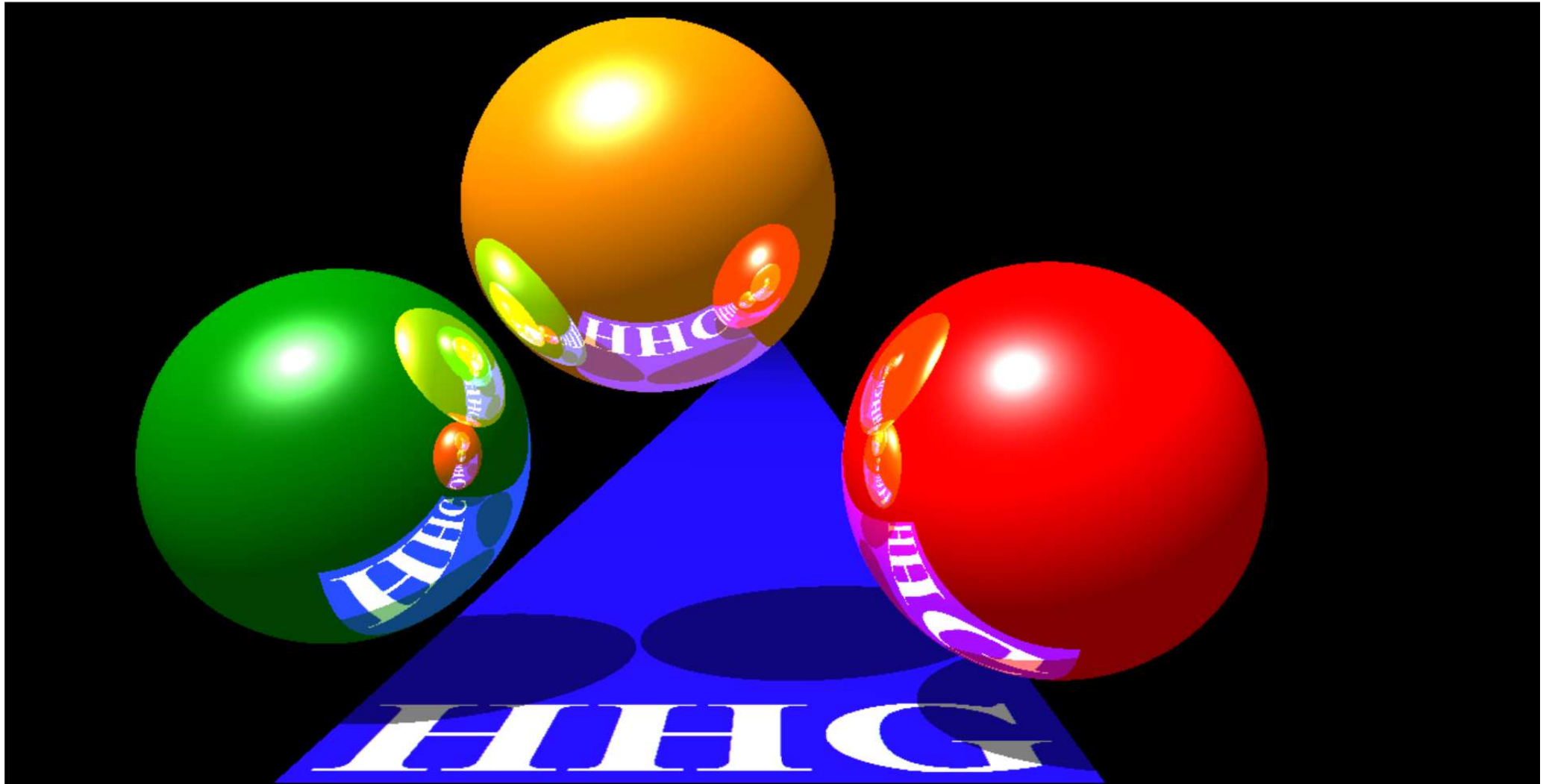


Raytracing

Form1



Ein Foto ist die Abbildung einer 3-dimensionalen Szene auf einer 2-dimensionalen Fläche.

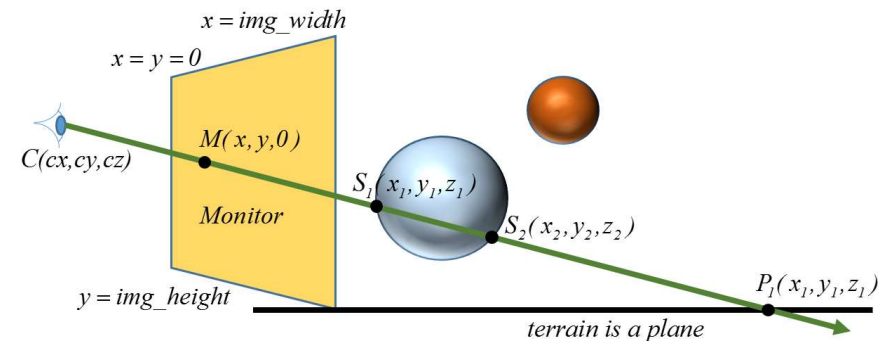
Möchte man dies auf dem Computer nachbilden, bekommt man die Möglichkeit seine Kenntnisse aus der Mathematik der Oberstufe und Physik (Farbenlehre) anzuwenden.

Zum Einsatz kommen das Lösen quadratischer Gleichungen, die Anwendung trigonometrischer Funktionen, die Berechnung von Schnittpunkten zwischen Gerade und Ebene und zwischen Gerade und Kugel, das Skalarprodukt, sowie die Berechnung eines an einem Spiegel reflektierten Strahls. Aus der Physik wissen wir, wie man Licht in seine einzelnen Farbkomponenten zerlegt. Wir setzen sie wieder zusammen, indem wir die drei Komponenten Rot(R), Grün(G) und Blau(B) munter addieren, verstärken und schwächen und sogar auf dem Computer miteinander multiplizieren, um so einen natürlich wirkenden Farbeindruck zu erzeugen.

Unser Vorhaben geht zurück auf Douglas Scott Kay und Turner Whitted.

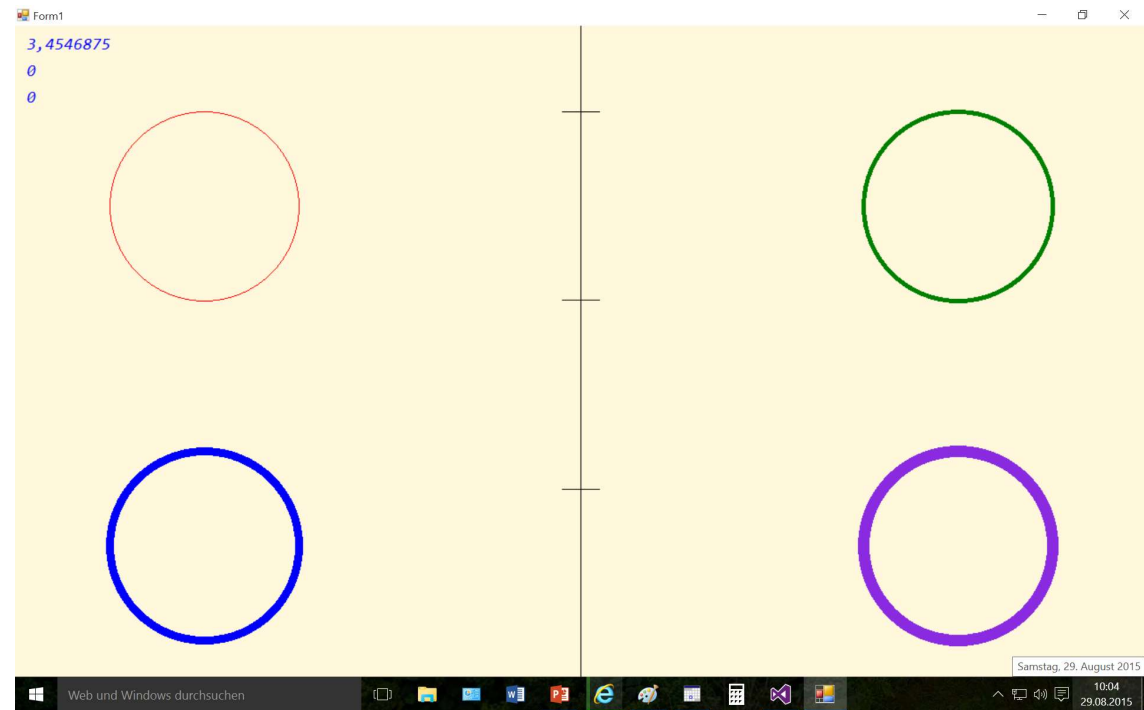
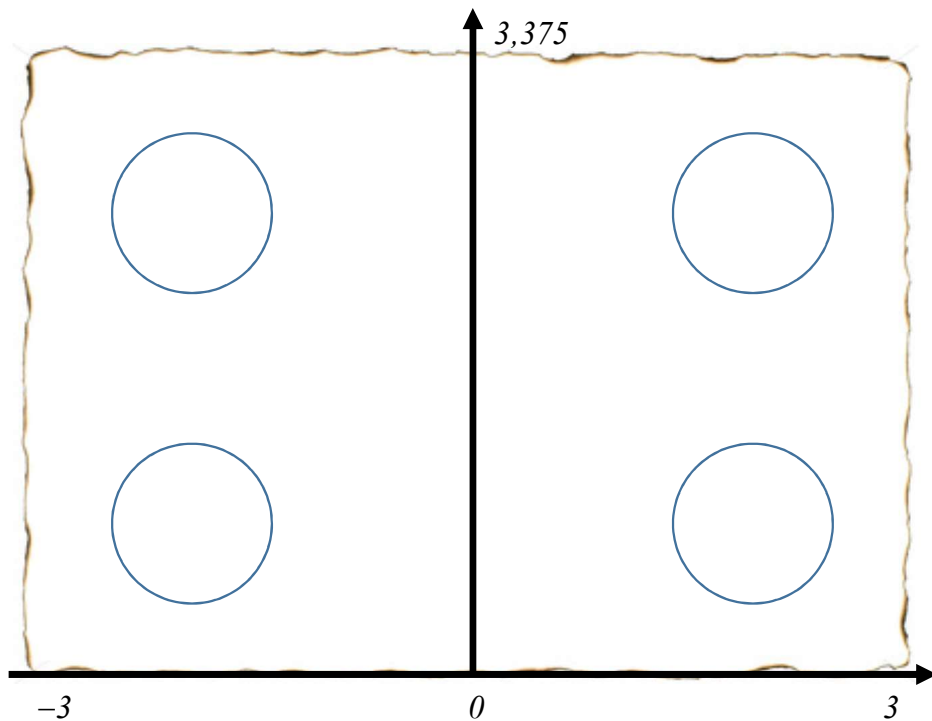
Ein Raytracing Algorithmus hat in Pseudocode die Form:

```
public void raytracer(Graphics g) {  
    //Camera cam  
    for (int x = 0; x < image_width; x++)  
        for (int y = 0; y < image_height; y++) {  
            Ray cam_ray = new Ray(cam.pos, (x,y,0)-cam.pos);  
            Intersection closest_point = getClosestIntersectionPoint(Ray, scene);  
            g.DrawPoint(closest_point.findColor, x, y);  
        }  
}
```



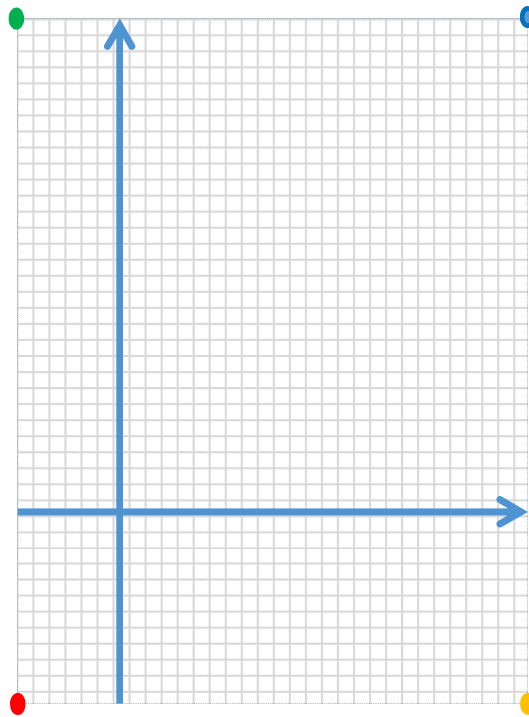
Wie zeichne ich einen Kreis kreisförmig statt elliptisch?

Angenommen der Monitor hat einen $x : y = 16 : 9$ Bildschirm und unsere x -Achse auf dem Papier geht von $x_{\min} = -3$ bis $x_{\max} = 3$; unsere y -Achse soll von $y_{\min} = 0$ bis y_{\max} gehen. Welchen Wert muss y_{\max} haben, damit keine Verzerrung auftritt? Die Antwort ist: $y_{\max} = (x_{\max} - x_{\min}) \cdot \frac{9}{16} = (3 - (-3)) \cdot \frac{9}{16} = \frac{27}{8} = 3,375$. Wähle somit für $y_{\max} = 3,375$ auf dem Papier, damit Kreise kreisförmig bleiben und nicht elliptisch erscheinen.



Vom Papier auf den Bildschirm

$(x / y) = (xmin / ymax)$ $(x / y) = (xmax / ymax)$

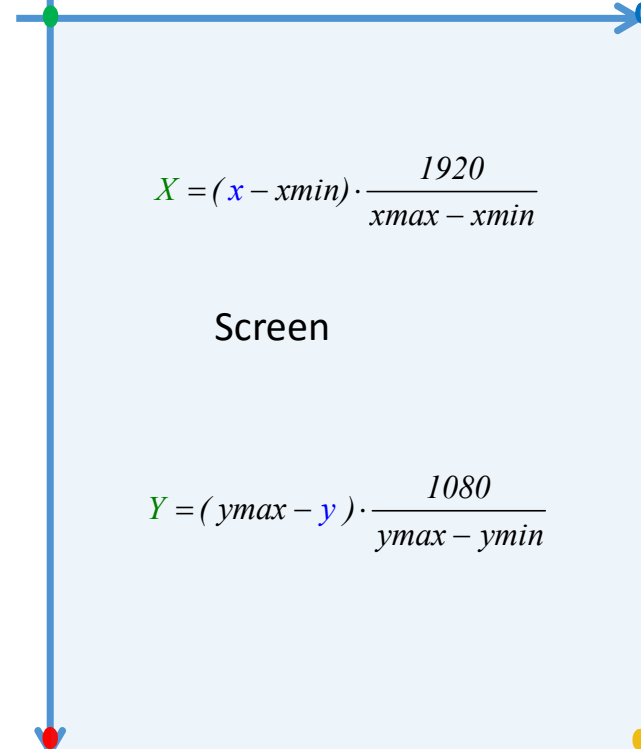


$(x / y) = (xmin / ymin)$ $(x / y) = (xmax / ymin)$

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$



$(X / Y) = (0 / 0)$ $(X / Y) = (1920 / 0)$



$(X / Y) = (0 / 1080)$ $(X / Y) = (1920 / 1080)$

$$X = (x - xmin) \cdot \frac{1920}{xmax - xmin}$$

Screen

$$Y = (ymax - y) \cdot \frac{1080}{ymax - ymin}$$

```
public int xX(double x)
{
    return (int)Math.Round(xFactor * (x - xmin));
}
```

```
public int yY(double y)
{
    return (int)Math.Round(yFactor * (ymax - y));
}
```

*Dieses kleine Testprogramm demonstriert die
Anwendung der verzerrungsfreien Transformation
einer Skizze vom Papier auf den Bildschirm.*

```
public partial class Form1 : Form
{
    private double xmin, xmax, ymin, ymax;
    private double xFactor, yFactor;

    public Form1() {
        InitializeComponent();
        xmin = -3; xmax = 3;
        ymin = 0;
    }

    public int xX(double x) { return (int)Math.Round(xFactor * (x - xmin)); }

    public int yY(double y) { return (int)Math.Round(yFactor * (ymax - y)); }

    public void drawCircle(Graphics g, Color c, int d, double center_x, double center_y, double radius) {
        g.DrawEllipse(new Pen(c,d), xX(center_x - radius), yY(center_y + radius), 2 * xX(xmin + radius), 2 * yY(ymax - radius)); }

    private void Form1_Paint(object sender, PaintEventArgs e) {
        ymax = (xmax - xmin) * this.ClientRectangle.Height / this.ClientRectangle.Width;
        xFactor = this.ClientRectangle.Width / (xmax - xmin);
        yFactor = this.ClientRectangle.Height / (ymax - ymin);
        Graphics g = e.Graphics;
        g.Clear(Color.Cornsilk);
        Font fn = new Font("Consolas", 14, FontStyle.Italic);
        Brush br = new SolidBrush(Color.Blue);
        g.DrawString(ymax.ToString(), fn, br, 10, 10);
        g.DrawString(xX(xmin).ToString(), fn, br, 10, 40);
        g.DrawString(yY(ymax).ToString(), fn, br, 10, 70);
        g.DrawLine(new Pen(Color.Black, 1), xX(0), yY(xmin), xX(0), yY(ymax)); //yAchse
        for(int i = 1; i < ymax; i++)
            g.DrawLine(new Pen(Color.Black, 1), xX(0-0.1), yY(i), xX(0+0.1), yY(i)); //yAchse
        drawCircle(g, Color.Red, 1, -2, 2.5, 0.5); drawCircle(g, Color.Green, 5, 2, 2.5, 0.5);
        drawCircle(g, Color.Blue, 9, -2, 0.7, 0.5); drawCircle(g, Color.BlueViolet, 13, 2, 0.7, 0.5); }
}
```

Die Szene

Welches Bild zeichnet die Kamera auf, wenn alle Beteiligten wie folgt positioniert sind:

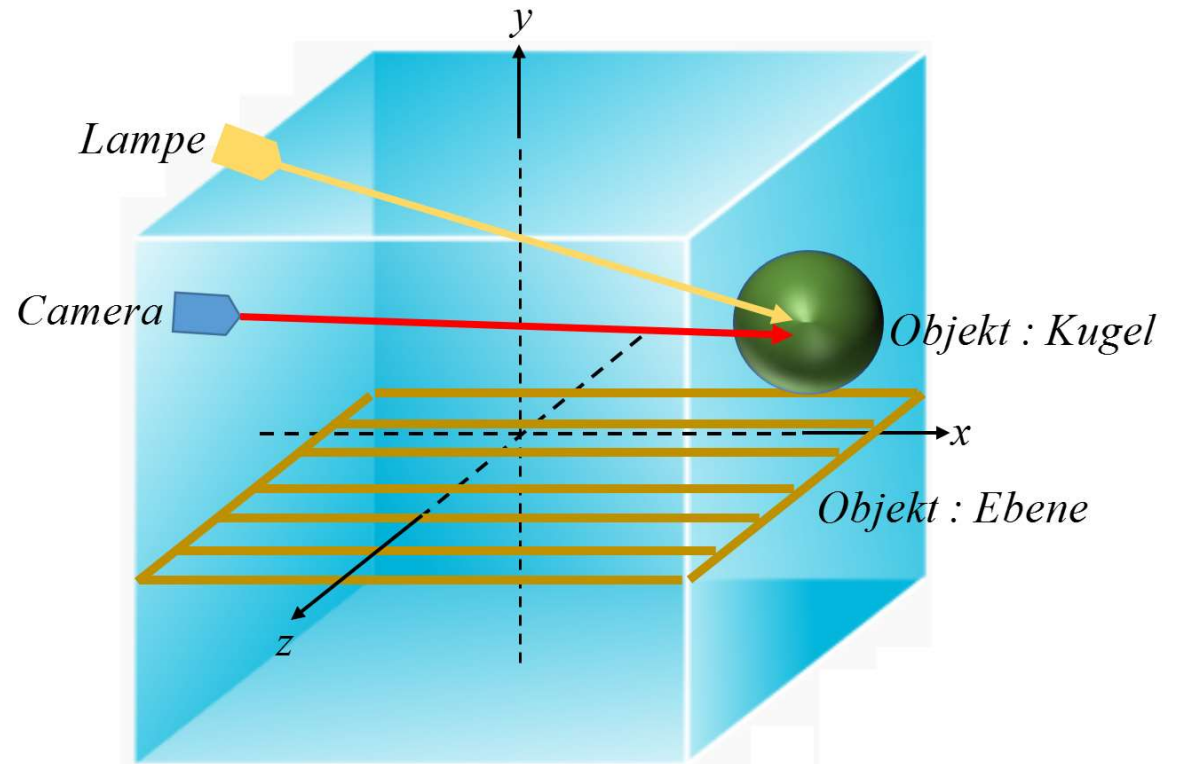
Camera $C(0/1,5/4)$, Lampe $L(7/20/10)$, Kugel

$K_{0,75}(1/1/-0,5)$, Kugel $K_{0,75}(-1,75/1/-1)$,

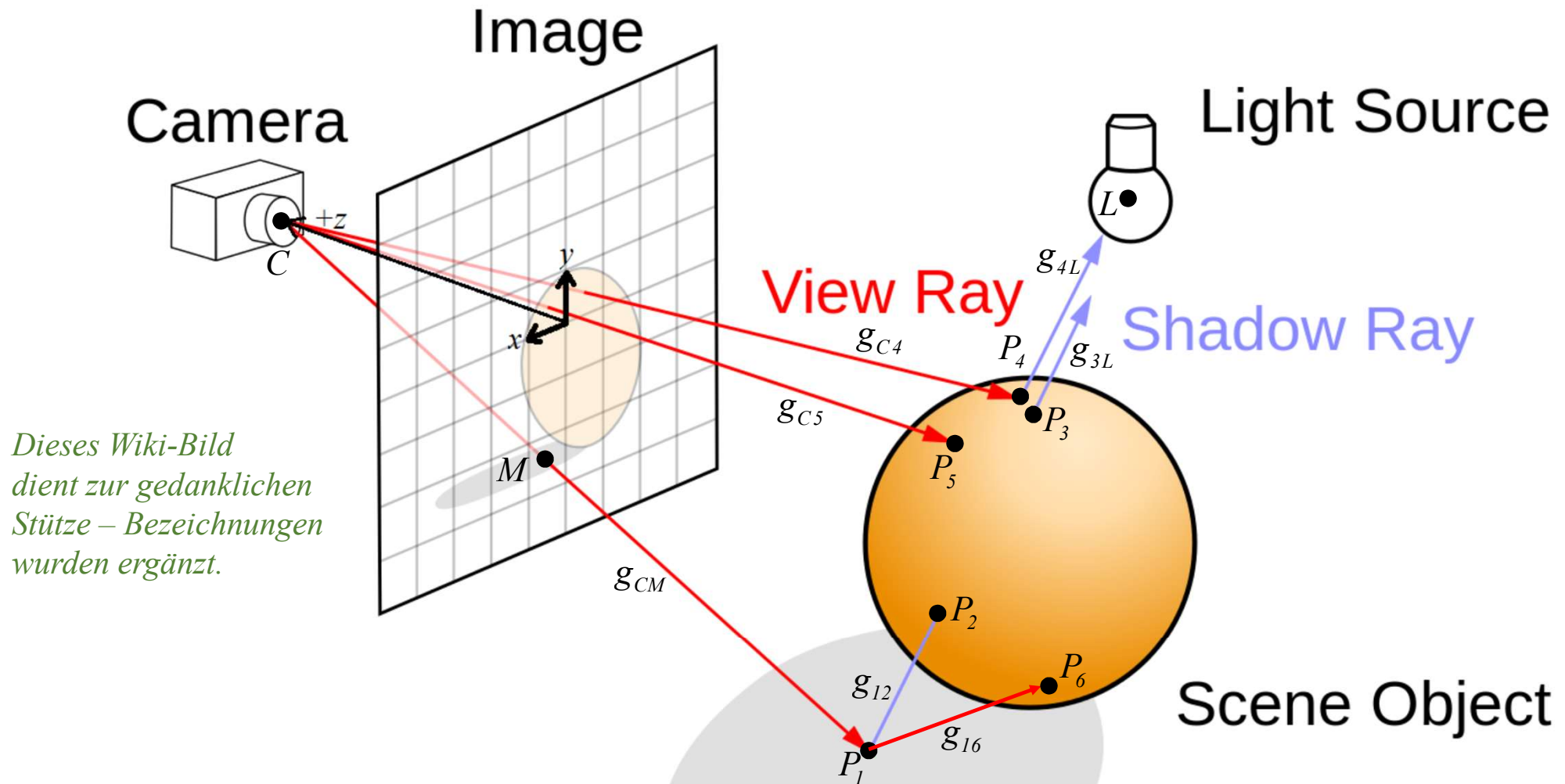
$K_{0,75}(-0,5/2/-1)$ und die Ebene sei die

xz -Koordinatenebene; das Bild entsteht in der xy -Ebene mit

$xmin = -2,5$, $xmax = 2,5$, $ymin = 0$ und $ymax$ wird im Programm wie bekannt berechnet.



```
camera = new Camera(new V3(0, 1.5, +4), ex, ey, ez);
terrain = new Plane(ey, 0, new Color_0_to_1(1, Color.White));
sphere_1 = new Sphere(1 * ex + 1 * ey - 0.5 * ez, 0.75, new Color_0_to_1(1, Color.Red));
sphere_2 = new Sphere(-1.75 * ex + 1 * ey - 1 * ez, 0.75, new Color_0_to_1(1, Color.Green));
sphere_3 = new Sphere(-0.5 * ex + 2 * ey - 1 * ez, 0.75, new Color_0_to_1(1, Color.DarkOrange));
light = new Light(-7 * ex + 20 * ey + 10 * ez, new Color_0_to_1(0, Color.WhiteSmoke));
```



*Dieses Wiki-Bild
dient zur gedanklichen
Stütze – Bezeichnungen
wurden ergänzt.*

Es geht im ersten Schritt stets um die Geraden g_{CM} , wobei Punkt C immer fest ist und M alle Bildpunkte annimmt, weshalb es auch $1920 \cdot 1080 = 2.019.600$ solcher Camera-Geraden gibt, die alle die Szene durchschießen und deren zum Bild liegende nächste Schnittpunkte berechnet werden. Zur Farbbildung und Schattengebung werden zusätzlich noch Unmengen an Geraden wie g_{16} und g_{12} berechnet.

Schnitt von Gerade und Ebene

$$\text{Ebene: } \langle \vec{n}, \vec{x} \rangle - \langle \vec{n}, \vec{b} \rangle = 0$$

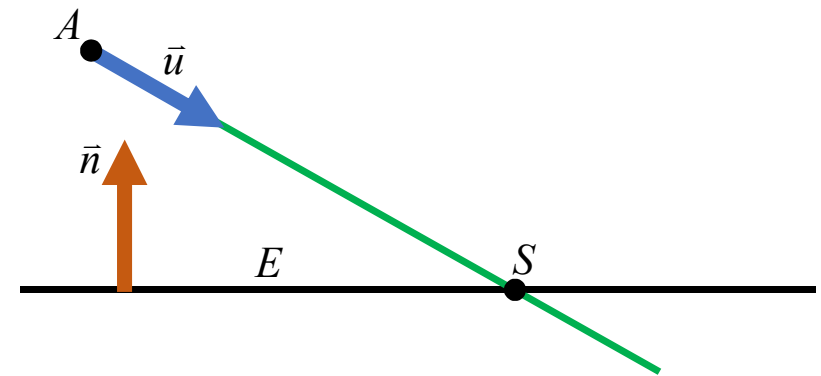
einsetzen: \uparrow

$$\text{Gerade: } \vec{x} = \vec{a} + \lambda \cdot \vec{u}$$

$$\langle \vec{n}, \vec{a} + \lambda \cdot \vec{u} \rangle - \langle \vec{n}, \vec{b} \rangle = 0 = \langle \vec{n}, \vec{a} \rangle + \lambda \cdot \langle \vec{n}, \vec{u} \rangle - \langle \vec{n}, \vec{b} \rangle$$

$$\lambda = \frac{\langle \vec{n}, \vec{b} - \vec{a} \rangle}{\langle \vec{n}, \vec{u} \rangle} \Rightarrow \text{Schnittpunkt } S: \vec{s} = \vec{a} + \frac{\langle \vec{n}, \vec{b} - \vec{a} \rangle}{\langle \vec{n}, \vec{u} \rangle} \cdot \vec{u}$$

A : Aufpunkt E : Ebene \vec{u} : Richtungsvektor \vec{n} : Normalenvektor
 S : Schnittpunkt



```
public override double Lambda_min(Ray ray)
```

```
{
```

```
    double a = ray.get_u * normal;  $\leftarrow \langle \vec{n}, \vec{u} \rangle$ 
```

```
    if (a == 0) return -1; // b = d_PO * normal
```

```
    else // r = <n, b-a>/<n, u>, g: x=a+r*u, E: <n, x>-<n, b>=0
```

```
        return normal * (d_PO * normal - ray.get_a) / a;  $\leftarrow \lambda = \frac{\langle \vec{n}, \vec{b} - \vec{a} \rangle}{\langle \vec{n}, \vec{u} \rangle}$ 
```

```
}
```


Schnitt von Gerade und Kugel

$$\text{Gerade: } \vec{x} = \vec{a} + \lambda \cdot \vec{u}_0$$

↓

$$\text{Kugel: } \langle \vec{x} - \vec{m}, \vec{x} - \vec{m} \rangle = r^2$$

$$\langle \vec{a} + \lambda \cdot \vec{u}_0 - \vec{m}, \vec{a} + \lambda \cdot \vec{u}_0 - \vec{m} \rangle = r^2$$

$$\langle \vec{a} - \vec{m} + \lambda \cdot \vec{u}_0, \vec{a} - \vec{m} + \lambda \cdot \vec{u}_0 \rangle - r^2 = 0$$

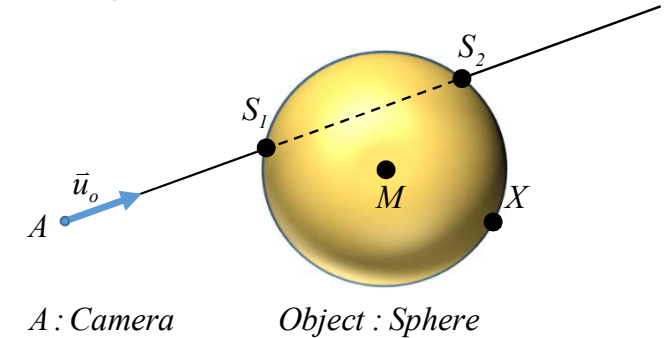
$$\langle \vec{u}_0, \vec{u}_0 \rangle \cdot \lambda^2 + 2 \cdot \langle \vec{a} - \vec{m}, \vec{u}_0 \rangle \cdot \lambda + \langle \vec{a} - \vec{m}, \vec{a} - \vec{m} \rangle - r^2 = 0$$

$$a := \langle \vec{u}_0, \vec{u}_0 \rangle = 1, \quad b := 2 \cdot \langle \vec{a} - \vec{m}, \vec{u}_0 \rangle, \quad c := \langle \vec{a} - \vec{m}, \vec{a} - \vec{m} \rangle - r^2$$

```
public override double Lambda_min(Ray ray) {
    //double a = 1; because a=u*u; ray g:x=a+r*u; u is normalized
    double b = 2 * (ray.get_a - get_m) * ray.get_u; ←  $b := 2 \cdot \langle \vec{a} - \vec{m}, \vec{u}_0 \rangle$ 
    double c = (ray.get_a - get_m) * (ray.get_a - get_m) - radius*radius; ←  $c := \langle \vec{a} - \vec{m}, \vec{a} - \vec{m} \rangle - r^2$ 
    //double discriminant = b * b - 4 * a * c;
    double discriminant = b * b - 4 * c;
    if (discriminant > 0) { //ray intersects sphere  $x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$ 
        //(-b +/- sqrt(b^2-4ac))/(2a)
        double r1 = (-b - Math.Sqrt(discriminant)) / 2.0 - 0.0001; //a=1
        if (r1 > 0) return r1; //closest point of intersection
        else { //Camera is inside the sphere
            double r2 = (-b + Math.Sqrt(discriminant)) / 2.0 - 0.0001;
            return r2;
        }
    } else return -1; } //no intersection of ray and sphere
```

$$\text{Gerade: } \vec{x} = \vec{a} + \lambda \cdot \vec{u}_0$$

$$\text{Kugel: } \langle \vec{x} - \vec{m}, \vec{x} - \vec{m} \rangle = r^2$$



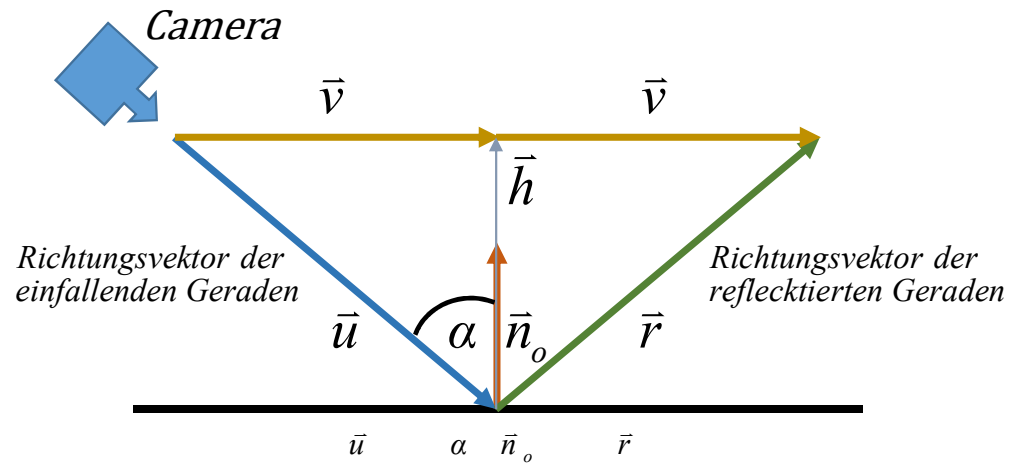
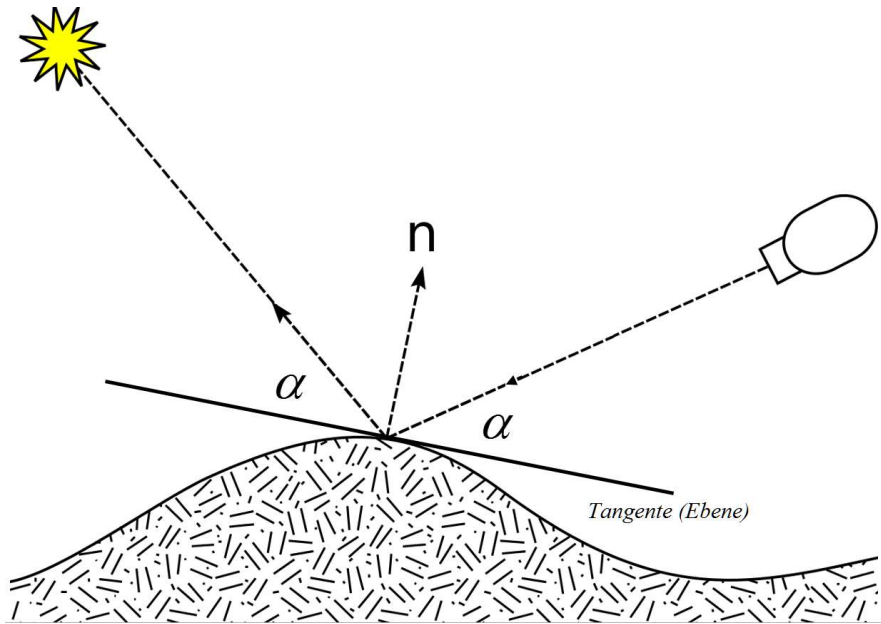
Es wird immer der λ -Wert desjenigen Schnittpunktes S_1, S_2 berechnet, der am nächsten vor der Kamera liegt.

Reflexion einer Geraden

$$\cos(\alpha) = \frac{\langle -\vec{u}, \vec{n}_o \rangle}{\|\vec{u}\|} = \frac{AK}{HY} = \frac{\|\vec{h}\|}{\|\vec{u}\|} \Rightarrow \vec{h} = \|\vec{h}\| \cdot \vec{n}_o = \langle -\vec{u}, \vec{n}_o \rangle \cdot \vec{n}_o$$

$$\vec{v} = \vec{u} + \vec{h} = \vec{u} + \langle -\vec{u}, \vec{n}_o \rangle \cdot \vec{n}_o = \vec{u} - \langle \vec{u}, \vec{n}_o \rangle \cdot \vec{n}_o$$

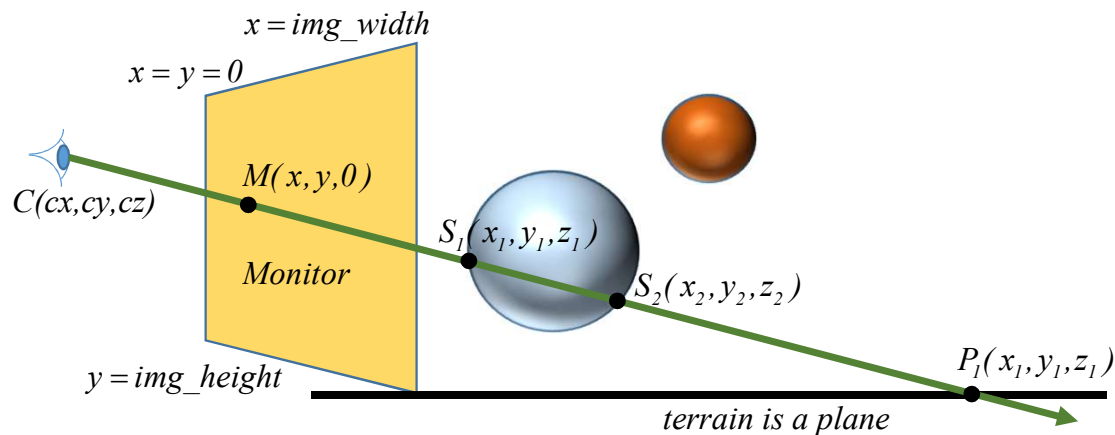
$$\vec{r} = -\vec{u} + 2 \cdot \vec{v} = -\vec{u} + 2 \cdot (\vec{u} - \langle \vec{u}, \vec{n}_o \rangle \cdot \vec{n}_o) = \vec{u} - 2 \cdot \langle \vec{u}, \vec{n}_o \rangle \cdot \vec{n}_o$$



```
V3 co_n = ((AObject)scene_objects[ico]).get_n(A);
V3 r = u - 2 * co_n * u * co_n;
List<double> lambda_ref = find_pt_ob(new Ray(A, r));
```

Die Camera-Geraden g_{CM}

```
img_width = this.ClientSize.Width; img_height = this.ClientSize.Height;
ymax = (xmax - xmin) / img_width * img_height;
xFactor = img_width / (xmax - xmin);
yFactor = img_height / (ymax - ymin);
double dx =(xmax-xmin)/img_width, dy=(ymax-ymin)/img_height;
double x1 = xmin, y1 = ymax;
for (int x = 0; x < img_width; x++) {
    for (int y = 0; y < img_height; y++) {
        Ray h = new Ray(camera.getcampos, new V3(x1, y1, 0) - camera.getcampos); ←  $g_{CM}$ 
```

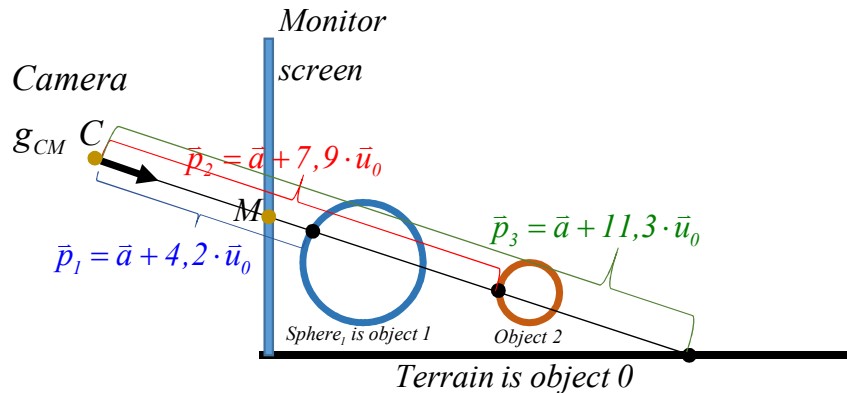


```
public Ray(V3 o, V3 d)
{
    A = o;
    u = d.normalize;
}
```

Die Gerade **cam_ray** mit dem Aufpunkt $C(cx, cy, cz)$ erhält vom Constructor der **ray class** einen Richtungsvektor, der stets normiert ist.

cam_ray trifft zwei Objekte: die blaue Kugel und die Ebene – im Programm immer als **terrain** bezeichnet.

Wer darf aufs Bild?



Die Gerade g_{CM} , ausgehend von der Camera durch einen Punkt des Bildschirms, durchstößt die Objekte der Szene. Die Funktion `find_pt_ob(g_{CM})` sucht nach Schnittpunkten und gibt den in der Szene am nächsten zum Bildschirm liegenden Schnittpunkt zurück, und zwar in Form des λ -Wertes und der Objektnummer (Index). Dazu sind die beiden ersten Zellen der Liste reserviert: Zelle Nummer 0 enthält den λ -Wert und Zelle Nummer 1 den zugehörigen Objektindex aus der Liste der Szenenobjekte. Hat ein Objekt mehrere Schnittpunkte, so wird nur der λ -Wert in die Liste aufgenommen, dessen Schnittpunkt am nächsten zum Bildschirm liegt. Da der Richtungsvektor der Camera-Geraden g_{CM} vom Constructor normiert wird, sind die λ -Werte auch die Abstände zwischen Camera und Schnittpunkt: $\lambda = d(C, S)$

```
public List<double> find_pt_ob(Ray g)
```

0: 4,2	1: 1	2: 11,3	3: 4,2	4: 7,9
--------	------	---------	--------	--------

```
public List<double> find_pt_ob(Ray g) //find intersectionpoint(pt) and the object(ob)
{ //first: calculate all intersections
    List<double> L = new List<double>(); //reserve
    L.Add(0); L.Add(0); //L[0]=lambda; L[1]=object index in scene_objects
    for (int n = 0; n < scene_objects.Count; n++)
        L.Add(((AObject)scene_objects[n]).Lambda_min(g));
    L[0] = L.Max(); L[1] = -1; //and then find the closest intersection to the screen
    if (L[0] > 0) //in front of the camera
        for (int n = 2; n < L.Count; n++)
            if (0 < L[n] && L[n] <= L[0])
                { //... now we look for smaller ones
                    L[0] = L[n];
                    L[1] = n - 2; //0:terrain; 1:sphere
                } // "n - 2": remember L[0] and L[1] are reserved
    return L;
}
```

Farbe des Bildpunktes M

Die Farbe von M ist

zunächst einmal die Farbe von P_1 .

$\text{Color}(M) = \text{ColorAt}(g) \leftarrow \text{ambientlight} \cdot \text{Color}(P_1)$

denn g trifft P_1 ; dieser Punkt ist nicht selbstleuchtend,

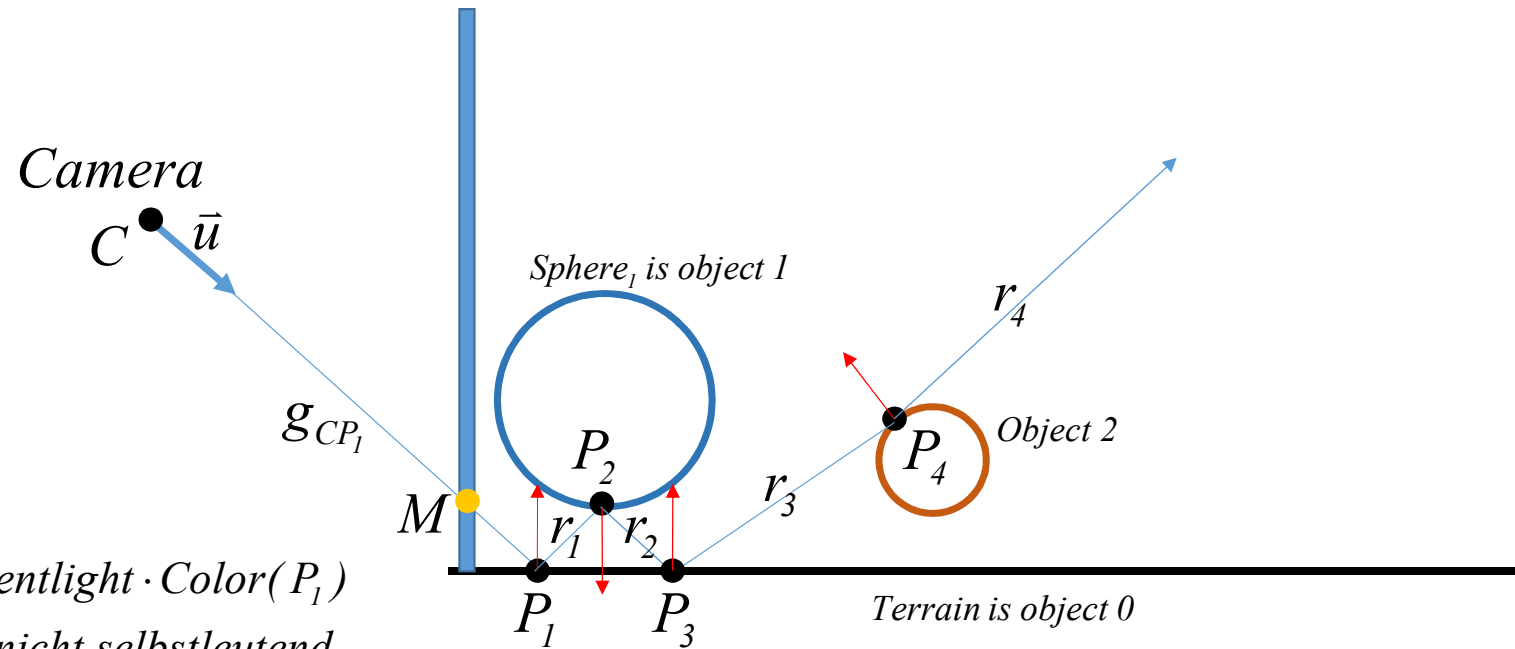
weshalb die Farbe durch die Umgebungshelligkeit bestimmt wird. Da P_2 , P_3 und P_4 auch Einfluss auf die Farbe von P_1 haben, muss deren Anteil zu $\text{Color}(M)$ hinzukommen. Sei dazu $k := \text{ambientlight}$ und $P_n := \text{Color}(P_n)$

$$P_3 \leftarrow P_3 + k \cdot P_4$$

$$P_2 \leftarrow P_2 + k \cdot (P_3 + k \cdot P_4)$$

$$P_1 \leftarrow P_1 + k \cdot (P_2 + k \cdot (P_3 + k \cdot P_4))$$

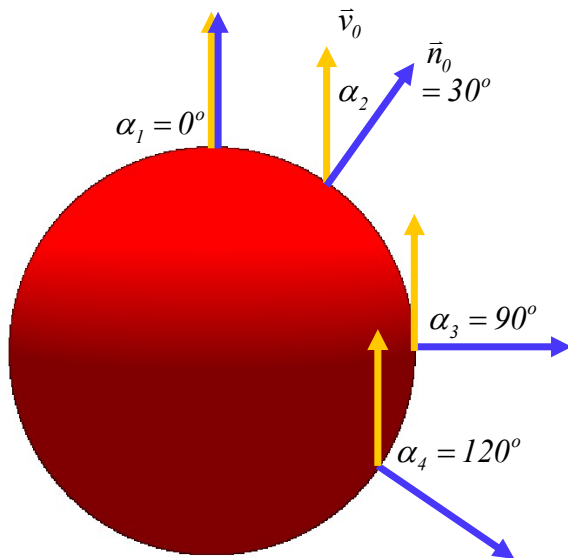
$\text{Color}(M) = \text{ColorAt}(g) \leftarrow \{\text{ambientlight} \cdot \text{Color}(P_1) + \text{ColorAt}(r_1)\}$; ColorAt arbeitet rekursiv!



Beitrag der Lichtquelle



Lichtquelle, weit oberhalb der Kugel



Im Falle einer weit oberhalb über der Kugel platzierten Lichtquelle können alle gelben normierten Vektoren \vec{v}_0 als nahezu parallel betrachtet werden. Die blauen Vektoren \vec{n}_0 sind die normierten Normalenvektoren der Kugel.

$$\text{Dann gilt: } \langle \vec{n}_0, \vec{v}_0 \rangle = \underbrace{\|\vec{n}_0\| \cdot \|\vec{v}_0\|}_{1 \cdot 1} \cdot \cos(\alpha) = \cos(\alpha)$$

Somit kann das Skalarprodukt als Maß für den Lichteinfall an den dargestellten Stellen betrachtet werden:

$$\langle \vec{n}_0, \vec{v}_0 \rangle = \cos(0^\circ) = 1; \quad \langle \vec{n}_0, \vec{v}_0 \rangle = \cos(30^\circ) = 0,866;$$

$$\langle \vec{n}_0, \vec{v}_0 \rangle = \cos(90^\circ) = 0; \quad \langle \vec{n}_0, \vec{v}_0 \rangle = \cos(120^\circ) = -0,5;$$

An Stellen ohne direkter Lichteinstrahlung ist $\cos(\alpha)$ negativ.

```
V3 v = light.get_p - A;
double cos_alpha = co_n * v.normalize;//<n0,v0> = cos(alpha)*1*1
if (cos_alpha > 0)//that is if lightsource is in front and not behind(cos<0)
    final_color = final_color + cos_alpha * co_color * light.getColor;
```

Einfluss der Lampe auf die Farbe des Bildpunktes M

Sei ambientlight := 0,4

$$Color(M) = ColorAt(g) \leftarrow \left\{ 0,4 \cdot Color(P_l) + \underbrace{ColorAt(r_l)}_{Rekursion} + \underbrace{\cos(\alpha)}_{falls > 0} \cdot Color(P_l) \cdot Color(Lampe) \right\}$$

Farbe des Bildpunktes M bei spiegelnder Kugeloberfläche



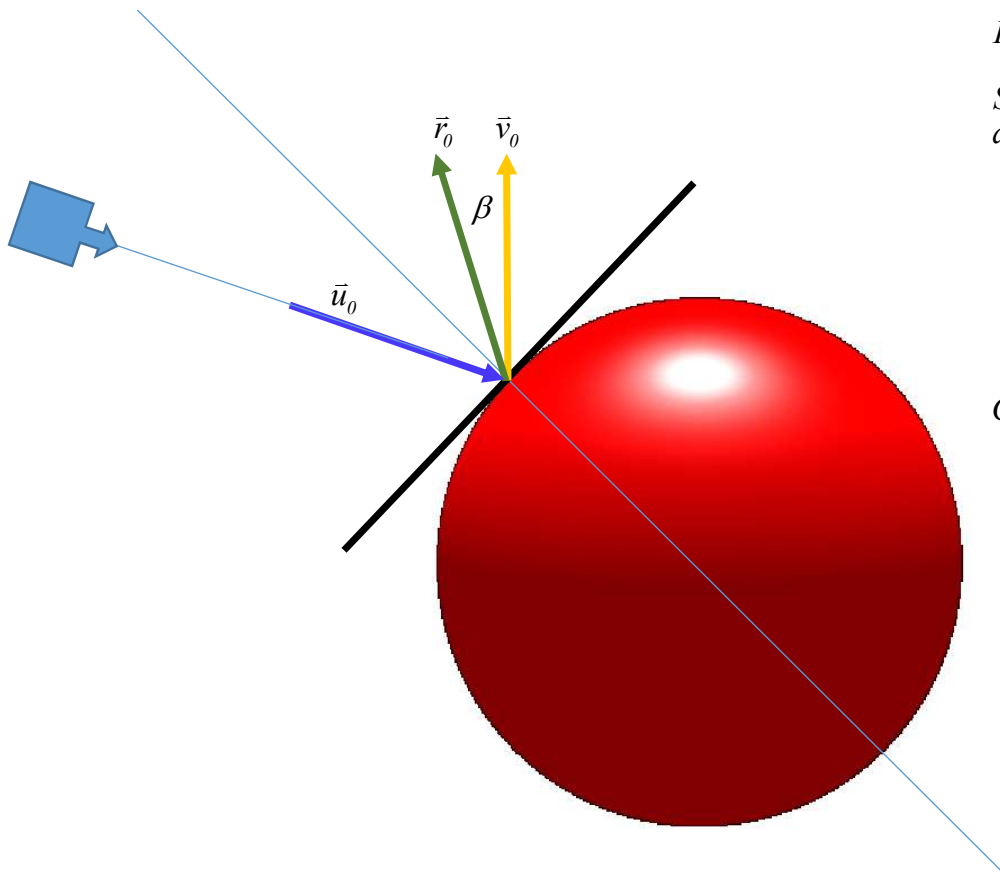
Ähnliche Situation wie auf Seite 14, nur dass der Normalenvektor durch den normierten Richtungsvektor \vec{r}_0 der Reflexionsgeraden ersetzt wird.

Analog erhält man:

$$\cos(\beta) = \langle \vec{r}_0, \vec{v}_0 \rangle$$

Im Falle $\cos(\beta) > 0$ erhält man Glanzeffekte (Spiegelung), andernfalls nicht.

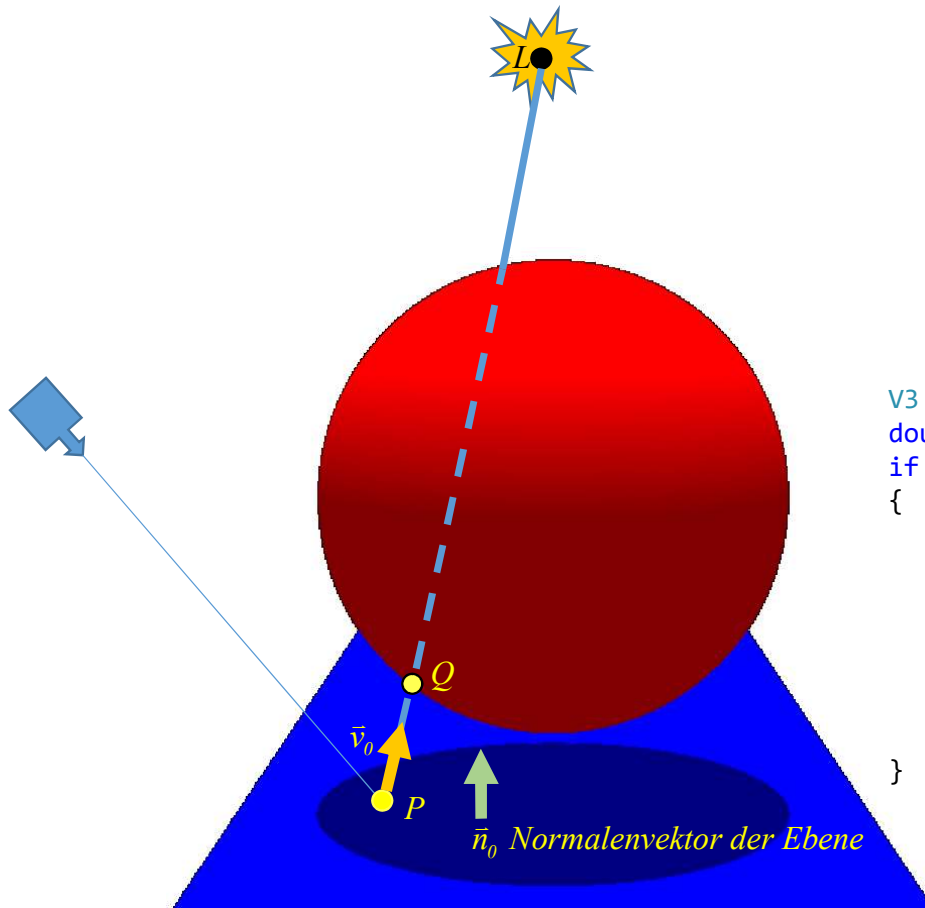
Sei $\text{ambientlight} := 0,4$ und die Fähigkeit der Stelle P_1 zu glänzen: $\text{specular}(P_1) = 0,95$, dann gilt für die Farbe des Bildpunktes M in erster einfacher Näherung:



$$\text{Color}(M) = \text{ColorAt}(g) \leftarrow \left\{ \begin{array}{l} A: \quad 0,4 \cdot \text{Color}(P_1) \\ B: \quad + \underbrace{\text{ColorAt}(r_1)}_{\text{Rekursion}} \\ C: \quad + \underbrace{\cos(\alpha)}_{\text{falls } > 0} \cdot \text{Color}(P_1) \cdot \text{Color}(\text{Lampe}) \\ D: \quad + \underbrace{\cos^{10}(\beta)}_{\text{falls } \cos(\beta) > 0} \cdot 0,95 \cdot \text{Color}(\text{Lampe}) \end{array} \right\}$$

```
double cos_beta = r.normalize * v.normalize;
if (cos_beta > 0) //mirroring
    final_color = final_color + Math.Pow(cos_beta, 10) * co_color.C01_S * light.getColor;
```

Schatten



Da \vec{v}_0 normiert ist, ist λ in $\vec{q} = \vec{p} + \lambda \cdot \vec{v}_0$ auch der Abstand $d(P, Q)$.

Der Abstand zwischen P und der Lichtquelle ist $\|\overline{PL}\|$.

Würde die Kugel über der Lichtquelle liegen, wäre $\lambda > \|\overline{PL}\|$ und es gäbe keinen Schatten; nun ist jedoch in der Skizze $\lambda < \|\overline{PL}\|$, was Schatten zur Folge hat. Im Code auf Seite 14 ist der Schatten noch nicht berücksichtigt. Hier die Codeerweiterung:

```
V3 v = light.get_p - A;
double cos_alpha = co_n * v.normalize(); ← <  $\vec{u}_0, \vec{n}_0$  >
if (cos_alpha > 0) // that is if lightsource is in front and not behind (cos < 0)
{
    // ray for shadow or not shadow
    List<double> lambda2 = find_pt_ob(new Ray(A, v));
    bool noShadow = true;
    for (int c = 2; c < lambda2.Count; c++)
        if (fuzzy < lambda2[c] && lambda2[c] < v.Length) ←  $\lambda < \|\overline{PL}\|$ 
            { noShadow = false; break; }
    if (noShadow)
        final_color = final_color + cos_alpha * co_color * light.getColor;
}
```

Die gesamte Prozedur der Farbermittlung auf einen Blick

```
public Color_0_to_1 getColorAt(V3 A, V3 u, int ico)//ico: index of closest object
{
    //co is closest object we=eye=camera see; A: initial point; u: u of intersection ray
    Color_0_to_1 co_color;

    //A:-----terrain-HHG-----
    if (ico == 0)//only terrain
    {
        int xx = Math.Min(xX(A.X), HHG.Width-5), yy = Math.Min(xX(A.Z-0.6), HHG.Height-5);
        xx = xx < 0 ? 0 : xx; yy = yy < 0 ? 0 : yy;
        co_color = new Color_0_to_1(0, HHG.GetPixel(xx, yy));
    } else co_color = ((AObject)scene_objects[ico]).getColor();
    Color_0_to_1 final_color = ambientlight * co_color; //first portion of color

    //B:-----Color-from-the-r-direction-----
    V3 co_n = ((AObject)scene_objects[ico]).get_n(A);
    V3 r = u - 2 * co_n * u * co_n;//calculate refraction r = u-2*<u,n0>*n0
    if (ico != 0)//except terrain
    {
        List<double> lambda_ref = find_pt_ob(new Ray(A, r));//ray x=A+lam*r hits the scene
        if (lambda_ref[1] != -1)//ray hits plane or sphere
        {
            if (lambda_ref[0] > fuzzy)
            {
                //r only affects the final color if it is reflected from something
                V3 a2 = A + r * lambda_ref[0];//a further intersection point ...
                double s2 = ((AObject)scene_objects[(int)lambda_ref[1]]).getColor.C01_S;
                /*recursive-call*/
                final_color = final_color + s2 * getColorAt(a2, r, (int)lambda_ref[1]);
            }
            //...and the second portion of color
        }
    }

    //C:-----Contribution of the light sources
    V3 v = light.get_p - A;
    double cos_alpha = co_n * v.normalize;//<n0,v0> = cos(alpha)*1*1
    if (cos_alpha > 0)//that is if lightsource is in front and not behind(cos<0)
    {
        List<double> lambda2 = find_pt_ob(new Ray(A, v));//ray for shadow or not shadow
        bool noShadow = true;//c = 2, da 0 für lambda u. 1 für obj. reserviert sind
        for (int c = 2; c < lambda2.Count; c++)
        {
            if (fuzzy < lambda2[c] && lambda2[c] < v.Length)
            {
                noShadow = false; break; }
        }
        if (noShadow)
        {
            final_color = final_color + cos_alpha * co_color * light.getColor();
        }
    }

    //D:-----reflective-surface-----
    if (ico!=0)//except terrain
    {
        double cos_beta = r.normalize * v.normalize;
        if (cos_beta > 0)//mirroring
        {
            final_color = final_color + Math.Pow(cos_beta, 10) * co_color.C01_S * light.getColor();
        }
    }

    return final_color.clip();
}
```

→ Seite 13

→ Seite 13

→ Seite 14, 15 und 17

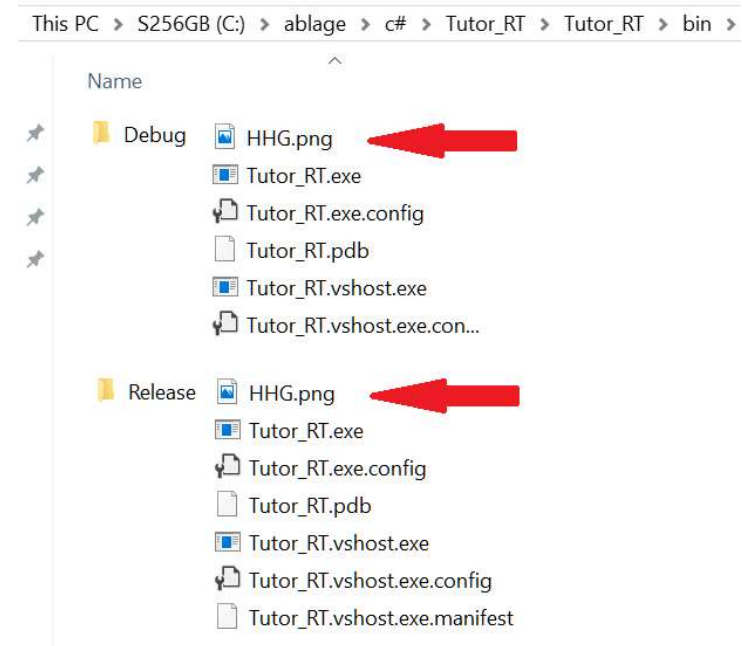
→ Seite 16

Zum Schluss noch einige Bemerkungen

*Das Projekt ist mit der kostenlosen **Microsoft Visual Studio Community** Software erstellt*

*Das Bild **HHG.png** muss je nach Build (Release or Debug) im entsprechenden Verzeichnis liegen, also in dem Verzeichnis, in dem auch die *.exe-Datei liegt*

Die rekursive Funktion `getColorAt(...)` benutzt zum Abbruch den Objektindex -1, `if (lambda_ref[1] != -1)` was bei zu beliebig positionierten Szenenobjekten zum Speicherüberlauf führt, was man natürlich durch eine Maximalanzahl von Aufrufen verhindern könnte.



... Edgar Marx (edgarmarx@t-online.de)