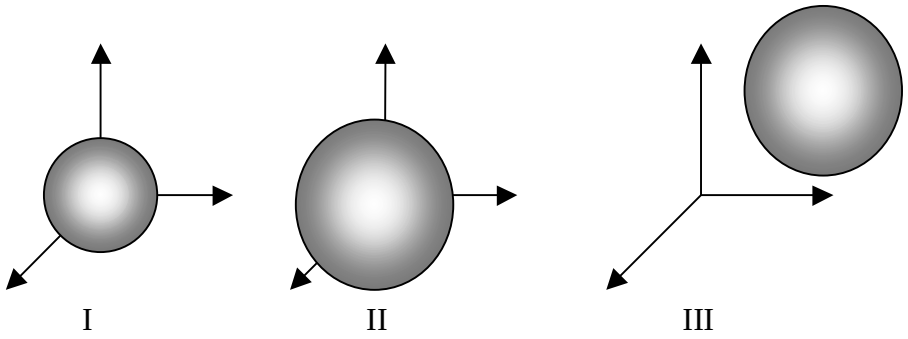
Aufgabe 3

Es erscheint sinnvoll, zuerst eine Kugel zu modellieren, deren Mittelpunkt im Koordinatenursprung liegt und die den Radius 1 besitzt („Einheitskugel“). Anschließend werden alle Punkte mit dem Radius skaliert und entlang des Vektors, der den „wahren“ Mittelpunkten repräsentiert, verschoben. Nachfolgende Graphik veranschaulicht den Sachverhalt:

Abbildung 1



Die Tessellation findet nur in Stufe I statt, die Stufen II und III sind von der generierten Struktur der Kugel unabhängig, da sie nur mit den einzelnen Vektorkoordinaten hantieren.

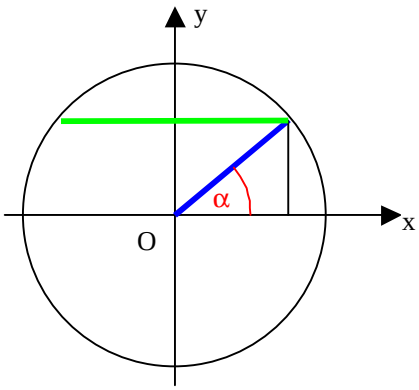
Noch eine Anmerkung vorweg

Meine Algorithmen zielen auf eine gute Verständlichkeit ab und sind nicht geschwindigkeitsoptimiert. Viele Vektoren bzw. Winkel werden redundant berechnet, was man in einer echten Anwendung natürlich vermeidet.

Viereckstessellation

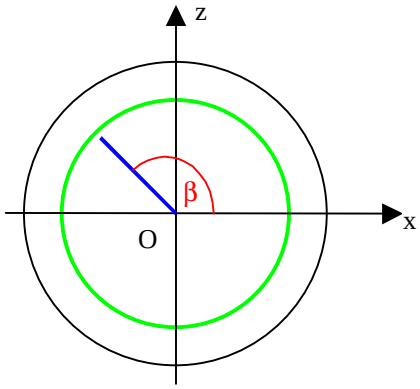
Ein Vektor im R^3 kann entweder über drei Koordinaten (kartesisches System) oder zwei Winkel und eine Länge (Kugelkoordinaten) dargestellt werden. Insbesondere der zweite Weg ist sehr einfach zu beschreiten, wenn man – wie eingangs erklärt – die Länge 1 annimmt. Betrachtet man eine Kugel entlang der z-Achse, dann sieht man einen Kreis:

Abbildung 2



Der Winkel α rotiert um die z-Achse. Wenn man nun den dadurch beschriebenen normierten Vektor (blaue Linie) um die y-Achse rotiert, dann entsteht aus der Sicht entlang der y-Achse das Bild:

Abbildung 3



Der blaue Vektor wirkt verzerrt, hat aber weiterhin die Länge 1. Der grüne Kreis war in der Ansicht entlang der z-Achse noch eine Linie, er stellt eine Breite auf einem Globus dar.
Fasst man diese Vorstellung mit Hilfe trigonometrischer Funktionen in Gleichungen zusammen, so sieht das System sehr kompakt aus:

$$\alpha \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right],\ \beta \in [0, 2\pi]$$
$$x = \cos\alpha \cdot \sin\beta$$
$$y = \sin\alpha$$
$$z = \cos\alpha \cdot \cos\beta$$

Der Grad der Tessellation wird nun dadurch bestimmt, mit welcher Schrittweite die Winkel α bzw. β ihre Intervalle durchlaufen.

Die Vorgehensweise formuliert sich in Pseudocode:

```
n = Tessellationsgrad

δα = π/n
δβ = 2π/n

α = -π/2
while α<π/2 do
{
   β = 0
   while β<2π do
   {
      x1 = cos(α)sin(β)
      y1 = sin(α)
      z1 = cos(α)cos(β)

      x2 = cos(α+δα)sin(β)
      y2 = sin(α+δα)
      z2 = cos(α+δα)cos(β)

      x3 = cos(α+δα)sin(β+δβ)
      y3 = sin(α+δα)
      z3 = cos(α+δα)cos(β+δβ)
```

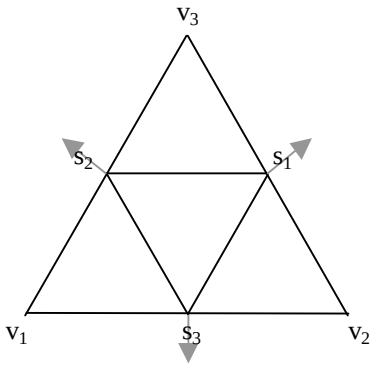
```
            x4 = cos(α)sin(β+δβ)
            y4 = sin(α)
            z4 = cos(α)cos(β+δβ)

            DrawQuad(x1,y1,z1, x2,y2,z2, x3,y3,z3, x4,y4,z4)
            β += δβ
        }
        α += δα
    }
```

Ich habe diesen Algorithmus in Aufgabe 4 als OpenGL-Programm implementiert, dort findet sich auch der entsprechende Sourcecode in C++.

Dreieckstessellation

Jede Seite eines regelmäßigen Tetraeders, die ein gleichseitiges Dreieck ist, wird rekursiv in 4 ebenfalls gleichseitige Dreiecke zerlegt, indem die Seitenhalbierenden jeweils verbunden werden.

Abbildung 4



Die zu tessellierende Kugel ist der Umkreis des Tetraeders. Die Seitenhalbierenden liegen daher *innerhalb* der Kugel. Um eine gute Approximation der Kugel zu erreichen, müssen die Seitenhalbierenden derart transformiert werden, dass sie *auf* der Kugeloberfläche liegen. Am besten eignet sich dafür die Umwandlung in Kugelkoordinaten (wie in Aufgabe a), so dass die Länge auf 1 gesetzt wird (da sie kleiner 1 war, deute ich dies durch Pfeile in obiger Grafik an), eine anschließende Rücktransformation in ein kartesisches System ist für die Bildschirmdarstellung notwendig.

Die Zerlegung eines Dreieckes in 4 kleinere kann rekursiv erfolgen, wobei die Rekursionstiefe ein Maß für die Qualität darstellt. Zu beachten ist das exponentielle Wachstum an Dreiecken, es wird eine Anzahl in der Größenordnung $4^{Rekursionstiefe+1}$ erzeugt.

Der Pseudocode für diesen Algorithmus lautet:

```
// Vektor auf Länge 1 normieren
// Parameter müssen per Referenz übergeben werden !
function Normalize(x, y, z)
{
      length = sqrt(x²+y²+z²)
      x = x/length
      y = y/length
      z = z/length
}

// gleichseitiges Dreick tessellieren und zeichnen
function TessellateTriangle(v₁, v₂, v₃, depth)
{
      if (depth>1)
      {
            Vector s₁, s₂, s₃

            // Seitenhalbierende ermitteln
            s₁.x = (v₂.x+v₃.x)/2
            s₁.y = (v₂.y+v₃.y)/2
            s₁.z = (v₂.z+v₃.z)/2
            s₂.x = (v₁.x+v₃.x)/2
            s₂.y = (v₁.y+v₃.y)/2
            s₂.z = (v₁.z+v₃.z)/2
            s₃.x = (v₁.x+v₂.x)/2
            s₃.y = (v₁.y+v₂.y)/2
            s₃.z = (v₁.z+v₂.z)/2

            // auf Kugeloberfläche projezieren
            Normalize (s₁.x, s₁.y, s₁.z)
            Normalize (s₂.x, s₂.y, s₂.z)
            Normalize (s₃.x, s₃.y, s₃.z)

            // rekursiv weiter tessellieren
            TessellateTriangle(s₁, s₂, s₃, depth-1)
            TessellateTriangle(v₁, s₃, s₂, depth-1)
            TessellateTriangle(s₃, v₂, s₁, depth-1)
            TessellateTriangle(s₂, s₁, v₃, depth-1)
      }
      else
            DrawTriangle(v₁, v₂, v₃)
}
```

Das besonders interessante an diesem Verfahren besteht darin, dass jeder (!) Figur, deren Eckpunkte alle auf der Kugeloberfläche und nicht in einer Ebene liegen, als Ausgangsfigur geeignet ist, es sich also nicht notwendigerweise um einen Tetraeder handeln muss.

Eingangs habe ich beschrieben, dass die Kugel den Umkreis des Tetraeders bildet. Gleichzeitig haben Tetraeder die interessante Eigenschaft, dass sie in einen Würfel einbeschreiben kann. Um eine im Koordinatenursprung zentrierte Kugel mit dem Radius 1 zu erhalten, muss der Diagonale des Würfels das Doppelte, also 2, betragen. Der Würfel setzt sich aus 8 Vektoren zusammen, die man an den folgenden Koordinaten findet:

$$v_1 = \begin{pmatrix} w \\ w \\ w \end{pmatrix}, v_2 = \begin{pmatrix} w \\ w \\ -w \end{pmatrix}, v_3 = \begin{pmatrix} w \\ -w \\ -w \end{pmatrix}, v_4 = \begin{pmatrix} w \\ -w \\ w \end{pmatrix}, v_5 = \begin{pmatrix} -w \\ w \\ w \end{pmatrix}, v_6 = \begin{pmatrix} -w \\ w \\ -w \end{pmatrix}, v_7 = \begin{pmatrix} -w \\ -w \\ -w \end{pmatrix}, v_8 = \begin{pmatrix} -w \\ -w \\ w \end{pmatrix}$$

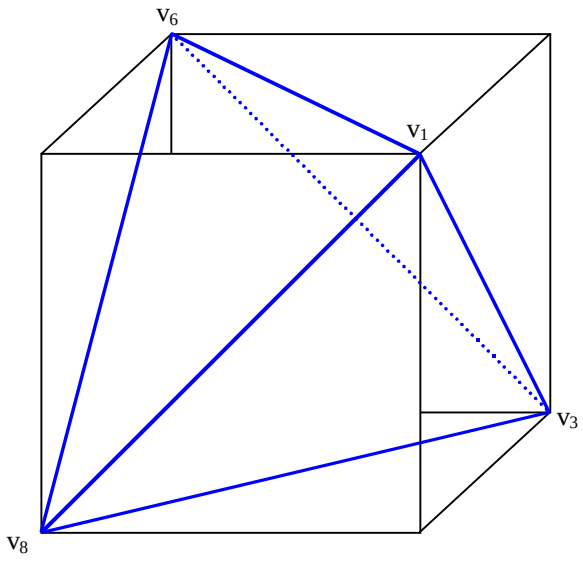
$$w = \frac{1}{\sqrt{3}}$$

Davon sind nur 4 Vektoren für den Tetraeder interessant, namentlich v_1, v_8, v_6 und v_3, d.h.

$$v_1 = \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{pmatrix}, v_3 = \begin{pmatrix} \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{3}} \end{pmatrix}, v_6 = \begin{pmatrix} -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{3}} \end{pmatrix}, v_8 = \begin{pmatrix} -\frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{pmatrix}$$

In einer Grafik erkennt man den Zusammenhang zwischen Tetraeder und Würfel besser:

Abbildung 5



Unter Verwendung der Formel für die Bestimmung der Diagonalen in einem Würfel entsteht:

$$d = \sqrt{3} \cdot a$$

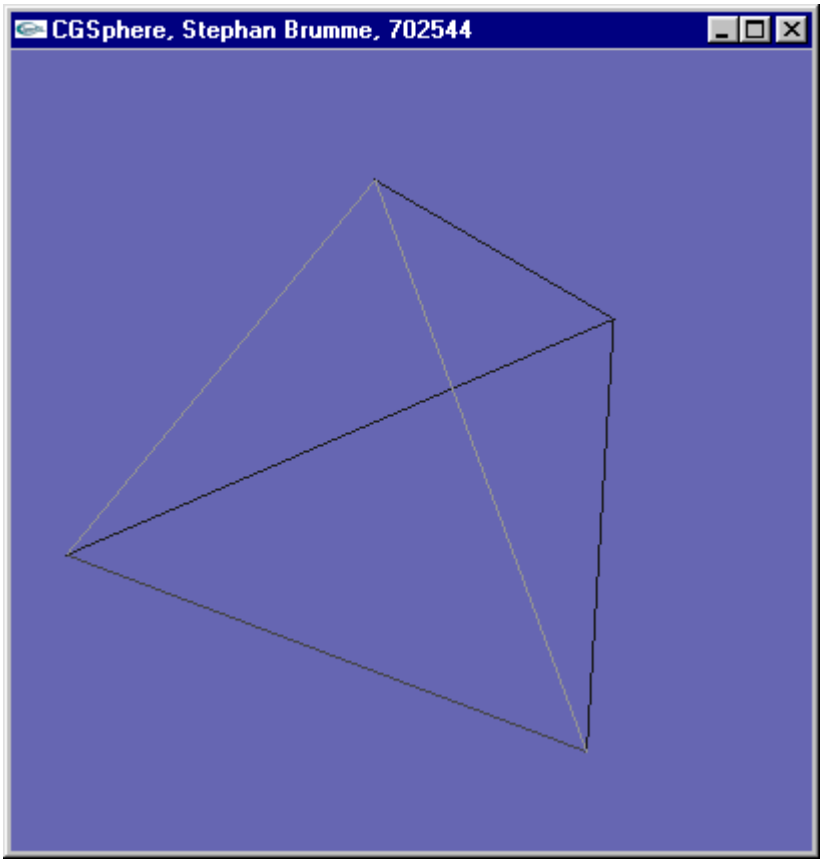
$$\frac{a}{2} = r \cdot \frac{1}{\sqrt{3}}$$

Der Radius ist bei der Einheitskugel 1, da der Mittelpunkt genau in der Mitte des Würfels liegt, gilt:

$$w = \frac{a}{2} = \frac{1}{\sqrt{3}}$$

was die oben angegebenen Koordinaten bestätigt.

Abbildung 6

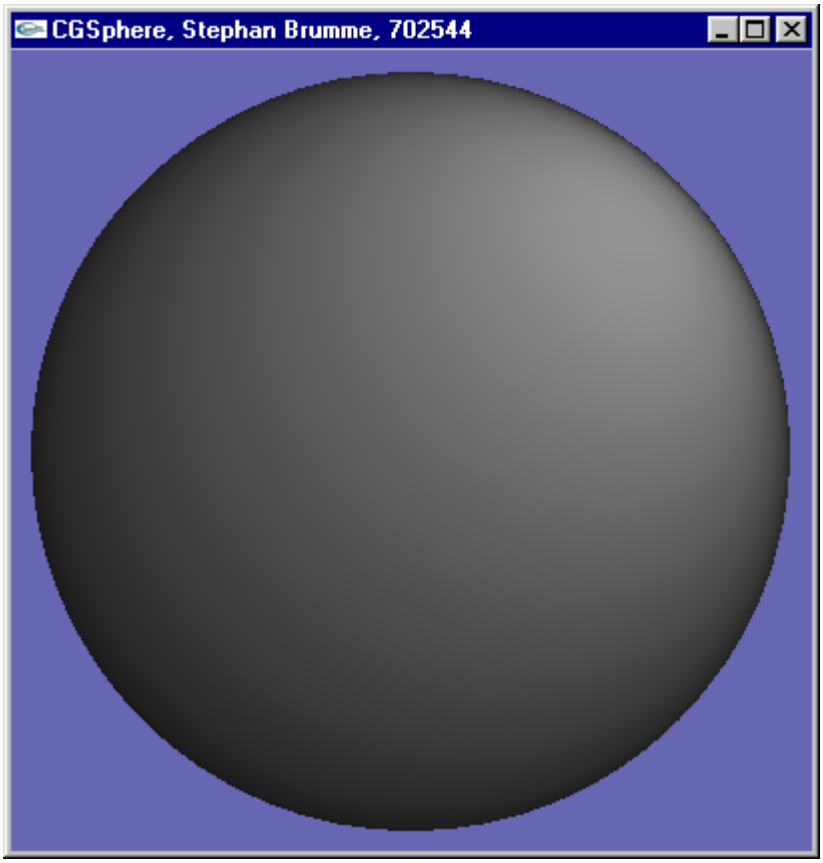


In Aufgabe 4 wurde auch dieser Algorithmus in OpenGL umgesetzt.

Aufgabe 4

Mein OpenGL-Programm ist in der Lage, beide in Aufgabe 3 vorstellten Algorithmen umzusetzen. Nach dem Programmstart erscheint eine nach dem Dreiecksverfahren modellierte Figur, die einer Kugel schon sehr nahe kommt. Insbesondere die Aktivierung von `GL_SMOOTH` zeigt sehr gute Ergebnisse:

Abbildung 7



Die Tastaturkommandos lauten:

Taste	Aktion
`t`	Umschalten zwischen beiden Tessellationsverfahren
`s`	De-/Aktivierung des Weichzeichners
`w`	Umschalten zwischen Drahtgitter- und Hüllenmodell
`+,-`	Erhöhung/Senkung der Tessellationstiefe
`x,y,z`	Drehung und die jeweiligen Achsen in positiver Orientierung
`X,Y,Z`	Drehung und die jeweiligen Achsen in negativer Orientierung
`q`	Programm beenden

Es ist auf Groß-/Kleinschreibung zu achten !
Weiterhin besteht ein großer Unterschied bezüglich der Tessellationstiefe zwischen beiden Verfahren, da die Anzahl der Primitive mit der Viereckmethode linear, mit der Dreiecksmethode jedoch exponentiell wächst.

Ich nahm keinerlei Änderungen in CGApplication.cpp/CGApplication.h vor, daher drucke ich den entsprechenden Quellcode auch nicht ab. Zusätzlich wurde die Vektorklasse benutzt, die aber auch in ihrer ursprünglichen Form blieb (Vector.cpp/Vector.h).

source file "cgsphere.h"

```
// Computergrafik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Programmrahmen fuer Aufgabe 4

// Stephan Brumme, 702544
// last changes: May 07, 2001

#ifndef CG_SPHERE_H
#define CG_SPHERE_H

#include "cgapplication.h"
#include "vector.h"


class CGSphere : public CGApplication {
public:
    CGSphere();
    virtual ~CGSphere();

    // The following event methods will be implemented.
    virtual void onInit();
    virtual void onDraw();
    virtual void onSize(int newWidth, int newHeight);
    virtual void onKey(unsigned char key);

private:
    void TessellateTriangle(const Vector& v1, const Vector& v2, const Vector& v3, int nLevel);
    GLfloat m_fRadius;
    GLint   m_nTessellationLevel;
    bool    m_bUseFlatShade;
    bool    m_bUseWireframe;
    bool    m_bUseTriangleTessellation;
};

#endif // CG_SPHERE_H
```

source file "cgsphere.cpp"

```
// Computergrafik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Programmrahmen fuer Aufgabe 4


// Stephan Brumme, 702544
// last changes: May 07, 2001


// ------------------------------------------------------------
// It might be useful to use the class "Vector" presented in
// the exercises
// ------------------------------------------------------------

#include "cgsphere.h"
#include <math.h>



CGSphere::CGSphere() {
    // adjust to fit the window
    m_fRadius            = 0.95;
    m_nTessellationLevel = 6;

    m_bUseFlatShade      = true;
    m_bUseWireframe      = false;
    m_bUseTriangleTessellation = true;
}

CGSphere::~CGSphere() {
```

```
}

void CGSphere::onInit() {
    // Set background color.
    glClearColor(0.4, 0.4, 0.7, 0); // blue

    // Set Shading
    if (m_bUseFlatShade)
       glShadeModel(GL_FLAT);
    else
       glShadeModel(GL_SMOOTH);

    // Enable Lighting
    GLfloat am[] = {0.5, 0.5, 0.5, 1.0};
    GLfloat df[] = {1.0, 1.0, 1.0, 1.0};
    GLfloat sp[] = {0.0, 0.0, 0.0, 1.0};
    GLfloat pos[] = {1.0, 1.0, 0.0, 1.0};
    glEnable(GL_LIGHTING);
    glLightfv(GL_LIGHT0, GL_AMBIENT, am);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, df);
    glLightfv(GL_LIGHT0, GL_SPECULAR, sp);
    glLightfv(GL_LIGHT0, GL_POSITION, pos);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.2);
    glEnable(GL_LIGHT0);

    // Let OpenGL compute normals automatically
    glEnable(GL_NORMALIZE);

    glEnable(GL_DEPTH_TEST);
}

void CGSphere::TessellateTriangle(const Vector& v1, const Vector& v2, const Vector& v3, int
nLevel)
{
    if (nLevel == 1)
    {
        // Dreieck zeichnen (eventuell als Drahtgittermodell)
        if (m_bUseWireframe)
            glBegin(GL_LINE_LOOP);
        else
            glBegin(GL_TRIANGLES);

        glVertex3f(v1[0], v1[1], v1[2]);
        glVertex3f(v2[0], v2[1], v2[2]);
        glVertex3f(v3[0], v3[1], v3[2]);
        glEnd();
    }
    else
    {
        // Mittelpunkte der Seiten bestimmen
        Vector s1((v2[0]+v3[0])/2, (v2[1]+v3[1])/2, (v2[2]+v3[2])/2);
        Vector s2((v1[0]+v3[0])/2, (v1[1]+v3[1])/2, (v1[2]+v3[2])/2);
        Vector s3((v2[0]+v1[0])/2, (v2[1]+v1[1])/2, (v2[2]+v1[2])/2);

        // und normalisieren
        s1.normalize();
        s2.normalize();
        s3.normalize();

        // weiter tessellieren
        TessellateTriangle(s1, s2, s3, nLevel-1);
        TessellateTriangle(v1, s3, s2, nLevel-1);
        TessellateTriangle(v2, s1, s3, nLevel-1);
        TessellateTriangle(v3, s1, s2, nLevel-1);
    }
}

void CGSphere::onDraw() {
    // Clear framebuffer using background color.
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Set current color for drawing.
    glColor3f (1.0, 1.0, 1.0); // white

    glPushMatrix();
```

```
    // Radius anpassen (da eigentlich nur Einheitskugel berechnet)
    glScalef (m_fRadius, m_fRadius, m_fRadius);

    // urgently needed ...
    const GLfloat fPi         = 3.1415926;

    if (m_bUseTriangleTessellation)
    {
        // triangle tessellation

        // Eckpunkte festlegen
        const GLfloat fOffset = 1/sqrt(3);
        Vector v[4] = { Vector( fOffset, fOffset, fOffset),
                        Vector(-fOffset,-fOffset, fOffset),
                        Vector(-fOffset, fOffset,-fOffset),
                        Vector( fOffset,-fOffset,-fOffset)};

        // mit den 4 Seiten des Tetraeders die Tessellation beginnen
        TessellateTriangle(v[0], v[1], v[2], m_nTessellationLevel);
        TessellateTriangle(v[0], v[1], v[3], m_nTessellationLevel);
        TessellateTriangle(v[0], v[2], v[3], m_nTessellationLevel);
        TessellateTriangle(v[1], v[2], v[3], m_nTessellationLevel);
    }
    else
    {
        // quad tessellation

        // Winkelschrittweite
        const GLfloat fDeltaAlpha =   fPi/m_nTessellationLevel;
        const GLfloat fDeltaBeta  = 2*fPi/m_nTessellationLevel;

        // "Breitengrade"
        for (GLfloat fAlpha = -fPi/2; fAlpha < fPi/2; fAlpha += fDeltaAlpha)
        {
            // "Längengrade"
            for (GLfloat fBeta = 0; fBeta < 2*fPi; fBeta += fDeltaBeta)
            {
                // 4 Eckpunkte berechnen
                // Formeln in Aufgabe 3 erläutert
                Vector v1(cos(fAlpha)*sin(fBeta),
                          sin(fAlpha),
                          cos(fAlpha)*cos(fBeta));

                Vector v2(cos(fAlpha+fDeltaAlpha)*sin(fBeta),
                          sin(fAlpha+fDeltaAlpha),
                          cos(fAlpha+fDeltaAlpha)*cos(fBeta));

                Vector v3(cos(fAlpha+fDeltaAlpha)*sin(fBeta+fDeltaBeta),
                          sin(fAlpha+fDeltaAlpha),
                          cos(fAlpha+fDeltaAlpha)*cos(fBeta+fDeltaBeta));

                Vector v4(cos(fAlpha)*sin(fBeta+fDeltaBeta),
                          sin(fAlpha),
                          cos(fAlpha)*cos(fBeta+fDeltaBeta));

                // Viereck zeichnen (eventuell als Drahtgittermodell)
                if (m_bUseWireframe)
                    glBegin(GL_LINE_LOOP);
                else
                    glBegin(GL_QUADS);

                glVertex3f(v1[0], v1[1], v1[2]);
                glVertex3f(v2[0], v2[1], v2[2]);
                glVertex3f(v3[0], v3[1], v3[2]);
                glVertex3f(v4[0], v4[1], v4[2]);
                glEnd();
            }
        }
    }

    glPopMatrix();

    glFlush();
    // Do not forget to swap front- and backbuffer!
```

```
    swapBuffers();
}

void CGSphere::onKey(unsigned char key) {
    cout << key << endl;
    switch (key) {
        case 'q': exit(0); break; // Quit application.

            // Tessellationsgrad erhöhen
        case '+': m_nTessellationLevel++;
                  cout << m_nTessellationLevel << endl;
                  break;

            // Tessellationsgrad senken
        case '-': if (m_nTessellationLevel > 1)
                     m_nTessellationLevel--;
                  cout << m_nTessellationLevel << endl;
                  break;

            // Rotationen rückgängig machen
        case 'r': glLoadIdentity();
                  break;

            // Shadingmodell umschalten flat/smooth
        case 's': m_bUseFlatShade = !m_bUseFlatShade;
                  if (m_bUseFlatShade)
                     glShadeModel(GL_FLAT);
                  else
                     glShadeModel(GL_SMOOTH);
                  break;

            // Tessellationsmethode umschalten Dreieck/Viereck
        case 't': m_bUseTriangleTessellation = !m_bUseTriangleTessellation;
                  break;

            // umschalten solid/Drahtgitter
        case 'w': m_bUseWireframe = !m_bUseWireframe;
                  break;

            // um x-Achse drehen
        case 'x': glRotated(5, 1, 0, 0);
                  break;
        case 'X': glRotated(-5, 1, 0, 0);
                  break;
            // um y-Achse drehen
        case 'y': glRotated(5, 0, 1, 0);
                  break;
        case 'Y': glRotated(-5, 0, 1, 0);
                  break;
            // um z-Achse drehen
        case 'z': glRotated(5, 0, 0, 1);
                  break;
        case 'Z': glRotated(-5, 0, 0, 1);
                  break;
    }

    // Neuzeichnen erzwingen
    glutPostRedisplay();
}

void CGSphere::onSize(int newWidth, int newHeight) {
    if((newWidth > 0) && (newHeight > 0)) {
        // Adjust OpenGL-Viewport according to new window dimensions.
        glViewport(0, 0, newWidth - 1, newHeight - 1);

        // Switch to modelview matrix stack and initialize
        // with identity matrix.
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }
}

// main program
int main(int argc, char* argv[]) {
    // Generate an instance of the sample application.
```

```
    CGSphere* sample = new CGSphere();

    // Starts the sample application.
    sample->start("CGSphere, Stephan Brumme, 702544");
    return(0);
}
```

Aufgabe 5

Etwas abweichend vom Original-Programmrahmen sind alle Pixel als `GL_FLOAT` implementiert, um mir so die Arbeit mit Umrechnung bzw. Rundungen zu vereinfachen.

Die drei zu implementierenden Brushmuster verteilen sich auf die Funktionen `setConstantPattern`, `setRadialPattern` und `setArbitraryPattern`. Erstere setzt alle Alphawerte im `m_brushPattern` auf den übergebenen Parameter-Wert, momentan ist dies 0,5. setRadialPattern basiert auf der Formel

$$\alpha = 1 - d(P,M) / \max_{P} d(P,M)$$

mit P als den jeweiligen Punkt im Pinsel und M als Mittelpunkt. Das Ergebnis ist ein runder Airbrush, dessen Intensität in der Mitte 100% beträgt und zu allen Seiten hin gleichmäßig auf 0% abfällt.
Das Einlesen eines Pinsel ist momentan auf die Datei 32x32.pat beschränkt, die die einzelnen Alphawerte zeilenweise enthält. Es wird keine Überprüfung durchgeführt, ob die Anzahl der darin gespeicherten Zahlen ausreichend ist, ich liefere die Datei mit 32x32 Werten aus.

Mit der rechten Maustaste kann man einen Brush aus der Zeichenfläche aufnehmen, d.h. die Pixel unter dem Mauszeiger werden in `brushPattern_` kopiert. Zu beachten bleibt, dass alle Alphawerte auf 100% gesetzt werden.

Zusammenfassend lauten die benötigten Tasten:

Taste	Aktion
c	Bildschirm löschen
r	radialer Airbrush (Standard)
k	konstanter Airbrush
e	Airbrush aus 32x32.pat laden
0-7	Farbe ändern
q	Programm beenden

Quelltext

source file "cgairbrush.h"

```
// Computergrafik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Programmrahmen fuer Aufgabe 5

// Stephan Brumme, 702544
// last changes: May 07, 2001

#ifndef CG_BRUSH_H
#define CG_BRUSH_H

#include "cgapplication.h"

class CGAirbrush : public CGApplication {
public:
    CGAirbrush(unsigned int brushPatternSize);
    virtual ~CGAirbrush();

    // Auf folgende Ereignisse soll reagiert werden:
    virtual void onInit();
    virtual void onDraw();
    virtual void onSize(unsigned int newWidth, unsigned int newHeight);
    virtual void onButton(MouseButton button, int x, int y);
```

```
    virtual void onMove(int x, int y);
    virtual void onKey(unsigned char key);

    void setArbitraryPattern();
    void setConstantPattern(float fOpacity);
    void setRadialPattern();
    void setColor(GLfloat red, GLfloat green, GLfloat blue);

private:
    // Eine Hilfsklasse, die eine Farbe als
    // Tupel (r, g, b, alpha) darstellt.
    struct Color {
        GLfloat r;
        GLfloat g;
        GLfloat b;
        GLfloat alpha;
    };

    // Speichert die Groesse der Schablone.
    // Die Schablone ist quadratisch.
    unsigned int m_brushPatternSize;

    // Enthaelt die Farbinformationen der einzelnen Pixel
    // der Schablone.
    Color*       m_brushPattern;
};

#endif // CG_BRUSH_H
```

source file "cgairbrush.cpp"

```
// Computergrafik I
// Prof. Dr. Juergen Doellner
// Sommersemester 2001
//
// Programmrahmen fuer Aufgabe 5

// Stephan Brumme, 702544
// last changes: May 07, 2001

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


CGAirbrush::CGAirbrush(unsigned int brushPatternSize)
    : m_brushPatternSize(brushPatternSize) {
    assert(brushPatternSize >= 10); // sollte groesser/gleich 10x10 sein!

    // erzeuge brush pattern dynamisch
    m_brushPattern = new Color[brushPatternSize*brushPatternSize];

    // Initialisiere die RGB- und Alpha-Werte der Schablone
    // kreisförmiger Pinsel
    setRadialPattern();
    // rot
    setColor(1.0, 0.0, 0.0);
}

CGAirbrush::~CGAirbrush() {
    // Nicht vergessen: Freigabe des dynamisch erzeugten Objekte
    delete [] m_brushPattern;
}

void CGAirbrush::onInit() {
    // Setze die Hintergrundfarbe auf weiss
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // Die Schablonen-Reihen sind dicht gepackt
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    // Schalte das Blending ein:
    glEnable(GL_BLEND);
    // mische Farbwerte
```

```
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    glFlush();
}

void CGAirbrush::onDraw() {
    // Loesche den Farb-Buffer
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();

    // Wir verwenden Single-Buffering und koennen
    // die bisher gezeichneten Figuren nicht mehr
    // restaurieren.
}

void CGAirbrush::onSize(unsigned int newWidth, unsigned int newHeight) {
    // Nach jeder Groessenaenderung des OpenGL-Fensters
    // MUSS der Viewport entsprechend angepasst werden:
    glViewport(0, 0, newWidth - 1, newHeight - 1);

    // Lege das zu verwendende Koordinatensystem fest:
    // die linke untere Fensterecke hat die Koordinaten (0, 0)
    // und die rechte obere die Koord. (newWidth, newHeight)
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, newWidth, 0.0, newHeight);
}

void CGAirbrush::onButton(MouseButton button, int x, int y) {

    if (button == LeftMouseButton)
        // Mache das selbe wie bei einer Mausbewegung:
        onMove(x, y);
    else
    {
        // setze Pinsel auf Bildinhalt an der aktuellen Position
        glReadPixels(x-m_brushPatternSize/2, y-m_brushPatternSize/2,
                     m_brushPatternSize, m_brushPatternSize,
                     GL_RGBA, GL_FLOAT, m_brushPattern);
    }
}

void CGAirbrush::onMove(int x, int y)
{
    // lege obere linke Ecke fest,
    // sodass Mauscursor der Mittelpunkt der gezeichneten Fläche ist
    glRasterPos2i(x-m_brushPatternSize/2, y-m_brushPatternSize/2);
    // zeichne Pinsel
    glDrawPixels(m_brushPatternSize, m_brushPatternSize,
                 GL_RGBA, GL_FLOAT, m_brushPattern);

    // Pipeline abarbeiten
    glFlush();
}

void CGAirbrush::onKey(unsigned char key) {
    // Keys
    switch(key) {
        // beenden
    case 'q' : exit(0); break;

        // Farbe setzen
    case '0' : setColor(1.0, 0.0, 0.0); break;
    case '1' : setColor(0.0, 1.0, 0.0); break;
    case '2' : setColor(0.0, 0.0, 1.0); break;
    case '3' : setColor(1.0, 1.0, 0.0); break;
    case '4' : setColor(1.0, 0.0, 1.0); break;
    case '5' : setColor(0.0, 1.0, 1.0); break;
    case '6' : setColor(1.0, 1.0, 1.0); break;
    case '7' : setColor(0.0, 0.0, 0.0); break;

        // Buffer löschen (weiss)
    case 'c' : glClear(GL_COLOR_BUFFER_BIT); glFlush(); break;

        // verschiedene Pinsel setzen
```

```
    case 'r' : setRadialPattern(); break;
    case 'k' : setConstantPattern(0.5); break;
    case 'e' : setArbitraryPattern(); break;

    default:
        cerr << "No action defined for key: " << key << endl;
    }
}

void CGAirbrush::setRadialPattern()
{
    // Mitte des Pinsels bestimmen
    float nCenterX = m_brushPatternSize/2;
    float nCenterY = m_brushPatternSize/2;

    // Durchmesser des runden Pinsels
    float fMaxDistance = m_brushPatternSize/2;

    for (unsigned int nX = 0; nX < m_brushPatternSize; nX++)
        for (unsigned int nY = 0; nY < m_brushPatternSize; nY++)
        {
            // Abstand zur Pinselmitte
            float fDistance = sqrt((nCenterX-nX) * (nCenterX-nX) +
                                   (nCenterY-nY) * (nCenterY-nY));

            // 100% opak in der Pinselmitte, 0% am Rand
            float fOpacity = 1 - fDistance/fMaxDistance;

            // verhindere Underflow
            if (fOpacity < 0)
                fOpacity = 0;
            fOpacity *= 0.5;

            // setze Opazität
            m_brushPattern[nX*m_brushPatternSize+nY].alpha = fOpacity;
        }
}

void CGAirbrush::setConstantPattern(float fOpacity)
{
    // kompletter Pinsel mit einer Opazität
    for (unsigned int nX = 0; nX < m_brushPatternSize; nX++)
        for (unsigned int nY = 0; nY < m_brushPatternSize; nY++)
            m_brushPattern[nX*m_brushPatternSize+nY].alpha = fOpacity;
}

void CGAirbrush::setArbitraryPattern()
{
    // öffne Datei
    ifstream hPattern("32x32.pat");

    // lies Pinsel ein
    for (unsigned int nX = 0; nX < m_brushPatternSize; nX++)
        for (unsigned int nY = 0; nY < m_brushPatternSize; nY++)
            hPattern >> m_brushPattern[nX*m_brushPatternSize+nY].alpha;
}

void CGAirbrush::setColor(GLfloat red, GLfloat green, GLfloat blue)
{
    // ändere alle Farbwerte im Pinsel
    for (unsigned int nX = 0; nX < m_brushPatternSize; nX++)
        for (unsigned int nY = 0; nY < m_brushPatternSize; nY++)
        {
            m_brushPattern[nX*m_brushPatternSize+nY].r = red;
            m_brushPattern[nX*m_brushPatternSize+nY].g = green;
            m_brushPattern[nX*m_brushPatternSize+nY].b = blue;
        }
}

int main(int argc, char* argv[]) {
    CGAirbrush airbrush(32);
    airbrush.start("CGAirbrush, Stephan Brumme, 702544", false);
    return(0);
}
```