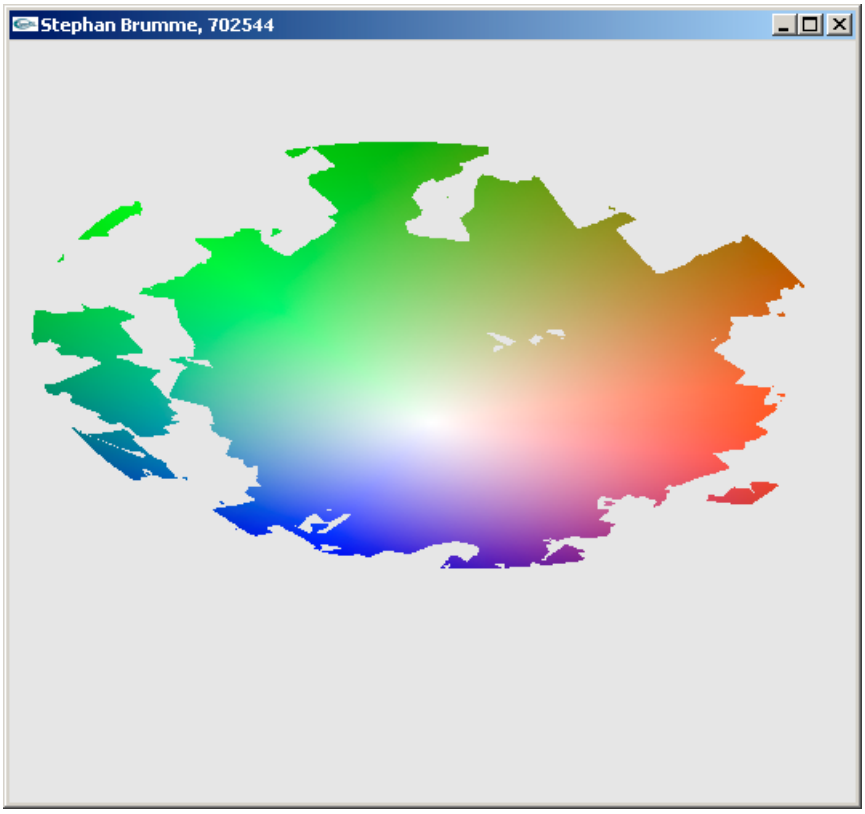
Aufgabe 21: Alpha-Texturen

Die Bedienung des Programms wird über diese Tasten realisiert:

Taste	Aktion
ESC	Programm beenden
Leertaste	Patch rotieren an/aus
+	in die Szene hineinzoomen
-	aus der Szene herauszoomen
a	Alpha-Test ein/aus
x	Alpha-Schwellwert erhöhen (+0,05)
y	Alpha-Schwellwert senken (-0,05)



Der Screenshot entsteht bei der Verwendung eines Schwellwertes von 0,15 und entspricht meines Erachtens genau der Vorlage aus der Aufgabenstellung.

Das Einbinden der Textur erfordert keine besonderen Einstellungen, es ist lediglich darauf zu achten, dass diesmal kein RGB-Bild vorliegt, stattdessen werden die Graustufen als Alpha-Werte interpretiert:

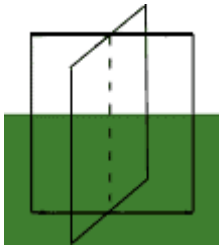
```
// bind image to texture object
glTexImage2D(GL_TEXTURE_2D, 0, GL_ALPHA, texWidth_, texHeight_, 0, GL_ALPHA,
             GL_UNSIGNED_BYTE, image_);
```

Abhängig vom Schwellwert `threshold_` setze ich die Alpha-Funktion bei jedem Bildaufbau:

```
// set alpha function
if (alpha_)
{
    glAlphaFunc(GL_GEQUAL, threshold_);
    glEnable(GL_ALPHA_TEST);
}
else
    glDisable(GL_ALPHA_TEST);
```

Die Texturkoordinaten entsprechen der Einfachheit halber den Dreieckskoordinaten des Objekts, ich muss aber darauf achten, dass der Wertebereich ein anderes ist und demzufolge eine kleine Skalierung und Verschiebung notwendig ist, damit $s,t \in [0,1]$ gilt:

```
// set texture coordinates (same as vertex)
glTexCoord2f((cos(rad)+1)/2, (sin(rad)+1)/2);
glVertex3f(cos(rad), sin(rad), 0);
```



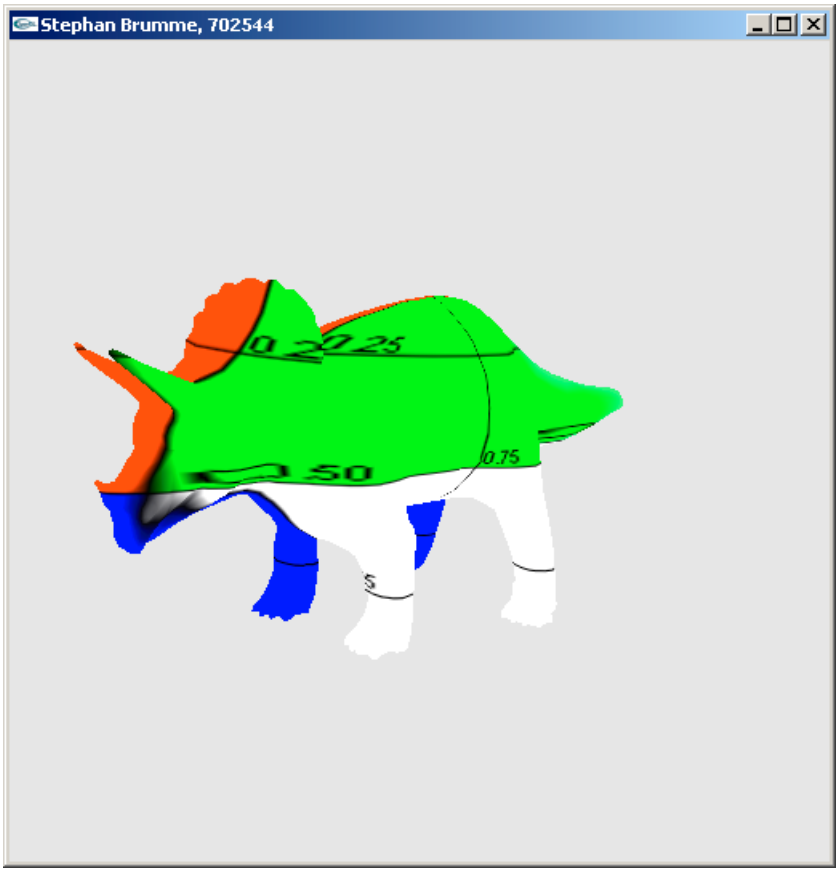
Sehr häufig werden Alpha-Texturen verwendet, um die geometrische Komplexität eines Baumes durch einfaches Texture-Mapping zu simulieren. Dabei nutzt man zwei oder mehr ineinander verdrehte Texturen, die Fotos aus den jeweiligen Blickwinkeln enthalten.

Gerade in dichten Wäldern fällt dieser Trick kaum auf, da man nicht die Möglichkeit hat, sich jeden einzelnen Baum genauer anzusehen. Wichtig ist, dass man nicht immer die gleichen Texturen verwendet, da identische Bäume unrealistisch sind:

Aufgabe 22: Two-Part Mapping

Mit diesen Tasten wird das Programm gesteuert:

Taste	Aktion
ESC	Programm beenden
Leertaste	Patch rotieren an/aus
+	in die Szene hineinzoomen
-	aus der Szene herauszoomen
q	Kugel-Intersektions-Mapping
w	Kugel-Normalen-Mapping
e	Zylinder-Intersektions-Mapping
r	Zylinder-Normalen-Mapping
l	Drahtgittermodell
f	Texture Mapping ein
c	Backface Culling ein/aus



Gemäß der Bezeichnung *Two-Part-Mapping* erfolgt die Abbildung eines Oberflächenpunktes auf eine Textur-Koordinate in zwei Schritten: zuerst projiziere ich den Oberflächenpunkt auf das Zwischenobjekt, anschließend wird von diesem in die entsprechenden Texturkoordinaten umgerechnet.

Die Kugel-Intersektion bestimme ich mit Hilfe des Codes aus dem Übungsblatt 6 (Raytracer), den ich lediglich dahingehend abgeändert habe, dass das Zwischenobjekt eine Einheitskugel ist (Mittelpunkt im Ursprung, Radius ist 1) und sich daraus ein paar Vereinfacherungen aus Multiplikationen mit Null bzw. Eins ergeben.

Eine Abbildung auf die Kugel-Normalen macht sich zu Nutze, dass die Gerade vom Mittelpunkt der Kugel durch den zu projizierenden Vektor als Richtungsvektor genau die gesuchte Normale hat. Als Resultat bleibt eine Zeile Code übrig:

```
return SphereSurfaceToTexture(point.normalized());
```

Im Aufgabenblatt 7 musste bereits die Abbildung einer Kugel auf Texturkoordinaten *s* und *t* hergeleitet werden. Was dort auf rein mathematischer Ebene geschah, habe ich für diese Aufgabe in echten Code umgesetzt. Eine kleine Erweiterung besteht darin, dass ich die Textur exakt in der gleichen Art und Weise wie im Beispielfoto der Aufgabenstellung auf den Dino drauflegen will, wozu eine Rotation um 180° notwendig ist.

$$Textur_{Kugel}\begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = \begin{pmatrix} s \\ t \end{pmatrix}$$

$$s = \begin{cases} \arccos\left(\dfrac{x_p}{\sin(\arccos y_p)}\right)\cdot\dfrac{1}{2\pi} & wenn\ z_p > 0 \\ 1-\arccos\left(\dfrac{x_p}{\sin(\arccos y_p)}\right)\cdot\dfrac{1}{2\pi} & sonst \end{cases}$$

$$t = \frac{\arccos y_p}{\pi}$$

```
Vector CGTwoPart::SphereSurfaceToTexture(const Vector& surface) const
{
    const double x_distortion = sin(acos(surface[1]));

    // first texture coordinate
    double s = 1.0;
    if (x_distortion != 0.0)
    {
        s = acos(surface[0] / x_distortion) / (2*PI);
        if (surface[2] > 0.0)
            s = 1-s;
    }

    // rotate by 180°
    s += 0.5;
    if (s > 1)
        s -= 1;

    // second texture coordinate
    double t = acos(surface[1]) / PI;

    return Vector(s,t);
}
```

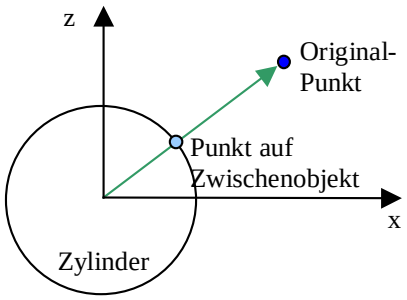
Als Zwischenobjekt ist ein Zylinder nicht ganz so einfach wie die Kugel. Die Intersektion konnte ich aus dem Aufgabenblatt 6 entnehmen, ich musste nur darauf achten, dass diesmal die Rotationsachse des Zylinders die y- statt der z-Achse ist. Ansonsten bleibt die Formel erhalten:

$$f(x,y,z) = x^2 + y^2 - r^2 = 0$$
$$i = x_2 - x_1$$
$$j = z_2 - z_1$$
$$0 = (x_1 + it)^2 + (y_1 + jt)^2 - r^2$$
$$0 = t^2 + t \cdot p + q$$
$$t_{1,2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$$
$$p = \frac{2 \cdot (ix_1 + jz_1)}{i^2 + j^2}$$
$$q = \frac{x_1^2 + z_1^2 - r^2}{i^2 + j^2}$$
$$t_{1,2} = -\frac{ix_1 + jy_1}{i^2 + j^2} \pm \sqrt{\left(\frac{ix_1 + jy_1}{i^2 + j^2}\right)^2 - \frac{x_1^2 + y_1^2 - r^2}{i^2 + j^2}}$$

Der Code setzt dies geradlinig um, er wählt die am nähesten liegende Intersektion aus:

```
Vector CGTwoPart::CylinderIntersection(const Vector& point) const
{
    // radius
    const double r = 0.5;

    // origin of line
    const double x = center_[0];
    const double z = center_[2];

    // direction
    const double i = point[0]   - x;
    const double j = point[2]   - z;

    // no degenerated lines
    if (i==0 && j==0)
        return Vector(0,0,0);

    // 0 = t²+pt+q
    const double p = 2*(i*x+j*z) / (i*i+j*j);
    const double q = (x*x+z*z-r*r) / (i*i+j*j);

    // square root
    const double D = (p*p/4)-q;
    // no solution
    if (D < 0)
        return Vector(0,0,0);

    // t1 may be equal to t2 but that doesn‘t matter
    const double t1 = -p/2 + sqrt(D);
    const double t2 = -p/2 - sqrt(D);

    // nearest intersetion
    double intersect = min(t1,t2);
    if (intersect < 0)
        intersect = max(t1,t2);

    // move point to the cylinder‘s surface and get texture coordinates
    return CylinderSurfaceToTexture(center_ + intersect*(point-center_));
}
```

Für die Zylinder-Normalen-Abbildung nutze ich mir bei der Projektion eigentlich nur die x- und die z-Koordinate. Schaut man entlang der y-Achse, dann entsteht folgendes Bild:



Der Abstand des Originalpunktes von der y-Achse ist relativ einfach zu berechnen:

$$d = \sqrt{x^2 + z^2}$$

Die Projektion auf das Zwischenobjekt verlangt, dass ein neues *d'* genau dem Radius *r* entspricht:

$$\begin{aligned}
d' &= r \\
x' &= \frac{x \cdot r}{d} \\
y' &= y \\
z' &= \frac{z \cdot r}{d} \\
\sqrt{x'^2 + z'^2} &= \sqrt{\left(\frac{x \cdot r}{d}\right)^2 + \left(\frac{z \cdot r}{d}\right)^2} \\
&= \sqrt{\frac{r^2}{d^2} \cdot \left(x^2 + z^2\right)} \\
&= \frac{r}{d} \cdot \sqrt{x^2 + z^2} \\
&= d'
\end{aligned}$$

Die y-Koordinate bleibt unverändert. In C++ sind die langen Formeln wesentlich kürzer und eleganter:

```
Vector CGTwoPart::CylinderNormal(const Vector& point) const
{
    // distance to y axis
    const double distance_to_y_axis = sqrt(point[0]*point[0] + point[2]*point[2]);
    const double radius = 0.5;

    // move point to the cylinder's surface
    const Vector surface(point[0] / (distance_to_y_axis/radius),
                         point[1],
                         point[2] / (distance_to_y_axis/radius));

    // get texture s,t
    return CylinderSurfaceToTexture(surface);
}
```

Zur Abbildung eines Punktes von der Zylinderoberfläche auf die Textur ziehe ich die Höhe, also *y*, und den Drehwinkel um die y-Achse heran. Das ganze ähnelt sehr der Kugel:

$$Textur_{Zylinder}\begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = \begin{pmatrix} s \\ t \end{pmatrix}$$

$$s = \begin{cases} \arccos\left(\dfrac{x_p}{\sin(\arccos y_p)}\right)\cdot\dfrac{1}{2\pi} & wenn\ z_p > 0 \\ 1-\arccos\left(\dfrac{x_p}{\sin(\arccos y_p)}\right)\cdot\dfrac{1}{2\pi} & sonst \end{cases}$$

$$t = y_p + \frac{1}{2}$$

Bezüglich *t* möchte ich einige Anmerkungen machen:
Die Textur erfordert Koordinaten mit einem Wertebereich von jeweils [0,1]. Entsprechend sind Skalierungen und Verschiebungen vorzunehmen, im Beispiel heißt das, dass 0,5 auf den y-Wert addiert werden müssen. Trotzdem kann ich den Fall nicht ausschließen, der *t* außerhalb von [0,1] generiert, was sich in Schnittpunkten in der Zylinderdeckel- bzw. bodenfläche äußert. Diese schneide ich einfach ab, d.h. negative Werte werden zu Null, Werte großer Eins werden zu genau Eins.
Um der Vorlage nahe zu kommen, rotiere ich *s* um 180°.

```
Vector CGTwoPart::CylinderSurfaceToTexture(const Vector& surface) const
{
    const double radius = 0.5;
    // determine rotation angle
    double s = acos(surface[0]/radius) / (2*PI);
    if (surface[2] > 0.0)
        s = 1-s;

    // rotate again by 180°
    s += 0.5;
    if (s > 1)
        s -= 1;

    // t comes from da height
    double t = 1 - (surface[1] + 0.5);
    if (t < 0)
        t = 0;
    if (t > 1)
        t = 1;

    return Vector(s,t);
}
```

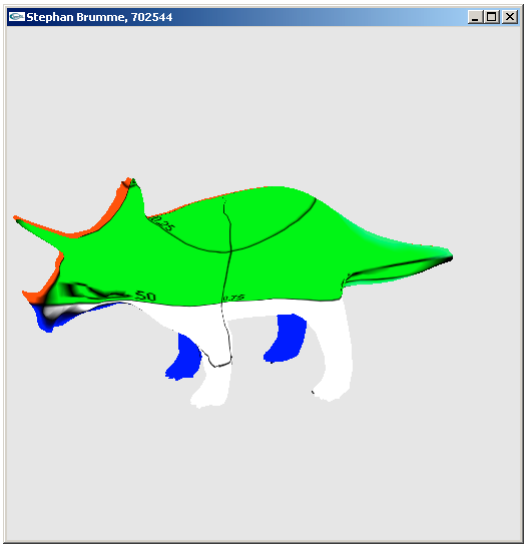
Bisher habe ich nicht erwähnt, wo denn diese ganzen Funktionen benutzt werden. Für diese Aufgabe ist `handleVertex` zuständig, indem es das aktuell verwendete Verfahren aus `mapping_` ausliest und das passende Mapping-Verfahren aufruft. Die Rückgabewerte dienen als Texturkoordinaten für OpenGL:

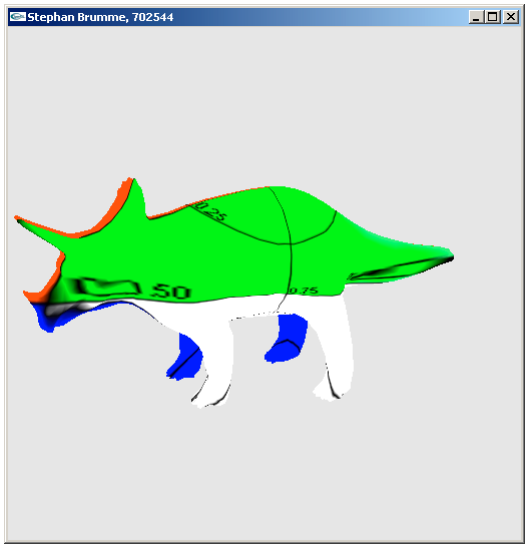
```
void CGTwoPart::handleVertex(const Vector& v) const{
    // texturing ...
    Vector texCoord;

    switch (mapping_)
    {
    case SPHERE_INTERSECTION:   texCoord = SphereIntersection(v);
                                break;
    case SPHERE_NORMAL:         texCoord = SphereNormal(v);
                                break;
    case CYLINDER_INTERSECTION: texCoord = CylinderIntersection(v);
                                break;
    case CYLINDER_NORMAL:       texCoord = CylinderNormal(v);
                                break;
    }
    glTexCoord2d(texCoord[0], texCoord[1]);

    glVertex3dv(v.rep());
}
```

Des öfteren tauchte der Mittelpunkt `center_` des Szenenobjektes auf. Er entsteht als „Schwerpunkt“, dieses berechne ich vereinfacht aus dem gewichteten Mittel aller Oberflächenpunkte:

$$center = \frac{1}{n} \cdot \sum_{i=1}^{n} vertex_i$$

Für den Dinosaurier liegt dieser Punkt bei ca. (0.3, 0, 0). Das ist auch der Grund dafür, dass sich die Intersektionsverfahren von den Normalenverfahren recht deutlich durch einen Shift entlang der x-Achse unterscheiden. Im linken Bild die Kugel-Intersektion (Mittelpunkt wandert nach links, da Rückseite des Dinos), im rechten Bild dagegen die Kugel-Normalen-Projektion:





Quellcode

Für Aufgabe 21 waren nur wenige Zeilen Code in alphatexture.cpp zu schreiben:

alphatexture.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 8
//

#include "cgalphatexture.h"
#include <fstream.h>
#include <stdlib.h>

#ifndef M_PI
const double M_PI = 3.14159265358979323846;
#endif

const int SLICES = 120;

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

    // initial threshold, ca. 50% is visible
    threshold_ = 0.5;
    // alpha blending is turned on
    alpha_     = true;

    run_ = false;
    zoom_ = 1.5;
}


CGAlphaTexture::~CGAlphaTexture() {
}

static GLfloat light_position1[] = {0.0, 0.0, 0.0, 1.0};  /* Point light location. */

void CGAlphaTexture::drawScene() {

    // set alpha function
    if (alpha_)
    {
        glAlphaFunc(GL_GEQUAL, threshold_);
        glEnable(GL_ALPHA_TEST);
    }
    else
        glDisable(GL_ALPHA_TEST);


    glPushMatrix();
    glRotatef(60,1,0,0);
    glScaled(zoom_, zoom_, zoom_);

    glBegin(GL_TRIANGLE_FAN);
        glColor3f(1.0, 1.0, 1.0);
        glTexCoord2f(0.5, 0.5);
        glVertex3f(0.0, 0.0, 0.0);
        for (int i=0; i<=SLICES; i++) {

            double col = 3*i/(double)SLICES;
            double colfloor = col - (int)col;
            if (col>=3.0) glColor3f(1.0, 0.0, 0.0);
            else if (col>=2.0) glColor3f(colfloor, 0, 1.0-colfloor);
            else if (col>=1.0) glColor3f(0, 1.0-colfloor, colfloor);
            else glColor3f(1.0-colfloor, colfloor, 0);

            double rad = i/(double)SLICES * M_PI * 2.0;
```

```
            // set texture coordinates (same as vertex)
            glTexCoord2f((cos(rad)+1)/2, (sin(rad)+1)/2);
            glVertex3f(cos(rad), sin(rad), 0);
        }
    glEnd();
    glPopMatrix();
}

void CGAlphaTexture::readImage() {
    // reads grayscale PPM-image

    const int BUF_SIZE = 1024;

    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 "P5"
    char ppm;
    in >> ppm; if(!in.good() || ppm != 'P') { cerr << "ppm format error" << endl; }
    in >> ppm; if(!in.good() || ppm != '5') { 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 [texWidth_ * texHeight_];
    in.read(image_, texWidth_ * texHeight_);
    if(!in.good()) { cerr << "ppm format error" << endl; }

    in.close();
}

void CGAlphaTexture::onInit() {

    // 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(0.9,0.9,0.9,1.0);

    readImage();

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

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

    // 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_ALPHA, texWidth_, texHeight_, 0, GL_ALPHA,
GL_UNSIGNED_BYTE, image_);

    // enable texture mapping
    glEnable(GL_TEXTURE_2D);
}

void CGAlphaTexture::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 CGAlphaTexture::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; }
    // increase threshold
    case 'x': { threshold_ += 0.025;
                if (threshold_ > 1)
                    threshold_ = 1;
                cout << "Schwellwert: " << threshold_ << endl;
                break; }
    // decrease threshold
    case 'y': { threshold_ -= 0.025;
                if (threshold_ < 0)
                    threshold_ = 0;
                cout << "Schwellwert: " << threshold_ << endl;
                break; }
    // enable/disable alpha blending at all
    case 'a': { alpha_ = !alpha_; break; }
    }

    onDraw();
}

void CGAlphaTexture::onIdle() {

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

void CGAlphaTexture::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:
    CGAlphaTexture sample("alpha.ppm");

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "+          Hineinzoomen" << endl
         << "-          Herauszoomen" << endl
         << "a          Alpha Blending ein/aus" << endl
         << "x          Alpha-Schwellwert erhoehen" << endl
         << "y          Alpha-Schwellwert senken" << endl << endl;

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

Alle Two-Part-Mapping-Verfahren sind Bestandteil von cgtwopart.cpp:

cgtwopart.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 8
//

#include "cgtwopart.h"
#include <fstream.h>

typedef Vector Camera;
typedef Vector Color;

#ifndef PI
const double PI = 3.14159265358979323846;
#endif

const unsigned int BUF_SIZE = 1024;

//
//  Application
//

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

    stop_=true;
    zoom_= 1.5;
    mapping_ = SPHERE_INTERSECTION;

    // read triangle data
    ifstream s("triceratops.txt");
    // ifstream s("triangle_small.txt");
    s >> size_;

    tris_ = new Triangle[size_];
    center_ = Vector(0,0,0);

    int i;
    for (i = 0; i < size_; i++) {
        s >> tris_[i];

        // just scale object
        tris_[i].setVertex(0, tris_[i].getVertex(0) * 0.1);
        tris_[i].setVertex(1, tris_[i].getVertex(1) * 0.1);
        tris_[i].setVertex(2, tris_[i].getVertex(2) * 0.1);

        // add up all vectors
        center_ += tris_[i].getVertex(0);
        center_ += tris_[i].getVertex(1);
        center_ += tris_[i].getVertex(2);
    }

    center_ *= 1/(3.0*size_);
//  center_ = Vector(0,0,0);
}


CGTwoPart::~CGTwoPart() {
}



void CGTwoPart::onInit() {

    // 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(0.9,0.9,0.9,1.0);

    // Import texture from file
    readImage();

    glGenTextures(1, &texName_);
    glBindTexture(GL_TEXTURE_2D, texName_);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texWidth_, texHeight_, 0, GL_RGB,
GL_UNSIGNED_BYTE, image_);
}

void CGTwoPart::onSize(unsigned int newWidth,unsigned int newHeight) {
    if((newWidth > 0) && (newHeight > 0)) {
        // Passe den OpenGL-Viewport an die neue Fenstergroesse an:
        glViewport(0, 0, newWidth - 1, newHeight - 1);

        // Passe die OpenGL-Projektionsmatrix an die neue
        // Fenstergroesse an:
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(40.0,float(newWidth)/float(newHeight),1.0, 10.0);

        // Schalte zurueck auf die Modelview-Matrix
        glMatrixMode(GL_MODELVIEW);
    }
}

void CGTwoPart::onKey(unsigned char key) {
    static GLfloat z = 3.;
    switch (key) {
    case 27: { exit(0); break; }
    case '+': { zoom_*= 1.1; break; }
    case '-': { zoom_*= 0.9; break; }
    case 'l' : { glPolygonMode(GL_FRONT_AND_BACK,GL_LINE); break; }
    case 'f' : { glPolygonMode(GL_FRONT_AND_BACK,GL_FILL); break; }
    case 'c' : { if (glIsEnabled(GL_CULL_FACE)) { glDisable(GL_CULL_FACE); } else {
glEnable(GL_CULL_FACE); } break; }
    case 'q' : { mapping_ = SPHERE_INTERSECTION; break; }
    case 'w' : { mapping_ = SPHERE_NORMAL; break; }
    case 'e' : { mapping_ = CYLINDER_INTERSECTION; break; }
    case 'r' : { mapping_ = CYLINDER_NORMAL; break; }
    case ' ' : { stop_ = !stop_; break; }
    }
    onDraw();
}

void CGTwoPart::onIdle() {
    if (!stop_) {
        glRotatef(-2.0f,0,1,0.1);
        onDraw();
    }
}

inline double max(double x, double y) {
    return (x<y) ? y : x;
}
inline double min(double x, double y) {
    return (x>y) ? y : x;
}
```

```
Vector CGTwoPart::SphereIntersection(const Vector& point) const
{
    // code taken from ray-sphere intersection of my raytracer

    // ray origin
    const Vector origin(center_);
    const double x = origin[0];
    const double y = origin[1];
    const double z = origin[2];

    // ray direction
    const Vector direction(point-center_);
    const double i = direction[0];
    const double j = direction[1];
    const double k = direction[2];

    // sphere's center
    const double r = 1.0;

    // compute parameters a,b,c for at^2+bt+c=0
    const double a = i*i + j*j + k*k;
    const double b = 2*(i*x + j*y + k*z);
    const double c = x*x+y*y+z*z - r*r;

    // solve at^2+bt+c=0
    // D = b^2-4ac
    const double D = b*b-4*a*c;


    // no intersection
    if (D < 0)
        return Vector(0,0,0);

    // nearest ray intersection
    double t;

    if (D > 0)
    {
        // two intersections
        // ray parameter
        const double t1 = (-b + sqrt(D))/(2*a);
        const double t2 = (-b - sqrt(D))/(2*a);
        t = min(t1,t2);
        if (t < 0)
            t = max(t1,t2);
    }
    else
        // only one intersection
        t = -b/(2*a);

    return SphereSurfaceToTexture(origin + t*direction);
}

Vector CGTwoPart::SphereNormal(const Vector& point) const
{
    return SphereSurfaceToTexture(point.normalized());
}

Vector CGTwoPart::SphereSurfaceToTexture(const Vector& surface) const
{
    const double x_distortion = sin(acos(surface[1]));

    // first texture coordinate
    double s = 1.0;
    if (x_distortion != 0.0)
    {
        s = acos(surface[0] / x_distortion) / (2*PI);
        if (surface[2] > 0.0)
            s = 1-s;
    }

    // rotate by 180°
    s += 0.5;
    if (s > 1)
        s -= 1;
```

```
    // second texture coordinate
    double t = acos(surface[1]) / PI;

    return Vector(s,t);
}

Vector CGTwoPart::CylinderIntersection(const Vector& point) const
{
    // radius
    const double r = 0.5;

    // origin of line
    const double x = center_[0];
    const double z = center_[2];

    // direction
    const double i = point[0]   - x;
    const double j = point[2]   - z;

    // no degenerated lines
    if (i==0 && j==0)
        return Vector(0,0,0);

    // 0 = t²+pt+q
    const double p = 2*(i*x+j*z) / (i*i+j*j);
    const double q = (x*x+z*z-r*r) / (i*i+j*j);

    // square root
    const double D = (p*p/4)-q;
    // no solution
    if (D < 0)
        return Vector(0,0,0);

    // t1 may be equal to t2 but that doesn't matter
    const double t1 = -p/2 + sqrt(D);
    const double t2 = -p/2 - sqrt(D);

    // nearest intersetion
    double intersect = min(t1,t2);
    if (intersect < 0)
        intersect = max(t1,t2);

    // move point to the cylinder's surface and get texture coordinates
    return CylinderSurfaceToTexture(center_ + intersect*(point-center_));
}

Vector CGTwoPart::CylinderNormal(const Vector& point) const
{
    // distance to y axis
    const double distance_to_y_axis = sqrt(point[0]*point[0] + point[2]*point[2]);
    const double radius = 0.5;

    // move point to the cylinder's surface
    const Vector surface(point[0] / (distance_to_y_axis/radius),
                         point[1],
                         point[2] / (distance_to_y_axis/radius));

    // get texture s,t
    return CylinderSurfaceToTexture(surface);
}

Vector CGTwoPart::CylinderSurfaceToTexture(const Vector& surface) const
{
    const double radius = 0.5;
    // determine rotation angle
    double s = acos(surface[0]/radius) / (2*PI);
    if (surface[2] > 0.0)
        s = 1-s;

    // rotate again by 180°
    s += 0.5;
    if (s > 1)
        s -= 1;

    // t comes from da height
```

```
    double t = 1 - (surface[1] + 0.5);
    if (t < 0)
        t = 0;
    if (t > 1)
        t = 1;

    return Vector(s,t);
}

void CGTwoPart::handleVertex(const Vector& v) const{
    // texturing ...
    Vector texCoord;

    switch (mapping_)
    {
    case SPHERE_INTERSECTION:   texCoord = SphereIntersection(v);
                                break;
    case SPHERE_NORMAL:         texCoord = SphereNormal(v);
                                break;
    case CYLINDER_INTERSECTION: texCoord = CylinderIntersection(v);
                                break;
    case CYLINDER_NORMAL:       texCoord = CylinderNormal(v);
                                break;
    }
    glTexCoord2d(texCoord[0], texCoord[1]);

    glVertex3dv(v.rep());
}

void CGTwoPart::onDraw() {
    // Loesche den Farb- und Tiefenspeicher
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


    glPushMatrix();
    // glRotatef(-30,0,1,0);
    glScaled(zoom_, zoom_, zoom_);

    glEnable(GL_TEXTURE_2D);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glBindTexture(GL_TEXTURE_2D, texName_);

    glBegin(GL_TRIANGLES);
    for (int i=0; i<size_; i++) {
        for (int k = 0; k < 3; k++) {
            handleVertex(tris_[i].getVertex(k));
        }
    }
    glEnd();
    glDisable(GL_TEXTURE_2D);

    glPopMatrix();

    // Nicht vergessen! Front- und Back-Buffer tauschen:
    swapBuffers();
}

void CGTwoPart::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; }

    // for creating an image with an alpha channel - but we don't need this?
/*
    unsigned char rgba[4];
    rgba[3] = 255; // alpha value

    for(int i=0; i<texWidth_; i++)
        for(int j=0; j<texHeight_; j++) {
            in.read(rgba, 3);
            if(!in.good()) { cerr << "ppm format error" << endl; }

            for (int k=0; k<4; k++) {
                image_[k + j*4 + i*(4*texHeight_)] = rgba[k];
            }
        }
*/
    in.close();
}

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

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "+          Hineinzoomen" << endl
         << "-          Herauszoomen" << endl
         << "q          Kugel-Intersektions-Mapping" << endl
         << "w          Kugel-Normalen-Mapping" << endl
         << "e          Zylinder-Intersektions-Mapping" << endl
         << "r          Zylinder-Normalen-Mapping" << endl
         << "l          Drahtgittermodell" << endl
         << "f          Texturieren" << endl
         << "c          Verdeckte Flaechen zeichnen an/aus" << endl;

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