AUF CD 1

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

Cube Mapping mit OpenGL

Kugeln im Spiegel

Berechnende Hardware: Eine NVidia-GeForce-256/2/3 oder ATI-Radeon-Grafikkarte zaubert Spiegelungen mit wenigen Programmzeilen OpenGL auf den Bildschirm.

CARSTEN DACHSBACHER

Die modernen Grafikkarten wie NVidia GeForce 256, GeForce 2, GeForce 3 und ATI Radeon verlagern die Berechnung immer komplexerer Funktionen in die Hardware und entlasten so die Hauptprozessoren. Zu diesen Funktionen zählen vor allem die Geometrietransformation und die Beleuchtungsberechnung. Texturkoordinaten generieren weitere Aufgaben, die die Grafik-Hardware übernimmt. Dieses Feature und die Cube Maps verwenden Sie, um Spiegelungen zwischen zwei 3D-Objekten und der umgebenden 3D-Szene darzustellen.

Environment Mapping (auch Reflection Mapping) ist eine Texturierungsmethode, die den Anschein erweckt, als spiegele sich die umgebende 3D-Szene auf der Oberfläche eines 3D-Objekts. Beim Texture Mapping sind einem Dreieck feste Texturkoordinaten zugewiesen.

Das Environment Mapping berechnet die Texturkoordinaten für jeden Vertex. Für die Berechnung sind die Oberflächennormale, die Blickrichtung des Betrachters und die Environment-Mapping-Technik ausschlaggebend. Die Berechnungen variieren von Technik zu Technik. Genauso verhält es sich mit den Anforderungen an die Daten, den Environment-Texturen. Diese sollen die umgebende 3D-Szene enthalten.

Spherical Environment Mapping

Spherical Environment Mapping ist die momentan wahrscheinlich am meisten verwendete Technik, welche die meisten 3D-Beschleuniger unterstützen. Die Umgebung wird hierbei in einer einzigen Textur repräsentiert. Eine exemplarische *Spheremap* sehen Sie im Bild unten.

Etwas über 20 Prozent der Fläche, der schwarze Bereich der Textur, werden nicht genutzt. Weil beim Sphere Mapping eine Kugel auf einer Ebene dargestellt wird, treten Verzerrungen auf. Daraus folgt, dass das Verhältnis aus der Fläche eines Pixels in der Spheremap und dem repräsentierten Winkelbereich in der Spiegelung nicht konstant ist, sondern variiert, was Aliasing-Effekte hervorruft.

Von Vorteil sind die gute Unterstützung durch die Hardware und die Grafik-APIs. Sie benötigen beim Sphere Mapping nur eine einzige Textur, um die Spiegelung der Umgebung darzustellen. Spheremaps sind immer vom Betrachterstandpunkt und der -blickrichtung abhängig *(view-dependent)*. Weil Sie dabei Verzerrungen einkalkulieren müssen, ist es nicht einfach, dynamische Spheremaps zu generieren.

Dual Paraboloid Maps

Im Gegensatz zu Spheremaps sind Dual Paraboloid Maps *view-independent*: Sie können sie einmal für die 3D-Szene generieren und den Betrachter frei bewegen. Außerdem gibt es weitaus weniger Verzerrungen als auf einer Spheremap. Ein Manko haben beide: Etwa 25 Prozent der Fläche bleiben ungenutzt.

Das Dual Paraboloid Mapping benötigt entweder Dual Texturing, wobei der 3D-Beschleuniger zwei Texturen gleichzeitig verwenden muss, oder die spiegelnden Oberflächen müssen zweimal gerendert werden. Der Aufwand ist gering, denn das Environment-Mapping-Verfahren, das Sie in dieser Ausgabe einsetzen, stellt weit höhere Anforderungen an die 3D-Hardware.

Dual Paraboloid Maps können allerdings nur schwer eine dynamische 3D-Szene gespiegelt darstellen, da die Generierung der Maps etwas aufwändiger ist.

Cube Environment Mapping

Das Cube Environment Mapping ist das Verfahren unserer Wahl. Es wird seit der Einführung des NVidia-GeForce-256-

UNSER BEISPIELPROGRAMM zeigt zwei spiegelnde Kugeln.

Grafikchips unterstützt. Hierbei präsentiert sich die umgebende 3D-Szene in sechs Texturen.

Der Vorteil: Die 3D-Hardware kann diese sechs Texturen zusammen adressieren. Sie müssen also keine speziellen Environment-Texturen berechnen. Die Verzerrungen, die beim Rendering auftreten, sind kleiner als bei den Dual Paraboloid Maps.

Stellen Sie sich die sechs Texturen einer Cubemap als einen Würfel vor, der aufgefaltet um den Koordinatenursprung liegt. Jedes Texel (Pixel einer Textur) repräsentiert den Teil der Umgebung, der vom Ursprung aus in dieser Richtung sichtbar ist. Vereinfachend nimmt man an, dass die Cubemaps von einem Punkt aus berechnet werden, und die Umgebung unendlich weit entfernt ist. Da diese Annahme nicht realisierbar ist, sind die Spiegelungen für nahe Objekte nicht exakt. Aber die Abweichungen sind akzeptabel, wovon Sie sich im Beispielprogramm überzeugen können.

Um Cubemaps zu berechnen, platzieren Sie eine Kamera in der Mitte des spiegelnden 3D-Objekts und rendern sechsmal die 3D-Szene ohne das spiegelnde Objekt – je eines entlang der positiven und negativen *x*-, *y*- und *z*-Achse.

Cubemapping in OpenGL

Ein Teil der OpenGL-API unterstützt Cubemapping, was neuere Grafikkartentreiber berücksichtigen. Deshalb sollten Sie vorerst prüfen, ob der Treiber die verwendeten Fähigkeiten beherrscht. Außerdem benötigen Sie die aktuellen OpenGL-Extension-Header (*glext.h*), die Sie beim Quelltext des Beispielprogramms finden.

Für das Cubemapping verwenden Sie die *GL_ARB_texture_cube_map* oder die *GL_NV_texgen_reflection*-Extension. So prüfen Sie die Fähigkeiten der Treiber:

```
char *extensions;
extensions = strdup( (char*)
 glGetString( GL_EXTENSIONS ) );
for ( unsigned int i = 0;
i < strlen( extensions ); i ++ )
  if ( extensions[ i ] == ' ' )
 extensions[ i ] = '\n';

if ( strstr( extensions,
 „GL_ARB_texture_cube_map“ ) ||
  strstr( extensions,
   „GL_NV_texgen_reflection“ ) )
{
// Extensions unterstützt
}
```

Unterstützt ein Treiber auch die Erweiterung *GL_SGIS_generate_mipmap*, können Sie diese verwenden. Damit ersparen Sie es sich, die Mipmaps für Texturen manuell aufzubauen, was vor allem für dynamisch generierte Texturen von Nutzen ist.

Wenn die Abfrage der Extensions erfolgreich verlaufen ist, beginnen Sie damit, die Cubemaps anzulegen. Durch die Cubemap-Extension ist auch ein neues Textur-Target definiert.

Beim Basis OpenGL gibt es ein- und zwei-dimensionale Texturen, deren Textur-Targets als *GL_TEXTURE_1D* und *GL_TEXTURE_2D* definiert sind. Diese Targets geben Sie an, um beim Texture Mapping Parameter zu setzen. Das neu definierte Target heißt

JEDER FARBBEREICH der Spheremap repräsentiert eine Blickrichtung entlang der Koordinatenachsen *+x,-x,+y,-y,+z,-z*.

GL_TEXTURE_CUBE_MAP_ARB. Sonst legen Sie die Textur so an, wie Sie es von OpenGL gewohnt sind:

```
GLuint texture;
glEnable
 ( GL_TEXTURE_CUBE_MAP_ARB );
glGenTextures( 1, &texture );
glBindTexture
 ( GL_TEXTURE_CUBE_MAP_ARB,
 texture );
```

Sofern der Treiber die automatische Generierung der Mipmaps unterstützt, verwenden Sie am besten folgende Parameter:

```
glTexParameteri
( GL_TEXTURE_CUBE_MAP_ARB,
 GL_TEXTURE_MIN_FILTER,
 GL_LINEAR_MIPMAP_LINEAR );
glTexParameteri
 ( GL_TEXTURE_CUBE_MAP_ARB,
  GL_TEXTURE_MAG_FILTER,
  GL_LINEAR );
glTexParameteri
 ( GL_TEXTURE_CUBE_MAP_ARB,
   GL_GENERATE_MIPMAP_SGIS,
   GL_TRUE );
```

Anderenfalls können Sie auf das Mipmapping verzichten und schalten die bilineare Filterung ein:

```
glTexParameteri
( GL_TEXTURE_CUBE_MAP_ARB,
  GL_TEXTURE_MIN_FILTER,
  GL_LINEAR );
glTexParameteri
 ( GL_TEXTURE_CUBE_MAP_ARB,
   GL_TEXTURE_MAG_FILTER,
   GL_LINEAR );
```

Als nächstes können Sie Daten in die Texturen kopieren. Denken Sie daran, dass Sie mit einem Aufruf von *glBindTexture(...)* sechs Texturen auswählen. Auf die einzelnen Texturen greifen Sie mit den Konstanten im Array *cubeMapConstants[]* zu:

```
GLuint cubeMapConstants[ 6 ] =
{
GL_TEXTURE_CUBE_MAP
➥_POSITIVE_X_ARB,
GL_TEXTURE_CUBE_MAP
➥_NEGATIVE_X_ARB,
GL_TEXTURE_CUBE_MAP
➥_POSITIVE_Y_ARB,
GL_TEXTURE_CUBE_MAP
➥_NEGATIVE_Y_ARB,
GL_TEXTURE_CUBE_MAP
➥_POSITIVE_Z_ARB,
GL_TEXTURE_CUBE_MAP
➥_NEGATIVE_Z_ARB
};

for ( int i = 0; i < 6; i++ )
  glTexImage2D(
cubeMapConstants[ i ],0,GL_RGB8,
    CUBEMAPSIZE, CUBEMAPSIZE,
    0, GL_RGB, GL_UNSIGNED_BYTE,
    DatenPtr );
```

Abschließend müssen Sie OpenGL noch mitteilen, dass es die Texturkoordinaten aus der Betrachterposition, -blickrichtung und Oberflächennormalen für das Cubemapping berechnen soll. Dazu gibt es eine spezielle Erweiterung, die *GL_REFLECTION_MAP_ARB*-Methode, die Sie mit folgenden Zeilen einbinden:

```
glTexGeni( GL_S,
 GL_TEXTURE_GEN_MODE,
 GL_REFLECTION_MAP_ARB );
glTexGeni( GL_T,
 GL_TEXTURE_GEN_MODE,
 GL_REFLECTION_MAP_ARB );
glTexGeni( GL_R,
 GL_TEXTURE_GEN_MODE,
 GL_REFLECTION_MAP_ARB );
```

Cubemaps in dynamischen 3D-Szenen

Berechnen Sie die Cubemaps für eine dynamische 3D-Szene und ein bewegliches spiegelndes Objekt. Untersuchen Sie die Hauptschleife des Renderings der 3D-Szene. Der wichtige Punkt ist die Abfolge der Transformationen (Verschiebungen und Rotationen). Zunächst platzieren Sie das spiegelnde Objekt, hier eine Kugel, und initialisieren Sie die Projektionsmatrix:

```
// Berechnung der Position
spherePos.x = ...;
spherePos.y = ...;
spherePos.z = ...;
```

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective
 ( 70,  aspectRatio, 1, 5000 );

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

Verwenden Sie als Transformationen in der Modelview-Matrix zwei Rotationen und eine Translation:

```
glRotatef
( xangle, 1.0f, 0.0f, 0.0f );
glTranslatef( 0, 30, 0 );
glRotatef
( zangle, 0.0f, 0.0f, 1.0f );
```

An dieser Stelle zeichnen Sie das spiegelnde Objekt. Der Einfachheit halber verwenden Sie eine Kugel, deren Geometrie die Funktionen der *GLUT*-Bibliothek darstellen kann. Diese Kugel können Sie durch ein beliebiges 3D-Objekt ersetzen, Sie müssen lediglich das Dreiecksnetz inklusive der Normalen der Vertices übergeben.

Zuerst das Setup des Texture Mapping:

```
// Cubemapping aktivieren
glEnable
( GL_TEXTURE_CUBE_MAP_ARB );
// und Cubemap wählen
glBindTexture
( GL_TEXTURE_CUBE_MAP_ARB,
 cubeMap);

glEnable( GL_TEXTURE_GEN_S );
glEnable( GL_TEXTURE_GEN_T );
glEnable( GL_TEXTURE_GEN_R );
```

Jetzt kommt der wichtige Teil mit der Umkehrung der Modelview-Transformationen, damit Sie Texturkoordinaten korrekt berechnen können:

```
glMatrixMode( GL_MODELVIEW );
glPushMatrix();
// 3D-Szene bzgl. spiegelnden
// Objekts verschieben:
glTranslatef
( spherePos.x, spherePos.y,
  spherePos.z );

glMatrixMode( GL_TEXTURE );
glPushMatrix();

// Rotationen rückgängig machen
glRotatef
 ( -zangle, 0.0f, 0.0f, 1.0f );
glRotatef
 ( -xangle, 1.0f, 0.0f, 0.0f );

// 3D-Objekt zeichnen
glutSolidSphere( SPHERE_SIZE,
SPHERE_SUBDIVX, SPHERE_SUBDIVY);

glPopMatrix();
```

Nach dem Zeichnen des Objekts räumen Sie den Matrix Stack wieder auf und deaktivieren die Texturkoordinaten-Generierung. Anschließend zeichnen Sie den Rest der 3D-Szene:

```
glMatrixMode( GL_MODELVIEW );
glPopMatrix();

glDisable( GL_TEXTURE_GEN_S );
glDisable( GL_TEXTURE_GEN_T );
glDisable( GL_TEXTURE_GEN_R );

// Rest der 3D-Szene zeichnen

renderScene();
```

Cubemaps generieren

Sie benötigen noch eine Cube-Environment-Map für Ihr spiegelndes Objekt, die die aktuelle 3D-Szene aus der Sicht des Objekts enthält. Eine Cubemap können Sie leichter als jede andere Environment-Map generieren.

Sie platzieren die OpenGL-Kamera und richten sie aus, rendern die 3D-Szene und kopieren das Resultat in die Cubemap. Diese Schritte sehen Sie an Hand von Codezeilen. Die Berechnung der Cubemap rufen Sie am besten vor der Render-Hauptschleife auf, die Sie zuvor gesehen haben. Legen Sie die Größe der gerenderten Cubemap-Seite fest, die der Größe entspricht, die Sie bei der Initialisierung gewählt haben. Anschließend setzen Sie die Parameter der Projektion. Verwenden Sie einen Kameraöffnungswinkel von 90 Grad. Andernfalls enthalten die Cubemaps nicht die gesamte, umgebende 3D-Szene.

```
// Größe festlegen
glViewport
(0,0,CUBEMAPSIZE,CUBEMAPSIZE);

// Projektion
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective(90, 1.0f,1,500);

glMatrixMode( GL_MODELVIEW );

// für jede Richtung
for( int i = 0; i < 6; i++ )
{
glClear( GL_COLOR_BUFFER_BIT |
         GL_DEPTH_BUFFER_BIT );
glLoadIdentity();
```

Richten Sie die Kamera entlang der Koordinatenachsen $+x$, $-x$, $+y$, $-y$, $+z$, $-z$ aus. Diese Ausrichtung fassen folgende Zeilen zusammen:

```
float cubeMapRotation[6][4] =
  {
    { -90, 0, 1, 0 },
    {  90, 0, 1, 0 },
    { -90, 1, 0, 0 },
    {  90, 1, 0, 0 },
    { 180, 1, 0, 0 },
    { 180, 0, 0, 1 }
  };

  glRotatef(
    cubeMapRotation[ i ][ 0 ],
    cubeMapRotation[ i ][ 1 ],
    cubeMapRotation[ i ][ 2 ],
    cubeMapRotation[ i ][ 3 ] );

  if( i < 2 )
    glRotatef
    ( 180.0f, 0.0f, 0.0f, 1.0f);
```

Verschieben Sie den Mittelpunkt der Kugel in den Ursprung – somit auch den Rest der 3D-Szene – und zeichnen Sie diese:

```
glTranslatef( -spherePos.x,
   -spherePos.y, -spherePos.z );
```

DIE AUFTEILUNG der spiegelnden 3D-Szene bei den Dual Paraboloid Maps.

```
    renderScene();
```

Damit können Sie einen Teil der Cubemap aus dem Framebuffer von OpenGL in die Textur kopieren. Der *glFlush(...)*-Befehl stellt sicher, dass das Rendern abgeschlossen ist, bevor der Kopiervorgang beginnt:

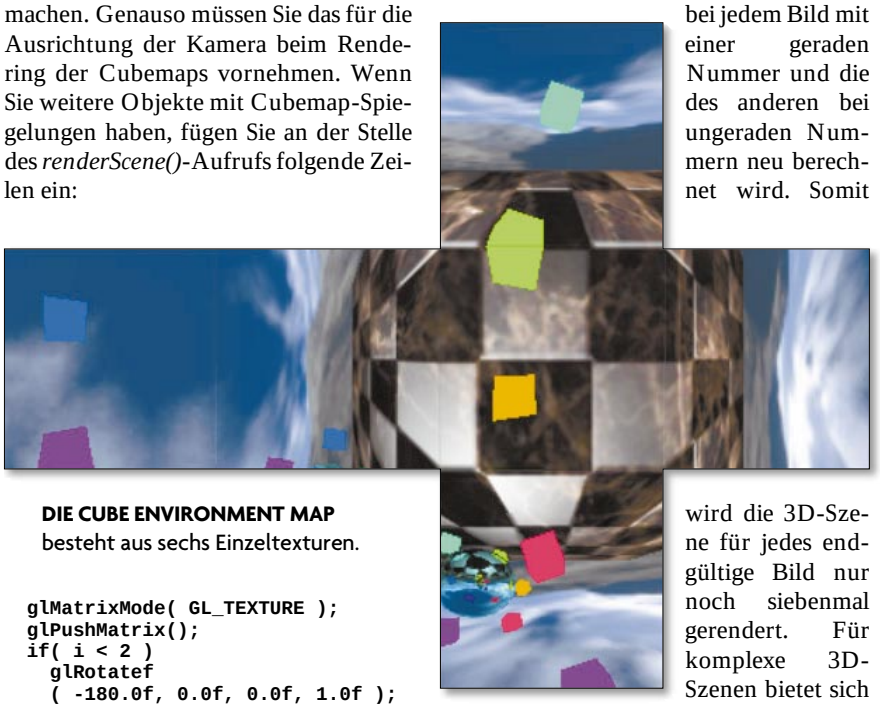
```
 glEnable
( GL_TEXTURE_CUBE_MAP_ARB );
  glFlush();
  glBindTexture
( GL_TEXTURE_CUBE_MAP_ARB,
  cubemap );

  glCopyTexSubImage2D(
    cubeMapConstants
    [ i ], 0, 0, 0, 0, 0,
    CUBEMAPSIZE, CUBEMAPSIZE );

  glFlush();
  glDisable
  ( GL_TEXTURE_CUBE_MAP_ARB );
```

Im Beispielprogramm zu dieser Ausgabe haben wir es nicht bei einem einzel-

nen spiegelnden Objekt belassen. Stattdessen sehen Sie zwei Kugeln, in denen sich die Umgebung und die jeweils andere Kugel spiegelt. In der Hauptschleife des Rendering macht es keinen Unterschied, ob Sie eines oder mehrere Objekte mit Cubemapping darstellen.

Mehrere spiegelnde Objekte

Wie lassen sich dynamische Cubemaps generieren? Wie Sie in der Hauptschleife sehen, müssen Sie die Transformationen der Modelview-Matrix rückgängig machen. Genauso müssen Sie das für die Ausrichtung der Kamera beim Rendering der Cubemaps vornehmen. Wenn Sie weitere Objekte mit Cubemap-Spiegelungen haben, fügen Sie an der Stelle des *renderScene()*-Aufrufs folgende Zeilen ein:



DIE CUBE ENVIRONMENT MAP besteht aus sechs Einzeltexturen.

```
glMatrixMode( GL_TEXTURE );
glPushMatrix();
if( i < 2 )
  glRotatef
  ( -180.0f, 0.0f, 0.0f, 1.0f );

glRotatef(
  -cubeMapRotation[ i ][ 0 ],
  cubeMapRotation[ i ][ 1 ],
  cubeMapRotation[ i ][ 2 ],
  cubeMapRotation[ i ][ 3 ] );

// Cubemapping anschalten
...

glMatrixMode( GL_MODELVIEW );
glPushMatrix();

glTranslatef( spherePos2.x,
  spherePos2.y, spherePos2.z );
glutSolidSphere( SPHERE_SIZE,
 SPHERE_SUBDIVX, SPHERE_SUBDIVY);

glMatrixMode( GL_MODELVIEW );
glPopMatrix();

// Cubemapping ausschalten
...
// sauber hinterlassen!
glMatrixMode( GL_TEXTURE );
glPopMatrix();
```

Tricks bei Cubemaps

Für jedes erzeugte Bild wird die gesamte 3D-Szene 13-mal gerendert: je sechsmal für jede Cubemap und einmal für das endgültige Bild. Reduzieren Sie diesen Aufwand.

• Für Cubemap-Texturen genügen deutlich kleinere Auflösungen als die Bildschirmauflösungen. So lassen sich mit 64 x 64 Pixeln pro Cubemap (also insgesamt 64 x 64 x 6=24576 Pixel) sehr gute Resultate erreichen. Eine Auflösung von 128 x 128 Pixeln genügt für die meisten Anwendungen.

• Aktualisieren Sie die Cubemaps nicht beim Rendern jedes einzelnen Bildes. Im Beispielprogramm erkennen Sie nicht, dass die Cubemap des einen Objekts nur bei jedem Bild mit einer geraden Nummer und die des anderen bei ungeraden Nummern neu berechnet wird. Somit wird die 3D-Szene für jedes endgültige Bild nur noch siebenmal gerendert. Für komplexe 3D-Szenen bietet sich außerdem an, die Geometrie der umgebenden Szene für die Spiegelung mit gröberen 3D-Netzen zu repräsentieren, was den Polygondurchsatz niedriger hält.

Cubemaps können Sie für viele Effekte, wie die Beleuchtungsberechnung oder als eine Art Lookup-Tabelle für die Vektornormalisierung einsetzen. Über das Dot-Product-Bumpmapping berichtet Heft 7/01, ab S. 226. Weitere Beispiele finden Sie auf den Webseiten von NVidia (www.nvidia.com) und ATI (www.ati.com). ET

Quellen:

www.dachsbacher.de/pcu
http://developer.nvidia.com/
www.ati.com/na/pages/resource_centre/dev_rel/devrel.html
„Advanced Rendering Techniques Using OpenGL“, SIGGRAPH 99 Course Notes
www.opengl.org/developers/code/sig99/index.html