Proseminar Computergrafik: OpenGL

Verfasser: Marcel Heckel

Technische Universität Dresden
Fakultät Informatik
Studiengang Informatik (Diplom)

0. Inhalt

1. Allgemein

1.1. Was ist OpenGL

OpenGL steht für Open Graphics Library. Diese geräte- und betriebssystemunabhängige Bibliothek wurde von Silicon Graphics Inc. (SGI) entwickelt. Ursprünglich war OpenGL für den Einsatz auf hochwertigen grafischen Workstations gedacht, doch es entwickelte sich zu einem weit verbreiteten Standard vieler Betriebssysteme und wird von allen modernen Grafikkarten unterstützt. Heute ist OpenGL die zweithäufigste genutzte Software-Schnittstelle (nach Direct3D) für qualitative 2D und 3D Grafiken und grafisches Rendering (engl. : Wiedergabe, *comp.:* automatisches Erzeugen eines Bildes durch ein 3D Model das sich im Computer befindet.).

1.2. Geschichte

OpenGL entstand aus der vom Silicon Graphics (SGI) entwickelten IRIS GL.
1992 wurde von der OpenGL-Standard von der ARB (Architecture Review Board) definiert. Das ARB wurde 1992 gegründet und besteht z.Z. u.a. aus folgenden Mitgliedern: 3Dlabs, Apple, ATI, Dell, IBM, Intel, NVidia, SGI und Sun.
Microsoft, eines der Gründungs-Mitglieder, hat das ARB im März 2003 verlassen. (Seit dem versuch MS OpenGL zu verdrängen und seinen eigenen Standart (Direct3D) weiter stärker zu verbreiten. So will MS in WindowsVista z.B. OpenGL nur bis Version 1.4. unterstützen und auch nur emuliert durch Direct3D-Befehle. Wenn man dann einen richtigen OpenGL-Treiber des Grafikkartenherstellers installiert, werden die Transparenzeffekte für die Fenster deaktiviert (Quelle: http://www.heise.de/newsticker/meldung/62708). Weiter ist auch in VisualStudio2005 keinerlei OpenGL-Unterstützung mehr vorhanden (wie es noch in VC6 war). So muss man sich selber die nötigen Header und Lib.-Datein verschaffen.)

Versionen:
OpenGL 1.0 (1. Juli 1992)
- erste Veröffentlichung

OpenGL 1.1 (1997)
- Vertex Arrays
- Texture Objects
- Polygon Offset

OpenGL 1.2 (16. März 1998)
- 3D Texturen
- Neue Pixelformate (BGRA, Packed)
- Level-Of-Detail Texturen

OpenGL 1.2.1 (14. Oktober 1998)
- ARB Extensions eingeführt
- ARB Multitexture

OpenGL 1.3 (14. August 2001)
- komprimierte Texturen
- Cube-Maps
- Multitexturing

OpenGL 1.4 (24. Juli 2002)
- Tiefentexturen (für Shadow-Mapping)
- Automatische MipMap-Erzeugung
- Nebelkoordinaten

OpenGL 1.5 (29. Juli 2003)
- Pufferobjekte (Vorhalten von Daten im Grafikspeicher)

- Occlusion Queries

OpenGL 2.0 (7. September 2004)

- Shaderprogramme OpenGL Shading Language
- Multiple Render Targets
- Texturen beliebiger Größe (nicht mehr 2^n für Höhe und Breite)

(*Quelle: vi.*)

2. Etwas 3D-Mathematik

2.1. Das Koordinatensystem

OpenGL nutzt das gebräuchliche kartesische (orthonormierte) Koordinatensystem. Es besteht aus den 3 senkrecht zu einander stehenden Achsen x, y, z. Auf den Bildschirm übertragen ist die x-Achse der untere Rand, die y-Achse ist der linke Rand und in den Monitor blickt man entlang der negativen z-Achse. Ein Punk wird mit den 3 Koordinaten angegeben: P(x; y; z).

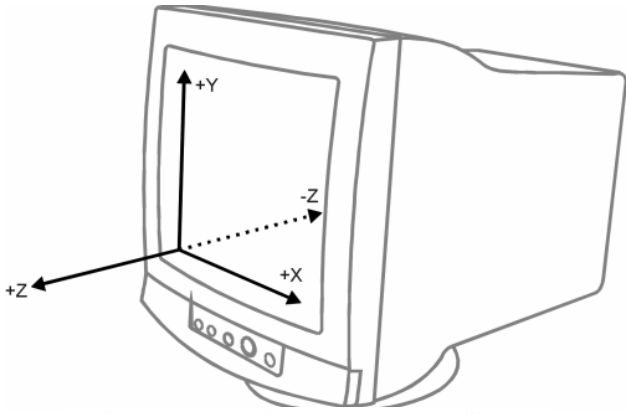


Bild 2.1-1: Koordinatensystem (Quelle: iv.)

2.2. Vektoren

- Länge ein Vektors: $|v| = \sqrt{v.x^2 + v.y^2 + v.z^2}$
- Einheitsvektor: $v_e = v / |v|$
- (Punkt-/)Skalarprodukt: $v_1 \bullet v_2 = v_1.x \cdot v_2.x + v_1.y \cdot v_2.y + v_1.z \cdot v_2.z = \cos \alpha \cdot |v_1| \cdot |v_2|$
- Kreutzprodukt: Senkrechter Vektor zu 2 anderen (Rechte-Hand-Regel: v_1 – Daume, v_2 – Zeigefinger, Mittelfinger – v_3)
  $v_3 = v_1 \times v_2 = (v_1.y \cdot v_2.z - v_1.z \cdot v_2.y ; v_1.z \cdot v_2.x - v_1.x \cdot v_2.z ; v_1.x \cdot v_2.y - v_1.y \cdot v_2.x)$

2.3. Matrizen

Die meisten Berechnungen mit vom Nutzer angegebenen Punkten werden über 4x4-Matrizen vorgenommen (Pos./Ausrichtungsmatrix für Objekte, Projektion, Texturkoordinaten). Grundzustand der Matrizen ist immer die Einheitsmatrix.

a01	a05	a09	a13
a02	a06	a10	a14
a03	a07	a11	a15
a04	a08	a12	a16

Bild 2.3-1: Matrix-Bereiche

Wenn die Matrix als als Pos./Ausrichtungsmatrix für Objekte genutzt wird ist die Bedeutung der Werte (Farbem im Bild 2.3-1):
Der grüne Teil (a13, a14, a15) gibt die (x, y, z)-Koordinaten im Raum an.
Der rote Teil die Ausrichtung von Objekten an:

(a01, a02, a03) – Links
(a05, a06, a07) – Oben
(a09, a10, a11) – vorn

Wenn die 3 Vektoren nicht ganz rechtwinklig zu einander sind bekommt man Verschiebungen/Verzerrungen, so dass z.B. rechtwinklige Objekte nicht mehr rechtwinklig aussehen.
Der blaue Teil kann für bestimmt Normierungen genutzt werden, sollte aber im Allgemeinen immer denn Wert (0,0,0,1) haben.
Die Nummerierung entspricht der von OpenGL erwarteten Reihenfolge, wenn man eine Matrix direkt übergibt

2.3.1. Translation

Bewirkt bei Multiplikation mit einer Ausrichtungsmatrix, eine Ortsänderung relativ entlang der aktuellen Ausrichtung.
Ein direktes Ändern der x, y, z-Werte in der Matrix bewirkt eine Bewegung entlang der Koordinatenachsen.

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

Bild 2.3.1-1: Translations-Matrix

2.3.2. Rotation

Rotationen um Winkel α um die x, y bzw. z Achse:

$$R_X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_Y = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_Z = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Bild 2.3.2-1: Rotations-Matrizen für Rotation um die x ,y bzw. z-Achse

2.3.3. Skalierung

Eine Multiplikation mit der S-Matrix würde eine Streckung(|X|>1), Stauchung (|X|<1) oder Spiegelung (X<0) bewirken entlang der jeweiligen Achse (mit $X \in \{x,y,z\}$).

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

Bild 2.3.3-1:Skalierungs-Matrix

2.3.4. Multiplikation

Matrixmultiplikationen kann man verschiedene Bewegungen zusammenfassen. Das sie aber nicht kommutativ ist, kommt es auf die Reihenfolge der Multiplikationen an. Beispielsweisen wenn man erst eine Translation ausführt und darauf eine Rotation, würde man eine Rotation auf einer Kreisbahn beobachten (Der Betrag der Translation senkrecht zur Rotationsachse wäre der Radius). Diese Kreisbahn könnte man vermeiden in dem man bei der Multiplikation die 4. Spalte der Matrizen weglässt.

3. Grundlegendes zu OpenGL

3.1. Arbeitsweise

OpenGL wurde als schlankes hardwareunabhängiges Interface entworfen. Es gibt keine Unterstützung für Fenster bzw. Benutzereingaben. So muss man je nach Betriebssystem dies speziell einrichten.
Auch bietet OpenGL keine Unterstützung für komplizierte 3D-Formen, wie Autos, Menschen oder Bäume. Man muss sein Modell aus einfachen Primitiven wie Dreiecke, Linien und Punkte zusammensetzen.
Es gibt aber ein paar Bibliotheken die solche Funktionalitäten auf OpenGL aufsetzten. Siehe da zu Abschnitt *3.3 Bibliotheken*

OpenGL wurde zudem als Zustandsautomat entworfen, d.h. einmal gesetzte Einstellungen bleiben erhalten und gelten für die folgenden Operationen, bis man sie wieder auf einen andern Wert setzt. So werden zum Beispiel nach eine bestimmte Farbe gesetzt wurde, alle Objekte in dieser Farbe gezeichnet, bis man die Farbe wieder ändert.

3.2. Primitiven

In OpenGL setzt sich jedes gezeichnete Objekt aus einfachen Primitiven zusammen. Die Primitiven setzen sich wieder aus einzelnen Punkten (engl.: Vertex / pl.: Vertices) zusammen. Die folgenden Primitiven sind in verfügbar:

Name	Bedeutung
GL_POINTS	jeder Vertex bildet einen Punkt
GL_LINES	je 2 Vertices bilden eine Linie
GL_LINE_STRIP	verbundene Linie von v0 bis v_n
GL_LINE_LOOP	wie STRIP, aber v0 und v_n werden noch verbuden
GL_TRIANGLES	je 3 Vertices bilden ein Dreieck
GL_TRIANGLES_STRIP	v0, v1, v2 dann v1, v2 ,v3 ... bilden je ein Dreieck
GL_TRIANGLE_FAN	v0, v1, v2 dann v0, v3, v4 ... bilden je ein Dreieck
GL_QUARDS	je 4 Vertices bilden ein Viereck
GL_QUARD_STRIP	v0, v1, v3, v2 dann v2, v3, v5, v4 ... bilden je ein Viereck
GL_POLYGON	zeichnet ein Polygon aus v0 – v_n

Tabelle 3.2-1: Primitiven

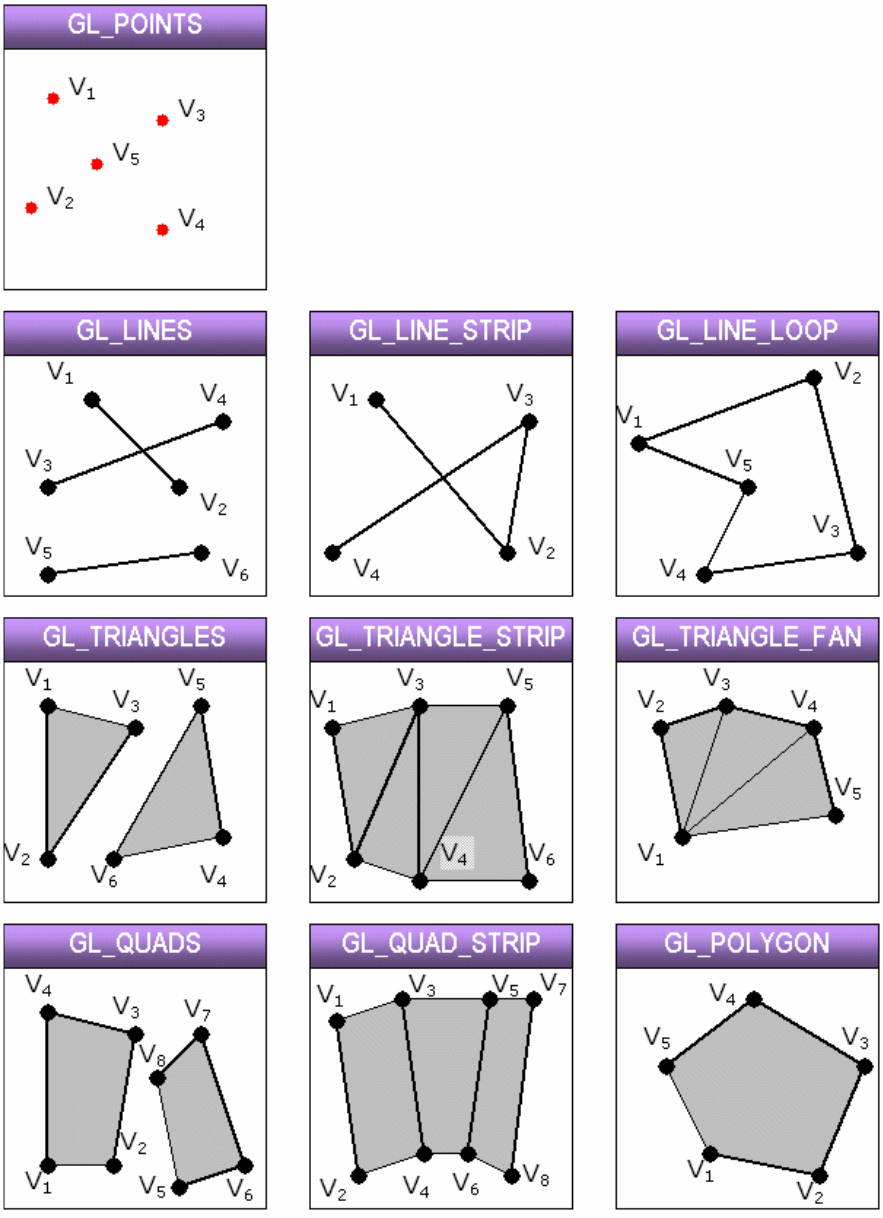
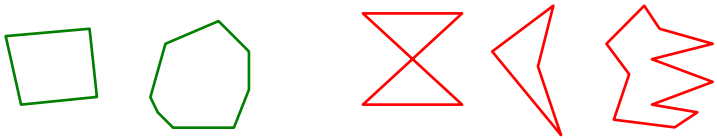


Bild 3.2-1: Primitiven (Quelle: iv.)

Bei Vierecken und Polygonen muss man noch beachten das sie konvex (nach außen gewölbt) sind und alle Punkte in einer Ebene liegen. Wenn die nicht der Fall ist, ist das Ergebnis nicht direkt vorhersagbar und kann ja nach OpenGL-Implementation variieren.



richtig falsch

Bild 3.2-2: konvexe (grün) und konkave (rot) Objekte

3.3. Bibliotheken

Die Standardbibliothek ist opengl32.dll (opengl32.lib, gl\gl.h). Ich empfehlen aber glew32.dll (glew32.lib, gl\glew.h). Diese Bibliothek bieten nehmen der normalen OpenGL Schnittstelle auch Funktionen zu Bestimmung, ob bestimmt extra Funktionen (Extensions) verfügbar sind.
Als zusätzliche Bibliotheken sind u.a. glu.dll (glu.lib, gl\glu.h) und glut (glut.lib, gl\glut.h) verfügbar.
Die „glu“ (OpenGL Utility Library) bietet mehrere nützliche Standardfunktionen zum Zeichnen.
Die „glut“ bietet u.a. Funktionalität zu Fenstererzeugung und Nutzereingaben (siehe Abschnitt *3.5 glut*)

3.4. Funktionsbezeichnungen und Datentypen

Die OpenGL Funktionen fangen allen mit spezifischen Buchstabenkombinationen an, um sie zu unterscheiden: gl...(glew32.dll), glu...(glu32.dll), glut...(glut.dll). OpenGL Funktionen speziell für Windows fangen mit wgl... an, für X-Window-Systeme mit glx.
Die OpenGL Funktionen wurden nicht überladen definiert (für bessere Kompatibilität), d.h. für unterschiedliche Funktionsparametertypen gibt es unterschiedliche Funktionsnamen. Zur Bezeichnung der Funktionen wurden bestimmte Suffixe benutzt (*Tabelle 3.4-1*). So gibt es z.B.: `glVertex*(...)`, wobei * für das Suffix steht. Konkrete Funktionsnamen währen: `glVertex2d, glVertex2f, glVertex2i, glVertex2s, glVertex3d, glVertex4d, glVertex2dv`...
→ Dabei steht die Zahl für die Anzahl der Parameter, [`d/f/i/s`] siehe *Tabelle 3.4-1* und `v` gibt an, dass die Parameter in Form eines Arrays übergeben werden.
Für OpenGL hat man zudem spezielle Datentypen-Bezeichnungen eingeführt. Die entsprechenden C-Typen zu dem GL-Typen sind in der *Tabelle 3.4-1* dargestellt.

Fkt.-Suffix	Datentyp	GL-Typ	C-Typ	Fkt.-Suffix	Datentyp	GL-Typ	C-Typ
b	8-Bit Integer	GLbyte	char	d	64-Bit Gleitkomma	GLdouble , GLclampd	double
s	16-Bit Integer	GLshort	short	ub	8-Bit vorzeichenloser Int.	GLubyte , GLboolean	unsigned char
i	32-Bit Integer	GLint, GLsizei	int	us	16-Bit vorzeichenloser Int.	GLushort	unsigned short
f	32-Bit Gleitkomma	GLfloat, GLclampf	float	ui	32-Bit vorzeichenloser Int.	GLuint , GLenum, GLbitfield	unsigned int

Tabelle 3.4-1: Funktions-Suffixe und Datentypen

3.5. glut

Die Beispielprogramme werden mit Hilfe der glut geschrieben. Deshalb hier mal eine etwas nähere Erläuterung dazu.

Glut ist eine Fenster-Betriebssystem unabhängige Bibliothek. Sie stellt Funktionen zu Fenstererzeugung, Auflöschungssetzung und einfache Benutzereingaben bereit. So kann man für kleine bis mittlere Programme betriebssystemunabhängigen Quellcode schreiben. Diese Bibliothek nimmt einem somit die Arbeit der betriebssystemabhängigen Initialisierung ab. Wer trotzdem daran interessiert ist, es lieber selber zu machen will, findet unter nehe.gamedev.net, neben anderen guten Tutorials, dazu gute Anleitungen für verschiedene Betriebssysteme.

3.5.1. wichtige Funktionen

- `void glutInit(int *argc, char **argv);`
  Initialisiert die glut.
  argc und *argv* sind die Parameter die der Main-Funktion unseres Programms übergeben wurden
- `void glutInitWindowPosition(int x, int y);`
  Legt die Fensterposition (*x*, *y*) fest. (-1) für *x* bedeutet die Default-Position des Fenstermanagers.
- `void glutInitWindowSize(int width, int height);`
  Legt die Fenstergröße fest.
- `void glutInitDisplayMode(unsigned int mode);`
  Einstellen des Anzeigemodus. Folgende Flag können bitweise verknüpft werden:
  - Farbmodel:
    `GLUT_RGBA / GLUT_RGB` – RGBA-Farben
    `GLUT_INDEX` – Farbpaletten
  - Pufferung, der Zeichenfläche:
    `GLUT_SINGLE` – keine Pufferung
    `GLUT_DOUBLE` – Gezeichnet wird in den Hintergrund-Puffer. Wenn das Zeichnen abgeschlossen ist werden die Zeiger für Vorder- und Hintergrundpuffer getauscht. Erzeugt flackerfreie Bildwechsel.
  - Zusätzliche Puffer (mehrere möglich):
    `GLUT_ACCUM` – Accumulation Buffer: Kann zur Speicherung zusätzlicher Farbinformationen genutzt werden (z.B. wenn der Farbpuffer nur 16Bit unterstützt, kann der ACCUM 32 Bit speichern, oder Infos für Anti-Aliasing durch Supersampling)
    `GLUT_STENCIL` – Damit können Bereche des Bildes vom Zeichnen ausgenommen werden. (z.B. für ungleichmäßige Umrahmung der Sichtfenster, wie Cockpits)
    `GLUT_DEPTH` – Puffer der die Tiefenwerte für die Bildpunkte speichert. Notwendig für eine korrekte Zeichnung von 3D-Bildern mit Tiefentest.
- `int glutCreateWindow(char *title);`
  Erzeugt ein Fenster mit den aktuellen Einstellungen für Position, Größe, und Anzeigemodus. Der Rückgabewert ist der Identifizierer des Fensters.
- `void glutDisplayFunc(void (*func)(void));`
  Legt die Callback-Funktion für die Zeichnung des Fensters fest.
- `void glutReshapeFunc(void (*func)(int width, int height));`
  Legt die Callback-Funktion fest, die aufgerufen wird wenn die Größer des Fensters geändert wird.
- `void glutIdleFunc(void (*func)(void));`
  Legt die Callback-Funktion fest, die aufgerufen wird wenn die Anwendung untätig ist. Für Animationen ohne Benutzereingaben.
- `void glutKeyboardFunc(void (*func) (unsigned char key, int x, int y));`
  `void glutSpecialFunc(void (*func) (int key, int x, int y));`
  Callback-Funktionen für Tastatur-Eingaben. (glutSpecialFunc für Spezialtasten wie F1-F12, Cursor, …) . *x* und *y* geben die aktuellen Mausposition an.
- `void glutMainLoop(void);`
  Startet die Nachrichtenschleife für die Anwendung.

- `void glutSwapBuffers();`
  Wechselt Vorder- und Hintergrundpuffer im Double-Buffered-Mode

3.6. kleines Beispiel

```
void init()
{ /* Initialisierung des Programms */ }
void renderScene(void)
{
      /* Szene zecihen */
      void glutSwapBuffers();
}
void changeSize(int w, int h)
{}
void keyFunction(unsigned char key, int x, int y)
{}

int main(int argc, char* argv[])
{
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
      glutInitWindowPosition(100,100);
      glutInitWindowSize(400,400);
      glutCreateWindow("kleines OpenGL Programm");
      glutDisplayFunc(renderScene);
      glutIdleFunc(renderScene);
      glutReshapeFunc(changeSize);
      glutKeyboardFunc(keyFunction);
      init();
      glutMainLoop();
      return 0;
}
```

4. Zeichen

4.1. Primitiven zeichnen

Wie schon gesagt wurde, kann OpenGL nur Primitiven zeichnen und die Primitiven bestehen wieder aus einzelnen Eckpunkten (Vertex). Um eine Privitive zu beginnen nutzt man die Funktion `glBegin()`, wenn man dann die Primitive abgeschlossen hat, ruft man `glEnd()` auf. Die Punkte der Primitive werden dann zw. `glBegin()` und `glEnd()` mit `glVertex*()` definiert.

- `void glBegin(GLenum mode);`
  Definiert den Anfang einer Liste von Vertices für das Zeichen von Primitiven. *mode* spezifiziert die Art der Primitive (siehe *Tabelle 3.2-1*).
- `void glEnd();`
  Markiert das Ende einer Vertices-Liste.
- `void glVertex{2/3/4}{s/i/f/d}[v](Typ Parameter);`
  Definiert einen Punkt (*x*, *y*, *z*, *w*). Bei 3 Parametern wird nur (*x*, *y*, *z*) angegeben, *w*=1; bei 2 Parametern (*x*,*y*), *z*=0, *w*=1. Wird nur zw. `glBegin()` und `glEnd()` akzeptiert.

(Zwischen `glBegin()` und `glEnd()` dürfen nur bestimmte Funktionen stehen(wichtige: `glVertex()`, `glColor()`, `glNormal()`, `glTexCoord()`)

4.1.1. Beispiel: Quadrate und Dreiecke

Programm: *Simple-double.exe*

```
glBegin(GL_TRIANGLES);
      glVertex3d(0.0,0.5,0.0);
      glVertex3d(0.0,0.0,0.0);
      glVertex3d(0.5,0.5,0.0);
```

```
      glVertex3d(0.5,0.0,0.0);
      glVertex3d(0.0,-0.5,0.0);
      glVertex3d(0.5,-0.5,0.0);
glEnd();

glBegin(GL_QUADS);
      glVertex3d(-0.9,0.9,0.0);
      glVertex3d(-0.9,-0.9,0.0);
      glVertex3d(-0.1,-0.9,0.0);
      glVertex3d(-0.1,0.9,0.0);
glEnd();
```

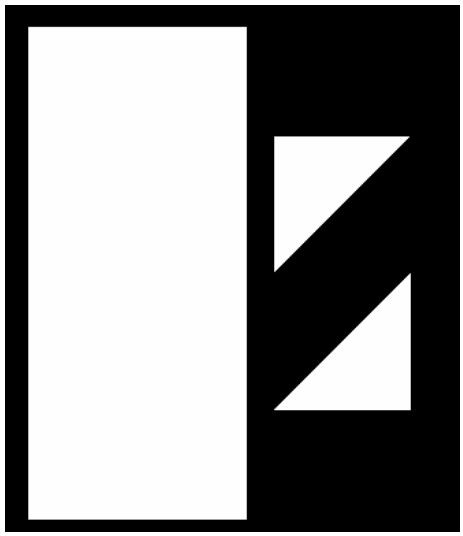


Bild 4.1.1-1: Ausgabe mit Standardeinstellungen

4.2. Das ganze mit etwas Farbe

Mit `glColor()` kann man zu jedem Vertex noch eine Farbe definieren. Wenn die Vertices einer Primitive verschiedene Farben haben, wird die Farbe zwischen den einzelnen Vertices interpoliert. Wenn man Texturen benutzt (siehe Abschnitt *6 Texturen*) werden dann auch nur die angegebenen Farbanteile der Textur genutzt.

```
glBegin(GL_TRIANGLES);
      glColor3f(1.0f,0.0f,0.0f);
      glVertex3f( 0.0f, 1.0f, 0.0f);
      glColor3f(0.0f,1.0f,0.0f);
      glVertex3f(-1.0f,-1.0f, 1.0f);
      glColor3f(0.0f,0.0f,1.0f);
      glVertex3f( 1.0f,-1.0f, 1.0f);
glEnd();
```

4.3. Vertex-Behandlung

Vertizes werden nach dem Pipeline-Verfahren abgearbeitet:

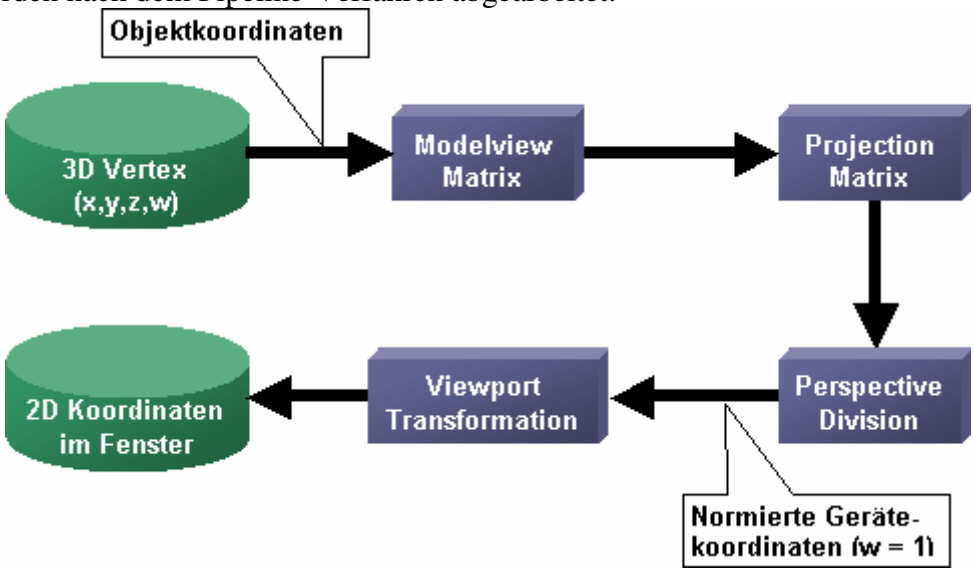


Bild 4.3-1: Vertex-Pipeline (Quelle: iv.)

4.3.2. Modelview

Nach dem ein Vertex definiert wurde, wird er mit der Modelview-Matrix multipliziert. So bekommt er die gewünschte Position und Ausrichtung in 3D-Raum, relativ zum Betrachter.

Der Vertex hat nur „Augenkoordinaten“. Dabei ist es egal ob man sich vorstellt, dass Objekt bewegt wurde oder der Betrachter sich bewegt hat. Denn wenn sich beispielsweise das Objekt nach vorn bewegt, müsste der Betrachter analog sich rückwärts bewegen.

4.3.3. Projektion

In diesem Schritt werden die „Augenkoordinaten“ aus dem 3D-Raum auf eine 2D-Fläche projiziert. Dies geschieht durch Multiplikation mit der Projektionsmatrix. In dem dabei entstehenden Vektor geben die (x,y)-Koordinaten schon den relativ den Punkt auf dem Bildschirm an. Die z-Koordinate ist der Wert der dann im Tiefenpuffer gespeichert wird.

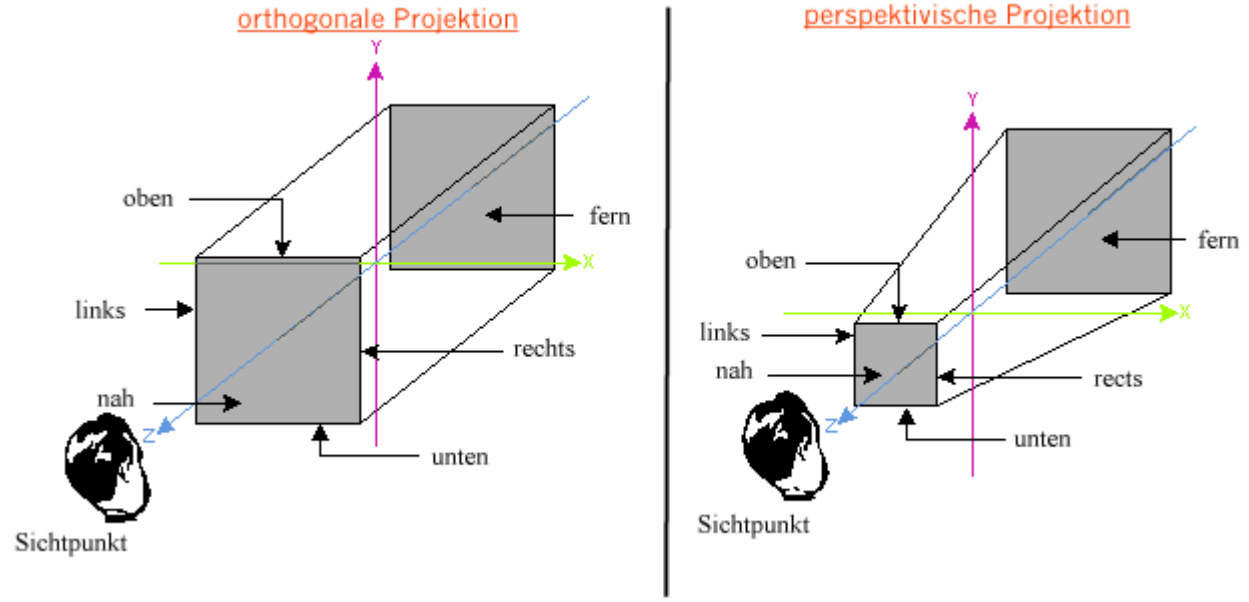


Bild 4.3.3-1: Projektionsarten

Bei der Projektion kann man zwischen orthogonaler und perspektivischer unterscheiden. Bei orthogonaler Projektion bleibt die Größe der Objekte erhalten unabhängig von der Entfernung. Hingegen ist perspektivische Projektion ist realistischer. Bei ihr werden die Objekte mit zunehmender Entfernung kleiner.
Weiter hin geschieht hier ein Stufe des Clippings: Alles was außerhalb der Begrenzungsebenen ist wird nicht gezeichnet. Das sind Punkte die zu nahe, bzw. hinter den Sichtpunkt sind, Punkte die außerhalb der Sichtpyramide liegen und Punkte die hinter der „Fern“-Fläche liegen.

4.3.4. Viewport-Transformation

Hierbei wird die projizierte Ebene sozusagen auf das angegebene Bildschirm-Rechteck gelegt und passend skaliert. Dabei entstehen die Fensterkoordinaten des Vertex.

4.3.5. Benötigte Funktionen

- `void glMatrixMode(GLenum mode);`
  Setzt den aktuellen Matrix-Modus. Akzeptierte Parameter sind GL_MODELVIEW (Modelview-Matrix), GL_PROJECTION (Projection-Matrix) und GL_TEXTURE (für Texturen, wird nicht in dieser Ausarbeitung nicht genutzt).
- `void glLoadMatrix{d/f}( Typ m);`
  Ersetzt die aktuelle Matrix mit dem 16-elementiken Array *m.*
- `void glMultMatrix{d/f}( Typ m);`
  Multipliziert die aktuelle Matrix mit dem 16-elementiken Array *m.*
- `void glPushMatrix(); void glPopMatrix();`
  `glPushMatrix()` kopiert die aktuelle Matrix und legt sie oben auf den Matrix-Stack.
  `glPopMatrix()` löscht die oberst Matrix im Stack und die darunter wird wieder zu aktuellen Matrix.
  Für jeden Matrixmodus gibt es einen eigenen Stack.

Diese Funktionen sind u.a. für Fällen, in denen man mehrere verschiedne Matrizen hat, die von einer Grundmatrix abhängen. So braut die Grundmatrix nicht ständig neu an OpenGL übergeben werden.

- `void glTranslate{d/f}( Typ x, Typ y, Typ z);`
  Die Translationsmatrix mit (*x ,y ,z*) wird mit der aktuellen Matrix multipliziert.
- `void glScalar{d/f}( Typ x, Typ y, Typ z);`
  Die Skalierungsmatrix mit (*x ,y ,z*) wird mit der aktuellen Matrix multipliziert.
- `void glRotate{d/f}( Typ angle, Typ x, Typ y, Typ z);`
  Führt eine Rotation von dem Winkel *angle* in Grad entgegen dem Uhrzeigersinn aus. Dabei wird um dem Vektor (*x ,y ,z*) rotiert.
- `void gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz :GLdouble);`
  Erzeugt eine Sichtmatrix von einem Augenpunkt zu einem Referenzpunkt. *eyex, eyey* und *eyez* bilden die Augenkoordinaten. *centerx, centery* und *centerz* den Referenzpunkt. *upx, upy* und *upz* bilden den Vektor für oben; dieser muss nicht senkrecht zum Sichtvektor sein.
- `void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble znear, GLdouble zfar );`
  Perspektivische Projektion: *left* und *right* geben die linke und rechte Begrenzungsebene und *botton* und *top* die untere und obere. *znear* und *zfar* geben den Abstand zur nahen und fernen Begrenzungsebene.
- `void glOrtho(GLdouble left, GLdouble right, GLdouble bottom GLdouble, top, GLdouble near, GLdouble far );`
  Orthogonale Projektion: Die Parameter haben die selbe Bedeutung wie bei *glFrustum( ).*
- `void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);`
  Perspektivische Projektion: *fovy* ist der Sichtwinkel in Grad, für *aspect* sollt man das Verhältnis aus Breite / Höhe des Sichtsbereiches übergeben. *zNear* und *zFar* siehe `glFrustum()`.
  Diese Funktion ist schöner zu verwendbar als `glFrustum()`.
- `void glViewport(x,y:GLint; width,height:GLsizei);`
  Legt den Bereich des Fensters fest in den OpenGL zeichnet. *x*, *y* geben die untere linke Ecke an, *width* gibt die Breite an und *height* die Höhe.

4.4. Tests vor dem Schreiben in den Puffer

Nach Position, Farbwert, usw. für das Fragment (also das zukünftige Pixel) feststehen, können je nach Einstellungen noch verschiedene Test vorgenommen werden. Beispielsweise wären dies Depth-Test, Alpha-Test, Stencil-Test.

4.4.1. Depth-Test

Dies ist der wohl wichtigste Test. Wie schon erwähn hat das Transformierte Fragment neben seinen x und y Werten noch seine z. Dies ist die „Tiefe“ des Punktes im Bild. Beim Tiefentest wird geprüft ob das im Z-Puffer stehender Pixel einen kleineren Tiefenwert hat als unser neues Fragment. Ist die der Fall, so bleibt das Pixel mit samt seinen Farbwerten erhalten. Anderenfalls wird es durch das neue ersetzt.

4.4.2. Alpha-Test

Der Alpha-Test bietet eine einfache Möglichkeit bestimmte Pixel auszublenden, um z.B. voll-transparente Teilflächen darzustellen. Im Test wird geprüft ob das Fragment einer bestimmten Bedingung entspricht (Alpha-Wert größer/gleich/kleiner Referenzwert. Der Alpha-Wert kann mittels `glColor4*()` und Texturen (siehe Abschnitt *6 Texturen*) gesetzt werden. Wenn die Bedingung nicht erfüllt ist wird das Fragment verworfen.

4.4.3. Stencil-Test

Hierbei wird ein Bitmap angegeben welches die Flächen, in die nicht gezeichnet werden soll markiert.

4.5. Beispiel: Farbige 3D Modelle:

Programm: *Color3D.exe*
Mit „D“ kann der Tiefenpuffer (de-)aktiviert werden.

Zu nächst muss man den Viewport gemeinsam mit der Modelview und Projektionsmatrix setzen:

```
void changeSize(int w, int h)
{
	glViewport(0, 0, w, h);//komplettes Fenster nutzen
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective( 90.0f,(float)(w) / (h) , 0.001f, 100.0f );
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}
```

(De-)Aktivieren des Tiffenpuffers:

```
void keyFunction(unsigned char key, int x, int y)
{
	static bool fDepthTest = false;
	switch(key)
	{
	case 'd':
		if(fDepthTest)
			glDisable(GL_DEPTH_TEST);
		else
			glEnable(GL_DEPTH_TEST);
		fDepthTest=!fDepthTest;
		break;
	}
}
```

Nun die Zeichenfunktion:

```
void renderScene(void)
{
	// Bild und Tiefenwerte löschen,
	// sonst würden wir in das alte Bild wieder rein malen
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	// glutGet(GLUT_ELAPSED_TIME) - gibt die vergangenen Millisekunden
	// seit glutInit() zurück
	glLoadIdentity();
	glTranslated(-3,0,-5);
	glRotated(glutGet(GLUT_ELAPSED_TIME)*0.1, 0.5,1.0,-0.5);
	renderCube();

	glLoadIdentity();
	glTranslated(3,0,-5);
	glRotated(glutGet(GLUT_ELAPSED_TIME)*0.1, 1.0,0.5,0.5);
	renderPyramid();
	//Puffer-Wechseln
	glutSwapBuffers();
}
```

…und die 3D Modelle:

```
void renderCube()
{
	glBegin(GL_QUADS);
	//oben
	glColor3f(0.0f,1.0f,0.0f);
	glVertex3f( 1.0f, 1.0f,-1.0f);
	glVertex3f(-1.0f, 1.0f,-1.0f);
	glVertex3f(-1.0f, 1.0f, 1.0f);
	glVertex3f( 1.0f, 1.0f, 1.0f);
```

```
        //unten
        glColor3f(1.0f,0.5f,0.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glVertex3f( 1.0f,-1.0f,-1.0f);
        //vorn
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 1.0f, 1.0f, 1.0f);
        glVertex3f(-1.0f, 1.0f, 1.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        //hinten
        glColor3f(1.0f,1.0f,0.0f);
        glVertex3f( 1.0f,-1.0f,-1.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glVertex3f(-1.0f, 1.0f,-1.0f);
        glVertex3f( 1.0f, 1.0f,-1.0f);
        //links
        glColor3f(0.0f,0.0f,1.0f);

        glVertex3f(-1.0f, 1.0f, 1.0f);
        glVertex3f(-1.0f, 1.0f,-1.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        //rechts
        glColor3f(1.0f,0.0f,1.0f);
        glVertex3f( 1.0f, 1.0f,-1.0f);
        glVertex3f( 1.0f, 1.0f, 1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        glVertex3f( 1.0f,-1.0f,-1.0f);
        glEnd();
}
void renderPyramid()
{
        glBegin(GL_TRIANGLES);
        //vorn
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        //rechts
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f( 1.0f,-1.0f, -1.0f);
        //hinten
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f( 1.0f,-1.0f, -1.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f(-1.0f,-1.0f, -1.0f);
        //links
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glEnd();
        glBegin( GL_QUADS );

        //unten
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f( 1.0f,-1.0f,-1.0f);
        glEnd();
}
```

5. Licht

5.1. Sinn von Licht

Licht ist ein wichtiger Bestandteil von 3D-Grafiken, da sie dadurch wesentlich realistischer Wirken. Zu dem kann man auch Szenen in verschiedenen Farben und Positionen beleuchten ohne etwas an der Szene ändern zu müssen (z.B.: ein Strand bei Tag und bei Sonnenuntergang). In OpenGL werden bis zu 8 Lichtquellen unterstützt.

5.2. Lichtarten

OpenGL unterscheidet zwischen 3 Lichtarten

- *ambient (umgebendes) Licht:* Allgemeines Licht, für die Umgebung. Es ist förmlich überall.
- *diffuse (zerstreutes/ausbreitendes) Licht:* Licht das beim Auftreffen auf eine Oberfläche in alle Reichungen zerstreut wird – wie bei matten Oberflächen.
- *specular (reflektiertes) Licht:* Licht das fast nur eine Richtung reflektiert wird, nach dem Auftreffen auf eine Oberfläche – wie bei glänzenden Oberflächen

Man kann zudem noch definieren welchen Abstrahlwinkel und Leuchtrichtung eine Lichtquelle hat. Standard ist 360° eingestellt, also rundum.

5.3. Licht und Normalen

Damit OpenGL weiß wo bei der Fläche oben sein soll, muss auch für jede Fläche bzw. Vertex eine Normale angegeben werden. Die Normale muss die Länge 1 haben (man kann das Erzeugen von Einheitsvektoren zwar auch OpenGL machen lassen, ist aber nur eine Arbeit, die dann unnötig viel Zeit kostet)

Wenn die Normale (teilweise) zu Lichtquelle zeigt, wird sie (in verschiedenen Abstufungen, je nach Einfallswinkel) beleuchtet dargestellt, sonst nur mit ambienten Licht. Da man die Normalen für jeden Vertex einzeln angeben kann, lassen sich gewölbte Flächen schöner darstellen: Man lässt die Normalen nicht ganze Senkrecht auf der Primitive stehen, sondern mehr so dass es der Wölbung entspricht. Es entsteht ein Vermischung der Beleuchtungswerte zwischen den einzelnen Vertices der Primitive → fliesenderer Übergang der Lichtes in der Primitive.

5.4. Licht aktivieren

In OpenGL ist Licht standardmäßig deaktiviert. Aktivieren kann man es mittels `glEnable()`:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);          // Licht 0 An
glEnable(GL_LIGHT1);          // Licht 1 An
...
```

Weitere Einstellungen kann man mit folgender Funktion vornehmen:

```
void glLightf{v} (GLenum light , GLenum pname , GLfloat {*}param);
```

light gibt die Lichtquelle an (LIGHT0 bis LIGHT7) . *pname* bestimmt den einzustellenden Wert und *param* ist der Wert.

pname	Standart	Beschreibung
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambiente RGBA Lichtintensität
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA Lichtintensität
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA Lichtintensität
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(*x, y, z, w*) Lichtposition
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(*x, y, z*) Leuchtrichtung
GL_SPOT_EXPONENT	0.0	Lichtstrahl Exponent
GL_SPOT_CUTOFF	180.0	Lichtstrahl-Halbkegel-Winkel
GL_CONSTANT_ATTENUATION	1.0	Konstanter Verringerungsfaktor
GL_LINEAR_ATTENUATION	0.0	linearer Verringerungsfaktor
GL_QUADRATIC_ATTENUATION	0.0	quadratischer Verringerungsfaktor

5.5. Licht-Beispielprogramm

Programm: *Licht.exe*
Mit A, D, W, S, Q, E kann die Szene rotiert werden. Mit den Cursor-Tasten lässt sich die Szene verschieben.

Ein Raum mit 2 Lichtquellen. Die erste ist blau und bewegt sich auf einer Kreisbahn im Raum, gekennzeichnet durch den weisen Punkt. Die zweite ist rot und fest oberhalb der Szene definiert. Es hat einen Abstrahlwinkel von 20° und leuchten nach untern.

Licht initialisieren:

```
void init()
{
	glEnable(GL_DEPTH_TEST);
	//Licht 0 (blaulich)
	GLfloat ambientLight0[] = { 0.0f, 0.0f, 0.2f, 0.0f };
	GLfloat diffuseLight0[] = { 0.0f, 0.0f, 0.5f, 0.0f
	GLfloat specularLight0[] = { 0.0f, 0.0f, 0.3f, 0.0f
	glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight0); // Ambientes Licht
	glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight0); // Verstreutes Licht
	glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight0); // Speculares Licht
	//Licht 0 (Rot)
```

```
	GLfloat ambientLight1[] = { 0.0f, 0.0f, 0.0f, 0.0f };
	GLfloat diffuseLight1[] = { 0.9f, 0.0f, 0.0f, 0.0f };
	GLfloat specularLight1[] = { 0.9f, 0.0f, 0.0f, 0.0f
	GLfloat posLight1[] = { 0.0f, 4.0f, 0.0f, 0.0f };
	GLfloat dirLight1[] = { 0.0f, -1.0f, 0.0f, 0.0f };
	glLightfv(GL_LIGHT1, GL_AMBIENT, ambientLight1);   // Ambientes Licht
	glLightfv(GL_LIGHT1, GL_DIFFUSE, diffuseLight1);   // Verstreutes Licht
	glLightfv(GL_LIGHT1, GL_SPECULAR, specularLight1); // Speculares Licht;
	glLightfv(GL_LIGHT1, GL_POSITION, posLight1); // Pos.
	glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 10);      // Speculares Licht;
	glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, dirLight1);

	glEnable(GL_LIGHTING);
	glEnable(GL_LIGHT0);          // Licht 0 An
	glEnable(GL_LIGHT1);          // Licht 0 An
	//Material-Beleuchtung
	// Farbe bei Lightning
	GLfloat specref[]      = { 0.9f, 0.9f, 0.9f, 0.0f };
	// Speculares Licht
	glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specref);
	// Spielgelung
	glMaterialf( GL_FRONT_AND_BACK, GL_SHININESS, 0.7);
}
```

Die Szene zeichnen und die Licht-Position der 1. Lichts berechnen

```
void renderScene(void)
{
	//neue Licht-Pos
	// glutGet(GLUT_ELAPSED_TIME)
	//    - gibt die vergangenen Millisekunden seit glutInit() zurück
	GLfloat posLight0[]     = {2.0f, 4*sin(glutGet(GLUT_ELAPSED_TIME)*0.001),
	                           4*cos(glutGet(GLUT_ELAPSED_TIME)*0.001), 1.0f};
	glLightfv(GL_LIGHT0, GL_POSITION, posLight0);
	//licht ausrichtung
	// Bild und Tiefenwerte löschen,
	// sonst würden wir in das alte Bild wirder rein malen
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	//
	glPushMatrix();
		glScaled(5.0,5.0,5.0);
		glColor3f(1.0f,1.0f,1.0f);
		renderCube();
	glPopMatrix();
	glPushMatrix();
		glutSolidTeapot(1);
	glPopMatrix();

	glDisable(GL_LIGHTING);
	glPushMatrix();
		glTranslated(posLight0[0],posLight0[1],posLight0[2]);
		glColor3f(1.0f,1.0f,1.0f);
		glutSolidSphere(0.1,10,10);
	glPopMatrix();
	glEnable(GL_LIGHTING);
	//Puffer-Wechseln
	glutSwapBuffers();
}
```

6. Texturen

6.1. Sinn von Texturen

Wenn man z.B. ein Ziegelsteinhaus möglicht realistisch zeichnen möchte, würde das sehr auswendig werden. Man müsste jeden Ziegelstein einzeln zeichnen. Dies würde zum einen eine großen Erstellungsaufwand bedeuten, zum anderen wäre Programm sehr langsam, da die ganzen einzelnen Flächen durch die Grafikkarte gejagt werden müssten.
Viel einfacher lässt sich das mit Texturen lösen. Texturen sind Bilder die man förmlich auf die Flächen klebt. Also können wir das Ziegelsteinhaus-Problem damit lösen dass man nur ein paar Flächen zeichnen und diese dann mit einer Textur von Ziegelsteinen überziehen.

6.2. Texturen allgemein

Texturen sind für OpenGL Arrays aus RGB(A)-Werten. Bis OpenGL 2.0 mussten Texturen Maße haben, die eine Potenz von 2 entsprachen. (also $h,b \in \{2^n \mid n \in N\}$).
Es werden 1D, 2D und sogar 3D-Texturen (seit Version 1.2) unterstützt. 2D-Texturen kann man sich als normale Bilder vorstellen. 1D ist ja nur der Spezialfall von 2D, bei dem das Bild die Höhe 1 hat.
3D-Texturen klingen, wenn man es das erste Mal hört, vielleicht etwas komisch. (Wie will man eine 3D Texturen auf einen 2D Bild darstellen – aber so darf man es aber nicht sehen). Eine 3D-Textur entsteht z.B. in einen Computer-Tomograph. Schichtenweise werden Bilder erzeugt die dann übereinander gestapelt werden. Nun will man einen bestimmten Querschnitt des 3D-Bildes sehen. So kann man ein Rechteck nehmen und für die Ecken nicht nur s, t-Koordinaten nehmen sondern noch einen Tiefenwerte (r) nehmen. Man kann auch einen schrägen Querschnitt bekommen in dem man verschiedene r-Werte für die Ecken nimmt.

6.3. Texturkoordinaten und Wiederholung von Texturen

Als Koordinatenbezeichnung nutzt man im Allgemeinen s, t, r (Breite, Höhe, Tiefe).
Texturkoordinaten werden mit `glTexCoord()` angeben. Wenn die Werte außerhalb [0.0; 1.0] liegen, wie die Textur entsprechend wiederholt. Man kann anstatt Wiederholung auch einstellen das die nur letzte Reihe der Textur gestreckt wird:

```
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_S_WRAP, GL_CLAMP);
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_T_WRAP, GL_CLAMP);
```

(GL_REPEAT anstatt GL_CLAMP für Wiederholung)

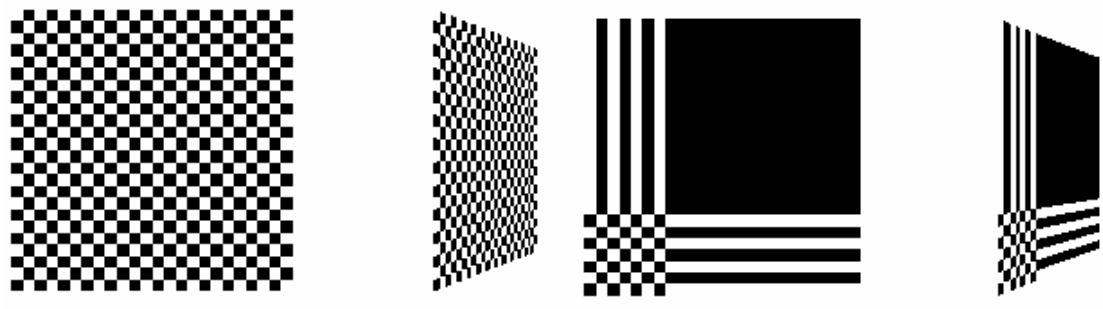


Bild 6.3-1: Texturwiederholung: links mit GL_REPEAT – rechts mit GL_CLAMP (Quelle: i.)

6.4. Filters

Wenn die Textur nicht genau die Größe hat wie die Primitive, muss die Textur verkleinert (*Minifikation*) bzw. vergrößert (Magnifikation) werden. Man kann dafür verschiedene Filter angeben.
OpenGL grundsätzlich folgende 2 Möglichkeiten für Vergrößerung:

- GL_NEAREST: es wird das Pixel genommen das am ehesten passt. Bewirkt beim Vergrößern, dass einfach nur große Pixel entstehen, die nicht besonders schön aussehen.
- GL_LINEAR: Bildung des Mittelwertes aus dem umgebenden Pixel. Das Bild wird somit beim Vergrößern unscharf, sieht aber besser aus als große Pixel.

Beim Verkleinern wird auch Nearest und Linear angeboten, wo bei sich die Qualität kaum unterscheidet. Ein Problem was bei beiden auftritt ist, wenn die Textur stark verkleinert wird: Wenn sich dann immer nur etwas die Größer ändert, kann bei bunten Texturen sein, dass immer mal eine andere Farbe bei der Verkleinerung „überlebt". So kann förmlich ein „Blicken" entstehen. Um dieses Problem zu beheben hat man MipMaps (Abschnitt *6.5*) eingeführt

6.5. Mip-Maps

Mip steht für „*multum in parvo"*, was so viel wie „*viel auf kleinem Platz"*bedeutet. Dabei wird neben dem Originalbild noch mehrere verkleinerte Versionen mit abgespeichert. Mit MipMaps kam man das unter *6.4* genannte Problem der blinkenden Texturen umgehen. Wenn die Textur stark verkleinert dargestellt werden muss, wird einfach eine kleinere Version der Bilder genommen.
Die Filterung zwischen 2 Mipmaps kann in OpenGL auch einstellen.
GL_[NEAREST/LINEAR]_MIPMAP_NEAREST: Es wird einfach die am besten passende Mipmap-Textur genommen.
GL_[NEAREST/LINEAR]_MIPMAP_ LINEAR: Es wird zwischen den zwei am besten passenden Mipmap-Texturen nochmals interpoliert. So vermeidet man teilweise sichtbare Sprünge zwischen 2 Mipmaps wie sie bei NEAREST auftreten können.

Das beste Ergebnis liefert so mit GL_LINEAR_MIPMAP_ LINEAR.

Die glu bietet die Funktion `gluBuild2DMipmaps` zur automatischen Erstellen von Mipmap.

Bei meinen Test waren Mipmap auch schneller, als normale Texturen. Bei Mipmap entfällt ja der Aufwand aus einer großen Textur eine kleine zu berechnen. Aber der Nachteil eines höheren Speicherverbrauchs tritt auf.

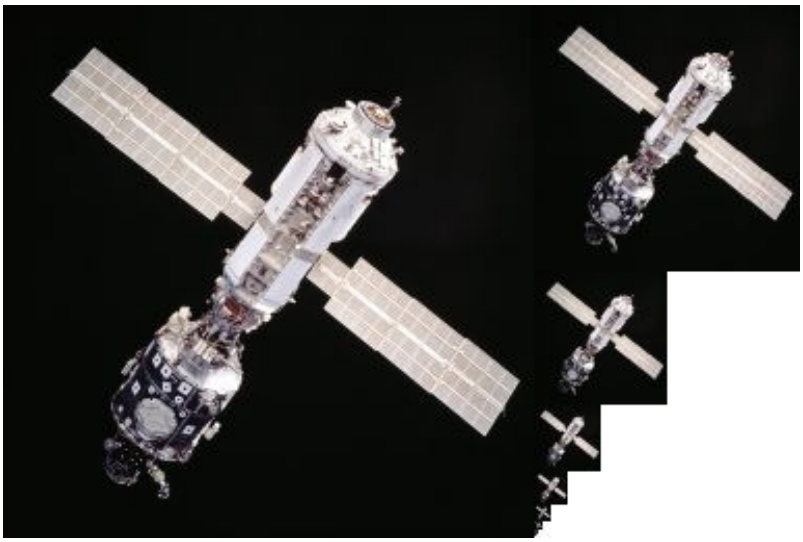


Bild 6.5-1: MipMap-Beispiel (Quelle: vi.)

6.6. Texturen definieren

Zunächst sollte man Texturen aktivieren:
`glEnable(GL_TEXTURE_2D);`
Texturen werden mittels Zahlen-ID unterschieden:

- `void glBindTexture(`*`target`*`:GLenum;`*`texture`*`:GLuint);`
  Erzeugt, wenn noch nicht vorhanden, die Textur mit der ID *texture*. Die Textur *texture* wird zur aktuell genutzten Textur der jeweiligen Dimension. *target* legt die Texturdimension fest (GL_TEXTURE_1D/2D/2D).
- `procedure glGenTextures(n:GLsizei;var textures:GLuint);`
  Erzeugt ein Array noch nicht genutzter Textur-IDs. *n* gibt die Anzahlt der im Array *textures* zu setzenden Werte (ID) an.

Um nun (hier eine 2D) Textur an OpenGL zu übergeben, muss man die Funktion `glTexImage2D()` aufrufen. Die eingestellten Werte bei `glTexParameter()` werden für die Textur genommen.

- `int glTexImage2D(GLenum target, components level, width GLint, GLsizei height, GLint border, type format,GLenum, pointer pixels);`
  Spezifiziert eine 2D-Textur. *target* muss GL_TEXTURE_2D sein. *level* ist die Detailnummer für Mipmaps (0 ist die Basis). *components* gibt die Anzahl der Farbkomponenten in der Textur an. *width* und *height* geben den Höhe und Breite an; Höhe und Breite müssen Maße einer 2er Potenz haben (ab OpenGL 2.0 nicht mehr). *border* Texturrand, *format* gibt die Art der Farbkomponenten an (meistens GL_RGB, GL_RGBA). *type* gibt den Datentyp für die Farbwerte an (meistens GL_UNSIGNED_BYTE). *pixels* ist ein Zeiger auf den Datenblock des Texturarrays. Rückgabewert 0 bei Erfolg.
- `int gluBuild2DMipmaps(GLenum target, GLint components, GLsizei width, GLsizei height, format,type:GLenum; pixels:pointer);`
  Erzeugt alle Mipmaps von *width-height* bis 1-1. *with* und *height* müssen hier nicht unbedingt von der Potenz 2 sein (sie werden passend gemacht). Parameter siehe *glTexImage2D*( ). Rückgabewert 0 bei Erfolg.

Um eine Primitive mit einer Textur zu überziehen, muss man für jeden Vertex die passende Texturkoordinaten angeben. Die Texturkoordinaten kann man mit `glTexCoord()` angeben.

- `void glTexCoord{1/2/3/4}{d/f/i/s}(Typ Parameter);`
  Definiert den Texturpunkt (s,t,r,q). Wenn Parameter nicht angegeben dann sind das die Default werter t=0, r=0, q=1. (0,0) ist die untere linke Ecke, (1,1) die rechte obere Ecke.

6.7. Textur-Beispielprogramm:

Programm: *MipMaps.exe*
Mit A, D, W, S, Q, E kann die Szene rotiert werden. Mit den Cursor-Tasten lässt sich die Szene verschieben.

Das Programm zeigt das Texturwiederholung und den unterschied zwischen normalen und MipMap Texturen. Zum laden von Bilder nutzt es die Bild-Bibliothek devil.dll (http://openil.sourceforge.net/).

Ein Viereck mit Texturkoordinaten:

```
void renderQuad()
{
      glBegin(GL_QUADS);
      glNormal3d(0,0,1);
      glTexCoord2d(10.0,0.0);
      glVertex3f( 1.0f,-1.0f,0.0f);
      glTexCoord2d(0.0,0.0);
      glVertex3f(-1.0f,-1.0f,0.0f);
      glTexCoord2d(0.0,10.0);
      glVertex3f(-1.0f, 1.0f,0.0f);
      glTexCoord2d(10.0,10.0);
      glVertex3f( 1.0f, 1.0f,0.0f);
      glEnd();
}
```

Zeichnen der Viereckes: einmal mit einer Mipmap-Textur (TEX_GRUND1) und einmal ohne (TEX_GRUND2).

```
void renderScene(void)
{
      // Bild und Tiefenwerte löschen,
```

```
	// sonst würden wir in das alte Bild wirder rein malen
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	glBindTexture(GL_TEXTURE_2D, g_Textures[TEX_GRUND1]);
	glPushMatrix();
		glTranslated(-1.1,0,0);
		renderQuad();
	glPopMatrix();
	glBindTexture(GL_TEXTURE_2D, g_Textures[TEX_GRUND2]);
	glPushMatrix();
		glTranslated(1.1,0,0);
		renderQuad();
	glPopMatrix();
	//Puffer-Wechseln
	glutSwapBuffers();
}
```

Texturen aktivieren und laden.

```
void init()
{
	glEnable(GL_DEPTH_TEST);
	//Textur laden
	glEnable(GL_TEXTURE_2D);

	////////////////////////////////////
	//OpenIL-init
	ilInit();
	ilGetError(); //Error-Puffer leeren

	glGenTextures(NUM_TEXTUREN, g_Textures);

	ILuint Imag[NUM_TEXTUREN] ;
	ilGenImages(NUM_TEXTUREN,Imag);
	//////////////
	ilBindImage(Imag[0]);
	Texture(g_Textures[TEX_GRUND1],g_TexPaths[TEX_GRUND1],
		GL_LINEAR_MIPMAP_LINEAR,GL_LINEAR);
	ilBindImage(Imag[1]);
	Texture(g_Textures[TEX_GRUND2],g_TexPaths[TEX_GRUND2],GL_NEAREST
		/*GL_LINEAR*/,GL_LINEAR);
	ilDeleteImages(NUM_TEXTUREN, Imag);
	ilShutDown();
	////////////////////////////////////
}
```

7. Quellen

Die römischen Ziffern im Text beziehen sich auf folgende Quellen:

i. Jackie Neider, Tom Davis, Mason Woo: „*OpenGL Programming Guide*“, Release 1, 1994
ii. http://nehe.gamedev.net
iii. http://www.lighthouse3d.com/opengl/glut/
iv. http://graphics.cs.uni-sb.de/Courses/ws9900/cg-seminar/Ausarbeitung/Philipp.Walter/
v. http://www.opengl.org
vi. http://de.wikipedia.org