Aufgabe 6

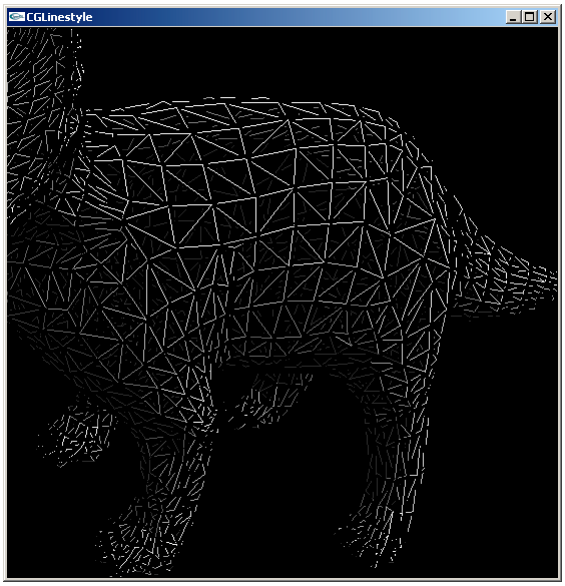
Diese Aufgabe wurde bereits auf dem Übungsblatt 2 gestellt, aufgrund der Verlängerung des Bearbeitungszeitraumes fiel sie jedoch mit dem Übungsblatt 3 zusammen.

Die Steuerung der Beispielapplikation entspricht den Anforderungen der Aufgabenstellung:

Taste	Aktion
ESC	Programm beenden
Leertaste	Objekt drehen
+	in die Szene hineinzoomen
-	aus der Szene herauszoomen
c	Culling de-/aktivieren
l	Beleuchtung de-/aktivieren
s	segmented surfaces
h	haloed wires
1	halo verringern
2	halo verstärken
3	Abstand der Segmente verringern
4	Abstand der Segmente vergrößern

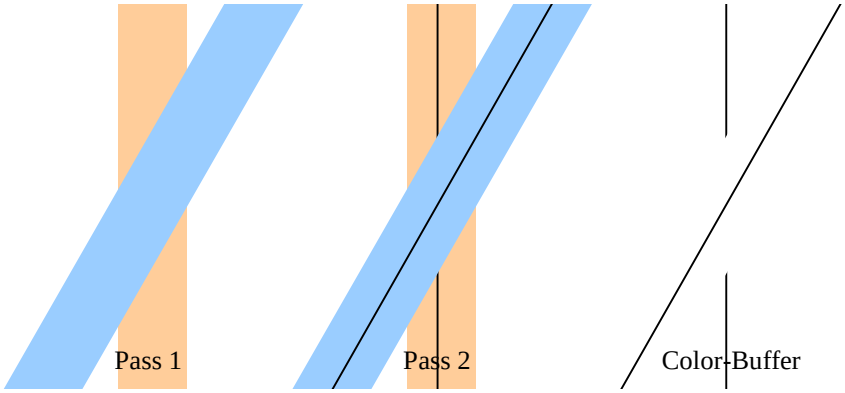
Haloed Wires

Der „haloed wires“-Effekt lässt um die einzelnen Linien eines Drahtgittermodells einen Freiraum, so dass sie sich besser voneinander unterscheiden lassen, wobei auch darauf geachtet wird, dass im Vordergrund liegende Linien nicht überzeichnet werden.



Technisch erreicht man diesen Effekt durch zweifaches Zeichnen der kompletten Geometrie. Im ersten Durchgang wird dazu eine hohe Linienstärke gewählt. Der Color-Buffer ist aber für Schreibzugriffe gesperrt, nur der z-Buffer darf verändert werden. Anschließend wird die Linienstärke verringert und in den Color-Buffer geschrieben. Dabei müssen die Werte im z-Buffer beachtet werden, so dass entfernte Linien nicht weiter vorn liegende Linien überschreiben können.

In der folgenden Skizze werden zunächst zwei dicke Linien (blau und beige) gezeichnet, sie verändern aber nur den z-Buffer und tauchen im Color-Buffer nicht auf. Anschließend werden im zweiten Renderpass die dünnen, schwarzen Linien gezeichnet, sie ändern den z-Buffer nicht mehr, greifen aber auf diesen lesend zu und werden daher nicht überall gezeichnet (senkrechte schwarze Linie ist *nicht* durchgehend). Das tatsächliche visuelle Resultat ist im rechten Drittel zu sehen:



Damit auch tatsächlich die Linien gezeichnet werden, muss die Tiefentestfunktion noch geändert werden. Standardmäßig wird `GL_LESS` verwendet, hier muss man `GL_LEQUAL` benutzen.

Die Initialisierung gleicht in weiten Teilen der vom „segmented surfaces“-Effekt:
Es ist der z-Buffer zu aktivieren und gemeinsam mit dem Colorbuffer zu löschen:

```
// clear buffers
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Natürlich wurde vorher OpenGL mitgeteilt, dass der Tiefentest durchzuführen ist:

```
// Tiefen Test aktivieren
glEnable(GL_DEPTH_TEST);
```

Ein Fallstrick besteht darin, dass OpenGL standardmäßig den Tiefentest mit der Funktion *less* durchführt, d.h. nur wenn der z-Wert des neues Fragmentes kleiner als der im Tiefenbuffer ist, wird das Fragment übernommen. Das zweimalige Zeichnen generiert im zweiten Durchlauf exakt die gleichen z-Werte. Demzufolge würde OpenGL darauf verzichten, im zweiten Durchlauf überhaupt irgendein Pixel zu zeichnen. Als Ausweg ändere ich die Vergleichfunktion auf *less-or-equal*:

```
// important ! default is GL_LESS
glDepthFunc(GL_LEQUAL);
```

Hiermit ist der gemeinsame Teil beider Effekte beendet und ich widme mich jetzt dem Code, der für „haloed wires“ zuständig ist. In OpenGL muss zuerst festgelegt werden, dass ein Drahtgittermodell gezeichnet wird:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

Die Linienstärke ist variabel und kann vom Benutzer über die Tasten 1 und 2 manipuliert werden, hierbei setzt aber die jeweilige OpenGL-Implementierung eine obere und untere Grenze:

```
glLineWidth(haloewidth_);
```

Ganz wichtig ist die Deaktivierung des Color-Buffers (der z-Buffer ist bereits für Schreibzugriffe freigeschalten worden):

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
```

Nun kann die Szene gezeichnet werden:

```
drawScene();
```

Nachdem die Tiefeninformationen im z-Buffer vorhanden sind, wird anschließend der Color-Buffer aktiviert. Zusätzlich setzt das Programm die Linienstärke auf 1:

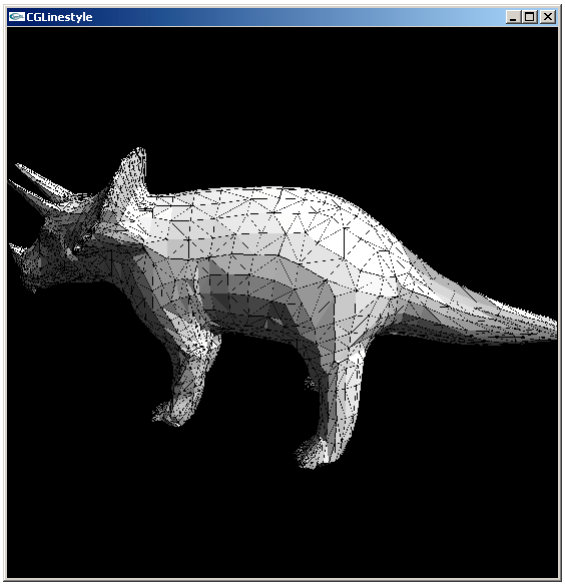
```
glLineWidth(1);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
```

Nochmaliges Zeichnen ergibt das komplette Bild:

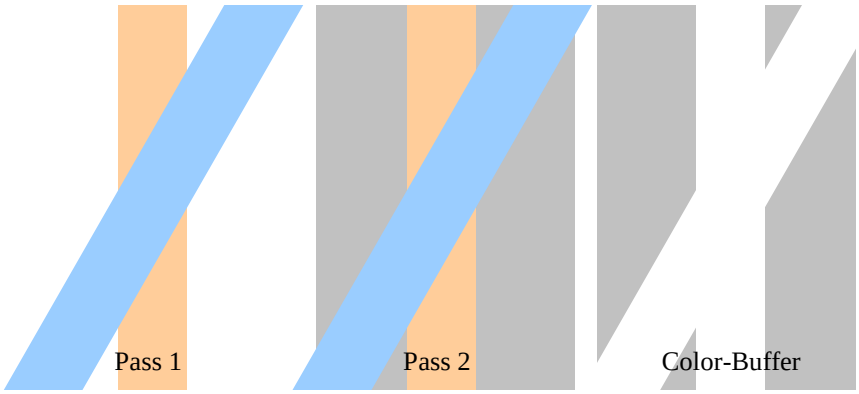
```
drawScene();
```

Segmented Surfaces

Dieser Effekt zeichnet *nicht* die Kanten zwischen den Polygonen und ist daher das Gegenstück zu haloed wires.



Der Algorithmus weist große Parallelen auf, denn erneut sind zwei Durchläufe notwendig:
Zuerst werden die Kanten gezeichnet, dabei sperrt man aber den Color-Buffer und ändert nur den z-Buffer. Anschließend aktiviert den Color-Buffer und zeichnet man die Polygone. Die Skizze ähnelt vom Aufbau her enorm der vorherigen:



Der erste Schritt entspricht ziemlich genau dem von haloed wires, es wird lediglich eine andere Variable für die Linienstärke benutzt:

```
glDepthFunc(GL_LESS);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

glLineWidth(segmentwidth_);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

drawScene();
```

Die Umschaltung in den Polygonmodus ist neu, die Colorbuffer-Aktivierung und das nochmalige Zeichnen der Szene dagegen nicht:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
drawScene();
```

Der komplette Sourcecode findet sich im Anhang.

Aufgabe 7

Die Aufgabenstellung erfordert, dass jedwedes Polygonmodell in Graustufen gezeichnet wird, wobei der Farbwert von der Entfernung zu einem Referenzpunkt abhängt. Es soll stets, unabhängig von der Entfernung des gesamten Modells, der nächste Punkt schwarz – im RGB-Farbraum: (0,0,0) – und der am weitesten entfernte weiss – also (1,1,1) – dargestellt werden.

In der Beispielanwendung sind die vom Programmrahmen vordefinierten Tasten unverändert übernommen worden:

Taste	Aktion
ESC	Programm beenden
Leertaste	Objekt drehen
+	in die Szene hineinzoomen
-	aus der Szene herauszoomen
c	Culling de-/aktivieren

Ich verwende zwei Schleifen, um die dazu notwendigen Informationen zu ermitteln. Im ersten Durchlauf bestimme ich die geringste und die größte Entfernung eines Punktes des Modells zu dem Referenzpunkt. Der Abstand selber entsteht durch die Anwendung von:

$$d = \sqrt{\left(x - x_{referenz}\right)^2 + \left(y - y_{referenz}\right)^2 + \left(z - z_{referenz}\right)^2}$$

Im RGB-Farbraum sind Graustufen durch die Eigenschaft $r = g = b$ gekennzeichnet. Die Abbildung des realen Abstandes zu dem Referenzpunkt in den Zahlenbereich $[0,1]$ geschieht über den Dreisatz:

$$Grauwert = \frac{d - d_{\min}}{d_{\max} - d_{\min}}$$

Um den minimalen und den maximalen Abstand zu bestimmen, gehe ich von unmöglichen Extremwerten aus. Schon der erste ermittelte Abstand verändert die zuständigen Variablen:

```
// normieren Sie Ihre Abstandsfunktion
double dMaxDistance = 0;
double dMinDistance = 999999999999;
for(unsigned int i=0; i<counter_; i++)
{
    double* pVector  = list_[i].rep();
    double dDistance = sqrt(((pVector[0] - ref_[0]) * (pVector[0] - ref_[0])) +
                            ((pVector[1] - ref_[1]) * (pVector[1] - ref_[1])) +
                            ((pVector[2] - ref_[2]) * (pVector[2] - ref_[2])));

    if (dDistance > dMaxDistance)
        dMaxDistance = dDistance;
    if (dDistance < dMinDistance)
        dMinDistance = dDistance;
}
```

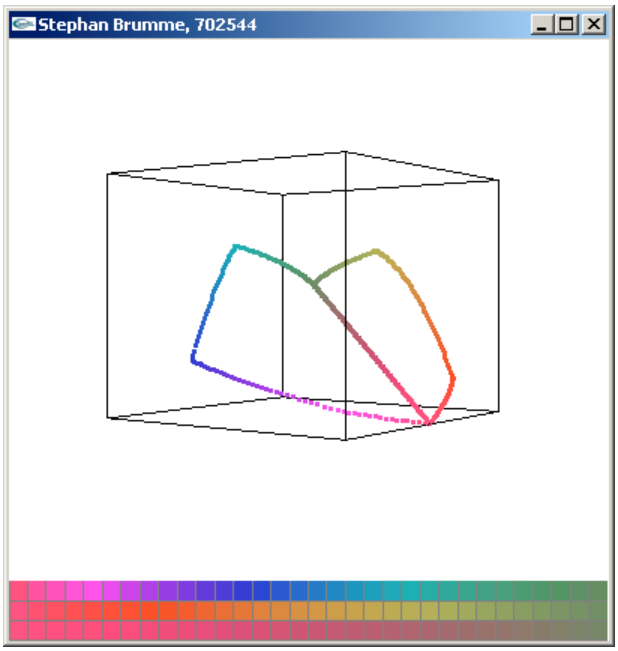
Leider vergisst das Programm bis auf den Minimal- und Maximalwert alle Abstandsinformationen. Natürlich könnte man diese in einem Array speichern, dies würde aber den Code aufblähen. Da alle Berechnungen nur einmal, nämlich beim Programmstart, durchgeführt werden, halte ich diese Vorgehensweise für erlaubt.

Das abschließende Zeichnen des Objektes ist ziemlich einfach: es wird erneut der Abstand ermittelt, dieser wird aber diesmal normiert (siehe obige Formel für *Grauwert*) und als Farbwert verwendet:

```
    glBegin(GL_TRIANGLES);
    for(i=0; i<counter_; i++) {

        double* pVector  = list_[i].rep();
        double dDistance = sqrt(((pVector[0] - ref_[0]) * (pVector[0] - ref_[0])) +
                                ((pVector[1] - ref_[1]) * (pVector[1] - ref_[1])) +
                                ((pVector[2] - ref_[2]) * (pVector[2] - ref_[2])));
        double dNormalized = (dDistance - dMinDistance) / (dMaxDistance - dMinDistance);

        glColor3d(dNormalized, dNormalized, dNormalized);
        glVertex3dv(list_[i].rep());

    }
    glEnd();

}
```

Aufgabe 8

Ich habe sowohl den Farbraum als auch seine dreidimensionale Darstellung in einem einzigen Fenster vereint. Der oberste Farbbalken entspricht dem RGB-Modell, in der Mitte findet sich HSV mit positiver *hue*-Orientierung, ganz unten HSV mit negativer *hue*-Orientierung:



Aufgrund der Zweifach-Nutzung des Fensters konnte die ursprünglich im Programmrahmen vorhandene Umschaltung entfallen. Alle anderen Features lassen sich mit diesen Tasten steuern:

Taste	Aktion
ESC	Programm beenden
Leertaste	Objekt drehen
a	Rotanteil der Startfarbe erhöhen
y	Rotanteil der Startfarbe verringern
s	Grünanteil der Startfarbe erhöhen
x	Grünanteil der Startfarbe verringern
d	Blauanteil der Startfarbe erhöhen
c	Blauanteil der Startfarbe verringern
f	Rotanteil der Zielfarbe erhöhen
v	Rotanteil der Zielfarbe verringern
g	Grünanteil der Zielfarbe erhöhen
b	Grünanteil der Zielfarbe verringern
h	Blauanteil der Zielfarbe erhöhen
n	Blauanteil der Zielfarbe verringern

Standardmäßig entspricht die Startfarbe (1, 0, ½) im RGB-Farbraum, was ein rosa Ton ist, die Zielfarbe ist dagegen ein Grau (0.45, 0.55, 0.4). Diese Werte müssten ziemlich genau mit dem Beispielphoto des Aufgabenblattes übereinstimmen.

Generell beruhen Interpolationen auf dem Schema:

$$f(a,b,t) = a \cdot (1-t) + b \cdot t$$

Im RGB-Farbraum kann man daher konsequent schreiben:

$$r'_t = r_{from} \cdot (1-t) + r_{to} \cdot t$$
$$g'_t = g_{from} \cdot (1-t) + g_{to} \cdot t$$
$$b'_t = b_{from} \cdot (1-t) + b_{to} \cdot t$$
$$\alpha'_t = \alpha_{from} \cdot (1-t) + \alpha_{to} \cdot t$$

bzw. als statische Funktion der Klasse `Color` (`dPercentage` entspricht *t*):

```
// interpolate two colors in RGBA space
Color Color::InterpolateRGB(const Color& from, const Color& to, double dPercentage)
{
    return Color(from[0]*(1-dPercentage) + to[0]*dPercentage,   // red
                 from[1]*(1-dPercentage) + to[1]*dPercentage,   // green
                 from[2]*(1-dPercentage) + to[2]*dPercentage,   // blue
                 from[3]*(1-dPercentage) + to[3]*dPercentage);  // alpha
}
```

Etwas komplizierter ist es im HSV-Farbraum, da *hue* ein Winkel ist, bei dem man den Umlaufsinn während der Interpolation bestimmen kann. Die drei anderen Attribute entsprechen wieder dem bekannten Schema:

$$saturation'_t = saturation_{from} \cdot (1-t) + saturation_{to} \cdot t$$
$$value'_t = value_{from} \cdot (1-t) + value_{to} \cdot t$$
$$\alpha'_t = \alpha_{from} \cdot (1-t) + \alpha_{to} \cdot t$$

Ein positiver Umlaufsinn ist daran zu erkennen, dass die Differenz $hue_{to} - hue_{from}$ positiv ist. Dann gilt:

$$\delta hue = \begin{cases} hue_{to} - hue_{from} & falls\ hue_{to} \geq hue_{from} \\ hue_{to} - hue_{from} + 360° & falls\ hue_{to} < hue_{from} \end{cases}$$
$$hue'_t = \delta hue \cdot t + hue_{from}$$

Entsprechend gilt für einen negativen Umlaufsinn:

$$\delta hue = \begin{cases} hue_{to} - hue_{from} & falls\ hue_{to} < hue_{from} \\ hue_{to} - hue_{from} - 360° & falls\ hue_{to} \geq hue_{from} \end{cases}$$
$$hue'_t = \delta hue \cdot t + hue_{from}$$

Der dazugehörige Code:

```
// interpolate two colors in RGBA space
Color Color::InterpolateHSV(const Color& from, const Color& to, double dPercentage, bool
bClockwise)
{
    double interpolate_hue;
    if (bClockwise)
    {
        // positive hue
        double delta_hue = to.hue() - from.hue();
        // MUST be positive
        if (delta_hue < 0.0)
            delta_hue += 360.0;
        // interpolate
        interpolate_hue = delta_hue*dPercentage + from.hue();
    }
    else
    {
        // negative hue
        double delta_hue = to.hue() - from.hue();
        // MUST be negative
        if (delta_hue >= 0.0)
            delta_hue -= 360.0;
        // interpolate
        interpolate_hue = delta_hue*dPercentage + from.hue();
    }

    // interpolate
    double interpolate_sat = (to.saturation() - from.saturation())*dPercentage +
from.saturation();
    double interpolate_val = (to.value()      - from.value())     *dPercentage + from.value();

    // get RGB from HSV
    Color result = hsv(interpolate_hue, interpolate_sat, interpolate_val);
    // interpolate alpha channel
    return Color(result[0], result[1], result[2], from[3]*(1-dPercentage) +
to[3]*dPercentage);
}
```

Bei der Umsetzung habe ich mich dazu entschlossen, sowohl den Farbverlauf als auch die dreidimensionale Veranschaulichung in einem Bild zu vereinen. Mit OpenGL kann man sehr bequem den zu zeichnenden Bildausschnitt bestimmen:

```
glViewport(0, 0, winWidth_-1, winHeight_/10);
```

Die drei Farbbalken nehmen somit 10% des im Fenster zur Verfügung stehenden Platzes ein. In einer Schleife erfolgt die Ausgabe von 32 Abstufungen (= Rasterbreite):

```
raster_.clear();
const int colors = raster_.width();
for (int x=0; x<colors; x++)
{
    // RGB
    raster_.setPixel(x,0,Color::InterpolateRGB(clrFrom, clrTo, x/(double)colors));
    // HSV +
    raster_.setPixel(x,1,Color::InterpolateHSV(clrFrom, clrTo, x/(double)colors, true));
    // HSV -
    raster_.setPixel(x,2,Color::InterpolateHSV(clrFrom, clrTo, x/(double)colors, false));
}
raster_.draw();
```

Danach wird erneut die Zeichenfläche umgeschalten, um diesmal den Würfel darstellen zu können:

```
glViewport(0, winHeight_/10, winWidth_-1, winHeight_*90);
```

Die räumliche Visualisierung der Farbverläufe in den jeweiligen Farbsystemen beruht darauf, dass ich den Rotanteil als x-, den Grünanteil als y-, und den Blauanteil als z-Koordinate interpretiere:

```
glPointSize(3);
glBegin(GL_POINTS);
for (x=0; x<SHOWNCOLORS; x++)
{
    Color clrInterpolate;

    // RGB
    clrInterpolate = Color::InterpolateRGB(clrFrom, clrTo, x/(double)SHOWNCOLORS);
    glColor3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);
    glVertex3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);

    // HSV +
    clrInterpolate = Color::InterpolateHSV(clrFrom, clrTo, x/(double)SHOWNCOLORS, true);
    glColor3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);
    glVertex3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);

    // HSV -
    clrInterpolate = Color::InterpolateHSV(clrFrom, clrTo, x/(double)SHOWNCOLORS, false);
    glColor3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);
    glVertex3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);
}
glEnd();
```

Anhang

Quellcode Aufgabe 6

Nur cglinestyle.h und cglinestyle.cpp mussten verändert werden:

cglinestyle.h:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 2
//

#ifndef CG_LINESTYLE_H
#define CG_LINESTYLE_H

#include "cgapplication.h"

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

    // Ueberschreibe alle diese Ereignisse:
    virtual void onInit();
    virtual void onDraw();
    virtual void onIdle();
    virtual void onKey(unsigned char key);
    virtual void onSize(unsigned int newWidth, unsigned int newHeight);

    // value Methode
    unsigned char value(int x, int z) const;

private:
    enum { HALOED, SEGMENTED };

    void drawScene();

    double haloewidth_;
    double segmentwidth_;
    int width_;
    int height_;
    double zoom_;
    bool run_;
    bool culling_;
    bool lighting_;
    int mode_;
};

#endif
```

cglinestlye.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 2
//

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

CGLinestyle::CGLinestyle() {
    run_ = false;
    zoom_ = 0.15;
    culling_ = false;
    lighting_ = true;
    mode_ = SEGMENTED;
    haloewidth_ = 5.0;
    segmentwidth_ = 1.0;
}

CGLinestyle::~CGLinestyle( ) {
}

void CGLinestyle::drawScene() {

    glPushMatrix();
    glScaled(zoom_, zoom_, zoom_);

    static GLuint cache = 0;
    if (cache == 0) {
        cache = glGenLists(1);
        glNewList(cache, GL_COMPILE_AND_EXECUTE);
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();

        //ifstream s("small.txt");
        //ifstream s("stegarosaurus.txt");
        ifstream s("triceratops.txt");
        char buf[100];

        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();
    } else {
        glCallList(cache);
    }

    glPopMatrix();
}

void CGLinestyle::onInit() {
    // OpenGL Lichtquelle
    static GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};  /* diffuse light. */
    static GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};  /* Infinite light location. */
```

```
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    // automatische Normalisierung
    glEnable(GL_NORMALIZE);

    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);

    // Tiefen Test aktivieren
    glEnable(GL_DEPTH_TEST);

    // Smooth Schattierung aktivieren
    glShadeModel(GL_SMOOTH);

    // Projection
    glMatrixMode(GL_PROJECTION);
    gluPerspective(60.0,1.0,2,2000.0);

    // 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(0.0,0.0,0.0,1.0);
}

void CGLinestyle::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 CGLinestyle::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 'c': { culling_ = !culling_; break; }
    case 'l': { lighting_ = !lighting_; break; }
    case 's': { mode_ = SEGMENTED; break; }
    case 'h': { mode_ = HALOED; break; }
    case '1': { haloewidth_*= 0.9; break; }
    case '2': { haloewidth_*= 1.1; break; }
    case '3': { segmentwidth_*= 0.9; break; }
    case '4': { segmentwidth_*= 1.1; break; }
    }
    onDraw();
}

void CGLinestyle::onIdle() {
    if (run_) {
        glRotatef(1,0.1,1,0.2);
        onDraw();
    }
}

void CGLinestyle::onDraw() {
    // clear buffers
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
    glDepthMask(GL_TRUE);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


    if (culling_) glEnable(GL_CULL_FACE);
    if (!lighting_) glDisable(GL_LIGHTING);
```

```
    if (mode_ == SEGMENTED)
    {
        // SEGMENTED

        // important ! default is GL_LESS
        glDepthFunc(GL_LESS);

        // wireframe
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

        glLineWidth(segmentwidth_);
        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

        drawScene();

        // filled
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

        drawScene();
    }
    else
    {
        // HALOED

        // important ! default is GL_LESS
        glDepthFunc(GL_LEQUAL);

        // wireframe I
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

        glLineWidth(haloewidth_);
        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

        drawScene();

        // wireframe II
        glLineWidth(1);
        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

        drawScene();
    }

    // clean up
    glEnable(GL_LIGHTING);
    glDisable(GL_CULL_FACE);

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

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

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "+          in die Szene hineinzoomen" << endl
         << "-          aus der Szene herauszoomen" << endl
         << "c          Culling de-/aktivieren" << endl
         << "l          Beleuchtung de-/aktivieren" << endl
         << "s          segmented surfaces" << endl
         << "h          haloed wires" << endl
         << "1          halo verringern" << endl
         << "2          halo verstärken" << endl
         << "3          Abstand der Segmente verringern" << endl
         << "4          Abstand der Segmente vergrößern" << endl;

    // Starte die Beispiel-Anwendung:
    sample.start("Stephan Brumme, 702544", true, 550, 550);
    return(0);
}
```

Quellcode Aufgabe 7

Nur cginterpolation.cpp ist von Interesse:

cginterpolation.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 3
//

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

CGInterpolation::CGInterpolation() {
    run_ = false;
    zoom_ = 0.15;
    culling_ = false;

    max_ = 100;                    // maximale Laenge des Arrays
    counter_ = 0;                  // Anzahl der Vertices
    list_ = new Vector[max_];      // Array mit Vertices
    ref_ = Vector(8.0, -1.0, 1.0); // Referenzpunkt

    loadGeometry();                // Import der Geometriedaten
}

CGInterpolation::~CGInterpolation() {
}

void CGInterpolation::drawScene() {

    glPushMatrix();
    glScaled(zoom_, zoom_, zoom_);

    static GLuint cache = 0;
    if (cache == 0) {
        cache = glGenLists(1); // Display-Liste
        glNewList(cache, GL_COMPILE_AND_EXECUTE); // Display-Liste

        glColor3f(0.5, 0.4, 0.4);
        drawObject(ref_);

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glTranslatef(ref_[0],ref_[1],ref_[2]);

        glColor3f(0.9, 0.1, 0.2);
        glutSolidSphere(0.1,16,16);
        glPopMatrix();

        glEndList(); // Display-Liste
    } else {
        glCallList(cache); // Display-Liste
    }

    glPopMatrix();
}

void CGInterpolation::drawObject(const Vector& ref) {
    // draw geometry

    // normieren Sie Ihre Abstandsfunktion
    double dMaxDistance = 0;
    double dMinDistance = 999999999999;
    for(unsigned int i=0; i<counter_; i++)
    {
        double* pVector  = list_[i].rep();
```

```
        double dDistance = sqrt(((pVector[0] - ref_[0]) * (pVector[0] - ref_[0])) +
                                ((pVector[1] - ref_[1]) * (pVector[1] - ref_[1])) +
                                ((pVector[2] - ref_[2]) * (pVector[2] - ref_[2])));

        if (dDistance > dMaxDistance)
            dMaxDistance = dDistance;
        if (dDistance < dMinDistance)
            dMinDistance = dDistance;
    }


    // Zeichnen Sie die Dreiecke mit Farbwerten an den Vertices.
    glBegin(GL_TRIANGLES);
    for(i=0; i<counter_; i++) {

        double* pVector  = list_[i].rep();
        double dDistance = sqrt(((pVector[0] - ref_[0]) * (pVector[0] - ref_[0])) +
                                ((pVector[1] - ref_[1]) * (pVector[1] - ref_[1])) +
                                ((pVector[2] - ref_[2]) * (pVector[2] - ref_[2])));
        double dNormalized = (dDistance - dMinDistance) / (dMaxDistance - dMinDistance);

        glColor3d(dNormalized, dNormalized, dNormalized);
        glVertex3dv(list_[i].rep());

    }
    glEnd();

}

void CGInterpolation::loadGeometry() {
    //ifstream s("small.txt");
    ifstream s("stegarosaurus.txt");
    //ifstream s("triceratops.txt");

    char buf[100];

    while (!s.eof()) {
        s >> buf;

        while (s.good()) {
            Vector v0;
            Vector v1;
            Vector v2;
            s >> v0 >> v1 >> v2;
            if(counter_+3 >= max_) {
                max_*=2;
                Vector* newList = new Vector[max_];
                for(unsigned int i=0; i<counter_; i++) {
                    newList[i] = list_[i];
                }
                delete list_;
                list_ = newList;
            }
            list_[counter_] = v0; counter_++;
            list_[counter_] = v1; counter_++;
            list_[counter_] = v2; counter_++;
        }
        if (s.rdstate() & ios::failbit) {
            s.clear(s.rdstate() & ~ios::failbit);
        }
    }
}

void CGInterpolation::onInit() {

    // automatische Normalisierung
    glEnable(GL_NORMALIZE);

    // Tiefen Test aktivieren
    glEnable(GL_DEPTH_TEST);

    // Smooth Schattierung aktivieren
    glShadeModel(GL_SMOOTH);

    // Projection
    glMatrixMode(GL_PROJECTION);
```

```
    gluPerspective(60.0,1.0,2,100.0);

    // 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);
}

void CGInterpolation::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 CGInterpolation::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 'c': { culling_ = !culling_; break; }
    }
    onDraw();
}

void CGInterpolation::onIdle() {
    if (run_) {
        glRotatef(1, 0.0, 1.0, 0.0);
        onDraw();
    }
}

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

    if (culling_) glEnable(GL_CULL_FACE);

    drawScene();

    glDisable(GL_CULL_FACE);

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

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

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "+          in die Szene hineinzoomen" << endl
         << "-          aus der Szene herauszoomen" << endl
         << "c          Culling de-/aktivieren" << endl;

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

Quellcode Aufgabe 8

Ich bearbeitete nur die Klassen `Color` und CGHSV`Colors`, den Rest übernahm ich unverändert.
Die Klasse `Color` wurde um Methoden zur Interpolation im RGB- bzw. HSV-Farbraum erweitert:

color.h:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/2002
//
// Rahmenprogramm fuer Aufgabenzettel 3
//
// Autoren: Florian Kirsch (kirsch@hpi.uni-potsdam.de)
//          Marc Nienhaus (nienhaus@hpi.uni-potsdam.de)
//          Juergen Doellner (doellner@hpi.uni-potsdam.de)
//

#ifndef CG_COLOR_H
#define CG_COLOR_H

#include <iostream.h>

class Color {
public:
    Color(double grey=0);
    Color(double red, double green, double blue, double alpha = 1.0);
        /* A color is specified by the red, green, and blue coefficients
          and the alpha (e.g., transmission) coefficient. All coefficients
          are clamped to the double interval [0.0, 1.0]. */

    bool operator==(const Color&) const;
    bool operator!=(const Color&) const;

    Color operator+(const Color& col) const;
    Color operator-(const Color& col) const;
    Color operator*(double coef) const;
        /* These operators transform the red, green, and blue components.
          They do *not* apply to the alpha value. The result of each
          operation is clamped to [0.0, 1.0]. */

    double operator[](int i) const;
    double& operator[](int i);
        /* The four components are indexed by R=[0], G=[1], B=[2], and A=[3].
          The index is checked for out of range errors. */

    static Color hsv(double hue, double saturation, double value, double alpha=1.0);
    double hue() const;
    double saturation() const;
    double value() const;
        /* Provide the conversion between the RGB and HSV color models.
          `hsv' can specify achromatic (gray-scale) colors by
          setting `s' to 0.0 and `v' to the gray coefficient between [0,1].
          The `hue' factor must be specified in degrees in the range [-360,360].
          If hue is smaller 0, 360 degrees are added before the conversion
          is invoked.
          Both saturation and value must be in the range [0,1]. */

    static Color InterpolateRGB(const Color& from, const Color& to, double dPercentage);
    static Color InterpolateHSV(const Color& from, const Color& to, double dPercentage, bool
bClockwise = true);

    friend ostream& operator<<(ostream&, const Color&);
    friend istream& operator>>(istream&, Color&);

private:
    double rgba_[4];
};

#endif // CG_COLOR_H
```

color.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/2002
//
// Rahmenprogramm fuer Aufgabenzettel 3
//
// Autoren: Florian Kirsch (kirsch@hpi.uni-potsdam.de)
//          Marc Nienhaus (nienhaus@hpi.uni-potsdam.de)
//          Juergen Doellner (doellner@hpi.uni-potsdam.de)
//

#include "color.h"
#include <assert.h>
#include <iostream.h>

template<class T>
static inline const T& Min(const T& a, const T& b) {
    return((a < b) ? a : b);
}

template<class T>
static inline const T& Max(const T& a, const T& b) {
    return((a > b) ? a : b);
}

inline double clamp(double v) {
    return v>1 ? 1.0 : (v<0 ? 0 : v);
}

Color::Color(double grey) {
    rgba_[0] = rgba_[1] = rgba_[2] = grey;
    rgba_[3] = 1.0;
}

Color::Color(double r, double g, double b, double a) {
    rgba_[0] = r;
    rgba_[1] = g;
    rgba_[2] = b;
    rgba_[3] = a;
}

double Color::operator[](int i) const {
    assert(i>=0 && i<=3);
    return rgba_[i];
}

double& Color::operator[](int i) {
    assert(i>=0 && i<=3);
    return rgba_[i];
}

bool Color::operator==(const Color& mc) const {
    return rgba_[0]==mc.rgba_[0] &&
           rgba_[1]==mc.rgba_[1] &&
           rgba_[2]==mc.rgba_[2] &&
           rgba_[3]==mc.rgba_[3];
}

bool Color::operator!=(const Color& mc) const {
    return rgba_[0]!=mc.rgba_[0] ||
           rgba_[1]!=mc.rgba_[1] ||
           rgba_[2]!=mc.rgba_[2] ||
           rgba_[3]!=mc.rgba_[3];
}

Color Color::operator+(const Color& mc) const {
    double r = clamp(rgba_[0]+mc.rgba_[0]);
    double g = clamp(rgba_[1]+mc.rgba_[1]);
    double b = clamp(rgba_[2]+mc.rgba_[2]);
    double a = rgba_[3];
    return Color(r,g,b,a);
}
```

```
Color Color::operator-(const Color& mc) const {
    double r = clamp(rgba_[0]-mc.rgba_[0]);
    double g = clamp(rgba_[1]-mc.rgba_[1]);
    double b = clamp(rgba_[2]-mc.rgba_[2]);
    double a = rgba_[3];
    return Color(r,g,b,a);
}

Color Color::operator*(double d) const {
    double r = clamp(rgba_[0]*d);
    double g = clamp(rgba_[1]*d);
    double b = clamp(rgba_[2]*d);
    double a = rgba_[3];
    return Color(r,g,b,a);
}

//
// RGB - HSV
//

Color Color::hsv(double h, double s, double v, double alpha) {
    // normalize hue angle
    while (h<   0.0) h += 360.0;
    while (h>=360.0) h -= 360.0;

    assert(h>=-360.0 && h<=360.0);
    assert(v>=0.0 && v<=1.0);
    assert(s>=0.0 && s<=1.0);

    double R, G, B;
    if (s==0.0) {
        // color on black-white center line
        // achromatic color!
        R = v;
        G = v;
        B = v;
    }  else {
        // chromatic color
        if (h==360.0) h = 0.0;
        if (h<0.0) h+=360.0;
        h = h/60.0;
        int i = (int)h;
        double f = h - i;
        double p = v*(1.0-s);
        double q = v*(1.0-(s*f));
        double t = v*(1.0-(s*(1.0-f)));
        switch(i) {
        case 0: R = v; G = t; B = p; break;
        case 1: R = q; G = v; B = p; break;
        case 2: R = p; G = v; B = t; break;
        case 3: R = p; G = q; B = v; break;
        case 4: R = t; G = p; B = v; break;
        case 5: R = v; G = p; B = q; break;
        default: ;
        }
    }
    return Color(R,G,B,alpha);
}

double Color::value() const {
    return Max(rgba_[0], Max(rgba_[1], rgba_[2]));
}

double Color::saturation() const {
    double v = Max(rgba_[0], Max(rgba_[1], rgba_[2]));
    if(v!=0.0) {
        double m = Min(rgba_[0], Min(rgba_[1], rgba_[2]));
        return (v-m)/v;
    } else {
        return 0.0;
    }
}

double Color::hue() const {
    double s = saturation();
```

```
    if(s==0.0) return 0.0; // hue is undefined

    double mmax = Max(rgba_[0], Max(rgba_[1], rgba_[2]));
    double mmin = Min(rgba_[0], Min(rgba_[1], rgba_[2]));
    double delta = mmax - mmin;

    double h;
    if(rgba_[0]==mmax)
        h = 0.0 + (rgba_[1]-rgba_[2])/delta;
    else if(rgba_[1]==mmax)
        h = 2.0 + (rgba_[2]-rgba_[0])/delta;
    else
        h = 4.0 + (rgba_[0]-rgba_[1])/delta;

    h = h * 60.0;
    if(h<0.0) h += 360.0;

    return h;
}


// interpolate two colors in RGBA space
Color Color::InterpolateRGB(const Color& from, const Color& to, double dPercentage)
{
    return Color(from[0]*(1-dPercentage) + to[0]*dPercentage,   // red
                 from[1]*(1-dPercentage) + to[1]*dPercentage,   // green
                 from[2]*(1-dPercentage) + to[2]*dPercentage,   // blue
                 from[3]*(1-dPercentage) + to[3]*dPercentage);  // alpha
}


// interpolate two colors in RGBA space
Color Color::InterpolateHSV(const Color& from, const Color& to, double dPercentage, bool
bClockwise)
{
    double interpolate_hue;
    if (bClockwise)
    {
        // positive hue
        double delta_hue = to.hue() - from.hue();
        // MUST be positive
        if (delta_hue < 0.0)
            delta_hue += 360.0;
        // interpolate
        interpolate_hue = delta_hue*dPercentage + from.hue();
    }
    else
    {
        // negative hue
        double delta_hue = to.hue() - from.hue();
        // MUST be negative
        if (delta_hue >= 0.0)
            delta_hue -= 360.0;
        // interpolate
        interpolate_hue = delta_hue*dPercentage + from.hue();
    }

    // interpolate
    double interpolate_sat = (to.saturation() - from.saturation())*dPercentage +
from.saturation();
    double interpolate_val = (to.value()      - from.value())     *dPercentage + from.value();

    // get RGB from HSV
    Color result = hsv(interpolate_hue, interpolate_sat, interpolate_val);
    // interpolate alpha channel
    return Color(result[0], result[1], result[2], from[3]*(1-dPercentage) +
to[3]*dPercentage);
}


ostream& operator<<(ostream& ostr, const Color& mc) {
    ostr << mc[0] << " " << mc[1] << " " << mc[2] << " " << mc[3] << " ";
    return ostr;
}

istream& operator>>(istream& s, Color& mc) {
```

```
    char ch;

    s >> ch;
    if(ch ==‘{‘) {  // format "{ r g b a }"
        s >> mc[0] >> mc[1] >> mc[2] >> mc[3] >> ch;
        if (ch != ‘}‘) {
            s.clear(ios::badbit);
        }
    } else { // format "r g b a"
        s.putback(ch);
        s >> mc[0] >> mc[1] >> mc[2] >> mc[3];
    }

    return s;
}
```

Im Programmrahmen fügte ich die Anzeige des Farbverlaufes bzw. seine Abbildung in den drei-dimensionalen Raum ein:

cghsvcolors.h:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/2002
//
// Rahmenprogramm fuer Aufgabenzettel 3
//
// Autoren: Florian Kirsch (kirsch@hpi.uni-potsdam.de)
//          Marc Nienhaus (nienhaus@hpi.uni-potsdam.de)
//          Juergen Doellner (doellner@hpi.uni-potsdam.de)
//

#ifndef CG_HSVCOLORS_H
#define CG_HSVCOLORS_H

#include "cgapplication.h"
#include "cgraster.h"

class CGHSVColors : public CGApplication {
public:
    CGHSVColors(int width, int height);

    virtual void onInit();
    virtual void onDraw();
    virtual void onSize(unsigned int newWidth,unsigned  int newHeight);
    virtual void onKey(unsigned char key);
    virtual void onIdle();

    virtual void onButton(MouseButton button, int x, int y);

    enum { SHOWNCOLORS = 128 };

private:

    enum MODE { DrawGrid, DrawCube };
    MODE mode_;

    // interne Raster-Klasse
    CGRaster raster_;

    // Fenstergroesse speichern
    int winWidth_;
    int winHeight_;
    int hSubWindow;

    Color clrFrom;
    Color clrTo;

    bool run_;
    double angle_;

};

#endif // CG_HSVCOLORS_H
```

cghsvcolors.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/2002
//
// Rahmenprogramm fuer Aufgabenzettel 3
//
// Autoren: Florian Kirsch (kirsch@hpi.uni-potsdam.de)
//          Marc Nienhaus (nienhaus@hpi.uni-potsdam.de)
//          Juergen Doellner (doellner@hpi.uni-potsdam.de)
//

#include "cghsvcolors.h"


CGHSVColors::CGHSVColors(int width, int height) : raster_(width,height) {
    mode_ = DrawGrid;
    run_ = false;
    angle_ = 30;

    clrFrom = Color(1.0, 0.0, 0.5);
    clrTo   = Color(0.45, 0.55, 0.4);
}

void CGHSVColors::onInit() {
    glClearColor(1, 1, 1, 1);
    // anti-aliased points
//  glHint(GL_POINT_SMOOTH_HINT, GL_FASTEST);
//  glEnable(GL_POINT_SMOOTH);
//  glEnable(GL_BLEND);
//  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}

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

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

//    if (mode_ == DrawGrid) {

    // color banner
    glViewport(0, 0, winWidth_-1, winHeight_/10);

        glDisable(GL_DEPTH_TEST);

        glMatrixMode(GL_PROJECTION);
        gluOrtho2D(0, raster_.width(), 0, raster_.height());
        glMatrixMode(GL_MODELVIEW);

        // Hier 2D-Raster zeichnen
        raster_.clear();
        const int colors = raster_.width();
        for (int x=0; x<colors; x++)
        {
            // RGB
            raster_.setPixel(x,0,Color::InterpolateRGB(clrFrom, clrTo, x/(double)colors));
            // HSV +
            raster_.setPixel(x,1,Color::InterpolateHSV(clrFrom, clrTo, x/(double)colors,
true));
            // HSV -
            raster_.setPixel(x,2,Color::InterpolateHSV(clrFrom, clrTo, x/(double)colors,
false));
        }
        raster_.draw();

//    } else if (mode_ == DrawCube) {

    // cube
    glViewport(0, winHeight_/10, winWidth_-1, winHeight_*0.9);

        glEnable(GL_DEPTH_TEST);
```

```
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(30.0,1.0,2,2000.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        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

        glRotatef(angle_, 0.0, 1.0, 0.0);
        glColor4d(0.0, 0.0, 0.0, 1.0);
        glutWireCube(1.0);
        glTranslatef(-0.5, -0.5, -0.5);

        // Hier 3D-Punkte zeichnen

        glPointSize(3);
        glBegin(GL_POINTS);
        for (x=0; x<SHOWNCOLORS; x++)
        {
            Color clrInterpolate;

            // RGB
            clrInterpolate = Color::InterpolateRGB(clrFrom, clrTo, x/(double)SHOWNCOLORS);
            glColor3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);
            glVertex3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);

            // HSV +
            clrInterpolate = Color::InterpolateHSV(clrFrom, clrTo, x/(double)SHOWNCOLORS,
true);
            glColor3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);
            glVertex3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);

            // HSV -
            clrInterpolate = Color::InterpolateHSV(clrFrom, clrTo, x/(double)SHOWNCOLORS,
false);
            glColor3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);
            glVertex3d(clrInterpolate[0], clrInterpolate[1], clrInterpolate[2]);
        }
        glEnd();

//    }

    swapBuffers();
}

void CGHSVColors::onSize(unsigned int newWidth,unsigned int newHeight) {
    winWidth_ = newWidth;
    winHeight_ = newHeight;
    glViewport(0, 0, newWidth - 1, newHeight - 1);
}

void CGHSVColors::onButton(MouseButton button, int x, int y) {
    onDraw();
}

void CGHSVColors::onIdle() {
    if (run_) {
        angle_ += 1.0;
        onDraw();
    }
}

void CGHSVColors::onKey(unsigned char key) {
    switch (key) {
    case 27: { exit(0); break; }
//    case 'g': mode_ = DrawGrid; break;
//    case 'c': mode_ = DrawCube; break;

        // change "from" color
    case 'a': if (clrFrom[0] <= 0.95) clrFrom[0] += 0.05; break;
    case 'y': if (clrFrom[0] >= 0.05) clrFrom[0] -= 0.05; break;
    case 's': if (clrFrom[1] <= 0.95) clrFrom[1] += 0.05; break;
    case 'x': if (clrFrom[1] >= 0.05) clrFrom[1] -= 0.05; break;
    case 'd': if (clrFrom[2] <= 0.95) clrFrom[2] += 0.05; break;
```

```
    case 'c': if (clrFrom[2] >= 0.05) clrFrom[2] -= 0.05; break;
        // change "to" color
    case 'f': if (clrTo  [0] <= 0.95) clrTo  [0] += 0.05; break;
    case 'v': if (clrTo  [0] >= 0.05) clrTo  [0] -= 0.05; break;
    case 'g': if (clrTo  [1] <= 0.95) clrTo  [1] += 0.05; break;
    case 'b': if (clrTo  [1] >= 0.05) clrTo  [1] -= 0.05; break;
    case 'h': if (clrTo  [2] <= 0.95) clrTo  [2] += 0.05; break;
    case 'n': if (clrTo  [2] >= 0.05) clrTo  [2] -= 0.05; break;

    case ' ': run_ = !run_; break;
    }
    onDraw();
}

int main(int argc, char* argv[]) {
    CGHSVColors hsvcolors(32, 3);

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "a          Rotanteil  der Startfarbe erhöhen" << endl
         << "y          Rotanteil  der Startfarbe verringern" << endl
         << "s          Grünanteil der Startfarbe erhöhen" << endl
         << "x          Grünanteil der Startfarbe verringern" << endl
         << "d          Blauanteil der Startfarbe erhöhen" << endl
         << "c          Blauanteil der Startfarbe verringern" << endl
         << "f          Rotanteil  der Zielfarbe  erhöhen" << endl
         << "v          Rotanteil  der Zielfarbe  verringern" << endl
         << "g          Grünanteil der Zielfarbe  erhöhen" << endl
         << "b          Grünanteil der Zielfarbe  verringern" << endl
         << "h          Blauanteil der Zielfarbe  erhöhen" << endl
         << "n          Blauanteil der Zielfarbe  verringern" << endl;

    hsvcolors.start("Stephan Brumme, 702544", true, 400, 400);

    return(0);
}
```