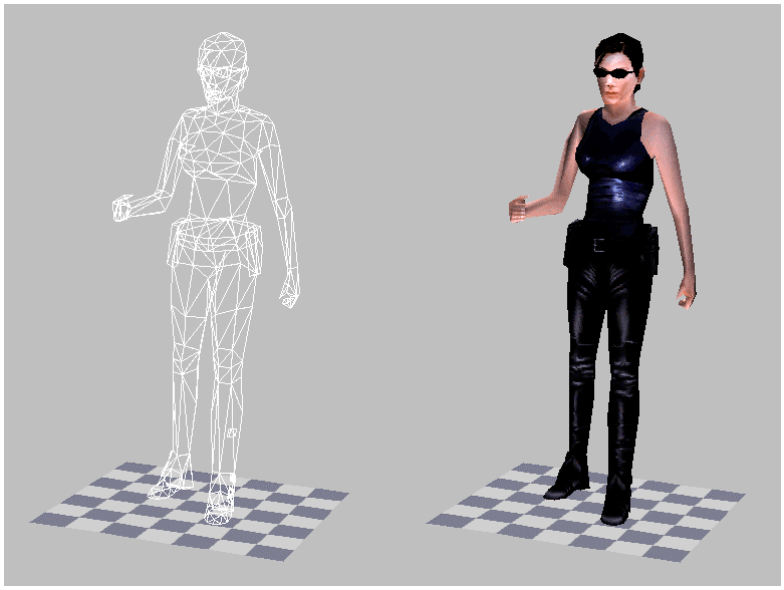
Aufgabe 18: Oberflächengestaltung

Texture Mapping macht sich zunutze, dass die Oberflächen dreidimensionaler Objekte in den zweidimensionalen Raum abgebildet werden können. In diesem sind viele Operationen sehr effizient durchführbar.

Da das menschliche Auge die optisch ansprechende Erscheinung eines Objektes meist an dessen detaillierter Färbung und Beleuchtung misst und nicht an deren exakten äußeren Form, begnügt man sich mit hinreichend gut tessellierten Körper und belegt diese mit (präparierten) Fotos oder berechneten Bildern. Der Rechen- und Modellierungsaufwand für diese ist dabei wesentlich geringer als der für geometrische Formen. Gerade die Wiedergabe profitiert von hardwarebeschleunigten Möglichkeiten zur Texturierung, die quasi ohne zusätzliche Kosten von modernen Grafikkarten durchgeführt werden, während die Anzahl an Eckpunkten eines Objektes die Laufzeit entscheidend bestimmt. Einer Demo zum Film *Matrix* entnahm ich dieses Foto:



Das Drahtgittermodell auf der linken Seite wirkt sehr kantig und unecht. Nur durch Verwendung von Texture Mapping kommt auf der rechten Seite ein ansprechenden Resultat heraus, das deutlich detailreicher wirkt, obwohl die Geometrie unverändert ist. Die Texturen sind relativ einfach austauschbar, man könnte auch eine andere Person mit der gleichen Geometrie, aber veränderter Textur darstellen, ohne dass es sofort auffällt (naja, will man einen Mann mit obiger Geometrie darstellen, so ist es wohl ratsam, die Oberweite irgendwie zu korrigieren, das könnte sonst doch auffallen ...)

Mit Hilfe von Texture Mapping können viele physikalische Effekte mit wenig Aufwand vorgetäuscht werden. So ist etwa die geometrisch korrekte Berechnung der Beleuchtung kompliziert, mit geeigneten Texturen kann ein ziemlich realistisches Ergebnis trotzdem einfach werden. Die dabei auftretenden Ungenauigkeiten werden aufgrund der enormen Zeitersparnis toleriert. Selbst wenn ist notwendig ist, ein Objekt mehrfach zu texturieren (*Multi-Texturing*), so ist dies dennoch effizienter als die entsprechenden geometrischen Berechnungen durchzuführen. Ein Beispiel ist das *Environmental Bump Mapping* des Spiels *Expandable*. Der obere Teil des Screenshot sieht schon recht gut aus, scheint aber sehr flach und konturlos zu sein (was er ja geometrisch auch ist). Eine zusätzliche Texturierung mit einer speziell präparierten Bitmap erweckt dagegen den Eindruck einer realistisch strukturierten Wasseroberfläche mit echter Wellenbewegung. Das dem nicht so ist, fällt in Echtzeit-anwendungen, wie es die meisten Computerspiele sind, kaum auf.

Texture Mapping ist eine Per-Fragment-Operation. Die erzielte Geschwindigkeit hängt also von der Anzahl der zu zeichnenden Pixel ab. Durch Auswahl einer geeigneten Bildschirmauflösung lässt sich die Geschwindigkeit der Anwendung zur Laufzeit einfach an das verwendete System anpassen, eine Quelltextänderung oder Objekt-neumodellierung ist nicht notwendig.

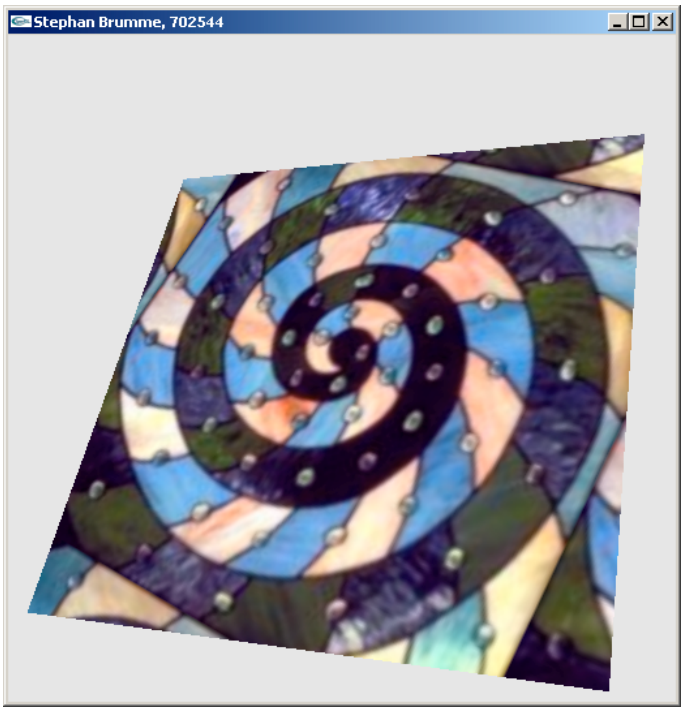
Die Qualität der Texturierung ist einstellbar, über Mip-Mapping und Filterung (Point-Sampling, bilinear, trilinear, anisotrop ...) wird eine Optimierung der Bildqualität in Abhängigkeit von der Entfernung des Betrachters im virtuellen Raum vorgenommen. Je nach Qualität der Texturen können dabei beeindruckende Resultate entstehen.

Trotz aller Vorteile von Texture Mapping soll nicht verschwiegen werden, dass Details nur *vorgetäuscht* werden. Genaues Hinsehen kann oft Fehler und Ungenauigkeiten in den Szenenobjekten entlarven. Nachdem heutige Grafikkarten ausgereifte Texturierungseinheiten besitzen, richtet sich das Augenmerk der Hersteller zunehmend auf die Weiterentwicklung der Verarbeitungsgeschwindigkeit von geometrisch aufwändigen Objekten. Fast alle aktuellen Chips beherrschen sogenanntes Transformation&Lighting (T&L) in Hardware. NVidia demonstriert mit dem (letzten ...) Screenshot, was heute bereits auf ganz normalen Mittelklasse-PCs möglich ist.

Aufgabe 19: Texturanimation

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	Textur im Uhrzeigersinn rotieren an/aus



Die Initialisierung des Texture Mappings in OpenGL erfordert, dass eine eindeutige Textur-ID generiert wird. Da im Programm nur eine einzige Textur verwendet wird, wird sie auch gleich in den aktuellen Kontext eingebunden:

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

Eine Forderung der Aufgabenstellung besteht darin, dass die Textur sich notfalls unendlich oft wiederholen soll, wenn der Bereich [0,0] – [1,1] verlassen wird:

```
// repeat texture if neccessary
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Um ein besseres optisches Resultat zu erzielen, wird eine lineare Filterung aktiviert. Diese Einstellung wende ich sowohl auf die Magnifikation als auch auf die Minifikation an, um unabhängig von der momentanen Bildschirmauflösung stets gute Ergebnisse zu liefern:

```
// 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);
```

Die letzte Schritt in der Initialisierung ist die Bindung der eingelesenen Bilddaten an die Textur und der abschließenden Aktivierung des Texture Mappings:

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

    // enable texture mapping
    glEnable(GL_TEXTURE_2D);
```

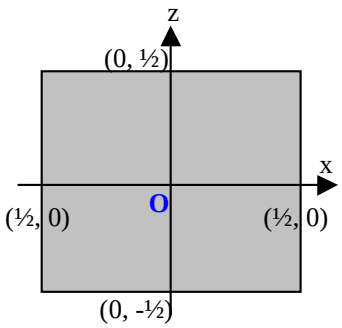
Es ist dabei zu beachten, dass die Rohdaten nur im RGB-Format vorliegen. Aus Geschwindigkeitsgründen werden sie nach RGBA konvertiert (32-Bit-Alignment).

Die eigentliche Texturierung befindet sich in der Funktion `HandleVertex`, die u.a. auch die Eckpunkte der Dreiecke der Grundfläche zeichnet:

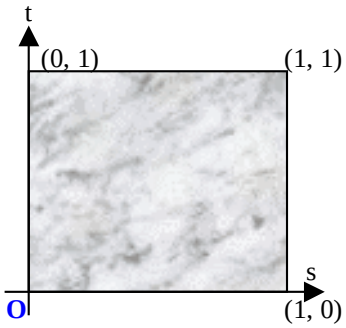
```
void CGTexture::handleVertex(const Vector& v) {
    // Hier müssen die Texturkoordinaten berechnet werden
    glTexCoord2f(v[0]+0.5, v[2]+0.5);

    // draw patch
    glVertex3dv(v.rep());
}
```

Es ist zu beachten, dass sich zwar die Texturkoordinaten aus den Dreieckskoordinaten ableiten, aber dennoch eine Verschiebung notwendig ist, da der Patch im Ursprung zentriert ist, die Textur dagegen nicht. In der untenstehenden Abbildung sieht man den verschobenen Ursprung, der korrigiert werden muss:



Patch



Textur

Das letzte noch offene Problem ist die Rotation der Textur. Eine von mir zuerst verfolgte Strategie bestand darin, in `HandleVertex` die Parameter *s* und *t* über Winkelfunktionen zu verändern. Ich stellte jedoch schnell fest, dass mit dem mir bislang unbekannten Texture Matrix Stack eine wesentliche elegantere Lösung möglich ist. Nimmt man die homogenen Koordinate (*s*, *t*, *r*, *q*), so benötigt man nur eine Verschiebung des Ursprungs, eine Rotation um die *r*-Achse (im Uhrzeigersinn bedeutet um einen negativen Winkel) und eine Rücktranslation des Ursprungs. In OpenGL sind das ganze 5 Zeilen, die alle in `OnIdle` stecken:

```
    // use texture matrix stack
    glMatrixMode(GL_TEXTURE);

    // move origin
    glTranslatef(0.5, 0.5, 0);
    // rotate
    glRotatef(-1, 0, 0, 1);
    // move back
    glTranslatef(-0.5, -0.5, 0);

    // switch back to model view matrix stack
    glMatrixMode(GL_MODELVIEW);
```

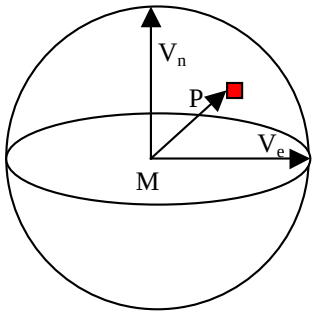
Aufgabe 20: Texturkoordinatenberechnung für geometrische Objekte

Texturierung einer Kugel

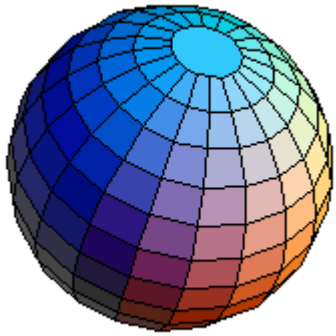
Für das Texture Mapping einer Einheitskugel kann man folgende Annahmen treffen:

$$M = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},\ r = 1,\ V_n = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix},\ V_e = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

wobei M der Kugelmittelpunkt ist und r den Radius darstellt. Die beiden Vektoren V_n und V_e dienen dazu, den „Nordpol“ zu beschreiben (V_n) bzw. den „Äquator“ anzugeben (V_e). Beide sind normiert. In einer Skizze wird dies deutlicher:



Die Textur soll sich dergestalt um die Kugel wickeln, dass *s* bzw. *t* dem Höhen- bzw. Breitengrad entsprechen. Die oberste Zeile der Textur gehört zum Nordpol, die unterste demzufolge zum Südpol. Die y-Achse verläuft durch Nord- und Südpol.



Für einen zu texturierenden Punkt *P* ergibt sich der Breitengrad (und damit *t*) allein aus dem Winkel, den *P* und V_n einschließen:

$$\cos\varphi = P \bullet V_n$$

$$t = \frac{\varphi}{\pi}$$

$$= \frac{\arccos\left(P \bullet V_n\right)}{\pi}$$

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

Aufgrund der Verwendung von Radiant als Winkeleinheit um den sich ergebenden Wertebereich [0, π] ist φ durch π zu dividieren.

Der Längengrad ist leider etwas komplizierter in der Ermittlung. Als erstes berechne ich den von *P* und V_e eingeschlossenen Winkel:

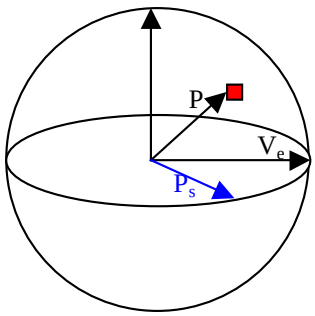
$$\cos\vartheta = P \bullet V_e$$

Da dieser Winkel aber von φ abhängt, muss er um den Sinus korrigiert werden:

$$\cos\vartheta = \frac{P \bullet V_e}{\sin\varphi} = \frac{P}{\sin\varphi} \bullet V_e$$

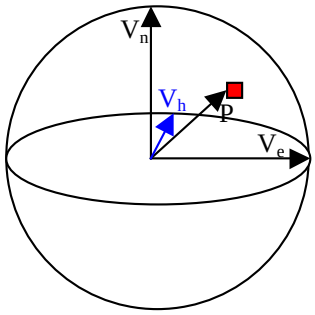
$$= \frac{x_p}{\sin\left(\arccos y_p\right)}$$

In einer Skizze ist P_s die Abbildung von *P* auf die Ebene, die senkrecht zu V_n steht und V_e beinhaltet. P_s geht durch Anwendung des Sinus aus *P* hervor:



Leider ist aus diesem Winkel nicht ersichtlich, in welcher Richtung er bezüglich V_e steht, so dass für die Vorderseite der Kugel die gleichen Werte berechnet werden wie für die Rückseite. Diesem Dilemma entgehe ich, indem ich einen Hilfsvektor V_h einführe:

$$V_h = V_n \times V_e = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$



Nur wenn $V_h \times P > 0$ ist, dann liegt P auf der Rückseite der Kugel. Die endgültige Formel für den Längengrad lautet damit:

$$s = \begin{cases} \dfrac{\vartheta}{2\pi} & wenn\ V_H \bullet P > 0 \\ 1 - \dfrac{\vartheta}{2\pi} & sonst \end{cases}$$

$$\vartheta = \arccos\left(\frac{x_p}{\sin(\arccos\ y_p)}\right)$$

Alles zusammen:

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

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

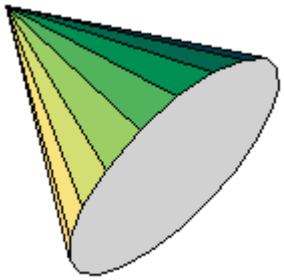
$$t = \frac{\arccos\ y_p}{\pi}$$

Die Umkehrabbildung entsteht durch einfache Umformung der Gleichungen:

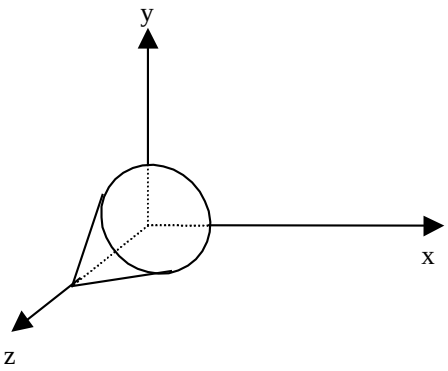
$$Textur^{-1}_{Kugel}(s,t) = \begin{pmatrix} \cos(2\pi \cdot s) \cdot \sin(\pi \cdot t) \\ \cos(\pi \cdot t) \\ \sin(2\pi \cdot s) \cdot \sin(\pi \cdot t) \end{pmatrix}$$

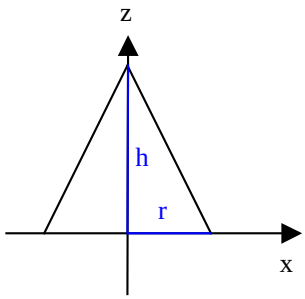
Texturierung eines Kegels

Ich definiere, dass für mich die Grundfläche des Kegels *nicht* Bestandteil des Kegels ist und demzufolge auch *nicht* texturiert wird.



Weiterhin lege ich fest, dass die Grundfläche des Kegels in der xy-Ebene liegt, dies wurde nicht so eindeutig in der Aufgabenstellung gesagt. Die wichtigen bekannten Parameter sind die Höhe und der Radius:



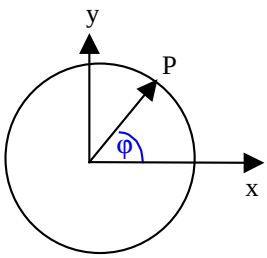


Mein Kegel-Texturierungs-Algorithmus für einen Punkt *P* ordnet *s* einem Drehwinkel von *P* um die *z*-Achse zu, während *t* sich aus dem Abstand des Punktes *P* von der xy-Ebene ergibt.

Für *t* gilt deshalb (1 entspricht der Kegelspitze):

$$t = \frac{z_p}{h}$$

Den Drehwinkel φ veranschauliche ich durch Ansicht des Kegels entlang der *z*-Achse:



Somit gilt unter Beachtung der Normalisierung von x_p auf den Einheitskreis:

$$\varphi = \begin{cases} \arccos \frac{x_p}{l} & falls\ y_p \geq 0 \\ 2\pi - \arccos \frac{x_p}{l} & sonst \end{cases}$$

$$l = \frac{h - z_p}{h} \cdot r$$

Und für die Texturkoordinate s, da φ den Wertebereich $[0, 2\pi]$ hat:

$$s = \frac{\varphi}{2\pi}$$

Zusammengefasst:

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

$$s = \begin{cases} \frac{1}{2\pi} \cdot \arccos \frac{2x_p}{1 - z_p} & falls\ y_p \geq 0 \\ 1 - \frac{1}{2\pi} \cdot \arccos \frac{2x_p}{1 - z_p} & sonst \end{cases}$$

$$t = z_p$$

Die Umkehrabbildung lässt sich aus diesen Gleichungen wieder sofort herleiten:

$$Textur_{Kegel}^{-1}(s,t) = \begin{pmatrix} \frac{1-t}{2} \cdot \cos(2\pi \cdot s) \\ \frac{1-t}{2} \cdot \sin(2\pi \cdot s) \\ t \end{pmatrix}$$

Sollte es unerwünscht sein, dass t=1 der Kegelspitze entspricht, so kann man t‘=1-t benutzen.

Texturierung eines Paraboloids

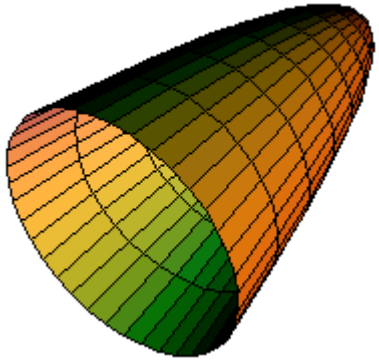
Ein Paraboloid ähnelt insofern einem Kegel, als dass die Grundidee der Texturierung die gleiche ist: *s* entsteht aus einem Drehwinkel von *P* um die *z*-Achse, *t* basiert auf dem Abstand von *P* zur *z*-Achse. Aus den Gleichungen

$$z = x^2 + y^2$$
$$z \le 1$$

kann man sofort entnehmen, dass

$$h = 1$$
$$r = 1$$

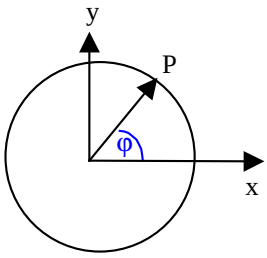
Der einzige Unterschied zum Kegel besteht nun nur noch darin, dass die Radii von der Grundfläche zur Spitze hin nicht mehr linear abnehmen, sondern den Verlauf der Wurzelfunktion nachahmen.



Die Formel für *t* kann ich daher unverändert übernehmen:

$$t = \frac{z_p}{h} = z_p$$

Auch für φ kommt die gleiche Skizze zum Tragen:



Nur *l* ist anders:

$$\varphi = \begin{cases} \arccos\dfrac{x_p}{l} & falls\ y_p \geq 0 \\ 2\pi - \arccos\dfrac{x_p}{l} & sonst \end{cases}$$

$$l = \frac{\sqrt{x_p^2 + y_p^2}}{h} = \frac{\sqrt{z_p}}{h} = \sqrt{z_p}$$

Für *s* gilt:

$$s = \frac{\varphi}{2\pi}$$

Alles zusammen:

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

$$s = \begin{cases} \dfrac{1}{2\pi}\cdot\arccos\dfrac{x_p}{\sqrt{z_p}} & falls\ y_p \geq 0 \\ 1 - \dfrac{1}{2\pi}\cdot\arccos\dfrac{x_p}{\sqrt{z_p}} & sonst \end{cases}$$

$$t = z_p$$

Und die Umkehrabbildung:

$$Textur^{-1}_{Paraboloid}(s,t) = \begin{pmatrix} \cos(2\pi\cdot s)\cdot\sqrt{t} \\ \sin(2\pi\cdot s)\cdot\sqrt{t} \\ t \end{pmatrix}$$

Anmerkung: Während die Texture-Mapping-Verfahren für die Kugel und den Kegel relativ eindeutig sind, kann man bei Paraboloiden verschiedene Techniken anwenden, die sich in Hinblick auf die Verzerrung an der Spitze unterscheiden. Mein Algorithmus wurde mit Ziel größtmöglicher Einfachheit und Geschwindigkeit entworfen, wobei ich stets den Kegel als Vorbild hatte.

Konvexes Polygon

Für die Texturierung eines konvexen Polygons sind mir zwei Verfahren eingefallen, die sich hinsichtlich Qualität und Zeitaufwand aber deutlich unterscheiden. Zunächst erkläre ich ausführlich die einfachere Methode der Texturierung:

(1) Simples Texture Mapping

Als ersten Schritt sind die Polygoneckkoordinaten in den zweidimensionalen Raum zu transformieren. Wenn die Polygon-Normale nicht senkrecht auf der z-Achse steht (z-Anteil ist Null), dann könnte z.B. diese Abbildung durch einfaches Weglassen der z-Koordinate geschehen.

$$x'_p = x_p$$
$$y'_p = y_p$$

Aufwändiger, aber stets funktionierend, ist die Idee, dass man den Normalenvektor und alle Punkte des Polygons, die in einer Ebene liegen, auf die z-Achse rotiert. Anschließend kann man stets problemlos die z-Koordinate weglassen.

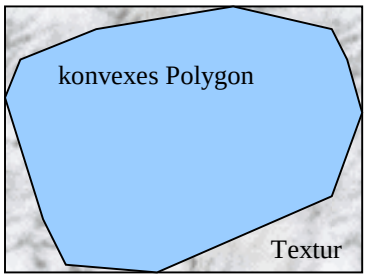
Man sucht die Extremwerte heraus, d.h. man speichert den kleinsten Wert x_{min} und y_{min} bzw. den größten Wert x_{max} und y_{max}. Diese dienen dazu, die Begrenzungen der Textur zu bestimmen, wobei x_{min} bzw. y_{min} auf 0 und x_{max} bzw. y_{max} auf 1 abgebildet werden:

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

$$s = \frac{x_p - x_{\min}}{x_{\max} - x_{\min}}$$

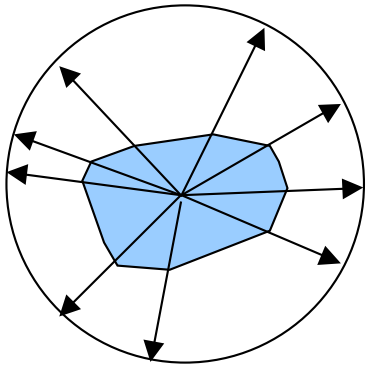
$$t = \frac{y_p - y_{\min}}{y_{\max} - y_{\min}}$$

Ein erheblicher Nachteil dieses sehr einfachen Verfahrens besteht darin, dass die Textur nicht komplett genutzt wird:



(2) Qualitativ besseres Verfahren

Am einfachen Verfahren stört mich, dass die Textur nur selten vollständig genutzt werden kann. Deshalb bilde ich das Polygon auf einen Einheitskreis ab, in diesem bilden Drehwinkel und Abstand zum Mittelpunkt die Texturkoordinaten *s* und *t*:



Nachdem alle Punkte vom 3D- in den 2D-Raum transformiert wurden (siehe Überlegungen beim einfachen Verfahren), ist der Mittelpunkt der Schwerpunkt des Polygons:

$$M = \begin{pmatrix} \frac{1}{n} \cdot \sum_{i=1}^{n} x_i \\ \frac{1}{n} \cdot \sum_{i=1}^{n} y_i \end{pmatrix}$$

Danach bestimme ich für alle Ecken des Polygons ihren Abstand vom Mittelpunkt und den Drehwinkel bezüglich der positiven x-Achse:

$$d_i = \sqrt{(x_i - x_m)^2 + (y_i - y_m)^2}$$

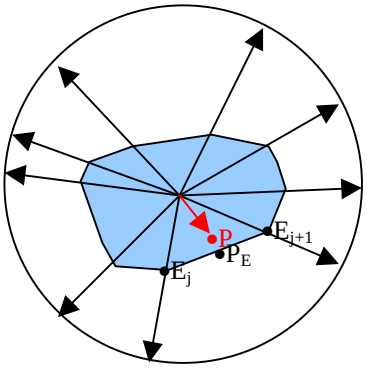
$$\alpha_i = \begin{cases} \arccos \frac{x_i - x_m}{d} & falls\ y_i - y_m \geq 0 \\ 2\pi - \arccos \frac{x_i - x_m}{d} & sonst \end{cases}$$

Wenn ein Punkt P des Polygons texturiert werden soll, so bestimme ich auch für diesen den Abstand und den Drehwinkel:

$$d_p = \sqrt{(x_p - x_m)^2 + (y_p - y_m)^2}$$

$$\alpha_p = \begin{cases} \arccos \frac{x_p - x_m}{d} & falls\ y_p - y_m \geq 0 \\ 2\pi - \arccos \frac{x_p - x_m}{d} & sonst \end{cases}$$

Anhand von α_p kann ich nun zuordnen, in welcher Partition des Polygons der Punkt P liegt:

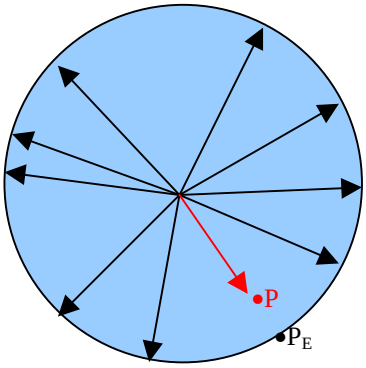


Aus den beiden benachbarten Ecken E_j und E_{j+1} kann man bestimmen, wie weit die Kante des Polygons in Richtung von P entfernt ist:

$$d_{Ep} = \frac{\alpha_p - \alpha_j}{\alpha_{j+1} - \alpha_j} \cdot \left(d_{j+1} - d_j\right) + d_j$$

Um eine Verzerrung auf den Einheitskreis zu erzielen, korrigiere ich den Abstand von P und den Abstand der Kante:

$$d_p^{'} = \frac{d_p}{d_{Ep}}$$



Nachdem ich nun sowohl Abstand als auch Drehwinkel bestimmt habe, kann ich diese beiden Werte als Texturkoordinaten verwenden:

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

$$s = d_p^{'}$$

$$t = \frac{\alpha_p}{2\pi}$$

Die Textur ist mit jetzt allerdings stark verzerrt, die obere linke Ecke der Textur (0,0) liegt im Schwerpunkt des Polygons. Dem kann man entgegen wirken, indem man den Kreis auf ein Quadrat abbildet und dieses dann als Grundlage für *s* und *t* nimmt.

Es wird die komplette Textur für das Polygon benutzt. Dies stellt den größten Unterschied zum vorher entwickelten einfachen Verfahren dar. Leider steigt der dazu notwendige Rechenaufwand enorm an, insbesondere die Winkelfunktionen sind kritisch. Einen Mittelweg kann man beschreiten, wenn man das Polygon in lauter Dreiecke zerlegt. Diese können schnell und einfach texturiert werden, durch Berücksichtigung vorberechneter Texturkoordinaten für die Eckpunkte können recht ansprechende Ergebnisse erzielt werden.

Anhang

Quelltext Aufgabe 19

Das komplette Texture Mapping findet in cgtexture.cpp statt.

cgtexture.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 7
//

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

#ifndef PI
const double PI = 3.14159265358979323846;
#endif

const unsigned int BUF_SIZE = 1024;
const unsigned int PATCH_SIZE = 20;

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

    run_ = false;
    rotate_ = false;

    zoom_ = 1.5;
    list_ = NULL;

    buildPatch(PATCH_SIZE);
}

CGTexture::~CGTexture() {
}

void CGTexture::drawScene() {

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

    drawObject();
    glPopMatrix();
}

void CGTexture::handleVertex(const Vector& v) {
    // Hier müssen die Texturkoordinaten berechnet werden
    glTexCoord2f(v[0]+0.5, v[2]+0.5);

    // draw patch
    glVertex3dv(v.rep());
}

void CGTexture::drawObject() {
    // draw geometry

    glBindTexture(GL_TEXTURE_2D, textureID_);

    int c=0;
    for(int j=0; j<num_; j++) {
        glBegin(GL_TRIANGLE_STRIP); // Performance

        for(int i=0; i<((num_+1)*2); i++) {
            handleVertex(list_[c]); c++;
```

```
        }

        glEnd();
    }
}

void CGTexture::buildPatch(int detail) {
    if(list_) delete list_;
    list_ = new Vector[(detail)*(4+((detail-1)*2))];

    double offset = 1.0/detail;

    double x;
    double z;
    num_ = detail;
    int c = 0;
    for(int j=0; j<detail; j++) {
        x = -0.5;
        z =  0.5 - (j+1)*offset;
        list_[c] = Vector(x,0,z); c++;

        x = -0.5;
        z =  0.5 - j*offset;
        list_[c] = Vector(x,0,z); c++;

        x = -0.5 + offset;
        z =  0.5 - (j+1)*offset;
        list_[c] = Vector(x,0,z); c++;

        for(int i=0; i<detail; ) {
            if( (c%2)!=0) {
                x = -0.5 + (i+1)*offset;
                z =  0.5 - (j)*offset;
                i++;
            } else {
                x = -0.5 + (i+1)*offset;
                z =  0.5 - (j+1)*offset;
            }
            list_[c] = Vector(x,0,z); c++;
        }
    }
}

void CGTexture::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

    // 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);

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

    in.close();
}

void CGTexture::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);

    // liest ein ppm-bild im format RGB ein (kein alpha-kanal!)
    readImage();

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

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

    // repeat texture if necessary
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // 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_RGBA, texWidth_, texHeight_, 0, GL_RGB,
GL_UNSIGNED_BYTE, image_);

    // enable texture mapping
    glEnable(GL_TEXTURE_2D);
}

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 'a': { rotate_ = !rotate_; break; };
    }
    onDraw();
}

void CGTexture::onIdle() {

    // rotate object
    if (run_)
        glRotatef(.5, 0.0, 1.0, 0.0);

    // rotate texture
    if (rotate_)
    {
        // use texture matrix stack
        glMatrixMode(GL_TEXTURE);

        // move origin
        glTranslatef(0.5, 0.5, 0);
        // rotate
        glRotatef(-1, 0, 0, 1);
        // move back
        glTranslatef(-0.5, -0.5, 0);

        // switch back to model view matrix stack
        glMatrixMode(GL_MODELVIEW);
    }

    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("church_spiral.ppm");
    //CGTexture sample("deep_spiral.ppm");

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "+          Hineinzoomen" << endl
         << "-          Herauszoomen" << endl
         << "a          Textur rotieren" << endl;

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