0,4,4, 0,1,0);
```

der zweite *gluLookAt* – Aufruf das Ergebnis des ersten „überschreibt“. Es werden stattdessen die Transformationsmatrizen die aus beiden Aufrufen erzeugt werden multipliziert.

Eine Ausnahme hiervon bildet die Viewporttransformation, die an beliebiger Stelle und beliebig oft nacheinander aufgerufen werden kann.

7.5 Der Viewport

Die GL-Funktionen zur Projektion, sorgen auch dafür, daß das definierte Viewingvolumen auf einen normierten Raum abgebildet wird, und zwar in einen Einheitswürfel, der in jeder Richtung die Ausdehnung 0 bis 1 hat. Um das zu realisieren benutzt man eine Normierungsmatrix. Diese wird von *glFrustum()* usw. bereits mit einbezogen, so daß *glFrustum()* im Prinzip folgendes

$$M_{aktuell} = M_{aktuell} * M_{normalisierung} * M_{perspektiv}$$

realisiert:

Die Szene liegt jetzt in Form von normierten Gerätekoordinaten, *normalized device coordinates*, vor, sozusagen als Negativ, um auf die Kamera-Analogie zurückzukommen. Aufgabe der Viewporttransformation ist es, das entgültige Bild daraus zu "entwickeln".

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

Der Viewport ist der Bereich des Ausgabefensters, in den gezeichnet wird. *glViewport* legt die Abmessungen und die Position dieses Bereiches fest. *(x,y)* ist die linke untere Ecke des Viewports, *width* und *height* geben die Ausdehnung an. Die normierten Koordinaten werden genau auf den Viewport gemappt. (0,0) wird auf (x,y) abgebildet, (1,1) auf (x+width,y+height).

Beispiel:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1,1,-0.75,0.75,1.5,3);
gluLookAt(1,1,1, 0,0,0, -1,1,-1);
glViewPort(0,0,80,80);
glCallList(wuerfel);
glFlush();
```

glViewPort(0,0,80,80) bewirkt, daß das Bild des Würfels auf einen 80x80 Pixel großen Bereich (ausgehend von (0,0), der unteren linken Ecke des Fensters) abgebildet wird.

Um das zur Verfügung stehende Fenster immer genau auszufüllen, kann folgendes Programmfragment genutzt werden:

```
void CMinGLView::OnDraw(CDC* pDC) {
    CMinGLDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    wglMakeCurrent(pDC->m_hDC, m_hRC);
    CRect vr;
    GetClientRect(&vr);
    glViewport(0,0,(GLint)(vr.right-vr.left),(GLint)(vr.bottom-vr.top));
    DrawWithGL();
    wglMakeCurrent(pDC->m_hDC, NULL);
}
```

Hinweis:

Wenn die Seitenverhältnisse von Viewingvolumen und Viewport nicht übereinstimmen, so führt das zu Stauchen bzw. Strecken des Bildes. Dieser Effekt kann natürlich auch bewußt angewandt werden.

8 VERDECKUNGSBERECHNUNG

8.1 Backface Culling

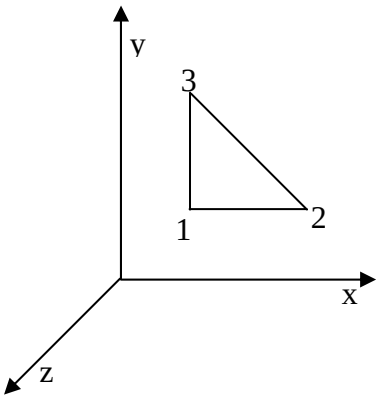
Bei geschlossenen Körpern, zum Beispiel bei einem Würfel, ist es nicht nötig, die Rückseiten der Flächen zu zeichnen. Die Rückseiten sind in diesem Fall die Seiten der Würfelflächen die nach innen zeigen. Wenn der Würfel nun gezeichnet wird, würden die Flächen, deren Rückseite man sieht, sowieso von den Flächen überschrieben, deren Vorderseite man sieht. Die Rückseiten müssen also nicht gezeichnet werden. Bei großen Szenen kann das eine erhebliche Geschwindigkeitssteigerung mit sich bringen, da nur in etwa die Hälfte aller Flächen gezeichnet werden muß.

Wie unterscheidet GL ob Vorder- oder Rückseite einer Fläche sichtbar ist? Kriterium hierfür ist, in welcher Reihenfolge die Vertices definiert sind.

```
void glFrontFace(
    GLenum mode
    );
```

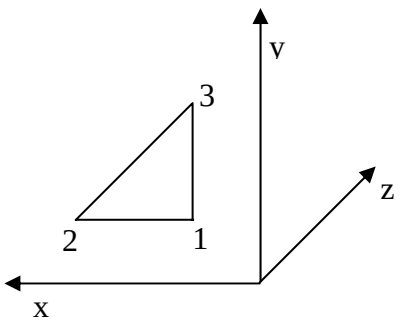
Mit *glFrontFace* teilt man GL mit, welche Seite eines Polygons die Vorderseite ist. Als Parameter für *mode* können ***GL_CW*** oder ***GL_CCW*** angegeben werden. ***GL_CW*** bedeutet, im Uhrzeigersinn, ***GL_CCW*** gegen den Uhrzeigersinn. ***GL_CCW*** ist standardmäßig eingestellt.

Angenommen die Punkte eines Dreiecks sind in der Reihenfolge (0.5, 0.5, 0), (1, 0.5, 0), (0.5, 1, 0) definiert worden. Wenn man von der positiven z-Achse aus auf die Fläche sieht, ergibt sich folgendes Bild:



Wir sehen hier also von vorn auf die Fläche, da die Punkte gegen den Uhrzeigersinn geordnet sind.

Wenn man dagegen von der negativen z-Achse her auf die Fläche schaut, ergibt sich:



Die Punkte sind im Uhrzeigersinn geordnet, das heißt, wir sehen auf die Rückseite der Fläche.

Um GL zu veranlassen, die Rückseiten nicht zu zeichnen, benutzt man

```
glEnable(GL_CULL_FACE);
```

Außerdem kann man mit Hilfe von

```
void glCullFace(
    GLenum mode
    );
```

festlegen ob Vorder- *(mode* = ***GL_FRONT)*** oder Rückseiten ***(GL_BACK)*** entfernt werden. Voreingestell ist ***GL_BACK***.

8.2 Der z-Buffer

Zur Verdeckungsberechnung bedient sich OpenGL des ***depth bufferings***.

Das Funktionsprinzip ist folgendes:

> Neben dem Bildschirmspeicher, in dem zu jedem Punkt (x,y) die Farbe gespeichert wird, existiert ein z-Buffer *(depth buffer)*, in dem zu jedem Punkt dessen z-Koordinate gespeichert wird. Beim Zeichnen eines Punktes, wird zunächst z-Koordinate dieses Punktes mit dem z-Wert, der an der entsprechenden Stelle im z-Buffer steht, verglichen. Nur wenn die z-Koordinate des Punktes kleiner ist, das heißt, der Punkt näher am Betrachter liegt, als der Punkt, der bisher an dieser Stelle stand, wird der neue Punkt gezeichnet. Der Wert im z-Buffer wird durch den z-Wert des Punktes ersetzt, so daß der Punkt wiederum nur von Punkten überschrieben werden kann, die noch näher am Betrachter liegen als er.

Um *depth buffering* einzuschalten, verwendet man *glEnable()*:

```
glEnable(GL_DEPTH_TEST);
```

Um sinnvolle Ergebnisse zu erzielen, sollte der z-Buffer gelöscht werden, bevor gezeichnet wird:

```
glClear(GL_DEPTH_BUFFER_BIT);
```

Der z-Buffer wird standardmäßig mit dem Wert 1.0 gelöscht, es kann aber auch expilizit ein anderer Löschwert angegeben werden:

```
void glClearDepth(
    GLclampd depth
    );
```

Auch die Art des Vergleiches zwischen altem und neuem z-Wert kann beeinflußt werden, und zwar mit

```
void glDepthFunc(
    GLenum func
    );
```

Der Parameter *func* bestimmt, wie der Vergleich durchgeführt wird:

GL_NEVER	Es wird nicht verglichen, es wird niemals ein neuer Wert in den z-Buffer übernommen.
GL_ALWAYS	Es wird ebenfalls kein Vergleich durchgeführt
GL_LESS	Wenn bei neu hinzukommenden Fragmenten $z_{neu}<z_{alt}$ gilt, werden diese im z-Buffer gespeichert. Dies ist die Voreinstellung.
GL_LEQUAL	Gilt bei neuen Fragmenten $z_{neu}<=z_{alt}$, dann werden diese im z-Buffer gespeichert.
GL_EQUAL	Die Pixel werden nur gespeichert wenn deren z-Wert gleich einem schon im z-Buffer stehenden Wert ist ($z_{neu}=z_{alt}$).
GL_GEQUAL	Ein neu hinzukommender Pixel wird nur gespeichert, wenn gilt $z_{neu}<=z_{alt..}$
GL_GREATER	Nur wenn $z_{neu>}z_{alt}$ gilt, wird das betreffende Pixel gespeichert.
GL_NOTEQUAL	Das Pixel wird gespeichert, wenn $z_{neu}<>z_{alt}$ gilt.

Hinweis:

Die Wahl der Perspektive (durch *glFrustum* bzw. *glPerspective*), beeinflußt die Genauigkeit des z-Buffers. Je größer das Verhältnis *zfar* zu *znear* ist, desto uneffektiver wird der z-Buffer bei der Unterscheidung von nahe bei einander liegenden Flächen sein. Es gehen ungefähr

$$ld(\frac{zfar}{znear})$$

bit z-Buffer-Präzision verloren. Es ist also davon abzuraten, kleine Werte für *znear* zu wählen.

9 MATERIALEIGENSCHAFTEN UND BELEUCHTUNG

9.1 Das Farbmodell von OpenGL

Wie bereits aus vorangegangenen Beispielen bekannt sein sollte, wird die aktuelle Farbe durch *glColor* gesetzt. Alle Vertices, die nun definiert werden erhalten diese Farbe. Die aktuelle Farbe gilt, bis zum nächsten Aufruf von *glColor*.

Von *glColor* gibt es wie von fast allen GL Funktionen verschiedene Varianten. Zu beachten ist, das *glColor* entweder drei (RGB) oder vier (RGBA) Parameter erwartet, also *glColor3f()* oder *glColor4f()*. Die Bedeutung des vierten Parameters (Alpha) wird im Kapitel **Transparenz** erläutert. Für die jeweiligen Farbanteile gelten je nach Parametertyp verschiedene Bereiche. Folgende Übersicht zeigt die jeweiligen Werte maximaler Intensität:

Typ	***Maximale Intensität***
GLfloat	1.0
GLubyte	255
GLushort	65535
GLuint	4294967295
GLbyte	127
GLshort	32767
GLint	2147483647

Die minimale Intensität ist in allen Fällen 0

9.2 Shading

Es ist ohne Weiteres möglich, für jeden Eckpunkt einer Fläche eine andere Farbe anzugeben:

```
glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex2f(-0.5,    -0.5);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex2f(0.5,     -0.5);
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex2f(-0.5,    0.5);
glEnd();
```

Wie dies von GL interpretiert wird, hängt vom **Shading-Modell** ab. OpenGL stellt zwei Modelle zur Verfügung, *flat-shading* und *gouraud-shading*.

Beim flat-shading wird die ganze Fläche mit derselben Farbe gefüllt. Ausschlaggebend ist dabei die Farbe des letzten Vertex der zum jeweiligen Primitivsegment gehört. Einzige Ausnahme ist ***GL_POLYGON***, hier ist der erste Vertex ausschlaggebend (warum auch immer).

Beim gouraud-shading werden die Farbwerte der Eckpunkte über die Fläche interpoliert. Obiges Programmfragment liefert also ein Dreieck mit einem weichen Farbverlauf zwischen den Ecken (rot, grün und blau).

```
void glShadeModel(
    GLenum mode
    );
```

Mit *glShadeModel* wird das Shading-Modell gewählt. *mode* kann die Werte ***GL_FLAT*** (flat-shading) oder ***GL_SMOOTH*** (gouraud-shading) annehmen.

9.3 Transparenz

Die vierte Farbkomponente, der *Alpha*-Wert, wird verwendet, um transparente Flächen zu realisieren. Normalerweise, bedeutet dabei ein Wert von 1.0f eine undurchsichtige Fläche, ein Wert von 0.0f eine unsichtbare Fläche (nicht sehr sinnvoll). Die entsprechende Fragmentoperation, das ***Blending***, ist eine der letzten in der Renderingpipeline. Das äußert sich dadurch, daß der Transparenzeffekt nur zur Geltung kommt, wenn die transparente Fläche **nach** den Flächen gezeichnet wird, die sie verdeckt.

Das Blending wird eingeschaltet mit

```
glEnable(GL_BLEND);
```

Dadurch ergibt sich noch kein unmittelbarer Effekt, die Blendingoperation ist standardmäßig so definiert, daß jede neue Fläche alle darunterliegenden vollständig verdeckt, unabhängig vom Alphawert. Die Blendingoperation wird folgendermaßen durchgeführt (komponentenweise für R, G, B und A):

$$Farbe = Farbe^{neu} * S^{neu} + Farbe^{alt} * S^{alt}$$

Neu bezeichnet dabei das Fragment, das gezeichnet wird, Alt die Farbkomponenten des Bereiches, auf den das neue Fragment gezeichnet wird.

Die Skalierungsfaktoren S_{alt} und S_{neu} können mit der Funktion

```
void glBlendFunc(
    GLenum sneu,
    GLenum salt
    );
```

Als Werte für sneu sind folgende Konstanten möglich:

Konstante	S_{neu}=
GL_ZERO	0
GL_ONE	1
GL_DST_COLOR	$Farbe_{alt}$ (komponentenweise)
GL_ONE_MINUS_DST_COLOR	$Farbe_{alt}$ (komponentenweise)
GL_SRC_ALPHA	$Alpha_{neu}$
GL_ONE_MINUS_SRC_ALPHA	1- $Alpha_{neu}$
GL_DST_ALPHA	$Alpha_{alt}$
GL_ONE_MINUS_DST_ALPHA	1- $Alpha_{alt}$
GL_SRC_ALPHA_SATURATE	$S_{R,G,B}$ = min($Alpha_{neu}$, 1-$Alpha_{alt}$)

Als Werte für *salt* sind folgende Konstanten möglich:

Konstante	S_{alt}=
GL_ZERO	0
GL_ONE	1
GL_SRC_COLOR	$Farbe_{neu}$ (komponentenweise)
GL_ONE_MINUS_SRC_COLOR	1- $Farbe_{neu}$ (komponentenweise)
Konstante	S_{alt}=
GL_SRC_ALPHA	$Alpha_{neu}$
GL_ONE_MINUS_SRC_ALPHA	1- $Alpha_{neu}$
GL_DST_ALPHA	$Alpha_{alt}$
GL_ONE_MINUS_DST_ALPHA	1- $Alpha_{alt}$

Dadurch lassen sich sehr viele sinnvolle und sinnlose Kombinationen realisieren, die teilweise über das landläufige Verständnis von Transparenz hinausgehen.

Beispiel 1:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glBegin(GL_TRIANGLES);
    glColor4f(1.0f, 0.0f, 0.0f, 1.0);
    glVertex2f(-0.5,    -0.5);
```

```
        glVertex2f(0.5,     -0.5);
        glVertex2f(-0.5,    0.5);
        glColor4f(0.0f, 0.0f, 1.0f, 0.5);
        glVertex2f(0.5,     0.5);
        glVertex2f(0,  -0.5);
        glVertex2f(-0.5,    1);
    glEnd();
    glFlush();
```

Es werden zwei Dreiecke gezeichnet, ein rotes und darüber ein halbdurchlässiges blaues. Als Blendingfunktion wird $\mathbf{F = F_{neu} * Alpha_{neu} + F_{alt} * (1 - Alpha_{neu})}$ benutzt, womit eine „normale“ Transparenz bewirkt wird.

Beispiel 2:

```
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);

    glBegin(GL_TRIANGLES);
        glColor4f(1.0f, 0.0f, 0.0f, 1.0);
        glVertex2f(-0.5,    0.5);
        glVertex2f(0.5,     0.5);
        glVertex2f(0,  -0.5);
        glColor4f(0.0f, 1.0f, 0.0f, 1.0);
        glVertex2f(-0.5,    0.5);
        glVertex2f(0.5,     0);
        glVertex2f(-0.5,    -0.5);
        glColor4f(0.0f, 0.0f, 1.0f, 1.0);
        glVertex2f(0.5,     0.5);
        glVertex2f(-0.5,    0);
        glVertex2f(0.5,     -0.5);
    glEnd();
    glFlush();
```

Es werden drei Dreiecke gezeichnet, ein rotes, ein blaues und ein grünes. Als Blendingfunktion wird $\mathbf{F = F_{neu} * Alpha_{neu} + F_{alt} * 1}$ benutzt. Das Ergebnis ist eine additive Farbmischung. An der Stelle, wo sich alle Dreiecke überschneiden, ist die resultierende Farbe weiß.

9.4 Materialeigenschaften

Für eine realistischere Darstellung ist es möglich, das Aussehen von Objekten anstelle von Farben mit Materialeigenschaften zu beschreiben. Das Material wird dabei von drei Komponenten beschrieben, **diffus, ambient** und **spekular**. Die Komponenten geben an, wie stark der jeweilige Anteil des eintreffenden Lichtes reflektiert wird. Die Komponenten werden jeweil als RGBA Werte spezifiziert, wodurch angegeben wird, welcher Teil des Lichts reflektiert wird. Enthält das Licht diesen Teil nicht, wird auch nichts reflektiert, eine blaue Fläche erscheint also bei rotem Licht schwarz.

Diffuse Reflexion

Das eintreffende Licht, wird gleichmäßig in alle Richtungen reflektiert, das heißt, je nach Winkel und Entfernung zur Lichtquelle erhält die betreffende Fläche eine Farbe, unabhängig vom Betrachterstandpunkt.

Ambiente Reflexion

Die Reflexion des umgebenden Streulichts. Mit ambientem Licht wird die Erscheinung der realen Welt nachgebildet, das auf jeden Körper nicht nur das Licht direkt aus Lichtquellen trifft, sondern auch das Licht, das von anderen Körpern (diffus) reflektiert wird. Die ambiente Komponente ist meist gleich der diffusen. Das ambiente Licht kommt aus jeder Richtung, und ist somit an jeder Stelle des Körpers gleich.

Spekulare Reflexion

Mit Hilfe dieser Reflexionsart werden Highlights auf die Körper gesetzt. Dies ist sozusagen die „richtige“ Reflexion. Hier spielt der Winkel zwischen Lichtquelle, Objekt und Betrachter eine Rolle.

9.4.1 Zuweisen von Materialeigenschaften

Um die aktuellen Materialeigenschaften zu setzen bedient man sich

```
void glMaterialfv(
    GLenum face,
    GLenum pname,
    const GLfloat* params
    );
```

Materialeigenschaften können für Vorder- und Rückseite eines Objektes getrennt definiert werden. *face* bezeichnet die Seite, auf die sich die Materialdefinition beziehen soll, möglich sind ***GL_FRONT, GL_BACK*** oder ***GL_FRONT_AND_BACK***.

pname bezeichnet den Materialparameter, der geändert werden soll. Möglich sind folgende Konstanten:

Konstante	***Bedeutung***
GL_AMBIENT	RGBA für die ambiente Reflexion soll geändert werden. Standardmäßig ist hierfür (0.2, 0.2, 0.2, 1.0) eigestellt.
GL_DIFFUSE	RGBA für die diffuse Reflexion soll geändert werden. Standardmäßig ist hierfür (0.8, 0.8, 0.8, 1.0) eigestellt.
GL_AMBIENT_AND_DIFFUSE	RGBA für die diffuse und ambiente Reflexion soll geändert werden (auf den gleichen Wert).
GL_SPECULAR	RGBA für die spekulare Reflexion soll geändert werden. Standardmäßig ist hierfür (0.0, 0.0, 0.0, 1.0) eigestellt.
GL_EMISSION	Ein Sonderfall. Eine Licht (von der Farbe RGBA) emittierende Fläche wird simuliert, das heißt, die Fläche sieht so aus, als würde sie leuchten. Das heißt aber nicht, daß die Fläche wirklich Licht aussendet, welches andere Objekte beleuchtet!

Der letzte Parameter **params* ist ein Zeiger auf ein Array, welches die RGBA-Werte enthält. Man geht also ungefähr folgendermaßen vor:

```
GLfloat ambi[4] = {1,1,1,1};
glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT,  ambi );
```

Für die spekulare Reflexion ist es außerdem möglich, den *spekularen Exponenten* einzustellen, der Für die Größe und die Helligkeit des Highlights verantwortlich ist.

```
glMaterialf(
        GLenum face,
        GL_SHININESS,
        GLfloat exponent
        );
```

exponent kann im Bereich von 0.0 bis 128.0 liegen. Je kleiner der Wert ist, desto größer wird das Highlight. Voreingestellt ist 0.0.

Wie alle Attribute der GL gelten auch die Materialeigenschaften solange, bis sie neu definiert werden. Da es sinnlos wäre, für jede Komponente einen anderen Alphawert zu benutzen wird der der diffusen Komponente verwendet.

9.5 Lichtquellen

Das Beleuchtungsmodell der OpenGL unterscheidet vier verschiedene Arten von Lichtquellen:

Ambientes Licht:

Dies ist das umgebende Streulicht, das von allen Seiten gleichmäßig auf die Objekte fällt. Es ist daher auch sinnlos, mehrere ambiente Lichtquellen zu definieren, da sie sich sowieso an allen Orten gleichmäßig überlagern würden. In der GL gibt es also nur **das** ambiente Licht.

Die Intensität des von einem Objekt zurückgeworfenen ambienten Lichts wird berechnet

$$I = I_a * k_a$$

I_a ist die Intensität des ambienten Lichts, k_a ist der ambiente Reflexionskoeffizient des Objektes.

Gerichtetes Licht:

Ein Lichtstrom, der aus einer Richtung kommt. Die Lichtstrahlen sind parallel. Beispiel hierfür in der realen Welt wäre das Sonnenlicht.

Positioniertes Licht:

Diese Art von Licht stammt aus Punktlichtquellen. Eine Punktlichtquelle strahlt gleichmäßig in alle Richtungen. Eine Glühlampe ist ein Beispiel für eine solche Lichtquelle.

Spotlights:

Punktlichtquellen, die auf einen bestimmten Öffnungswinkel beschränkt sind. Beispiel: Scheinwerfer.

Gerichtetes und positioniertes Licht sowie Spotlights werden zur Berechnung der diffusen und spekularen Reflexion herangezogen. Die Intensität des von einem Objekt reflektierten diffusen Lichts wird berechnet

$$I = I_d * k_d * \cos\alpha$$

I_d ist die Intensität der Lichtquelle, k_d der diffuse Reflexionskoeffizient der Fläche, und α der Winkel zwischen dem Einfallsvektor des Lichts und dem Normalenvektor der Fläche.

Bei der spekularen Reflexion geht nun auch der Betrachterstandpunkt mit ein. Die Berechnung erfolgt folgendermaßen:

$$I = I_d * k_s * \cos^n \phi$$

I_d ist die Intensität der Lichtquelle, k_s der spekulare Reflexionskoeffizient der Fläche und n der spekulare Koeffizient.

Die Gesamtintensität, also die Farbe mit der der Punkt entgültig dargestellt wird, ergibt sich aus der Addition der Einzelintensitäten. Wie bereits angedeutet, spielen bei der Beleuchtungsrechnung auch die Normalenvektoren der Flächen eine Rolle. Die Funktion zu Definition eines Normalenvektors ist

```
void glNormal3f(
        GLfloat nx,
        GLfloat ny,
        GLfloat nz
        );
```

Auch Normalenvektoren gelten solange, bis sie neu definiert werden. Ein Beispiel für die Definition einer Fläche mit Normalenvektor wäre

```
glBegin(GL_POLYGON);
        glNormal3f(0, -1, 0);
        glColor3f(0.5f, 0.5f, 0.5f);
        glVertex3f(-0.3,  -0.3,  0.3);
        glVertex3f(0.3,  -0.3,  0.3);
        glVertex3f(0.3,  -0.3,  -0.3);
        glVertex3f(-0.3,  -0.3,  -0.3);
glEnd();
```

Die Normalenvektoren können auch für jeden Vertex neu definiert werden. Auch die Beleuchtungsrechnung bezieht sich auf die einzelnen Vertices, so daß mit verschiedenen Normalenvektoren für jeden Vertex der Effekt einer gekrümmten Fläche erzielbar ist. Zeichnet man beispielsweise eine mit Flächen angenäherte Kugel, so erhält man einen „runderen“ Effekt, wenn als Richtung für den Normalenvektor eines jeden Vertex die Richtung **(Kugelzentrum→Vertex)** gewählt wird.

Je nachdem, welches *Shading-Modell* gewählt wird, läuft die Beleuchtungsrechnung anders ab. Ist ***GL_FLAT*** aktiv, so wird die Beleuchtungsrechnung nur für den letzten Vertex eines Segmentes durchgeführt, und danach dem gesamten Segment zugewiesen. Ist ***GL_SMOOTH*** aktiviert, so erfolgt die Berechnung für jeden Vertex, und die resultierenden Farbwerte werden über die Segmentfläche interpoliert. ***GL_SMOOTH*** liefert hier also die schöneren Ergebnisse, dagegen ist ***GL_FLAT*** schneller.

9.5.1 Definition von Lichtquellen

Soviel zur Theorie. Kommen wir nun zu den GL-Funktionen, mit deren Hilfe sich Lichtquellen erzeugen lassen. Zuerst muß die Beleuchtungsrechnung aktiviert werden:

*glEnable(**GL_LIGHTING**);*

Ziehen wir nun wieder das gute alte **Würfelbeispiel** heran:

```
void CMinGLView::DrawWithGL()
{
      glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
      glClear(GL_COLOR_BUFFER_BIT);
      glEnable(GL_DEPTH_TEST);
      glClear(GL_DEPTH_BUFFER_BIT);

      GLfloat ambi[4] = {0.2, 0.2, 0.9, 1};    //ambiente Reflexion
      GLfloat diff[4] = {0.2, 0.2, 0.9, 1};    //diffuse Reflexion
      GLfloat spek[4] = {1.0, 1.0, 1.0, 1};    //spekulare Reflexion

      glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,  ambi);
      glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,  diff);
      glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, spek);
      glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 50.0);

      //Würfel definieren
      GLuint wuerfel = glGenLists(1);
      glNewList(wuerfel,GL_COMPILE);
      //Unterseite
      glBegin(GL_POLYGON);
            glNormal3f(0,  -1,   0);
            glVertex3f(-0.3,  -0.3, 0.3);
            glVertex3f(0.3,   -0.3, 0.3);
            glVertex3f(0.3,   -0.3, -0.3);
            glVertex3f(-0.3,  -0.3, -0.3);
      glEnd();
      //Oberseite
      glBegin(GL_POLYGON);
            glNormal3f(0,   1,   0);
            glVertex3f(-0.3,  0.3,  0.3);
            glVertex3f(-0.3,  0.3,  -0.3);
            glVertex3f(0.3,   0.3,  -0.3);
            glVertex3f(0.3,   0.3,  0.3);
      glEnd();
      //Vorderseite
      glBegin(GL_POLYGON);
            glNormal3f(0,   0,   1);
            glVertex3f(-0.3,  -0.3, 0.3);
            glVertex3f(-0.3,  0.3,  0.3);
            glVertex3f(0.3,   0.3,  0.3);
            glVertex3f(0.3,   -0.3, 0.3);
      glEnd();
      //Hinterseite
      glBegin(GL_POLYGON);
            glNormal3f(0,   0,  -1);
            glVertex3f(0.3,   -0.3, -0.3);
            glVertex3f(0.3,   0.3,  -0.3);
            glVertex3f(-0.3,  0.3,  -0.3);
            glVertex3f(-0.3,  -0.3, -0.3);
```

```
      glEnd();
      //linke Seite
      glBegin(GL_POLYGON);
            glNormal3f(-1,   0,   0);
            glVertex3f(-0.3,  -0.3, -0.3);
            glVertex3f(-0.3,  0.3,  -0.3);
            glVertex3f(-0.3,  0.3,  0.3);
            glVertex3f(-0.3,  -0.3, 0.3);
      glEnd();
      //rechte Seite
      glBegin(GL_POLYGON);
            glNormal3f(1,   0,   0);
            glVertex3f(0.3,   -0.3, 0.3);
            glVertex3f(0.3,   0.3,  0.3);
            glVertex3f(0.3,   0.3,  -0.3);
            glVertex3f(0.3,   -0.3, -0.3);
      glEnd();
      glEndList();

      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glFrustum(-1,1,-0.75,0.75,2,4);
      gluLookAt(2,2,2, 0,0,0, -1,1,-1);

      glEnable(GL_LIGHTING);

      glCallList(wuerfel);
      glFlush();
}
```

Das Ergebnis ist nicht besonders beeindruckend, da die Beleuchtung zwar angeschaltet, aber noch keine Lichtquellen definiert wurden. Die Tatsache, daß der Würfel trotzdem zu sehen ist, ist dem Umstand zu verdanken, daß eine gewisse ambiente Helligkeit (0.2,0.2,0.2,1) voreingestellt ist.

Diese ambiente Helligkeit kann mittels

void glLightModelfv(

***GL_LIGHT_MODEL_AMBIENT**,*

*const GLfloat *params*

);

geändert werden. In **params* wird ein Zeiger auf ein Array mit den RGBA Werten für die ambiente Beleuchtung übergeben.

Jede OpenGL - Implementation stellt mindestens 8 Lichtquellen (numeriert von 0 bis 7) zur Verfügung. Diese lassen sich einzeln zuschalten, positionieren und mit Attributen versehen. Lichtquelle 0 wird eingeschaltet mit

*glEnable(**GL_LIGHT0**);*

Eine Lichtquelle wird positioniert mit

void glLightfv(
GLenum light,
***GL_POSITION**,*
*const GLfloat *params*
);

light spezifiziert die Lichtquelle und kann die Konstanten ***GL_LIGHT0*** bis ***GL_LIGHT7*** annehmen. In **params* wird ein Zeiger auf ein Array mit den Werten X, Y, Z und W erwartet. Die W-Koordinate hat dabei eine besondere Bedeutung: Wenn W=0.0 ist, werden X, Y und Z als Richtungsvektor interpretiert, wir haben es dann mit gerichtetem, parallelen Licht zu tun. Die Lichtquelle wird als unendlich weit entfernt angenommen. Ist W≠0.0, so werden X, Y und Z als Position einer Punktlichtquelle interpretiert.

Die Intensität der verschiedenen Lichtanteile wird mit

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

definiert, wobei *pname* die Werte ***GL_DIFFUSE*** oder ***GL_SPECULAR*** annimmt. In **params* wird ein Zeiger auf ein Array mit den RGBA-Werten für den jeweiligen Lichtanteil übergeben.

Beispiel:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glShadeModel(GL_SMOOTH);

glMatrixMode(GL_MODELVIEW);
GLfloat position0[4] = {-2,2,0,1};
GLfloat color0[4] = {1,1,1,1};
glLightfv(GL_LIGHT0, GL_POSITION, position0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, color0);
glLightfv(GL_LIGHT0, GL_SPECULAR, color0);

GLfloat ambi[4] = {0.2, 0.2, 0.9, 1}; //ambiente Reflexion
GLfloat diff[4] = {0.2, 0.2, 0.9, 1}; //diffuse Reflexion
GLfloat spek[4] = {1.0, 1.0, 1.0, 1}; //spekulare Reflexion
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,  ambi);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,  diff);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, spek);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 100.0);

GLUquadricObj *kugel = gluNewQuadric();
gluSphere(kugel,0.5,100,100);

glFlush();
```

Es wird eine Kugel dargestellt, die von einer weißen Punktlichtquelle (Position (-2,2,0)) beleuchtet wird. Der Unterschied zwischen diffuser und spekularer Beleuchtung ist hier gut zu sehen.

Hinweis:

Um die Berechnung von Highlights realistischer zu gestalten, kann

```
void glLightModelf(
    GL_LIGHT_MODEL_LOCAL_VIEWER,
    GLfloat params
    );
```

verwendet werden. Wenn *params* auf 0 gesetzt wird (was auch standardmäßig so eingestellt ist), dann wird zur Berechnung der spekularen Reflektion die Richtung der negativen z-Achse verwendet, unabhängig von der Position des Vertex im Betrachterkoordinatensystem. Ist *params* von 0 verschieden, so werden spekulare Reflexionen vom Ursprung des Betrachterkoordinatensystems (=Betrachterstandpunkt) berechnet. Das bringt einen höheren Rechenaufwand mit sich. Es sollte von Fall zu Fall entschieden werden, ob der Realitätsgewinn den Zeitverlust aufwiegt.

9.5.2 Spotlights

Ein Spotlight wird als Erweiterung einer Punktlichtquelle gesehen. Zusätzlich zu den bisher eingeführten Parametern müssen jetzt nur noch Richtung und Öffnungswinkel definiert werden.

Mit

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

wird die Richtung definiert, in die der Spot leuchtet. In **params* wird ein Zeiger auf ein Array mit den Werten X, Y und Z übergeben, die einen Richtungsvektor definieren. Mit

```
void glLightf(
    GLenum light,
    GL_SPOT_CUTOFF,
    GLfloat param
    );
```

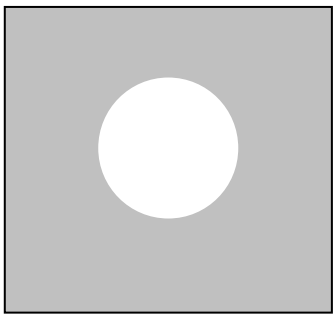
wird der Öffnungswinkel des Spots festgelegt. *Param* gibt den Öffnungswinkel in Grad an.

Beispiel:

Die Beleuchtungsdefinition des vorigen Beispiels wird um

```
GLfloat direction0[3] = {2,-2,0};
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, direction0);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 8.0);
```

erweitert. Hier ist ein unschöner Nebeneffekt des Gouraud-Shadings zu sehen. Der Rand des Spotlights wirkt ausgefranzt. Das rührt daher, daß die Beleuchtungsrechnung nur für die Eckpunkte durchgeführt wird. Der Extremfall wäre folgendes:



Der Spot leuchtet in die Mitte einer Fläche. Da aber kein Eckpunkt vom Spot erfaßt wird, wird die ganze Fläche dunkel gezeichnet. Dem kann man entgegenwirken, indem man große Flächen in kleinere zerlegt. Je kleiner die Flächen sind, desto besser sieht das Ergebnis aus, desto langsamer wird das Ganze aber auch.

9.5.3 Dämpfung

In der Realität erscheinen Objekte mit zunehmender dunkler, die Intensität des Lichts wird gedämpft. Auch dies kann von GL nachgebildet werden. Der Dämpfungsfaktor setzt sich aus

$$k = \frac{1}{k_{konst} + k_{linear} * d + k_{quadr} * d^2}$$

konstanter, linearer und quadratischer Dämpfung zusammen:

Gesetzt werden die Dämpfungsfaktoren durch

```
void glLightfv(
        GLenum light,
        GLenum pname,
        GLfloat param
        );
```

Der Parameter *pname* kann dabei die Werte ***GL_CONSTANT_ATTENUATION***, ***GL_LINEAR_ATTENUATION*** oder ***GL_QUADRATIC_ATTENUATION*** annehmen. *Param* ist der jeweils zugewiesene Wert. Voreingestellt sind (1,0,0), was gleichbedeutend ist mit „gar keine Dämpfung“.

Bei Spots existiert zusätzlich die Möglichkeit, die Intensität des Lichts zu den Rändern hin abnehmen zu lassen. In diesem Fall nimmt *pname* den Wert ***GL_SPOT_EXPONENT*** an. Standardmäßig ist 0.0 eingestellt, was gleiche Intensität im gesamten Kegel bedeutet.

Beispiel:

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
glClear(GL_DEPTH_BUFFER_BIT);

GLfloat ambi[4] = {0.2, 0.2, 0.2, 1};
GLfloat diff[4] = {0.2, 0.2, 0.9, 1};
GLfloat spek[4] = {1.0, 1.0, 1.0, 1};
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,  ambi);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,  diff);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, spek);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 100.0);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1,1,-0.75,0.75,2,4);
gluLookAt(0,0,2, 0,0,0, 0,1,0);

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glMatrixMode(GL_MODELVIEW);
GLfloat position0[4] = {0.3,0.3,0.1,1};
GLfloat color0[4] = {1,1,1,1};
glLightfv(GL_LIGHT0, GL_POSITION, position0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, color0);
glLightfv(GL_LIGHT0, GL_SPECULAR, color0);
glLightf(GL_LIGHT0,GL_LINEAR_ATTENUATION, 1.5);
glLightf(GL_LIGHT0,GL_QUADRATIC_ATTENUATION, 0.5);

glShadeModel(GL_SMOOTH);

GLUquadricObj *disk = gluNewQuadric();
gluDisk(disk,0.1,0.7,100,100);

glFlush();
```

10 TEXTUREN UND OPENGL

10.1 Was ist eine Textur ?

Unter einer Textur versteht man eine bildliche Nachbildung eines komplexeren geometrischen Objektes. Dies ist in den meisten Fällen eine 2-dimensionale Fläche von Bildpunkten, oder auch Texeln. Diese wird dann durch das sogenannte Texture Mapping auf ein geometrisches Modell, z.B. eine Fläche „geklebt“. Dadurch entsteht für den Betrachter der Eindruck, dass die gerenderte Szene aus vielen kleinen Einzelflächen zusammengesetzt sei. Eine Textur stellt also eine komfortable Möglichkeit dar, der gerenderten Szene eine gewisse grafische Komplexität zu verleihen. Texturen werden immer dann angewandt, wenn es darum geht, durch sich ständig wiederholende Oberflächen Objekten eine gewisse Struktur zu verleihen.

Großer Beliebtheit erfreuen sich Texturen bei 3D-Walk-Trough-Spielen wie z.B. Quake oder Unreal. Die Hardware von 3D-Beschleunigerkarten ist hauptsächlich auf das Texture Mapping ausgelegt und bietet hierfür eine recht beachtliche Leistung. Genug der Vorrede, beginnen wir mit der Praxis in OpenGL.

10.2 Texture Mapping mit OpenGL

In OpenGL besteht die Möglichkeit, Texturen eindimensional, zweidimensional oder mehrdimensional zu übergeben. Dieses Kapitel beschäftigt sich ausschließlich mit dem Texture Mapping basierend auf zweidimensionalen Texturen. Die Autoren halten dies für günstig und einleuchtender, da eine zweidimensionale Textur als ein digitalisiertes Bild verstanden werden kann.

Eine Textur besteht, wie bereits erwähnt aus einzelnen Bildpunkten, den sogenannten Texels (**Tex**ture **El**ements). Diese wird nun durch das Texture Mapping auf das grafische Primitiv gebracht. Dabei gibt es folgende Sonderfälle:

- Viele Texel werden auf einen Pixel abgebildet, dann muß eine Mittelung der Texelwerte stattfinden.

- Ein Texel wird auf mehrere Pixel abgebildet. Hierbei muß insbesondere an den Kanten zwischen den Texeln eine Mittelung berechnet werden.

Weiterhin gelten folgende Grundregeln für das Texture Mapping:

- Die Textur muß „ortstabil“ mit dem Objekt verbunden sein. Bewegt sich das Objekt, dann bewegt sich auch seine Textur. Daraus folgt, daß die Texturkoordinaten nicht den Transformationen unterliegen, denen die Objekte unterliegen. Sie werden nur vor der Verbindung mit dem Objekt transformiert.

- Komplexere geometrische Objekte, die aus mehreren Dreiecken bestehen, sollten gemeinsame Texturkoordinaten an den Ecken haben und die Textur sollte das Objekt sinnvoll überdecken.

- Wenn Texturen nicht während des Einsatzes dynamisch neu erzeugt werden, sind sie endlich. Für den Fall, das Texel in einem Breich auf dem Objekt benötigt werden, der nicht innerhalb dieses endlichen Bereiches liegt muß Vorsorge getroffen werden. Dafür gibt die folgenden zwei Möglichkeiten:
  - Die Textur wird zyklisch wiederholt. Dabei ist insbesondere auf den Übergang von einem wiederholten Teil zu dem nächsten zu achten, weil hier die Kanten an diesen Stellen hervortreten können.
  - Die Textur wird nicht wiederholt. Texturierte Bereiche, deren Texturkoordinaten ungültig sind, bleiben untexturiert.

- Eine Textur kann eine „Border“, oder auch Rand besitzen. Ist dies der Fall, verbreitert und verlängert sich die Textur um jeweils 2.

10.3 Wir definieren uns eine Textur

Nun wollen wir endlich zum praktischen Teil des Texture Mappings in OpenGL kommen. Da wir uns eingangs auf zweidimensionale Texturen geeinigt haben, wollen wir uns die Texturdefinition anhand der Funktion

```
glTexImage2D(
        GLenum target,
        GLint level,
        GLint components,
        GLsizei width,
        GLsizei height,
        GLint border,
        GLenum format,
        GLenum type,
        const GLvoid *pixels
        );
```

ansehen. Für die Konstante, die für *target* erwartet wird, wird ***GL_TEXTURE_2D*** gesetzt. Hiermit wird der OpenGL mitgeteilt, dass es sich im folgenden um eine zweidimensionale Textur handelt. Bei der Definition einer Textur in der GL wird immer davon ausgegangen, daß die Texturen in Texturebenen übergeben werden. Als Erläuterung hierfür ist zu bemerken, dass OpenGL eine spezielle Technik, die als **Mipmapping** bezeichnet wird, beherrscht. Mipmapping besagt, dass Texturen „gestaffelt“ im Speicher abgelegt werden.

Die Grundidee hierfür ist, dass der Detailreichtum mit der Entfernung von der Textur abnimmt. Die Grundtextur mit den ursprünglichen Abmaßen bekommt das Level „0“. Jede weitere Textur nimmt jeweils um die Hälfte in Breite wie in Höhe ab und bekommt jeweils das nächstgrößere Level. Da

diese Mipmaps im Normalfall vorher berechnet werden, ist dies eine Kosten sparende Variante, die Qualität von Texturen und ihren visuellen Eindruck zu verbessern. In unserem Fall ist also für *level* 0 einzusetzen. Logischerweise sind Werte kleiner 0 sowie Werte größer ***ld GL_MAX_TEXTURE_SIZE*** (die maximale Texturgröße) nicht zulässig.

components nimmt einen der Werte 1,2,3 oder 4, was davon abhängig ist, wieviele Farbkomponenten die Textur hat:

Wert für components	***Bedeutung***
1	Die Textur hat nur den Rotanteil.
2	Die Textur hat den Rot- und den Alphaanteil.
3	Die Textur hat Rot-, Grün- und Blauanteil.
4	Die Textur hat Rot-, Grün-, Blau- und Alphaanteil.

Tabelle: components

Je nachdem, ob eine Textur einen Rand besitzt wird *border*=1 oder *border*=0 gesetzt. Hat eine Textur einen Rand, dann muß *width* einen Wert $width=2^n+2$ annehmen, andernfalls $width=2^n$. Gleiches gilt für *height*. *format* und *type* definieren, wie der Inhalt des durch *pixels* übergebenen Speicherbereiches zu interpretieren ist. Die für *format* zulässigen Konstanten sind in der nachfolgenden Tabelle zusammengestellt.

Wert für format	***Bedeutung***
GL_COLOR_INDEX	Jeder Eintrag ist ein Einzelwert, ein Farbindex.
GL_RED	Jeder Eintrag ist eine einzelne Rotkomponente. Dieses wird intern als RGBA- Tupel dargestellt. Dabei wird für den Grün-und Blauanteil 0.0f und 1.0f für den Alphanteil gesetzt.
GL_GREEN	analog zu ***GL_RED***
GL_BLUE	analog zu ***GL_RED***
GL_ALPHA	Jeder Eintrag ist eine einzelne Alphakomponente. Dieses wird intern als RGBA- Tupel dargestellt. Dabei wird für den Grün-und Blau- und Rotanteil 0.0f gesetzt.
GL_RGB	Jeder Eintrag ist ein RGB-Tripel. Dieses wird intern in ein RGBA-Tupel überführt, wobei für die Alphakomponente 1.0f gesetzt wird.
GL_RGBA	Jeder Eintrag ist ein komplettes RGBA-Tupel.
GL_BGR_EXT	Jedes Texel ist eine Gruppe der drei Komponenten Blau, Grün und Rot. ***GL_BGR_EXT*** stellt ein Format zur Verfügung, dass sich mit den Windows **D**evice-**I**ndependent **B**itmaps (DIBs) deckt.
GL_BGRA_EXT	Jedes Texel wird als ein Tupel der vier Komponenten Blau, Grün, Rot und Alpha verstanden. Diese Konventionen decken sich ebenfalls

GL_BGRA_EXT	Jedes Texel ist ein Tupel der vier Komponenten Blau, Grün, Rot und Alpha. Dieses Format ist ebenfalls DIB-kompatibel.
GL_LUMINANCE	Jeder Eintrag ist ein luminance- Wert. Der Wert wird in das Floating-Point-Format überführt und danach auf die Anteile Rot, Grün und Blau verteilt. Der Aplhawert ist 1.0f.
GL_LUMINANCE_ALPHA	Jeder Eintrag ist ein luminance-Alpha-Wertpaar.

Tabelle: Formatangaben in Texturdefinitionen

Die folgende Tabelle zeigt die *type* zulässigen Konstanten, also die Typen der übergebenen Texturwerte:

Wert für type	***Bedeutung***
GL_UNSIGNED_BYTE	Die übergebenen Texturwerte werden als 8 Bit Integer ohne Vorzeichen interpretiert.
GL_BYTE	Die übergebenen Texturwerte werden als 8 Bit Integer mit Vorzeichen interpretiert.
GL_BITMAP	Die Werte entsprechen GL_UNSIGNED_BYTE aber haben ein vorzeichen-signifikantes unterstes Bit.
GL_UNSIGNED_SHORT	Die übergebenen Texturwerte werden als 16 Bit Integer ohne Vorzeichen interpretiert.
GL_SHORT	Die übergebenen Texturwerte werden als 16 Bit Integer mit Vorzeichen interpretiert.
GL_UNSIGNED_INT	Die übergebenen Texturwerte werden als 32 Bit Integer ohne Vorzeichen interpretiert.
GL_INT	Die übergebenen Texturwerte werden als 32 Bit Integer mit Vorzeichen interpretiert.
GL_FLOAT	Die übergebenen Texturwerte werden als 32 Bit Fließkomma– Werte interpretiert.

Tabelle: type

Es sei an dieser Stelle nochmals darauf hingewiesen, dass Texturen in OpenGL keineswegs quadratisch sein müssen. Die einzige Bedingung, die für gelten muss ist:

$$breite = 2^m + 2*border$$

$$hoehe = 2^n + 2*border \quad (mit\ border=0\ oder\ border=1).$$

Da eine Textur für mehrere Flächen auf Dauer langweilig wird, sollte es eine Möglichkeit geben, verschiedenen Flächen (oder sogar ein und derselben Fläche) verschiedene Texturen zuweisen zu

können. Dies ist ohne weiteres möglich, indem die verschiedenen Texturdefinition in Darstellungslisten gelagert werden, z.B.:

```
glNewList(TEXTUR_DEFINITION1);
        // evtl. Definition der einzelnen MipMap - Level
        ...
        glTexImage2D(..,4,...);
        ...
glEndList();
```

Durch einfaches Aufrufen von *glCallList(**TEXTUR_DEFINTION1**)* kann nun unsere oben definierte Textur aktiviert werden.

Im nachfolgenden wird systematisch ein kleines Beispielprogramm erarbeitet, das einen Würfel der Kantenlänge 0.6 texturiert auf dem Bildschirm darstellt. Es wird hier bewußt auf die Anwendung von MipMaps verzichtet. Die nun nachfolgende Routine lädt zunächst eine RBG-Datei im Format 256x256x3 Byte in ein Array.

```
unsigned char RGBTexture[256*256*3];
void make_texture() {
        int datei;char *path;
        path = "dateiname.raw";
        datei = _open(path,_O_RDONLY);
        _read(datei,RGBTexture,256*256*3);
        _close(datei);
}
```

Nachdem dies nun geschehen ist, wollen wir nun die in RGBTexture gespeicherte Textur auf den Würfel mappen. Hierfür müssen aber erst einmal einige Einstellungen im OpenGL-Status vornehmen. Mit

```
glEnable(GL_TEXTURE_2D);
```

teilen wir der OpenGL mit, daß die nun folgenden Texturkommandos von der OpenGL als 2D-Texturkommandos interpretiert werden.

Um nun Einstellungen für die Steuerung der Texturebenen und deren Filterung vornehmen zu können bedienen wir uns der Funktion:

```
glTexParameteri(
        GLenum target,
        GLenum pname,
```

GLint param

);

Diese Funktion ist vom Typ void, da hier keine Rückgabewerte benötigt werden. Für target wird eine der beiden Konstanten ***GL_TEXTURE_1D*** oder ***GL_TEXTURE_2D*** erwartet, je nachdem mit welcher Art von Texturen gearbeitet werden soll. In unserem Fall also ***GL_TEXTURE_2D***.

Für pname sind nun verschiedene Konstanten möglich:

Wert für pname	***Bedeutung***
GL_TEXTURE_MIN_FILTER	Hiermit wird der Filter für die Verkleinerung der Textur gesetzt.
GL_TEXTURE_MAG_FILTER	Hiermit wird der Filter für die Vergrößerung der Textur gesetzt.
GL_TEXTURE_WRAP_S	Setzen der Texturwiederholungseigenschaften.
GL_TEXTURE_WRAP_T	Setzen der Texturwiederholungseigenschaften.
GL_TEXTURE_BORDER_COLOR	Setzen der Farbe des Texturrandes.

Die letzte der in der obigen Tabelle aufgeführten Konstanten, ***GL_TEXTURE_BORDER_COLOR,*** kann nur der vektoriellen Form der Funktion benutzt werden.

Wird nun für *pname* ***GL_TEXTURE_MIN_FILTER*** gesetzt, also der Filter, der bei einer Verkleinerung der Textur aktiv wird, ergeben sich für *param* folgende Möglichkeiten:

Wert für param	***Bedeutung***
GL_NEAREST	Es wird dem Pixel die Farbe des Texels gegeben, dessen transformierter Mittelpunkt dem des Pixels am nächsten liegt.
GL_LINEAR	Es wird dem Pixel die Farbe der vier gemittelten Texel gegeben, die dem transformierten Mittelpunkt des Texels am nächsten liegt.
GL_NEAREST_MIPMAP_NEAREST	Es wird dem Pixel die Farbe des Texels derjenigen Texturebene gegeben, bei der das Verhältnis Texel/Pixel = 1 am besten erfüllt ist.
GL_LINEAR_MIPMAP_NEAREST	Es wird dem Pixel die Farbe der vier gemittelten Texel derjenigen Texturebene gegeben, bei der das Verhältnis Texel/Pixel = 1 am besten erfüllt ist.
GL_NEAREST_MIPMAP_LINEAR	Es wird dem Pixel die linear gewichtete Farbe der nächstliegenden zwei Texturebebenen gegeben, bei denen das Verhältnis Texel/Pixel = 1 am besten erfüllt ist

GL_LINEAR_MIPMAP_LINEAR	Analog zu GL_NEAREST_MIPMAP_LINEAR aber es werden die dem Texel nächstgelegenen vier Texelwerte mit einkalkuliert.

Da für den Vergrößerungsfall der Textur nicht allzuviele Varianten denkbar sind ergeben sich nur die zwei Möglichkeiten, nämlich ***GL_NEAREST*** und ***GL_LINEAR***.

Der nun nachfolgende Code, der einen von allen Seiten texturierten Würfel beschreibt, könnte in die Methode DrawWithGL() eingefügt werden und würde so bei jedem ***WM_PAINT***-Ereignis ausgeführt werden.

```
// begin texture specific
make_texture();
glEnable(GL_TEXTURE_2D);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexImage2D(GL_TEXTURE_2D,0,3,256,256,0,
             GL_RGB,GL_UNSIGNED_BYTE,RGBTexture);
// end texture specific

// define cube
GLuint cube = glGenLists(1);
glNewList(cube,GL_COMPILE);
      //bottom grey
      glBegin(GL_POLYGON);
            glColor4f(0.5f, 0.5f, 0.5f,1.0f);
            glTexCoord2f(0,1);            // texture-coords first
            glVertex3f(-0.3,-0.3,0.3f);   // then vertex-coords
            glTexCoord2f(1,1);
            glVertex3f(0.3,-0.3,0.3);
            glTexCoord2f(1,0);
            glVertex3f(0.3,-0.3,-0.3);
            glTexCoord2f(0,0);
            glVertex3f(-0.3,-0.3,-0.3);
      glEnd();
      //top white
      glBegin(GL_POLYGON);
            glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
            glTexCoord2f(0,1);
            glVertex3f(-0.3,0.3,0.3);
            glTexCoord2f(0,0);
            glVertex3f(-0.3,0.3,-0.3);
            glTexCoord2f(1,0);
            glVertex3f(0.3,0.3,-0.3);
            glTexCoord2f(1,1);
            glVertex3f(0.3,0.3,0.3);
      glEnd();
      //front red
      glBegin(GL_POLYGON);
            glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
            glTexCoord2f(0,0);
            glVertex3f(-0.3,-0.3,0.3);
            glTexCoord2f(0,1);
            glVertex3f(-0.3,0.3,0.3);
            glTexCoord2f(1,1);
            glVertex3f(0.3,0.3,0.3);
            glTexCoord2f(1,0);
```

```
            glVertex3f(0.3,-0.3,0.3);
      glEnd();

      //back blue
      glBegin(GL_POLYGON);
            glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
            glTexCoord2f(1,0);
            glVertex3f(0.3,-0.3,-0.3);
            glTexCoord2f(1,1);
            glVertex3f(0.3,0.3,-0.3);
            glTexCoord2f(0,1);
            glVertex3f(-0.3,0.3,-0.3);
            glTexCoord2f(0,0);
            glVertex3f(-0.3,-0.3,-0.3);
      glEnd();
      //left side green
      glBegin(GL_POLYGON);
            glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
            glTexCoord2f(0,0);
            glVertex3f(-0.3,-0.3,-0.3);
            glTexCoord2f(1,0);
            glVertex3f(-0.3,0.3,-0.3);
            glTexCoord2f(1,1);
            glVertex3f(-0.3,0.3,0.3);
            glTexCoord2f(0,1);
            glVertex3f(-0.3,-0.3,0.3);
      glEnd();
      //right side yellow
      glBegin(GL_POLYGON);
            glColor4f(1.0f, 1.0f, 0.0f, 1.0f);
            glTexCoord2f(0,1);
            glVertex3f(0.3,-0.3,0.3);
            glTexCoord2f(1,1);
            glVertex3f(0.3,0.3,0.3);
            glTexCoord2f(1,0);
            glVertex3f(0.3,0.3,-0.3);
            glTexCoord2f(0,0);
            glVertex3f(0.3,-0.3,-0.3);
      glEnd();
glEndList();

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
glClear(GL_DEPTH_BUFFER_BIT);

glCallList(cube);
glFlush();
```

Bei dem oben aufgeführten Beispiel wird zunächst *make_texture()* aufgerufen und damit die Textur geladen. Danach folgen einige Einstellungen des OpenGL – Texturierungszustandes. Jetzt wird die Textur geladen. Dieser Aufruf bezieht sich auf das Feld, das im Zusammenhang mit der Funktion *make_texture()* erzeugt wurde. Nachdem dies nun geschehen ist, werden in Form einer Darstellungsliste die einzelnen Seiten des Würfels definiert, wobei den Vertices Texturkoordinaten zugewiesen werden. Hierbei fällt auf, daß die Aufrufe von *glTexCoord2f()* stets vor dem Aufruf von *glVertex3f()* erfolgen. Der Grund hierfür ist, daß Texturkoordinaten, wie alle Attribute, solange gelten, bis neue definiert werden. Abschließend wird noch der z-Buffer aktiviert und gelöscht, um eine ordnungsgemäße Funktion der Verdeckungsberechnung zu garantieren.

11 ANIMATIONEN MIT OPENGL

Animationen sollten aus diversen Computerspielen hinreichend bekannt sein und kommen immer dann zum Einsatz, wenn es darum geht eine virtuelle Bewegung auf dem Bildschirm darzustellen. Animationen laufen im Prinzip immer nach dem selben Schema ab:

1. Seite 1 anzeigen; auf Seite 2 im Hintergrund malen
2. Seite 2 anzeigen; Seite 1 löschen und im Hintergrund neu beschreiben
3. Seite 1 anzeigen; Seite 2 löschen und im Hintergrund neu beschreiben ...usw.

Bei Animationen wird grundsätzlich mit 2 (evtl. auch mit 3) Bildschirmseiten gearbeitet. Während die eine angezeigt wird, wird auf der anderen im Hintergrund gezeichnet. Ist das Zeichnen beendet, werden beide getauscht, die eben noch aktive Seite gerät in den Hintergrund und die andere wird aktiv und kommt zur Anzeige. Um in OpenGL mit zwei Bildschirmseiten arbeiten zu können muß diese Einstellung beim **PIXELFORMATDESCRIPTOR** vor der Initialisierung der OpenGL getätigt werden.

```
PIXELFORMATDESCRIPTOR pfd  = {
    sizeof(PIXELFORMATDESCRIPTOR),      // Structure size.
    1,                                  // Structure version number.
    PFD_DRAW_TO_WINDOW |                // Property flags.
    PFD_SUPPORT_OPENGL |
    PFD_DOUBLEBUFFER,
    PFD_TYPE_RGBA,
    24,                                 // 24-bit color.
    0, 0, 0, 0, 0, 0,                   // Not concerned with these.
    0, 0, 0, 0, 0, 0, 0,                // No alpha or accum buffer.
    32,                                 // 32-bit depth buffer.
    0, 0,                               // No stencil or aux buffer.
    PFD_MAIN_PLANE,                     // Main layer type.
    0,                                  // Reserved.
    0, 0, 0                             // Unsupported.
    };
```

Wie wir sehen wird bei den property – Flags das Flag ***PFD_DOUBLEBUFFER*** logisch hinzugeodert. Damit wird die Arbeit mit zwei Bildschirmseiten aktiviert. Damit nun zwischen der aktiven und der inaktiven Bildschirmseite hin- und hergeschaltet werden kann, bietet Windows die Funktion

```
SwapBuffers( HDC hdc );
```

Die Variable *hdc* ist vom Typ *HDC* (Handle to Device Context). Die Funktion erwartet also den aktuellen Device Context, bei dem die Buffer getauscht werden sollen. Aber wo bekommen wir diesen her? Da unsere *DrawWithGL()* Methode von der *OnDraw()* – Methode aufgerufen wird, können wir der *DrawWithGL()* – Methode den aktuellen Device Context mit übergeben:

// Diesen Aufruf gegen den alten Aufruf in OnDraw() tauschen

DrawWithGL(pDC->m_hDC);

Jetzt sind natürlich noch die *DrawWithGL()* - Funktionsköpfe in *MinGLView.cpp* und *MinGLView.h* zu ändern

void CMinGLView::DrawWithGL(HDC hdc).

11.1 Eine einfache Animation

Mit dem nun folgenden Programmfragment wird die Rotation eines Objektes auf dem Bildschirm realisiert. Dieses Objekt soll eine Teekanne sein. Was konkret diese Teekanne ist und wie sie implementiert wird ist im Kapitel **Langsam – langsamer – Primitive höherer Ordnung** nachzulesen.

```
double degree=0;
glPolygonMode(GL_FRONT_AND_BACK,GL_POINT);
glEnable(GL_DEPTH_TEST);
glCullFace(GL_FRONT);
glEnable(GL_CULL_FACE);                         //cull frontfaces
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

do {
      degree+=5;                                // increment degree
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      glRotatef(degree,1,1,0);                  // rotate by degree
      glClear(GL_COLOR_BUFFER_BIT);             // background black
      glClear(GL_DEPTH_BUFFER_BIT);
      glColor3f(.3.0,.1,.7);                    // color the teapot
      auxSolidTeapot(.5);
      SwapBuffers(hdc);                         // exchange buffers
} while(degree<360);
```

Durch den Befehl

*glPolygonMode(**GL_FRONT_AND_BACK,GL_POINT**);*

wird der Darstellungsmodus für Rück- wie für die Vorderseite auf den Punktmodus gesetzt. Soll die Teekanne anders dargestellt werden, ist ***GL_POINT*** durch ***GL_LINE*** oder ***GL_FILL*** zu ersetzen. Danach folgt eine Transformation der Modelviewmatrix; das Objekt wird um den sich ändernden Winkel gedreht. Als nächstes werden Bildschirm und z-Buffer gelöscht.

Mit

auxSolidTeapot(1);

wird dann schließlich ein „fester“ Teepott dargestellt. Um diesen Befehl benutzen zu können, muß vorher in der MinGLView.cpp

#include <gl/glaux.h>

eingefügt werden. Außerdem ist noch **Settings...** im Menü **Project zu** wählen und im **Link-**Tab bei **Object/library modules glaux.lib** einzufügen, um auf die **glaux32.dll** zugreifen zu können.

Für denjenigen, der sich nun darüber wundert, daß hier kein Zeichenkommando, wie *glFlush()* vorkommt sei bemerkt, daß *SwapBuffers()* ein *glFlush()* erzwingt.

Daß diese Form der Animation nun nicht die schnellste ist, wurde schon bald von den Vätern der OpenGL erkannt. Deshalb werden durch die GL weitere Möglichkeiten angeboten, um die Darstellung zu beschleunigen. So ist es möglich, der OpenGL mitzuteilen, welcher Bildausschnitt denn nun dargestellt werden soll. Alles was außerhalb dieser sogenannten ScissorBox liegt wird dann nicht dargestellt. Grundvoraussetzung für die ordnungsgemäße Funktionsweise dieser ist der stencil-buffer. Dieser soll aber nicht im Rahmen dieses Lehrbriefes behandelt werden.

12 BÉZIER-KURVEN UND -FLÄCHEN

Die OpenGL kann mit Hilfe von sogenannten *Evaluatoren* gekrümmte Kurven und Oberflächen generieren. Grundsätzlich basiert die Berechnung auf einer Annäherung an eine Menge von vorgegebenen Kontrollpunkten. Physikalisch sind Bézier-Kurven vergleichbar mit einem Stahlband, welches an den Endpunkten eingespannt ist und an weiteren Punkten ausgelenkt wird.

Die *Evaluatoren* generieren dabei aus der spezifizierten Kurve normale Vertex-Daten, auf die dann sämtliche Transformationen angewendet werden können.

12.1 Mathematische Grundlagen

Diese Methode zur Kurvenapproximation ist nach Pierre Bézier benannt, der zwischen 1960 und 1970 für den französischen Autohersteller Renault eine Modellierungshilfe für Autokarosserien entwickelte.

Die dabei verwendeten Bézier-Kurven ***P(t)*** vom Grad n werden über n+1 Stützpunkte P_i in Vektorform durch folgende Gleichung bestimmt:

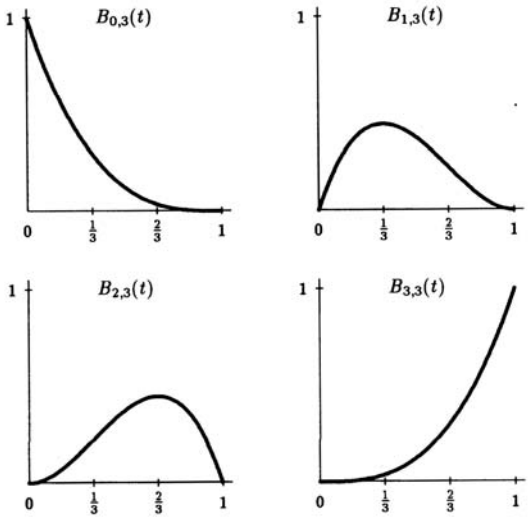
$$P(t) = \sum_{i=0}^{n} B_{i,n}(t) \cdot P_i$$

$$0 \le t \le 1$$

$\mathbf{B}_{i,n}(t)$ sind dabei Bernsteinpolynome der Form

$$B_{i,n}(t) = \frac{n!}{i! \cdot (n-i)!} \cdot t^i \cdot (1-t)^{n-i}$$

Durch diese Polynome wird die Stärke des Einflusses der Stützstellen auf den Kurvenverlauf festgelegt. Folgende Abbildung zeigt beispielhaft die Bernsteinpolynome für den Fall n=3, die den Einfluß der Stützpunkte P_0 bis P_3 auf den Kurvenverlauf angeben.



Wie man sieht, nehmen an der Stelle t=0 alle Polynome außer $\mathbf{B}_{0,3}(t)$ den Wert 0 an. Analog dazu bestimmt an der Stelle t=1 nur P_3 den Kurvenverlauf. P_1 bis P_2 nehmen den größten Einfluß an den Stellen t=1/3 bzw. t=2/3.

Die wichtigsten Eigenschaften von Bézier-Kurven sind:

- Die Bezierkurve liegt innerhalb der konvexen Hülle des Polygons, das durch die Stützpunkte gebildet wird.
- Da für 0<t<1 alle Bernsteinpolynome ungleich Null sind, beeinflußt jeder Stützpunkt die Kurve in diesem Intervall. Lokale Veränderungen sind nicht möglich; die Verschiebung einer Stützstelle verändert die gesamte Kurve.
- Im allgemeinen liegen nur Anfangs und Endpunkt des Polygons aufder Kurve.
- Die Tangenten am Anfangs- und Endpunkt der Kurve stimmen mit dem ersten und letzten Polygonsegment überein. Diese Eigenschaft kann genutzt werden, um Stetigkeit bei Kurvenzügen, die aus mehreren Bézier-Kurven zusammengesetzt sind, zu gewährleisten.

Für Bézier-Flächen gilt analog folgende Definition:

$$S(t,s) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(t) B_j^m(s) P_{ij}$$

Anstelle eines Parameters werden nun zwei, *s* und *t*, verwendet, sowie ein „Netz“ von Stützstellen P_{ij}.

12.2 Bézierkurven

Um eine Bézierkurve zu **definieren**, bedient man sich der Funktion

```
void glMap1f(
        GLenum target,
        GLfloat u1,
        GLfloat u2,
        GLint stride,
        GLint order,
        const GLfloat *points
        );
```

Hiermit definiert man die Berechnung einer eindimensionalen Kurve oder, anders gesagt, den Verlauf von spezifischen Werten anhand von Stützstellen. Der Parameter *target* gibt dabei an, welcher Art die Stützstellen sind, die in **points* übergeben werden. So ist möglich Verläufe zwischen Vertices, Farbwerten, Normalenvektoren oder Texturkoordinaten zu realisieren. Folgende Tabelle zeigt die möglichen Konstanten für *target*, die Bedeutung dieser Konstanten sowie die Art der (internen) Funktionsaufrufe, die generiert werden, wenn die Map ausgewertet wird:

Target	***Art der Stützstellenwerte***	***Funktionsaufruf***
GL_MAP1_VERTEX_3	x, y und z Koordinate	*glVertex3f*
GL_MAP1_VERTEX_4	x, y, z und w Koordinate	*glVertex4f*

GL_MAP1_INDEX	Index aus der Farbtabelle	*glIndex*
GL_MAP1_COLOR_4	Farbwert als RGBA	*glColor4f*
GL_MAP1_NORMAL	Normalenvektor x, y, z	*glNormal3f*
GL_MAP1_TEXTURE_COORD_1	Texturkoordinate s	*glTexCoord1f*
GL_MAP1_TEXTURE_COORD_2	Texturkoordinaten s, t	*glTexCoord2f*
GL_MAP1_TEXTURE_COORD_3	Texturkoordinaten s, t, r	*glTexCoord3f*
GL_MAP1_TEXTURE_COORD_4	Texturkoordinaten s, t, r, q	*glTexCoord4f*

Die Variablen *u1* und *u2* geben Unter- und Obergrenze des Bereiches an, in dem sich die Variable *u* bewegen kann. *u* wird benutzt, wenn später die durch *glMap1f()* definierte Bézier-Kurve ***P****(t)* abgefragt wird. Der Parameter *t* wird dabei wie folgt aus *u* berechnet:

$$t = \left(\frac{u - u_1}{u_2 - u_1} \right)$$

Das heißt *u1* wird auf 0 und *u2* auf 1 gemappt.

In **points* wird ein Zeiger auf ein Array mit den Werten der Stützstellen übergeben. *stride* gibt an, wieviele *floats* zwischen dem Beginn eines Stützpunktes und des nächsten liegen. So ist es möglich in das **points* Array zusätzliche Informationen zwischen den Stützstellen einzubetten. *order* gibt die Ordnung der Kurve, und damit die Anzahl der Stützstellen an.

Der in *glMap1f()* definierte Kurvenverlauf muß anschließend mit *glEnable(target);* aktualisiert werden. *target* kann dabei die oben definierten Werte annehmen.

Die eigentliche Berechnung der aktualisierten Kurve erfolgt mit der Funktion

```
void glEvalCoord1f(
        GLfloat u
        );
```

Der Parameter *u* stellt dabei einen Wert aus dem Bereich zwischen *u1* und *u2* dar. *glEvalCoord* generiert dann intern die Funktionsaufrufe aus obiger Tabelle (für die *glEnable*ten Konstanten).

Beispiel 1:

```
float stuetzpunkte[4*3]      = { -0.9, 0.0, 0, -0.4, 0.9, 0, 0.4, -0.4, 0, 0.9, 0.4, 0};
float stfarbe[3*4]           = { 1.0,  1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.4,  0.8, 0.0, 1.0};

glMap1f(GL_MAP1_VERTEX_3, 0, 1, 3, 4, stuetzpunkte);
glMap1f(GL_MAP1_COLOR_4, 0, 1, 4, 3, stfarbe);
```

```
glEnable(GL_MAP1_VERTEX_3);
glEnable(GL_MAP1_COLOR_4);

glBegin(GL_LINE_STRIP);
    for(i=0; i<31; i++)
        glEvalCoord1f( ((float)i) / 30.0f );
glEnd();

glPointSize(3);
glColor3f(1,1,0);
glBegin(GL_POINTS);
    for(int i=0; i<4; i++)
        glVertex3f( stuetzpunkte[i*3], stuetzpunkte[i*3+1], stuetzpunkte[i*3+2]);
glEnd();

glFlush();
```

Es wird ein Array von 4 Vertex-Stützstellen definiert (jeweils x, y, z), sowie ein Array von 3 Farben (jeweils RGBA). Mit Hilfe von *glMap1f* werden die entsprechenden Verläufe definiert und anschließend mit *glEnable* aktualisiert. Als Bereich für *u* wurde jeweils [0,1] gewählt. Dann werden 31 Werte aus diesem Bereich in einer Schleife mit *glEvalCoord1f* abgefragt und als *GL_LINE_STRIP* dargestellt. *glEvalCoord1f* generiert hierbei intern *glVertex3f* und *glColor4f* Aufrufe. Das Ergebnis ist eine Bézier-Kurve, die einen Farbverlauf aufweist. Anschließend werden zur Veranschaulichung noch nie Stützstellen ausgegeben.

Hinweis:

glEvalCoord1f generiert zwar intern *glColor4f, ...* Anweisungen, diese verändern aber nicht die aktuell gesetzte Zeichenfarbe.

Hinweis:

Als Parameter *u* von *glEvalCoord1f* verlangt OpenGL nicht explizit Werte aus *[u1,u2]*, so daß prinzipiell auch Extrapolation der Kurve möglich ist.

In den meisten Fällen soll die Variable *u* mit konstanten Inkrementen innerhalb des Definitionsbereiches verändert werden. Hierfür gibt es eine effizientere Methode als das im obigen Beispiel praktizierte Abfragen einzelner Werte.

```
void glMapGrid1f(
    GLint un,
```

```
    GLfloat u1,
    GLfloat u2
    );
```

definiert Unter- und Obergrenze des Bereiches sowie die Schrittweite. *u1* ist die Unter-, *u2* die Obergrenze. *un* gibt die Anzahl der Stellen an, in die der Bereich unterteilt werden soll.

Mit Hilfe von

```
void glEvalMesh1(
    GLenum mode,
    GLint i1,
    GLint i2
    );
```

kann dann die durch *glMapGrid1f* definierte Schrittzahl auf die Kurvendefinition angewendet werden. Das Ergebnis ist das gleiche wie bei einer Sequenz von *glEvalCoord1f* Anweisungen. *mode* kann die Werte ***GL_LINE*** oder ***GL_POINT*** annehmen. Die Kurve wird dann als Linienzug oder als Folge von Punkten dargestellt. *i1* und *i2* geben an, von und bis zu welchem Schritt die Berechnung der Kurve vorgenommen wird.

Beispiel 2:

```
glMapGrid1f( 30, 0, 1);
glEvalMesh1(GL_LINE, 0, 30);
```

Dieses Fragment bewirkt das gleiche wie die Sequenz

```
glBegin(GL_LINE_STRIP);
    for(i=0; i<31; i++)
        glEvalCoord1f( ((float)i) / 30.0f );
glEnd();
```

aus Beispiel 1.

12.3 Bézierflächen

Grundsätzlich gelten für Flächen die gleichen Funktionen wie für Kurven, nur daß jetzt jeweils zwei Parameter, *u* und *v*, benötigt werden.

Für die Definition eines Werteverlaufes:

```
void glMap2f(
```

```
GLenum target,
GLfloat u1,
GLfloat u2,
GLint ustride,
GLint uorder,
GLfloat v1,
GLfloat v2,
GLint vstride,
GLint vorder,
const GLfloat *points
);
```

u1, u2, v1, v2 geben die Wertebereichsgrenzen für *u* und *v* an. *ustride* gibt an, wieviele *floats* zwischen dem Beginn von Stützstelle *P(ij)* und dem Beginn von Stützstelle *P*((*i+1*)*j)* liegen. *vstride* gibt an, wieviele *floats* zwischen dem Beginn von Stützstelle *P(ij)* und dem Beginn von Stützstelle *P*(*i(j+1))* liegen. *uorder* gibt die Ordnung der Kurve in *u*-Richtung, *vorder* in *v*-Richtung an. Der Parameter *target* kann die selben Werte wie bei *glMap1f* annehmen, wobei in den Konstantennamen die Zeichenkette ***MAP1*** durch ***MAP2*** zu ersetzen ist.

Der in *glMap2f()* definierte Kurvenverlauf wird wie gehabt mit *glEnable(target)* aktualisiert.

Einzelne Werte können mit

```
void glEvalCoord2f(
    GLfloat u
    GLfloat v
    );
```

berechnet werden. *glEvalCoord2f* funktioniert analog zu *glEvalCoord1f*.

Sollen *u* und *v* mit konstanten Inkrementen innerhalb des Definitionsbereiches verändert werden, bedient man sich der Funktion

```
void glMapGrid2f(
    GLint un,
    GLfloat u1,
    GLfloat u2
    GLint vn,
```

GLfloat v1,

GLfloat v2

);

u1 ist die Unter-, *u2* die Obergrenze, *un* die Anzahl der Stellen der Unterteilung des *u*-Bereiches. *v1* ist die Unter-, *v2* die Obergrenze, *vn* die Anzahl der Stellen der Unterteilung des *v*-Bereiches.

Mit Hilfe von

void glEvalMesh2(

GLenum mode,

GLint i1,

GLint i2

GLint j1,

GLint j2

);

können die durch *glMapGrid2f* definierten Schrittzahlen auf die Flächendefinition (*glMap2f*) angewendet werden. *mode* kann die Werte ***GL_FILL***, ***GL_LINE*** oder ***GL_POINT*** annehmen. Die Fläche wird dann gefüllt, als Drahtgitter oder als Folge von Punkten dargestellt. *i1* und *i2* geben an, von und bis zu welchem Schritt die Berechnung der in *u*-Richtung vorgenommen wird. Das gleiche erledigen *j1* und *j2* für die *v*-Richtung.

Beispiel:

```
float winkel=0.0;
float stuetzpunkte[9*3] = {   -0.9, -0.9, 0,          0.0, -0.9, 0.6,
                               0.9, -0.9, 0,         -0.9,  0.0, 0.6,
                               0.0,  0.0, -0.8,       0.9,  0.0, 0.6,
                              -0.9,  0.9, 0,          0.0,  0.9, 0.6,
                               0.9,  0.9, 0 };

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum( -1,   1,  -1,   1,   1.0,   3 );
gluLookAt( 0, 0, 2,   0, 0, 0,   0, 1, 0 );
glMatrixMode(GL_MODELVIEW);

glMap2f(GL_MAP2_VERTEX_3, 0,1,3,3, 0,1,9,3, stuetzpunkte);
glEnable(GL_MAP2_VERTEX_3);
glEnable(GL_AUTO_NORMAL);
glMapGrid2f(20,0,1,20,0,1);

//LIGHT0 DEFINITION
GLfloat position0[4]    = {0.5, 0.5, 0.4, 1.0};
GLfloat color0[4]       = {1.0, 1.0, 1.0, 1.0};
GLfloat color1[4]       = {0.0, 0.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_POSITION, position0);
```

```
glLightfv(GL_LIGHT0, GL_DIFFUSE, color0);
glLightfv(GL_LIGHT0, GL_SPECULAR, color1);
glShadeModel(GL_SMOOTH);
glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER,1);
glLightModelf(GL_LIGHT_MODEL_TWO_SIDE,1);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);

//MATERIAL DEFINITION
GLfloat ambi[4]   = {0.3, 0.4, 0.1, 1.0}; //ambiente Reflexion
GLfloat diff[4]   = {0.7, 1.0, 0.7, 1.0}; //diffuse Reflexion vorn
GLfloat diffb[4]  = {0.0, 0.0, 0.0, 1.0}; //diffuse Reflexion hinten
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,  ambi);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, diffb);
glMaterialfv(GL_BACK, GL_DIFFUSE,  diffb);
glMaterialfv(GL_FRONT, GL_DIFFUSE,  diff);
glFrontFace(GL_CW);
glEnable(GL_DEPTH_TEST);

while(true) {
      glClear(GL_COLOR_BUFFER_BIT);
      glClear(GL_DEPTH_BUFFER_BIT);

      glLoadIdentity();
      winkel+=3;
      glRotatef(winkel,0,1,0);

      glLightfv(GL_LIGHT0, GL_POSITION, position0);
      glEvalMesh2(GL_FILL,0,20,0,20);

      glFlush();
      SwapBuffers(hdc);
}
```

Es handelt sich hierbei um die Animation einer beleuchteten Bezierfläche. Mit *glEnable(**GL_AUTO_NORMAL**)* weist man OpenGL an, die Normalenvektoren der Teilflächen selbst zu berechnen, was ja für die Beleuchtung nicht ganz unwichtig ist. Bei jedem Animationsschritt wird die Fläche (samt Lichtquelle) um 3 Grad um die y-Achse gedreht.

13 NURBS

Die GLU stellt einige Funktionen für die Arbeit mit NURBS (Non-Uniform-Rational B-Splines) bereit. Die *Basis-Splines* oder *B-Splines* gelten als das Nonplusultra des Freiformdesigns. Der wesentliche Unterschied zu den Bézier-Kurven besteht darin, daß die Stützstellen **lokalen** Einfluss auf die Kurve ausüben.

13.1 Mathematische Grundlagen

Eine Verallgemeinerung von Bézier-Kurven stellen die *B-Splines* dar. Wie bereits erwähnt, ist einer der wesentlichsten Unterschiede im Vergleich zu Bézier-Kurven die Eigenschaft, daß die Stützpunkte nur einen lokalen Einfluß ausüben und daß der Grad der Polynome unabhängig von der Anzahl der Punkte ist.

Die Definition der *B-Splinekurve P(t)* vom Grad *k-1* (das heißt von der Ordnung *k*) erfolgt über *n+1* Stützpunkte P_i , $0 \le i \le n$, und einen Vektor von ganzen Zahlen, den *Knotenvektor* $T = (t_0 , \ldots , t_{n+k})$, $t_j \le t_{j+1}$, in Parameterform durch

$$P(t) = \sum_{i=0}^{n} N_{i,k}(t) \cdot P_i$$

Die *Basis* des B-Splines sind die Polynome $N_{i,k}(t)$ vom Grad *k-1*, die durch folgende Rekursionsformel berechnet werden:

$$N_{i,1}(t) = \begin{cases} 1 & \textit{falls} \quad t_i < t_{i+1} \quad \textit{und} \quad t_i \le t \le t_{i+1} \\ 0 & \textit{sonst} \end{cases}$$

$$N_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} \cdot N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} \cdot N_{i+1,k-1}(t), \quad k > 1$$

Da in Abhängigkeit vom Knotenvektor eine Division durch Null auftreten kann, wird für diesen Fall 0/0 gleich 0 gesetzt.

Sind die Knoten *t* im Knotenvektor äquidistant angeordnet, so handelt es sich um *uniforme* B-Splines, ansonsten um *nonuniforme.*

Eine häufig verwendete Möglichkeit zur Festlegung der Knotenwerte t_j , $0 \le j \le n+k$ ist folgende Formel

$$t_j = \begin{cases} 0 & \textit{falls} \quad j < k \\ j - k + 1 & \textit{falls} \quad k \le j \le n \\ n - k + 2 & \textit{falls} \quad j > n \end{cases}$$

wodurch *t* im Intervall [0, *n-k+2*] definiert ist. Wählt man zum Beispiel $k = 3$ und $n = 4$ so erhält man mit obiger Formel den Knotenvektor *T = (0,0,0,1,2,3,3,3).*

13.2 NURBS-Kurven

Zuerst benötigt man von der GLU eine Referenz auf ein NURBS-Objekt, über welche man bei nachfolgenden Operationen die NURBS identifizieren kann. Die entsprechende Funktion ist

```
GLUnurbsObj* gluNewNurbsRenderer();
```

Sie liefert einen Zeiger auf *GLUnurbsObj* zurück, womit die entsprechende Kurve eindeutig identifiziert wird.

Wird das NURBS-Objekt nicht mehr benötigt, kann man es mittels

```
void gluDeleteNurbsRenderer(
    GLUnurbsObj *nobj
    );
```

wieder entfernen lassen. Hierbei ist **nobj* der durch *gluNewNurbsRenderer* erhaltene Zeiger.

Die Definition einer NURBS-Kurve wird geklammert ,ähnlich wie die Vertexdefinition zwischen *glBegin* und *glEnd*, durch folgendes Funktionenpaar:

```
void gluBeginCurve(
    GLUnurbsObj *nobj
    );
void gluEndCurve(
    GLUnurbsObj *nobj
     );
```

Zwischen diesen beiden Funktionen darf jeweils die Definition **einer** Kurve erfolgen. Die Definition selbst erfolgt mit der Funktion

```
void gluNurbsCurve(
    GLUnurbsObj *nobj,
    GLint nknots,
    GLfloat *knot,
    GLint stride,
    GLfloat *ctlarray,
    GLint order,
    GLenum type
    );
```

**nobj* ist der von *gluNewNurbsRenderer* erhaltene Zeiger.

Der Parameter *nknots* gibt die Anzahl der Knoten des Knotenvektors an, **knots* ist ein Zeiger auf diesen Vektor, das heißt auf ein Array von *float*-Werten.

In **ctlarray* wird ein Zeiger auf ein Array mit den Werten der Stützstellen übergeben. *stride* gibt an, wieviele *floats* zwischen dem Beginn einer Stützstelle und dem Beginn der nächsten liegen. Die Anzahl der Stützstellen ergibt sich aus *nknots* und *order*. Der Parameter *order* gibt die Ordnung der Kurve an (Grad+1). *nknots* errechnet sich aus Ordnung + Anzahl der Kontrollpunkte.

Der Parameter *type* gibt die Art der Stützstellen an. Hierbei sind folgende Konstanten möglich (siehe auch *glMap1f*):

target	***Art der Stützstellenwerte***
GL_MAP1_VERTEX_3	x, y und z Koordinate
GL_MAP1_VERTEX_4	x, y, z und w Koordinate
GL_MAP1_INDEX	Index aus der Farbtabelle
GL_MAP1_COLOR_4	Farbwert als RGBA
GL_MAP1_NORMAL	Normalenvektor x, y, z
GL_MAP1_TEXTURE_COORD_1	Texturkoordinate s
GL_MAP1_TEXTURE_COORD_2	Texturkoordinaten s, t
GL_MAP1_TEXTURE_COORD_3	Texturkoordinaten s, t, r
GL_MAP1_TEXTURE_COORD_4	Texturkoordinaten s, t, r, q

Beispiel:

```
GLUnurbsObj *id = gluNewNurbsRenderer();
GLfloat knots[9] = {0,0,0,0,1,2,2,2,2};
GLfloat points[5*3] = { -0.9, -0.9, 0,    0.2, -0.6, 0,
                        0.3,  0.3, 0,     0.6, -0.4, 0,
                        0.0,  0.0, 0};

gluBeginCurve(id);
      gluNurbsCurve(id, 9, knots, 3, points, 4, GL_MAP1_VERTEX_3);
gluEndCurve(id);

glPointSize(3);
glColor3f(1,1,0);
glBegin(GL_POINTS);
      for(int i=0;i<5;i++)
            glVertex3f( points[i*3], points[i*3+1], points[i*3+2]);
glEnd();
glFlush();
```

Das Beispiel zeigt die Definition einer kubischen B-Spline mit fünf Stützstellen. Zusätzlich zur NURBS werden auch die Stützstellen angezeigt.

Es ist möglich, einem NURBS-Objekt verschiedene Eigenschaften zuzuordnen, die seine Darstellung maßgeblich beeinflussen (zum Beispiel den Grad der Tesselierung). Dies erfolgt mit Hilfe von

```
void gluNurbsProperty(
        GLUnurbsObj *nobj,
        GLenum property,
        GLfloat value
        );
```

**nobj* identifiziert das entsprechende NURBS-Objekt. Der Parameter *property* gibt an, welche Eigenschaft geändert werden soll. Diese Eigenschaft wird dann auf *value* gesetzt. Folgende symbolische Konstanten können als *property* angegeben werden:

Konstante	***Bedeutung***
GLU_SAMPLING_TOLERANCE	NURBS werden zum Zeichnen in Polygone zerlegt. Dieser Parameter gibt an, wie groß die Kantenlänge eines dieser Polygone maximal sein darf. Voreingestellt ist 50.
GLU_PARAMETRIC_TOLERANCE	Gibt den maximalen Abstand in Pixeln an, den die Approximationspolygone von der tatsächlichen NURBS haben dürfen. Voreingestellt ist 0,5.
GLU_U_STEP	Gibt an in wieviele Teile der Kurvenverlauf bei der Approximation zerlegt wird (in *u*-Richtung). Voreingestellt ist 100.
GLU_V_STEP	Gibt an in wieviele Teile der Kurvenverlauf bei der Approximation zerlegt wird (in *v*-Richtung, wird nur bei NURBS-Flächen benutzt). Voreingestellt ist 100.
GLU_SAMPLING_METHOD	Gibt an welche Methode bei der Polygon-Tesselation benutzt wird. Folgende Werte sind möglich: ***GLU_PATH_LENGTH*** Voreingestellt. Die Kantenlänge der Polygone darf den in *GLU_SAMPLING_TOLERANCE* eingestellten Wert nicht überschreiten. ***GLU_PARAMETRIC_ERROR*** Zwischen der Kurve und den Polygonen, die sie approximieren dürfen maximal *GLU_PARAMETRIC_TOLERANCE* Pixel liegen. ***GLU_DOMAIN_DISTANCE*** Die Kurve wird in *GLU_U_STEP* bzw. *GLU_V_STEP* Teile zerlegt.
GLU_DISPLAY_MODE	Gibt an, wie die Tesselationspolygone dargestellt werden. Folgende Werte sind möglich: ***GLU_FILL*** Voreingestellt. Ausgefüllte Polygone werden

	gezeichnet. ***GLU_OUTLINE_POLYGON*** Nur die Außenkanten der Polygone werden gezeichnet. ***GLU_OUTLINE_PATCH*** Nur die Außenkanten von Flächenstücken und Trimmkurven (siehe NURBS-Flächen) werden gezeichnet.
GLU_AUTO_LOAD_MATRIX	Die GLU benötigt Modelview- und Projektionsmatrix sowie die Viewportdefinition um zum Beispiel die oben angeführten Toleranzberechnungen durchzuführen. Diese Konstante gibt an, ob sich die GLU die Matrizen selbstständig vom Server holt oder ob sie übergeben werden müssen (mittels *gluLoadSamplingMatrices*) Voreingestellt ist *GL_TRUE*.
GLU_CULLING	Wenn *GL_TRUE* angegeben wird, wird die NURBS nicht berechnet, falls alle Stützpunkte außerhalb des Viewports liegen. Voreingestellt ist *GL_FALSE*, da eine NURBS nicht unbedingt vollständig innerhalb der konvexen Hülle ihrer Stützpunkte liegt.

Falls für ***GLU_AUTO_LOAD_MATRIX*** *GL_FALSE* eingestellt wurde, müssen die entsprechenden Matrizen „von Hand“ an die GLU übergeben werden. Dazu dient

```
void gluLoadSamplingMatrices(
    GLUnurbsObj *nobj,
    const GLfloat modelMatrix[16],
    const GLfloat projMatrix[16],
    const GLint viewport[4]
    );
```

Die entsprechenden Matrizen kann man sich über die entsprechenden *glGet...* Funktionen besorgen. Zum **Beispiel:**

```
GLfloat     ma_modelview[16];
GLfloat     ma_projection[16];
GLint       ma_viewport[4];
glGetFloatv(GL_MODELVIEW_MATRIX, ma_modelview);
glGetFloatv(GL_PROJECTION_MATRIX, ma_projection);
glGetIntegerv(GL_VIEWPORT, ma_viewport);
gluLoadSamplingMatrices(id, ma_modelview, ma_projection, ma_viewport);
```

13.3 NURBS-Flächen

Analog zur Spezifikation einer NURBS-Kurve erfolgt auch die einer Fläche. Es kommt eine weitere parametrische Dimension hinzu (Parameter sind jetzt *s* und *t)*. Um eine Fläche zu zeichnen benötigt man wieder einen Zeiger auf ein NURBS-Objekt, den man durch *gluNewNurbsRenderer* erhält.

Die eigentliche Definition wird wieder geklammert durch folgendes Funktionenpaar:

```
void gluBeginSurface(
        GLUnurbsObj *nobj
        );

void gluEndSurface(
        GLUnurbsObj *nobj
         );
```

Zwischen diesen „Klammern“ erfolgt die Spezifikation der Fläche mittels

```
void gluNurbsSurface(
        GLUnurbsObj *nobj,
        GLint sknot_count,
        GLfloat *sknot,
        GLint tknot_count,
        GLfloat *tknot,
        GLint s_stride,
        GLint t_stride,
        GLfloat *ctlarray,
        GLint sorder,
        GLint torder,
        GLenum type
        );
```

**nobj* ist der von *gluNewNurbsRenderer* erhaltene Zeiger.

Der Parameter *sknot_count* gibt die Anzahl der Knoten des Knotenvektors in *s*-Richtung an, **sknot* ist ein Zeiger auf diesen Vektor, das heißt auf ein Array von *float*-Werten. Den gleichen Zweck erfüllen *tknot_count* und **tknot* für die *t*-Richtung.

In **ctlarray* wird ein Zeiger auf ein Array mit den Werten der Stützstellen übergeben.

s_stride gibt an, wieviele *floats* zwischen dem Beginn einer Stützstelle und dem Beginn der nächsten in *s*-Richtung liegen. Analog in *t*-Richtung: *t_stride.*

Die Parameter *sorder* und *torder* geben die Ordnung der Kurven (Grad+1) in *s*- bzw. *t*-Richtung an.

Die Anzahl der Stützstellen ergibt sich aus *(sknot_count – sorder)*(tknot_count – torder)* und.

Der Parameter *type* gibt die Art der Stützstellen an. Hierbei sind folgende Konstanten möglich (siehe auch *glMap2f*):

Target	***Art der Stützstellenwerte***
GL_MAP2_VERTEX_3	x, y und z Koordinate
GL_MAP2_VERTEX_4	x, y, z und w Koordinate
GL_MAP2_INDEX	Index aus der Farbtabelle
GL_MAP2_COLOR_4	Farbwert als RGBA
GL_MAP2_NORMAL	Normalenvektor x, y, z
GL_MAP2_TEXTURE_COORD_1	Texturkoordinate s
GL_MAP2_TEXTURE_COORD_2	Texturkoordinaten s, t
GL_MAP2_TEXTURE_COORD_3	Texturkoordinaten s, t, r
GL_MAP2_TEXTURE_COORD_4	Texturkoordinaten s, t, r, q

Beispiel:

```
GLUnurbsObj *id = gluNewNurbsRenderer();
GLfloat knots[6]  = {0,0,0,1,1,1};
GLfloat points[9*3] = { -0.9,-0.9,-1,     0.0,-0.8,0,       0.9,-0.9,-1,
                        -0.7,-0.2,-0.5,   0.2, -0.1, 0,     0.8,0.2,-0.5,
                        -0.9,0.7,-1,      0.0,0.9,0,        0.9,0.6,-1 };
GLfloat colors[9*4] = { 1,0,0,1,    1,0,1,1,    1,1,1,1,
                        0,1,0,1,          1,1,1,1,    0,0,1,1,
                        0,0,1,1,          1,0,0,1,    0,1,0,1 };

gluNurbsProperty(id,GLU_DISPLAY_MODE,GLU_FILL);
gluBeginSurface(id);
   gluNurbsSurface(id,6,knots,6,knots,3,9,points,3,3,GL_MAP2_VERTEX_3);
   gluNurbsSurface(id,6,knots,6,knots,4,12,colors,3,3,GL_MAP2_COLOR_4);
gluEndSurface(id);
glFlush();
```

Das Beispiel zeigt die Definition einer NURBS-Fläche über 3×3 Stützstellen. Es werden Vertexkoordinaten und RGBA Werte spezifiziert.

14 DER NEBEL DES GRAUENS

In der Realität wird die Sicht meist durch winzige Partikel oder Wasserdampf in der Atmosphäre beeinflußt. Im Nebel werden die Objekte mit zunehmender Entfernung undeutlicher, ihre Farbe vermischt sich mit der Nebelfarbe, bis sie ganz in diese übergeht.

Genau dieses Vermischen mit der Nebelfarbe ist auch in der OpenGL realisiert. Die Mischoperation kann auf verschiedene Weise erfolgen. Die besten Ergebnisse erreicht man natürlich, wenn die Nebelrechnung für jedes Fragment abläuft. Schneller ist es, pro Vertex eine Farbe zu errechnen und den Rest der Schattierungsrechnung (*flat* oder *smooth*) zu überlassen. Die zu verwendente Methode läßt sich mit

```
void glHint(
    GL_FOG_HINT,
    GLenum mode
    );
```

wählen. Für *mode* können folgende Konstanten angegeben werden:

Konstante	***Bedeutung***
GL_FASTEST	Berechnung per Vertex.
GL_NICEST	Berechnung per Fragment.
GL_DONT_CARE	egal. Die von der Implementation eingestellte Methode wird verwendet.

Die Farbe des Nebels stellt man ein mit der Funktion

```
void glFogfv(
    GL_FOG_COLOR,
    const GLfloat *params
    );
```

**params* ist ein Zeiger auf ein Array mit den RGBA Werten der Nebelfarbe.

Die Vermischung mit dieser Farbe erfolgt über folgende Rechnung:

$$K_{neu} = S \cdot K_{Fragment} + (1-S) \cdot K_{Nebel}$$

K steht jeweils für eine Komponente der entsprechenden Farbe. Für den Skalierungsfaktor *S* sind drei verschiedene Berechnungsverfahren implementiert, zwischen denen mit folgender Funktion gewählt wird:

```
void glFogi(
    GL_FOG_MODE,
```

```
    GLint param
    );
```

param kann dabei die Werte ***GL_EXP, GL_EXP2*** und ***GL_LINEAR*** annehmen. Die ersten beiden Berechnungsvarianten bedienen sich der *e*-Funktion. Die Gleichungen lauten:

$$S = e^{-(d \cdot z)}$$

(für *GL_EXP*)

und

$$S = e^{-(d \cdot z)^2}$$

(für *GL_EXP2*).

z gibt dabei die Entfernung des Fragments zum Augpunkt an, *d* ist die Dichte des Nebels. Die Dichte kann mittels

```
void glFogf(
    GL_FOG_DENSITY,
    GLfloat param
    );
```

verändert werden, wobei *param* die Dichte angibt. Voreingestellt ist 1,0.

Die dritte Berechnungsvariante ist eine Verhältnisgleichung, die Farbmischung erfolgt linear:

$$S = \frac{z_{\max} - z}{z_{\max} - z_{\min}}$$

(für *GL_LINEAR*)

z stellt auch hier den Abstand zum Augpunkt dar. Die Parameter z_{min} und z_{max} geben den Bereich an, auf den die Farbmischung linear abzubilden ist. Diese Werte können gesetzt werden mit:

```
void glFogf(
    GLenum pname,
    GLfloat param
    );
```

pname gibt an, welcher Wert verändert werden soll. ***GL_FOG_START*** entspricht z_{min} und ***GL_FOG_END*** z_{max} . Der Parameter *param* gibt den Wert an, auf den die jeweilige Variable gesetzt werden soll. Voreingestellt sind $z_{min} = 0$ und $z_{max} = 1$.

Um die Nebelberechnung durchführen zu lassen, muß außerdem

*glEnable(**GL_FOG**);*

aufgerufen werden.

Beispiel:

```
glClearColor(0.3f, 0.3f, 0.6f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);

GLfloat fogcolor[4] = {0.3,0.3,0.6,0};
glFogfv(GL_FOG_COLOR,fogcolor);
glFogi(GL_FOG_MODE,GL_EXP);
glFogf(GL_FOG_DENSITY,0.8f);
glHint(GL_FOG_HINT, GL_NICEST);
glEnable(GL_FOG);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum( -1,   1,  -1,   1,   1.0,   5 );
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0.5, 1, 2,   0, 0, 0,   0, 1, 0 );

glColor4f(1,0,0,0);
for(float z=-2; z<2; z+=1) {
      glBegin(GL_POLYGON);
            glVertex3f(-0.9,-0.9,z);
            glVertex3f(0.5,-0.9,z);
            glVertex3f(-0.9,0.5,z);
      glEnd();
}

glFlush();
```

Dieses Beispiel zeigt die Verwendung von Nebel der Dichte 0,8. Als Nebelfarbe wurde ein Blauton gewählt, als Berechnungsmodell ***GL_EXP***. Es werden vier rote Dreiecke in verschiedenen Abständen zum Betrachter gezeichnet.

15 ANTIALIASING

Es geht hier um die Vermeidung des bekannten „Treppenstufen-Effektes“ bei Linien und Kanten. Welche Möglichkeiten OpenGL dazu bietet, soll im folgenden beschrieben werden.

15.1 Antialiasing mit Hilfe von Alpha – Blending

Um Kanten oder Linien scheinbar zu verwischen, muß über wenige Pixel hinweg ein Farbverlauf von der Linienfarbe zur Hintergrundfarbe erzeugt werden. Diese kann im Prinzip für jedes Pixel der Linie / Kante unterschiedlich sein, je nachdem, was vorher dort gezeichnet wurde. Der Mechanismus den OpenGL hier anbietet geht davon aus, das jede Linie eine bestimmte Breite hat (einstellbar mit *glLineWidth*). Durch diese Breite überdeckt die Linie die Rechtecke (Pixel) in bestimmten Verhältnissen. Bei eingeschaltetem *Antialiasing* wird der Alphawert des Fragmentes diesen Verhältnissen entsprechend gesetzt. Alles was dann noch zu tun ist, ist eine geeignete *Blending*-Funktion auszuwählen.

Antialiasing wird eingeschaltet mit

*glEnable(**GL_LINE_SMOOTH**);*

bzw. *glEnable(**GL_POLYGON_SMOOTH**);*

Dann muß noch *Blending* eingeschaltet und die *Blending*-Funktion gewählt werden:

*glBlendFunc(**GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA**);*

*glEnable(**GL_BLEND**);*

Beispiel:

```
glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glColor3f(1,1,0);
glLineWidth(2.0);

glDisable(GL_LINE_SMOOTH);
glBegin(GL_LINES);
      glVertex2f(-0.7,-0.9);
      glVertex2f(0.7, -0.8);
glEnd();

glEnable(GL_LINE_SMOOTH);
glBegin(GL_LINES);
      glVertex2f(-0.7,-0.7);
      glVertex2f(0.7, -0.6);
glEnd();

glDisable(GL_POLYGON_SMOOTH);
glBegin(GL_POLYGON);
      glVertex2f(-0.7, 0);
      glVertex2f(-0.3, -0.1);
      glVertex2f(-0.2,  0.4);
glEnd();
```

```
glEnable(GL_POLYGON_SMOOTH);
glBegin(GL_POLYGON);
      glVertex2f(0.7, 0);
      glVertex2f(0.3, -0.1);
      glVertex2f(0.2,  0.4);
glEnd();
glFlush();
```

Es werden jeweils eine Linie und ein Polygon mit und ohne Antialiasing gezeichnet.

15.2 Antialiasing mit dem Accumulation - Buffer

Der *Accumulationbuffer* ist eine Art zusätzlicher Bildspeicher, auf den verschiedene Operationen ablaufen können. Es kann allerdings nicht direkt in den Accumulationbuffer gezeichnet werden. Zur Benutzung des Accumulationbuffers steht folgende Funktion zur Verfügung:

void glAccum(
GLenum op,
GLfloat value
);

Der Parameter *op* nimmt dabei einen der folgenden Werte an:

Konstante	***Bedeutung***
GL_LOAD	Der Inhalt des aktuellen Buffer (Bildschirmspeicher) wird in den Acc.Buf. übernommen. Dabei werden alle Farbkomponenten mit *value* multipliziert. Der alte Inhalt des Acc.Buf. wird überschrieben.
GL_ACCUM	Der Inhalt des aktuellen Buffers wird auf den Acc.Buf. addiert. Dabei werden alle Farbkomponenten mit *value* multipliziert und anschließend zum alten Inhalt des Acc.Buf addiert.
GL_ADD	Zu den Komponenten aller Pixel im Acc.Buf. wird jeweils *value* addiert.
GL_MULT	Die Komponenten aller Pixel im Acc.Buf. werden jeweils mit *value* multipliziert.
GL_RETURN	Der Inhalt des Acc.Buf. wird in den aktuellen Buffer (Bildschirmspeicher) geschrieben. Dabei werden alle Farbkomponenten mit *value* multipliziert.

Mit

void glClearAccum(
GLfloat red,
GLfloat green,
GLfloat blue,
GLfloat alpha
);

kann eine Löschfarbe (RGBA) für den Accumulationbuffer festgelegt werden. Der Buffer wird gelöscht mit

*glClear(**GL_ACCUM_BUFFER_BIT**);*

Für die Zwecke des *Antialiasing* kann man sich des akkumulierenden Ladens (***GL_ACCUM***) bedienen. Um ein „Verschwimmen“ des Bildes zu erreichen, zeichnet man es einfach mehrfach um Bruchteile eines Pixels versetzt. Die Resultate werden jeweils akkumuliert, und zwar multipliziert mit (1 / Anzahl der Versetzungen). Abschließend wird der Inhalt des Accumulationbuffers zurückgeschrieben. Das Resultat ist ein Verwischen des Bildes, vergleichbar mit dem „Verwackeln“ beim Fotoapparat.

Folgendes **Beispiel** veranschaulicht die Vorgehensweise:

```
GLuint listid = glGenLists(1);
glNewList(listid,GL_COMPILE);
      glColor3f(1,1,0);
      glBegin(GL_LINES);
            glVertex2f(-0.7,-0.7);
            glVertex2f(0.7, -0.6);
      glEnd();
      glBegin(GL_POLYGON);
            glVertex2f(-0.7, 0);
            glVertex2f(-0.3, -0.1);
            glVertex2f(-0.2,  0.4);
      glEnd();
glEndList();

glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
glClearAccum(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_ACCUM_BUFFER_BIT);
for(int i=0; i<3; i++)
      for(int j=0; j<3; j++) {
            glClear(GL_COLOR_BUFFER_BIT);
            glLoadIdentity();
            glTranslatef((i*0.002)-0.002, (j*0.002)-0.002, 0);
            glCallList(listid);
            glAccum(GL_ACCUM, 1.0/9);
      }
glAccum(GL_RETURN, 1);
glFlush();
```

Welche Rechengenauigkeit der Accumulationbuffer zur Verfügung stellt, kann man mit folgender Funktion erfragen:

```
void glGetIntegerv(
        GLenum pname,
        GLint *params
        );
```

Wenn für *pname* ***GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS*** bzw. ***GL_ACCUM_BLUE_BITS*** eingesetzt wird, erfährt man wieviele Bits für die jeweilige Farbkomponente im Accumulationbuffer vorhanden sind. **params* ist ein Zeiger auf einen *Integer*-wert, in dem die erfragte Information abgelegt wird.

16 DER ATTRIBUT – STACK

Oft müssen beim Ablauf eines OpenGL-Programmes viele Zustandsvariablen geändert und wiederhergestellt werden. Um dieses zu vereinfachen implementiert die OpenGL einen Attributstack. Es können bestimmte Attributgruppen auf diesem Stack gesichert und wiederhergestellt werden. Die Tiefe des Stacks ist implementationsabhängig beträgt aber mindestens 16. Folgende Funktion legt ein Gruppe von Attributen auf den Stack:

```
void glPushAttrib(
    GLbitfield mask
    );
```

Der Parameter *mask* gibt dabei an, welche Attribute auf den Stack sollen. *mask* kann dabei eine OR-Verknüpfung der folgenden Konstanten sein:

Konstante	***Bedeutung***
GL_ACCUM_BUFFER_BIT	die Werte zum Löschen des Accumulation-Buffer
GL_ALL_ATTRIB_BITS	alle Attribute und Zustände
GL_COLO_BUFFER_BIT	Einstellungen für Bildspeicher, Blending und Alpha-Test
GL_CURRENT_BIT	Farbe, Normalen, Texturkoordinaten, Positionen
GL_DEPTH_BUFFER_BIT	Einstellungen für den z-Buffer
GL_ENABLE_BIT	alle Zustände, die mit *glEnable* und *glDisable* gesetzt werden können
GL_EVAL_BIT	Parameter der Evaluatoren
GL_FOG_BIT	alle Einstellungen für atmosphärische Effekte
GL_HINT_BIT	die Einstellungen von *glHint*
GL_LIGHTING_BIT	Reflexionseigenschaften, Beleuchtungsmodell, Lichtquellen
GL_LINE_BIT	Linienstärke, Linienmuster
GL_LIST_BIT	Parameter von *glListBase*
GL_PIXEL_MODE_BIT	Einstellungen für Bitmap-Operationen
GL_POINT_BIT	die Attribute der Punkte
GL_POLYGON_BIT	die Attribute der Polygone, Einstellungen für backface culling, den Umlaufsinn
GL_POLYGON_STIPPLE_BIT	das gültige Füllmuster für Polygone
GL_SCISSOR_BIT	die Parameter für den Scissor-Test
GL_STENCIL_BUFFER_BIT	alle Einstellungen für den Stencil-Test
GL_TEXTURE_BIT	Texturparameter
GL_TRANSFORM_BIT	clipping planes, Matrix-Modus
GL_VIEWPORT_BIT	den Viewport, near- und far-Werte

Die Funktion

```
void glPopAttrib(
        void
        );
```

restauriert die zuletzt abgelegte Gruppe wieder.