AUF CD

Die Quelltexte sowie die fertig übersetzten Routinen finden Sie unter *Heft Add-ons/Programmierung/PC Underground.*

Non Photorealistic Rendering

Beethoven aus Bits und Bytes

Bringen Sie Ihre Grafikkarte mit OpenGL dazu, Bilder in Echtzeit darzustellen, die **wie handgemalt** aussehen. Die Grundlagen dazu lernen Sie in diesem Artikel kennen!

CARSTEN DACHSBACHER

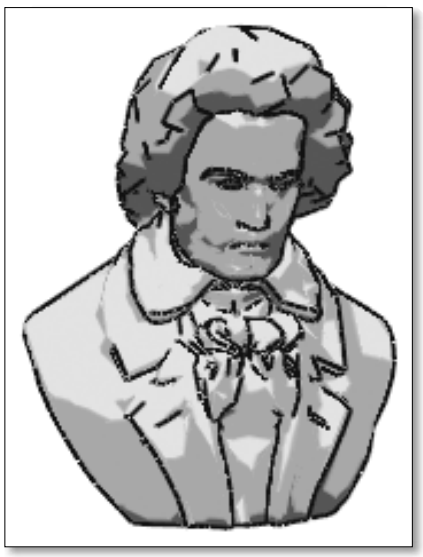
In den bisherigen PC-Underground-Artikeln über 3D-Grafikprogrammierung haben Sie sich immer bemüht, möglichst realistische Berechnungen und Darstellungen durchzuführen. Das Ziel ist es, die Realität treffend nachzuahmen. Diesem Ziel streben Sie auch in dieser Ausgabe nach. Doch diesmal handelt es sich nicht um natürliche Phänomäne, sondern um handgezeichnete Grafiken, wie Sie sie aus Zeichentrickfilmen (Cartoons) kennen. Damit steigen Sie in die Grundlagen des 3D-Cartoon-Renderings ein.

Das Cartoon-Rendering ist eines der Gebiete, die unter dem Begriff Non-Photorealistic Rendering (NPR) zusammengefasst werden. Andere Themen von NPR sind, mit Hilfe von 3D-Hardware Schwarzweiß zu illustrieren, den Eindruck von Wasserfarben zu vermitteln oder Maltechniken mit Pinselstrichen nachzuahmen.

Inking

Die erste Technik, die wir Ihnen hier vorstellen, ist das so genannte Inking. Dieser Begriff (ink: Tinte) bezeichnet, wie Sie die Randlinien einer Comicfigur oder eines Objekts bestimmen. Der Begriff Painting bezeichnet, wie Sie das durch Linien begrenzte Innere füllen oder färben. Damit folgt das Painting dem Inking.

Mit diesen beiden Techniken können Sie herkömmliche 3D-Modelle darstellen, wie Sie sie mit Milkshape 3D, 3D-Studio Max, Lightwave und anderen Modelling-Programmen geschaffen haben. Das Resultat unseres Beispielprogramms dieser Ausgabe kann sich sehen lassen!



DAS CARTOON-Rendering-Verfahren stellt die Beethoven-Büste vor.

Für das Inking benötigen Sie außer der Liste der Knoten und Dreiecke des 3D-Modells noch eine Liste mit allen Kanten. Verwenden Sie folgende Datenstrukturen für ein 3D-Modell:

```
// ein Knoten
typedef struct
{
  GLfloat  x, y, z;
}VERTEX3D;

// Fläche (3 Indizes + Normale)
typedef struct
{
  GLint    a, b, c;
  VERTEX3D normal;
}FACE;

// Kante
typedef struct
{
  GLint  a, b;
  int    boundary;
  int    poly[ 2 ];
}EDGE;

VERTEX3D  *pVertexList;
VERTEX3D  *pNormalList;
EDGE      *pEdgeList;
EDGE      *pRenderEdge;
FACE      *pFaceList;

int       nVertices,
          nFaces,
          nEdges,
          nRenderEdges;
```

Die *pVertexList* und die *pFaceList* erhalten Sie aus den Daten der 3D-Modelle. Auf der Heft-CD finden Sie neben dem Beispielprogramm, das diese Daten einliest, einige 3D-Modelle im ASCII Format. Sie berechnen selbst die Normalen der Dreiecksflächen, der Knoten und die Kantenliste. Beginnen Sie zunächst mit den Dreiecksnormalen. Diese Routine verwendet überladene Operatoren für die Vektorrechnung, die Sie in der Datei *VERTEX3DOP.h* finden:

```
for ( i = 0; i < nFaces; i++)
{
VERTEX3D *a1, *a2, *a3;
VERTEX3D a, b;

// Eckpunkte des Dreiecks
a1=&pVertexList[pFaceList[i].a];
a2=&pVertexList[pFaceList[i].b];
a3=&pVertexList[pFaceList[i].c];

// zwei Kanten des Dreiecks
a = *a2 - *a1;
b = *a3 - *a1;

// Normale durch Kreuzprodukt
// der Kanten bestimmen

pFaceList[ i ].normal = a ^ b;

// und normalisieren
~pFaceList[ i ].normal;
}
```

Die Normalen an den Vertices (Knoten) erhalten Sie, indem Sie jeweils den Summenvektor der Normalen aller Dreiecke nehmen, an denen der betrachtete Vertex beteiligt ist. Diesen Summenvektor müssen Sie anschließend normalisieren:

```
for (i =0;i< nVertices;i++)
  pNormalList[ i ].x =
  pNormalList[ i ].y =
  pNormalList[ i ].z = 0.0f;

for ( i = 0;i< nFaces;i++)
{
pNormalList[pFaceList[i].a]+=
 pFaceList[ i ].normal;
pNormalList[pFaceList[i].b]+=
 pFaceList[ i ].normal;
pNormalList[pFaceList[i].c]+=
```

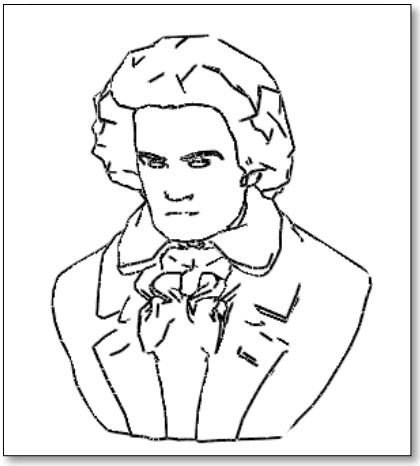
```
 pFaceList[ i ].normal;
}

for ( i = 0; i < nVertices; i++)
  ~pNormalList[ i ];
```

Als nächstes sind die Kanten des 3D-Objekts an der Reihe. Sie benötigen eine Liste, in der jede Kante nur einmal vorkommt. Eine Kante ist entweder eine Randkante, oder sie wird von zwei Dreiecken geteilt. Diese Unterscheidung werden Sie später noch benötigen, um festzustellen, welche Kanten Sie beim Inking zeichnen.

Legen Sie zwei Kantenlisten an. Die ers-te (*pEdgeList*) enthält alle Kanten, jede nur einmal. Die zweite Liste (*pRenderEdge*) ist zu Beginn leer. In dieser Liste werden für jedes gerenderte Bild die Kanten eingetragen, die beim Inking gezeichnet werden sollen. Bauen Sie wie folgt die Kantenliste auf:

```
// genug Speicherplatz
pEdgeList   =new EDGE[nFaces*3];
pRenderEdge =new EDGE[nFaces*3];
nEdges = 0;

// Makro

#define ADDEDGE2LIST(pp,aa,bb)\
low  = min( aa, bb ); \
high = max( aa, bb ); \
found = 0; \
for ( j = 0; j < nEdges; j++ ) \
  if (pEdgeList[j].a==low && \
pEdgeList[ j ].b == high ) \
  { \
    found = 1; break; \
  } \
if ( !found ) \
{ \
pEdgeList[nEdges].a =low; \
pEdgeList[nEdges].b = high; \
pEdgeList[nEdges].poly[0]=pp; \
pEdgeList[nEdges++].boundary=1; \
} else \
{ \
  pEdgeList[j].poly[1]=pp; \
  pEdgeList[j].boundary ++; \
}

// alle Kanten in die Liste
// übernehmen
for ( i = 0; i < nFaces; i++ )
{
  int low, high, found, j;
  ADDEDGE2LIST( i,
 pFaceList[i].a,pFaceList[i].b);
  ADDEDGE2LIST( i,
 pFaceList[i].a,pFaceList[i].c);
  ADDEDGE2LIST( i,
 pFaceList[i].b,pFaceList[i].c);
}
```

An dieser Stelle haben Sie alle Vorbereitungen für die Daten getroffen, welche der Kanten des 3D-Objekts (aus einer bestimmten Perspektive betrachtet) für



BEETHOVEN muss sich mit den Kanten des Inking begnügen.

das Inking wichtig sind. Diese Aufgabe wird als *Silhouette Edge Detection* bezeichnet. Außer der Silhouette des 3D-Objekts gibt es noch zwei weitere Arten von Kanten, die für die Darstellung relevant sind: diejenigen, die sich am Rand des Dreiecksnetzes befinden und solche, an denen die Oberfläche des Dreiecksgitters einen starken Knick hat.

Die Berechnungen der Silhouette sind abhängig von der Blickrichtung und Betrachterposition. Deshalb müssen Sie für jedes Frame (jedes gerenderte Bild) neu rechnen. Die Randkanten haben Sie bereits bestimmt, als Sie die Kanten aufgelistet haben: Sie sind nur Teil eines Dreiecks. Diese Definition lässt sich erweitern: Dazu legen Sie fest, dass es sich bei einer inneren Kante (die Teil zweier Dreiecke ist) um eine Randkante handelt. Das heißt, dass die benachbarten Dreiecke dort unterschiedliche Materialdefinitionen, Texturkoordinaten oder Normalen besitzen. Diese Betrachtung ist später leicht in die Berechnung einzubauen.

Die Kriterien für eine Kante mit dem Index *i*, die beim Inking zu zeichnen ist, sind hier aufgeführt:

- Die Kante ist Randkante, dann ist *pEdgeList[i].boundary=1*
- Handelt es sich um eine innere Kante, dann ist *(pEdgeList[i].boundary=2)*

Hier befindet sie sich an einem starkem Knick der Oberfläche. Der Winkel überschreitet zwischen den Normalen der benachbarten Dreiecke eine vorher festgelegte Schwelle. Die Verweise auf die Nachbardreiecke haben Sie in der Kantenstruktur gespeichert.

- Die Kante ist Teil der Silhouette. Dann ist *pEdgeList[i].boundary= 2* und eine der Normalen der benachbarten Dreiecke zeigt zum Betrachter hin, die andere weg. Für diese Berechnung benötigen Sie die Blickrichtung des Betrachters. Diese Information finden Sie bei OpenGL in der Modelview Matrix. Hier die Berechnungen in C-Syntax:

```
nRenderEdges = 0;
for (i = 0; i < nEdges; i++)

{
  int add2List = 0;

  // Randkante ?
```

BEETHOVEN mit den drei Rendering-Optionen unseres Beispielprogramms

```
if (pEdgeList[ i ].boundary ==1)
    add2List = 1;

if (pEdgeList[ i ].boundary ==2)
  {
// Nachbardreiecke und Normalen

int p1 = pEdgeList[ i ].poly[0];
int p2 = pEdgeList[ i ].poly[1];
    VERTEX3D
  *n1 = &pFaceList[ p1 ].normal,
  *n2 = &pFaceList[ p2 ].normal;

// Auf Knick in der Oberfläche
// prüfen
// Skalarprodukt ist Cosinus des
// Winkels zwischen den Normalen

    float    dot = *n1 * *n2;

// 0.4 ->arccos(0.4)=66 Grad

    if ( dot < 0.4f )
      add2List = 1;

// Kante der Silhoutte ?
//ein Vertex der Kante als Bezug

    VERTEX3D *vertex =
&pVertexList[ pEdgeList[ i ].a];

    // Blickrichtung !

    float matrix[ 16 ];
    glGetFloatv(
   GL_MODELVIEW_MATRIX, matrix);

    VERTEX3D viewVector;
    viewVector.x = matrix[ 0+2];
    viewVector.y = matrix[ 4+2];
    viewVector.z = matrix[ 8+2];
    ~viewVector;

  // Test, ob unterschiedliche
  // Richtung der Normalen bzgl.
  // der Blickrichtung

  float dot1 = viewVector * *n1;
  float dot2 = viewVector * *n2;

 if( (dot1*dot2) <= 0.00000001f)
    add2List = 1;
  }

  // Hinzufügen, wenn eins der
  // Kriterien erfüllt ist

  if ( add2List )
 pRenderEdge[ nRenderEdges ++] =
    pEdgeList[ i ];
}
```

Jetzt müssen Sie die gewonnenen Ergebnisse, also die Kanten, auf den Bildschirm bringen. Um die Linien, die nicht sichtbar sein sollten, weil sie durch die Oberfläche des 3D-Objekts verdeckt wären, unsichtbar zu machen, zeichnen Sie die Dreiecksflächen des Objekts, allerdings in der Hintergrundfarbe und geringfügig vom Betrachter weg verschoben. Wenn Sie die Flächen des Objekts einfärben wollen, können Sie auch jede andere Farbe wählen. Der Zweck ist, den Z-Buffer (Tiefenpuffer) mit Werten zu füllen, so dass Kanten auf der Rückseite des Objekts nicht gezeichnet werden:

```
glDisable( GL_BLEND );
glDisable( GL_LIGHTING );
glColor3ub( 255, 255, 255 );

glEnable
 (GL_POLYGON_OFFSET_FILL);
glPolygonOffset( 1.0f, 5.0f );

glBegin( GL_TRIANGLES );
for ( i = 0; i < nFaces; i++ )
{
  FACE *f = &pFaceList[ i ];
glVertex3fv(&pVertexList[f->a]);
glVertex3fv(&pVertexList[f->b]);
glVertex3fv(&pVertexList[f->c]);
}
glEnd();

glDisable( GL_TEXTURE_2D );
glDisable
 (GL_POLYGON_OFFSET_FILL);
```

Zeichnen Sie die Kanten. Außerdem sollten Sie, falls Sie den optionalen Anti-Aliasing-Teil verwenden, nochmals gesondert die Vertices als Punkte zeichnen. So vermeiden Sie Lücken in den Linienzügen. Beachten Sie, dass das Anti-Aliasing auf älteren 3D-Karten eventuell nicht unterstützt wird – oder sehr langsam ist.

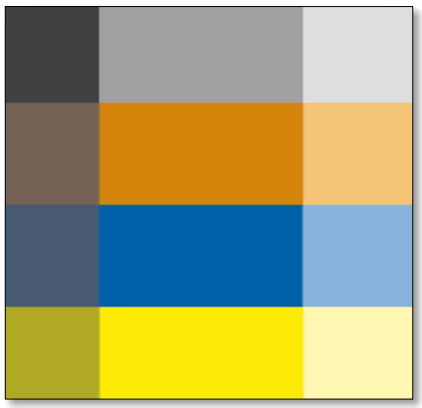
Gleiches gilt übrigens auch für den Schalter *GL_POLYGON_SMOOTH*. Diesen sollten Sie standardmäßig mit *glDisable(...)* ausschalten, weil Treiber wie von GeForce diesen aktivieren. Mit dem Code zeichnen Sie die Linien:

```
glDepthFunc( GL_LEQUAL );

// Linienfarbe und -dicke wählen

glColor3ub( 0, 0, 0 );
glLineWidth( 2 );
glPointSize( 2 );

// -optional-
// anti-aliasing
glEnable( GL_BLEND );
glBlendFunc( GL_SRC_ALPHA,
       GL_ONE_MINUS_SRC_ALPHA );
glEnable( GL_LINE_SMOOTH );
glEnable( GL_POINT_SMOOTH );
// -optional ende-

glBegin( GL_LINES );
for (i = 0;i < nRenderEdges;i++)
{
  EDGE *e = &pRenderEdge[ i ];
glVertex3fv(&pVertexList[e->a]);
glVertex3fv(&pVertexList[e->b]);
}
glEnd();

glDepthFunc( GL_LESS );
glBegin( GL_POINTS );
for (i = 0;i< nRenderEdges;i++)
{
  EDGE *e = &pRenderEdge[ i ];
  glVertex3fv
  ( &pVertexList[ e->a ] );
  glVertex3fv
  ( &pVertexList[ e->b ] );
}
glEnd();
```

Painting

Nun können Sie die relevanten Kanten herausfinden und zeichnen. Füllen Sie im nächsten Schritt die Flächen aus. Auch hier greift der Artikel eine wichtige Methode von vielen heraus, um das Non-Photorealistic Rendering zu programmieren. In Cartoons ist oft zu beobachten, dass die Oberflächen der Objekte zwar schattiert werden, aber nur sehr wenige Farben zeigen, meist nur zwei bis drei: eine für im Schatten befindliche Teile, die Grundfarbe und eventuell noch eine Farbe für Highlights.

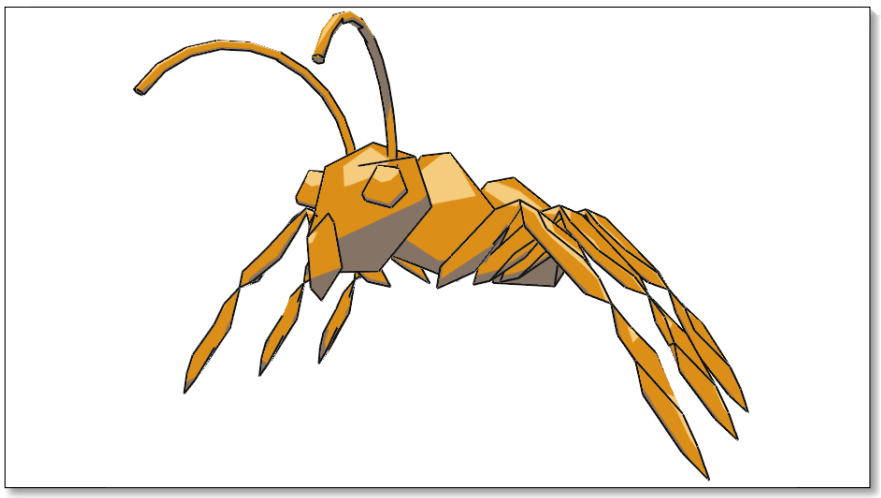
Wie simulieren Sie diesen Effekt mit Ihrer 3D-Hardware? Die Beleuchtung berechnen Sie selbst. Es genügt nicht, wenn Sie die Farben der Flächen oder Vertices bestimmen und an OpenGL weitergeben. Verwenden Sie Flatshading, bekommt jedes Dreieck eine eigene Farbe: Doch damit werden die Dreiecksgrenzen selbst sichtbar. Oder Sie verwenden Gouraud-Shading, womit Sie Farbüberläufe auf einzelnen Dreiecken erhalten, obwohl Sie die Zahl



DIE SCHATTIERUNG der Oberflächen definieren Sie mit einer einfachen Textur.

der Farben reduzieren wollten. Die Lösung liegt im Texture-Mapping. Legen Sie in einem Bildbearbeitungsprogramm Ihrer Wahl eine Textur an.

In dieser wenig spektakulär wirkenden Textur sehen Sie untereinander waagrecht vier frei definierte Farbübergänge, die aus drei Farbstufen bestehen. Sie können auch zwei oder mehr als drei Farben verwenden. Damit berechnen Sie die Beleuchtung für jeden Vertex. Den Helligkeitswert, den Sie erhalten, verwenden Sie als Texture-Mapping-Koordinate für die waagrechte Komponente. Die Farbe der Oberfläche, also welchen der Farbverläufe Sie wollen, bestimmen Sie mit der anderen, der vertikalen Komponente. Rendern Sie die Flächen des 3D-Modells statt einfarbig mit dieser Textur.



SIE RENDERN die Flächen der Ameisen mit der Schattierung aus dem Bild auf S. 213.

Zunächst müssen Sie für das 3D-Objekt eine weitere Liste speichern. Darin speichern Sie die Texturkoordinaten:

```
float *pTexCoordList =
 new float[2*nVertices];
```

Die Textur (auf der Heft-CD) laden Sie über die *PCUTexture*-Wrapper-Klasse. Damit laden Sie *bmp*-Dateien einfacher als OpenGL Texturen:

```
PCUTexture *colors =
 new PCUTexture();
colors->loadBMP( „texture.bmp“);
```

Wenn Sie eine geladene Textur in OpenGL verwenden wollen, wählen Sie sie mit *colors->select()* aus.

Sie berechnen die Beleuchtung, bevor Sie die Dreiecke zeichnen. Dazu müssen Sie die Normalen der Vertices aus dem Objectspace (das lokale Koordinatensystem des 3D-Modells) in den Worldspace transformieren. Der Worldspace ist das Koordinatensystem, in dem alle 3D-Modelle und der Betrachter platziert werden. Diese Transformation nehmen Sie mit der inversen Matrix zur Modelview Matrix vor. Anschließend berechnen Sie den Helligkeitswert eines Vertex aus dem Skalarprodukt der transformierten Normalen und einer festgelegten Lichtrichtung.

```
MATRIX44 modelview, invmodel;

// Matrix holen...

glGetFloatv(
GL_MODELVIEW_MATRIX, modelview);

// ... und invertieren

InverseMatrixAnglePreserving(
  modelview, invmodel);

// Beleuchtungsberechnung

for ( i = 0; i < nVertices; i++)
{
  // Normale holen
  VERTEX3D *n = &pNormalList[ i];

// in Worldspace transformieren

  VERTEX3D tn;
tn.x=*n * *(VERTEX3D*)
 &invmodel[0];
tn.y=*n * *(VERTEX3D*)
 &invmodel[4];
tn.z=*n * *(VERTEX3D*)
 &invmodel[8];
~tn;

// Lichtrichtung

VERTEX3D lightDir =
 {0.0f,1.0f,1.0f};
// normalisieren
~lightDir;

// Helligkeit berechnen

float light = lightDir * tn;

if ( light < 0.1 )
  light = 0.1f;
if ( light > 0.9f )
  light = 0.9f;

// Texturekoordinaten speichern
// Helligkeit

pTexCoordList[ i*2+0 ] = light;

// Farbverlauf wählen

pTexCoordList[i*2+1]=texOffset;
}
```

Aktivieren Sie das Textur-Mapping, um die Dreiecke zu zeichnen, und übergeben Sie die Texturkoordinaten an OpenGL:

```
glEnable( GL_TEXTURE_2D );
colors->select();

glBegin( GL_TRIANGLES );
for ( i = 0; i < nFaces; i++)
{
FACE *f = &pFaceList[ i ];
glTexCoord2fv
 (&pTexCoordList[f->a*2]);
glVertex3fv
 ( &pVertexList[ f->a ] );
glTexCoord2fv
 (&pTexCoordList[f->b*2]);
glVertex3fv
 ( &pVertexList[ f->b ] );
glTexCoord2fv
 (&pTexCoordList[f->c*2]);
glVertex3fv
 ( &pVertexList[ f->c ] );
}
glEnd();

glDisable( GL_TEXTURE_2D );
```

Alternative Techniken

Die Techniken, die Sie in diesem Artikel kennengelernt haben, bilden die Grundlage für das Inking und Painting.

Das auf Kanten basierende Inking ist eine Software-Lösung und daher portierbar. Die Kantendetektion der Silhouette ist von jedem Blickwinkel aus betrachtet korrekt. Die Schwelle für Knicke der Oberfläche ist frei wählbar. Die 3D-Hardware macht mit ihrem Z-Buffer die Kanten für Sie sichtbar. Der Nachteil ist der beträchtliche Rechenaufwand, der erforderlich ist, um die Kanten zu bestimmen.

Einen ganz anderen Ansatz verfolgt die nächste Technik, die Sie in *Cartoon Rendering* von Sim Dietrich finden. Sie erhalten das Dokument unter der URL http://developer.nvidia.com/view.asp?IO=Cartoon_Rendering_GeForce_256

Das Verfahren berechnet mit der Hardware für jeden Pixel das Skalarprodukt aus der Normalen des Pixels und der Viewspace Position. Mit Hilfe einer gewählten Schwelle lässt sich feststellen, ob ein Pixel zur Silhouette gehört.

Der Vorteil dieser Methode liegt im Geschwindigkeitsgewinn, also einer besseren Performanz auf der heutigen 3D-Hardware. Die Darstellung ist aber weniger genau. Um Knicke in der Objektoberfläche zu gestalten, bedarf es aufwändiger Rechenverfahren.

Sie berechnen mit Vertex-Shadern Texturkoordinaten aus der Beleuchtung des 3D-Objekts. Somit können Sie diese Aufgabe der Hardware überlassen. Wenn Sie zusätzlich zur Schattierung eine weitere Textur für Ihr 3D-Objekt verwenden wollen, müssen Sie hierzu allerdings auf die zweite Texturestage ausweichen. ET

Interessantes zur 3D-Programmierung finden Sie auf den folgenden Websites:

www.dachsbacher.de/pcu
viele freie 3D-Modelle

www.3dcafe.com
Sim Dietrich, NVidia Corporation
http://developer.nvidia.com/view.asp?IO=Cartoon_Rendering_GeForce_256

Advanced Rendering Techniques Using OpenGL, SIGGRAPH 99 Course Notes