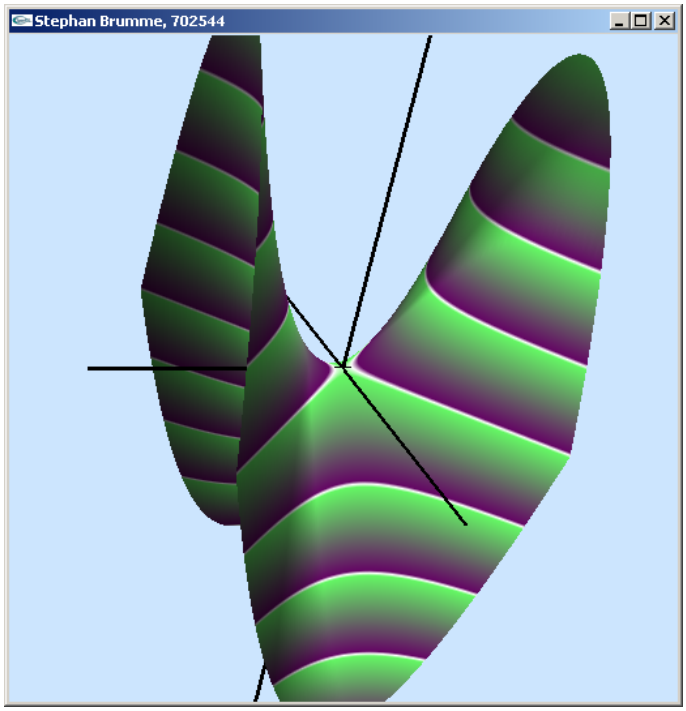
Aufgabe 23

Die Bedienung ist auf 7 Tasten reduziert:



Taste	Aktion
ESC	Programm beenden
Leertaste	Patch rotieren an/aus
+	in die Szene hineinzoomen
-	aus der Szene herauszoomen
1	starre Lichtquelle an/aus
2	rotierende Lichtquelle an/aus
n	Normalen zeichnen an/aus

Die Funktion

$$f(x,z) = x^2 - z^2$$

berechne ich nur bei der ersten Darstellung und speichere dabei die Ergebnisse im Feld `arPatch` ab. In jedem nachfolgenden Durchlauf verwende ich die Daten wieder, um so einen kleinen Geschwindigkeitsgewinn zu erzielen. Zusätzlich zu den y-Werten ist es notwendig, für die Beleuchtung die entsprechenden Normalen zu ermitteln:

$$direction = \begin{pmatrix} 0 \\ x^2 - 2z \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2x - z^2 \\ 0 \end{pmatrix}$$

$$normal = \frac{direction}{|direction|}$$

Im Code:

```
for (int x=0; x<PATCHSIZE; x++)
    for (int z=0; z<PATCHSIZE; z++)
    {
        const double x_coord = MINIMUM + x*DELTA;
        const double z_coord = MINIMUM + z*DELTA;

        // f(x,z) = x^2 - z^2
        arPatch[x][z]   = x_coord*x_coord - z_coord*z_coord;

        // f(x,z)/dx = 2x-z^2
        // f(x,z)/dz = x^2-2z
        // normal = crossproduct((0,f/dz,1), (1,f/dx,0))
        arNormals[x][z] = (Vector(0, x_coord*x_coord-2*z_coord, 1) *
                           Vector(1, 2*x_coord-z_coord*z_coord, 0)).normalized();
    }
```

Ich habe mich dafür entschieden, die Funktion mit Hilfe von Triangle-Strips darzustellen. Jeder Strip (gibt's denn dafür kein vernünftiges deutsches Wort ? Streifen hört sich auch blöd an !) verläuft entlang der z-Achse.

```
// display function
for (int x=1; x<PATCHSIZE; x++)
{
    // one triangle strip per row
    glBegin(GL_TRIANGLE_STRIP);

    // draw one row (as a triangle strip)
    for (int z=0; z<PATCHSIZE; z++)
    {
        const double x_coord = MINIMUM + x*DELTA;
        const double z_coord = MINIMUM + z*DELTA;

        // left border of the row
        glNormal3dv(arNormals[x-1][z].rep());
        glVertex3d(x_coord-DELTA, arPatch[x-1][z], z_coord);

        // right border of the row
        glNormal3dv(arNormals[x  ][z].rep());
        glVertex3d(x_coord      , arPatch[x  ][z], z_coord);
    }

    glEnd();
}
```

Die 1D-Textur wird prozedural als Farbverlauf von Grün nach Lila erzeugt. An der Indexposition Null sorgt Weiß dafür, dass y-Werte, die keine Nachkommastellen haben - z.B. 1,0 - besonders hervorgehoben werden.

```
// prozedural eine Textur erzeugen
for (int i=0; i<TEXTURE_SIZE; i++)
{
    image_[i*3+0] = 100;
    image_[i*3+1] = (i*255)/TEXTURE_SIZE;
    image_[i*3+2] = 100;
}

// represents y = -2, -1, 0, +1, +2 etc.
image_[0] = image_[1] = image_[2] = 255;
```

Die Einbindung in OpenGL erfolgt als linear interpolierte Textur, die wiederholt und mit der Helligkeit des Untergrundes vermischt wird:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

// generate a unique ID
glGenTextures  (1, &texName_);
// use this texture ID
glBindTexture  (GL_TEXTURE_1D, texName_);

// repeat texture if necessary
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
// use filter "linear" both for magnification and minification
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

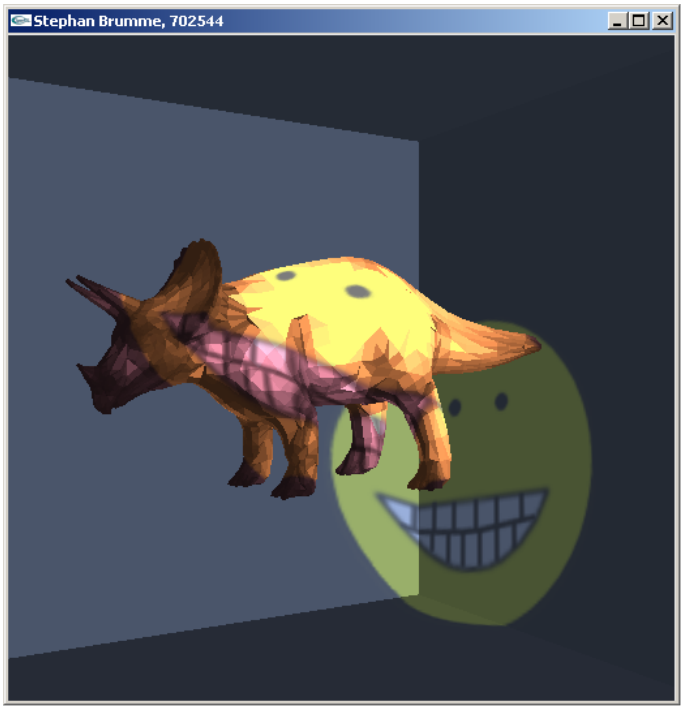
```
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexEnvf      (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,  GL_MODULATE);
// bind image to texture object
glTexImage1D   (GL_TEXTURE_1D, 0, GL_RGB, TEXTURE_SIZE,
                                0, GL_RGB, GL_UNSIGNED_BYTE, image_);
```

Die automatische Generierung der Texturkoordinaten benötigt eine Projektionsebene, die die xz-Ebene ist. Weiterhin erfolgt die Projektion im Object-Space:

```
// define texture projection plane (object space !)
static GLfloat xz_plane[] = { 0.0, 1.0, 0.0, 0.0 };
glTexGeni (GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGenfv(GL_S, GL_OBJECT_PLANE, xz_plane);

// enable automatic 1D texturing
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_1D);
```

Aufgabe 24

Das Programm kann mit diesen Tasten gesteuert werden:



Taste	Aktion
ESC	Programm beenden
Leertaste	Patch rotieren an/aus
+	in die Szene hineinzoomen
-	aus der Szene herauszoomen
q	Textur verkleinern
w	Textur vergrößern
a	Textur nach rechts schwenken
s	Textur nach links schwenken
y	Textur nach oben schwenken
x	Textur nach unten schwenken

Das Rahmenprogramm hat glücklicherweise schon sehr Arbeit mir abgenommen, so dass ich mich voll auf die Texturprojektion beschränken kann. Zuerst muss die Textur natürlich OpenGL bekannt gemacht werden, den dafür zuständigen Code kann ich 1:1 aus vorherigen Aufgaben übernehmen:

```
// generate a unique ID
glGenTextures(1, &texName_);
// use this texture ID
glBindTexture(GL_TEXTURE_2D, texName_);

// don't repeat texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
// use filter "linear" both for magnification and minification
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Eine kleine Änderung besteht darin, dass ich diesmal *nicht* den `GL_REPEAT`-Modus verwende, sondern auf `GL_CLAMP` ausweiche, da ich nur *einen* Smilies sehen will. Als nächstes ist es notwendig, die automatische Generierung der Texturkoordination einzustellen. Ich finde ich besser, die Abbildung mit Hilfe der EYE-Projektion durchzuführen, die Ebenen entsprechen den Standardwerten:

```
    // texture coordinate generation
    static const GLfloat Splane[] = { 1, 0, 0, 0 };
    static const GLfloat Tplane[] = { 0, 1, 0, 0 };
    static const GLfloat Rplane[] = { 0, 0, 1, 0 };
    static const GLfloat Qplane[] = { 0, 0, 0, 1 };
    glTexGeni (GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGeni (GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGeni (GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGeni (GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_S, GL_EYE_PLANE, Splane);
    glTexGenfv(GL_T, GL_EYE_PLANE, Tplane);
    glTexGenfv(GL_R, GL_EYE_PLANE, Rplane);
    glTexGenfv(GL_Q, GL_EYE_PLANE, Qplane);
```

Natürlich ist es noch erforderlich, die Generierung auch zu aktivieren:

```
    // switch it on
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    glEnable(GL_TEXTURE_GEN_R);
    glEnable(GL_TEXTURE_GEN_Q);
```

Eine perspektivische Projektion bildet grundsätzlich die Punkte in den Raum $[-1,1]^2$ ab. Für die Texturkoordinaten ist jedoch $[0,1]^2$ unbedingte Voraussetzung:

```
    // Texturmatrix für projektive Textur einstellen
    glMatrixMode(GL_TEXTURE);

    // texture coordinates must be within [0,1]^2
    glLoadIdentity();
    glTranslatef(0.5, 0.5, 0.0);
    glScalef    (0.5,-0.5, 1.0);
```

Der Öffnungswinkel wird durch die bereits gegebene Variable `angle_` bestimmt, die Clipping-Planes setze ich ziemlich willkürlich:

```
    // perspective projection
    gluPerspective(angle_, 1, 1, 2*radius_);
```

Um den Ursprung der Texturprojektion einzustellen, nutze ich Kugelkoordinaten, für C++ müssen diese als Radiant vorliegen:

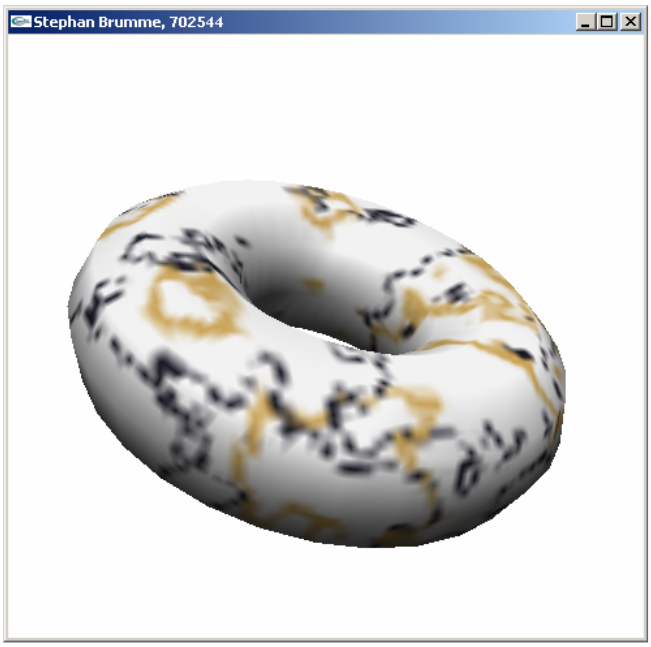
```
    // convert angles to radiant
    const double theta = theta_ / 180*M_PI;
    const double phi   = phi_   / 180*M_PI;
    // select view
    gluLookAt(radius_*sin(theta)*cos(phi), radius_*sin(phi), radius_*cos(theta)*cos(phi), //
from
              0.0, 0.0, 0.0, // to
              0.0, 1.0, 0.0);// up
```

Als letztes ist die Texturierung grundsätzlich zu aktivieren:

```
    // Texturierung einschalten
    glEnable(GL_TEXTURE_2D);
```

Aufgabe 25

Auch für die letzte Aufgabe in diesem Semester gibt es eine ergonomisch durchdachte Bedienung über ein auf Drucktasten beruhenden Interaktionsgerät, das im Volksmund als *Tastatur* bezeichnet wird:



Taste	Aktion
ESC	Programm beenden
Leertaste	Patch rotieren an/aus
+	in die Szene hineinzoomen
-	aus der Szene herauszoomen
t	zwischen Torus und Quadrat umschalten
q	zweifarbiger Marmor
w	dunkler Marmor
e	bräunlicher Marmor
1-6	Tessellationsgrad des Torus bestimmen (6 = maximal)

Zuerst möchte ich darauf hinweisen, dass die Benutzung von 3D-Texturen relativ aktuelle Hardware und entsprechende Treiber verlangt. Minimalanforderung ist eine Grafikkarte der GeForce-Generation und eine GHz-CPU. OpenGL muss min. in der Version 1.2 vorliegen. Auf den meisten Systemen, die diesen Anforderungen genügen, wird wahrscheinlich doch die Texturierung in Software durchgeführt, ich war zufrieden, obiges Bild mit einer Geschwindigkeit von etwa 2 Bildern/Sekunde zu erhalten.

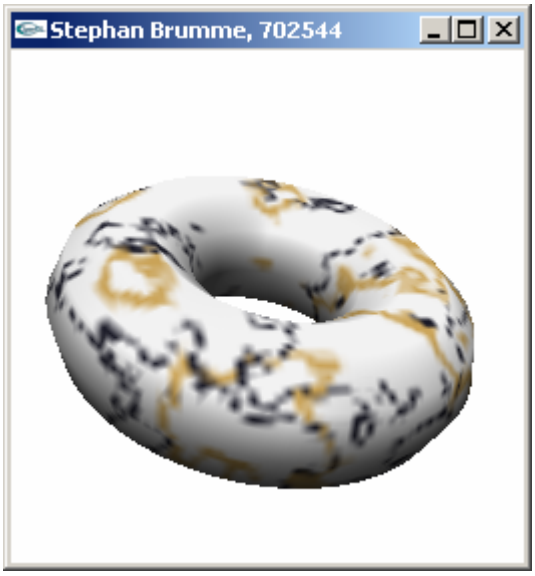
Da es relativ einfach ist, 3D-Texturen in OpenGL zu *verwenden*, gehe ich darauf zuerst ein. Die *prozedurale Generierung* erkläre ich im Anschluss, auch wenn beide Abschnitte im Quelltext im umgekehrter Reihenfolge auftauchen.

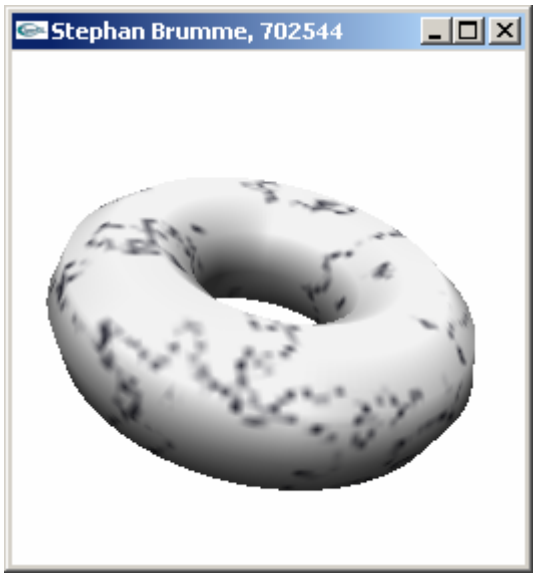
Es gibt fast keinen Unterschied in der prinzipiellen Vorgehensweise bei der Benutzung von 3D-Texturen im Vergleich zu herkömmlichen 2D-Texturen. Statt des Schlüsselwortes `GL_TEXTURE_2D` ist eben `GL_TEXTURE_3D` zu schreiben und die Einstellungen für die Textur-Parameter *s* und *t* sind auch auf *r* auszuweiten. Für das Beispiel wird eine lineare Filterung aktiviert, sie ist zwar langsam, liefert aber recht gute Bilder. Die Textur wird notfalls wiederholt (`GL_REPEAT`) und die Beleuchtung fließt ebenfalls in die Berechnung mit ein (`GL_MODULATE`).

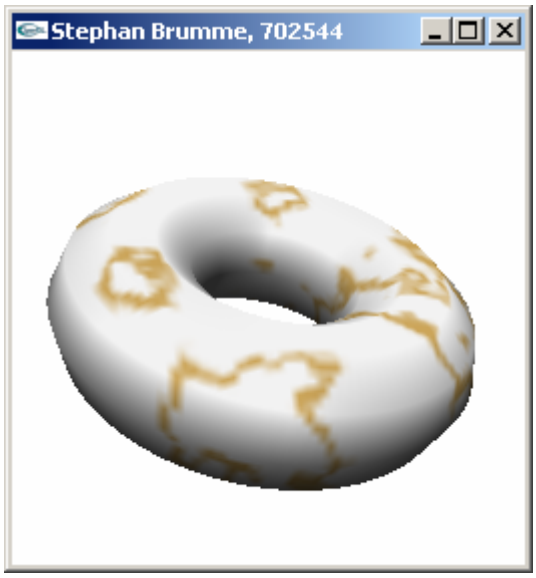
```
// Textur parameter
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glGenTextures(1, &texName1_);

// texture I
glBindTexture(GL_TEXTURE_3D, texName1_);
```

```
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S,     GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T,     GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R,     GL_REPEAT);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, w, h, d, 0, GL_RGB, GL_UNSIGNED_BYTE, img1);
```

Ich habe auf meiner GeForce2 MX mit 32 MB Speicher (+64 MB per AGP) lediglich Texturgrößen bis 64x64x64 erreichen können, für bessere optische Ergebnisse wäre etwas mehr wünschenswert gewesen, da selbst 256x256x256 nur ca. 50 MB belegen. Egal. Um meine Verwunderung über diesen seltsamen Engpass zu unterdrücken, werden vom Programm gleich 3 Marmor-Texturen erzeugt, zwischen denen man hin- und herwechseln kann. Hier alle drei zum Vergleich:







Die prozedurale Erzeugung beruht auf der Perlin-Noise-Funktion. Im vorgegebenen Rahmenprogramm ist sie bereits vorhanden, sie bekommt als Parameter eine 3D-Koordinate und liefert dann einen Skalar zurück. Meine Erfahrungen haben gezeigt, dass dieser zwischen -0,5 und +0,5 liegt.

In drei verschachtelten Schleifen (je eine für die x-, y- und z-Dimension) berechne ich jeden einzelnen Punkt der Texturen. Dazu muss ich den Offset im Speicher bestimmen:

```
// texture offset
const int baseoffset = 3 * (x*h*d + y*d +z);
```

Beide Marmorstrukturen setzen auf der Perlin-Noise-Funktion auf, ihre genaue Funktionsweise entstand aber mehr einem Experimentierprozess und ist in keiner Art wissenschaftlich begründet. Im wesentlich kommt eine 4-oktavige Funktion mit einer Persistenz von 0,25 zum Zuge, ich störe leicht den x-Wert in jeder Oktave und sorge für ein lokales Rauschen durch eine zusätzliche Perlin-Noise-Funktion.

```
// little noise
float point1[3];
point1[0] = (4.0*x)/w;
point1[1] = (4.0*y)/h;
point1[2] = (4.0*z)/d;

// little noise (differs from first one)
float point2[3];
point2[1] = (4.0*x)/w;
point2[2] = (4.0*y)/h;
point2[0] = (4.0*z)/d;

// really noisy
float point3[3];
point3[1] = (16.0*x)/w;
point3[2] = (16.0*y)/h;
point3[0] = (16.0*z)/d;

float persistence1 = 0.25;
float persistence2 = 0.25;
float noise1 = 0.0;
float noise2 = 0.0;

// 4 octaves
for (int octave = 0; octave<4; octave++)
{
    // brown marble's noise
    // shift point
    point1[0] += 0.4*cos(noise1);
    point1[1] += 0.4*sin(noise1+PI/2);
    // get noise
    float currentnoise1 = (Noise::noise3(point1)+0.5) * persistence1;
    noise1 += currentnoise1;
    // add a bit of noisy noise (?!)
    noise1 += 0.2 * Noise::noise3(point3) * persistence1;

    // dark marble's noise
    point2[0] += 0.4*cos(noise2);
    point2[1] += 0.4*sin(noise2+PI/2);
    // get noise
    float currentnoise2 = (Noise::noise3(point2)+0.5) * persistence2;
    noise2 += currentnoise2;
    // add a bit of noisy noise (?!)
    noise2 += 0.2 * Noise::noise3(point3) * persistence2;
}
```

Ich habe dafür Sorge zu tragen, dass die Ergebnisse `noise1` und `noise2` auch tatsächlich im Intervall [0;1] liegen:

```
// clamp noises
if (noise1 < 0) noise1 = 0;
if (noise1 > 1) noise1 = 1;
if (noise2 < 0) noise2 = 0;
if (noise2 > 1) noise2 = 1;
```

Im weiteren Verlauf wird die Texturfarbe `color` generiert. Dabei setze ich intensiv die Farbmischfunktion `mix` und die Interpolationsfunktion `smoothstep` ein:

```
Vector mix(const Vector& color1, const Vector& color2, const float alpha)
{
    return (1-alpha)*color1 + alpha*color2;
}

inline float smoothstep(float a, float b, float x) {
    if(x<=a) return 0.0;
    if(x>=b) return 1.0;
    x = (x-a)/(b-a); // normalized interval [0,1]
    return x*x*(3-2*x);
}
```

Der Braunton entsteht im Intervall [0,55; 0,68], wobei in [0,55; 0,60] und [0,65; 0,68] ein weicher Übergang erfolgt. In allen anderen Bereichen hat die Textur eine sehr helle Farbe (nahezu Weiß):

```
    // brown marble
    color = mix(bright, brown,  smoothstep(0.55, 0.60, noise1));
    color = mix(color,  bright, smoothstep(0.65, 0.68, noise1));
```

Diesen Wert kann ich gleich für Textur III benutzen:

```
    // texture III: brown marble
    img3[baseoffset+0] = color[0]*255;
    img3[baseoffset+1] = color[1]*255;
    img3[baseoffset+2] = color[2]*255;
```

Die Verwendung einer zweiten Marmorfarbe (sehr dunkel, fast ein Schwarz) wirft das Problem auf, dass ich zwar mit `color` die neue Farbe vermischen muss, die alten aber nicht verloren gehen dürfen bzw. weiche Übergänge erwünscht sind. Deshalb sind `if`s notwendig:

```
    // dark marble
    if (noise2 <= 0.55)
        color = mix(color, dark,  smoothstep(0.50, 0.52, noise2));
    if (noise2 >= 0.52)
        color = mix(dark,  color, smoothstep(0.52, 0.55, noise2));
```

Somit ist Textur I fertig:

```
    // texture I: combined
    img1[baseoffset+0] = color[0]*255;
    img1[baseoffset+1] = color[1]*255;
    img1[baseoffset+2] = color[2]*255;
```

Textur II soll nur schwarzen Marmor aufweisen. Da dieser aber bereits als Mischton mit dem braunen Marmor berechnet wurde, komme ich um eine erneute Berechnung nicht drum rum:

```
    // dark marble (for texture II)
    color = mix(bright, dark,   smoothstep(0.50, 0.52, noise2));
    color = mix(color,  bright, smoothstep(0.52, 0.55, noise2));

    // texture II: dark marble
    img2[baseoffset+0] = color[0]*255;
    img2[baseoffset+1] = color[1]*255;
    img2[baseoffset+2] = color[2]*255;
```

Quelltext

Aufgabe 23:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 9
//

#include "cgcontour.h"
#include "vector.h"


// degree of tessellation
const int    PATCHSIZE =  100;
// range of x,z
const double MINIMUM   = -2.0;
const double MAXIMUM   = +2.0;
const double SPAN      = MAXIMUM - MINIMUM;
const double DELTA     = SPAN/(PATCHSIZE-1);

// store values of the function
bool calculationdone = false;
double arPatch[PATCHSIZE][PATCHSIZE];
// corresponding normals
Vector arNormals[PATCHSIZE][PATCHSIZE];



CGContour::CGContour() {
    run_ = false;
    normals_ = false;
    zoom_ = 0.4;
}

CGContour::~CGContour() {
}

void CGContour::drawScene() {

    glPushMatrix();

        glRotated(30, 1.0, 0.0, 0.0);
        glScaled(zoom_, zoom_, zoom_);

        // Achsen
        glLineWidth(3.0);
        glColor3f(0.0, 0.0, 0.0);

        glDisable(GL_LIGHTING);
        glBegin(GL_LINES);
        glVertex3f( 0.0, 0.0, 0.0);
        glVertex3f( 3.0, 0.0, 0.0);
        glVertex3f( 0.0, 0.0, 0.0);
        glVertex3f( 0.0, 6.0, 0.0);
        glVertex3f( 0.0, 0.0, 0.0);
        glVertex3f( 0.0, 0.0, 3.0);
        glEnd();
        glEnable(GL_LIGHTING);

        // Licht 1
        static GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};      //  diffuse light
        static GLfloat light_position[]= {0.0, 1.0, 0.0, 0.0};      //  infinite light
location

        glLightfv(GL_LIGHT1, GL_DIFFUSE,  light_diffuse );
        glLightfv(GL_LIGHT1, GL_POSITION, light_position);

        // Funktion
        drawFunction();
```

```
    glPopMatrix();
}

void CGContour::drawFunction() {
    // Hier die Funktion zeichnen.

    // perform calculation only once
    if (!calculationdone)
    {
        for (int x=0; x<PATCHSIZE; x++)
            for (int z=0; z<PATCHSIZE; z++)
            {
                const double x_coord = MINIMUM + x*DELTA;
                const double z_coord = MINIMUM + z*DELTA;

                // f(x,z) = x^2 - z^2
                arPatch[x][z]   = x_coord*x_coord - z_coord*z_coord;

                // f(x,z)/dx = 2x-z^2
                // f(x,z)/dz = x^2-2z
                // normal = crossproduct((0,f/dz,1), (1,f/dx,0))
                arNormals[x][z] = (Vector(0, x_coord*x_coord-2*z_coord, 1) *
                                   Vector(1, 2*x_coord-z_coord*z_coord, 0)).normalized();
            }

        // function doesn't change
        calculationdone = true;
    }

    // define texture projection plane (object space !)
    static GLfloat xz_plane[] = { 0.0, 1.0, 0.0, 0.0 };
    glTexGeni (GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glTexGenfv(GL_S, GL_OBJECT_PLANE, xz_plane);

    // enable automatic 1D texturing
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_1D);

    // display function
    for (int x=1; x<PATCHSIZE; x++)
    {
        // one triangle strip per row
        glBegin(GL_TRIANGLE_STRIP);

        // draw one row (as a triangle strip)
        for (int z=0; z<PATCHSIZE; z++)
        {
            const double x_coord = MINIMUM + x*DELTA;
            const double z_coord = MINIMUM + z*DELTA;

            // left border of the row
            glNormal3dv(arNormals[x-1][z].rep());
            glVertex3d(x_coord-DELTA, arPatch[x-1][z], z_coord);

            // right border of the row
            glNormal3dv(arNormals[x  ][z].rep());
            glVertex3d(x_coord      , arPatch[x  ][z], z_coord);
        }

        glEnd();
    }
    glDisable(GL_TEXTURE_1D);

    // draw normals if necessary
    if (normals_)
    {
        // no lighting
        glDisable(GL_LIGHTING);

        // draw all normals
        for (x=1; x<PATCHSIZE; x++)
        {
            glLineWidth(1);
            glColor3f(1,0,0);
            glBegin(GL_LINES);
```

```
            for (int z=0; z<PATCHSIZE; z++)
            {
                const double x_coord = MINIMUM + x*DELTA;
                const double z_coord = MINIMUM + z*DELTA;

                // short normals for better visualization
                Vector normal = arNormals[x][z] * 0.2;

                glVertex3d(x_coord, arPatch[x][z], z_coord);
                glVertex3d(x_coord+normal[0], arPatch[x][z]+normal[1], z_coord+normal[2]);
            }

            glEnd();
        }
        glEnable(GL_LIGHTING);
    }
}

void CGContour::createTexture() {
    // prozedural eine Textur erzeugen
    for (int i=0; i<TEXTURE_SIZE; i++)
    {
        image_[i*3+0] = 100;
        image_[i*3+1] = (i*255)/TEXTURE_SIZE;
        image_[i*3+2] = 100;
    }

    // represents y = -2, -1, 0, +1, +2 etc.
    image_[0] = image_[1] = image_[2] = 255;


    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    // generate a unique ID
    glGenTextures  (1, &texName_);
    // use this texture ID
    glBindTexture  (GL_TEXTURE_1D, texName_);

    // repeat texture if necessary
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    // use filter "linear" both for magnification and minification
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexEnvf      (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,  GL_MODULATE);
    // bind image to texture object
    glTexImage1D   (GL_TEXTURE_1D, 0, GL_RGB, TEXTURE_SIZE,
                                   0, GL_RGB, GL_UNSIGNED_BYTE, image_);
}

void CGContour::onInit() {

    // Tiefen Test aktivieren
    glEnable(GL_DEPTH_TEST);

    // Smooth Schattierung aktivieren
    glShadeModel(GL_SMOOTH);

    // Projection
    glMatrixMode(GL_PROJECTION);
    gluPerspective(40.0, 1.0, 1.0, 20.0);

    // LookAt
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(
        0.0, 0.0, 4.0,  // from (0,0,4)
        0.0, 0.0, 0.0,  // to (0,0,0)
        0.0, 1.0, 0.);  // up

    glClearColor(0.8, 0.9, 1.0, 0.0);

    createTexture();

    // set lights
    static GLfloat light_position[]= {0.0, 0.0, 1.0, 0.0};
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

```
    glEnable(GL_LIGHT0); light0_ = true;
    glEnable(GL_LIGHT1); light1_ = true;

    glEnable(GL_LIGHTING);

    static float material_diffuse[]= {0.6, 0.6, 0.6, 1.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, material_diffuse);
}

void CGContour::onSize(unsigned int newWidth,unsigned int newHeight) {
    width_ = newWidth;
    height_ = newHeight;
    glMatrixMode(GL_PROJECTION);
    glViewport(0, 0, width_ - 1, height_ - 1);
    glLoadIdentity();
    gluPerspective(40.0,float(width_)/float(height_),2.0, 100.0);
    glMatrixMode(GL_MODELVIEW);
}

void CGContour::onKey(unsigned char key) {
    switch (key) {
    case 27: { exit(0); break; }
    case '+': { zoom_*= 1.1; break; }
    case '-': { zoom_/= 1.1; break; }
    case ' ': { run_ = !run_; break; }
    case '1': { if (light0_) glDisable(GL_LIGHT0);
                else         glEnable (GL_LIGHT0);
                light0_ = !light0_;
                break; };
    case '2': { if (light1_) glDisable(GL_LIGHT1);
                else         glEnable (GL_LIGHT1);
                light1_ = !light1_;
                break; };
    case 'n': { normals_ = !normals_; break; }
    }
    onDraw();
}

void CGContour::onIdle() {

    if (run_) {
        glRotatef(.5, 0.0, 1.0, 0.0);
    }
    onDraw();
}

void CGContour::onDraw() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawScene();

    swapBuffers();
}

// Hauptprogramm
int main(int argc, char* argv[]) {
    // Erzeuge eine Instanz der Beispiel-Anwendung:
    CGContour sample;

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "1          starre Lichtquelle an/aus" << endl
         << "2          rotierende Lichtquelle an/aus" << endl
         << "+          Hineinzoomen" << endl
         << "-          Herauszoomen" << endl
         << "n          Normalen zeichnen an/aus" << endl;

    // Starte die Beispiel-Anwendung:
    sample.start("Stephan Brumme, 702544", true, 512, 512);
    return(0);
}
```

Aufgabe 24 manifestiert sich vor allem in cgprojtexture.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 9
//

#include "cgprojtexture.h"
#include "vector.h"
#include <fstream.h>
#include <stdlib.h>

#ifndef M_PI
const double M_PI = 3.14159265358979323846;
#endif

const unsigned int BUF_SIZE = 1024;

CGProjTexture::CGProjTexture(char* filename) {
    filename_ = filename;

    run_ = false;
    zoom_ = 0.16;

    angle_ = 15.0;
    radius_ = 4.0;
    theta_ = 0.0;
    phi_ = 0.0;
}

CGProjTexture::~CGProjTexture() {
}

void CGProjTexture::drawScene() {

    // texture coordinate generation
    static const GLfloat Splane[] = { 1, 0, 0, 0 };
    static const GLfloat Tplane[] = { 0, 1, 0, 0 };
    static const GLfloat Rplane[] = { 0, 0, 1, 0 };
    static const GLfloat Qplane[] = { 0, 0, 0, 1 };
    glTexGeni (GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGeni (GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGeni (GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGeni (GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_S, GL_EYE_PLANE, Splane);
    glTexGenfv(GL_T, GL_EYE_PLANE, Tplane);
    glTexGenfv(GL_R, GL_EYE_PLANE, Rplane);
    glTexGenfv(GL_Q, GL_EYE_PLANE, Qplane);

    // switch it on
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
    glEnable(GL_TEXTURE_GEN_R);
    glEnable(GL_TEXTURE_GEN_Q);

    // Texturmatrix für projektive Textur einstellen
    glMatrixMode(GL_TEXTURE);

    // texture coordinates must be within [0,1]^2
    glLoadIdentity();
    glTranslatef(0.5, 0.5, 0.0);
    glScalef    (0.5,-0.5, 1.0);

    // perspective projection
    gluPerspective(angle_, 1, 1, 2*radius_);

    // convert angles to radiant
    const double theta = theta_ / 180*M_PI;
    const double phi   = phi_   / 180*M_PI;
    // select view
    gluLookAt(radius_*sin(theta)*cos(phi), radius_*sin(phi), radius_*cos(theta)*cos(phi), //
from
              0.0, 0.0, 0.0, // to
```

```
                0.0, 1.0, 0.0);// up

    // Texturierung einschalten
    glEnable(GL_TEXTURE_2D);

    glMatrixMode(GL_MODELVIEW);

    // draw dino
    glPushMatrix();
    glScaled(zoom_, zoom_, zoom_);
    drawObject();
    glPopMatrix();

    // gray cube
    static float material_diffuse[] = {0.7, 0.8, 1.0, 1.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, material_diffuse);
    glCullFace(GL_FRONT);
    glutSolidCube(3.0);
    glCullFace(GL_BACK);
}

void CGProjTexture::drawObject() {
    static float material_diffuse[] = {1.0, 0.6, 0.7, 1.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, material_diffuse);

    // create display list (done only once)
    static GLuint cache = 0;
    if (cache == 0) {
        cache = glGenLists(1);
        glNewList(cache, GL_COMPILE);
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();

        ifstream s("triceratops.txt");
        char buf[BUF_SIZE];

        glBegin(GL_TRIANGLES);
        Vector v[3];
        while (!s.eof()) {
            s >> buf;

            while (s.good()) {
                s >> v[0] >> v[1] >> v[2];
                Vector n = (v[1]-v[0])*(v[2]-v[0]);
                glNormal3dv(n.rep());
                glVertex3dv(v[0].rep());
                glVertex3dv(v[1].rep());
                glVertex3dv(v[2].rep());
            }
            if (s.rdstate() & ios::failbit) {
                s.clear(s.rdstate() & ~ios::failbit);
            }
        }
        glEnd();
        glPopMatrix();
        glEndList();
    }

    // execute display list
    glCallList(cache);
}

void CGProjTexture::readImage() {

    char buf[BUF_SIZE];

    // create input stream
    ifstream in(filename_);
    if(!in) {
        cerr << "no ppm image" << endl;
        exit(-1);
    }

    cout << "Reading image from file \"" << filename_ << "\" " << endl;

    // PPM header
```

```
#ifdef _MSC_VER
    in >> binary;
#endif

    // Read "P6"
    char ppm;
    in >> ppm; if(!in.good() || ppm != 'P') { cerr << "ppm format error" << endl; }
    in >> ppm; if(!in.good() || ppm != '6') { cerr << "ppm format error" << endl; }

    // forward to next line
    in.getline(buf, BUF_SIZE);

    // normally read comments, but we assume that no comments are there
    // in.getline(buf, BUF_SIZE);
    // while (buf[0] == '#')
    //     in.getline(buf, BUF_SIZE);

    // Read width and height
    in >> texWidth_; if(!in.good()) { cerr << "ppm format error" << endl; }
    in >> texHeight_; if(!in.good()) { cerr << "ppm format error" << endl; }

    // Read 255
    unsigned int res;
    in >> res; if(!in.good() || res != 255) { cerr << "ppm format error" << endl; }

    // forward to next line
    in.getline(buf, BUF_SIZE);

    image_ = new GLubyte [3 * texWidth_ * texHeight_];
    in.read(image_, 3 * texWidth_ * texHeight_);
    if(!in.good()) { cerr << "ppm format error" << endl; }

    in.close();
}

void CGProjTexture::onInit() {

    // Tiefen Test aktivieren
    glEnable(GL_DEPTH_TEST);

    // Smooth Schattierung aktivieren
    glShadeModel(GL_SMOOTH);

    // Culling aktivieren
    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);

    // Projection
    glMatrixMode(GL_PROJECTION);
    gluPerspective(40.0, 1.0, 1.0, 20.0);

    // LookAt
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(
        0.0, 0.0, 4.0,  // from (0,0,4)
        0.0, 0.0, 0.0,  // to (0,0,0)
        0.0, 1.0, 0.);  // up

    glClearColor(0.9,0.9,0.9,1.0);

    // Textur einlesen
    readImage();
    // Texturobjekt anlegen, Grundeinstellungen für projektive Textur festlegen

    ////////////////////////////////
    // create texture

    // generate a unique ID
    glGenTextures(1, &texName_);
    // use this texture ID
    glBindTexture(GL_TEXTURE_2D, texName_);

    // don't repeat texture
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    // use filter "linear" both for magnification and minification
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    // bind image to texture object
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth_, texHeight_, 0, GL_RGB, GL_UNSIGNED_BYTE,
image_);

    // texture blending
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    // OpenGL Lichtquellen
    static GLfloat light_diffuse[] = {0.7, 0.7, 0.7, 1.0};
    static GLfloat light_position1[] = {1.0, 0.5, 1.0, 0.0};
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position1);
    glEnable(GL_LIGHT0);

    static GLfloat light_position2[] = {-0.5, -0.4, -1.0, 0.0};
    glLightfv(GL_LIGHT1, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT1, GL_POSITION, light_position2);
    glEnable(GL_LIGHT1);
    glEnable(GL_LIGHTING);
}

void CGProjTexture::onSize(unsigned int newWidth,unsigned int newHeight) {
    width_ = newWidth;
    height_ = newHeight;
    glMatrixMode(GL_PROJECTION);
    glViewport(0, 0, width_ - 1, height_ - 1);
    glLoadIdentity();
    gluPerspective(40.0,float(width_)/float(height_),1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}

void CGProjTexture::onKey(unsigned char key) {
    switch (key) {
    case 27: { exit(0); break; }
    case '+': { zoom_*= 1.1; break; }
    case '-': { zoom_/= 1.1; break; }
    case ' ': { run_ = !run_; break; }
    case 'w': { if (angle_ < 180) angle_ *= 1.05; break; }
    case 'q': { if (angle_ > 0.001) angle_ /= 1.05; break; }
    case 's': { theta_ += 2; if (theta_>=360) theta_-=360; break; }
    case 'a': { theta_ -= 2; if (theta_<=0) theta_+=360; break; }
    case 'x': { phi_ += 2; if (phi_>=90) phi_=90; break; }
    case 'y': { phi_ -= 2; if (phi_<=-90) phi_=-90;break; }
    }
    onDraw();
}

void CGProjTexture::onIdle() {

    if (run_) {
        glRotatef(.5, 0.0, 1.0, 0.0);
    }
    onDraw();
}

void CGProjTexture::onDraw() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawScene();

    swapBuffers();
}

// Hauptprogramm
int main(int argc, char* argv[]) {
    // Erzeuge eine Instanz der Beispiel-Anwendung:
    CGProjTexture sample("smily.ppm");

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "+          Hineinzoomen" << endl
         << "-          Herauszoomen" << endl
```

```
        << "q          Textur verkleinern" << endl
        << "w          Textur vergroessern" << endl
        << "a          Textur nach rechts schwenken" << endl
        << "s          Textur nach links schwenken" << endl
        << "y          Textur nach oben schwenken" << endl
        << "x          Textur nach unten schwenken" << endl;

    // Starte die Beispiel-Anwendung:
    sample.start("Stephan Brumme, 702544", true, 512, 512);
    return(0);
}
```

Aufgabe 25:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 9
//

#include "cgtexture.h"
#include "vector.h"
#include "noise.h"

#include <fstream.h>
#include <stdio.h>
#include "glext.h"

#ifndef APIENTRY
#define WIN32_LEAN_AND_MEAN 1
#include <windows.h>
#endif /*APIENTRY*/

#ifdef _WIN32

/* This has already been done by glext.h */
/*typedef void (APIENTRY* PFNGLTEXIMAGE3DEXTPROC)(GLenum target, GLint level, GLenum
internalFormat,
                                                 GLsizei width, GLsizei height, GLsizei depth,
                                                 GLint border, GLenum format, GLenum type,
                                                 const GLvoid *image);*/

PFNGLTEXIMAGE3DPROC glTexImage3D;

#endif _WIN32*/

int isExtensionSupported(const char *extension) {
    const GLubyte *extensions = NULL;
    const GLubyte *start;
    GLubyte *where, *terminator;

    /* Extension names should not have spaces. */
    where = (GLubyte*) strchr(extension, ' ');
    if (where || *extension == '\0')
        return 0;

    extensions = glGetString(GL_EXTENSIONS);
    /* It takes a bit of care to bew fol-proof about parsing the
       OpenGL extensions string. Donn't be fooled by sub-strings,
       etc. */
    start = extensions;
    for(;;) {
        where = (GLubyte*) strstr((const char*) start, extension);
        if (!where)
            break;
        terminator = where + strlen(extension);
        if (where == start ||*(where-1)==' ')
            if (*terminator == ' ' || *terminator == '\0')
                return 1;
        start = terminator;
    }
    return 0;
}

void buildAvailableExtensions() {
    // Beispiel zum Abfragen des Extensionstrings ing OpenGL:
    // int hasImaging = isExtensionSupported("GL_ARB_imaging");
    #ifdef _WIN32
        glTexImage3D = (PFNGLTEXIMAGE3DPROC)
        wglGetProcAddress("glTexImage3D");
    #endif
};


#ifndef PI
const double PI = 3.14159265358979323846;
```

```
#endif


CGTexture::CGTexture() {
    run_ = false;
    zoom_ = 1.0;
    usetorus_ = true;

    texWidth_ = texHeight_ = texDepth_ = 64;
}

CGTexture::~CGTexture() {
}

void CGTexture::drawScene() {

    glPushMatrix();
    glRotatef(200, 0.0, 1.0, 0.0);
    glRotatef(55,1,0,0);
    glScaled(zoom_, zoom_, zoom_);

    glEnable(GL_TEXTURE_3D);

    if (usetorus_)
    {
        // our oopy-doopy torus
        drawObject();
    }
    else
    {
        // a simple quad to test the 3D texture
        static float depth = 0;
        static float increment = 0.01;
        glDisable(GL_LIGHTING);
        glBegin(GL_QUADS);
        glTexCoord3f(0,0,depth); glVertex3f(-1,-1,0);
        glTexCoord3f(1,0,depth); glVertex3f(+1,-1,0);
        glTexCoord3f(1,1,depth); glVertex3f(+1,+1,0);
        glTexCoord3f(0,1,depth); glVertex3f(-1,+1,0);
        glEnd();
        glEnable(GL_LIGHTING);

        depth += increment;
        if (depth < 0 || depth > 1)
        {
            increment = -increment;
            depth += increment;
        }
    }

    glDisable(GL_TEXTURE_3D);
    glPopMatrix();
}

void CGTexture::drawObject() {
    // draw torus
    glCallList(torus_);
}

void CGTexture::buildPatch(int detail) {
    if (glIsList(torus_)) glDeleteLists(torus_, 1);
    GLint i, j;
    float theta1, phi1, theta2, phi2, rings, sides;
    float v0[03], v1[3], v2[3], v3[3];
    float t0[03], t1[3], t2[3], t3[3];
    float n0[3], n1[3], n2[3], n3[3];
    float innerRadius=0.4;
    float outerRadius=0.8;
    float scalFac;

    rings = detail;
    sides = 10;
    scalFac=1/((outerRadius+innerRadius)*2);

    glNewList(torus_, GL_COMPILE);
    glBegin(GL_QUADS);
```

```
    for (i = 0; i < rings; i++) {
        theta1 = (float)i * 2.0 * PI / rings;
        theta2 = (float)(i + 1) * 2.0 * PI / rings;
        for (j = 0; j < sides; j++) {
            phi1 = (float)j * 2.0 * PI / sides;
            phi2 = (float)(j + 1) * 2.0 * PI / sides;

            v0[0] = cos(theta1) * (outerRadius + innerRadius * cos(phi1));
            v0[1] = -sin(theta1) * (outerRadius + innerRadius * cos(phi1));
            v0[2] = innerRadius * sin(phi1);

            v1[0] = cos(theta2) * (outerRadius + innerRadius * cos(phi1));
            v1[1] = -sin(theta2) * (outerRadius + innerRadius * cos(phi1));
            v1[2] = innerRadius * sin(phi1);
            v2[0] = cos(theta2) * (outerRadius + innerRadius * cos(phi2));
            v2[1] = -sin(theta2) * (outerRadius + innerRadius * cos(phi2));
            v2[2] = innerRadius * sin(phi2);

            v3[0] = cos(theta1) * (outerRadius + innerRadius * cos(phi2));
            v3[1] = -sin(theta1) * (outerRadius + innerRadius * cos(phi2));
            v3[2] = innerRadius * sin(phi2);

            n0[0] = cos(theta1) * (cos(phi1));
            n0[1] = -sin(theta1) * (cos(phi1));
            n0[2] = sin(phi1);

            n1[0] = cos(theta2) * (cos(phi1));
            n1[1] = -sin(theta2) * (cos(phi1));
            n1[2] = sin(phi1);

            n2[0] = cos(theta2) * (cos(phi2));
            n2[1] = -sin(theta2) * (cos(phi2));
            n2[2] = sin(phi2);

            n3[0] = cos(theta1) * (cos(phi2));
            n3[1] = -sin(theta1) * (cos(phi2));
            n3[2] = sin(phi2);

            t0[0] = v0[0]*scalFac + 0.5;
            t0[1] = v0[1]*scalFac + 0.5;
            t0[2] = v0[2]*scalFac + 0.5;

            t1[0] = v1[0]*scalFac + 0.5;
            t1[1] = v1[1]*scalFac + 0.5;
            t1[2] = v1[2]*scalFac + 0.5;

            t2[0] = v2[0]*scalFac + 0.5;
            t2[1] = v2[1]*scalFac + 0.5;
            t2[2] = v2[2]*scalFac + 0.5;

            t3[0] = v3[0]*scalFac + 0.5;
            t3[1] = v3[1]*scalFac + 0.5;
            t3[2] = v3[2]*scalFac + 0.5;

                glNormal3fv(n3); glTexCoord3fv(t3); glVertex3fv(v3);
                glNormal3fv(n2); glTexCoord3fv(t2); glVertex3fv(v2);
                glNormal3fv(n1); glTexCoord3fv(t1); glVertex3fv(v1);
                glNormal3fv(n0); glTexCoord3fv(t0); glVertex3fv(v0);
        }
    }
    glEnd();
    glEndList();
}

//
// Einige nützliche Funktionen:
//

// distance from a point
inline float distance(float x, float y, float z, float cx, float cy, float cz) {
  float dx = x-cx;
  float dy = y-cy;
  float dz = y-cz;
  return sqrt(dx*dx + dy*dy + dz*dz);
}
```

```
// clamp x to be between a and b
inline float clamp(float x, float a, float b) {
    return (x < a ? a : (x > b ? b : x));
}

inline float smoothstep(float a, float b, float x) {
  if(x<=a) return 0.0;
  if(x>=b) return 1.0;
  x = (x-a)/(b-a); // normalized interval [0,1]
  return x*x*(3-2*x);
}

// Spline Interpolation
Vector spline(float x, int nknots, Vector* knot) {
    int span;
    int nspans = nknots-3;

    Vector c0,c1,c2,c3;

    if(nspans<1) {
        cout << "Error in spline function" << endl;
        return NULL;
    }

    x=clamp(x,0,1)*nspans;
    span = (int) x;
    if(span >= nknots-3) span = nknots-3;
    x-=span;

    // Horner
    c3 = -0.5 * knot[span] + 1.5 * knot[1+span] - 1.5 * knot[2+span] + 0.5 * knot[3+span];
    c2 =  1.0 * knot[span] - 2.5 * knot[1+span] + 2.0 * knot[2+span] - 0.5 * knot[3+span];
    c1 = -0.5 * knot[span] + 0.0 * knot[1+span] + 0.5 * knot[2+span] + 0.0 * knot[3+span];
    c0 =  0.0 * knot[span] + 1.0 * knot[1+span] + 0.0 * knot[2+span] + 0.0 * knot[3+span];

    return ((c3*x + c2)*x + c1)*x + c0;
}

Vector mix(const Vector& color1, const Vector& color2, const float alpha)
{
    return (1-alpha)*color1 + alpha*color2;
}


static Vector dark       = Vector(0.10, 0.10, 0.15);
static Vector bright     = Vector(0.95, 0.95, 0.95);
static Vector brown      = Vector(0.80, 0.65, 0.35);

void CGTexture::createNoiseTexture3D() {

    int w = texWidth_;
    int h = texHeight_;
    int d = texDepth_;

    // Notwendiger Aufruf zur Initialisierung der Hashtable-Struktur:
    Noise::init();

    GLubyte *img1 = new GLubyte[w * h * d * 3]; // 3D Textur I
    GLubyte *img2 = new GLubyte[w * h * d * 3]; // 3D Textur II
    GLubyte *img3 = new GLubyte[w * h * d * 3]; // 3D Textur III

    //
    // Generieren Sie eine 3D-Textur, die Marmor oder eine Steinstruktur simuliert
    //
    // Benutzen Sie die Perlin-Noise-Funktion: Noise::noise3(point) (!)
    // ...
    //
    // Lassen Sie Ihrer Kreativitaet freien Lauf!
    //

    cout << endl
         <<"3x 3D Textures ("<<w<<"x"<<h<<"x"<<d<<" Texels, "<<w*h*d*3*3/1024<<" KBytes) "
         << endl;

    for (int x=0; x<w; x++)
```

```
{
    for (int y=0; y<h; y++)
        for (int z=0; z<d; z++)
        {
            // texture offset
            const int baseoffset = 3 * (x*h*d + y*d +z);

            // little noise
            float point1[3];
            point1[0] = (4.0*x)/w;
            point1[1] = (4.0*y)/h;
            point1[2] = (4.0*z)/d;

            // little noise (differs from first one)
            float point2[3];
            point2[1] = (4.0*x)/w;
            point2[2] = (4.0*y)/h;
            point2[0] = (4.0*z)/d;

            // really noisy
            float point3[3];
            point3[1] = (16.0*x)/w;
            point3[2] = (16.0*y)/h;
            point3[0] = (16.0*z)/d;

            float persistence1 = 0.25;
            float persistence2 = 0.25;
            float noise1 = 0.0;
            float noise2 = 0.0;

            // 4 octaves
            for (int octave = 0; octave<4; octave++)
            {
                // brown marble's noise
                // shift point
                point1[0] += 0.4*cos(noise1);
                point1[1] += 0.4*sin(noise1+PI/2);
                // get noise
                float currentnoise1 = (Noise::noise3(point1)+0.5) * persistence1;
                noise1 += currentnoise1;
                // add a bit of noisy noise (?!)
                noise1 += 0.2 * Noise::noise3(point3) * persistence1;

                // dark marble's noise
                point2[0] += 0.4*cos(noise2);
                point2[1] += 0.4*sin(noise2+PI/2);
                // get noise
                float currentnoise2 = (Noise::noise3(point2)+0.5) * persistence2;
                noise2 += currentnoise2;
                // add a bit of noisy noise (?!)
                noise2 += 0.2 * Noise::noise3(point3) * persistence2;
            }

            // clamp noises
            if (noise1 < 0) noise1 = 0;
            if (noise1 > 1) noise1 = 1;
            if (noise2 < 0) noise2 = 0;
            if (noise2 > 1) noise2 = 1;

            // color
            Vector color(0,0,0);

            // brown marble
            color = mix(bright, brown,  smoothstep(0.55, 0.60, noise1));
            color = mix(color,  bright, smoothstep(0.65, 0.68, noise1));

            // texture III: brown marble
            img3[baseoffset+0] = color[0]*255;
            img3[baseoffset+1] = color[1]*255;
            img3[baseoffset+2] = color[2]*255;

            // dark marble
            if (noise2 <= 0.55)
                color = mix(color, dark,  smoothstep(0.50, 0.52, noise2));
            if (noise2 >= 0.52)
                color = mix(dark,  color, smoothstep(0.52, 0.55, noise2));
```

```
                // texture I: combined
                img1[baseoffset+0] = color[0]*255;
                img1[baseoffset+1] = color[1]*255;
                img1[baseoffset+2] = color[2]*255;

                // dark marble (for texture II)
                color = mix(bright, dark,   smoothstep(0.50, 0.52, noise2));
                color = mix(color,  bright, smoothstep(0.52, 0.55, noise2));

                // texture II: dark marble
                img2[baseoffset+0] = color[0]*255;
                img2[baseoffset+1] = color[1]*255;
                img2[baseoffset+2] = color[2]*255;
            }

        // show progress (1 point = ca. 10%)
        if (x % 7 == 0)
        {
            cout << ".";
            cout.flush();
        }
    }

    // Textur parameter
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glGenTextures(1, &texName1_);
    glGenTextures(2, &texName2_);
    glGenTextures(3, &texName3_);

    // texture I
    glBindTexture(GL_TEXTURE_3D, texName1_);

    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S,     GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T,     GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R,     GL_REPEAT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, w, h, d, 0, GL_RGB, GL_UNSIGNED_BYTE, img1);

    // texture II
    glBindTexture(GL_TEXTURE_3D, texName2_);

    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S,     GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T,     GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R,     GL_REPEAT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, w, h, d, 0, GL_RGB, GL_UNSIGNED_BYTE, img2);

    // texture III
    glBindTexture(GL_TEXTURE_3D, texName3_);

    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S,     GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T,     GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R,     GL_REPEAT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, w, h, d, 0, GL_RGB, GL_UNSIGNED_BYTE, img3);

    // activate texture I
    glBindTexture(GL_TEXTURE_3D, texName1_);

    delete[] img1;
    delete[] img2;
    delete[] img3;
}

void CGTexture::onInit() {
```

```
    buildAvailableExtensions();
    torus_ = glGenLists(1);
    buildPatch(32);

    // Prozedurale 3D Textur
    createNoiseTexture3D();


    /* init light */
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 25.0 };
    GLfloat gray[] = { 0.6, 0.6, 0.6, 0.0 };
    GLfloat white[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat light_position[] = { 0.0, 3.0, 3.0, 0.0 };

    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_AMBIENT, gray);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
    glLightfv(GL_LIGHT0, GL_SPECULAR, white);
    glColorMaterial(GL_FRONT, GL_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    // Tiefen Test aktivieren
    glEnable(GL_DEPTH_TEST);

    // Smooth Schattierung aktivieren
    glShadeModel(GL_SMOOTH);

    // Projection
    glMatrixMode(GL_PROJECTION);
    gluPerspective(60.0, 1.0, 2.0, 50.0);

    // LookAt
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(
        0.0, 0.0, 4.0,  // from (0,0,4)
        0.0, 0.0, 0.0,  // to (0,0,0)
        0.0, 1.0, 0.);  // up

    glClearColor(1, 1, 1, 1);
}

void CGTexture::onSize(unsigned int newWidth,unsigned int newHeight) {
    width_ = newWidth;
    height_ = newHeight;
    glMatrixMode(GL_PROJECTION);
    glViewport(0, 0, width_ - 1, height_ - 1);
    glLoadIdentity();
    gluPerspective(40.0,float(width_)/float(height_),2.0, 100.0);
    glMatrixMode(GL_MODELVIEW);
}

void CGTexture::onKey(unsigned char key) {
    switch (key) {
    case 27: { exit(0); break; }
    case '+': { zoom_*= 1.1; break; }
    case '-': { zoom_*= 0.9; break; }
    case ' ': { run_ = !run_; break; }
    case '1': { buildPatch(4); break; }
    case '2': { buildPatch(8); break; }
    case '3': { buildPatch(16); break; }
    case '4': { buildPatch(32); break; }
    case '5': { buildPatch(64); break; }
    case '6': { buildPatch(128); break; }
    case 't': { usetorus_ = !usetorus_; break; }
    case 'q': { glBindTexture(GL_TEXTURE_3D, texName1_); break; }
    case 'w': { glBindTexture(GL_TEXTURE_3D, texName2_); break; }
    case 'e': { glBindTexture(GL_TEXTURE_3D, texName3_); break; }
    }

    onDraw();
}
```

```
void CGTexture::onIdle() {
    if (run_) {
        glRotatef(.5, 0.0, 1.0, 0.0);
    }
    onDraw();
}

void CGTexture::onDraw() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawScene();
    swapBuffers();
}

// Hauptprogramm
int main(int argc, char* argv[]) {
    // Erzeuge eine Instanz der Beispiel-Anwendung:
    CGTexture sample;

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "+          Hineinzoomen" << endl
         << "-          Herauszoomen" << endl
         << "t          Torus oder Quadrat zeichnen" << endl
         << "q          gemischter Marmor" << endl
         << "w          schwarzer Marmor" << endl
         << "e          brauner Marmor" << endl
         << "1-6        Tessellationsgrad des Torus (max=6)" << endl;

    // Starte die Beispiel-Anwendung:
    sample.start("Stephan Brumme, 702544", true, 256, 256);
    return(0);
}
```