Genesis-Projekt: Landschaften rendern mit OpenGL

3D-Landschaft im Eigenbau

3D-Spiele entführen den Spieler in virtuelle Landschaften. Lesen Sie, wie Sie von den **Erfahrungen eines Spieleprogrammierers** profitieren können.

CARSTEN DACHSBACHER

Die Spielewelten von *Drakan* und *Warcraft III* beeindrucken durch ihre Landschaften. Wir führen Ihnen vor, wie Sie diese mit Hilfe von OpenGL darstellen.

Kommerzielle 3D-Engines, die auf die Darstellung von Landschaften ausgerichtet sind, können Sie in vielen Echtzeit-Strategiespielen bewundern. In diesem Projekt lernen Sie alle nötigen Methoden und Algorithmen kennen, um eine vollwertige 3D-Landschafts-Engine zu schreiben.

Landschaften stellen Sie im Allgemeinen mit vielen Polygonen dar, die Sie der 3D-Hardware übergeben. Dazu müssen Sie die zu verwendende 3D-API (OpenGL) kennen und wissen, wie Sie Daten für den 3D-Beschleuniger aufbereiten und an diesen übergeben. Algorithmen setzen Sie ein, um die Polygonlast einzuschränken – selbst wenn Sie eine 3D-Grafikkarte haben, die mit sehr vielen Polygonen pro Sekunde fertig wird. Wenn Sie die 3D-Engine in einem Computerspiel einsetzen, hat der Rechner noch weit mehr zu tun, als nur die Landschaft zu rendern.

DAS GENESIS-PROJEKT

Unser Genesis-Projekt gliedert sich in folgende drei Teile, die Sie von den OpenGL-Grundlagen bis zum Einsatz praxistauglicher Algorithmen führen.

Teil 1: Landschaften rendern mit OpenGL
Teil 2: Eigene Landschaftsdaten generieren
Teil 3: Methoden des Landschafts-Texturierens und Spezialeffekte

Die OpenGL-API

Verknüpfen Sie die Ausgabe von OpenGL mit einem Windows-Fenster. Um es anzulegen, registrieren Sie eine Fensterklasse mit *RegisterClass(CONST WNDCLASS *lpWndClass)* und erzeugen ein Fenster zum Beispiel mit der Funktion *CreateWindowEx(...)*.

Die im Folgenden verwendeten Funktionen befinden sich in *user32.lib* und *opengl32.lib*, die Sie in Ihr Projekt einfügen. Die Definitionen der Funktionsrümpfe stehen in *windows.h* oder *wingdi.h*. Um auf die Client Area eines Windows-Fensters zu zeichnen, benötigen Sie den *Device Context*. Diesen bekommen Sie mit

```
hDC = GetDC( hWND );
```

Für OpenGL brauchen Sie ein bestimmtes Pixelformat für Ihr Fenster: OpenGL benötigt nicht nur einen Speicherbereich für die Farbwerte, sondern zwei Buffers (einer wird dargestellt, der andere solange bearbeitet), zusätzlich einen Z-Buffer, der die Tiefeninformation enthält. Je nach Wunsch und Bedarf können Sie weitere anfordern wie zum Beispiel *Accumulation-* oder *Stencil-*Buffers. Geben Sie beim Pixelformat auch die gewünschte Farbtiefe an. Da Sie die Angaben machen, ohne zu wissen, ob die gerade verwendete Hardware diese unterstützt, müssen Sie die *PIXELFORMATDESCRIPTOR*-Struktur ausfüllen. Damit kann Windows das bestmögliche, vorhandene Pixelformat wählen:

```
static PIXELFORMATDESCRIPTOR pfd
=
{
//Größe PixelFormatDescriptor
 sizeof(PIXELFORMATDESCRIPTOR),
 // Version
 1,
 // Format muß Fenster, OpenGL
 // DoubleBuffering unterstützen
 PFD_DRAW_TO_WINDOW |
 PFD_SUPPORT_OPENGL |
 PFD_DOUBLEBUFFER,
 // RGBA Pixel
 PFD_TYPE_RGBA,
 // Farbtiefe
 bits,
 0, 0, 0, 0, 0, 0,
 // kein Alpha Buffer
 0,
 0,
 // kein Accumulation Buffer
 0,
 0, 0, 0, 0,
 // 16 Bit Z-Buffer
 16,
 // kein Stencil Buffer
 0,
 // kein Auxiliary Buffer
 0,
 PFD_MAIN_PLANE, 0, 0, 0, 0
};

// suche optimales Pixelformat
int PixelFormat =
 ChoosePixelFormat( hDC, &pfd );

//dieses für Device Context:
SetPixelFormat
 ( hDC, PixelFormat, &pfd );
```

Erzeugen Sie einen *OpenGL Rendering Context*: Damit können Sie auf den Device Context Ihres Fensters (also im Fenster) rendern. Dieser Rendering Context verwendet das Pixelformat, das Sie soeben festgelegt haben.

```
HGLRC hRC =
wglCreateContext( hDC );
```

Mit dem folgenden Aufruf aktivieren Sie den Rendering Context. Dabei wirken sich alle OpenGL-Aufrufe Ihres Programms auf die Client Area Ihres Fensters aus.

```
wglMakeCurrent( hDC, hRC );
```

Mit OpenGL-Befehlen können Sie etwas in Ihrem Fenster darstellen. So deaktivieren Sie den Rendering Context:

```
// deaktivieren
wglMakeCurrent( NULL, NULL );
// ... und löschen
wglDeleteContext( hRC );
```

OpenGL verwendet die eingestellte Auflösung und Farbtiefe des Bildschirms. Wenn Sie Ihre OpenGL-Anwendung im Vollbild und nicht im Fenster laufen lassen wollen, stellen Sie die Auflösung und Farbtiefe unter Windows ein, bevor Sie das Fenster erzeugen und den Rendering Context anlegen. Dazu verwenden Sie folgende Codezeilen:

```
DEVMODE screenRes;
memset( &screenRes, 0, sizeof
  ( screenRes ) );
screenRes.dmSize =
  sizeof(screenRes);
```

AUF CD 1

Die Quelltexte sowie die fertig übersetzten Routinen finden Sie im Verzeichnis *Praxis/Programmierung/PC Underground*.

```
// Breite
screenRes.dmPelsWidth = 640;
// Höhe
screenRes.dmPelsHeight = 480;
// Farbtiefe
screenRes.dmBitsPerPel = 32;
screenRes.dmFields =
DM_BITSPERPEL|DM_
➥PELSWIDTH|DM_PELSHEIGHT;
ChangeDisplaySettings
(&screenRes, CDS_FULLSCREEN);
```

TIPP **Überprüfen Sie bei der Initialisierung die Rückgabewerte der Funktionen, und fangen Sie eventuelle Fehler ab. Im Quellcode zu dieser Ausgabe ist das berücksichtigt.**

Geometrische Objekte zeichnen

Nach der erfolgreichen Initialisierung können Sie zeichnen. Dazu müssen Sie in Ihr Programm zwei OpenGL-Header-Dateien einbinden:

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

Bevor Sie zu rendern beginnen, müssen Sie den Colorbuffer, der die Farbwerte enthält, löschen. Dazu legen Sie einmalig die Hintergrundfarbe mit Rot-, Grün-, Blau- und Alpha-Werten fest. Alle Werte liegen im Bereich zwischen *0.0* und *1.0*:

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

Außerdem löschen Sie den Z-Buffer. Dieser enthält für jeden Pixel den Abstand zwischen Betrachter und dem gerenderten Objekt an der entsprechenden Stelle. OpenGL führt beim Rendern diesen Vergleich durch und aktualisiert den Z-Buffer:

```
zeichne Pixel an (x,y) mit Ab-
stand. z
wenn z < als Z-Buffer-Wert bei
(x, y)
  dann setze Pixel im Colorbuffer
setze Z-Buffer bei (x, y) auf
Wert z
```

Genauso wie die Farbe bestimmen Sie einen Wert, mit dem Sie den Z-Buffer beim Löschen füllen:

```
glClearDepth( 1.0 );
```

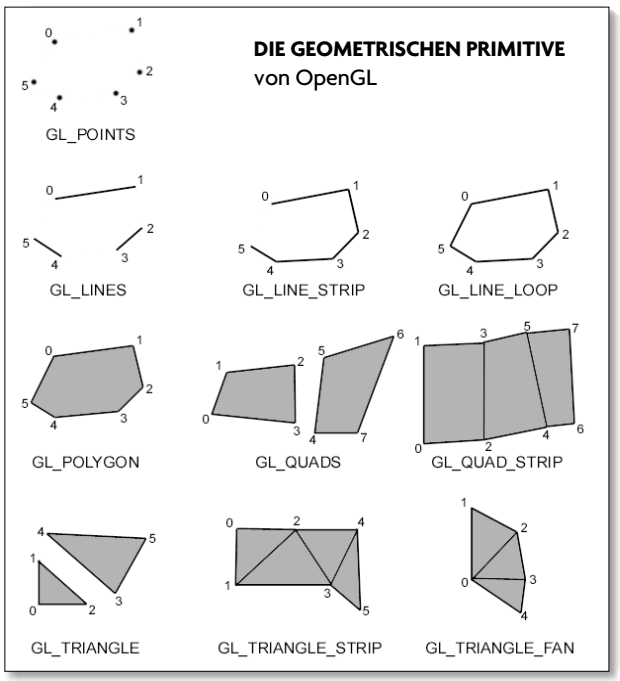
Sie können auch einen anderen Wert wählen und den Vergleich (kleiner als) durch einen anderen ersetzen:

```
glDepthFunc( GLenum func )
```

Bevor Sie einen neuen Frame (Bild) zeichnen wollen, löschen Sie zuvor den Colorbuffer und den Z-Buffer mit

```
glClear( GL_COLOR_BUFFER_BIT
| GL_DEPTH_BUFFER_BIT );
```

In OpenGL gibt es einige Zustandsvariablen, die das Rendern beeinflussen. Einen Zustand setzen Sie mit der Funktion




```
glEnable(...)
```

Sie deaktivieren den Zustand mit

```
glDisable(...)
```

Als Parameter bekommen diese Funktionen vordefinierte Konstanten. Schreiben Sie beispielsweise, um den Z-Buffer zu aktivieren:

```
glEnable( GL_DEPTH_TEST );
```

Wenn Sie geometrische Objekte zeichnen wollen, legen Sie zuerst deren Farbe fest. Dazu stehen folgende Befehle zur Verfügung:

```
glColor{3,4}{b,d,f,i,s,ub,ui,us}
[v]( TYPE colors );
```

OpenGL stammt noch aus der Zeit, als C statt C++ verwendet wurde. In C gibt es keine überladenen Funktionen (namensgleiche Funktionen, die unterschiedliche Typen von Parametern entgegennehmen). Deshalb gibt es für jeden Parametertyp eine eigene Funktion.

So arbeitet die Syntax:

- Der Befehl heißt *glColor*.
- Das nächste Zeichen ist eine *3* oder eine *4* – je nachdem, ob Sie nur Rot-, Grün- und Blau-Werte oder zusätzlich noch einen Alpha-Wert übergeben wollen.
- Eine der Kennungen aus der nächsten geschweiften Klammer gibt den Typ an, zum Beispiel *f* für Float oder *i* für Integer.
- Optional können Sie dem Befehlsnamen noch ein *v* anhängen. Das bedeutet, dass Sie einen Pointer auf *3* oder *4* Variablen Ihres Typs angeben und nicht die Variablen selbst übergeben. Verwenden Sie folgende Variante:

```
glColor3f( rot,
grün, blau );
```

Zeichnen Sie geometrische Primitive: Punkte, Linien, Dreiecke, Vierecke und Polygone. Dazu zählen auch Triangle Strips (Streifen aus Dreiecken). Der Vorteil daran ist, dass Sie weniger Vertices (Knoten- bzw. Eckpunkte) pro Dreieck angeben müssen, als wenn Sie einzelne Dreiecke zeichnen. Damit können Sie die Ausgabe beschleunigen.

Bevor Sie zeichnen, rufen Sie

```
glBegin(...)
```

auf. Als Parameter übergeben Sie eine Konstante, die festlegt, was OpenGL zeichnen soll. Jeden Programmblock, in dem Sie etwas zeichnen, beenden Sie mit einem Aufruf von

```
glEnd(...)
```

Sie haben auch zwei Blöcke, wenn Sie erst Dreiecke und anschließend Linien zeichnen wollen. Innerhalb der Blocks übergeben Sie die Vertices an OpenGL. Dazu verwenden Sie

```
glVertex{234}{sifd}[v]
  ( TYPE coords );
```

Als Beispiel zeichnen Sie ein weißes Dreieck:

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

OpenGL ist eine *State Machine*: Es gibt Zustände, die Sie an- und ausschalten oder ändern und die so lange gelten, bis Sie geändert werden. Einer dieser Zustände ist die Farbe. Im obigen Beispiel sehen Sie einen Aufruf von *glColor**. Wenn Sie für die Eckpunkte jeweils andere Farben einstellen wollen, müssen Sie vor jeden Aufruf von *glVertex** einen *glColor**-Aufruf schreiben.

Wenn Sie alles gezeichnet haben und das neue Bild im Fenster sehen wollen, teilen Sie das OpenGL mit, indem Sie *glFlush()* oder *glFinish()* aufrufen. Vertauschen Sie den Arbeits- und Darstellungs-Colorbuffer Ihres Device-Contexts:

```
SwapBuffers( hDC )
```

Für Sie ist vor allem das Rendern von Polygonen, insbesondere von Drei-

ecken interessant, wenn Sie Landschaften (3D-Objekte) darstellen wollen. Polygone zeichnen Sie für gewöhnlich so, dass Sie die Pixel innerhalb der Polygonkanten setzen. Sie können aber auch nur den Rand oder die Eckpunkte zeichnen lassen. Das können Sie für Vorder- und Rückseite separat angeben:

```
//Vorderseite ausfüllen
glPolygonMode(GL_FRONT,GL_FILL);
// Rückseite, nur Rand
glPolygonMode(GL_BACK,GL_LINE );
```

Wo ist vorne und wo hinten? Die Computergrafik folgt der Konvention, Eckpunkte, die Sie im Polygon von vorne sehen, gegen den Uhrzeigersinn anzugeben. Diesen Test führt OpenGL durch. 3D-Modelle lassen sich so konstruieren, dass alle Polygone Ihrer Außenseite dieser Konvention folgen. Wenn Sie sich Polygongitter in Form einer Kugel vorstellen, sehen Sie immer nur die Polygone, die mit ihrer Vorderseite zu Ihnen zeigen. Alle anderen sind verdeckt und müssen nicht gezeichnet werden. Das können Sie OpenGL überlassen. Legen Sie zuerst fest, ob Sie mit oder gegen die Konvention arbeiten. Beachten Sie, dass Sie diese Entscheidung konsequent durchhalten:

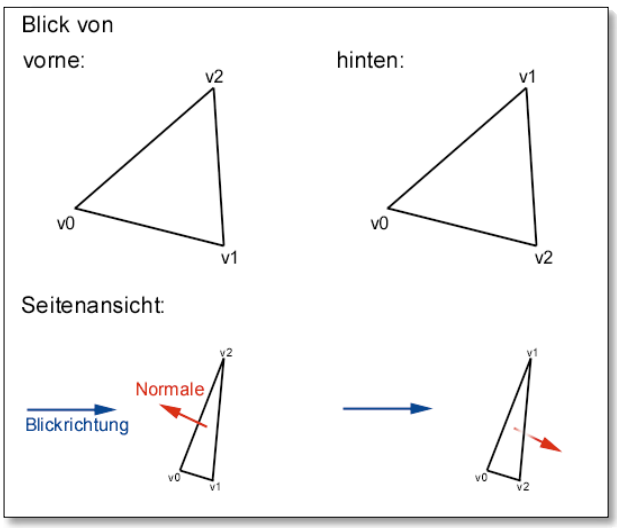
```
// entsprechend der Konvention
glFrontFace( GL_CCW );
// oder entgegen
glFrontFace( GL_CW );
```

Anschließend teilen Sie OpenGL mit, dass es Polygone, deren Rückseite sichtbar sind, nicht zeichnen soll:

```
glCullFace( GL_BACK );
```

Dieses Backface Culling müssen Sie noch aktivieren:

```
glEnable( GL_CULL_FACE );
```



VORNE ODER HINTEN? Die Richtung der Normalen entscheidet: *Backface Culling* in OpenGL.

Transformationen und Kamera-Einstellungen

Mit den bisher verwendeten Befehlen können Sie zwar geometrische Primitive zeichnen, diese befinden sich aber noch nicht im dreidimensionalen Raum. Bei 3D-Grafiken geben Sie nicht nur die 3D-Objekte in Form der Primitive an, sondern müssen noch andere Parameter festlegen. Dazu gehören Lage und Position der Objekte und die Einstellungen (Transformationen) der virtuellen Kamera, mit der Sie die Szene fotografieren.

Transformationen beschreiben Sie mathematisch mit einer Matrix. Eine Matrix ist zunächst eine Tabelle mit Zahlenwerten. Wenn Sie diese Werte geeignet wählen, beschreiben die Matrizen eine Drehung oder eine Projektion, wenn Sie mit einem Vektor (zum Beispiel einem Vertex) multipliziert werden. Sie können die Transformationen hintereinander ausführen, indem Sie das Ergebnis einer Transformation als Vektor für die nächste verwenden. Weniger Rechenzeit kostet es, die Matrizen miteinander zu multiplizieren. Als Ergebnis erhalten Sie eine Matrix, die alle Transformationen zusammen beschreibt. Diese Matrix-Matrix- und Matrix-Vektor-Multiplikationen übernimmt OpenGL für Sie.

Im Bild oben sehen Sie die Transformationsschritte, die ein Vertex durchlaufen muss, den Sie angeben, bis die 2D-Koordinaten innerhalb des Fensters berechnet werden. Dort tauchen die Begriffe *ModelView-* und *Projection-*Matrix auf. Diese beiden Matrizen verwaltet OpenGL in einem Stack. Sie übernehmen die Funktionen einer echten Kamera:

- Das Aufstellen der Kamera in der Szene entspricht der *Viewing Transformation*.
- Sie platzieren die Objekte der *Modelling Transformation*.
- Die Wahl der Kameralinse und den Zoomfaktor steuern sie über die *Projection Matrix*.
- Die *Viewport Transformation* skaliert die projezierten Koordinaten auf die Größe des Windows-Fensters.
- Die *Modelling-* und *Viewing-Transformation* zusammen ergeben die *ModelView Matrix*.

Indem Sie die Matrizen in einem Stack verwalten, können Sie hierarchische 3D-Modelle einfacher gestalten und eleganter mit Matrizen umgehen. Aus einem Stack können Sie Elemente *pushen* (hinzufügen) und *poppen* (herunternehmen). Wenn Sie den *Push*-Befehl

```
glPushMatrix()
```

aufrufen, wird die oberste Matrix auf dem Stack kopiert und nochmals auf den Stack gepackt.

```
glPopMatrix()
```

entfernt die oberste Matrix. Der Befehl

```
glLoadIdentity()
```

überschreibt die oberste Matrix mit der Einheitsmatrix. Nun stellen Sie eine virtuelle Kamera auf, wobei der Befehl

```
gluPerspective(...)
```

Ihnen viel Arbeit abnimmt. Wählen Sie die *Kameralinse*:

```
// Projection Matrix wählen
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
// Öffnungswinkel, Aspectratio..
gluPerspective
( 45.0f, width/height,
1.01f, 1000.0f );
// Breite und Höhe des Fensters
glViewport( 0,0,width,height );
```

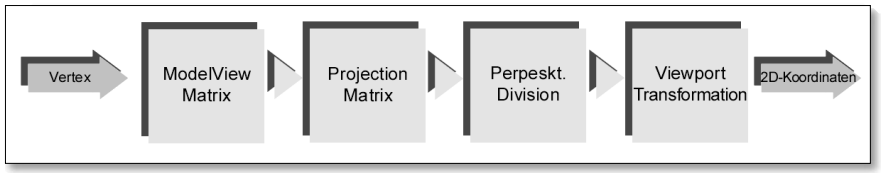
Beschreiben Sie die 3D-Szene. Platzieren Sie die Kamera:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
// Abstand vom Objekt
glTranslatef
 ( 0.0f, 0.0f, -80.0f );
```

Der Befehl

```
glTranslatef(...)
```

erzeugt eine Matrix, die eine Verschiebung (eine Translation) enthält, und multipliziert die oberste Matrix auf dem Stack mit dieser Matrix.



DIE TRANSFORMATIONEN, die ein Vertex über sich ergehen lassen muss

TIPP ACHTUNG! Die Reihenfolge der Transformationen, die ein Vertex durchläuft, ist umgekehrt zu der Reihenfolge, die Sie im Programmcode angeben.

Es gibt noch weitere Befehle für Transformationen:

```
//Winkeldrehung um x, y, z
glRotatef
( GLfloat angle, GLfloat x,
 GLfloat y, GLfloat z )
```

und

```
//Skalierung in x, y, z Richtung
glScalef( GLfloat x, GLfloat y,
  GLfloat z );
```

Das folgende Beispiel zeigt die Reihenfolge der Transformationen auf:

```
glPushMatrix();
glTranslatef( 2, 0, 0 );
glRotatef( 45, 1, 1, 0 );
drawCube();
glPopMatrix();
```

Damit drehen Sie den Würfel zuerst und verschieben ihn dann. Achten Sie auf die umschließenden Befehle

```
glPushMatrix()
...
glPopMatrix()
```

Diese gewährleisten, dass der Matrixstack nach dem Zeichnen und Transformieren des Würfels genauso wie vorher hinterlassen wird. Das ist wichtig, wenn Sie viele 3D-Objekte unabhängig voneinander bewegen wollen.

Licht an!

Nun können Sie Ihre 3D-Objekte und Ihre virtuelle Kamera beliebig im Raum platzieren. Leider erscheinen die Objekte noch einfarbig und fade. Es fehlt Licht!

In OpenGL gibt es drei Arten von Licht:

- Das ambiente Licht ist der Teil des Lichts, der durch die Umgebung so sehr gestreut ist, dass seine Richtung nicht mehr auszumachen ist.
- Das diffuse Licht ist deutlich heller. Es kommt von einer Lichtquelle, trifft auf eine Oberfläche und wird dort in alle Richtungen gestreut.
- Die Specular (spiegelnde) Reflexion entsteht etwa dadurch, dass Sie ein Spiegelbild der Lichtquelle wahrnehmen.

In OpenGL geben Sie für jede Lichtquelle, wovon Sie mehrere gleichzeitig verwenden können, Position, Farbe und Intensität des emittierten Lichts an. Dazu verwenden Sie

```
glLight{i,f}[v]
( GLenum light, GLenum pname,
  TYPE param );
```

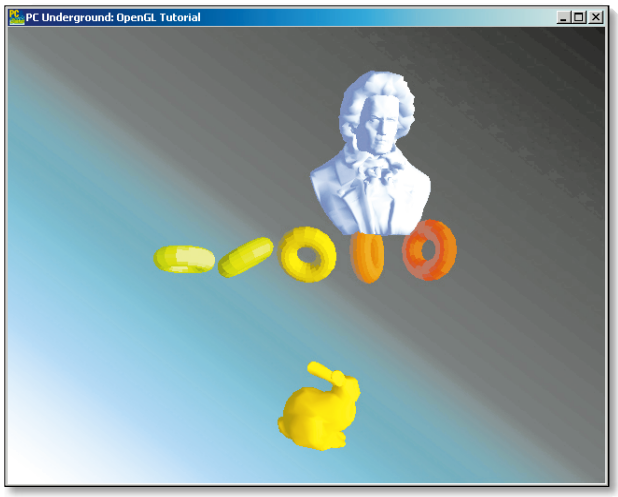
Als Beispiel setzen Sie eine Lichtquelle ein:

```
// Parameter setzen
GLfloat lAmbient[] =
 {0.0, 0.0, 0.0, 1.0};
GLfloat lDiffuse[] =
 {1.0, 1.0, 1.0, 1.0};
GLfloat lSpecular[] =
 {1.0, 1.0, 1.0, 1.0};
GLfloat lPosition[] =
 {1.0, 1.0, 1.0, 0.0};

glLightfv(GL_LIGHT0,GL_AMBIENT,
 lAmbient);
glLightfv(GL_LIGHT0,GL_DIFFUSE,
 lDiffuse);
glLightfv(GL_LIGHT0,GL_SPECULAR,
 lSpecular);
glLightfv(GL_LIGHT0,GL_POSITION,
 lPosition);

// Beleuchtungsberechnung an
glEnable( GL_LIGHTING );
// diese Lichtquelle anschalten
glEnable( GL_LIGHT0 );

// Schattierung der Polygone
glShadeModel( GL_FLAT );
```



UNSER BEISPIELPROGRAMM in Aktion: beleuchtete, animierte 3D-Modelle mit Flat- oder Gouraud Shading

Zur Beleuchtungsberechnung braucht OpenGl außer den Koordianten der Eckpunkte der 3D-Objekte noch die Oberflächennormalen.

Die Oberflächennormalen erhalten Sie entweder aus einem 3D-Modelling-Programm, wenn Sie damit Ihre 3D-Objekte gestalten, oder Sie berechnen sie selbst. Dazu verwenden Sie das Kreuzprodukt zweier Vektoren. Für ein Dreieck berechnen Sie die Normale wie folgt:

```
// Die Vertices des Dreiecks
GLfloat v0 ={ 0.0f, 0.0f, 0.0f }
GLfloat v1 ={ 1.0f, 1.0f, 0.0f }
GLfloat v2 ={ 3.0f, 0.0f, 1.0f }

GLfloat a[ 3 ], b[ 3 ], n[ 3 ];

a[0]=v1[0]-v0[0];
a[1]=v1[1]-v0[1];
a[2]=v1[2]-v0[2];

b[0]=v2[0]-v0[0];
b[1]=v2[1]-v0[1];
b[2]=v2[2]-v0[2];

// Normale n
n[0] =a[1] * b[2] - b[1] * a[2];
n[1] =a[2] * b[0] - b[2] * a[0];
n[2] =a[0] * b[1] - b[0] * a[1];
```

Die Normale übergeben Sie an OpenGL mit dem Befehl

```
glNormal3{b, d, f, i, s}
 [v]( TYPE coords );
```

In unserem Fall verwenden Sie die Parameter

```
glNormal3fv( n )
```

Eine angegebene Normale wird so lange jedem Vertex zugewiesen, bis Sie eine andere angegeben haben (genau wie bei *glColor**). Sie können auch für jeden Vertex eine Normale angeben. Diese Vertex-Normalen erhalten Sie, indem Sie die Normalen aller Dreiecke, an denen ein Vertex beteiligt ist, mitteln. Damit erhalten Sie unter OpenGL das *Gouraud Shading*: Diese berechnet die Beleuchtung an jedem Eckpunkt eines Dreiecks und interpoliert die Farbwerte linear. Damit wirken die Objekte visuell runder als bei dem *Flat Shading*. Das *Gouraud Shading* aktivieren Sie mit *glShadeModel (GL_SMOOTH)*.

Wenn Sie 3D-Objekte beleuchten wollen, müssen Sie den Oberflächen ein Material zuweisen. Dahinter verbirgt sich *nur* eine Farbe – aber jeweils eine für den ambienten, diffusen und spekulären Teil des Lichts. Ein Material definieren Sie wie folgt:

```
GLfloat mAmbient[] =
 { 1.0f, 1.0f, 1.0f };
GLfloat mDiffuse[] =
 { 1.0f, 1.0f, 0.0f };
glMaterialfv( GL_FRONT,
 GL_AMBIENT, mAmbient );
glMaterialfv( GL_FRONT,
 GL_DIFFUSE, mDiffuse );
```

In der nächsten Ausgabe werden Sie erste Landschaftsdaten generieren und darstellen. ET

Literatur

Jackie Neider, Tom Davis, Mason Woo: OpenGL Programming Guide, The Official Guide to Learning OpenGL, Release 1

Burg, Haf, Wille: Höhere Mathematik für Ingenieure, Band 2 Lineare Algebra, ISBN 3-519-22956-0

Bronstein, Semendjajew, Musiol, Mühlig: Taschenbuch der Mathematik, ISBN 3-8171-2002-8