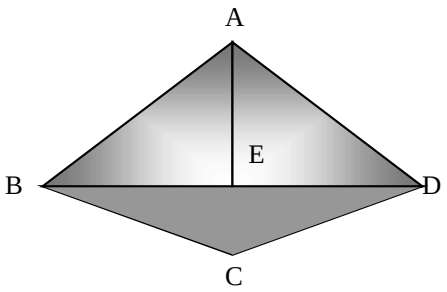
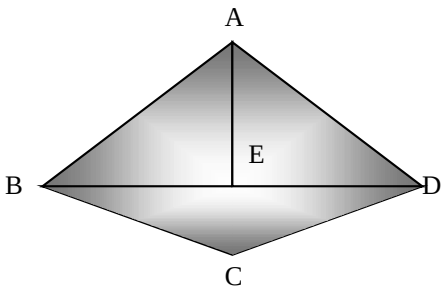
Aufgabe 12: Phong- und Gouraudschattierung

a) Der Hauptunterschied zwischen der Phong- und der Gouraud-Schattierung besteht darin, dass nach Gouraud die Lichtintensitäten nur an den Ecken bzw. Kanten der Objekte berechnet werden. Alle Pixelintensitäten des Fragmentes zwischen diesen Begrenzungen unterliegen lediglich einer (linearen) Interpolation. Da Phong aber für jedes Pixel die Lichtintensität explizit ermittelt, weil die Normalen zwischen den Begrenzungen interpoliert werden, können Glanzlichter, die inmitten eines Objektes auftreten, korrekt dargestellt werden. Zwei Beispiele sollen dies untermauern:

Zuerst der Fall, dass durch eine unpassende Tessellation eine T-Verbindung entstandt ist: Die Lichtintensität an den Punkten A bis D der Objekthülle sei identisch, in der Mitte, bei E, sei ein Spotlight. In der linken Abbildung, die mit der Gouraud-Schattierung erzeugt wurde, ist das untere Dreieck uni grau gefärbt, da die Interpolation der Eckwerte B, C und D das Glanzlicht in der Mitte nicht berücksichtigen kann, weil B, C und D gleich groß sind und jeder zwischen ihnen gemittelte Wert stets identisch ist. Die Phong-Schattierung in der rechten Abbildung zeigt dagegen das erwartete Resultat, da sich die Normalen der Eckpunkte alle voneinander unterscheiden, demzufolge auch die interpolierten Normalen der eingeschlossenen Flächen, woraus realitätsnahe Lichtintensitäten folgen:

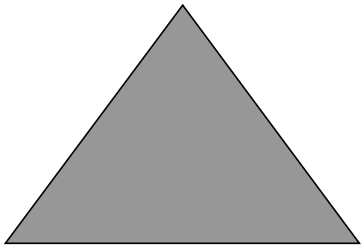


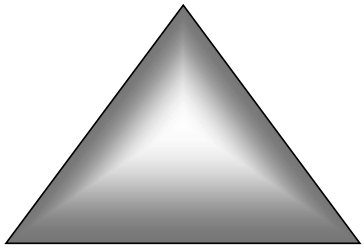
Gouraud-Schattierung



Phong-Schattierung

Ein weiterer Fall, der die visuelle Überlegenheit der Phong-Schattierung zeigt, ist ein sehr großes Objekt, in dessen Mitte ein Glanzlicht ist. Bei der Gouraud-Schattierung werden erneut nur die Lichtintensitäten an den Ecken gemessen, die Interpolation liefert im schlechtesten Fall eine einfarbige Fläche. Wenn die Phong-Schattierung die Normalen interpoliert, kann die Lichtintensität einigermaßen realistisch bestimmt werden.


Gouraud-Schattierung


Phong-Schattierung

Ich möchte noch anmerken, dass selbst die Phong-Schattierung nicht perfekt ist. Sie geht von planaren Flächen aus, was z.B. bei tessellierten Kugeln nicht der Fall ist. Wie man auch mit der schnelleren Gouraud-Schattierung ansprechende Ergebnisse erzielen kann, wie im nächsten Aufgabenteil erklärt.

Anmerkung: Die abgebildeten Grafiken entstammen einem Zeichenprogramm und nicht einer realen Umsetzung der Verfahren. Eventuell Farbverlaufsabweichungen sind daher vorprogrammiert aber unvermeidlich.

b) Ich habe soeben gezeigt, dass die Phong-Schattierung von der visuellen Qualität her das überlegene Modell ist. Leider ist die Ermittlung der Lichtintensität über interpolierte Normalen sehr aufwändig - wenn diese für jedes Pixel durchgeführt werden muss, dann hängt die Laufzeit von der Rastergröße des Bildschirmfensters ab. Die Gouraud- und die Flat-Schattierung haben eine Komplexität, die linear mit der Anzahl der Eckpunkte (Gouraud) bzw. Flächen (Flat) wächst, für wenige, aber große Polygone ergibt sich daraus ein erheblicher Geschwindigkeitszuwachs.

Viele Renderingsysteme setzen aus diesem Grunde die Gouraud-Schattierung (oder das einfache Flat-Shading) ein, da es sehr einfach zu implementieren und äußerst schnell ist. Ist man darauf angewiesen, eine sehr hohe Qualität zu erzielen, so kann man seine Objekte derart tessellieren, dass die Polygone in etwa nur 1 Pixel groß werden. Weil dann jeder Bildschirmpixel gleichzeitig ein Eckpunkt ist und man für diese die Lichtintensität berechnet, hat man indirekt den gleichen Effekt wie bei der Phong-Schattierung erzielt. Natürlich liegt dann auch der Rechenaufwand in der gleichen Dimension.

Aufgabe 13: Die Rendering-Gleichung von Kajiya

Die von Kajiya auf der SIGGRAPH 1986 vorgestellte „rendering equation“ hat die Form:

$$I(x,x') = g(x,x') \cdot \left[\varepsilon(x,x') + \int_S \rho(x,x',x'') \cdot I(x',x'')\, dx'' \right]$$

Die Geometrieterme ergeben sich aus

$$g(x,x') = \begin{cases} 0 & \textit{wenn } x \textit{ und } x' \textit{ voneinander verdeckt sind} \\ 1/r^2 & \textit{sonst} \end{cases}$$

$$r = \|x - x'\|$$

und haben dann in etwa die Größenordnung:

- $g(A,D) = 0$, da Hindernisse zwischen der direkten Verbindung von A und D stehen (der Spiegel und die aufrechte Fläche)
- $g(B,C) = g(C,B)$, da g nur vom Abstand der Punkte B und C abhängt und r deshalb gleich ist
- $g(B,C) > g(A,B) > g(A,C) > g(A,D)$, da ich diese Abstandsverhältnisse in der Grafik *vermute*
- Alle Terme dürften, je nach Maßstab der Grafik, relativ klein sein und wahrscheinlich im Bereich $[0,1]$ befinden

Emittiertes Licht fließt über den ε-Term in die Gleichung ein:

$$\varepsilon(x,x') = \begin{cases} 0 & \textit{wenn } x \textit{ und } x' \textit{ voneinander verdeckt sind oder} \\ & x' \textit{ kein Licht emittiert} \\ > 0 & \textit{sonst} \end{cases}$$

Besonders die Bedingung „x‘ emittiert kein Licht“ ist interessant, da A die einzige Lichtquelle in der Szene ist. Es ergibt sich:

$$\varepsilon(A,B) = \varepsilon(A,C) = \varepsilon(A,D) = \varepsilon(B,C) = \varepsilon(C,B) = 0$$

Da ich aber vermute, dass Buchstabendreher in der Aufgabenstellung enthalten sind, gilt für die Paare (A,B) bzw. (A,C):

$$\varepsilon(B,A) \geq \varepsilon(C,A) > 0$$

Zwischen A und D liegt ein Hindernis und weder B noch C emittieren Licht:

$$\varepsilon(D,A) = \varepsilon(B,C) = \varepsilon(C,B) = 0$$

Der *scattering term* beschäftigt sich mit reflektiertem Licht:

$$\rho(x,x',x'') = \begin{cases} 0 & \textit{wenn } x'' \textit{ weder emittiert noch reflektiert} \\ & \textit{oder } x' \textit{ nicht reflektiert} \\ > 0 & \textit{sonst} \end{cases}$$

Alle Terme enthalten für *x‘‘* die Lichtquelle A. Weil die verwendeten Materialien über eine gewisse Reflektion verfügen (der Spiegel sehr stark, die Wände eher schwach), sind alle Terme größer Null. Sowohl B als auch B‘ liegen auf der gleichen Fläche, sie werden daher vermutlich die gleichen diffusen Eigenschaften besitzen, der einzige Unterschied zwischen ihnen liegt im Einfallswinkel der Lichtquelle A.

$$\rho(C,B,A) \approx \rho(C,B',A) \approx \rho(C',B,A) \approx \rho(C',B',A) = \textit{recht klein}$$

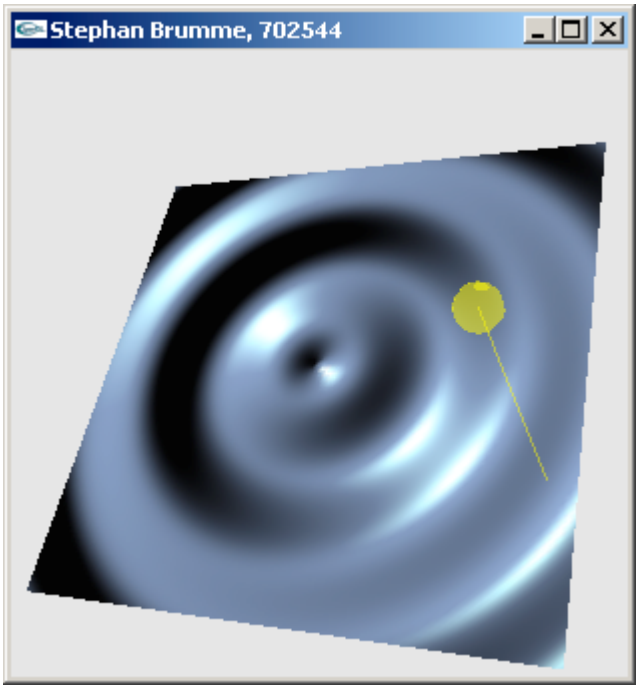
Im Bild sind die Terme dermaßen klein, dass sie keine visuelle Auswirkung haben und lediglich von mathematisch-theoretischer Bedeutung sind. Generell müssten die angegebenen Terme, die das emittierte Licht beschreiben, größer als die reflektiven sein, wenn beide auf die Lichtquelle verweisen:

$$\varepsilon(x,A) \geq \rho(x,x',A)$$

Aufgabe 14: Prozedurale Modellierung von Oberflächennormalen

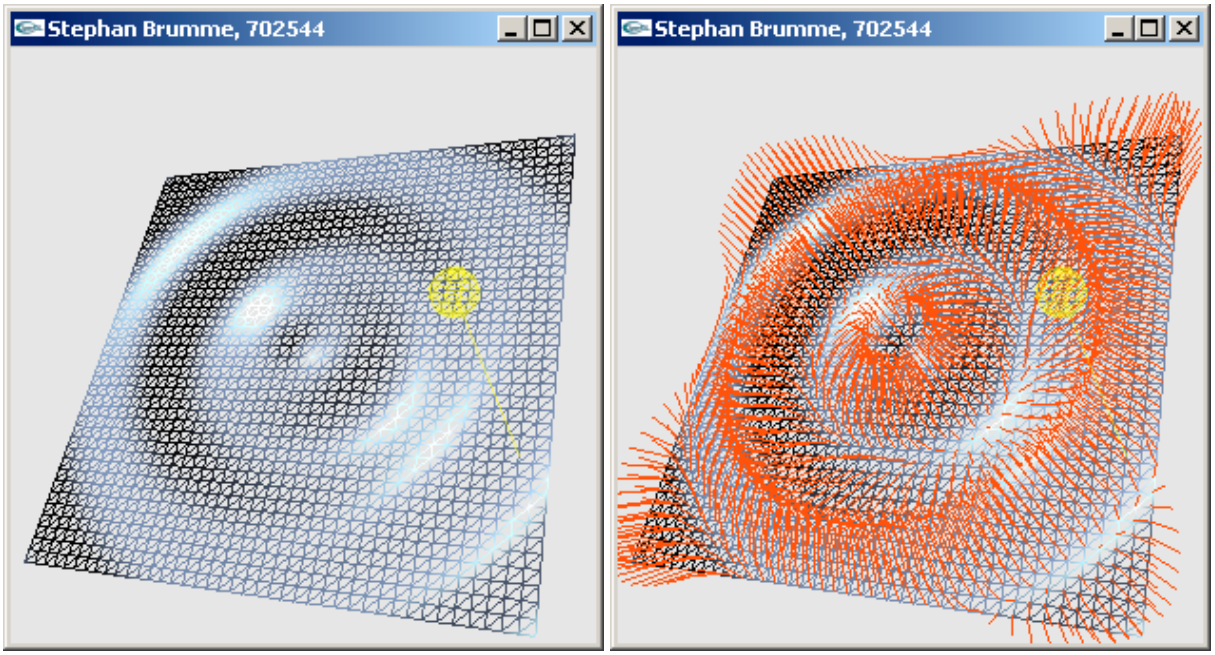
Die Bedienung des Programms wird über diese Tasten realisiert:

Taste	Aktion
ESC	Programm beenden
Leertaste	Patch rotieren an/aus
+	in die Szene hineinzoomen
-	aus der Szene herauszoomen
q	zusätzliche Lichtquelle an
Q	zusätzliche Lichtquelle aus
c	rückseitige Flächen zeichnen/verstecken
n	Normalen anzeigen/verstecken
p	Wellenbewegung an/aus
l	Lichtquelle bewegen an/aus
1-9	Größe des Patches verändern, sehr grob (1) bis sehr fein (9), Startwert: 5



Im Screenshot sieht man, dass die bewegliche Lichtquelle als gelbe Kugel symbolisiert wird, eine Linie hilft bei der Veranschaulichung der räumlichen Position, da sie genau senkrecht auf dem Patch steht.

Dass der Patch wirklich nur flach ist, kann man an seiner Geometrie sehen, der optische Effekt wird allein durch die Normalen erreicht:



Für die Berechnung der Normalen gibt es verschiedene Möglichkeiten, die zu mehr oder weniger guten Ergebnissen führen. Grundsätzlich ist es so, dass für jeden Punkt *p*, für den die Normale *n* ermittelt werden soll, sowohl ein Vektor *v*, der von der Patchmitte zu *p* zeigt und die Länge *d* von *v* errechnet werden müssen:

$$v = p - O$$
$$d = \|v\|$$

Glücklicherweise liegt die Mitte des Patches im Koordinatenursprung, sodass $O = 0$ und $v = p$ gilt.
Um aus diesen beiden Werten die Normale *n* herzuleiten, experimentierte ich anfänglich mit Kugelkoordinaten, sodass Gleichungen dieser Art entstanden:

$$\alpha = d \cdot k + offset$$
$$w = \frac{v}{\|v\|}$$
$$n = \begin{pmatrix} w_x \cdot \cos\alpha \\ \sin\alpha \\ w_z \cdot \cos\alpha \end{pmatrix}$$

Hier tauchen allerdings Probleme auf, da n_y negativ werden kann. Natürlich ist es möglich, durch Absolutwertbildung oder Winkelverschiebung das zu verhindern, allerdings befriedigte mich das visuelle Resultat nicht.

Ich möchte noch kurz anmerken, dass *offset* für die Wellenbewegung zuständig ist, während *k* eine subjektiv gewählte Konstante darstellt, die die Wellenlänge beeinflusst. Es stellte sich heraus, dass $k=20$ vernünftige Ergebnisse liefert.

Meine implementierte Lösung beruht einem ganz ähnlichen Ansatz. Ich setze den y-Anteil immer auf 1 (d.h. ich lasse die Sinus-Funktion außer Acht). Nun muss noch dafür gesorgt werden, dass *n* auch wirklich die Länge 1 hat, weshalb im Anschluss eine Normalisierung erforderlich ist:

$$\alpha = d \cdot k + offset$$

$$w = \frac{v}{\|v\|}$$

$$m = \begin{pmatrix} w_x \cdot \cos\alpha \\ 1 \\ w_z \cdot \cos\alpha \end{pmatrix}$$

$$n = \frac{m}{\|m\|}$$

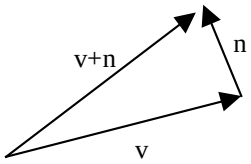
Rein physikalisch machen diese Gleichungen nicht mehr allzuviel Sinn, das visuelle Ergebnis ist dennoch überzeugend. Im Code habe ich eine Funktion `WavedNormal` eingeführt, die aus einem Vektor *v*, der einem Punkt des Patches entspricht, die dazugehörende Normale *n* berechnet und zurückgibt:

```
Vector CGWave::WavedNormal(const Vector& v) const
{
    // get distance, stretch wave length
    double distance = abs(v)*20;
    // add periode shift
    distance += periode_;

    // normalize to [0,2*PI[
//  while (distance >= 2*PI)
//      distance -= 2*PI;

    // normalize point vector
    const Vector v_norm = v.normalized();

    const double cos_dist = cos(distance);
    const double nx = v_norm[0]*cos_dist;
    const double ny = 1; // alternative: sin(distance);
    const double nz = v_norm[2]*cos_dist;

    return Vector(nx, ny, nz).normalized();
}
```

Die eigentliche Routine zum Zeichnen *eines* Dreieckseckpunkt inkl. Normale besteht nur noch aus zwei OpenGL-Befehlen:

```
void CGWave::handleVertex(const Vector& v) const {
    // compute normal according to wave functions
    glNormal3dv(WavedNormal(v).rep());
    // send vertex
    glVertex3dv(v.rep());
}
```

Die Darstellung der Normalen als rote Linien beruht auf der Vektoraddition:



In OpenGL wird demzufolge eine Linie von *v* nach *v+n* gezeichnet:

```
// show normals ?
if(showNormals_) {
    glDisable(GL_LIGHTING);
    // lady in red
    glColor3f(1,0,0);
    glBegin(GL_LINES);

    c = 0;
    // process all vertices
    for (int i=0; i<2*num_+2; i++)
        for (int j=0; j<num_; j++)
        {
            // get vertex
            Vector& v = list_[c++];
            // beginning from surface (small offset, though) ...
            glVertex3d (v[0], v[1]+0.01, v[2]);
            // ... to the end which we get by adding (stretched) normal to v
            glVertex3dv((v+(0.1*WavedNormal(v))).rep());
        }

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

Im Code sind zwei kleine Erweiterungen enthalten, die ich bisher nicht besprochen habe. Als erstes werden die Normalen nicht in ihrer vollen Länge 1 gezeichnet, da sie sonst den Patch zu sehr verdecken würden. Eine Verkürzung auf ein Zehntel umgeht dieses Problem. Zweitens störte mich, dass einige Normalen den Patch durchstießen und so auf der Rückseite des Patches unschöne vereinzelte rote Punkte auftauchten. Indem ich die Linien nicht direkt bei *v* beginnen lasse, sondern den y-Wert minimal erhöhe, verschwinden diese Artefakte.

Die letzte Teilaufgabe bestand darin, die Lichtquelle zu bewegen. Ich entschied mich dabei für eine kreisförmige Bahn, deren Koordinaten sich ergeben aus:

$$l = \begin{pmatrix} bahnradius \cdot \cos\dfrac{winkel \cdot \pi}{180} \\ höhe \\ bahnradius \cdot \sin\dfrac{winkel \cdot \pi}{180} \end{pmatrix}$$

Wichtig ist, dass C die trigonometrischen Funktionen auf Basis von Radiant statt Grad berechnet, daher ist die Umrechnung mit π notwendig. Die Konstanten *bahnradius* und *höhe* sind auf 0,5 gesetzt worden, dies sind subjektiv angenehme Werte. Der Winkel wird vom Programmrahmen kontinuierlich erhöht und findet sich in der Variablen `rot_` wieder. Mir war allerdings die Geschwindigkeit zu hoch, so dass ich sie auf ein Drittel drosselte (das ist der Fluch schneller CPUs im Zusammenspiel mit T&L-Grafikkarten ☺).

```
if(lightAnim_)
{
    rot_+=2; // Animierte Lichtquellen
    // slow down rotation
    double slow_rot = rot_/3;
    light_position1[0] = 0.5*cos(slow_rot*PI/180.0);
    light_position1[1] = 0.5;
    light_position1[2] = 0.5*sin(slow_rot*PI/180.0);
}
```

Quelltext

Die Header-Datei cgwave.h wurde nur minimal geändert, lediglich `WavedNormal` ist als konstante Memberfunktion definiert:

cgwave.h:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 5
//

#ifndef CG_WAVE_H
#define CG_WAVE_H

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

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

    // 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:

    void drawScene();
    void drawObject();
    void buildPatch(int detail);
    void handleVertex(const Vector& v) const;
    Vector WavedNormal(const Vector& v) const;

    int width_;
    int height_;
    double zoom_;
    double rot_;
    double periode_;

    bool culling_;
    bool run_;
    bool showNormals_;
    bool periodAnim_;
    bool lightAnim_;
    int num_;
    Vector* list_;
};

#endif // CG_WAVE_H
```

cgwave.cpp:

```
//
// Computergraphik II
// Prof. Dr. Juergen Doellner
// Wintersemester 2001/02
//
// Rahmenprogramm zu Aufgabenzettel 5
//

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

#ifndef PI
const double PI = 3.14159265358979323846;
#endif

CGWave::CGWave() {
    run_ = false;
    culling_ = false;
    showNormals_ = false;
    periodAnim_ = false;
    lightAnim_ = false;
    zoom_ = 1.5;
    periode_=0;
    list_ = NULL;
    rot_ = 180.0;

    buildPatch(40);
}

CGWave::~CGWave() {
}

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

void CGWave::drawScene() {

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


    glLightfv(GL_LIGHT1, GL_POSITION, light_position1);
    drawObject();
    // Positionieren bzw. Animieren der Lichtquelle 1


    // draw light source (yellow !)
    glPushMatrix();
    glDisable(GL_LIGHTING);

    // draw vector from light source
    glColor4f(1, 1, 0, 0.5);

    glBegin(GL_LINES);
    glVertex3f(light_position1[0], light_position1[1], light_position1[2]);
    glVertex3f(light_position1[0], 0, light_position1[2]);
    glEnd();
    // translate sphere that represents our light source

    glTranslatef(light_position1[0], light_position1[1], light_position1[2]);
    glutSolidSphere(0.03,10,10);

    glEnable(GL_LIGHTING);
    glPopMatrix();

    glPopMatrix();
}

Vector CGWave::WavedNormal(const Vector& v) const
{
    // get distance, stretch wave length
    double distance = abs(v)*20;
```

```
    // add periode shift
    distance += periode_;

    // normalize to [0,2*PI[
//  while (distance >= 2*PI)
//      distance -= 2*PI;

    // normalize point vector
    const Vector v_norm = v.normalized();

    const double cos_dist = cos(distance);
    const double nx = v_norm[0]*cos_dist;
    const double ny = 1; // alternative: sin(distance);
    const double nz = v_norm[2]*cos_dist;

    return Vector(nx, ny, nz).normalized();
}

void CGWave::handleVertex(const Vector& v) const {
    // compute normal according to wave functions
    glNormal3dv(WavedNormal(v).rep());
    // send vertex
    glVertex3dv(v.rep());
}

void CGWave::drawObject() {
    // draw geometry
    int c=0;
    for(int j=0; j<num_; j++) {
        glBegin(GL_TRIANGLE_STRIP); // Performance
        // 1)
        handleVertex(list_[c]); c++;
        // 2)
        handleVertex(list_[c]); c++;
        // 3)
        handleVertex(list_[c]); c++;

        for(int i=0; i<((num_+1)*2)-3; i++) {
            // Rest des Strips
            handleVertex(list_[c]); c++;
        }

        glEnd();
    }

    // show normals ?
    if(showNormals_) {
        glDisable(GL_LIGHTING);

        // lady in red
        glColor3f(1,0,0);
        glBegin(GL_LINES);

        c = 0;
        // process all vertices
        for (int i=0; i<2*num_+2; i++)
            for (int j=0; j<num_; j++)
            {
                // get vertex
                Vector& v = list_[c++];
                // beginning from surface (small offset, though) ...
                glVertex3d (v[0], v[1]+0.01, v[2]);
                // ... to the end which we get by adding (stretched) normal to v
                glVertex3dv((v+(0.1*WavedNormal(v))).rep());
            }

        glEnd();
        glEnable(GL_LIGHTING);
    }
}

void CGWave::buildPatch(int detail) {
    // In Dreiecken tessellierten Fläche
    if(list_) delete list_;

    const int points = (detail)*(4+((detail-1)*2));
```

```
    list_ = new Vector[points];

    double offset = 1.0/detail;

    double x;
    double z;
    num_ = detail;
    int c = 0;
    for(int j=0; j<detail; j++) { // passend zu Triangle-Strip (s.o.)
        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 CGWave::onInit() {
    // OpenGL Lichtquelle 0
    static GLfloat light_diffuse[] = {0.7, 0.8, 0.5, 1.0};
    static GLfloat light_position[] = {0.0, 1.0, 0.0, 1.0};
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    // glEnable(GL_LIGHT0); // wenn T&L nicht unterstützt

    // OpenGL Lichtquelle 1
    static GLfloat light_diffuse1[] = {0.7, 0.8, 1.0, 1.0};   /* Diffuse light. */
    static GLfloat light_specular1[] = {0.6, 0.8, 0.6, 1.0};  /* Positional light source */
    glLightfv(GL_LIGHT1, GL_DIFFUSE, light_diffuse1);
    glLightfv(GL_LIGHT1, GL_SPECULAR, light_specular1);
    glLightfv(GL_LIGHT1, GL_POSITION, light_position1);
    glEnable(GL_LIGHT1);

    glEnable(GL_LIGHTING);

    // OpenGL Material
    static GLfloat mat_specular[] = {0.8, 0.8, 0.8, 1.0};     /* Material property. */
    glMaterialf(GL_FRONT, GL_SHININESS, 32);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);

    // automatische Normalisierung
    glEnable(GL_NORMALIZE);

    // Tiefen Test aktivieren
    glEnable(GL_DEPTH_TEST);

    // Blending aktivieren
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    // Smooth Schattierung aktivieren
    glShadeModel(GL_SMOOTH);
//    glShadeModel(GL_FLAT);

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

```
    // LookAt tesselieren
    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

    glRotatef(-25, 0.0, 1.0, 0.0);
    glClearColor(0.9,0.9,0.9,1.0);
}

void CGWave::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 CGWave::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 'q': { glEnable(GL_LIGHT0); break; }
    case 'Q': { glDisable(GL_LIGHT0); break; }
    case 'c': { culling_ = !culling_; break; }
    case 'n': { showNormals_ = !showNormals_; break; }
    case 'p': { periodAnim_ = !periodAnim_; break; }
    case 'y': { glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); break;}
    case 'x': { glPolygonMode(GL_FRONT_AND_BACK, GL_FILL); break;}
    case 'l': { lightAnim_ = !lightAnim_; break; }
    case '1': { buildPatch(1); break; }
    case '2': { buildPatch(2); break; }
    case '3': { buildPatch(5); break; }
    case '4': { buildPatch(10); break; }
    case '5': { buildPatch(20); break; }
    case '6': { buildPatch(40); break; }
    case '7': { buildPatch(50); break; }
    case '8': { buildPatch(100); break; }
    case '9': { buildPatch(200); break; } // netter Versuch
    }
    onDraw();
}

void CGWave::onIdle() {
    if(periodAnim_)
    { // Periodische Wellenbewegung
        periode_ -= 0.05;
        if (periode_  < 0 )
            periode_ += 2*PI;
    }

    if(lightAnim_)
    {
        rot_+=2; // Animierte Lichtquellen

        // slow down rotation
        double slow_rot = rot_/3;
        light_position1[0] = 0.5*cos(slow_rot*PI/180.0);
        light_position1[1] = 0.5;
        light_position1[2] = 0.5*sin(slow_rot*PI/180.0);
    }

    if (run_)  // Simple Rotation
        glRotatef(.5, 0.0, 1.0, 0.0);

    // Redraw
    onDraw();
}

void CGWave::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:
    CGWave sample;

    cout << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
         << "Leertaste  Objekt drehen" << endl
         << "+          Hineinzoomen" << endl
         << "-          Herauszoomen" << endl
         << "n          Normalen anzeigen/verbergen" << endl
         << "p          Wellen animieren an/aus" << endl
         << "l          Lichtquelle bewegen" << endl
         << "q          zweite Lichtquelle an" << endl
         << "Q          zweite Lichtquelle aus" << endl
         << "y          Drahtgittermodell" << endl
         << "x          Flaechen ausfuellen" << endl
         << "1 bis 9    Tessellationsgrad (1 - ungenau, schnell   9 - sehr exakt, langsam)" <<
endl;

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