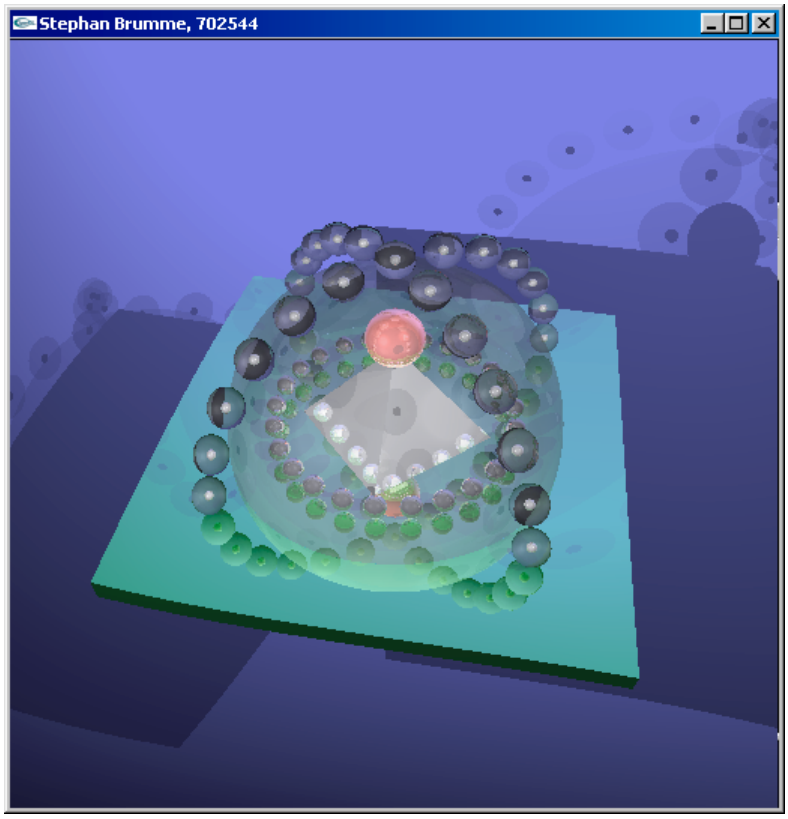
Aufgabe 17

Einleitung

Mein Raytracer erzeugt in Höchstgeschwindigkeit ein atemberaubendes Bild vom sogenannten *Heiligen Gral der Computergrafiker*. Alle, die bei der Betrachtung in spontane Begeisterung ausbrechen und dies auch mit der Vergabe von 8 Sonderpunkten belohnen, werden mit einem ewigen Leben in Glückseligkeit belohnt. Sollte dies nicht der Fall sein, so bestraft der *Heilige Gral der Computergrafiker* solch unhöfliches Verhalten mit min. 7 Jahren Unglück.



Bei Programmstart kann die Größe des zu berechnenden Bildes angegeben werden, der Raytracing-Prozess wird notfalls mit der Escape-Taste abgebrochen. Die Szene ist gemäß Aufgabenstellung im Quelltext fest verankert, es erfolgt eine automatische Speicherung des Bildes in der Datei raytracer.ppm. Während der Berechnung wird das Bild bereits auf dem Bildschirm schrittweise verfeinert dargestellt (unter Benutzung von OpenGL).

Es werden alle Anforderungen der Aufgabenstellung unterstützt, d.h. Kugeln, Dreiecke und achsenparallele Quader sind darstellbar. Lichtquellen sind verantwortlich für Reflektionen, Refraktionen und Schattenwurf. Alle Szenenobjekte können halbtransparent sein.

Ich habe mich bemüht, den Raytracer nicht nur optisch gut, sondern auch vor allem schnell zu machen. Selbst ohne Verwendung von Bounding-Boxes sind über 2,5 Mio. Intersektionstests pro Sekunde auf einer Mittelklasse-1,1-GHz-CPU (Intel Celeron/Coppermine-Core) möglich. Das abgedruckte Bild mit einer Auflösung von 512x512 wurde mit einer Rekursionstiefe von 7 in nur 39 Sekunden generiert.

Intersektionstests

Die Intersektionstests wurden theoretisch in der Vorlesungen bereits besprochen, ich gehe daher nicht weiter auf die mathematischen Grundlagen ein sondern beschränke mich auf die Besonderheiten meines Raytracers.

Die Intersektion eines Strahls mit einem Dreieck beruht auf der Verwendung von baryzentrischen Koordinaten. Der Algorithmus aus der Vorlesung konnte 1:1 umgesetzt werden. Die einzige Hürde besteht in der Erzeugung der Normalen, da ein Dreieck sowohl von der Vorder- als auch von der Rückseite betrachtet werden kann. Der Raytracer muss dies beachtet, ein Dreieck hat daher quasi zwei Normalen, die sich nur im Vorzeichen unterscheiden. Von welcher Seite man das Dreieck betrachtet, erkennt man am Vorzeichen des Skalarproduktes von Strahlrichtung und einer der beiden Normalen. Erhält man einen positiven Wert, so ist die benutzte Normale zu invertieren.

```
bool Triangle::intersect(const Ray& ray,Vector& point,Vector& normal, double& distance) const
{
    // edges of the triangle
    const Vector edge1 = v_[1] - v_[0];
    const Vector edge2 = v_[2] - v_[0];

    const Vector p    = ray.getDirection() * edge2;
    const double detA = dotProduct(p, edge1);

    // det(A) != 0
    if (detA == 0)
        return false;

    const double f = 1/detA;
    const Vector s = ray.getOrigin() - v_[0];
    const double u = f * dotProduct(p, s);

    // first barycentric value must be in [0,1]
    if (u<0 || u>1)
        return false;

    const Vector q = s*edge1;
    const double v = f * dotProduct(q, ray.getDirection());

    // second barycentric value must be in [0,1]
    if (v<0 || v>1 || u+v>1)
        return false;

    const double t = f * dotProduct(q, edge2);

    // object on the wrong side of the ray ?
    if (t <= 0)
        return false;


    // intersection found, barycentric coordinates are (u,v,t)
    point  = ray.getOrigin() + t*ray.getDirection();

    normal = triNrm_;
    // invert normal to allow the triangle being viewed from its back side
    if (dotProduct(normal, ray.getDirection()) > 0)
        normal = -normal;

    distance = t;//abs(point-ray.getOrigin());

    return true;
}
```

Die Intersektion eines Strahls mit einer Kugel entspricht erneut 1:1 dem Algorithmus aus der Vorlesung, es sind keinerlei Besonderheiten zu beachten:

```
bool Sphere::intersect(const Ray& ray,Vector& point,Vector& normal, double& distance) const {
    // ray-sphere intersection

    // ray direction
    const Vector direction(ray.getDirection());
    const double i = direction[0];
    const double j = direction[1];
    const double k = direction[2];

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

    // sphere's center
    const double l = center_[0];
    const double m = center_[1];
    const double n = center_[2];
    const double r = radius_;

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

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

    // no intersection
    if (D < 0)
        return false;

    // nearest ray intersection
    double t;

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

    // all intersections behind the origin ?
    if (t <= 0.001)
        return false;


    point  = ray.getOrigin() + t*ray.getDirection();
    normal = (point - center_)/r; // divison by "r" normalizes the vector
    distance = t;//abs(point-ray.getOrigin());

    return true;
}
```

Die Intersektion eines Strahls mit einer Box war am aufwändigsten zu implementieren, da das Skript hier nur unscharfe Ideen formulierte und so quasi alles von Grund auf entwickelt werden musste. Das Verfahren beruht darauf, dass ich nacheinander die Schnittpunkte der Ebenen, in denen die Seiten der Box liegen, bestimme. Dabei ist der Sonderfall der Parallelität zu berücksichtigen, ansonsten erhält man Divisionen durch Null. Schrittweise nähert sich `tnear` der tatsächlich nächsten Intersektion an. Da ich nicht im voraus weiß, wann `tnear` minimal ist, muss ich auf Verdacht die Normale jedes Mal setzen, wenn ich ein kleineres `tnear` gefunden habe. Die Hilfsvariable `tfar` ist notwendig, um erkennen zu können, ob ein Strahl die Box verfehlt.

```
bool Box::intersect(const Ray& ray,Vector& point,Vector& normal, double& distance) const {
    const static Vector NormalRight(+1,0,0);
    const static Vector NormalLeft (-1,0,0);
    const static Vector NormalUp   (0,+1,0);
    const static Vector NormalDown (0,-1,0);
    const static Vector NormalFront(0,0,+1);
    const static Vector NormalBack (0,0,-1);

    // ray origin
    const Vector origin = ray.getOrigin();
    // ray direction
    const Vector direction = ray.getDirection();


    // ray enters box (-infinity)
    double tnear = -999999999999;
    // ray leaves box (+infinity)
    double tfar  = +999999999999;


    ////////////////////////////////////////////////////
    // x
    const double& ox = origin[0];
    const double& rx = direction[0];

    // parallel to box ?
    if (rx == 0)
        if (ox < llf_[0] || ox > urb_[0])
            return false;

    // intersections
    const double tx1 = (llf_[0] - ox) / rx;
    const double tx2 = (urb_[0] - ox) / rx;

    // adjust tnear and tfar
    if (tx2 > tx1 && tx1 > tnear)
    {
        tnear  = tx1;
        normal = NormalLeft;
    }
    if (tx1 > tx2 && tx2 > tnear)
    {
        tnear = tx2;
        normal = NormalRight;
    }

    // ray doesn't hit box ?
    tfar = min(tfar, max(tx1, tx2));
    if (tnear > tfar || tfar < 0)
        return false;


    ////////////////////////////////////////////////////
    // y
    const double& oy = origin[1];
    const double& ry = direction[1];

    // parallel to box ?
    if (ry == 0)
        if (oy < llf_[1] || oy > urb_[1])
            return false;

    // intersections
    const double ty1 = (llf_[1] - oy) / ry;
    const double ty2 = (urb_[1] - oy) / ry;
```

```
    // adjust tnear and tfar
    if (ty2 > ty1 && ty1 > tnear)
    {
        tnear  = ty1;
        normal = NormalDown;
    }
    if (ty1 > ty2 && ty2 > tnear)
    {
        tnear  = ty2;
        normal = NormalUp;
    }

    // ray doesn't hit box ?
    tfar = min(tfar, max(ty1, ty2));
    if (tnear > tfar || tfar < 0)
        return false;


    ////////////////////////////////////////////////////
    // z
    const double& oz = origin[2];
    const double& rz = direction[2];

    // parallel to box ?
    if (rz == 0)
        if (oz < llf_[2] || oz > urb_[2])
            return false;

    // intersections
    const double tz1 = (llf_[2] - oz) / rz;
    const double tz2 = (urb_[2] - oz) / rz;

    // adjust tnear and tfar
    if (tz2 > tz1 && tz1 > tnear)
    {
        tnear  = tz1;
        normal = NormalBack;
    }
    if (tz1 > tz2 && tz2 > tnear)
    {
        tnear  = tz2;
        normal = NormalFront;
    }

    // ray doesn't hit box ?
    tfar = min(tfar, max(tz1, tz2));
    if (tnear > tfar || tfar < 0)
        return false;


    // intersection finally found !!!
    point  = ray.getOrigin() + tnear*ray.getDirection();
    distance = tnear;//abs(point-ray.getOrigin());

    // done !
    return true;
}
```

Generierung des Primärstrahls

Für die Erzeugung der Primärstrahlen benötigt man die Kameraeinstellungen (From, To, Up, Field-Of-View) und die Bildauflösung. Für jeden einzelnen Pixel des Bildes wird daraus ein Strahl konstruiert, der von der Kamera ausgehend in die Szene hineingeht.

$$Ray(Camera, Pixel) = (Camera.From,\ Direction)$$

Camera.From ist sofort bekannt, *Direction* entsteht erst unter Anwendung der Vektorgesetze. Zunächst konstruiere ich zwei normierte Vektoren *Up* und *Right*, die in ihrer Richtung die Projektionsebene beschreiben. Dazu benötige ich als Hilfsvektor die Blickrichtung *Look*:

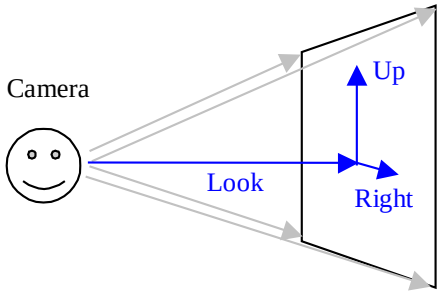


Abbildung 1: View plane

Look lässt sich sehr einfach aus den Kameraeinstellungen extrahieren:

$$Look = Camera.To - Camera.From$$

Da das Bild unverzerrt dargestellt werden soll, stehen *Look*, *Up* und *Right* senkrecht aufeinander, wobei ich ein rechtshändiges Koordinatensystem verwende. Es ist leider nicht garantiert, dass *Camera.Up* orthogonal zu *Look* ist, deshalb berechne ich es sicherheitshalber neu:

$$Right = Camera.Up \times Look$$
$$Up = Right \times Look$$

Für jeden einzelnen Strahl wird die jeweilige Richtung durch zwei Winkel *up_angle* und *right_angle* bestimmt. Sie haben den Wertebereich [*-fovy/2, +fovy/2*] bzw. [*-fovy*(width/height)*/2, + *fovy*(width/height)*/2]. Die Unterscheidung ist notwendig, um Verzerrung bei Breitbildern zu unterbinden.

$$Up' = Up \cdot |Look| \cdot \tan up\_angle$$
$$Right' = Right \cdot |Look| \cdot \tan right\_angle$$
$$Direction = \frac{Look + Up' + Right'}{|Look + Up' + Right'|}$$

In der Implementation muss man noch daran denken, dass C++ leider nur mit der Winkeleinheit Radiant umgehen kann:

$$1° \triangleq \frac{\pi}{180} rad$$

Ich habe verschiedene Techniken benutzt, um den Rechenaufwand zu minimieren, insbesondere zielen sie auf einen schnelleren Bildaufbau für das als Zusatzaufgabe gestellte *progressive-view*-Feature ab. Sie sind grau dargestellt und werden später separat erläutert:

```
void CGRayTracer::createImage(int width, int height) {
    // convert field-of-view angle to rad
    const double fovy_rad = camera_->fovy * PI/180;

    // look vector
    const Vector Look  = camera_->to - camera_->from;

    // delta angles
    const double up_angle_delta    = fovy_rad / height;
    const double right_angle_delta = fovy_rad / height;//width;

    // normalized vectors that describe the view plane (right handed !)
    const Vector Right = camera_->up.normalized() * Look.normalized();
    const Vector Up    = Right * Look.normalized();

    // image
    const double widthoffset  = imagex_/(double)width;
    const double heightoffset = imagey_/(double)height;
    const double pixelsize = detail_;

    for(int i=0; i<height; i++)
    {
        for(int j=0; j<width; j++)
        {
            // create rays for each pixel and send through scene

            // skip if already calculated
            const int truex = int(j*widthoffset);
            const int truey = int(i*heightoffset);
            Color& clrPixel = image_[truey*imagex_+truex];
            if (clrPixel[0] >= 0)
            {
                // draw already calculated pixels
#ifndef DONT_DRAW
                glColor3dv(clrPixel.rep());
                glRectd(truex, height_-truey,
                        truex+pixelsize, height_-(truey+pixelsize));
#endif
                continue;
            }

            // angles
            const double up_angle    = (i-height/2.0) * up_angle_delta;
            const double right_angle = (j-width /2.0) * right_angle_delta;

            // corresponding vectors
            const Vector CurrentUp    = Up    * abs(Look) * tan(up_angle);
            const Vector CurrentRight = Right * abs(Look) * tan(right_angle);

            // put it all together
            const Vector LookAt = Look + CurrentUp + CurrentRight;

            // ray tracing
            clrPixel = rayTrace(Ray(camera_->from, LookAt));

            // draw pixel
#ifndef DONT_DRAW
            glColor3dv(clrPixel.rep());
            glRectd(truex, height_-truey,
                    truex+pixelsize, height_-(truey+pixelsize));
#endif
        }
    }
}
```

Die letzten Zeilen (ab Kommentar „ray tracing“) dienen der Zuweisung der berechneten Farbe durch Aufruf von `rayTrace` mit dem aktuellen Strahl als Parameter. Dabei verzichte ich auf eine Normalisierung der Strahlrichtung, da dies der Konstruktor von `Ray` selbstständig durchführt.

Phong-Beleuchtung und Schattentest

Die Phong-Beleuchtung beruht auf der Aufgabe 10, wo ich sie bereits erfolgreich eingesetzt hatte. Als Erweiterung ist der Code jetzt in Lage, auch den Halbschatten von halbtransparenten Objekten zwischen dem zu beleuchteten Punkt und der Lichtquelle zu berücksichtigen. Physikalisch korrekt wäre es, wenn sich der Halbschatten durch die verdeckenden Objekte einfärben ließe, aus Geschwindigkeitsgründen verzichte ich aber darauf und belasse es bei einem Luminanzwert namens `dMaxIntensity`.

Der Schattentest (im Code im Block `if(shadows_)`) beruht darauf, dass ich vom zu beleuchtenden Punkt einen Strahl zu allen Lichtquellen aussende. Jede Intersektion mit einem halbtransparenten Szenenobjekt mindert dabei die einfallende Lichtintensität, opake Körper führen zu einem Schatten. Die Berechnung wird dann enorm beschleunigt, da man nur den ambienten Anteil zu berücksichtigen braucht und zur nächsten Lichtquelle übergehen kann.

Leider müssen die mathematisch exakten Formel zur Bestimmung des Schattenteststrahl auf dem Computer verändert werden, da die berechneten Intersektionspunkte durch Rundungsungenauigkeiten nur selten den theoretischen Werten entsprechen und manchmal innerhalb der Szenenobjekte liegen. Im Bild äußert sich dieses Phänomenen durch ein ziemlich starkes Pixel-Rauschen (linkes Bild). Wenn man den Startpunkt des Strahls leicht verschiebt, ergibt sich das gewünschte Ergebnis (rechtes Bild):

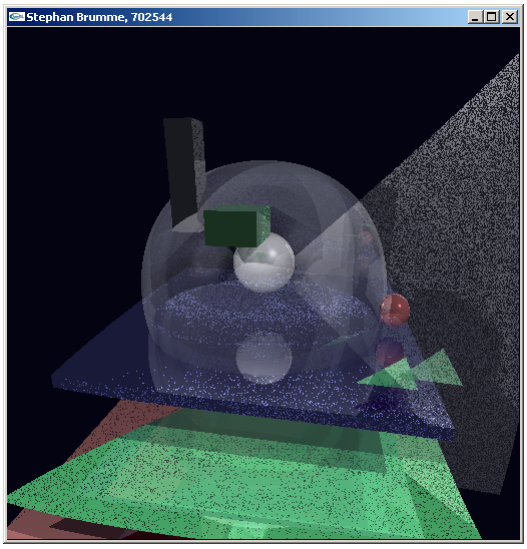


Abbildung 2: Nicht korrigierter Schattentest

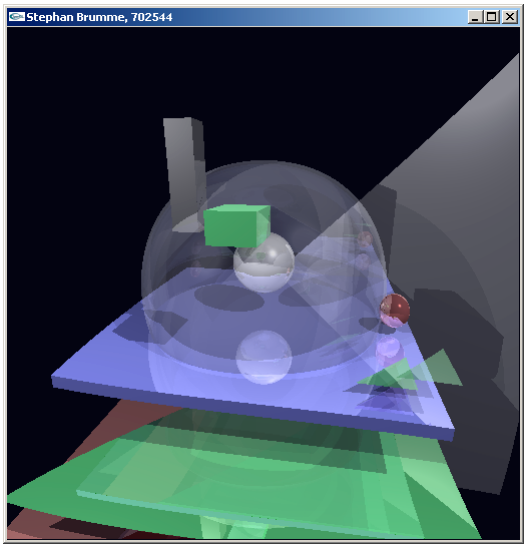
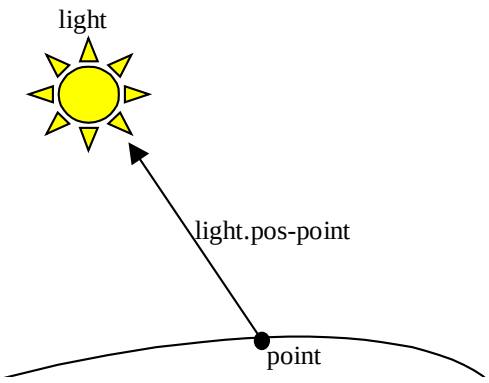
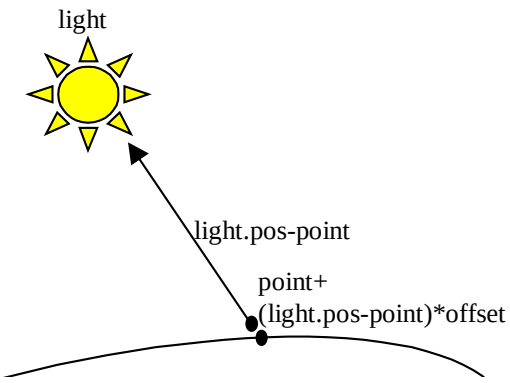


Abbildung 3: Korrigierter Schattentest

$$shadowray.From = point + (light.pos - point) * offset$$
$$shadowray.Direction = light.pos - point$$





Als akzeptabler Wert für *offset* hat sich 0,0001 erwiesen. Größere Werte führen zu Fehlern an Schnittkanten zwischen Körpern, kleinere erzeugen das erwähnte Pixelrauschen. Mathematisch ideal wäre es, wenn man *offset* auf Null setzen könnte. Je nach Rechengenauigkeit (ich benutze durchgehend `double`) kommt man der Null mehr oder weniger nahe.

```
Color CGRayTracer::phongIlluminationColor(const Vector& point, const Vector& normal, Material*
material) {
    // calculate Phong illumination

    // code is a slightly modified copy of my Phong homework

    // define intensity variables
    double dIntensityAmbient  = 0;
    double dIntensityDiffuse  = 0;
    double dIntensitySpecular = 0;

    // N = normal
    const Vector N_norm = normal.normalized();
    // V = view vector
    const Vector V = camera_->from - point;
    const Vector V_norm = V.normalized();

    // process all light sources
    for (int nLight=0; nLight < lightSize_; nLight++)
    {
        // get a single light source
        const Light* const light = light_[nLight];

        // ALWAYS ambient intensity, even if shadowed
        dIntensityAmbient  += light->ambient;

        // L = light vector
        const Vector L = light->pos - point;
        const Vector L_norm = L.normalized();

        // N*L
        const double NdotL = dotProduct(N_norm, L_norm);

        // intensity reaching the surface (attenuation)
        const double dLightDistance = abs(L);
        double dMaxIntensity = 1.0 / (light->att_constant + dLightDistance*light->att_linear
                                      + dLightDistance*dLightDistance*light->att_quadric);
        if (dMaxIntensity > 1)
            dMaxIntensity = 1;

        // get shadow
        if (shadows_)
        {
            // shadow ray must not start at the object's surface !
            // (else we get in trouble with intersection problems)
            const Ray shadowray(point+(light->pos - point)*INTERSECTION_OFFSET, light->pos -
point);
            bool shadow = false;

            Shape** ppShape = shape_;
            for (int shape=0; shape<shapeSize_; shape++)
            {
                // get values of each intersection
                double distance;
                static Vector _normal, _point;

                // does the ray intersect the object ?
                shadow = shape_[shape]->intersect(shadowray, _point, _normal, distance);
                nTests_++;

                // intersection between light source and surface point ?
                if (shadow && distance < dLightDistance)
                {
                    dMaxIntensity *= 1-material_[shape]->opacity;
                    if (dMaxIntensity > 0.01)
                        shadow = false;
                    else
                        // 100% shadow
                        break;
                }
```

```
            }
            // shadowed ! no diffuse or specular light possible
            if (shadow)
                continue;
        }

        // R*V = (2*N*(N*L)-L)*V
        const double RdotV = max(dotProduct((2*NdotL*N_norm - L_norm).normalized(), V_norm),
0);

        // diffuse intensity
        dIntensityDiffuse  += dMaxIntensity * material->kd * NdotL;
        // specular intensity
        dIntensitySpecular += dMaxIntensity * material->ks * pow(RdotV, material->n);
    }

    // return clamped color
    return material->color * min(dIntensityAmbient+dIntensityDiffuse+dIntensitySpecular,1);
}
```

Rekursive Reflektionsstrahlen

Die Methode `rayTrace` bestimmt für einen Strahl die Farbe in der Szene. Dabei wird sowohl eine Phong-Beleuchtung durchgeführt als auch die Reflektion und Refraktion beachtet. Für diese beiden ruft sich die Methode rekursiv selbst auf. Um eine unendliche Rekursion zu verhindern, bestimmt ein Parameter `nDepth` die aktuelle Rekursionstiefe. Überschreitet dieses einen global definierten Maximalwert, so wird abgebrochen und die Farbe Schwarz zurückgeliefert.

Ich teste den Sichtstrahl mit jedem Objekt auf Intersektion. Es ist durchaus möglich, dass der Strahl mehrere Szenenobjekte schneidet, in diesem Fall muss das ausgewählt werden, das am nächsten am Strahlursprung liegt. Diese Aufgabe erfüllt die Variable `closestDistance` in Kombination mit `closestShape`. Für die spätere Beleuchtung etc. merke ich mir zusätzlich den Schnittpunkt und dessen Normale (`closestPoint` und `closestNormal`). Wenn wider Erwarten keine Intersektion existiert, dann wird eine global definierte Hintergrundfarbe zurückgegeben (am besten schwarz).

Der nächste Schritt besteht in der Beleuchtung des Punktes auf Grundlage des Phong-Modells (siehe vorheriger Abschnitt).

Nur wenn der geschnittene Körper eine spekularen Anteil besitzt, kann der Lichtstrahl rekursiv reflektiert werden. Der Ursprung des neuen Lichtstrahls ist der Schnittpunkt `closestPoint`, die Richtung ergibt sich aus dem Grundsatz „Einfallswinkel gleich Ausfallswinkel“.

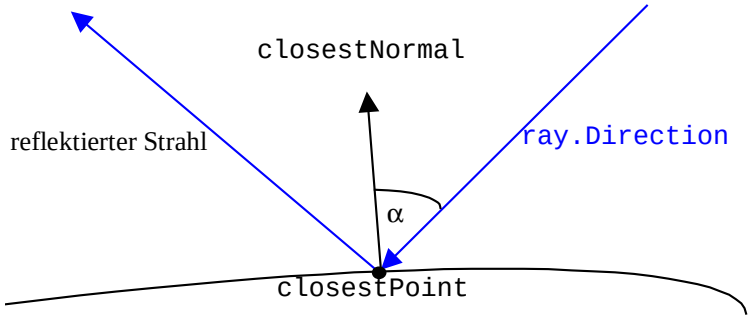


Abbildung 4: Reflektion an der äußeren Hülle

Bei der späteren Implementierung der Refraktion stellt sich das Problem, dass auch eine Reflektion *innerhalb* eines Körpers stattfinden kann. Hierzu invertiere ich die Normale, um das gewünschte Ergebnis zu erhalten:

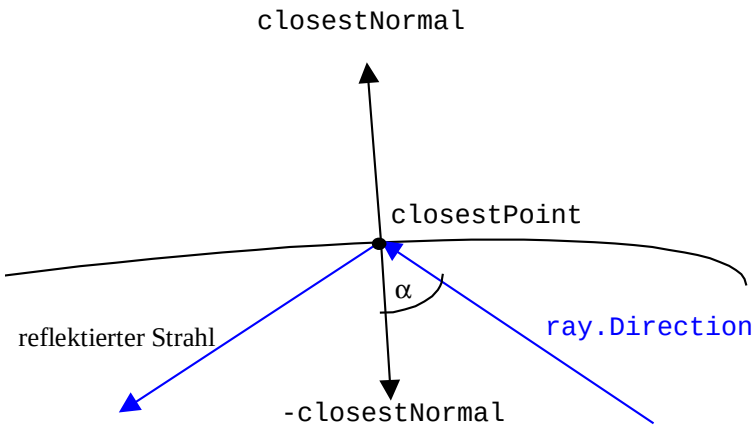


Abbildung 5: Reflektion an der inneren Hülle

Die Richtung des reflektierten Strahls ergibt sich aus:

$$\alpha = closestNormal \bullet ray.direction$$

$$reflected\_direction = ray.direction + 2\alpha \cdot closestNormal$$

Leider muss, analog zum Schattentest, wieder auf Rundungsungenauigkeiten Rücksicht genommen werden. Der Ursprung des Strahls ist deshalb nicht *closestPoint*, sondern wird leicht verschoben:

$$reflected\_origin = closestPoint + closestNormal \cdot offset$$

Als *offset* verwende ich auch 0,0001. Die ermittelte Farbe wird auf die bisher durch das Phong-Modell entstandene Farbe aufaddiert. Dabei können Überläufe auftreten (Farbanteile größer als 1), die am Ende der Methode `rayTrace` abgefangen und in auf 1 saturierte Werte umgeformt werden.

(Den Code für die Refraktion habe ich in das nächste Kapitel verschoben, er ist auch Bestandteil von `rayTrace`.)

```
Color CGRayTracer::rayTrace(const Ray& r, int nDepth) {
    // Check each shape for intersektion.
    // Return phong color of nearest shape
    // that is hit.

    if (nDepth++ > maxDepth_)
        return clrBackground;

    // shape that we found
    int closestShape = -1;
    // its parameters
    double closestDistance = 99999999;
    Vector closestPoint;
    Vector closestNormal;

    Shape** ppShape = shape_;
    for (int shape=0; shape<shapeSize_; shape++)
    {
        // get values of each intersection
        double distance;
        static Vector normal;
        static Vector point;

        // internal counter
        nTests_++;

        // does the ray intersect the object ?
        if (shape_[shape]->intersect(r, point, normal, distance))
            // closer than any intersection we tested before ?
            if (distance < closestDistance)
            {
                closestDistance = distance;
                closestShape    = shape;
                closestPoint    = point;
                closestNormal   = normal;
            }
    }

    // no intersection found
    if (closestShape < 0)
        return clrBackground;
    Material& material = *(material_[closestShape]);


    // object itself
    Color color = phongIlluminationColor(closestPoint, closestNormal, &material);

    // reflection
    if (material.ks >= 0.001)
    {
        // get reflected ray
        double reflected_angle     = -dotProduct(closestNormal, r.getDirection());
        if (reflected_angle < 0)
```

```
        {
            closestNormal   *= -1;
            reflected_angle *= -1;
        }

        const Vector reflected_direction = r.getDirection() + 2*reflected_angle*closestNormal;

        // reflected ray must not start at the object's surface !
        // (else we get in trouble with intersection problems)
        const Ray reflected_ray(closestPoint+closestNormal*INTERSECTION_OFFSET,
                                reflected_direction, material.lightspeed);

        // get color
        const Color reflected_color = rayTrace(reflected_ray, nDepth) * material.ks *
material.opacity;

        // add reflective part
        color += reflected_color;
    }

    // clamp color
    color[0] = min(color[0], 1);
    color[1] = min(color[1], 1);
    color[2] = min(color[2], 1);

    return color;
}
```

Refraktion transparenter Objekte

Der erste Schritt besteht darin, die Transparenz konsequent einzuführen. Dies hat z.B. Auswirkungen auf den Schattentest, die ich aber schon im entsprechenden Kapitel beschrieben habe.

Die Refraktion stellt eine Aufspaltung des Sichtstrahls dar:

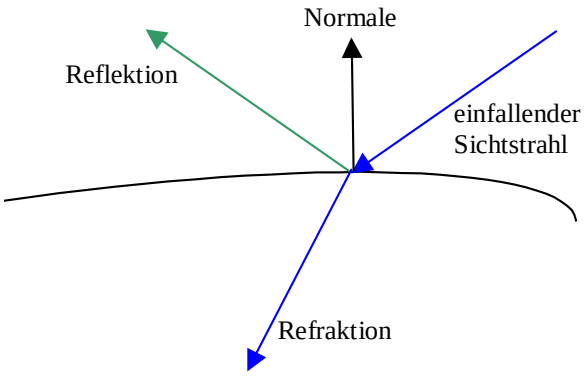


Abbildung 6: Aufteilung des Sichtstrahls in Reflektion und Refraktion

Die entscheidende Rolle bei der Ablenkung des Sichtstrahls spielt die Lichtgeschwindigkeit. Ich entschloss mich, die Klasse `Ray` um ein entsprechendes Attribut zu erweitern, da mir das als der sinnvollste Weg der Integration in den Raytracer schien. Hier ist leider die Namensgebung etwas irreführend, da ich nicht die reale Lichtgeschwindigkeit, sondern folgende Gleichung benutze:

$$lightspeed = \eta = \frac{c}{Lichtgeschwindigkeit\ im\ Objekt}$$

$$c \approx 300000\ km/s$$

Für das Vakuum ist *lightspeed*=1, alle anderen Werte müssen darüber liegen. Nach dem Brechungsgesetz von Snellius ist die Richtung des refraktierten Strahls:

$$T = \left(\eta \cdot (N \bullet L) - \sqrt{1 - \eta^2 \cdot \left(1 - (N \bullet L)^2\right)}\right) \cdot N - \eta \cdot L$$

Im Code habe ich diese etwas komplexere Formel aufgeteilt: zuerst das Skalarprodukt von N und L (`NdotL`), danach den Wurzelausdruck (`Root`), anschließend die gesamte Klammer (`Bracket`) und daraus `T`. Erneut habe ich mit den bekannten Problemen zu kämpfen und setze dagegen wieder die gleichen Mittel ein: der Ursprung des refraktierten Strahls wird von der Hülle weg verschoben und die Normale invertiert, falls notwendig.

Der hier abgedruckte Code ist Bestandteil von `rayTrace` (siehe auch Beschreibung bei der Reflektion):

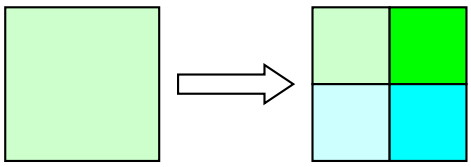
```
// refraction
if (material.opacity < 1)
{
    double lightspeedratio = r.getLightspeed() / material.lightspeed;

    double refracted_angle     = -dotProduct(closestNormal, r.getDirection());
    if (refracted_angle < 0)
    {
        closestNormal   *= -1;
        lightspeedratio = 1/lightspeedratio;
    }

    // compute refraction vector (its direction)
    const Vector L       = -r.getDirection();
    const double NdotL   = dotProduct(closestNormal, L);
    // Snell's Law (splitted up for better readibility)
    const double Root    = 1 - lightspeedratio*lightspeedratio*(1 - NdotL*NdotL);
```

```
        const double Bracket = lightspeedratio*NdotL - sqrt(Root);
        const Vector T       = Bracket*closestNormal - lightspeedratio*L;

        // slight shift of new ray origin
        Vector T_origin      = closestPoint;
        if (NdotL < 0)
            T_origin += closestNormal*10*INTERSECTION_OFFSET;
        else
            T_origin -= closestNormal*10*INTERSECTION_OFFSET;

        const Ray refracted_ray(T_origin, T, material.lightspeed);
                                //r.getDirection(), material.lightspeed);

        // get color
        const Color refracted_color = rayTrace(refracted_ray, nDepth) * (1-material.opacity);

        // add reflective part
        color *= material.opacity;
        color += refracted_color;
    }
```

Progressive View

Das schrittweise Aufbauen des Bildes kann auf mehreren Wegen geschehen. Mein Verfahren beruht darauf, dass es die Primärstrahlen in einem immer feiner werdenden Raster in die Szene aussendet. Schon nach dem ersten Durchlauf lässt sich das Bild in seinen Grundzügen erahnen.

Das Raster wird von der Variablen `detail_` bestimmt. Sie legt fest, welchen Flächeninhalt in Pixel ein Quadrat hat, das einem Ray zugeordnet wird. Der Ausgangswert muss eine Zweierpotenz sein, ich halte 2^8 für angemessen, dann beträgt die initiale Blöckchengröße 8x8. Diese Seitenlänge halbiert sich mit jedem Durchlauf, das Bild ist demzufolge nach 8 Raytracer-Passes komplett aufgebaut.



In einer Verfeinerung entspricht die Farbe des letzten Durchlaufes dem oberen linken Block in neuem Durchlauf. Da es unsinnig wäre, diesen neu zu berechnen, speichere ich das komplette Bild im Array `image_` ab. Diese Vorgehensweise erspart mir ein Viertel der Rechenzeit. Um erkennen zu können, welche Pixel noch nicht berechnet wurde, erfolgt zu Beginn eine Initialisierung von `image_` mit –1.

Zur Darstellung in OpenGL wird der aus den anderen Übungsaufgaben bekannt Programmrahmen mit `CGApplication` benutzt. Als Projektionsmethode setze ich eine orthogonale Darstellung ein, die in ihren Abmessungen exakt dem zu berechnenden Bild entspricht:

```
// no perspective drawing, just pure 2D display
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, imagex_-1, 0, imagey_-1);
```

Es erscheint mir am einfachsten, die Bildpunkte nicht durch (z.T. vergrößerte) Pixel, sondern durch Rechtecke zu erzeugen, die die entsprechende Kantenlänge `size` haben:

```
glRectd(j-size, imagey_-(i-size), j+size, imagey_-(i+size));
```

Je detailreicher ein Durchlauf ist, desto länger beträgt die dafür erforderliche Rechenzeit. Gerade die letzten Passes benötigen mehrere Sekunden. Damit der Benutzer trotzdem den Berechnungsfortschritt erkennen kann, verzichte ich auf Double-Buffering und überschreibe das bisherige Bild ohne vorheriges Löschen.

Der Hauptteil der progressive-view-Darstellung erfolgt in `createImage`. Da dort mehrere Aufgaben erledigt werden, habe ich nur die wesentlichen Zeilen kontrastreich hervorgehoben:

```
void CGRayTracer::createImage(int width, int height) {
    // convert field-of-view angle to rad
    const double fovy_rad = camera_->fovy * PI/180;

    // look vector
    const Vector Look  = camera_->to - camera_->from;

    // delta angles
    const double up_angle_delta    = fovy_rad / height;
    const double right_angle_delta = fovy_rad / height;//width;

    // normalized vectors that describe the view plane (right handed !)
    const Vector Right = camera_->up.normalized() * Look.normalized();
    const Vector Up    = Right * Look.normalized();

    // image
    const double widthoffset  = imagex_/(double)width;
    const double heightoffset = imagey_/(double)height;
    const double pixelsize = detail_;
```

```
    for(int i=0; i<height; i++)
    {
        for(int j=0; j<width; j++)
        {
            // create rays for each pixel and send through scene

            // skip if already calculated
            const int truex = int(j*widthoffset);
            const int truey = int(i*heightoffset);
            Color& clrPixel = image_[truey*imagex_+truex];
            if (clrPixel[0] >= 0)
            {
                // draw already calculated pixels
#ifndef DONT_DRAW
                glColor3dv(clrPixel.rep());
                glRectd(truex, height_-truey,
                        truex+pixelsize, height_-(truey+pixelsize));
#endif
                continue;
            }

            // angles
            const double up_angle    = (i-height/2.0) * up_angle_delta;
            const double right_angle = (j-width /2.0) * right_angle_delta;

            // corresponding vectors
            const Vector CurrentUp    = Up    * abs(Look) * tan(up_angle);
            const Vector CurrentRight = Right * abs(Look) * tan(right_angle);

            // put it all together
            const Vector LookAt = Look + CurrentUp + CurrentRight;

            // ray tracing
            clrPixel = rayTrace(Ray(camera_->from, LookAt));

            // draw pixel
#ifndef DONT_DRAW
            glColor3dv(clrPixel.rep());
            glRectd(truex, height_-truey,
                    truex+pixelsize, height_-(truey+pixelsize));
#endif
        }
    }
}
```

Quellcode

triangle.h:

```
#ifndef _TRIANGLE_H
#define _TRIANGLE_H

#include "shape.h"

//- Triangle
class Triangle : public Shape {
//. Triangle Class.
public:
    //- Triangle
    Triangle();
    Triangle(const Vector& v0, const Vector& v1, const Vector& v2);

    //- set/getVertex, set/GetNormal, triangleNormal
    const Vector& getVertex(int i) const;
    void setVertex(int i,const Vector& v);
    const Vector& getNormal(int i) const;
    void setNormal(int i,const Vector& n);
    const Vector& triangleNormal() const;

    void normalizeNormals();


    // intersect
    virtual bool intersect(const Ray& r,Vector& p,Vector& n, double& d) const;


    //- operator>>
    friend  istream& operator>>(istream&, Triangle&);

private:
    Vector v_[3];
    Vector n_[3];
    Vector triNrm_;
};


inline const Vector& Triangle::getVertex(int i) const { return v_[i]; }
inline void Triangle::setVertex(int i,const Vector& v) { v_[i] = v; }
inline const Vector& Triangle::getNormal(int i) const { return n_[i]; }
inline void Triangle::setNormal(int i,const Vector& n) { n_[i] = n; }
inline const Vector& Triangle::triangleNormal() const { return triNrm_;}


#endif // _TRIANGLE_H
```

triangle.cpp:

```
#include "triangle.h"
#include <iostream.h>
#include <assert.h>
#include <algorithm>

Triangle::Triangle() { }

Triangle::Triangle(const Vector& v0,const Vector& v1, const Vector& v2) {
    v_[0] = v0;
    v_[1] = v1;
    v_[2] = v2;
    triNrm_ = makeCWTriangleNormal(v_[0],v_[1],v_[2]);
    n_[0] = n_[1] = n_[2] = triNrm_;
}


void Triangle::normalizeNormals() {
    n_[0].normalize();
    n_[1].normalize();
    n_[2].normalize();
```

```
}

static inline int MeshMaxIndex(double a, double b, double c) {
    return((a > b) ? ((a > c) ? 0 : 2) : ((b > c) ? 1 : 2));
}

bool Triangle::intersect(const Ray& ray,Vector& point,Vector& normal, double& distance) const
{
    // edges of the triangle
    const Vector edge1 = v_[1] - v_[0];
    const Vector edge2 = v_[2] - v_[0];

    const Vector p    = ray.getDirection() * edge2;
    const double detA = dotProduct(p, edge1);

    // det(A) != 0
    if (detA == 0)
        return false;

    const double f = 1/detA;
    const Vector s = ray.getOrigin() - v_[0];
    const double u = f * dotProduct(p, s);

    // first barycentric value must be in [0,1]
    if (u<0 || u>1)
        return false;

    const Vector q = s*edge1;
    const double v = f * dotProduct(q, ray.getDirection());

    // second barycentric value must be in [0,1]
    if (v<0 || v>1 || u+v>1)
        return false;

    const double t = f * dotProduct(q, edge2);

    // object on the wrong side of the ray ?
    if (t <= 0)
        return false;


    // intersection found, barycentric coordinates are (u,v,t)
    point  = ray.getOrigin() + t*ray.getDirection();

    normal = triNrm_;
    // invert normal to allow the triangle being viewed from its back side
    if (dotProduct(normal, ray.getDirection()) > 0)
        normal = -normal;

    distance = t;//abs(point-ray.getOrigin());

    return true;
}

//
// IO
//

istream& operator>>(istream& s, Triangle& u) {
    Vector t;
    for (int i = 0; i < 3; i++) {
        if (!(s >> t[0] >> t[1] >> t[2])) {
            assert(0);
        }
        u.v_[i] = t;
    }
    u.triNrm_ = makeCWTriangleNormal(u.v_[0],u.v_[1],u.v_[2]);
    u.n_[0] = u.n_[1] = u.n_[2] = u.triNrm_;
    return s;
}
```

sphere.h:

```
#ifndef _SPHERE_H
#define _SPHERE_H

#include "shape.h"

//- Sphere
class Sphere : public Shape {
//. Sphere Class.
public:
    //- Sphere
    Sphere(const Vector& center, double radius);

    const Vector& getCenter() const;
    void setCenter(const Vector& c);

    double getRadius() const;
    void setRadius(double r);

    // intersect
    virtual bool intersect(const Ray& r,Vector& p,Vector& n, double& d) const;

private:
    Vector center_;
    double radius_;
};

#endif // _Sphere_H
```

sphere.cpp:

```
#include "sphere.h"

Sphere::Sphere(const Vector& c, double r) : center_(c), radius_(r) {
}


const Vector& Sphere::getCenter() const { return center_; }

void Sphere::setCenter(const Vector& v) { center_ = v; }

double Sphere::getRadius() const { return radius_; }

void Sphere::setRadius(double r) { radius_ = r; }



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

bool Sphere::intersect(const Ray& ray,Vector& point,Vector& normal, double& distance) const {
    // ray-sphere intersection

    // ray direction
    const Vector direction(ray.getDirection());
    const double i = direction[0];
    const double j = direction[1];
    const double k = direction[2];

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

    // sphere's center
    const double l = center_[0];
    const double m = center_[1];
```

```
    const double n = center_[2];
    const double r = radius_;

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

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

    // no intersection
    if (D < 0)
        return false;

    // nearest ray intersection
    double t;

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

    // all intersections behind the origin ?
    if (t <= 0.001)
        return false;


    point  = ray.getOrigin() + t*ray.getDirection();
    normal = (point - center_)/r; // divison by "r" normalizes the vector
    distance = t;//abs(point-ray.getOrigin());

    return true;
}
```

box.h:

```
#ifndef _BOX_H
#define _BOX_H

#include "shape.h"

//- Box
class Box : public Shape {
//. 3D Box Class.
public:
    Box(const Vector& llf, const Vector& urb);

    const Vector& getLLF() const;
    void setLLF(const Vector& llf);
    const Vector& getURB() const;
    void setURB(const Vector& n);

    // intersect
    virtual bool intersect(const Ray& r,Vector& p,Vector& n, double& d) const;

private:
    Vector llf_;
    Vector urb_;
};

#endif // _Box_H
```

box.cpp:

```
#include "box.h"

Box::Box(const Vector& llf,const Vector& urb) : llf_(llf), urb_(urb) {
}


const Vector& Box::getLLF() const { return llf_; }

void Box::setLLF(const Vector& v) { llf_ = v; }

const Vector& Box::getURB() const { return urb_; }

void Box::setURB(const Vector& n) { urb_ = n; }

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

bool Box::intersect(const Ray& ray,Vector& point,Vector& normal, double& distance) const {
    const static Vector NormalRight(+1,0,0);
    const static Vector NormalLeft (-1,0,0);
    const static Vector NormalUp  (0,+1,0);
    const static Vector NormalDown(0,-1,0);
    const static Vector NormalFront(0,0,+1);
    const static Vector NormalBack (0,0,-1);

    // ray origin
    const Vector origin = ray.getOrigin();
    // ray direction
    const Vector direction = ray.getDirection();


    // ray enters box (-infinity)
    double tnear = -999999999999;
    // ray leaves box (+infinity)
    double tfar  = +999999999999;


    ////////////////////////////////////////////////////
    // x
    const double& ox = origin[0];
```

```
    const double& rx = direction[0];

    // parallel to box ?
    if (rx == 0)
        if (ox < llf_[0] || ox > urb_[0])
            return false;

    // intersections
    const double tx1 = (llf_[0] - ox) / rx;
    const double tx2 = (urb_[0] - ox) / rx;

    // adjust tnear and tfar
    if (tx2 > tx1 && tx1 > tnear)
    {
        tnear  = tx1;
        normal = NormalLeft;
    }
    if (tx1 > tx2 && tx2 > tnear)
    {
        tnear = tx2;
        normal = NormalRight;
    }

    // ray doesn't hit box ?
    tfar = min(tfar, max(tx1, tx2));
    if (tnear > tfar || tfar < 0)
        return false;


    ////////////////////////////////////////////////////
    // y
    const double& oy = origin[1];
    const double& ry = direction[1];

    // parallel to box ?
    if (ry == 0)
        if (oy < llf_[1] || oy > urb_[1])
            return false;

    // intersections
    const double ty1 = (llf_[1] - oy) / ry;
    const double ty2 = (urb_[1] - oy) / ry;

    // adjust tnear and tfar
    if (ty2 > ty1 && ty1 > tnear)
    {
        tnear  = ty1;
        normal = NormalDown;
    }
    if (ty1 > ty2 && ty2 > tnear)
    {
        tnear  = ty2;
        normal = NormalUp;
    }

    // ray doesn't hit box ?
    tfar = min(tfar, max(ty1, ty2));
    if (tnear > tfar || tfar < 0)
        return false;


    ////////////////////////////////////////////////////
    // z
    const double& oz = origin[2];
    const double& rz = direction[2];

    // parallel to box ?
    if (rz == 0)
        if (oz < llf_[2] || oz > urb_[2])
            return false;

    // intersections
    const double tz1 = (llf_[2] - oz) / rz;
    const double tz2 = (urb_[2] - oz) / rz;

    // adjust tnear and tfar
```

```
    if (tz2 > tz1 && tz1 > tnear)
    {
        tnear  = tz1;
        normal = NormalBack;
    }
    if (tz1 > tz2 && tz2 > tnear)
    {
        tnear  = tz2;
        normal = NormalFront;
    }

    // ray doesn't hit box ?
    tfar = min(tfar, max(tz1, tz2));
    if (tnear > tfar || tfar < 0)
        return false;


    // intersection finally found !!!
    point  = ray.getOrigin() + tnear*ray.getDirection();
    distance = tnear;//abs(point-ray.getOrigin());

    // done !
    return true;
}
```

cgraytracer.h:

```
#ifndef CG_HEIGHTFIELD_H
#define CG_HEIGHTFIELD_H

#include "util.h"
#include "cgapplication.h"
#include <time.h>

// Ray Tracer Class
class CGRayTracer : public CGApplication {
public:

    // CGRayTracer, ~CGRayTracer
    CGRayTracer(unsigned int x, unsigned int y);
    virtual ~CGRayTracer();

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

    // createImage
    void createImage(int width, int height);
    void writeImage(char* fileName);

    // use shadows
    bool shadows_;
    // max recursion depth
    int maxDepth_;
    int nTests_;

private:
    // phongIlluminationColor
    Color phongIlluminationColor(
        const Vector& point,
        const Vector& normal,
        Material* material);

    // writePPMPixel
    void writePPMPixel(const Color& c);

    // rayTrace
    Color rayTrace(const Ray& r, int nDepth=1);

    // add an object to the scene
    void AddObject(Shape* shape, Material* material);

    // shape and material array
    int shapeSize_;
    Shape** shape_;
    Material** material_;

    // light array
    int lightSize_;
    Light** light_;

    // camera
    Camera* camera_;

    // dimension of viewport
    int width_;
    int height_;

    // memory buffer
    unsigned int imagex_, imagey_;
    Color* image_;
    int detail_;
    bool antialias_;

    // file
    ofstream* out_;

    // timer
    clock_t start_;
```

```
};

#endif // CG_HeightField_H
```

cgraytracer.cpp:

```
#include "cgraytracer.h"
#include <stdlib.h>
#include <time.h>


// watermark
#pragma comment(exestr, "(C)2001-2002 by Stephan Brumme")


// my parent's GPU driver is much too slow ... definitely !!!
//#define DONT_DRAW


static double PI = 3.1415926;
static double INTERSECTION_OFFSET = 0.0001;
static Color  clrBackground(0.05, 0.05, 0.1);
inline double frand() { return double(rand()) / (RAND_MAX); }

//
//  CGRayTracer Application
//


CGRayTracer::CGRayTracer(unsigned int x, unsigned int y) {

    // general settings
    // use shadows ?
    shadows_ = true;
    // recursion depth
    maxDepth_ = 7;
    // no. of intersection tests
    nTests_ = 0;
    // initial detail level (no. of pixels per ray)
    detail_ = 1<<8;
    antialias_ = false;

    // allocate memory to store the image
    imagex_ = x;
    imagey_ = y;
    image_  = new Color[(imagex_+1)*(imagey_+1)];
    for (unsigned int i=0; i<imagey_; i++)
        for (unsigned int j=0; j<imagex_; j++)
            image_[i*imagex_+j] = Color(-0.1,0,0);

    // number of shapes
    shapeSize_ = 0; // incremented by AddObject

    // create shape and material array
    shape_ = new Shape*[100];
    material_ = new Material*[100];



    // background
    AddObject(new Box(Vector(-1000, -1000, 4), Vector(1000, 1000, 4.1)),
              new Material (0.2, 0.5, 0.0, 10, 1, 1, Color(0.5,0.5,0.9)));

    // lower surface
    AddObject(new Box(Vector(-1, -0.1, -1), Vector(1, 0, 1)),
              new Material (0.2, 0.5, 0.8, 10, 1, 1, Color(0.1,0.4,0.1)));

    // main sphere
    AddObject(new Sphere (Vector(0,0,0), 0.5),
              new Material (0.2, 0.5, 0.7, 32, 0.3, 1, Color(0.9,0.9,0.9)));

    // pyramid
    AddObject(new Triangle(Vector(0, 0, -0.4), Vector(0.4, 0, 0), Vector(0, 0.4, 0)),
              new Material (0.2, 0.5, 0.7, 32, 1, 1, Color(0.8,0.8,0.8)));
    AddObject(new Triangle(Vector(0, 0, -0.4), Vector(-0.4, 0, 0), Vector(0, 0.4, 0)),
```

```
                new Material (0.2, 0.5, 0.7, 32, 1, 1, Color(0.8,0.8,0.8)));
    AddObject(new Triangle(Vector(0, 0, +0.4), Vector(0.4, 0, 0), Vector(0, 0.4, 0)),
                new Material (0.2, 0.5, 0.7, 32, 1, 1, Color(0.8,0.8,0.8)));
    AddObject(new Triangle(Vector(0, 0, +0.4), Vector(-0.4, 0, 0), Vector(0, 0.4, 0)),
                new Material (0.2, 0.5, 0.7, 32, 1, 1, Color(0.8,0.8,0.8)));

    // sphere on top of the pyramid
    AddObject(new Sphere (Vector(0,0.45,0), 0.08),
                new Material (0.2, 0.5, 0.99, 32, 1, 1, Color(1.0,0.3,0.3)));

    // two arcs
    const int ARCSIZE = 14;
    for (i=0; i<ARCSIZE; i++)
    {
        const double arc_angle = ((i+0.5)*180.0 / (ARCSIZE-1)) * PI/180;

        // arc 1
        AddObject(new Sphere (Vector(+cos(arc_angle)*0.6, sin(arc_angle)*0.8,
cos(arc_angle)*0.6), 0.05),
                  new Material (0.2, 0.5, 0.7, 32, 0.3, 1.3, Color(0.8,0.8,0.8)));
        AddObject(new Sphere (Vector(+cos(arc_angle)*0.6, sin(arc_angle)*0.8,
cos(arc_angle)*0.6), 0.01),
                  new Material (0.2, 0.5, 0.7, 32, 1, 1, Color(0.8,0.8,0.8)));

        // arc 2
        AddObject(new Sphere (Vector(-cos(arc_angle)*0.6, sin(arc_angle)*0.8,
cos(arc_angle)*0.6), 0.05),
                  new Material (0.2, 0.5, 0.7, 32, 0.3, 1.3, Color(0.8,0.8,0.8)));
        AddObject(new Sphere (Vector(-cos(arc_angle)*0.6, sin(arc_angle)*0.8,
cos(arc_angle)*0.6), 0.01),
                  new Material (0.2, 0.5, 0.7, 32, 1, 1, Color(0.8,0.8,0.8)));
    }

    // ring
    const int RINGSIZE = 25;
    for (i=0; i<RINGSIZE; i++)
    {
        const double ring_angle = (i * 360 / RINGSIZE) * PI/180;
        AddObject(new Sphere (Vector(cos(ring_angle)*0.5, 0.07, sin(ring_angle)*0.5), 0.03),
                  new Material (0.2, 0.5, 0.9, 32, 1, 1, Color(0.2,0.2,0.2)));
    }


    // set camera
    camera_ = new Camera(Vector(0.4, 2, -2), // from
                         Vector(0, 0, 0) , // to
                         Vector(0, 1, 0),  // up
                         90);              // fov

    // set light sources
    lightSize_ = 2;
    light_ = new Light*[lightSize_];
    light_[0] = new Light( Vector(2, 2.3, -1.5), 0.4, 0.1 );
    light_[1] = new Light( Vector(-1, 1, -1.5), 0.6, 0.1 );


    // INITIAL TEST SCENE, NOT USED ANYMORE !!!

    // main sphere
/*  AddObject(new Sphere (Vector(0,0,0), 0.4),
                new Material (0.2, 0.5, 0.7, 32, 0.3, 1, Color(0.8,0.8,0.8)));
    AddObject(new Sphere (Vector(0,0.1,0), 0.1),
                new Material (0.2, 0.5, 0.7, 32, 1, 1, Color(0.8,0.8,0.8)));

    // upper right sphere
//  AddObject(new Sphere (Vector(0.5,0.5,-0.5), 0.15),
//              new Material (0.2, 0.6, 0.8, 100, 0.5, 1.3, Color(0.6,0.6,1.0)));

    // ground
    AddObject(new Box (Vector(-0.7,-0.2,-0.7), Vector(0.7,-0.15,0.7)),
                new Material (0.2, 0.5, 0.5, 10, 1, 1, Color(0.5,0.5,0.9)));

    // upper right box
    AddObject(new Box (Vector(-0.05,0.3,-0.6), Vector(0.1,0.4,-0.4)),
                new Material (0.2, 0.5, 0.5, 10, 1, 1, Color(0.5,0.9,0.5)));
```

```
    // upper box
    AddObject(new Box (Vector(-0.25,0.3,-0.4), Vector(-0.15,0.7,-0.3)),
              new Material (0.2, 0.3, 0.5, 10, 1, 1, Color(0.5,0.5,0.5)));

    // back plane
    AddObject(new Triangle(Vector (-2, -2, 0.7), Vector ( 2,  1.8, 0.7), Vector ( 2, -2,
0.7)),
              new Material (0.2, 0.3, 0.7, 10, 1, 1, Color(0.5,0.5,0.5)));

    // bottom plane
    AddObject(new Box (Vector(-1, -1, -1), Vector(1,-0.9,1)),
              new Material (0.2, 0.3, 0.5, 10, 1, 1, Color(0.8,0.5,0.5)));
    AddObject(new Triangle(Vector(-0.8, -0.5, -1), Vector(0.8,-0.5,1), Vector(0.8,-0.5,-1)),
              new Material (0.2, 0.3, 0.5, 10, 1, 1, Color(0.5,0.9,0.5)));

    // mid-scene triangles
    AddObject(new Triangle(Vector (0.4, -0.1, -0.6), Vector (0.6, -0.1, -0.6), Vector (0.6, -
0.1, -0.4)),
              new Material (0.2, 0.3, 0.7, 10, 0.8, 1, Color(0.5,0.9,0.5)));
    AddObject(new Triangle(Vector (0.5, 0.0, -0.7), Vector (0.7, 0.0, -0.7), Vector (0.7, 0.0,
-0.5)),
              new Material (0.2, 0.3, 0.7, 10, 0.5, 1, Color(0.5,0.9,0.5)));

    // back plane
//  AddObject(new Box(Vector(-2, -2, 1.2), Vector(2, 2, 1.3)),
//            new Material (0.1, 0.2, 0.6, 50, 1, 1, Color(0.9,0.9,0.9)));

    // sphere on da right side
    AddObject(new Sphere (Vector(0.6,-0.05,-0.1), 0.05),
              new Material (0.2, 0.2, 0.8, 100, 1, 1, Color(1,0.5,0.5)));
*/
}


CGRayTracer::~CGRayTracer() {
    // clean up
    for (int i = 0; i < shapeSize_; i++) {
        delete shape_[i];
        delete material_[i];
    }
    for (int j = 0; j < lightSize_; j++) {
        delete light_[j];
    }

    delete[] shape_;
    delete[] material_;
    delete[] light_;
}

void CGRayTracer::AddObject(Shape* shape, Material* material)
{
    shape_[shapeSize_] = shape;
    material_[shapeSize_] = material;

    shapeSize_++;
}

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

Color CGRayTracer::phongIlluminationColor(const Vector& point, const Vector& normal, Material*
material) {
    // calculate Phong illumination

    // code is a slightly modified copy of my Phong homework

    // define intensity variables
    double dIntensityAmbient  = 0;
    double dIntensityDiffuse  = 0;
    double dIntensitySpecular = 0;

    // N = normal
```

```
    const Vector N_norm = normal.normalized();
    // V = view vector
    const Vector V = camera_->from - point;
    const Vector V_norm = V.normalized();

    // process all light sources
    for (int nLight=0; nLight < lightSize_; nLight++)
    {
        // get a single light source
        const Light* const light = light_[nLight];

        // ALWAYS ambient intensity, even if shadowed
        dIntensityAmbient  += light->ambient;

        // L = light vector
        const Vector L = light->pos - point;
        const Vector L_norm = L.normalized();

        // N*L
        const double NdotL = dotProduct(N_norm, L_norm);

        // intensity reaching the surface (attenuation)
        const double dLightDistance = abs(L);
        double dMaxIntensity = 1.0 / (light->att_constant + dLightDistance*light->att_linear
                                      + dLightDistance*dLightDistance*light->att_quadric);
        if (dMaxIntensity > 1)
            dMaxIntensity = 1;

        // get shadow
        if (shadows_)
        {
            // shadow ray must not start at the object's surface !
            // (else we get in trouble with intersection problems)
            const Ray shadowray(point+(light->pos - point)*INTERSECTION_OFFSET, light->pos -
point);
            bool shadow = false;

            Shape** ppShape = shape_;
            for (int shape=0; shape<shapeSize_; shape++)
            {
                // get values of each intersection
                double distance;
                static Vector _normal, _point;

                // does the ray intersect the object ?
                shadow = shape_[shape]->intersect(shadowray, _point, _normal, distance);
                nTests_++;

                // intersection between light source and surface point ?
                if (shadow && distance < dLightDistance)
                {
                    dMaxIntensity *= 1-material_[shape]->opacity;
                    if (dMaxIntensity > 0.01)
                        shadow = false;
                    else
                        // 100% shadow
                        break;
                }
            }
            // shadowed ! no diffuse or specular light possible
            if (shadow)
                continue;
        }

        // R*V = (2*N*(N*L)-L)*V
        const double RdotV = max(dotProduct((2*NdotL*N_norm - L_norm).normalized(), V_norm),
0);

        // diffuse intensity
        dIntensityDiffuse  += dMaxIntensity * material->kd * NdotL;
        // specular intensity
        dIntensitySpecular += dMaxIntensity * material->ks * pow(RdotV, material->n);
    }

    // return clamped color
    return material->color * min(dIntensityAmbient+dIntensityDiffuse+dIntensitySpecular,1);
```

```
}

Color CGRayTracer::rayTrace(const Ray& r, int nDepth) {
    // Check each shape for intersektion.
    // Return phong color of nearest shape
    // that is hit.

    if (nDepth++ > maxDepth_)
        return clrBackground;

    // shape that we found
    int closestShape = -1;
    // its parameters
    double closestDistance = 99999999;
    Vector closestPoint;
    Vector closestNormal;

    Shape** ppShape = shape_;
    for (int shape=0; shape<shapeSize_; shape++)
    {
        // get values of each intersection
        double distance;
        static Vector normal;
        static Vector point;

        // internal counter
        nTests_++;

        // does the ray intersect the object ?
        if (shape_[shape]->intersect(r, point, normal, distance))
            // closer than any intersection we tested before ?
            if (distance < closestDistance)
            {
                closestDistance = distance;
                closestShape    = shape;
                closestPoint    = point;
                closestNormal   = normal;
            }
    }

    // no intersection found
    if (closestShape < 0)
        return clrBackground;
    Material& material = *(material_[closestShape]);


    // object itself
    Color color = phongIlluminationColor(closestPoint, closestNormal, &material);

    // reflection
    if (material.ks >= 0.001)
    {
        // get reflected ray
        double reflected_angle     = -dotProduct(closestNormal, r.getDirection());
        if (reflected_angle < 0)
        {
            closestNormal   *= -1;
            reflected_angle *= -1;
        }

        const Vector reflected_direction = r.getDirection() + 2*reflected_angle*closestNormal;

        // reflected ray must not start at the object's surface !
        // (else we get in trouble with intersection problems)
        const Ray reflected_ray(closestPoint+closestNormal*INTERSECTION_OFFSET,
                                reflected_direction, material.lightspeed);

        // get color
        const Color reflected_color = rayTrace(reflected_ray, nDepth) * material.ks *
material.opacity;

        // add reflective part
        color += reflected_color;
    }

    // refraction
```

```
    if (material.opacity < 1)
    {
        double lightspeedratio = r.getLightspeed() / material.lightspeed;

        double refracted_angle     = -dotProduct(closestNormal, r.getDirection());
        if (refracted_angle < 0)
        {
            closestNormal   *= -1;
            lightspeedratio = 1/lightspeedratio;
        }

        // compute refraction vector (its direction)
        const Vector L       = -r.getDirection();
        const double NdotL   = dotProduct(closestNormal, L);
        // Snell's Law (splitted up for better readibility)
        const double Root    = 1 - lightspeedratio*lightspeedratio*(1 - NdotL*NdotL);
        const double Bracket = lightspeedratio*NdotL - sqrt(Root);
        const Vector T       = Bracket*closestNormal - lightspeedratio*L;

        // slight shift of new ray origin
        Vector T_origin      = closestPoint;
        if (NdotL < 0)
            T_origin += closestNormal*10*INTERSECTION_OFFSET;
        else
            T_origin -= closestNormal*10*INTERSECTION_OFFSET;

        const Ray refracted_ray(T_origin, T, material.lightspeed);
                                //r.getDirection(), material.lightspeed);

        // get color
        const Color refracted_color = rayTrace(refracted_ray, nDepth) * (1-material.opacity);

        // add reflective part
        color *= material.opacity;
        color += refracted_color;
    }

    // clamp color
    color[0] = min(color[0], 1);
    color[1] = min(color[1], 1);
    color[2] = min(color[2], 1);
//  color[0] = max(color[0], 0);
//  color[1] = max(color[1], 0);
//  color[2] = max(color[2], 0);

    return color;
}

void CGRayTracer::writePPMPixel(const Color& c) {
    // write pixel to file
    (*out_) << ((unsigned char)(c[0]*255))
            << ((unsigned char)(c[1]*255))
            << ((unsigned char)(c[2]*255));
}

void CGRayTracer::writeImage(char* fileName) {

    cout << "saving " << fileName << " - ";
    // create output stream
    out_ = new ofstream(fileName);

    // write PPM header
#ifdef _MSC_VER
    (*out_) << binary;
#endif
    (*out_) << "P6" << endl
            << width_ << " " << height_ << endl
            << 255 << endl;

    // write all pixel
    for(unsigned int i=0; i<imagey_; i++)
        for(unsigned int j=0; j<imagex_; j++)
            writePPMPixel(image_[i*imagex_+j]);

    // delete output stream
    delete out_;
```

```
    cout << "done." << endl;
}


void CGRayTracer::createImage(int width, int height) {
    // convert field-of-view angle to rad
    const double fovy_rad = camera_->fovy * PI/180;

    // look vector
    const Vector Look  = camera_->to - camera_->from;

    // delta angles
    const double up_angle_delta    = fovy_rad / height;
    const double right_angle_delta = fovy_rad / height;//width;

    // normalized vectors that describe the view plane (right handed !)
    const Vector Right = camera_->up.normalized() * Look.normalized();
    const Vector Up    = Right * Look.normalized();

    // image
    const double widthoffset  = imagex_/(double)width;
    const double heightoffset = imagey_/(double)height;
    const double pixelsize = detail_;

    for(int i=0; i<height; i++)
    {
        for(int j=0; j<width; j++)
        {
            // create rays for each pixel and send through scene

            // skip if already calculated
            const int truex = int(j*widthoffset);
            const int truey = int(i*heightoffset);
            Color& clrPixel = image_[truey*imagex_+truex];
            if (clrPixel[0] >= 0)
            {
                // draw already calculated pixels
#ifndef DONT_DRAW
                glColor3dv(clrPixel.rep());
                glRectd(truex, height_-truey,
                        truex+pixelsize, height_-(truey+pixelsize));
#endif
                continue;
            }

            // angles
            const double up_angle    = (i-height/2.0) * up_angle_delta;
            const double right_angle = (j-width /2.0) * right_angle_delta;

            // corresponding vectors
            const Vector CurrentUp    = Up    * abs(Look) * tan(up_angle);
            const Vector CurrentRight = Right * abs(Look) * tan(right_angle);

            // put it all together
            const Vector LookAt = Look + CurrentUp + CurrentRight;

            // ray tracing
            clrPixel = rayTrace(Ray(camera_->from, LookAt));

            // draw pixel
#ifndef DONT_DRAW
            glColor3dv(clrPixel.rep());
            glRectd(truex, height_-truey,
                    truex+pixelsize, height_-(truey+pixelsize));
#endif
        }
    }
}


void CGRayTracer::onInit() {
    // no perspective drawing, just pure 2D display
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, imagex_-1, 0, imagey_-1);
```

```
    glClearColor(clrBackground[0], clrBackground[1], clrBackground[2],1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // start timer
    start_ = clock();
}

void CGRayTracer::onSize(unsigned int newWidth,unsigned int newHeight) {
    width_ = newWidth;
    height_ = newHeight;

    glViewport(0,0, width_-1, height_-1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    onDraw();
}

void CGRayTracer::onKey(unsigned char key) {
    switch (key) {
    case 27: { exit(0); break; }
    case 'a': antialias_ = !antialias_; onDraw(); break;
    }
}

void CGRayTracer::onIdle() {
    if (detail_ > 1)
    {
        // next detail level
        detail_ /= 2;
        createImage(imagex_ / detail_, imagey_ / detail_);

        if (detail_ > 1)
        {
            // detail level completed
            cout << detail_ << "x" << detail_ << " level done" << endl;
        }
        else
        {
            // stop timer
            clock_t finish = clock();
            cout << "... image completed !" << endl;

            writeImage("raytracer.ppm");

            double seconds = (finish - start_)/(double)CLOCKS_PER_SEC;
            // show some statistics
            cout << endl
                 << imagey_ << "x" << imagex_ << " pixels rendered using " << nTests_ << "
intersection tests in "
                 << seconds << " seconds" << endl
                 << "=> " << int(imagex_*imagey_/seconds) << " pixels/sec and "
                 << nTests_/double(imagex_*imagey_) << " tests/pixel (ray depth=" << maxDepth_
<< ")" << endl;

        }
    }
}

void CGRayTracer::onDraw() {
//    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // "pixel" size
    const double size = detail_;

    if (!antialias_ || detail_ > 1)
        // process all pixels
        for (unsigned int i=0; i<imagey_; i++)
            for (unsigned int j=0; j<imagex_; j++)
            {
                // draw only if already calculated
                const Color& color = image_[i*imagex_+j];
                if (color[0] >= 0)
                {
                    glColor3dv(color.rep());
                    glRectd(j, imagey_-i, j+size, imagey_-(i+size));
                }
```

```
            }
    else
        // process all pixels
        for (unsigned int i=0; i<imagey_; i++)
            for (unsigned int j=0; j<imagex_; j++)
            {
                if (i==0 || j==0)
                {
                    glColor3dv(image_[i*imagex_+j].rep());
                    glRectd(j, imagey_-i, j+size, imagey_-(i+size));
                }
                else
                {
                    // weight neighbours
                    const Color color = (image_[(i-1)*imagex_+j] +
                                         image_[(i-1)*imagex_+j-1] +
                                         image_[(i-1)*imagex_+j+1] +
                                       4*image_[ i   *imagex_+j] +
                                         image_[ i   *imagex_+j-1] +
                                         image_[ i   *imagex_+j+1] +
                                         image_[(i+1)*imagex_+j] +
                                         image_[(i+1)*imagex_+j-1] +
                                         image_[(i+1)*imagex_+j+1]) / 12;
                    glColor3dv(color.rep());
                    glRectd(j, imagey_-i, j+size, imagey_-(i+size));
                }
            }
}


// Hauptprogramm
int main(int argc, char* argv[]) {

    unsigned int maxx;
    unsigned int maxy;

    cout << "Bildformat bitte eingeben !" << endl
         << "Breite: ";
    cin  >> maxx;
    cout << "Hoehe : ";
    cin  >> maxy;

    // Erzeuge eine Instanz der Beispiel-Anwendung:
    CGRayTracer sample(maxx, maxy);

    cout << endl
         << "Tastenbelegung:" << endl
         << "ESC        Programm beenden" << endl
//       << "a          Antialiasing ein/aus" << endl
         << endl
         << "Das Bild wird automatisch unter \"raytracer.ppm\" gespeichert." << endl << endl;

    // Starte die Beispiel-Anwendung:

    sample.start("Stephan Brumme, 702544", false, maxx, maxy);

    return 0;
}
```