AUF DER HEFT-CD

DirectX 7 – Direct3DX Utility Library

Schrill und bunt

DirectX 7 macht Ihre Demos schneller, schöner und lauter. Spielabläufe bewegen sich flüssiger bei geringerem Programmieraufwand. Der Benutzer Ihres Programms kann multimediale Ereignisse steuern.

CARSTEN DACHSBACHER / OLIVER KÄFERSTEIN

Die Direct3DX Utility Library bietet Funktionen, die viele Aufgaben des 3D-Programmierers vereinfachen und sogar übernehmen. Diese Hilfsschicht setzt direkt auf den Direct3D- und DirectDraw-Komponenten von DirectX 7 auf.

Direct3DX wählt das 3D-Gerät für Setup und Ausgabe. In älteren Versionen von DirectX mussten Sie Vollbild- und Fenstermodus unterschiedlich programmieren.

Die neue DirectX-Technik erleichtert es Ihnen, Bildaufbau und -bewegung zu codieren. Sie nimmt Ihnen viel Arbeit ab, wenn Sie Dateien mit den Formaten *bmp*, *tga* und *dds* laden wollen. Diese Dateien entstammen von Vektor- und Matrix-Operationen sowie von Bild- und Textur-Laderoutinen. DirectX erleichtert zudem die schnellen Konvertierungen von Texturen und Farbformaten unabhängig von der verwendeten Grafikkarte.

Zunächst installieren Sie DirectX 7 und das DirectX 7-SDK. Letzteres laden Sie unter www.microsoft.com im Developer-Bereich.

Direct3DX initialisieren

Rufen Sie die Direct3DX-Befehle *D3DXInitialize()* und *D3DXUninitialize()* auf, bevor Sie Direct3DX verwenden und Ihr Programm beenden.

Nach der Initialisierung legen Sie ein *D3DXContext*-Objekt an. Mit diesem Schnittstellenobjekt zeichnen Sie auf einem D3DX-Device. Dieses Objekt repräsentiert eine Grafikkarte, die DirectDraw und Hardware-beschleunigte 3D-Grafik unterstützt. Einen D3DXContext rufen Sie mit der Funktion

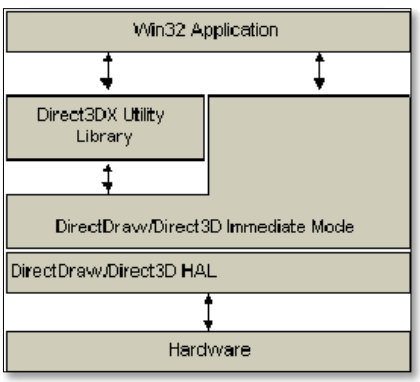
```
HRESULT D3DXCreateContext(
    DWORD deviceIndex,
    DWORD flags,
    HWND hwnd,
    DWORD width,
    DWORD height,
    LPD3DXCONTEXT* ppCtx
);
```

auf. Als Parameter verwenden Sie für *deviceIndex* die Konstante *D3DX_DEFAULT*, die das Gerät mit der besten Hardware-Beschleunigung auswählt. Bei *flags* bestimmen Sie, ob Sie eine Vollbild-, Fenster- oder Offscreen- (also im nicht sichtbaren Bildschirmspeicher) Ausgabe wünschen. Mit der Technik des Offscreen-Rendering berechnen Sie Spezialeffekte wie Spiegelungen:

- *hwnd* ist das Handle Ihres Windows-Fensters,
- *width* und *height* bestimmen die Breite und Höhe der Ausgabe, die Sie auch per *D3DX_DEFAULT* definieren können.
- *ppCtx* gibt die Adresse eines Zeigers auf ein *D3DXContext*-Objekt an.

Der einfachste Aufruf für eine Vollbildausgabe lautet:

```
ID3DXContext *pD3DX = NULL;
D3DXInitialize();
D3DXCreateContext (D3DX_DEFAULT,
        D3DX_CONTEXT_FULLSCREEN,
    my_hwnd, 640, 480, &pD3DX );
```



DAS SCHICHTENMODELL zeigt, wie Direct 3D, DirectDraw und Direct3DX miteinander zusammenhängen.

Die Quellcode-Fragmente in diesem Artikel vernachlässigen Fehlerabfragen. Aber bei der DirectX-Programmierung ist es notwendig, auftretende Fehler peinlichst genau zu überprüfen. Das gilt auch, wenn Sie einen Hardware-Beschleuniger verwenden. Stellen Sie fest, ob dieser unterstützt wird.

Wenn Sie einen D3DXContext erzeugt haben, lassen Sie sich einen Zeiger auf das darin enthaltene D3D-Ausgabegerät (D3DDevice) geben:

```
LPDIRECT3DDEVICE7
 pD3DDevice = NULL;
pD3DDevice =
 pD3DX->GetD3DDevice();
```

Als nächstes prüft Ihr Programm, ob das D3DDevice die nötigen Fähigkeiten hat. *DeviceCaps* geben Ihnen Auskunft darüber, was eine 3D-Hardware ausführen kann. *DeviceCaps* sind Windows-Strukturen, mit denen Sie die Funktionalität von Windows-Geräten (Treibern und Hardware) abfragen. Sie erfragen diese Werte mit der Syntax:

```
D3DDEVICEDESC7    D3DCaps;
memset(&D3DCaps, 0x00,
 sizeof(D3DDEVICEDESC7));
pD3DDevice->GetCaps(&D3DCaps);
```

Der Befehl liefert Daten als Bits und Flags zurück. Um deren Bedeutung herauszufinden, nutzen Sie definierte Konstanten. Zum Beispiel gibt die *D3DPBLENDCAPS_ONE*-Konstante Auskunft darüber, ob das Device Alpha-Blending beherrscht, also transparente Texturen darstellen kann:

```
BOOL Caps;
Caps =
(D3DCaps.dpcTriCaps.
dwSrcBlendCaps &
 D3DPBLENDCAPS_ONE) &&
(D3DCaps.dpcTriCaps.
 dwDestBlendCaps
 & D3DPBLENDCAPS_ONE) &&
(D3DCaps.dpcTriCaps.dwShadeCaps
& D3DPSHADECAPS_ALPHAFLATBLEND);

if(!Caps) return
 NICHTUNTERSTUETZT;
```

Die DirectX-Dokumentation beschreibt alle Caps-Konstanten.

Direct3D initialisieren

Setzen Sie den *Renderstates* (Begriff aus der DirextX-Welt), welcher das Aussehen der Grafikausgabe festlegt. Dieses beginnt mit der Hintergrundfarbe, geht über Textur-Mapping-Optionen und Effekte wie Nebel bis hin zu geometrischen Transformationen für die 3D-Daten. Die Hintergrundfarbe setzen Sie mit

```
D3DXCOLOR colorClear
 (1.0f, 0.0f, 0.0f, 1.0f);
pD3DX->SetClearColor
 (g_colorClear);
```

Die 3D-Daten, die Sie Direct3DX übergeben wollen, wandeln Sie auf zwei Arten für die Bildschirmausgabe um:

• Die sogenannte View-Matrix enthält die Information über Position und Blickrichtung des Betrachters.

• Die Projektionsmatrix bildet den dreidimensionalen Raum auf dem Bildschirm ab. Hierbei helfen Direct3DX-Matrixroutinen. Diese sind Bestandteil des Direct3X-Systems.

Definieren Sie eine Perspektiv-Transformation mit einem Kameraöffnungwinkel von 45 Grad:

```
D3DXMATRIX matProjection;
D3DXMatrixPerspectiveFov
 (&matProjection,
  D3DXToRadian(45.0f),
   3.0f/4.0f, 0.1f, 100.0f);
pD3DDevice->SetTransform
 (D3DTRANSFORMSTATE_PROJECTION,
  matProjection);
```

Der Wert *3.0f/4.0f* gibt das Höhen-Breiten-Verhältnis des Monitors an. Die Werte *0.1f*, *100.0f* markieren die minimale und maximale Distanz des sichtbaren Raums.

Für eine View-Matrix verwenden Sie eine affine Abbildung: eine Drehung und eine anschließende Verschiebung (Translation) im Raum. Direct3DX definiert Drehungen durch Quaternionen. Dabei wird eine Drehung nicht durch die Rotationswinkel um die Koordinatenachsen oder eine Rotationsmatrix beschrieben, sondern durch eine Achse, um die gedreht wird, und einen Drehwinkel. Da sich Quaternionen im Gegensatz zu Rotationsmatrizen interpolieren lassen, eignen sich erstere besser für Animationen.

Ein Quaternion können Sie sich automatisch aus Rotationswinkeln anlegen lassen. Für eine Drehung um die Achsen *X*, *Y*, *Z* mit den Winkeln *aX*, *aY* und *aZ* schreiben Sie so:

```
D3DXQUATERNION qR;
D3DXQuaternionRotation
➥YawPitchRoll(&qR, aY, aX, aZ);
```

Diese Reihenfolge stammt von dem englischen Begriffs-Tripel *Yaw*, *Pitch*, *Roll*. Damit stellen Sie die View-Matrix auf:

```
D3DXVECTOR3 Verschiebung
 (0.0f, 0.0f, 0.0f);
float  Skalierung = 1.25f;
D3DXMatrixAffineTransformation
(&matView, Skalierung, NULL,
 &qR, &Verschiebung);
D3DXMatrixInverse
 (&matView, NULL, &matView);
pD3DDevice->SetTransform
 (D3DTRANSFORMSTATE_VIEW,
  matView);
```

Übergeben Sie die Transformations-Matrizen mit

```
pD3DDevice->SetTransform
(D3DTRANSFORMSTATE_???, NULL);
```

Ob Sie Polygone von vorne, hinten oder von beiden Seiten sehen wollen, und wie Sie die Polygone schattieren wollen, bestimmen folgende Zeilen:

```
// Polygone nur von vorne
pD3DDevice->SetRenderState
(D3DRENDERSTATE_CULLMODE,
 D3DCULL_CCW);
// Flatshading(Schatten/Polygon)
pD3DDevice->SetRenderState
 (D3DRENDERSTATE_SHADEMODE,
  D3DSHADE_FLAT);
```

All diese Zustände (States) erklärt die DirectX-7-SDK-Hilfe.

Moderne 3D-Beschleuniger führen Ihnen zudem ein Texture Mapping vor, das eine Besonderheit aufweist: Multi-Texturing. Dabei legen sich mehrere Texturen übereinander auf ein 3D-Objekt und verknüpfen diese durch spezielle Operationen wie Überblenden oder Farb-Addition. Das legt der Befehl

```
IDirect3DDevice7::
SetTextureStageStage(...)
```

fest. Der erste Parameter gibt die Textur-Stage (Tiefe) an, deren Werte von *0* bis *7* reichen.

Die Optionen für die Textur-Stages unterteilen sich in Farb-, Alpha- und Textur-States mit zahlreichen Optionen. Für 24 Textur-Stage-States sind bis zu 24 weitere Werte zulässig.

Das folgende Beispiel verpasst Ihrem 3D-Objekt eine Textur ohne Extras. Die Textur wird nur beim Vergrößern und Verkleinern gefiltert

```
pD3DDevice->SetTextureStageState
 (0, D3DTSS_MINFILTER,
  D3DTFN_LINEAR);
pD3DDevice->SetTextureStageState
 (0, D3DTSS_MAGFILTER,
      D3DTFG_LINEAR);
pD3DDevice->SetTex-
tureStageState
 (0, D3DTSS_MIPFIL-
TER,
  D3DTFP_POINT);
pD3DDevice->SetTex-
tureStageState
 (0, D3DTSS_COLOROP,
  D3DTOP_MODULATE);
pD3DDevice->SetTex-
tureStageState
 (0, D3DTSS_ALPHAOP,
  D3DTOP_SELEC-
TARG1);
```

Da Sie nicht immer alle acht Stages verwenden (was die meisten 3D-Grafikkarten ohnehin nicht können), schalten Sie einige Stages ab. Um alle Stages ab dem Wert *1* zu deaktivieren, schreiben Sie:

```
pD3DDevice->SetTex-
tureStageState
 (1, D3DTSS_COLOROP,
  3DTOP_DISABLE);
pD3DDevice->SetTex-
tureStageState
```

MD2-DATEIEN in bewegter Animation, die Sie dazu noch selber steuern, bezaubern nicht nur Kinder, sondern auch Erwachsene.

LINKS ZU *MD2-MODELS*

Unter folgenden URLs finden Sie weitere Informationen zu *MD2*-Dateien und Texturen:
http://home.earthlink.net/~benhroop/tutbuild.html
www.ozemail.com.au/~darma/qhelp/hqmods.html
Beschreibungen von vielen Dateiformaten finden Sie unter www.wotsit.com

```
(1, D3DTSS_ALPHAOP,
 D3DTOP_DISABLE);
```

Die dazugehörige Textur, laden Sie einfach aus einer *bmp*-Datei:

```
LPDIRECTDRAWSURFACE7 ppTex;
D3DX_SURFACEFORMAT sf =
 D3DX_SF_UNKNOWN;
D3DXCreateTextureFromFile
 (pD3DDevice, 0, 0, 0, &sf,NULL,
 &ppTex, NULL, „texture.bmp“,
  D3DX_FT_LINEAR);
```

Wenn Sie diesen Setup-Code eingegeben haben, können Sie sofort mit dem Zeichnen loslegen. Bei früheren DirectX-Versionen mussten Sie zuerst ein Direct3D-Device zum Zeichnen suchen und dabei die Texturen in ein Format bringen, das Ihnen der 3D-Beschleuniger vorgab. Jetzt nimmt Ihnen Direct3DX diese Arbeit ab.

Rendering mit Direct3D

Zum Zeichnen eines Bildes beginnen Sie in Direct3D mit der Syntax:

```
pD3DDevice->BeginScene()
```

Im nächsten Schritt löschen Sie Bildschirm und Z-Buffer:

```
pD3DX->Clear(D3DCLEAR_TARGET
| D3DCLEAR_ZBUFFER);
```

Der Z-Buffer speichert für jeden Bildschirmpixel die Entfernung zum Betrachter. Daher sehen Sie Polygone korrekt im Vordergrund, die sich näher am Standpunkt des Betrachters befinden. Den Z-Buffer aktivieren Sie folgendermaßen:

```
pD3DDevice->SetRenderState
(D3DRENDERSTATE_ZWRITEENABLE,
TRUE);
```

Dann selektieren Sie die vorher geladene Textur, wobei der erste Parameter die Textur-Stage und der zweite den Zeiger auf die Textur bezeichnet:

```
pD3DDevice->SetTexture(0,ppTex);
```

Nun schicken Sie Direct3D die Polygone, die Ihre 3D-Grafikkarte zeichnen soll. Es gibt verschiedene Varianten:

• Die erste schickt die vorliegenden Daten unverändert mit folgendem Befehl an Direct3D

```
pD3DDevice->DrawPrimitive(...)
```

Dieser Funktion teilen Sie mit, was zu zeichnen ist: zum Beispiel Punkte, Linien, Dreiecke oder Dreiecksstreifen. Dazu geben Sie an, wie Ihre *Vertex*-Daten (Daten der 3D-Punkte) vorliegen. Wollen Sie zwei Dreiecke zeichnen, ohne sich um Texturen und Beleuchtung zu kümmern, schreiben Sie:

```
D3DLVERTEX data[ 6 ];
// Koordinaten setzen
for ( i = 0; i < 6; i++ ) {
data[ i ].x = ...;
data[ i ].y = ...;
data[ i ].z = ...;
}
pD3DDevice->DrawPrimitive

(D3DPT_TRIANGLELIST,
 D3DFVF_LVERTEX,
 (LPVOID)&data, 6, 0 );
```

Die Vertex-Daten, die Sie dabei übergeben, werden mit den vorgegebenen Matrizen transformiert, projiziert und schließlich dargestellt. Um 3D-Objekte schneller auszugeben, verwenden Sie

```
DrawIndexedPrimitive(...)
```

Dieser Befehl verwendet eine Liste mit Vertices und definiert die Dreiecke mit jeweils drei Indizes dieser Liste. Diese Form der Daten heißt *Shared Vertex*-Struktur. Der Vorteil ist, dass fast immer weniger Daten transportiert werden müssen. Bei größeren 3D-Objekten gilt:

```
DrawPrimitiveDraw
➥IndexedPrimitive
Anzahl Dreieckenn
Anzahl Verticesn*3n/2
```

• Noch effizienter verschicken Sie die Polygon-Daten mit dem *Vertex-Buffer*. Sie definieren, wie Ihre Daten aussehen, packen diese in ein Paket und übergeben es Direct3D. Die Technik dahinter organisiert und erledigt den Rest. Einen Vertex-Buffer legen Sie während der Initialisierungsphase an.

Beschreiben Sie darin zuerst die Vertex-Daten. Füllen Sie einen Vertex-Buffer mit noch untransformierten Vertices, Textur-Koordinaten und einem Farbwert pro Vertex, mit dem Sie dann Dreiecke zeichnen. Die Struktur dieser Beschreibung füllen Sie folgendermaßen:

```
D3DVERTEXBUFFERDESC vbdesc;
vbdesc.dwSize = sizeof(vbdesc);

vbdesc.dwCaps = 0;
vbdesc.dwFVF = D3DFVF_XYZ |
 D3DFVF_DIFFUSE | D3DFVF_TEX1 |
 D3DFVF_TEXCOORDSIZE2(0);
vbdesc.dwNumVertices = nVertices;
```

Dabei müssen Sie die maximale Anzahl der Vertices, die Sie in diesem Buffer speichern wollen, vorher wissen. Der Eintrag *dwFVF* bedeutet *flexible vertex format*. Damit legen Sie einen Vertex-Buffer nach dieser Beschreibung an:

```
LPDIRECT3DVERTEXBUFFER7
 pvbVertices;
pD3D->CreateVertexBuffer
 (&vbdesc, &pvbVertices, 0);
```

Als erste Sicherheitsmaßnahme beantragen Sie den Zugriff auf den Vertex-Buffer. Damit gewährleisten Sie, dass Sie keine Daten überschreiben, die eventuell noch gar nicht verarbeitet wurden. Sie müssen den Vertex-Buffer deshalb also gegen andere Zugriffe wie zum Beispiel von Grafikkartentreibern sperren. Gleichzeitig erhalten Sie einen Zeiger auf den Speicher, in dem die Vertex-Daten stehen.

Während des Zeichnens füllen Sie den Buffer zwischen *BeginScene()* und *EndScene()* mit Ihren Vertex-Daten.

Die Struktur des Buffers definieren Sie so, dass sie Ihrem Vertex-Format entspricht. Dies haben Sie mit dem Vertex-Buffer angegeben:

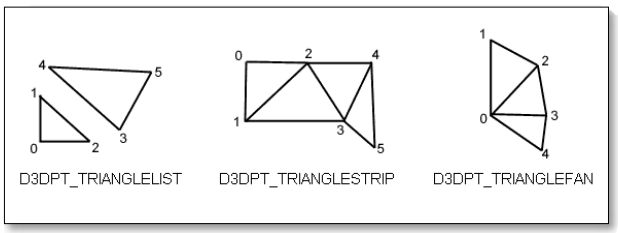
```
typedef struct MY_VERTEX
{    D3DXVECTOR3 pos;
     D3DCOLOR    color;
     D3DXVECTOR2 texcoord;
} MY_VERTEX;

MY_VERTEX *pVertices;
pvbVertices->Lock(DDLOCK_WAIT
| DDLOCK_WRITEONLY,
  (void **) &pVertices, NULL);
```

Greifen Sie auf den Speicher so zu, dass Sie Ihre Vertex-Daten schreiben können. Achten Sie darauf, dass Sie nicht mehr Daten schreiben, als der Vertex-Buffer aufnimmt. Dafür haben Sie eine

KOMPONENTEN VON DIRECTX 7

• Mit Direct3D programmieren Sie einfach und Hardware-unabhängig 3D-Grafiken in zwei verschiedenen Modi:
— zum einen mit dem *Retained Mode*-Interface, einer High-Level Schnittstelle mit abstrakter Sichtweise,
— und mit dem *Low-Level Immediate Mode*, der die gesamte Rendering-Pipeline kontrolliert.
• Direct3D umfasst seit der Version 7 mit der Direct3DX-Utility-Library eine zusätzliche Schicht. Diese übernimmt die wiederkehrenden Aufgaben des Immediate Mode.
• Mit DirectDraw greifen Sie direkt auf den Grafikkartenspeicher zu, wobei Sie zusätzliche Bitmaps im Hintergrund halten.
• DirectInput steuert Eingabe- und Force-Feedback-Geräte aller Art an.
• DirectMusic spielt als Komplettsystem Musik und Soundeffekte ab.
• Mit DirectPlay programmieren Sie Spiele für das Netz per Modem-, LAN- oder WAN-Übertragung.
• Die DirectSetup-API installiert Komponenten in einem Windows-DirectX-System.
• DirectSound gibt Wave-Sounds wieder und unterstützt dabei Hardware- and Software-Mixing der Klangdaten mitsamt einer 3D-Positionierung und vielem mehr.



FEINARBEIT: So übergeben Sie die Polygonnetze.

maximale Anzahl von Vertices angegeben. Füllen Sie den Vertex-Buffer so:

```
for ( i = 0; i < 6; i++ ) {
    pVertices->pos.x = ...;
    pVertices->pos.y = ...;
    pVertices->pos.z = ...;
    pVertices->color =
 D3DCOLOR(1.0f, 0.0f, 1.0f);
    pVertices->texcoord =
 D3DXVECTOR2(0.0f, 1.0f);
    pVertices++;
}
```

Ein *Unlock*-Befehl gibt den Vertex-Buffer wieder frei:

```
pvbVertices->Unlock();
```

Verschicken Sie den Inhalt des Vertex-Buffers zum Zeichnen an Direct3D:

```
pD3DDevice->DrawPrimitiveVB
(D3DPT_TRIANGLELIST,
 pvbVertices, 0, 6, 0 );
```

Ein weiterer Vorteil der Vertex-Buffer: Wenn Sie Direct3D die Transformationen übernehmen lassen und ein nicht animiertes 3D-Objekt (nur Vertices) im dreidimensionalen Raum übergeben, können Sie einen Vertex-Buffer einmalig anlegen und immer wiederverwenden. Sie können das Objekt aber noch frei mit Matrizen bewegen und drehen. Direct3D passt häufig verwendete Vertex-Buffer automatisch und optimiert für das verwendete Direct3D-Device an. Dazu dient die Funktion

```
pvbVertices->Optimize
 ( pD3DDevice, 0 );
```

Wenn Sie Ihre Vertex-Daten auf Performance optimieren, sollten Sie Vertex-Buffer mit dem *DrawIndexedVB(...)*-Befehl verwenden. Alle Polygon-Daten lassen sich in indizierte Polygonnetze umwandeln. Unser Artikel bietet dazu mit *poly2ver.cpp* den Pseudocode. Als Feinarbeit übergeben Sie die Polygonnetze nicht lose, sondern in sogenannten *Triangle-Strips* oder *Triangle-Fans*.

Der Vorteil der Übergabe von *Triangle-Strips* oder *Triangle-Fans*: Der 3D-Beschleuniger hat es mit weniger unterschiedlichen Kanten zu tun. Er kann bei Triangle-Lists nicht erkennen, dass sich zwei Dreiecke eine Kante teilen, wenn sie die gleichen Indizes verwenden. Bei Triangle-Strips/Fans ist das per Definition gegeben. Weniger Kanten teilen sich Polygone, die nicht vollständig auf dem Bildschirm liegen, bei denen also ein Teil abgeschnitten (geclipped) wird. Dann braucht der 3D-Beschleuniger eine Kante nur einmal zu clippen.

Wenn Sie alles mit oder ohne Vertex-Buffer gezeichnet haben, beenden Sie den Vorgang mit

```
pD3DDevice->EndScene();
```

und stellen das Bild dar:

```
pD3DX->UpdateFrame(0);
```

Der Vergleich älterer Direct3D-Versionen mit Direct3DX zeigt, dass sich Direct3D OpenGL mit seiner GLUtility-Library (GLUT) annähert, was das Handling der Transformationen und Texturen angeht. Außer den bereits erwähnten Features erlaubt Ihnen Direct3DX wie GLUT, mit einem Matrix-Stack zu arbeiten und einfache geometrische Primitive wie Kugel, Kegel oder Torus, zu zeichnen.

Auf einem Matrix-Stack können Sie Matrizen mit *Push*- und *Pop*-Operationen speichern. Matrix-Operationen verändern nur die oberste Matrix auf dem Stack. Matrix-Stacks sind vor allem praktisch, wenn Sie mit einem hierarchisch aufgebauten, animierten 3D-Objekt arbeiten.

Das funktioniert ähnlich wie bei dem Bewegungsablauf von Körper-Oberarm-Unterarm-Hand. Wenn sich der Körper bewegt, bewegen sich alle drei anderen Teile auch. Bewegt sich der Unterarm, ist nur noch die Hand betroffen.

Geometrische Primitive erzeugen Sie mit dem *ID3DXSimpleShade*-Interface von Direct3DX. Die Funktionen dieses Interfaces liefern Ihnen die Daten in Form eines Vertex-Buffers mit den Vertices und Texture-Mapping-Koordinaten und einer Indexliste für die Polygone.

Beispielprogramm: Direct3D

Das Beispielprogramm liest *MD2*-Dateien ein, welche Vertex-, Textur- und Animationsdaten speichern, und stellt diese dar. In der Datei *MD2model.cpp* (auf der Heft-CD) finden Sie die Routinen, um *MD2*-Dateien zu lesen und die darin enthaltenen Daten für die Ausgabe mit Direct3D aufzubereiten. *app.cpp* initialisiert und steuert dabei den Ablauf des Beispielprogramms. ET

Die Quelltexte sowie die fertig übersetzten Packprogramme finden Sie auf unserer Heft-CD 1 im Verzeichnis *Praxis/PC-Underground* und auf unserer Website unter www.pc-magazin.de/magazin/extras.htm

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

1 poly2ver.cpp

```
// poly2ver.cpp
// Polygon- in Shared-Vertex-Polygon
typedef struct
{
  VERTEX a, b, c;
}TRIANGLE;

// Shared Vertex Daten
VERTEXVertexList[MAXV];
intnVertices;
intTriangleList[MAXT][3];
intnIdxTriangle;
...
// Sucht Vertex in VertexList
// und liefert dessen Index.
// Kein Vertex vorhanden? Einfügen!
int IndexOf( VERTEX v )
{
for ( int i = 0; i < nVertices; i++)
  {
    if ( Equal( v, VertexList[ i ] )
return i;
  }
  VertexList[ nVertices ] = v;
  nVertices ++;
  return nVertices - 1;
}

voidConvert2Idx
( TRIANGLE *src, int nTriangles )
{
nVertices = 0;
nIdxTriangle = nTriangles;
for ( int i =0; i < nTriangles;i++ )
{
TriangleList[ i ][ 0 ] =
   IndexOf( src[ i ].a );
TriangleList[ i ][ 1 ] =
   IndexOf( src[ i ].b );
TriangleList[ i ][ 2 ] =
   IndexOf( src[ i ].c );
}
}
```

***poly2ver.cpp* wandelt ein Polygon- in ein Shared-Vertex-Polygon-Netz um.**