AUF DER HEFT-CD

Demo-Programmierung unter Windows 95/98

Effektzauberei mit MP3

Für den weitverbreiteten MP3-Player Winamp schreiben Sie zwei Plugins und lernen nebenbei noch ein wenig OpenGL.

CARSTEN DACHSBACHER/
NILS PIPENBRINCK

Musik aus dem Internet ist in: MP3-Dateien genießen große Popularität, ebenso Winamp, der Player schlechthin für dieses Dateiformat. Dank verschiedener Skins (Erscheinungsbilder) können Sie das Aussehen dieses Programms individuell anpassen. Skins sind jedoch nicht der einzige Weg, Winamp ganz nach Ihrem Geschmack auszustatten. Sie können auch Plugins laden, die zum Beispiel grafische Effekte passend zur Musik zeigen. Dabei liefert Winamp alle Daten, das heißt die aktuelle Ausgangsspannung des Verstärkers und die Amplitude der Frequenzspektren. Das Plugin muß sich nur um die Darstellung kümmern.

Wie Sie solche Plugins ohne großen Aufwand selbst schreiben, zeigen wir Ihnen in dieser Ausgabe. Zuerst programmieren wir ein einfaches, aber eindrucksvolles Plugin unter DirectDraw. Im zweiten Plugin kommt zusätzlich OpenGL zum Einsatz.

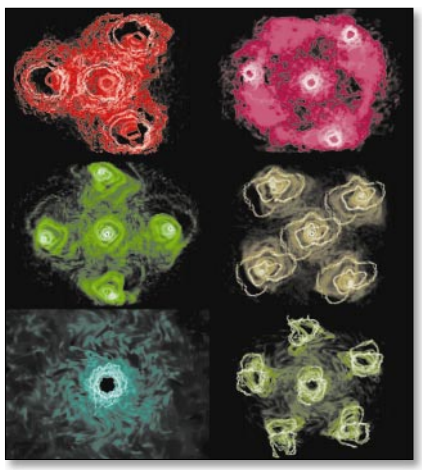
Die Struktur eines Plugins

Die Visualisierungs-Plugins für Winamp sind keine gewöhnlichen Programme, sondern DLL-Dateien (Dynamic Link Libraries). Deren Aufbau ist in diesem Fall jedoch einfach. Jedes Plugin kann aus mehreren Modulen bestehen, wobei die Programmierer von Winamp mit Modul einen Grafikeffekt meinen.

Um eine reibungslose Zusammenarbeit mit dem MP3-Player zu gewährleisten, füllen Sie für jedes dieser Module eine Struktur aus. Darin steht der Name des Moduls, welche Daten es braucht und welche Routinen aufzurufen sind:

```
winampVisModule PCUModul =
{
  PluginName,  // Modul-Name
  NULL, //* Fenster-Handle
        //  von Winamp
  NULL, //* DLL Instance Handle
  0, //* Sampling-Rate
  0, //* Anzahl der Channels
     //  (1=Mono, 2=Stereo)
  0, // Ausgabe-Latenz
  0, // Verzögerung der
     // Grafik-Ausgabe
  0, // Keine Spektrum-Analy-
     // ser-Daten anfordern
  2, // Stereo-Waveform-Daten
     // anfordern
  { 0, }, //* Spektrumdaten
  { 0, }, //* Waveform-Daten
  config, // Konfigurations-
          // Routine
  init, // Initialisierungs-
          // Routine
  render, // Berechnungs-
          // Routine
  quit // Beendigungsroutine
};
```

Die mit einem Sternchen (*) im Kommentar gekennzeichneten Felder füllt Winamp aus. Um die übrigen kümmern Sie sich selbst.



SECHS MOMENTAUFNAHMEN des ersten Winamp-Plugins zeigen farbenfrohe Zufallsmuster.

Wichtig sind vor allem die letzten vier Felder mit den Programmroutinen.

Die *config*-Funktion wird aufgerufen, wenn der Benutzer bei der Auswahl Ihres Plugins den Button *Configure* drückt. Dort können Sie sich einen Dialog anzeigen lassen, über den der Anwender die Effekte nach seinen Wünschen anpassen kann.

Das Beispiel-Plugin kommt ohne einen solchen Dialog aus, es zeigt hier statt dessen eine kleine Infobox:

```
void config(struct
  winampVisModule *this_mod)
{
  MessageBox(this_mod->
    hwndParent,
    „PCU Winamp Plugin“,
    „About“, MB_OK);
}
```

Als Parameter wird immer ein Zeiger auf Ihre Modulstruktur übergeben. Dies wird wichtig, wenn Sie in Ihrem Plugin mehrere Module implementieren wollen, die alle den gleichen Konfigurationsdialog oder die gleiche Initialisierungs- und Beendigungsroutine benutzen.

Der Initialisierungscode

```
int init(struct
  winampVisModule *this_mod)
```

wird gleich nach dem Start des Plugins aufgerufen. Unser Plugin öffnet in dieser Routine ein einfaches Fenster und startet dann DirectDraw im Fullscreen-Modus.

Der Initialisierungscode ähnelt dem aus der Grafikbibliothek, die Sie aus früheren Ausgaben von PC Underground kennen. Im Unterschied zur Grafikbibliothek müssen Sie jetzt aber nicht dafür sorgen, daß Ihre Berechnungsroutine regelmäßig aufgerufen wird. Das erledigt Winamp für Sie.

Die Beendigungsroutine

```
void quit(struct
  winampVisModule *this_mod)
```

ist das nötige Gegenstück zur *init*-Funktion. Wenn das Plugin verlassen wird, beendet diese Funktion DirectDraw und schließt das erzeugte Fenster wieder.

Dazwischen erledigt die Render-Routine

```
int render(struct
  winampVisModule *this_mod)
```

die eigentliche Arbeit. Während die Musik läuft, wird sie regelmäßig von Winamp aus aufgerufen.

Um Ihre Ideen zur Visualisierung in Pixel umsetzen zu können, haben Sie hier Zugriff auf einige interessante Daten. Winamp füllt die Modulstruktur mit den Informationen, die Sie angefordert haben. So nutzen Sie entweder das Frequenzspektrum der Musik oder die Sample-Daten – oder auch beides gleichzeitig, wenn Sie möchten.

Im Array *waveformData* finden Sie die jeweils aktuellen 576 Sample-Werte. Diese können Sie zur Anzeige eines Oszilloskops verwenden. Das Array *spectrumData* hingegen enthält das aktuelle Frequenzspektrum und ist ebenfalls 576 Einträge lang. Die Baßanteile befinden sich dabei in den unteren Werten, während die höchste Frequenz bei Element *576* zu finden ist.

Da Sie möglicherweise Stereodaten vorliegen haben, sind die Arrays zweidimensional. Die Samples für den linken Kanal finden Sie in *waveformData[0][i]*, die für den rechten in *waveformData[1][i]*. Das gleiche gilt analog für das Array *spectrumData*.

576 Sampling-Werte sind nicht besonders viel. Bei einer Wiedergabefrequenz von 44 100 Hz, die Sie bei CD-Qualität erreichen, entsprechen die übergebenen Werte einem Zeitfenster von etwa 13 Millisekunden. Daher sollte auch Ihr Effekt nicht viel Rechenzeit kosten.

Ist Ihre Rendering-Funktion zu langsam, verpassen Sie einen Teil der Daten, und Ihr Plugin verliert an Genauigkeit. Dies ist zwar nicht sonderlich schlimm, aber Ihr Plugin kann dabei einen Teil seines optischen Reizes verlieren.

Die Schnittstelle zu Winamp

Jetzt informieren Sie Winamp darüber, welche Module Sie in Ihrem Plugin programmiert haben. Dafür brauchen Sie zwei Funktionen.

- *getModule* ist eine sogenannte Callback-Funktion. Winamp wird sie mehrfach aufrufen und die DLL fragen, welche Module verfügbar sind. Da Sie zur Zeit nur ein Modul haben, fällt sie relativ einfach aus:

```
winampVisModule
  *getModule(int which)
{
  switch (which)
  {
    case 0: return &PCUModul;
    default: return NULL;
  }
}
```

Möchten Sie ein zusätzliches Modul programmieren, erweitern Sie lediglich das *switch*-Statement um den Fall *1*. Das dritte Modul erhält die Nummer *2* usw.

- Die zweite Funktion ist sehr viel interessanter: Sie ist der Einsprungspunkt der DLL. In etwa entspricht sie der *main()*-Funktion eines normalen C-Programms. Beim Laden von Winamp werden auch alle installierten Plugins geladen und diese Einsprungroutinen aufgerufen. Dabei geben diese den Namen des Plugins, die Versionsnummer und einen Pointer auf die *getModule*-Funktion zurück. Winamp fragt dann Informationen über die Module ab und läßt die Plugins bis zu ihrer Aktivierung erst einmal ruhen.

```
extern „C“ __declspec(dllexport)
  winampVisHeader
  *winampVisGetHeader()
{
  static winampVisHeader
    PluginHeader;
  //Felder der Header-
  //Struktur ausfüllen
  PluginHeader.description =
    PluginName;  // Name
  PluginHeader.version =
    VIS_HDRVER;  // Version
  PluginHeader.getModule =
    getModule;
  // getModule-Funktion
  return &PluginHeader;
}
```

Damit Winamp diese Funktion in Ihrer DLL findet, muß sie mit einem bestimmten Namen exportiert werden. Die Anweisung *extern „C“* sorgt dafür, daß Ihr C++-Compiler den Namen der Funktion nicht ändert. Bei C++ ist es nämlich in der Regel so, daß die Parameter und Rückgabetypen in den internen Namen codiert werden. Da es leider keinen allgemeinen Standard für diese Codierung gibt, programmieren Sie bei DLLs exportierte Funktionen im „C“-Standard.

Der Zusatz *__declspec(dllexport)* sorgt schließlich dafür, daß die Funktion in die sogenannte Exporttabelle der DLL aufgenommen wird. Lediglich exportierte Funktionen sind von außen zu sehen. Sie können dabei auch mehr als eine Routine exportieren.

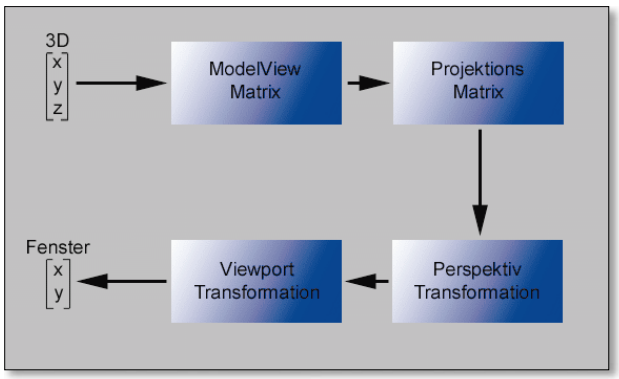
Die Kommunikation zwischen den Plugins und Winamp erscheint am Anfang vielleicht etwas verwirrend – aber wenn Sie sich damit etwas näher beschäftigen, werden Sie schnell die Annehmlichkeiten dieser Methode zu schätzen wissen.

Ein erstes Plugin

Als Einstieg in die Plugin-Programmierung wählen Sie zunächst einen einfachen Effekt. Damit Sie dabei nicht auf tolle optische Reize verzichten müssen, wenden Sie die sogenannte Movelist-Technik in einer verfeinerten Variante an.

Bei Movelists legen Sie – wie der Name schon sagt – eine Tabelle an, die für jedes Pixel eine neue Position angibt. Das ist noch nichts Neues. Daher erweitern Sie die Movelist so, daß sie mit höherer Genauigkeit arbeitet. Auch benutzen Sie keine Textur, sondern wenden die Movelist immer auf das vorherige Bild an. Diese sehr beeindruckenden Effekte, bei denen ein Ergebnis wieder in die Berechnung des nächsten einfließt, nennt man Feedbacks.

Für jeden Punkt auf dem Bildschirm brauchen Sie zwei Tabelleneinträge, die angeben, von welcher Position der Punkt kopiert werden soll. Wegen der erhöhten Genauigkeit speichern Sie diesen Wert in einem 32-Bit-Integer-Wert. Die oberen 16 Bit geben direkt einen Teil der Koordinate an, während die unteren 16 Bit festlegen, an welcher Position „zwischen“ den Pixeln kopiert werden soll. Sie können natürlich nicht zwischen zwei Speicherstellen lesen, deshalb simulieren Sie dies mit Hilfe der bilinearen Interpolation.



MEHRERE HINTEREINANDERGESCHALTETE Transformationen bilden einen Punkt von 3D nach 2D ab.

Im Movelist-Array werden x- und y-Koordinate jeweils nacheinander abgelegt. Das ist sinnvoll, da Sie beide Koordinaten benötigen und alle Punkte des Bilds nacheinander berechnen. Der leicht vereinfachte Code zum Zeichnen des Feedbacks sieht so aus:

```
int lerp (int a, int b, int x)
```

```
// Lineare 16-Bit Interpolation
{
  return a + (((b-a)*x)>16);
}
```

Zunächst erfolgt die lineare Interpolation zweier Werte *a* und *b*. Die Funktion *lerp* liefert einen Wert zwischen *a* und *b* zurück. Wo genau das Ergebnis liegt, hängt von dem Wert *x* ab. Ist *x* gleich *0*, bekommen Sie *a*. Ist *x* gleich *65 536*, erhalten Sie *b* zurück. Alle anderen Belegungen von *x* liefern Werte zwischen *a* und *b*.

```
long * source = movelist1;
for ( int i=0; i<width*height;
  i++)
{
  int x  = *source++;
  int y  = *source++;
  int offset = (x>16)+
    width*(y>16);

  int a = lerp(buffer1[offset],
    buffer1[offset+1],x&0xffff);
  int b = lerp(buffer1[offset+
    width],buffer1[offset+1+
    width],x&0xffff);
  buffer2[i] = lerp (a,b,
    y&0xffff);
}
```

Dies ist die Hauptschleife des Movelist-Feedbacks. Darin lesen Sie zunächst die Quellkoordinaten *x* und *y* aus der Movelist aus. Die Pixeladresse *offset* berechnen Sie aus den oberen 16 Bit der Koordinaten.

Dann interpolieren Sie zwischen den Pixeln des Bilds in *buffer1*. Da Sie eine zweidimensionale bilineare Interpolation brauchen, rufen Sie die *lerp*-Funktion mehrfach auf. Das interpolierte Ergebnis schreiben Sie schließlich in das neue Bild bei *buffer2*.

Mit diesen wenigen Zeilen Code können Sie jetzt Bilder um Bruchteile von Pixeln verschieben, drehen, verzerren, vergrößern und stauchen. Das Resultat hängt nur davon ab, was Sie in Ihre Movelist schreiben.

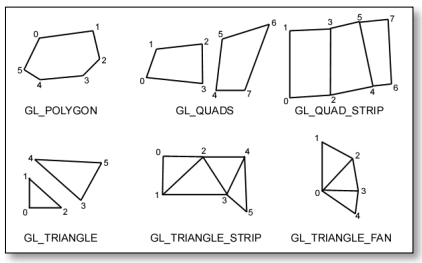
Die Effekt-Movelist

Bei Feedbacks sollten Sie die Bewegung nicht zu schnell laufen lassen. Nur so kommt der Effekt voll zur Geltung. Als kleine Anregung hier eine kombinierte Rotation und Vergrößerung:

```
double sinval=sin(0.01)*1.01;
double cosval=cos(0.01)*1.01;
```

Hier berechnen Sie die Rotationswerte vor. Der Winkel beträgt 0,01 rad, was in etwa 0,6 Grad entspricht. Die Multiplikation mit 1,01 sorgt für einen Zoom-Wert von einem Prozent.

```
long * dest = movelist1;
for (int py=0; py<height; py++)
for (int px=0; px<width; px++)
{
  double x = (double)
    (px-(width/2))/(width/2);
  double y = (double)
    (line-(height/2))/(height/2);
```



DAS ZWEITE PLUGIN VERWENDET verschiedene Polygon-Primitive, hier mit ihren Bezeichnungen in OpenGL.

Diese Zeilen skalieren die Koordinaten *px* und *py* in den Bereich von *-1* bis *1*. Dadurch werden die Berechnungen unabhängig von der Breite und Höhe der Movelist.

```
  double xx = x * cosval -
    y * sinval + 1.0;
  double yy = y * cosval +
    x * sinval + 1.0;
  xx = Clamp(xx*width /2.0,
    width-1,  1);
  yy = Clamp(yy*height/2.0,
    height-1, 1);
```

Der Punkt wird nun mit den vorberechneten Werten *xx* und *yy* rotiert und danach wieder auf Bildgröße skaliert. Die Funktion *Clamp* sorgt dafür, daß die rotierten Werte nicht den Bildbereich verlassen. Sonst würde Ihre Feedback-Routine beim Auslesen der Pixel unweigerlich abstürzen.

```
    *dest++=(long)(xx*65536.0);
    *dest++=(long)(yy*65536.0);
}
```

Zuletzt schreiben Sie die Koordinaten in die Movelist. Zuerst kommt die x-, dann die y-Koordinate. Die Multiplikation mit der Konstanten 65 536 sorgt für die Aufteilung in eine 16-Bit-Koordinate und in eine 16-Bit-Subkoordinate. Das Schreiben der Koordinaten mit dem Befehl **dest++=* sieht etwas ungewöhnlich aus. Es funktioniert, weil *dest* ein Pointer ist. Nachdem der Wert an die entsprechende Adresse geschrieben wurde, erhöht die Operation ++ den Pointer, der dann auf das nächste Element zeigt. Bei dieser Vorgehensweise sparen Sie eine Variable, und der Compiler kann möglicherweise effizienteren Code erzeugen.

Im Beispielcode haben wir noch etwas mehr Aufwand getrieben, um den Effekt wilder zu gestalten. Wie Sie bemerken werden, haben wir einfach mehrere Rotationen übereinandergelegt.

Wenn Sie Ihr Plugin so starten, sehen Sie noch nichts. Denn es fehlt noch der Code, der die Sample- oder Spektrumdaten benutzt, um dem Feedback brauchbare Bilder zu liefern. Zeichnen Sie einfach die Samples als Wellenform im Kreis über das aktuelle Bild. Dies können Sie ähnlich wie die Funktion *Movelist_Draw()* machen.

Wir haben es uns nicht nehmen lassen, noch einige Extras in das Plugin einzubauen. So können Sie das Aussehen mit den Cursor-Tasten verändern. Das Bild auf S. 212 zeigt einige psychedelisch wirkende Schnappschüsse des Plugins.

Einführung in OpenGL

OpenGL (Open Graphics Library) ist ein Standard der Computerindustrie für 3D-Grafik. Er stammt ursprünglich von der internen Grafikbibliothek von Silicon Graphics (SGI) und wird jetzt von SGI, Microsoft, IBM, Intel und DEC weiterentwickelt. Die Vorteile von OpenGL sind die genaue Spezifikation des Standards – er arbeitet gleichermaßen unter Betriebssystemen wie Windows, Unix sowie Linux – und die Unterstützung durch 3D-Hardware.

Es ist ganz einfach, mit OpenGL beeindruckende 3D-Grafiken zu pro-

KOMPILIEREN UND INSTALLIEREN

Da es sich bei den Plugins um keine *exe*-Dateien handelt, weicht der Kompiliervorgang etwas von der üblichen Vorgehensweise ab. Je nachdem, welchen Compiler Sie benutzen, sind einige Änderungen nötig.

Oftmals genügt es, bei den Linker-Einstellungen als Zieldatei den Typ *DLL* statt *EXE* auszuwählen. Benutzer von Watcom C++ müssen wir diesmal leider enttäuschen: Der Compiler ließ sich bei unseren Tests nicht dazu bewegen, funktionierende Plugins zu liefern. Da die Beispielcodes DirectDraw benutzen, müssen Sie auch die Bibliotheken *ddraw.lib* und *dxguid.lib* mit einbinden.

Die erzeugten DLLs müssen alle mit dem Namen *vis_* beginnen. Winamp sucht nur nach Plugin-Dateien, die dieser Konvention entsprechen.

Um das Plugin zu installieren, kopieren Sie es lediglich in das Plugin-Verzeichnis innerhalb der Winamp-Installation. Über die Tastenkombination *[Umschalt fest-Strg-K]* wählen Sie ein Plugin und starten es.

grammieren. Dabei lernen Sie die Funktionen kennen, die Sie für ein Winamp-Plugin brauchen.

OpenGL stellt Ihnen Funktionen zum Zeichnen von Primitiven, also von Punkten, Linien und Polygonen, zur Verfügung. Es gibt auch Support-Routinen, mit denen Sie Kurven, Bézier-Oberflächen oder Text darstellen können. Die Polygonprimitive können Sie dabei mit Texture Mapping und Schattierung ausstatten.

Sobald Sie eine 3D-Szene aus Primitiven aufgebaut haben, definieren Sie Beleuchtungseffekte, das Blickfeld und Spezialeffekte wie Nebel oder Transparenz. OpenGL erledigt dann den Rest für Sie: die Schattierung, das perspektivische Rendering, das „Wegwerfen" der nicht sichtbaren Polygone (Hidden Surface Removal) und das Clipping. Wenn Sie das Blickfeld oder die Beleuchtung ändern oder die definierten Objekte bewegen, berechnet OpenGL alles für Sie neu.

OpenGL ist als eine sogenannte State Machine implementiert: Das heißt, daß ein festgelegter Zustand (zum Beispiel eine Farbe) so lange aktuell ist, bis Sie ihn wieder ändern. Solange also beispielsweise die aktuelle Zeichenfarbe Rot ist, erhalten alle definierten Primitive diese gesetzte Farbe. Zudem ist OpenGL dafür konzipiert, in einem Client-Server-Modell zu arbeiten. Client und Server können natürlich auch in einem Rechner vereint sein, so wie es bei uns der Fall ist.

OpenGL-Programmierung

Zuerst einmal bringen Sie OpenGL dazu, in ein Windows-Fenster zu rendern. Da OpenGL plattformunabhängig ist, stellt Microsoft die Befehle dazu zur Verfügung. Dabei registrieren Sie wie bei einem normalen Windows-Programm eine Fensterklasse, öffnen ein Fenster und erzeugen mit folgenden Befehlen einen sogenannten OpenGL-Kontext:

```
// Pixelformat des Windows-
// Bildschirms lesen
int FormatIndex =
  ChoosePixelFormat(
  WindowDC,&FormatDescriptor);
SetPixelFormat(WindowDC,
  FormatIndex,&FormatDescriptor);
// OpenGL Kontext erzeugen
WindowRC =
  wglCreateContext(WindowDC);
wglMakeCurrent(WindowDC,
  WindowRC);
```

Das ist schon alles. Diese Zeilen dienen nur als Beispiel, die vollständige Routine finden Sie im Quelltext des OpenGL-Plugins.

Nun können Sie schon mit der Beschreibung der 3D-Szene beginnen. Im Bild auf S. 213 sehen Sie, welche Transformationen ein Punkt im Raum – gegeben durch (x,y,z) – durchläuft, bis er die Bildschirmkoordinaten (x,y) erhält. Beachten Sie dabei, daß x und y im Raum und im Fenster verschieden sind.

Alle Transformationen in OpenGL beschreiben Sie mit Matrizen. Sie können Ihr Mathematikbuch aber ruhig in der Ecke lassen, denn Matrizen werden hier abstrakt behandelt.

Stellen Sie sich eine Matrix einfach als ein mathematisches Gebilde vor, das eine Verschiebung, Drehung oder Skalierung darstellt. Verschiedene Matrizen können Sie aufeinander anwenden und so eine einzige Matrix errechnen, die alle dabei durchgeführten Transformationen enthält.

Für das OpenGL-Plugin beschreiben Sie zunächst eine Projektionsmatrix, die sowohl die Perspektiv- als auch die Viewport-Transformation beschreibt. Bei der Perspektivprojektion kann es sich zum Beispiel um eine Zentral- oder Orthogonalprojektion handeln. Mit der Bezeichnung Viewport ist das Fenster gemeint – die entsprechende Transformation bildet also die einzelnen Punkte auf die Bildebene ab.

Die in dieser Matrix gespeicherten Transformationen stellen quasi das Kameramodell dar. Nun teilen Sie OpenGL mit, daß Sie die Projektionsmatrix bearbeiten möchten:

```
glMatrixMode(GL_PROJECTION);
```

Danach laden Sie die Einheitsmatrix, das neutrale Element beim Arbeiten mit Matrizen:

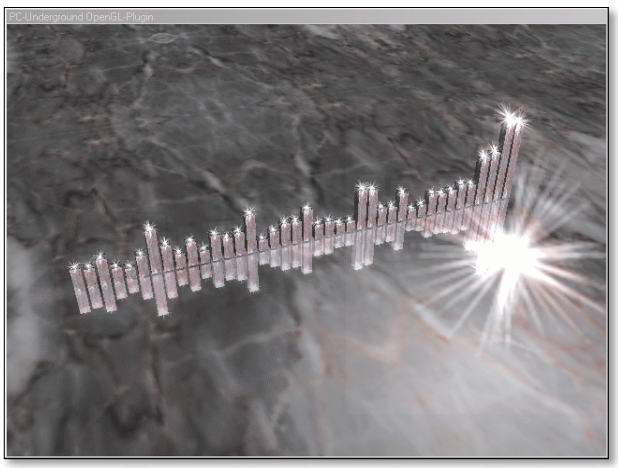
```
glLoadIdentity();
```

Egal, ob Sie diese Matrix auf eine andere Matrix oder einen Vektor anwenden, Sie erhalten als Ergebnis immer wieder den unveränderten Operanden zurück.

Die Kamera- bzw. die Perspektiv- und die Viewport-Transformation legen Sie am einfachsten mit einem Befehl aus der *Glut Library* fest. Dabei handelt es sich um eine Sammlung von Highlevel-Befehlen, die sehr komfortable OpenGL-Funktionen enthalten.

```
gluPerspective(45.0f,1.33f,
  1.0f,1000.0f);
glViewport(0,0,
  SCREEN_X,SCREEN_Y);
```

Die erste Zeile definiert eine Kamera mit einem Öffnungswinkel von 45 Grad. Der zweite Parameter der Funktion *gluPerspective* beschreibt den Aspect-Ratio-Wert, also das Verhältnis von der Breite zur Höhe des Bildschirms. Zuletzt übergeben Sie noch die gewünschten Entfernungs-Clipping-Ebenen mit Abstandswerten von 1 und 1000.



EINE STIMMUNGSVOLLE UND DIFFUSE BELEUCHTUNG sorgt beim zweiten Plugin für glitzernde Lichteffekte.

Dank des folgenden Befehls *glViewport* weiß OpenGL, wie groß das Fenster ist. Jetzt verschaffen Sie der Kamera noch etwas Abstand von der Projektionsebene, auf die die Primitive projiziert werden:

```
glTranslatef(0.0f,0.0f,-30.0f);
```

Das war die Definition der Kamera.

Es gibt noch ein paar zusätzliche Initialisierungsaufrufe, die Sie einmalig beim Start des Programms einsetzen:

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

// Flatshading, d.h. ein Hellig-
// keitswert pro Polygon
glShadeModel(GL_FLAT);

// Z-Buffer-Vergleichsfunktion
glDepthFunc(GL_LEQUAL);
```

Nun legen Sie die Objekte der Szenerie fest. Dazu wählen Sie die Modelview-Matrix, die die Bewegung und Drehung eines Objektes bestimmt. Danach laden Sie wieder die Identität, also die Einheitsmatrix:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

Für eine Verschiebung, Skalierung oder Drehung stehen dann folgende drei Befehle zur Verfügung:

```
glTranslatef(float x,float y,
  float z)
glScalef(float x,float y,
  float z)
glRotatef(float drehwinkel,
  float achse_x,float achse_y,
  float achse_z)
```

Dabei spielt die Reihenfolge der Transformationen durchaus eine Rolle: Je nachdem, ob Sie ein Objekt zum Beispiel vor oder nach einer Drehung verschieben, erhalten Sie ein anderes Ergebnis.

Nachdem Sie die Transformation festgelegt haben, übermitteln Sie die Polygon-Primitive, aus denen Sie Ihre Objekte zusammensetzen, an OpenGL. Eine Übersicht der in diesem Artikel verwendeten Primitive zeigt das Bild auf S. 214. Die Zahlen an den Eckpunkten deuten die Reihenfolge an, in der Sie die Punkte übergeben, um die Primitive zu zeichnen.

Ein Primitiv wie ein Dreieck beginnen Sie mit dem Befehl:

```
glBegin( GL_TRIANGLE );
```

Bevor Sie die Eckpunkte übergeben, wählen Sie noch die Farben, Oberflächennormalen und Texture-Mapping-Koordinaten aus. Denken Sie daran, daß Sie es mit einer State Machine zu tun haben – alle Zustände wie Farben, Normalen und Koordinaten gelten für alle Eckpunkte, solange Sie sie nicht ändern:

```
glNormal3f(0,-1,0);
glTexCoord2d(0.0,0.0);
glVertex3f(1.0,2.0,1.0);
glTexCoord2d(1.0,0.0);
glVertex3f(3.0,2.0,1.0);
glTexCoord2d(1.0,1.0);
glVertex3f(1.0,1.0,0.0);
glEnd();
```

Bei den Primitiven, die eine unbestimmte Anzahl von Eckpunkten (Vertices) enthalten können, übergeben Sie so viele Punkte, wie Sie wollen.

Texture Mapping in OpenGL

Um Ihre Objekte mit Texturen zu versehen, genügen in OpenGL wenige Programmzeilen. Aktivieren Sie mit dem Befehl *glEnable* das Texture Mapping. Dann legen Sie fest, daß Texture-Mapping-Koordinaten größer als 1,0 eine Wiederholung (Kachelung) der Textur bedeuten:

```
glEnable(GL_TEXTURE_2D);
glTexParameterf(GL_TEXTURE_2D,
  GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D,
  GL_TEXTURE_WRAP_T,GL_REPEAT);
```

Wählen Sie über eine Nummer die Textur aus, deren Zustand Sie verändern möchten:

```
glBindTexture(GL_TEXTURE_2D,
  int Nummer);
```

Bei einer Größenänderung kann OpenGL die Textur entweder per bilinearer Interpolation oder per Mipmapping anpassen. Wie sich das Texture Mapping hier verhalten soll, wählen Sie – jeweils separat für die Vergrößerung und die Verkleinerung – über das Kommando *glTexParameteri*.

Danach übergeben Sie der Funktion *glTexImage2D* die Texturdaten:

```
glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MIN_FILTER,
  minFilter);
glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MAG_FILTER,
  maxFilter);
void glTexImage2D(GL_TEXTURE_2D,
  0,4,256,256,0, GL_BGRA_EXT ,
  GL_UNSIGNED_BYTE,
  void *texturedaten);
```

In diesem Beispiel übergeben Sie eine 256 x 256 Pixel große Textur mit 4 Byte Farbtiefe, wobei die Konstante *GL_BGRA_EXT* jeweils 8 Bit für den Blau-, Rot-, Grün- und den Alphakanal reserviert. Der Zeiger *texturedaten* verweist auf die geladene Textur im Speicher.

Das war schon alles, um OpenGL Texturen zu entlocken. Doch im Moment sehen diese Texturen noch sehr blaß aus...

Lichtquellen in OpenGL

Die richtige Beleuchtung nimmt entscheidenden Einfluß auf die Wirkung einer Szene. In OpenGL legen Sie zunächst fest, wie die Oberflächen auf die Lichtquelle reagieren sollen. Zum Beispiel können Sie für die Vorderseiten der Polygone eine stimmungsvolle und diffuse Beleuchtung einschalten. Die Vorder- und Rückseite bestimmen Sie durch die Reihenfolge der Eckpunkte – sind diese im Uhrzeigersinn angeordnet, sehen Sie das Polygon von vorne.

Mit *glEnable* aktivieren Sie die Beleuchtungsberechnung:

```
glEnable(GL_LIGHTING);
GLfloat Intensität[4]=
  {1.0,1.0,1.0,1.0};
glMaterialfv(GL_FRONT,
  GL_AMBIENT_AND_DIFFUSE,
  Intensität);
```

Wählen Sie nun eine Lichtquelle. OpenGL stellt Ihnen dabei – je nach Implementierung – mindestens acht verschiedene zur Verfügung. Danach legen Sie die Lichtfarbe sowie die Position und Richtung des Lichts fest:

```
glEnable(GL_LIGHT0);
GLfloat light_diffuse[]=
  {1.0,1.0,1.0,1.0};
GLfloat light_ambient[]=
  {0.1,0.1,0.1,1.0};

glLightfv(GL_LIGHT0,
  GL_AMBIENT,light_ambient);
glLightfv(GL_LIGHT0,
  GL_DIFFUSE,light_diffuse);

glLightfv(GL_LIGHT0,
  GL_POSITION,
  light_position);
glLightfv(GL_LIGHT0,
  GL_SPOT_DIRECTION,
  light_direction);
```

Das ist alles, was Sie für ein respektables OpenGL-Programm benötigen. Das Bild auf S. 218 zeigt das fertige Plugin im Einsatz.

Am besten experimentieren Sie ein bißchen mit dem Quellcode und verändern ein paar Parameter. So bekommen Sie mehr Gefühl für die Wahl der passenden Transformationsmatrix und für eine optimale Beleuchtung.

Möchten Sie sich in OpenGL vertiefen, lohnt sich ein Besuch der Homepage von Silicon Graphics unter

```
www.sgi.com
```

Hier stoßen Sie auf jede Menge Sourcecodes, Tips und Tutorials. So zum Beispiel auch auf der Seite des Silicon-Graphics-Mitarbeiters Mark Kilgard:

```
http://reality.sgi.com/mjk/tips
```

Wenn Sie die Plugins ausprobieren wollen, kopieren Sie die jeweilige DLL-Datei und für das OpenGL-Programm noch die *raw*- und *tga*-Dateien in das Unterverzeichnis *Plugin* von Winamp. Die aktuelle Version des MP3-Players Winamp erhalten Sie unter

```
www.winamp.com
```

Dort finden Sie auch verschiedene Benutzeroberflächen in unterschiedlichem Design und viele Plugins. PEI

Die Quelltexte sowie die fertig übersetzten Winamp-Plugins finden Sie zusammen mit der zugrundeliegenden Grafikbibliothek auf unserer Heft-CD im Verzeichnis *praxis\pc-under* und im Internet-Angebot des *PC Magazin* unter

```
www.pc-magazin.de/magazin/
➥extras.htm
```

Klicken Sie unter *Online Extras* im Menü *Praxis* auf das entsprechende *Download*-Feld.

Literatur: Jackie Neider, Tom Davis und Mason Woo: OpenGL Programming Guide - The Official Guide to Learning, Addison-Wesley-Verlag, 47 US-Dollar, ISBN 0-2014-6138-2. Eine Online-Ausgabe dieser OpenGL-Einführung finden Sie unter:

```
http://ask.ii.uib.no/ebt-bin/
➥nph-dweb/SGI_Developer/OpenGL
➥_PG/@Generic_BookView
```