Mathematische Grundlagen

1.9 Besprechung der Übungsaufgaben

Übungsaufgabe 1.1:

Aus der Formulierung: »die kurze Seite verhält sich zur langen wie die lange zur Summe der beiden Seiten« folgt, dass je größer die kurze Seite ist, desto größer auch die längere sein muss, da beide Seiten positive Längen haben. Demnach handelt es sich hier um eine proportionale Zuordnung:

Kurze Seite	Lange Seite
1	ϕ
ϕ	$1+\phi$

$$\frac{\phi}{1} = \frac{1+\phi}{\phi} \quad \Leftrightarrow \quad \phi^2 = 1+\phi \quad \Leftrightarrow \quad \phi^2 - \phi - 1 = 0 \quad \Leftrightarrow$$

$$\phi_1 = \frac{1}{2} + \sqrt{\frac{1}{4} + 1} = \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{4}{4}} = \frac{1}{2} + \sqrt{\frac{5}{4}} = \frac{1+\sqrt{5}}{2}$$

$$\phi_2 = \frac{1-\sqrt{5}}{2}$$

Die Lösung ϕ_2 kann hierbei ignoriert werden, denn sie entspricht der Länge der kurzen Seite, wenn die Länge der größeren Seite den Wert 1 hat. Das gesuchte Seitenverhältnis ist somit:

$$1 \quad : \quad \frac{1+\sqrt{5}}{2}$$

Bei der Lösung der hier beschriebenen Gleichung wird die pq-Formel verwendet; die beiden Lösungen der Gleichung:

$$x^2 + p * x + q = 0 \qquad \textit{für } p, q \in R$$

lassen sich wie folgt berechnen:

$$x_1 = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q} \qquad und \qquad x_2 = -\frac{p}{2} - \sqrt{\frac{p^2}{4} - q}$$

Übungsaufgabe 1.2:

Bei der Menge **I** handelt es sich um die Punkte, die zwar von der zweiten, aber nicht von der ersten Lichtquelle beleuchtet werden, das heißt die Punkte, die in **N**, aber nicht in **M** enthalten sind. **I** ist somit eine einelementige Menge:

$$I = N \setminus M = \{\, p \mid p \in N \; und \; p \notin M \,\} = \{\, c, d, e \,\} \setminus \{\, a, b, c, d \,\} = \{\, e \,\}$$

Da der Punkt e nur von der schwachen Lichtquelle beleuchtet wird, muss ihm eine Helligkeit zugeordnet werden, die geringer ist als diejenige der Punkte aus H, aber höher als die Helligkeit, mit der die übrigen Punkte des Gegenstandes gezeichnet werden.

Übungsaufgabe 1.3:

1. $Q = \{\, (x, y) \in R^2 \mid x \geq 2 \; und \; x \leq 4 \; und \; y \geq 3 \; und \; y \leq 5 \,\}$
2. $L = M \cup N$

   wobei

   $$M = \{\, (x, y) \in R^2 \mid x \geq 3 \; und \; x \leq 5 \; und \; y \geq 2 \; und \; y \leq 6 \,\}$$

   $$N = \{\, (x, y) \in R^2 \mid x \geq 5 \; und \; x \leq 7 \; und \; y \geq 2 \; und \; y \leq 4 \,\}$$

3. Das linke Koordinatensystem ist rechts-, die beiden anderen linkshändig.
4. p0 = (a, 0.5, 0) p1 = (a, –0.5, 0) p2 = (–a, –0.5, 0) p3 = (–a, 0.5, 0)
   p4 = (0.5, 0, –a) p5 = (–0.5, 0, –a) p6 = (–0.5, 0, a) p7 = (0.5, 0, a)
   p8 = (0, a, 0.5) p9 = (0, –a, 0.5) p10 = (0, –a, –0.5) p11 = (0, a, –0.5)

   Die großen Seiten der Rechtecke besitzen die in Übungsaufgabe 1.1 berechnete Länge ϕ. Für die Strecke **a** gilt: **a = 0.5** * ϕ, weil der Mittelpunkt jedes der drei Rechtecke im Ursprung des Koordinatensystems liegt.

Übungsaufgabe 1.4:

$$|\vec{u}| = \sqrt{5^2 + (-9)^2 + 3^2} = \sqrt{115} \approx 10.72$$

Das Ergebnis der Multiplikation einer beliebigen Zahl mit sich selbst ist stets eine positive Zahl, und die Summe positiver Zahlen ist immer positiv. Aus diesem Grund wird bei Verwendung reeller Vektorkomponenten stets die Wurzel aus einem positiven Wert gezogen, ein Abfangen des beschriebenen Sonderfalls ist nicht erforderlich.

Übungsaufgabe 1.5:

$$|t*\vec{v}| = \left|\begin{pmatrix} t*x \\ t*y \\ t*z \end{pmatrix}\right| = \sqrt{(t*x)^2 + (t*y)^2 + (t*z)^2} = \sqrt{t^2*x^2 + t^2*y^2 + t^2*z^2}$$

$$= \sqrt{t^2*(x^2+y^2+z^2)} = \sqrt{t^2} * \sqrt{(x^2+y^2+z^2)} = |t| * |\vec{v}|$$

Der Ausdruck $\sqrt{t^2}$ muss nach | t | umgeformt werden, denn **t** kann durchaus einen negativen Wert besitzen, wobei die Länge des Vektors t * $\vec{v}$ jedoch als das Ergebnis einer Wurzelberechnung im Reellen einen positiven Wert haben muss.

Übungsaufgabe 1.6:

Denselben Rechenweg wie bei der letzten Übungsaufgabe befolgend, ergibt sich die folgende Gleichung. Man beachte, dass der Wert des Bruchs und damit auch **t** positiv ist – ein **t** < 0 würde die Richtung des Vektors verändern:

$$|t*\vec{v}| = |t|*|\vec{v}| = n \qquad \Rightarrow \qquad t = \frac{n}{|\vec{v}|} \qquad\qquad \textit{für } |\vec{v}| \neq 0$$

Hierbei handelt es sich um eine sehr wichtige Beziehung: Um den Skalar zu berechnen, mit dessen Hilfe die Länge eines Vektors $\vec{v}$ auf **n** > **0** gesetzt wird, muss man **n** durch den Betrag von $\vec{v}$ teilen.

Übungsaufgabe 1.7:

Hierbei berechnet man zunächst, wie im Beispiel angegeben, die Länge der Hypotenuse **h** mit einer Länge von 40 Metern. Anschließend:

$$\cos(30°) = \frac{a}{40\,m} \quad \Leftrightarrow \quad a = 40\,m * \cos(30°) \approx 34{,}64\,m$$

Diese Berechnung kann aber auch direkt durchgeführt werden:

$$\tan(30°) = \frac{20\,m}{a} \quad \Leftrightarrow \quad a = \frac{20\,m}{\tan(30°)} \approx 36{,}64\,m$$

Übungsaufgabe 1.8:

Rotation um die y-Achse:

$$\begin{aligned} z' &= z*\cos(\alpha) - x*\sin(\alpha) \\ x' &= z*\sin(\alpha) + x*\cos(\alpha) \\ y' &= y \end{aligned}$$

Übungsaufgabe 1.9:

1. Da aus der Definition folgt: $(p+\lambda*\vec{v}) \in G$ für eine beliebige reelle Zahl λ, muss gelten: $p+0*\vec{v}=p \quad \Rightarrow \quad p \in G$. Der Punkt c liegt in Abbildung 1.23 im Buch in der dem Vektor $\vec{v}$ entgegengesetzten Richtung. Weil λ jedoch auch negative Werte annehmen kann, liegt c in G, denn die Multiplikation eines Vektors mit einem negativen Skalar kehrt die Richtung des Vektors um.

2. Anders als bei der Definition von **G** darf die Skalarmultiplikation die Orientierung des Richtungsvektors von **S** nicht umdrehen:

   $$S=\{\ p+\lambda*\vec{v}\ \mid \lambda \in R \ und\ \lambda \geq 0\ \}$$

   $$L=\{\ p+\lambda*\vec{v}\ \mid \lambda \in R \ und\ \lambda \geq 0 \ und\ \lambda \leq 1\ \} \qquad wobei\ \ \vec{v}=q-p$$

   Der Vektor $\vec{v}$ ist außerhalb von **L** definiert und somit konstant. Aus seiner Definition ergibt sich, dass für λ = **0.0** die Summe den Anfangspunkt **p** ergibt und für λ = **1.0** den Endpunkt **q**. Bei λ = **0.5** wäre das Ergebnis der Punkt, in der Mitte der Strecke – weil der Skalar lediglich Werte zwischen **0.0** und **1.0** annehmen darf, enthält **L** alle Punkte auf der Verbindungslinie zwischen **p** und **q**.

Übungsaufgabe 1.10:

Additivität des Skalarprodukts in der zweiten Komponente:

$$\begin{aligned} sp(\vec{u},\vec{v}+\vec{w}) &\overset{Symmetrie}{=} sp(\vec{v}+\vec{w},\vec{u}) \overset{Additivit\ddot{a}t}{=} sp(\vec{v},\vec{u})+sp(\vec{w},\vec{u}) \overset{Symmetrie}{\Rightarrow} \\ sp(\vec{u},\vec{v}+\vec{w}) &= sp(\vec{u},\vec{v})+sp(\vec{u},\vec{w}) \end{aligned}$$

Das Standard-Skalarprodukt:

1. *Additivität*:

$$\begin{aligned} & s(\vec{a}+\vec{b},\vec{c}) \\ =\ & (ax+bx)*cx+(ay+by)*cy+(az+bz)*cz \\ =\ & ax*cx+bx*cx+ay*cy+by*cy+az*cz+bz*cz \\ =\ & (ax*cx+ay*cy+az*cz) \quad + \quad (bx*cx+by*cy+bz*cz) \\ =\ & s(\vec{a},\vec{c})+s(\vec{b},\vec{c}) \end{aligned}$$

2. *Homogenität*:

$$\begin{aligned} & s(\lambda*\vec{a},\vec{b}) \\ =\ & \lambda*ax*bx+\lambda*ay*by+\lambda*az*bz \\ =\ & \lambda \quad * \quad (ax*bx+ay*by+az*bz) \\ =\ & \lambda \quad * \quad s(\vec{a},\vec{b}) \end{aligned}$$

3. *Positivität*:

$$s(\vec{a},\vec{a}) \;=\; ax^2 + ay^2 + az^2 \;\geq\; 0$$

$$s(\vec{a},\vec{a}) = 0 \quad \Rightarrow \quad ax^2 + ay^2 + az^2 = 0 \qquad \Rightarrow$$
$$ax = ay = az = 0 \quad da\; ax^2 > 0\; und\; ay^2 > 0\; und\; az^2 > 0$$

$$f\ddot{u}r\; alle\; ax, ay, az \in R$$

4. *Symmetrie*:

$$\begin{aligned} & s(\vec{a},\vec{b}) \\ = \;& ax * bx + ay * by + az * bz \\ = \;& bx * ax + by * ay + bz * az \\ = \;& s(\vec{b},\vec{a}) \end{aligned}$$

Übungsaufgabe 1.11:

$$gegeben: \quad sp(\vec{u},\vec{n}) = 0 \quad und \quad sp(\vec{v},\vec{n}) = 0$$
$$\Rightarrow$$
$$sp(\lambda * \vec{u} + \eta * \vec{v}, \vec{n}) = sp(\lambda * \vec{u}, \vec{n}) + sp(\eta * \vec{v}, \vec{n}) = \lambda * sp(\vec{u},\vec{n}) + \eta * sp(\vec{v},\vec{n})$$
$$= \lambda * 0 + \eta * 0 = 0 \quad f\ddot{u}r\; alle\; \lambda, \eta \in R$$

für ein beliebiges Skalarprodukt **sp()**.

Übungsaufgabe 1.12:

$$sp(\vec{s} - \vec{q}, \vec{n}) \;=\; 0 \;\overset{(1)}{\Leftrightarrow}\; (-1) * sp(\vec{s} - \vec{q}, \vec{n}) \;=\; (-1) * 0 \;\overset{(2)}{\Leftrightarrow}$$
$$sp[\,(-1) * (\vec{s} - \vec{q}), \vec{n}\,] \;=\; 0 \;\overset{(3)}{\Leftrightarrow}\; sp(-\vec{s} + \vec{q}, \vec{n}) = 0 \;\overset{(4)}{\Leftrightarrow}\; sp(\vec{q} - \vec{s}, \vec{n}) = 0$$

(1): Multiplikation beider Seiten der Gleichung mit **(–1)**
(2): Homogenität des Skalarproduktes
(3): Dritte Eigenschaft der Skalarmultiplikation
(4): Kommutativität der Vektoraddition

1.10 Besprechung der Projekte

1.10.1 Die Koordinaten regelmäßiger Polygone

Nach Vorgabe besitzen alle Punkte denselben Abstand zum Mittelpunkt des n-Ecks – das heißt, dass alle Punkte auf einem Kreis mit Radius **r** liegen. Wie man anhand der Abbildung 1.15 im Buch sieht, kann für jeden Punkt ein rechtwinkliges Dreieck definiert werden, mit dessen Hilfe sich leicht Sinus- und Kosinusbeziehungen aufstellen lassen; die vollständige Definition der Menge **P** lautet somit:

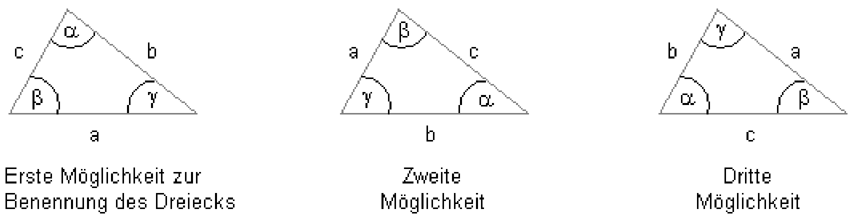
$$P = \{\ p_i \in R^2 \mid p_i = [\ r * \cos(i * \alpha), r * \sin(i * \alpha)\ ]\ und\ i \in N_0\ und\ 0 \le i < n\ \}$$

$$wobei \quad \alpha = \frac{360°}{n}$$

1.10.2 Der Winkel zwischen zwei Vektoren

1. Schritt:

Die gesuchten Ausdrücke können zwar durch normales Nachrechnen ermittelt werden, hier lernen wir jedoch eine zweite Methode kennen: die Aufstellung der Gleichungen durch genaues Hinsehen.



Erste Benennung	Zweite Benennung
a	b
b	c
α	β
β	γ

Erste Benennung	Dritte Benennung
a	c
b	a
α	γ
β	α

Abb. 1.1: Einander entsprechende Seiten und Winkel in der ursprünglichen und der gedrehten Version des Dreiecks

Der gegebene Ausdruck:

$$c = b * \cos(\alpha) + a * \cos(\beta)$$

macht in erster Linie eine Aussage über die beiden oberen Seiten und die zwei Basiswinkel des Dreiecks. Zufällig tragen diese die Bezeichnungen a, b, α und β. *Dreht* man jedoch das Dreieck, sodass **a** die Grundseite bildet, bleibt der Ausdruck bis auf die neuen Bezeichnungen der Seiten und Winkel unverändert. Kennt man die einander entsprechenden Seiten und Winkel, lassen sich die gesuchten Ausdrücke sofort formulieren:

$$a = c * \cos(\beta) + b * \cos(\gamma)$$
$$b = a * \cos(\gamma) + c * \cos(\alpha)$$

Von großer Wichtigkeit ist, dass in der rotierten Version die Beziehungen der Seiten und Winkel untereinander unverändert bleibt: Sowohl im unveränderten als auch in den beiden gedrehten Versionen entgegen dem Uhrzeigersinn angeordnet.

2. Schritt:

$$\cos(\alpha) = \frac{c - a * \cos(\beta)}{b} \quad \textit{und} \quad \cos(\alpha) = \frac{b - a * \cos(\gamma)}{c} \quad \Leftrightarrow$$

$$\frac{c - a * \cos(\beta)}{b} = \frac{b - a * \cos(\gamma)}{c}$$

3. Schritt:

$$\frac{c - a * \cos(\beta)}{b} = \frac{b - a * \cos(\gamma)}{c} \quad \Leftrightarrow$$

$$c^2 - a * c * \cos(\beta) = b^2 - a * b * \cos(\gamma) \quad \overset{*}{\Leftrightarrow}$$

$$c^2 - a * [a - b * \cos(\gamma)] = b^2 - a * b * \cos(\gamma) \quad \Leftrightarrow$$

$$c^2 - a^2 + a * b * \cos(\gamma) = b^2 - a * b * \cos(\gamma) \quad \Leftrightarrow$$

$$c^2 = a^2 + b^2 - 2 * a * b * \cos(\gamma)$$

Hierbei wird in der zweiten Umformung der gegebene Ausdruck:

$$\cos(\beta) = \frac{a - b * \cos(\gamma)}{c}$$

nach $c * \cos(\beta)$ umgeformt und in die Gleichung eingesetzt.

4. Schritt:

$$|\vec{c}\,|^2 = |\vec{a}\,|^2 + |\vec{b}\,|^2 - 2 * |\vec{a}\,| * |\vec{b}\,| * \cos(\gamma) \quad \Leftrightarrow$$

$$|\vec{b} - \vec{a}\,|^2 = |\vec{a}\,|^2 + |\vec{b}\,|^2 - 2 * |\vec{a}\,| * |\vec{b}\,| * \cos(\gamma) \quad \Leftrightarrow$$

$$\cos(\gamma) = \frac{(bx - ax)^2 + (by - ay)^2 + (bz - az)^2 - |\vec{a}\,|^2 - |\vec{b}\,|^2}{-2 * |\vec{a}\,| * |\vec{b}\,|} \quad \Leftrightarrow$$

$$\cos(\gamma) = \frac{|\vec{a}\,|^2 + |\vec{b}\,|^2 - 2 * ax * bx - 2 * ay * by - 2 * az * bz - |\vec{a}\,|^2 - |\vec{b}\,|^2}{-2 * |\vec{a}\,| * |\vec{b}\,|}$$

$$\cos(\gamma) = \frac{ax * bx + ay * by + az * bz}{|\vec{a}\,| * |\vec{b}\,|} \quad \Leftrightarrow$$

$$ax * bx + ay * by + az * bz \quad = \quad |\vec{a}\,| * |\vec{b}\,| * \cos(\gamma)$$

Hierbei wird in der 4. Umformung gleich die binomische Formel ausgeführt und die Summe der Quadrate der Komponenten zur Betragsschreibweise zusammengeführt.

1.10.3 Die Abstände von Punkten, Geraden und Ebenen

Abstand (Punkt, Ebene)

Gegeben: Punkt **p**, Ebene $E = \{ \vec{q} \mid s(\vec{a} - \vec{q}, \vec{n}) = 0 \}$
Gesucht: $dist(p, E) \quad = \quad \mid \vec{p} - \vec{b} \mid$ mit:

1. $\vec{b} = \vec{p} - \lambda * \vec{n}$
2. $s(\vec{a} - \vec{b}, \vec{n}) = 0$

Eingesetzt:

$$s(\vec{a} - \vec{b}, \vec{n}) = 0 \quad \overset{(1)}{\Leftrightarrow} \quad s(\vec{a} - \vec{p} + \lambda * \vec{n}, \vec{n}) = 0 \quad \overset{(2)}{\Leftrightarrow}$$
$$s(\vec{a} - \vec{p}, \vec{n}) + \lambda * s(\vec{n}, \vec{n}) = 0 \quad \overset{(3)}{\Leftrightarrow} \quad \lambda \overset{(3)}{=} -\frac{s(\vec{a} - \vec{p}, \vec{n})}{s(\vec{n}, \vec{n})} \overset{(4)}{=} -\frac{s(\vec{a} - \vec{p}, \vec{n})}{\mid \vec{n} \mid^2}$$

$$dist(p, E) \overset{(5)}{=} \mid \vec{p} - \vec{b} \mid \overset{(1)}{=} \mid \vec{p} - \vec{p} + \lambda * \vec{n} \mid \overset{(6)}{=} \mid \lambda \mid * \mid \vec{n} \mid \quad \overset{(7)}{=} \quad \frac{\mid s(\vec{a} - \vec{p}, \vec{n}) \mid}{\mid \vec{n} \mid}$$

(1): Definition von $\vec{b}$

(2): Additivität und Homogenität des Skalarproduktes

(3): Einfache algebraische Umformung

(4): Definition der Länge eines Vektors unter Verwendung eines Skalarproduktes **s()**, beschrieben im Abschnitt »Skalarprodukte und Längen von Vektoren« auf Seite 65 im Buch

(5): Definition von dist()

(6): Zweite Eigenschaft der Längenfunktion, beschrieben im Abschnitt »Skalarprodukte und Längen von Vektoren« auf Seite 65.

Abstand (Gerade, Gerade)

In dieser Übungsaufgabe betrachten wir lediglich parallele Geraden, das heißt Geraden mit demselben Richtungsvektor $\vec{v}$ und unterschiedlichen Fußpunkten $a, b \in R^3$. Abstände beliebig verlaufender Geraden, die sich nicht schneiden, untersuchen wir im 4. Kapitel im Zusammenhang mit dem *Kreuzprodukt von Vektoren.*
Gegeben: Gerade $G = \{ a + \lambda * \vec{v} \mid \lambda \in R \}$, Gerade $H = \{ b + \eta * \vec{v} \mid \eta \in R \}$
Gesucht: $dist(G, H) \quad = \quad \mid \vec{p} - \vec{b} \mid$ mit:

1. $\vec{p} = \vec{a} + \lambda * \vec{v}$
2. $s(\ \vec{p} - \vec{b},\ \vec{v}\ ) = 0$

Eingesetzt:

$$s(\vec{p}-\vec{b},\ \vec{v})=0 \overset{(1)}{\Leftrightarrow} s(\vec{a}+\lambda*\vec{v}-\vec{b},\ \vec{v})=0 \overset{(2)}{\Leftrightarrow}$$

$$s(\vec{a}-\vec{b},\ \vec{v})+\lambda*s(\vec{v},\ \vec{v})=0 \overset{(3)}{\Leftrightarrow} \lambda=-\frac{s(\vec{a}-\vec{b},\ \vec{v})}{s(\vec{v},\ \vec{v})} \overset{(4)}{=} \frac{s(\vec{b}-\vec{a},\ \vec{v})}{|\vec{v}|^2}$$

$$dist(G,H) \ = \ |\vec{p}-\vec{b}| \ = \ |\ \vec{a} + \frac{s(\vec{b}-\vec{a},\ \vec{v})}{|\vec{v}|^2}*\vec{v} - \vec{b}\ |$$

(1): Definition von $\vec{p}$

(2): Additivität und Homogenität des Skalarproduktes

(3): Einfache algebraische Umformung

(4): *Zähler:* Eigenschaften des Skalarproduktes und Addition von Vektoren, ausführlich beschrieben in **Übungsaufgabe 1.12**. *Nenner:* Definition der Länge eines Vektors unter Verwendung eines Skalarproduktes **s()**, beschrieben im Abschnitt »Skalarprodukte und Längen von Vektoren« auf Seite 65

Abstand (Gerade, Ebene)

Gegeben: Gerade $G=\{\ a+\lambda*\vec{v}\ |\ \lambda\in R\ \}$, Ebene $E=\{\ \vec{q}\ |\ s(\vec{a}-\vec{q},\vec{n})=0\ \}$
Gesucht: $dist(\ G,E\ )$
Da wir uns im $\mathbf{R}^3$ befinden und die untersuchten Mengen nach Voraussetzung keine gemeinsamen Punkte besitzen dürfen, verläuft die Gerade parallel zur Ebene. Daraus folgt, dass *alle* Punkte aus **G** einschließlich dem Fußpunkt **a** in demselben Abstand zu **E** liegen – der gesuchte Abstand ist somit identisch mit dem bereits besprochenen Abstand **dist(a, E)** zwischen dem Fußpunkt **a** und der Gerade **E**.

Abstand (Ebene, Ebene)

Gegeben: Ebene $E=\{\ \vec{q}\ |\ s(\vec{a}-\vec{q},\ \vec{n})=0\ \}$

Ebene $M=\{\ \vec{u}\ |\ s(\vec{b}-\vec{u},\ \vec{m})=0\ \}$

Gesucht: $dist(\ E,M\ )$

Aufgrund derselben Argumentation wie beim Abstand (Gerade, Ebene) gilt die folgende Beziehung, wobei **a** der Fußpunkt der Ebene **E** ist:

$$dist(\ E,M\) \ = \ dist(\ a,M\ )$$

Einführung in die Grafikprogrammierung

2.10 Besprechung der Übungsaufgaben

Übungsaufgabe 2.1:

1. Die Breite des Bildschirms beträgt 640 Pixel, diejenige des Rechtecks 256; beide Abstände zwischen den vertikalen Seiten des Rechtecks und den senkrechten Bildschirmrändern haben zusammen somit eine Länge von **(640 – 256)** Pixel. Weil das Rechteck nach Vorgabe in der Mitte gezeichnet werden soll, haben beide Abstände die gleiche Länge von **(640 – 256) / 2** Pixel. Entsprechendes gilt auch für die Länge. Bis auf die folgenden Anweisungen ist das Programm a2_2 mit seinem Vorgänger identisch:

```
long start_x = (640 - 256) / 2;
long start_y = (480 - 200) / 2;

while( 1 )
{
  if( key_pressed() ) break;

  for( long x=0 ; x<256 ; x++ )
     for( long y=0 ; y<200 ; y++ )
        if( x >= 0 && x < 639 && y >= 0 && y < 479 )
          screen[(y+start_y) * x_res + (x+start_x)]=
                palette[ x ];
}
```

2. Gegeben sei ein Pixel **p=(x, y)** mit $y \leq 478$. Sein Nachbar **q**, der sich auf dem Bildschirm direkt unter **p** befindet, besitzt demnach die Koordinaten **(x, y+1)**. Berechnet man die Positionen der Farbkomponenten beider Pixel im Videospeicher, ergibt sich:

$$offset(q) = (y+1) * 640 + x = (y * 640 + x) + 640 = offset(p) + 640$$

Daraus folgt, dass die Farbposition von **q** lediglich durch die Addition der horizontalen Bildschirmauflösung zur bekannten Farbposition von **p** ermittelt werden kann. Die optimierte Version der beiden Schleifen des Programms a2_1 besitzt demnach folgendes Aussehen:

```
for( long x=0 ; x<256 ; x++ )
{
  long offset = x;
  for( long y=0 ; y<480 ; y++, offset += 640 )
     screen[ offset ] = palette[ x ];
}
```

3. Wenn eine Farbpalette mit **256** Elementen vier gleich lange Farbverläufe enthalten soll, stehen jedem Verlauf **64** Farben zur Verfügung. Eine rote, grüne oder blaue Komponente einer Farbe kann insgesamt **256** unterschiedliche Werte annehmen; verteilt man diese auf die **64** Positionen, muss jeder Komponente die Konstante **256 / 64 = 4** hinzuaddiert werden, um den Wert der entsprechenden Komponente der jeweils nächsten Farbe des Verlaufes zu erhalten:

```
pixel_32 palette[ 256 ];

void load_palette( void )
{
  long x;  uchar c;

  for( x=0, c=0 ; x<64 ; x++, c+=4 )
  {
    palette[ x     ] = pixel_32( c, 0, 0 );
    palette[ x+64  ] = pixel_32( c, c, c );
    palette[ x+128 ] = pixel_32( 0, c, 0 );
    palette[ x+192 ] = pixel_32( 0, 0, c );
  }
}
```

Übungsaufgabe 2.2:

1. Für die Farbposition **offset** existieren zwei natürliche Zahlen **(x, y)** in den vorgegebenen Grenzen mit **offset = y * 640 + x**. Weil die Zahl **x** echt kleiner **640** ist, gilt: **x = offset % 640**; mit anderen Worten ist **x** der Rest, der bei der Integerteilung von **offset** durch **640** entsteht. Der Wert von **y** lässt sich direkt durch Ganzzahl-Division ermitteln: **y = offset / 640**.
2. In den Übungsaufgaben des ersten Kapitels wurde die Menge **L** aller Punkte definiert, die sich auf einer Linie zwischen zwei gegebenen Anfangspunkten befinden. Dieselbe Idee, die bei der Definition von **L** eine grundlegende Rolle spielt, lässt sich auch hier einsetzen: Angenommen, man besitzt eine Zufalls-

zahl **t**, deren Wert zwischen **0** und **1** liegt, lassen sich alle Zahlen zwischen **a** und **b** durch den Ausdruck **a + t *(b – a)** erreichen. Da `RAND_MAX` ungleich **0** ist, lässt sich **t** mit Hilfe von **rand()** berechnen:

$$c = a + \frac{rand()}{RAND\_MAX} * (b - a)$$

Übungsaufgabe 2.3:

1. Eine einfache Möglichkeit der Verwaltung von Koordinaten zweidimensionaler Punkte besteht in der Definition einer entsprechenden Struktur:

```
struct svertex
{
  long sx, sy;

  svertex( long x, long y ) : sx( x ), sy( y ) { }
};
```

Wird das vorgegebene Dreieck **D** in einem Array aus Elementen vom Typ `svertex` gespeichert, kann mit der im Text hergeleiteten Formel auf einfache Weise ein zufälliger Punkt ausgewählt werden – Initialisierung sowie die beiden Schritte des Chaosspiels lassen sich somit folgendermaßen implementieren:

```
svertex points[ 3 ] =
{
  svertex(  95, 435 ),
  svertex( 320,  45 ),
  svertex( 545, 435 )
};

svertex act_point = points[ rand() % 3 ];

while( 1 )
{
  if( key_pressed() ) break;

  uchar index = rand() % 3;

  act_point.sx =
    (act_point.sx + points[ index ].sx) / 2;
  act_point.sy =
```

```
    (act_point.sy + points[ index ].sy) / 2;

  screen[ act_point.sy * 640 + act_point.sx ] =
    palette[ 15 ];

  for( double wait=0 ; wait<999999 ; wait++ ) ;
}
```

Der Mittelpunkt der Strecke zwischen **act_point** und dem zweiten zufällig ausgesuchten Punkt, der im zweiten Schritt des Chaosspiels berechnet wird, lässt sich durch die gewöhnliche Quersumme berechnen, nacheinander auf die x- und auf die y-Koordinaten angewandt.

Aufgrund der passiven Wiederholungsanweisung dauert es eine gewisse Zeit, bis die hohe Symmetrie des Sierpinski-Dreiecks erkennbar wird.

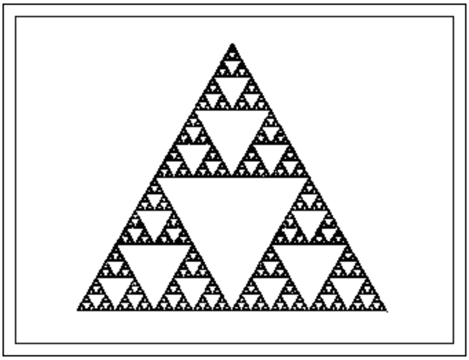


Abb. 2.1: Erste Ausgabe des Programms *a2_4:* Das Sierpinski-Fraktal in einem gleichseitigen Dreieck

2. Die Darstellung des Sierpinski-Fraktals in einem beliebigen Dreieck lässt sich durch die Verwendung einer Funktion `draw_triangle()` vereinfachen, die aus den entsprechenden Anweisungen des vorherigen Programms aufgebaut ist:

```
void draw_triangle
(
  svertex points[ 3 ], uchar color, pixel_32 *screen
)
{
  svertex act_point = points[ rand() % 3 ];

  for( long a=0 ; a<100000 ; a++ )
```

```
  {
    uchar index = rand() % 3;

    act_point.sx
      = (act_point.sx + points[ index ].sx) / 2;
    act_point.sy
      = (act_point.sy + points[ index ].sy) / 2;

    screen[ act_point.sy * 640 + act_point.sx ] =
      palette[ color ];
  }
}
```

Initialisierung des Arrays `points[]` sowie der Aufruf von `draw_triangle()` erfolgen innerhalb der Hauptschleife der Funktion `WinMain()`:

```
svertex points[ 3 ];
while( 1 )
{
  for( long a=0 ; a<4 ; a++ )
  {
    for( long x=0 ; x<3 ; x++ )
       points[ x ] =
         svertex( rand() % 640, rand() % 480 );

    draw_triangle( points, rand() % 250, screen );

    for( double wait=0 ; wait<22222222 ; wait++ )
       if( key_pressed() )
       {
         screen_interface.release_screen_pointer();
         return msg.wParam;
       }
  }

  for( long x=0 ; x<640*480 ; x++ )
     screen[ x ] = palette[ 0 ];
}
```

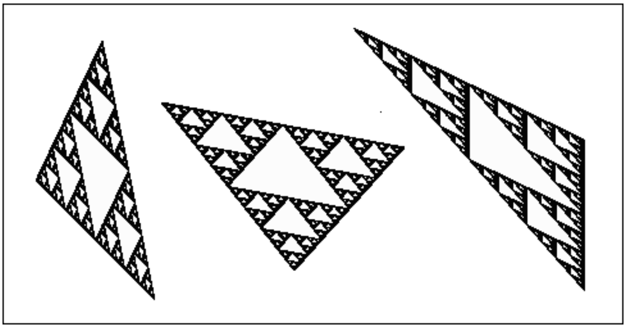


Abb. 2.2: Zweite Ausgabe des Programms *a2_4:* Das Sierpinski-Fraktal in beliebigen Dreiecken

Bemerkenswert ist, dass die Abfrage der Tastatur sowie die beiden Anweisungen, die für gewöhnlich am Ende unserer Programme stehen, hier im Rumpf der passiven Wiederholungsanweisung zu finden sind. Gibt es einen besonderen Grund für diese Vorgehensweise?

Übungsaufgabe 2.4:

$$M = \{\ \alpha,\ \beta,\ \gamma,\ \delta\ \}$$
$$E = (\ B,\quad M,\quad \{ (1,\alpha), (1,\gamma), (2,\beta), (2,\gamma) \}\quad)$$
$$(E \circ D) = (\ A,\quad M,\quad \{ (v,\alpha), (v,\gamma), (w,\alpha), (w,\gamma), (w,\beta) \}\quad)$$

Übungsaufgabe 2.5:

1. Die Anfangs- und Endkoordinaten der darzustellenden Linie werden mit Hilfe der beiden Variablen `begin` und `end` vom Typ `svertex` verwaltet. Damit der Anfangspunkt der nächsten Linie identisch mit dem Endpunkt der gerade gezeichneten Strecke ist, wird `begin` am Ende der Hauptschleife der Wert von `end` zugewiesen, während `end` neue, zufällige Koordinaten erhält:

```
svertex begin = svertex( rand() % 640, rand() % 480 );
svertex end = svertex( rand() % 640, rand() % 480 );

uchar running = 1;
while( running )
{
  draw_line( begin, end, palette[ rand() % 256 ], screen );

   for( double wait = 0 ; wait < 9999999 ; wait++ )
      if( key_pressed() ) running = 0;
```

```
  begin = end;
  end = svertex( rand() % 640, rand() % 480 );
}
```

In bestimmten Fällen ist das Platzieren von Tastaturabfragen innerhalb von Wiederholungsanweisungen, wie in diesem Programm demonstriert, unbedingt erforderlich. Angenommen, die Warteschleife wird innerhalb von `draw_line()` nach dem Setzen jedes Pixels ausgeführt – gehören diese Pixel zu einer längeren Strecke und findet die Tastaturabfrage außerhalb von `draw_line()` statt, müsste der Benutzer nach dem Tastendruck viel zu lange warten, bis das Programm schließlich beendet ist.

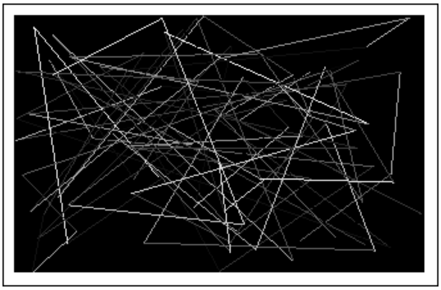


Abb. 2.3: Ausgabe des Programms *a2_5*

2. $a + 0.5(b - a) = a + 0.5b - 0.5a = 0.5a + 0.5b = (a + b)/2$

3. Nach dem Satz des Pythagoras gilt für die Höhe **h** eines gleichseitigen Dreiecks mit der Seitenlänge **a**:

$$h^2 + \left(\frac{a}{2}\right)^2 = a^2 \quad \Rightarrow \quad h = \frac{\sqrt{3}}{2} * a$$

Die Punkte **p1** und **p3** können nach Vorgabe die untere Seite des Dreiecks bilden, die parallel zum Bildschirm verläuft, während **p2** in der Mitte über dieser Seite liegt. Da die Höhe senkrecht auf der unteren Seite steht und der Punkt **(320, 240)** in der Mitte von h zu finden ist, ergeben sich folgende Bildschirmkoordinaten:

p1 = [(320–0.5*a), (240+0.5*h)]
p2 = [320, (240–0.5*h)]
p3 = [(320+0.5*a), (240+0.5*h)]

Übungsaufgabe 2.6:

1. Alle drei Kategorien von Linien lassen sich problemlos mit der hergeleiteten Funktion `draw_line()` auf dem Bildschirm zeichnen. Die horizontale Linie

wird wie eine langsam steigende Gerade behandelt, und in jedem Schleifendurchlauf wird die x-Koordinate des darzustellenden Pixels erhöht. Weil `delta_y` den Wert **0** besitzt, während `delta_x` größer **0** ist, nimmt die Bedingung `e >= delta_x` stets den Wert `FALSE` an und die y-Koordinaten der gezeichneten Pixel bleiben unverändert, so dass alle innerhalb derselben Pixelzeile bleiben.

Entsprechendes gilt auch für die senkrechte Linie, die aufgrund der Anfangsbeziehung `delta_x < delta_y` wie eine schnell steigende Gerade behandelt wird. Im Fall der diagonal verlaufenden Linie mit der Steigung **m = 1.0** besitzen `delta_y` und `delta_x` denselben Wert; aus diesem Grund ist die Bedingung `e >= delta_x` stets erfüllt, und in jedem Schleifendurchlauf werden sowohl die x- als auch die y-Koordinaten erhöht.

2. Das Programm `a2_8` ist nach demselben Prinzip wie sein direkter Vorgänger aufgebaut. Damit der Benutzer die Darstellung der Linien verfolgen kann, wird die Wiederholungsanweisung:

```
for( double wait = 0 ; wait < 99999 ; wait++ )
   if( key_pressed() ) {  running = 0;  return;  }
```

nach dem Setzen jedes Pixels in der Funktion `draw_line()` ausgeführt. Drückt der Benutzer eine beliebige Taste, wird der globalen Variable `running` der Wert **0** zugewiesen, und `draw_line()` mit `return` abgebrochen. Weil `running` in der Kontrollanweisung der Hauptschleife `while( running == 1 )` des Programms auftritt, wird dadurch das Programm beendet.

Übungsaufgabe 2.7:

1. Da eine achsenparallele Ellipse mit dem Mittelpunkt **(mx, my)** nichts anderes als ein Kreis ist, dessen x- und y-Koordinaten mit unterschiedlichen Werten **a** und **b** skaliert worden sind, ergibt sich für **e()** folgende Definition:

$$\begin{aligned} e &: [\,0 \;..\; 2*\pi\,] \rightarrow R^2 \\ &\delta \mapsto [\quad a*cos(\delta) + mx, \quad b*sin(\delta) + my \quad] \end{aligned}$$

Auch hierbei gilt, dass der Definitionsbereich von **e()** auch ganz **R** sein darf.

2. Funktionen können auch bei der Definition von Mengen eingesetzt werden; als praktisches Beispiel wird die Ellipse aus Abbildung 2.17 im Buch unter Verwendung von **e()** konstruiert, wobei **(mx, my) = (0, 0)**.

Der Sinus- und der Kosinuswert eines festen Winkels geben nach Definition die Längen der Katheten eines rechtwinkligen Dreieckes an, dessen Hypotenuse die Länge **1.0** hat. Nach dem Satz von Pythagoras gilt demnach, dass die Summe der Quadrate dieser beiden Werte ebenfalls **1.0** ist. Für **E** gilt somit:

$$E = \{ \quad (a * cos(\delta), b * sin(\delta)) \quad | \quad \delta \in R \quad \}$$
$$p = (px, py) \in E \iff es\ existiert\ ein\ \delta \in R\ mit\ px = a * cos(\delta)\ und$$
$$py = b * sin(\delta) \iff 1 = sin(\delta)^2 + cos(\delta)^2 = \frac{b^2 * sin(\delta)^2}{b^2} + \frac{a^2 * cos(\delta)^2}{a^2}$$
$$\iff \quad 1 = \frac{px^2}{a^2} + \frac{py^2}{b^2}$$

Hierbei wird vorausgesetzt, dass die Längen **a** und **b** positive reelle Zahlen ungleich **0.0** sind.

3. Die bekannte Version der Funktion `draw_funktion()` braucht nur geringfügig verändert zu werden, um Funktionen von **R** nach **R**^{2} darstellen zu können:

```
svertex project( vertex p )
{
  long sx = long( (p.wx - begin_x) *
                  (double( x_res-1 ) /
                  (end_x - begin_x)) );
  long sy = long( (p.wy - end_y) *
                  (double( y_res-1 ) /
                  (begin_y - end_y)) );

  return svertex( sx, sy );
}

void draw_function
(
  vertex (*f)( double x ), double start, double end,
  pixel_32 color, pixel_32 *screen
)
{
  svertex p = project( f( start ) );

  double dl = 0.01;
  for( double a = start + dl ; a <= end ; a += dl )
  {
    svertex q = project( f( a ) );
    draw_line( p, q, color, screen );

    p = q;
  }
}
```

Da bei Funktionen von **R** nach **R**^{2} keine direkte Beziehung mehr zwischen der x-Achse und dem Funktionsgraph besteht, wird der Detaillevel in `draw_function()` wieder direkt in Form der Variablen `dl` angegeben.

Hat man eine dieser Funktionen definiert:

```
vertex e( double alpha )
{
  double wx = 3 * cos( alpha ) + 4;
  double wy = 2 * sin( alpha ) + 3;

  return vertex( wx, wy );
}
```

lässt sich `draw_funktion()` in `WinMain()` wie folgt aufrufen:

```
draw_function( e, 0, 7, pixel_32(0, 0, 255), screen );
```

Übungsaufgabe 2.8:

1. Die Werte **begin_x** < **end_x** können beliebig gewählt werden; weil die horizontale Auflösung **x_res** ungleich **y_res** ist, müssen **begin_y** und **end_y** unter Verwendung dieser Konstanten berechnet werden. Für die rechte und obere Begrenzung gilt hierbei: Je größer **end_x** ist, desto länger ist der Abstand zwischen zwei waagerecht direkt nebeneinander liegenden Pixeln, desto größer muss aufgrund der geforderten Einheitengleichheit auch der Wert von **end_y** sein. Hierbei handelt es sich um eine proportionale Zuordnung:

Mathematische Angaben	Bildschirmangaben
end_x	x_res
end_y	y_res

$$\Rightarrow \qquad end\_y = \frac{end\_x}{x\_res} * y\_res \qquad begin\_y = \frac{begin\_x}{x\_res} * y\_res$$

Die Berechnung von **begin_y** erfolgt unter Verwendung einer ähnlichen Zuordnung. Hierbei handelt es sich um eine sehr einfache Variante zur Festlegung dieser mathematischen Begrenzungen, bei der man die Position des Ursprunges auf dem Bildschirm nicht direkt festlegen kann.

Möchte man diese Position dennoch selbst festlegen, muss neben **begin_x** und **end_x** auch **begin_y** angegeben sein; die dazugehörige Zuordnung wird nach derselben Grundidee aufgestellt:

Mathematische Angaben	Bildschirmangaben
$end\_x - begin\_x$	x_res
$end\_y - begin\_y$	y_res

$$\Rightarrow \qquad end\_y = \frac{end\_x - begin\_x}{x\_res} * y\_res + begin\_y$$

2. Um einen Kreis in eine Spirale umzuwandeln, müssen lediglich beide Koordinatenfunktionen mit dem Eingabewert α multipliziert werden, der auch den Rotationswinkel angibt. Für $\tau = \eta$ entsteht eine runde Spirale, ungleiche Werte ergeben eine in die Breite bzw. Länge gedehnte Figur.

$$\begin{aligned} e \;:\; & R_0^+ \;\rightarrow\; R^2 \\ & \alpha \;\mapsto\; [\quad \tau * \alpha * r * cos(\alpha), \quad \eta * \alpha * r * sin(\alpha) \quad] \end{aligned}$$

Übungsaufgabe 2.9:

Das Programm befindet sich auf der beiliegenden CD im Verzeichnis `a2_10`.

Übungsaufgabe 2.10:

Die drei Zykloiden lassen sich wie gehabt als Funktionen von **R** nach $\mathbf{R}^2$ definieren, die anschließend mit Hilfe von `draw_function()` auf dem Bildschirm gezeichnet werden können.

```
svertex z( double alpha )
{
  double t = 1;

  return svertex
  (
    t*r * sin( alpha ) + alpha*r,
    t*r * cos( alpha ) + r
  );
}

svertex zs( double alpha )
{
  double t = -0.3;

  return svertex
```

```
  (
    t*r * sin( alpha ) + alpha*r,
    t*r * cos( alpha ) + r
  );
}

svertex zb( double alpha )
{
  double t = 1.72;

  return svertex
  (
    t*r * sin( alpha ) + alpha*r,
    t*r * cos( alpha ) + r
  );
 }
```

Übungsaufgabe 2.11:

Die Definition der gesuchten Funktion `zk()` wird während der Besprechung des vierten Schrittes des Projekts `a2_14` angegeben.

```
svertex zk( double alpha )
{
  double s = 3;
  double r = 1;
  double t = 1.5;

  double beta = (alpha * r) / s;

  double sx = t*r * sin( alpha ) * cos( beta ) -
           (t*r * cos( alpha ) - s + r) * sin( beta );

  double sy = t*r * sin( alpha ) * sin( beta ) +
           (t*r * cos( alpha ) - s + r) * cos( beta );

  return svertex( sx, sy );
}
```

Übungsaufgabe 2.12:

```
struct vertex
{
  double wx, wy;

  vertex( void ) : wx( 0 ), wy( 0 ) { }
  vertex( double x, double y ) : wx( x ), wy( y ) { }
};

double begin_x = -7,    end_x = 7;
double begin_y = -4.4,  end_y = 4.4;

vertex k[ 4 ] =
{
  vertex( -4,  0 ), vertex( 6, 4 ),
  vertex( -6, -4 ), vertex( 4, 0 )
};

vertex b( double t )
{
  double bs0 = (1-t)*(1-t)*(1-t);
  double bs1 = 3*t*(1-t)*(1-t);
  double bs2 = 3*t*t*(1-t);
  double bs3 = t*t*t;

  double wx = bs0 * k[ 0 ].wx + bs1 * k[ 1 ].wx +
              bs2 * k[ 2 ].wx + bs3 * k[ 3 ].wx;
  double wy = bs0 * k[ 0 ].wy + bs1 * k[ 1 ].wy +
              bs2 * k[ 2 ].wy + bs3 * k[ 3 ].wy;

  return vertex( wx, wy );
}
```

Um aufeinanderfolgende bzw. innerhalb des Arrays `k[]` nebeneinanderliegende Kontrollpunkte mit Linien zu verbinden, kann folgende Schleife eingesetzt werden:

```
for( char x=0 ; x<3 ; x++ )
   draw_line
   (
     project( k[ x ] ), project( k[ x+1 ] ),
     pixel_32( 0, 0, 150 ), screen
   );
```

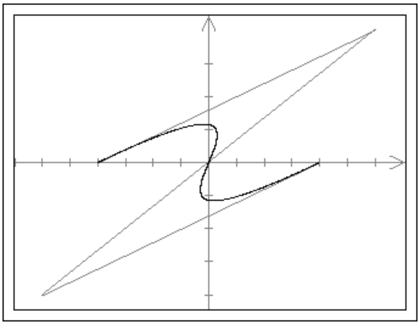


Abb. 2.4: Ausgabe des Programms `a2_15_1`: Der Graph gewöhnlicher Bézierkurven wird weitaus schwächer von den Kontrollpunkten angezogen als man zunächst erwarten würde.

Übungsaufgabe 2.13:

1. $c = b_L(\frac{1}{5})$ $\quad d = b_M(\frac{5}{7})$ $\quad q = b_M(\frac{6}{7})$
2. Die allgemeine Formel zur Darstellung von Bézierkurven kann sehr leicht durch das Ersetzen des Summenzeichens durch eine `for()`-Schleife implementiert werden.

```
double n_choose_i( long n, long i )
{
  return fac( n ) / (fac( i ) * fac( n - i ));
}

vertex b( double t )
{
  double wx = 0;
  double wy = 0;
```

```
  for( long i=0 ; i <= n ; i++ )
  {
    double bsi =
      n_choose_i( n, i )*pow( t, i ) * pow( 1-t, n-i );

    wx += bsi * k[ i ].wx;
    wy += bsi * k[ i ].wy;
  }

  return vertex( wx, wy );
}
```

Für die Berechnung der Exponentialfunktion kann die vordefinierte Funktion `pow()` eingesetzt werden, die in `<math.h>` wie folgt deklariert ist:

```
double pow( double base, double exp );
```

Hierbei gilt: **pow(x, y) =** x^y für reelle Zahlen **x** und **y**.

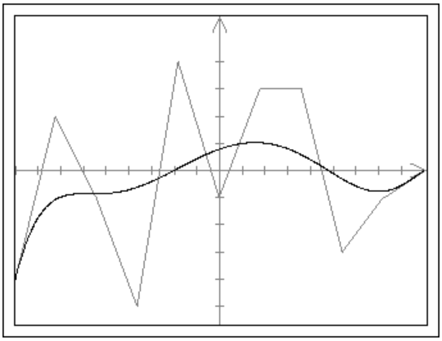


Abb. 2.5: Ausgabe des Programms *a2_15*

3. Die vorgegebene Funktion f() hebt sechs Punkte aus dem Intervall [0 .. 1] hervor, die in demselben Abstand voneinander entfernt sind. Hierbei handelt es sich um den Anfangs- und Endpunkt sowie die vier lokalen Extrema. Der Einsatz einer Bézierkurve mit sechs Kontrollpunkten ist somit naheliegend.

   Die Dimension der Kontrollpunkte gibt die Dimension des Punktes an, der von der Bézierkurve für ein bestimmtes $\mathbf{t} \in [\,\mathbf{0..1}\,]$ ausgerechnet wird. Da in unserem Fall f() eine reelle Zahl ausgibt, müssen die sechs Kontrollpunkte ebenfalls eindimensional sein und somit Höhen auf der y-Achse angeben.

   Angenommen, die y-Koordinate des ersten Extrempunktes hat den Wert $\mathbf{h} \in \mathbf{R}$ mit $h > 0$. Wie man anhand der Abbildung 2.37 im Buch erkennen kann, betragen die y-Koordinaten der drei anderen Extrema 0.5 * h, 0.75 * h und –0.5 * h; die gesuchte Menge **K** der Kontrollpunkte kann somit als:

K = { 0, h, 0.5*h, 0.75*h, –0.5*h, 0 }

definiert werden. Die Bézierkurve, die auf der Grundlage dieser Kontrollpunkte auf dem Bildschirm dargestellt wird, hat jedoch wenig Ähnlichkeit mit der Vorgabe aus Abbildung 2.37 im Buch – der Grund für diese fehlerhafte Ausgabe ist die **Ungenauigkeit** der Rechenvorschrift der Bézierkurven.

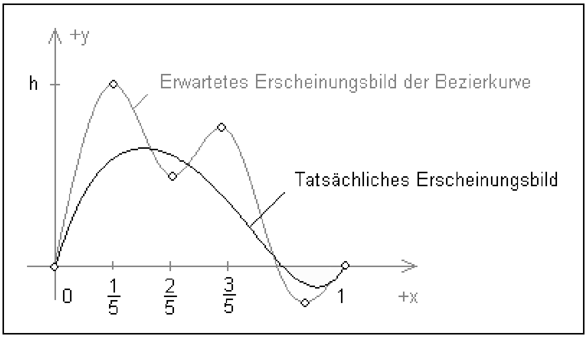


Abb. 2.6: Die *Variation Diminishing Property* verhindert, dass eine Kurve mit dem gewünschten Erscheinungsbild mit Hilfe einer einfachen Bézierkurve generiert werden kann.

Sind zu viele Details angegeben, wird die Kurve gleichzeitig in verschiedenen Richtungen von den dort positionierten Kontrollpunkten angezogen; die Details heben sich somit praktisch gegenseitig auf und werden im Verlauf der Kurve somit nicht angezeigt. Dieses Problem besitzt eine eigene Bezeichnung: das *Phänomen der Detailverringerung* bzw. die *Variation Diminishing Property*.

Um unser Anfangsproblem dennoch lösen zu können, das heißt eine Funktion zu generieren, deren Graph dieselben Merkmale wie die Kurve aus Abbildung 2.37 im Buch aufweist, benötigt man eine genauere Rechenvorschrift – diese Vorgabe erfüllen die **Splines**.

Übungsaufgabe 2.14:

Erste Gleichung:

$$(6) \quad \Rightarrow \quad 2 * c_0 = 0 \quad \Leftrightarrow \quad c_0 = 0 \quad \Leftrightarrow \quad 3 * p_1 - 3 * p_0 - 2 * s_0 - s_1 = 0$$

$$\Leftrightarrow \quad 3 * p_1 - 3 * p_0 = 2 * s_0 + s_1$$

Zweite Gleichung:

$$
\begin{aligned}
&(7) \quad \Rightarrow \quad 2*c_3+6*d_3=0 \quad \Leftrightarrow \quad c_3+3*d_3=0 \\
&\Leftrightarrow \quad 3*p_4-3*p_3-2*s_3-s_4+6*p_3-6*p_4+3*s_3+3*s_4=0 \\
&\Leftrightarrow \quad -3*p_4+3*p_3+s_3+2*s_4=0 \\
&\Leftrightarrow \quad s_3+2*s_4=3*p_4-3*p_3
\end{aligned}
$$

Übungsaufgabe 2.15:

$$
\left.\begin{array}{lcll}
0*s_0+1*s_1+v_1*s_2+0*s_3+0*s_4 &=& q_1 & | \\
0*s_0+1*s_1+4*s_2+1*s_3+0*s_4 &=& y_2 &
\end{array}\right\}\textit{Subtraktion der Zeilen}
$$

$$
\begin{array}{lcll}
0*s_0+1*s_1+v_1*s_2+0*s_3+0*s_4 &=& q_1 & \\
0*s_0+0*s_1+(4-v_1)*s_2+1*s_3+0*s_4 &=& y_2-q_1 & | \quad *\dfrac{1}{4-v_1}
\end{array}
$$

$$
\begin{array}{lcl}
0*s_0+1*s_1+v_1*s_2+0*s_3+0*s_4 &=& q_1 \\
0*s_0+0*s_1+1*s_2+\dfrac{1}{4-v_1}*s_3+0*s_4 &=& (y_2-q_1)*\dfrac{1}{4-v_1}
\end{array}
$$

Übungsaufgabe 2.16:

1. Die Grundlage einer Möglichkeit zur Darstellung der Kontrollpunkte einer Spline bieten die Voraussetzungen **(1)** und **(2)**, wonach die Anfangs- und Endpunkte benachbarter Teilpolynome identisch sein müssen. Kennt man **f_i()**, lassen sich somit die ersten **(n – 1)** Kontrollpunkte einzeichnen, indem man die Grenzen der Teilintervalle als x- und die Werte **f_i(0)** als y-Koordinaten verwendet. Die Koordinaten des letzten Kontrollpunktes **p_{n-1}** lauten schließlich: **p_{n-1} =** [**end_x**, **f_{n-2}(1)**].

```
void spline::draw_points( double start, double end )
{
  glPointSize( 5 );
  glColor3d( 1, 0, 0 );

  glBegin( GL_POINTS );

    for( long i=0 ; i<point_count-1 ; i++ )
    {
     double act_x = start + i * ( (end - start) /
                (point_count - 1) );
```

```
     glVertex2d( act_x, f[ i ].execute( 0 ) );
    }

   glVertex2d( end, f[ point_count-2 ].execute(1) );

  glEnd();
}
```

2. Die im Text beschriebene Darstellungsfunktion berechnet 100 Punkte auf der Kurve jedes Teilpolynoms, die aufgrund des Befehls `GL_LINE_STRIP` mit Linien verbunden werden. Um lediglich vier Zwischenpunkte einzuzeichnen, müssen 5 Punkte berechnet und mit `GL_POINTS` eingezeichnet werden:

```
void spline::draw( double start, double end )
{
  double int_length = (end-start) / (point_count-1);

  glColor3ub( 255, 255, 0 );

  for( long i=0 ; i<point_count-1 ; i++ )
  {
   glBegin( GL_POINTS );

      for( double t=0 ; t<=1 ; t+=0.2 )
      {
        double act_x = start + i*int_length +
                                t*int_length;

        glVertex2d( act_x, f[ i ].execute( t ) );
      }

    glEnd();
  }

  draw_points( start, end );
}
```

Weil die Zählvariable `t` mit dem Wert `0` initialisiert und in der Kontrollanweisung der Operator <= auftritt, werden die Kontrollpunkte mehrfach gezeichnet.

Übungsaufgabe 2.17:

Die Darstellung der Kontrollpunkte zweidimensionaler Splines erfolgt nach demselben Prinzip wie bei den eindimensionalen Splines:

```
void spline::draw_points( void )
{
  glPointSize( 5 );
  glColor3ub( 255, 0, 0 );

  glBegin( GL_POINTS );

    for( long i=0 ; i<point_count-1 ; i++ )
      glVertex2d
      (
        xf[ i ].execute( 0 ), yf[ i ].execute( 0 )
      );

   glVertex2d
   (
     xf[ point_count-2 ].execute( 1 ),
     yf[ point_count-2 ].execute( 1 )
   );

  glEnd();
}
```

2.11 Besprechung der Projekte

Projekt a2_7, erster Schritt:

```
////////////////////        a2_7_1          ////////////////////
//                                                            //
//  Darstellung der Funktionsgraphen reeller Polynome         //
//  Darstellungsart: Software                                 //
//                                                            //
////////////////////////////////////////////////////////////////

double begin_x = -9,  end_x = 14;
double begin_y = -3,  end_y =  4;

int WINAPI WinMain
```

```
(
  HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int iCmdShow
)
{
  screen_interface.open_window( hInstance, 640, 480, 32 );
  pixel_32 *screen = (pixel_32 *)
    screen_interface.get_screen_pointer();

  while( 1 )
  {
    if( key_pressed() ) break;

    draw_xy( pixel_32( 128, 128, 128 ), screen );

    draw_function( f, begin_x, end_x,
                   pixel_32( 255, 255, 0 ), screen );
    draw_function( df, begin_x, end_x,
                   pixel_32( 255, 0, 0 ), screen );
    draw_function( ddf, begin_x, end_x,
                   pixel_32( 0, 0, 255 ), screen );
  }

  screen_interface.release_screen_pointer();

  return msg.wParam;
}

//////////////////        Ende a2_7_1        ///////////////////
```

Im Mittelpunkt dieses ersten Programms steht die Funktion `draw_function()`, welche die darzustellende Funktion `f()` als Parameter entgegennimmt, und ihren Graph im Intervall `[ start .. end ]` auf dem Bildschirm zeichnet:

```
void draw_function
(
  double (*f)( double x ), double start, double end,
```

```
  pixel_32 color, pixel_32 *screen
)
{
  for( double x = start ; x <= end ; x += 0.01 )
  {
    svertex p = project( x, f( x ) );

    if
    (
       p.sx >= 0 && p.sx < x_res &&
       p.sy >= 0 && p.sy < y_res
    )
    screen[ p.sy * 640 + p.sx ] = color;
  }
}
```

Funktionen werden in C++ stets in Form von Zeigern angegeben; das gilt selbst für die einfachsten Programme. Dieser Umstand ist von besonderer Wichtigkeit, wenn man wie im vorliegenden Fall einer Funktion `draw_function()` eine weitere Funktion `f()` als Parameter übergibt: Während man sich im Fall von Variablen frei aussuchen darf, ob die Übergabe *by Value* oder *by Reference* stattfinden soll, besteht diese Wahlmöglichkeit bei Funktionen nicht – aus dem oben genannten Grund werden diese immer *by Reference* übergeben.

Dieser Umstand ist im Prototypen von `draw_function()` zu berücksichtigen. Der komplexe Ausdruck: `double (*f)( double x )`, wird von innen nach außen gelesen in der von der Priorität der Operatoren angegebenen Reihenfolge: `f` ist ein Zeiger auf eine Funktion, die ein Parameter vom Typ `double` entgegennimmt und einen Rückgabewert vom Typ `double` ausgibt.

Da Funktionen stets in Form von Zeigern angegeben werden, braucht beim Aufruf von `draw_function()` der Adressoperator `&` nicht eingesetzt zu werden:

```
draw_function( f, -1.0, 1.0, pixel_32( 255, 255, 0 ), screen );
```

Ebenso lässt sich der Zeiger `(*f)` während der Ausführung von `draw_function()` wie eine ganze gewöhnliche Funktion behandeln:

```
double y = f( x );
svertex p = project( a, y );
```

Neben der Funktion `f()` selbst muss `draw_function()` noch der Definitionsbereich von `f()` übergeben werden. Diese Vorgehensweise ist wichtig, denn dieser Definitionsbereich unterscheidet sich in einigen Fällen von dem Abschnitt auf der x-Achse, der von **begin_x** und **end_x** angegeben wird.

Der Grund hierfür besteht darin, dass einige Funktionen nicht für alle möglichen reellen Zahlen definiert sind: Die Wurzelfunktion `double sqrt( double x )` darf beispielsweise nicht mit negativen Werten **x < 0** aufgerufen werden; die inverse Kosinusfunktion `double acos( double c )` darf nur für $-1 \leq c \leq +1$ eingesetzt werden, da Kosinuswerte nach Definition niemals kleiner **–1.0** bzw. größer **+1.0** sein können.

Pixel dürfen erst dann gezeichnet werden, wenn es sich bei den projizierten Werten `(sx, sy)` um gültige Bildschirmkoordinaten handelt. Die hierzu notwendige Anweisung ist bereits aus `a2_2` bekannt:

```
if( p.sx >= 0 && p.sx < x_res && p.sy >= 0 && p.sy < y_res )
  screen[ p.sy * x_res + p.sx ] = color;
```

Diese Überprüfung ist besonders wichtig, denn die Graphen der Gelb und Rot dargestellten Polynome verlassen in der Nähe von **begin_x** und **end_x** den durch die Grenzen **begin_y** und **end_y** definierten Bereich. Da auch `draw_line()` Pixel auf dem Bildschirm zeichnet, muss auch dort eine ähnliche Überprüfung stattfinden:

```
if
(
  begin.sx < 0 || begin.sx >= x_res ||
  end.sx < 0   || end.sx >= x_res   ||
  begin.sy < 0 || begin.sy >= y_res ||
  end.sy < 0   || end.sy >= y_res
)
return;

screen[ offset ] = c;
```

Dieser Ausdruck ist zwar aufgrund der Multiplikation aufwändig; eine elegantere Methode, die auf dasselbe Ergebnis hinausläuft, werden wir im Zusammenhang mit dem *Polygon Clipping Algorithmus* kennen lernen.

Die Koordinatenachsen verlaufen durch den Punkt **(0, 0)**; um diese schließlich darstellen zu können, müssen lediglich die Pixel verbunden werden, deren mathematische Koordinaten die Werte **(begin_x, 0)**, **(end_x, 0)**, **(0, begin_y)**, **(0, end_y)** besitzen:

```
void draw_xy( pixel_32 color, pixel_32 *screen )
{
  draw_line( project( begin_x, 0 ), project( end_x, 0 ),
             color, screen );
  draw_line( project( 0, begin_y ), project( 0, end_y ),
             color, screen );
}
```

Projekt a2_7, zweiter Schritt:

Um die aus zwei Linien bestehende Pfeilspitze der x-Achse zu zeichnen, werden zunächst die Bildschirmkoordinaten des letzten sichtbaren Pixels ermittelt:

```
svertex right = project( end_x, 0 );
```

Von diesen Koordinaten ausgehend, bewegt man sich 11 Pixel nach links und jeweils 5 Pixel nach oben und nach unten, um diese Linien schließlich darzustellen:

```
draw_line( svertex( right.sx-11, right.sy-5 ), right,
           color, screen );
draw_line( svertex( right.sx-11, right.sy+5 ), right,
           color, screen );
```

Dieselbe Grundidee wird auch bei der Einteilung beider Achsen eingesetzt; hier sorgen `if()`-Anweisungen dafür, dass die waagerechten und senkrechten Linien die zuvor gezeichneten Pfeilspitzen nicht überschreiben:

```
void draw_xy( pixel_32 color, pixel_32 *screen )
{
  draw_line( project( begin_x, 0 ), project( end_x, 0 ),
             color, screen );
  draw_line( project( 0, begin_y ), project( 0, end_y ),
             color, screen );

  svertex right = project( end_x, 0 );
  svertex top = project( 0, end_y );

  draw_line( svertex( right.sx-11, right.sy-5 ), right,
             color, screen );
```

```
  draw_line( svertex( right.sx-11, right.sy+5 ), right,
             color, screen );
  draw_line( svertex( top.sx-5, top.sy+11 ), top, color,
             screen );
  draw_line( svertex( top.sx+5, top.sy+11 ), top, color,
             screen );

  for( long x = long( begin_x ) ; x <= long( end_x ) ; x++ )
  {
    svertex p = project( x, 0 );

    if( p.sx < right.sx - 11 )
      draw_line
        ( svertex( p.sx, p.sy+3 ), svertex( p.sx, p.sy-3 ),
          color, screen );
  }

  for( long y = long( begin_y ) ; y <= long( end_y ) ; y++ )
  {
    svertex p = project( 0, y );

    if( p.sy > top.sy + 11 )
      draw_line
        ( svertex( p.sx-3, p.sy ), svertex( p.sx+3, p.sy ),
          color, screen );
  }
}
```

Projekt a2_7, dritter Schritt:

Bei der Verbindung der Funktionswerte benachbarter Pixel mit Linien wird dieselbe Grundidee verwendet, die wir bereits in den Programmen `a2_7` und `a2_8` kennen gelernt haben, im Zusammenhang mit der Darstellung einer Linie, deren Anfangspunkt identisch ist mit dem Endpunkt ihres Vorgängers:

```
void draw_function
(
  double (*f)( double x ), double start, double end,
```

```
  pixel_32 color, pixel_32 *screen
)
{
  svertex p = project( start, f( start ) );

  double step = fabs( end_x - begin_x ) / 640.0;
  for( double x = start + step ; x <= end ; x += step )
  {
    svertex q = project( x, f( x ) );

    draw_line( p, q, color, screen );

    p = q;
  }
}
```

Projekt a2_12, erster Schritt:

Der erste Schritt des Projektes `a2_12` ist nahezu identisch mit dem Programm `a2_3`. Die in Übungsaufgabe 2.1 vorgestellten Anweisungen werden in Form der Funktion `load_palette()` implementiert, `white_background()` weist dem Hintergrund des Bildschirms die Farbe Weiß zu.

Projekt a2_12, zweiter und dritter Schritt:

Um ein Rechteck zu definieren, kann eine Struktur `rectangle` definiert werden, welche die Koordinaten des oberen linken und unteren rechten Punktes enthält:

```
struct rectangle
{
  long xt, yt, xb, yb;

  rectangle( long x1, long y1, long x2, long y2 ) :
    xt( x1 ), yt( y1 ), xb( x2 ), yb( y2 ) { }
};
```

Anschließend wird eine Funktion `frame_point()` definiert, die das äußere Rechteck `a` und das innere Rechteck `i` entgegennimmt und den Variablen, auf die `*x` und `*y` verweisen, die Koordinaten eines zufälligen Punktes des Rahmens zuweist:

```
void frame_point
( rectangle a, rectangle i, long *x, long *y )
{
  do
  {
    *x = a.xt + (rand() % (a.xb - a.xt + 1));
    *y = a.yt + (rand() % (a.yb - a.yt + 1));
  } while
    (
      *x >= i.xt && *x <= i.xb &&
      *y >= i.yt && *y <= i.yb
    );
}
```

Hierzu werden in einer `do-while` Schleife so lange zufällige Koordinaten des äußeren Rechtecks erzeugt, bis man ein Koordinatenpaar findet, das nicht im kleineren Rechteck liegt. Die Arbeitsweise dieser Funktion orientiert sich somit strikt an den im Aufgabentext definierten Begriff des *Rahmens* von zwei Rechtecken.

Im letzten Schritt des Projektes wird diese Funktion schließlich zweimal aufgerufen, einmal für den äußeren, einmal für den inneren Rahmen; das innere Rechteck **I** wird mit der Methode des Programms `a2_11` mit zufälligen Punkten gefüllt.

Projekt a2_14, erster Schritt

Die im ersten Kapitel hergeleiteten Gleichungen beschreiben eine Rotation im mathematisch positiven bzw. entgegen dem Uhrzeigersinn. Die gesuchte Funktion muss somit die Rotation um den negativen Winkel durchführen. Aus der Definition am Einheitskreis ergeben sich für den Sinus und Kosinus eines beliebigen Winkels α folgende Beziehungen: $\mathbf{sin(}\ -\alpha\ \mathbf{)} = \mathbf{-sin(}\ \alpha\ \mathbf{)}$ und $\mathbf{cos(}\ -\alpha\ \mathbf{)} = \mathbf{cos(}\ \alpha\ \mathbf{)}$. Eingesetzt, folgt schließlich:

$$\begin{aligned} x' &= x * cos(-\alpha) - y * sin(-\alpha) &&= x * cos(\alpha) + y * sin(\alpha) \\ y' &= x * sin(-\alpha) + y * cos(-\alpha) &&= -x * sin(\alpha) + y * cos(\alpha) \end{aligned}$$

Diese Gleichungen beschreiben nach wie vor eine Drehung um den Ursprung. Die gesuchte Rotation um den lokalen Mittelpunkt **(0, r)** erhält man schließlich durch die Durchführung der folgenden zwei Schritte:

1. Rotation von **p** um den Winkel α
2. Verschiebung des sich ergebenden Punktes um **r** Einheiten nach oben durch die Addition der Konstanten +**r** zur y-Koordinate

Unser Punkt **p** besitzt nach Voraussetzung die Anfangskoordinaten **(0, 2*r) = (0, 1*r + r)**. Der Ausdruck **1*r** gibt hierbei den Abstand zwischen **p** und dem Mittelpunkt **(0, r)** der Kreisscheibe an. Weil die Drehung um diesen speziellen Punkt stattfindet, muss im ersten Schritt der Punkt **(0, 1*r)** um den Ursprung rotiert werden, damit **lr()** schließlich die Vorgabe aus Abbildung 2.30 im Buch erfüllen kann.

$$\begin{aligned} lr \;:\; & R \rightarrow R^2 \\ & \alpha \mapsto [\quad t*r*sin(\alpha), \quad t*r*cos(\alpha)+r \quad] \end{aligned}$$

Anstatt des konstanten Ausdruckes **1*r** wird der Abstand zwischen **p** und dem Rotationsmittelpunkt durch das etwas allgemeinere **tb*r** angegeben; hierbei darf die reelle Zahl **t** neben **1.0** auch andere Werte annehmen.

Projekt a2_14, zweiter Schritt

$$\begin{aligned} zg \;:\; & R \rightarrow R^2 \\ & \alpha \mapsto lr(\alpha)+\begin{pmatrix} \alpha*r \\ 0 \end{pmatrix} \quad = \quad [\, t*r*sin(\alpha)+\alpha*r, \quad t*r*cos(\alpha)+r \,] \end{aligned}$$

wobei **t** eine konstante reelle Zahl ist.

Projekt a2_14, dritter Schritt

Die zweite Definition von **lr()** erfolgt nach demselben Prinzip wie die erste: Der Punkt **p**, dessen Abstand zum Rotationszentrum **t*r** beträgt, wird um den Punkt mit den Koordinaten **(0, –s + r)** rotiert:

$$\begin{aligned} lr2 \;:\; & R \rightarrow R^2 \\ & \alpha \mapsto [\quad t*r*sin(\alpha), \quad t*r*cos(\alpha)+(-s+r) \quad] \end{aligned}$$

Projekt a2_14, vierter Schritt

```
svertex zk( double alpha )
{
  double s = 3;
  double r = 1;
  double t = 1.5;

  double beta = (alpha * r) / s;

  double sx = t*r * sin( alpha ) * cos( beta ) -
             (t*r * cos( alpha ) - s + r) * sin( beta );
```

```
  double sy = t*r * sin( alpha ) * sin( beta ) +
              (t*r * cos( alpha ) - s + r) * cos( beta );

  return svertex( sx, sy );
}
```

Einführung in die 3D-Programmierung

3.10 Besprechung der Übungsaufgaben

Übungsaufgabe 3.1:

1. M_n = { p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9 }, wobei:
   p_1=g, p_2=o, p_3=n, p_4=m, p_5=l, p_6=k, p_7=j, p_8=j, p_9=h
2. Eine beliebige Neuordnung der Punkte eines Polygons ist im Allgemeinen kein Polygon mehr. Da **M** ein Polygon angibt, darf man sich diesen wie den abgeschlossenen Kantenzug auf der linken Seite der Abbildung 3.3 im Buch vorstellen; der Mengenschnitt der ersten und der dritten Seite $\mathbf{S_1}$ und $\mathbf{S_3}$ der gegebenen Neuordnung $\mathbf{M_2}$ ist nicht leer, was der zweiten Bedingung der Polygondefinition widerspricht. Hierbei handelt es sich um den Fall, der auf der linken Seite der Abbildung 3.2 im Buch vorgestellt wird.

Übungsaufgabe 3.2:

Um das Aussehen der Linie **A** auf dem Bildschirm zu erhalten, beschreiben die Projektionsgleichungen zwei Linien, die vom Ursprung ausgehen und im Anfangs- bzw. Endpunkt von **A** enden. Die Linie **B** besitzt dieselbe Länge wie **A** – aufgrund der größeren Entfernung erscheint diese auf dem Bildschirm jedoch kürzer.

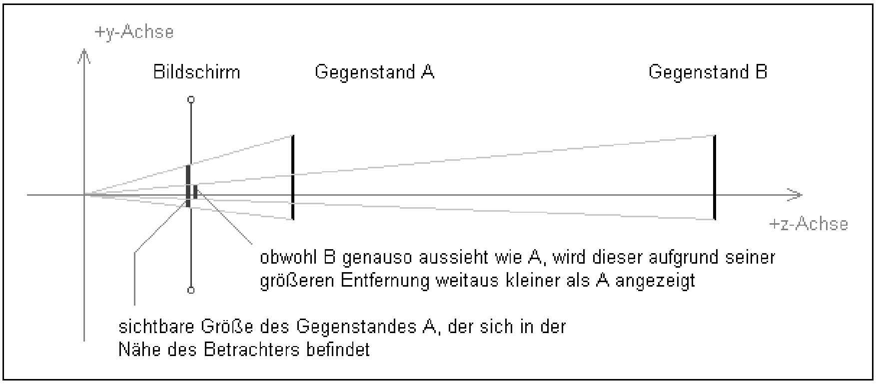


Abb. 3.1: Aufgrund ihrer unterschiedlichen Entfernung zum Betrachter besitzen die identischen Gegenstände **A** und **B** auf dem Bildschirm unterschiedliche Größen.

Übungsaufgabe 3.3:

Die vorgegebene Funktion **f()** ist linear, denn nach dem Distributivgesetz folgt:

$$f(\begin{pmatrix} x \\ y \\ z \end{pmatrix}) = (\frac{1}{2} - 3^5) * x + (-\sqrt{5} - 2) * y + 8 * z$$

Die Werte $\frac{1}{2}$ und 3^5 sind konstant, aus diesem Grund stellt auch das Ergebnis der Subtraktion $(\frac{1}{2} - 3^5)$ eine konstante reelle Zahl dar. Dasselbe gilt auch für die Konstante, mit der **y** multipliziert wird.

Übungsaufgabe 3.4:

1. Das gegebene Gleichungssystem lässt sich wie folgt in Form einer Funktion von $\mathbf{R}^3$ nach $\mathbf{R}^3$ ausdrücken:

$$ry \; : \; R^3 \; \rightarrow \; R^3$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} ry_1(x, y, z) \\ ry_2(x, y, z) \\ ry_3(x, y, z) \end{pmatrix} = \begin{pmatrix} cos(\alpha) * x + 0 * y + sin(\alpha) * z \\ 0 * x + 1 * y + 0 * z \\ - sin(\alpha) * x + 0 * y + cos(\alpha) * z \end{pmatrix}$$

$$wobei\; ry_i \; : \; R^3 \; \rightarrow \; R \qquad f\ddot{u}r \;\; i = 1, 2, 3$$

Anhand dieser Schreibweise lässt sich sofort die Linearität von **ry()** nach der ersten Definition erkennen; die Standardmethode zur Überprüfung der Linearität besteht jedoch darin, die Funktion auf die Erfüllung der beiden Bedingungen der zweiten abstrakteren Definition zu testen:

1. Additivität wird von ry() erfüllt, weil:

$$ry(\vec{p} + \vec{q}) \overset{(1)}{=} ry(\begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} u \\ v \\ w \end{pmatrix}) \overset{(2)}{=} ry(\begin{pmatrix} x+u \\ y+v \\ z+w \end{pmatrix})$$

$$\overset{(3)}{=} \begin{pmatrix} cos(\alpha) * (x+u) + sin(\alpha) * (z+w) \\ y+v \\ - sin(\alpha) * (x+u) + cos(\alpha) * (z+w) \end{pmatrix}$$

$$\overset{(4)}{=} \begin{pmatrix} cos(\alpha) * x + sin(\alpha) * z \\ y \\ - sin(\alpha) * x + cos(\alpha) * z \end{pmatrix} + \begin{pmatrix} cos(\alpha) * u + sin(\alpha) * w \\ v \\ - sin(\alpha) * u + cos(\alpha) * w \end{pmatrix}$$

$$\overset{(3)}{=} ry(\begin{pmatrix} x \\ y \\ z \end{pmatrix}) + ry(\begin{pmatrix} u \\ v \\ w \end{pmatrix}) \overset{(1)}{=} ry(\vec{p}) + ry(\vec{q})$$

Bei der Aufstellung dieser Gleichungskette werden folgende Regeln befolgt:

(1): Definition von $\vec{p}$ und $\vec{q}$

(2): Vektoraddition

(3): Definition der Funktion **ry()**

(4): Distributivgesetz reeller Zahlen

2. Homogenität wird von ry() erfüllt, weil:

$$
\begin{aligned}
ry(\lambda * \vec{p}) &\overset{(1)}{=} ry\left(\begin{pmatrix} \lambda * x \\ \lambda * y \\ \lambda * z \end{pmatrix}\right) \overset{(3)}{=} \begin{pmatrix} cos(\alpha) * (\lambda * x) + sin(\alpha) * (\lambda * z) \\ \lambda * y \\ -sin(\alpha) * (\lambda * x) + cos(\alpha) * (\lambda * z) \end{pmatrix} \\
&\overset{(5)}{=} \begin{pmatrix} \lambda * (cos(\alpha) * x) + \lambda * (sin(\alpha) * z) \\ \lambda * y \\ \lambda * (-sin(\alpha) * x) + \lambda * (cos(\alpha) * z) \end{pmatrix} \\
&\overset{(6)}{=} \lambda * \begin{pmatrix} cos(\alpha) * x + sin(\alpha) * z \\ y \\ -sin(\alpha) * x + cos(\alpha) * z \end{pmatrix} \overset{(3)}{=} \lambda * ry\left(\begin{pmatrix} x \\ y \\ z \end{pmatrix}\right) \\
&\overset{(1)}{=} \lambda * ry(\vec{p})
\end{aligned}
$$

Neben den Regeln, die bereits bei der Überprüfung der Additivität angewandt worden sind, kommen noch hinzu:

(5): Assoziativ- und Kommutativgesetz reeller Zahlen

(6): Skalarmultiplikation von Vektoren

2. Der Aufbau der Matrix $\mathbf{R_y}$ ergibt sich sofort aus der Definition von **ry()**:

$$
R_y = \begin{pmatrix} cos(\alpha) & 0 & sin(\alpha) \\ 0 & 1 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) \end{pmatrix}
$$

3. Die Rotation von **(3, 4, 5)** entspricht der Multiplikation von $\mathbf{R_y}$ mit dem Ortsvektor des Punktes:

$$
\begin{aligned}
R_y * \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix} &= \begin{pmatrix} cos(45°) & 0 & sin(45°) \\ 0 & 1 & 0 \\ -sin(45°) & 0 & cos(45°) \end{pmatrix} * \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix} \\
&= \begin{pmatrix} 3 * cos(45°) + 5 * sin(45°) \\ 4 \\ 3 * (-sin(45°)) + 5 * cos(45°) \end{pmatrix} \approx \begin{pmatrix} 5.7 \\ 4 \\ 1.4 \end{pmatrix}
\end{aligned}
$$

Zu beachten ist: implementiert man diese Funktion in einem Programm, müssen den Funktionen `sin()` und `cos()` Winkel in Bogenmaß angegeben werden.

4. Die Einheitsmatrix **E** ist eine andere Schreibweise für die Identitätsfunktion im Dreidimensionalen, wobei **e** : $\mathbf{R}^3 \rightarrow \mathbf{R}$ mit **e(x, y, z) = (x, y, z)**

$$E = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Übungsaufgabe 3.5:

$$f(\lambda * \vec{p}) \overset{(1)}{=} \begin{pmatrix} f_1(\lambda * \vec{p}) \\ f_2(\lambda * \vec{p}) \\ \vdots \\ f_m(\lambda * \vec{p}) \end{pmatrix} \overset{(2)}{=} \begin{pmatrix} \lambda * f_1(\vec{p}) \\ \lambda * f_2(\vec{p}) \\ \vdots \\ \lambda * f_m(\vec{p}) \end{pmatrix} \overset{(3)}{=} \lambda * \begin{pmatrix} f_1(\vec{q}) \\ f_2(\vec{q}) \\ \vdots \\ f_m(\vec{q}) \end{pmatrix}$$
$$\overset{(1)}{=} \lambda * f(\vec{p})$$

(1): Definition von Funktionen von $\mathbf{R}^n$ nach $\mathbf{R}^m$ unter Verwendung von Koordinatenfunktionen $\mathbf{f}_i : \mathbf{R}^n \rightarrow \mathbf{R}$ mit $1 \leq i \leq m$

(2): Homogenität der Koordinatenfunktionen $\mathbf{f}_i$**()**, zweiter Punkt der abstrakten Definition des Linearitätsbegriffes

(3): Addition von Vektoren

Übungsaufgabe 3.6:

$$G = \begin{pmatrix} sf & 0 & 0 & 0 \\ 0 & sf & 0 & 0 \\ 0 & 0 & sf & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[Gleichmäßige Skalierung]

Übungsaufgabe 3.7:

Hierzu müssen die Komponenten **xs**, **ys** und **zs** aus der Matrix **S** lediglich denselben Wert annehmen: **xs = xy = zs = sf**.

Übungsaufgabe 3.8:

Definiert man die Elementarmatrizen in der Reihenfolge ihres Austretens als $\mathbf{D}_x$, **S**, $\mathbf{D}_z$ und **T**, gilt für die Hauptmatrix **B**:

$$B = T * D_z * S * D_x = T * D_z * S * D_x * E$$

Die Multiplikation mit der Einheitsmatrix **E** ist ein Hinweis darauf, dass **B** mit den Komponenten von **E** initialisiert wird. Für $\mathbf{p}_i'$, der veränderten Version von $\mathbf{p}_i$, gilt somit:

$$\vec{p}_i{}' = B * \vec{p}_i \qquad \textit{für} \quad 1 \le i \le 8$$

Übungsaufgabe 3.9:

Um ein beliebiges Polygon `blue` gleichmäßig zu skalieren, ohne dass seine räumliche Position verändert wird, muss dieser unter Verwendung der folgenden Matrix bewegt werden:

```
matrix m;

m.translate( -blue.pos.wx, -blue.pos.wy, -blue.pos.wz );
m.scale( 3, 3, 3 );
m.translate( blue.pos.wx, blue.pos.wy, blue.pos.wz );
```

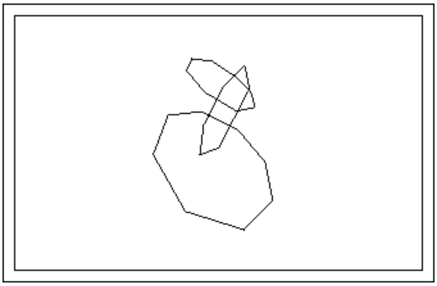


Abb. 3.2: Ausgabe des Programms *a3_12:* Die bei der gleichmäßigen Skalierung auftretende Verschiebung wird durch den Einsatz der Matrix **m** vermieden.

3.11 Besprechung der Projekte

Projekt a3_13, Erster Schritt:

Die einfachste Möglichkeit, die Koordinaten der Punkte eines regelmäßigen n-Ecks zu berechnen, besteht in der Rotation eines Punktes mit den Koordinaten **(10, 0, 0)** um die z-Achse:

Bemerkenswert ist, dass die Eintragung von neuen Rotationsinformationen in jeder Iteration der Schleife beabsichtigt ist und zum gewünschten Ergebnis führt.

```
void polygon::load( long pc, pixel_32 c )
{
  point_count = pc;  color = c;

  if( point_count >= largest_point_count )
    exit_error( "polygon::load(): point_count \
                 größer als largest_point_count.\n" );

  if( (wpoint = new vertex[ point_count ]) == NULL )
    exit_error( "polygon::load(): Fehler bei der \
                 Reservierung von Arbeitsspeicher.\n" );

  const double pi = 3.1415926535;
  matrix m;

  for( long x=0 ; x<point_count ; x++ )
  {
    vertex v = vertex( 10, 0, 0 );

    wpoint[ x ] = m * v;

    m.rotate_z( (2*pi) / point_count );
  }

  pos = vertex( 0, 0, 0 );
}
```

Die Funktion `matrix::clear()` darf in diesem Zusammenhang nicht aufgerufen werden.

Projekt a3_13, zweiter Schritt:

Die Definition der Klasse `thing` ist sehr einfach, und setzt im Wesentlichen die Funktionen der Klasse `poylgon` ein:

```
class thing
{
  private:
    polygon ps[ 3 ];
```

```
  public:
    vertex pos;

    void update_pos( matrix m );
    void display( pixel_32 *sbuffer );

    thing( long pc, pixel_32 color );
};

thing::thing( long pc, pixel_32 color )
{
  for( uchar x=0 ; x<3 ; x++ )
     ps[ x ].load( pc, color );

  const double pi = 3.1415926535;
  matrix tm;

  tm.rotate_x( 0.5 * pi );
  ps[ 1 ].update_pos( tm );
  tm.clear();

  tm.rotate_y( 0.5 * pi );
  ps[ 2 ].update_pos( tm );

  pos = vertex( 0, 0, 0 );
}

void thing::update_pos( matrix m )
{
  pos = m * pos;

  for( uchar x=0 ; x<3 ; x++ ) ps[ x ].update_pos( m );
}

void thing::display( pixel_32 *sbuffer )
{
```

```
  for( uchar x=0 ; x<3 ; x++ ) ps[ x ].display(sbuffer);

}
```

Aufgrund der großen Ähnlichkeit zwischen den beiden Klassen erfolgen Definition und Verschiebung eines Gegenstandes vom Typ `thing` an eine bestimmte Position unter Verwendung derselben Anweisungen wie im Fall der Klasse `polygon`:

```
thing cs( 32, pixel_32( 255, 255, 255 ) );

matrix m;
m.translate( 10, 0, 25 );
cs.update_pos( m );
```

Die Darstellung im Bildschirmspeicher erfolgt schließlich während der Ausführung der Programmschleife durch:

```
cs.display( sbuffer );
```

Projekt a3_13, dritter Schritt:

```
void thing::update_polygons( void )
{
  matrix m[ 3 ];

  uchar x;

  for( x=0 ; x<3 ; x++ )
     m[ x ].translate( -pos.wx, -pos.wy, -pos.wz );

  m[ 0 ].rotate_x( 0.01 );  m[ 0 ].rotate_y( 0.02 );
  m[ 0 ].rotate_z( 0.01 );

  m[ 1 ].rotate_x( 0.03 );  m[ 1 ].rotate_y( 0.01 );
  m[ 1 ].rotate_z( 0.01 );

  m[ 2 ].rotate_x( 0    );  m[ 2 ].rotate_y( 0.01 );
  m[ 2 ].rotate_z( 0.04 );
```

```
  for( x=0 ; x<3 ; x++ )
  {
    m[ x ].translate( pos.wx, pos.wy, pos.wz );

    ps[ x ].update_pos( m[ x ] );
  }
}
```

Diese Funktion, die eine Rotation der Polygone um die eigenen Achsen durchführt, ist in `thing` unter dem Zugriffsspezifizierer `private` deklariert; aufgerufen wird sie während der Ausführung von `thing::display()`:

```
void thing::display( pixel_32 *sbuffer )
{
  update_polygons();

  for( uchar x=0 ; x<3 ; x++ ) ps[ x ].display(sbuffer);
}
```

Projekt a3_13, vierter Schritt:

Aufgrund der großen Ähnlichkeit zwischen den Klassen `polygon` und `thing` erfolgt die Rotation des Gegenstandes `cs` im vierten Programm des Projekts durch den Aufruf derselben Funktionen wie bei der Drehung eines einzelnen Polygons. Die wichtigsten Anweisungen der Funktion `WinMain()`:

```
// Initialisierung und Verschiebung von cs

vertex p( 0, 0, 40 );

m.translate( -p.wx, -p.wy, -p.wz );
m.rotate_y( 0.01 );
m.translate( p.wx, p.wy, p.wz );

while( key_pressed() == 0 )
{
  long x;
  for( x=0 ; x<pixel_count ; x++ )
```

```
      sbuffer[ x ] = pixel_32( 0, 0, 0 );

  cs.update_pos( m );
  cs.display( sbuffer );

   for( x=0 ; x<pixel_count ; x++ )
      screen[ x ] = sbuffer[ x ];
}
```

Polyeder aus gefüllten Polygonen

4.9 Besprechung der Übungsaufgaben

Übungsaufgabe 4.1:

1. 1. Der Richtungsvektor $\vec{v}$ besitzt in Abbildung 4.3 die Komponenten β und δ.

   Daraus folgt: $\vec{n} = \begin{pmatrix} -\delta \\ \beta \end{pmatrix}$. Die Normalenform der Geraden wird im ersten Kapitel unter Verwendung des Standard-Skalarproduktes **s()** definiert. Setzt man diese beiden Vektoren in **s()** ein, ergibt sich:

   $$s(\vec{n}, \vec{v}) \quad = \quad \beta * (-\delta) + \delta * \beta \quad = \quad 0 \qquad\qquad \vec{n}, \vec{v} \in R^2$$

   Nach einer Folgerung aus dem verallgemeinerten Satz des Pythagoras gilt zusätzlich noch:

   $$s(\vec{n}, \vec{v}) \quad = \quad |\vec{n}| * |\vec{v}| * cos(\lambda)$$

   Aus diesen beiden Beziehungen folgt, dass der Kosinus des zwischen den beiden Vektoren eingeschlossenen Winkels λ den Wert 0 besitzt; damit stehen $\vec{n}$ und $\vec{v}$ senkrecht aufeinander.

2. Gegeben sei eine Gerade $\mathbf{G} \subset \mathbf{R}^2$ mit dem Fußpunkt **p** und dem Normalenvektor $\vec{n}$. Aus der Definition am Einheitskreis ergibt sich, dass der Kosinuswert von Winkeln zwischen 90° und 270° kleiner 0.0 ist. Aus der oben beschriebenen Folgerung des verallgemeinerten Satzes des Pythagoras folgt somit, dass ein Punkt $\mathbf{a} \in \mathbf{R}^2$ auf der **Rückseite** von **G** liegt, falls folgende Voraussetzung erfüllt ist: $s(\ a - p,\ \vec{n}\ ) < 0$.

   Der Betrag eines Vektors ist immer größer oder gleich 0.0, somit bestimmt das Vorzeichen des Kosinuswertes auch das Vorzeichen des Skalarproduktes. Dementsprechend gilt für ein Punkt **w**, der sich auf der **Vorderseite** von **G** befindet: $s(\ w - p,\ \vec{n}\ ) > 0$.

Übungsaufgabe 4.2:

Gegeben: Gerade $G = \{ a + \lambda * \vec{u} \mid \lambda \in R \}$, Gerade $H = \{ b + \eta * \vec{v} \mid \eta \in R \}$

mit: $a, b \in R^3$ und $a \neq b$ und $\lambda * \vec{u} + \eta * \vec{v} = \vec{0} \quad \Rightarrow \quad \lambda = \eta = 0$.

Gesucht: **dist(G, H)**

Die Grundidee der Lösung dieser Aufgabe ist die Definition einer Ebene **E** unter Verwendung des Fußpunktes **a** sowie der beiden Richtungsvektoren $\vec{u}$ und $\vec{v}$:

$$E \quad = \quad \{ \; a + \beta * \vec{u} + \delta * \vec{v} \quad | \quad \beta, \delta \in R \; \}$$

Diese Ebene ist eindeutig, denn die beiden Richtungsvektoren sind nach Voraussetzung nicht parallel. Es ist offensichtlich, dass die Gerade **G** vollständig innerhalb von **E** liegt; eine weitere Überlegung ergibt, dass die zweite Gerade **H** *parallel* zu **E** verläuft – die beiden Fußpunkte **a** und **b** sind ungleich, und der Vektor $\vec{v}$ ist bei der Definition beider Mengen beteiligt.

Die Ermittlung des gesuchten Abstandes zwischen den beiden Geraden lässt sich somit auf die bereits bekannte Berechnung der Entfernung (Gerade, Ebene) zurückführen:

$$dist(\,G, H\,) \quad = \quad dist(\,H, E\,)$$

wobei der hierzu benötigte Normalenvektor $\vec{n}$ aus dem Kreuzprodukt der beiden gegebenen Richtungsvektoren berechnet werden kann:

$$E \quad = \quad \{ \; \vec{q} \;\; | \;\; s(\,\vec{a} - \vec{q}, \vec{n}\,) = 0 \; \} \qquad\qquad wobei \quad \vec{n} \; = \; \vec{u} \; \times \; \vec{v}$$

Übungsaufgabe 4.3:

1. Eine mögliche Definition sieht wie folgt aus:

```
void backface_removal( uchar state )
{
  glFrontFace( GL_CW );

  if( state == 0 ) glDisable( GL_CULL_FACE );
  else glEnable( GL_CULL_FACE );
}
```

2. Das am Ende des dritten Kapitels vorgestellte Programm `a3_20`, das den animierten Umriss eines Polygons auf dem Bildschirm zeichnet, bildet die Vorlage zur Erstellung der gewünschten OPENGL-Anwendung. Hinzu kommen die bereits bekannten Klassen `polyeder` und `polygon` aus dem Programm `a4_5` – die Definition der letzteren muss geringfügig verändert werden, um die Kompatibilität mit OPENGL gewährleisten zu können. Der vollständige Quelltext des Programms `a4_7` befindet sich auf der CD im gleichnamigen Verzeichnis.

   Analog dazu bildet das bereits bekannte Programm `a3_21` die Vorlage von `a4_8`, das einen Dodekaeder mit Hilfe von DIRECTX auf dem Bildschirm zeichnet.

3. Die Grundidee der gesuchten Lösung ist einfach: Das gefüllte Polygon wird zunächst mit eingeschaltetem *Backface Removal* gezeichnet. Gleich danach wird das *Backface Removal* ausgeschaltet, und der Umriss des Vielecks wird in derselben Farbe dargestellt:

```
screen_interface.clear();

screen_interface.backface_removal( 1 );
octagon.display_polygon();

screen_interface.backface_removal( 0 );
octagon.display_shape();

screen_interface.swap_buffers();
```

Dadurch ist die gefüllte Oberfläche nur dann sichtbar, wenn die Vorderseite dem Betrachter zugewandt ist. Theoretisch ist der Umriss ununterbrochen sichtbar – aufgrund der Farbgleichheit nimmt man diesen jedoch nur wahr, wenn die Rasterung des Polygons durch den *Backface Removal* unterbunden wird.

Übungsaufgabe 4.4:

```
void draw_line
(
  svertex begin, svertex end,
  pixel_32 c, pixel_32 *screen
)
{
  long delta_x, delta_y, e, xstep, ystep, length;
  long offset = begin.sy * x_res + begin.sx;

  delta_x = end.sx - begin.sx;
  delta_y = end.sy - begin.sy;
  xstep = 1;  ystep = x_res;

  if( delta_x < 0 ) { delta_x=-delta_x;  xstep=-xstep; }
  if( delta_y < 0 ) { delta_y=-delta_y;  ystep=-ystep; }

  if( delta_y > delta_x )
```

```
  {
    long t = delta_x;  delta_x = delta_y;  delta_y = t;
    t = xstep;  xstep = ystep;  ystep = t;
  }

  length = delta_x+1;  e = 0;
  double act_z = begin.sz;
  double zstep = (end.sz - begin.sz) / length;

  while( length-- > 0 )
  {
    if( act_z < zbuffer[ offset ] )
    {
      screen[ offset ] = c;
      zbuffer[ offset ] = act_z;
    }

    offset += xstep;
    act_z += zstep;

    e += delta_y;
    if( e >= delta_x )
    {
      e -= delta_x;  offset += ystep;
    }
  }
}
```

Die Anweisungen vor der Schleife passen lediglich die Variablen `xstep` und `ystep` an die jeweilige Kategorie der Linie an; nach ihrer Ausführung enthält `length` die Anzahl der zu setzenden Pixel und damit auch die Länge der zurückzulegenden Strecke.

Bei der Berechnung der Schrittweite `z_step` wird die verallgemeinerte Interpolationsformel eingesetzt, und die Differenz der End- und Anfangsgröße durch die zurückzulegende Strecke `length` geteilt. Dieselbe Vorgehensweise haben wir auch bei der Darstellung von Polygonen mit dem Z-Buffer-Algorithmus eingesetzt.

4.10 Besprechung der Projekte

4.10.1 Projekt: Offsetinterpolation

Gegeben:

$$\begin{aligned} a\_offset &= ay * x\_res + ax \\ offset\_step &= \frac{(ey * x\_res + ex) \; - \; (ay * x\_res + ax)}{ey - ay} \end{aligned}$$

Zu zeigen:

$$(ay + n) * x\_res \; + \; (ax + n * x\_step) \;\; = \;\; a\_offset \; + \; n * offset\_step$$

Lösungsweg:

$$\begin{aligned} & (ay + n) * x\_res \; + \; (ax + n * x\_step) \\ = \; & ay * x\_res + n * x\_res + ax + n * x\_step \\ = \; & a\_offset \; + \; n * (x\_res + x\_step) \\ = \; & a\_offset \; + \; n * \left(\frac{ey - ay}{ey - ay} * x\_res + \frac{ex - ax}{ey - ay} \right) \\ = \; & a\_offset \; + \; n \; * \; \frac{ey * x\_res - ay * x\_res + ex - ax}{ey - ay} \\ = \; & a\_offset \; + \; n \; * \; \frac{(ey * x\_res + ex) \; - \; ay * x\_res - ay}{ey - ay} \\ = \; & a\_offset \; + \; n \; * \; offset\_step \end{aligned}$$

4.10.2 Projekt: Platonische Körper

Die Seitenlänge des vorgestellten Tetraeders besitzt den Wert $\sqrt{2} * t$. Damit diese die gesuchte Länge **t** annimmt, muss der gesamte Polyeder demnach mit der Konstante $\frac{1}{\sqrt{2}}$ **gleichmäßig skaliert** werden.

Für die Länge α, die bei der Konstruktion des Oktaeders in dem Quadrat mit den Eckpunkten **a**, **b**, **c**, **d** eingetragen ist, gilt: $\alpha = \sqrt{(0.5 * t)^2 + (0.5 * t)^2} = \frac{1}{\sqrt{2}} * t$. Aufgrund der hohen Symmetrie des Oktaeders kann man bereits anhand der Abbildung folgende Beziehung erkennen: $h = \alpha$. Diese Gleichheit erhält man aber auch mit dem Satz des Pythagoras, durch Umformung der Gleichung $\alpha^2 + h^2 = t^2$ nach **t**.

Die Definitionen der fünf platonischen Körper befinden sich in den gleichnamigen Dateien im Verzeichnis a4_6 auf der mitgelieferten CD.

4.10.3 Projekt: Drehung um einen beliebigen Punkt mit benutzerdefinierter Rotationsebene

Eine mögliche Lösung dieser Aufgabenstellung besteht darin, die Planeten unter Verwendung entsprechend initialisierter Variablen vom Typ polyhedron zu verwalten. Die Drehung erfolgt wie bisher während der Ausführung einer Schleife; da jeder Gegenstand ein unterschiedliches Rotationsverhalten aufweist, muss jeder Planet über eine eigene Matrix verfügen, welche die entsprechenden Bewegungsinformationen enthält.

Es ist vorteilhaft, diese zwei unterschiedlichen Informationen innerhalb eines eigenen Datentyps zu speichern. Die Klasse planet verfügt über eine Komponente vom Typ polyhedron, welche das Aussehen des jeweiligen Planeten beschreibt; bei dem zweiten Element handelt es sich um eine Matrix. Die Initialisierung der beiden Elemente erfolgt während der Ausführung einer Funktion namens load(), deren Prototyp folgendermaßen aufgebaut ist:

```
void planet::load( double dist, double angle,
                   double scale_factor, vertex sun )
```

Der Parameter sf gibt den Faktor an, um welchen der Planet skaliert werden muss. Auf diese Weise kann seine Größe benutzerdefiniert festgelegt werden. Bei d handelt es sich um die Entfernung zwischen dem Planet und der Sonne. Die letzten beiden Parameter sind für die Initialisierung der Matrix erforderlich: Der Winkel, um welchen der Planet in jedem Frame um die y-Achse der Sonne rotiert werden muss, wird in Form von ra angegeben. Die Rotationsgeschwindigkeit des Planeten lässt sich in Form dieses Winkels festlegen. Der Parameter sun gibt schließlich die Koordinaten der Sonne an.

Die Klasse planet verfügt schließlich über eine Funktion namens display(), welche thing::update_pos() und thing::display() aufruft, um den Planeten zu bewegen und auf dem Bildschirm darzustellen:

```
class planet
{
  private:
    polyhedron t;
    matrix m;

  public:
```

```
    void load( double dist, double angle,
               double scale_factor, vertex sun );

    void display( pixel_32 *sbuffer )
    {
      t.update_pos( m );
      t.display( sbuffer );
    }
};
```

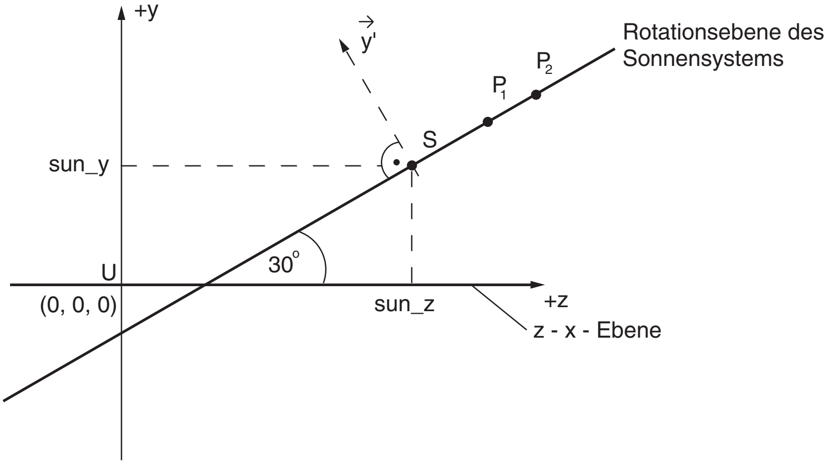
Die eigentliche Herausforderung besteht in der Definition der Funktion `planet::load()`. Die Aufgabe besteht darin, eine Rotation um einen beliebigen Punkt durchzuführen, dessen Koordinaten bekannt sind. Zusätzlich hierzu muss die Ebene, in welcher die Drehung um die y-Achse der Sonne durchgeführt werden muss, einen Winkel von **30** mit der x-z-Ebene bilden.

Das Problem ist nur, dass mithilfe unserer Rotationsgleichung lediglich Gegenstände rotiert werden können, deren Mittelpunkte Teil der xz-Ebene ist. Im letzten Kapitel wurde anhand der Rotation eines Polygons um die eigenen Achsen und um einen beliebigen Punkt ausführlich beschrieben, wie diese Einschränkung umgangen werden kann. Auch das aktuelle Problem lässt sich nach dem gleichen Vorbild lösen. Der Grundgedanke hierbei lautet: Wenn die durchzuführende Rotation um die y-Achse innerhalb einer beliebigen Ebene **E** stattfinden soll, müssen für die Dauer der Drehung folgende Voraussetzungen erfüllt sein:

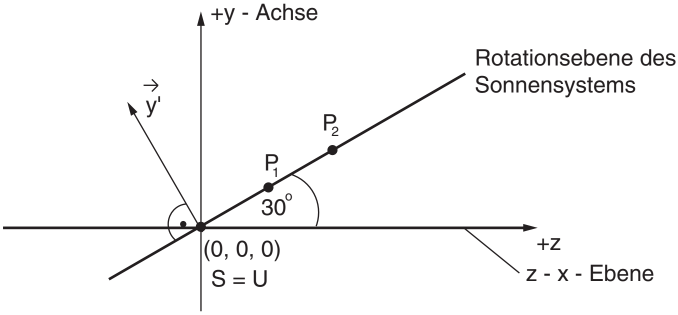
1. **E** muss mit der xz-Ebene deckungsgleich sein.
2. Das Rotationszentrum muss die Koordinaten des Ursprunges besitzen.

Hierfür ist die Durchführung einiger Schritte erforderlich. Wir gehen zunächst davon aus, dass die Sonne an einer beliebigen Position **S** innerhalb unserer Welt zu finden ist. Die einzelnen Planeten sind mit **P1**, **P2** usw. gekennzeichnet.

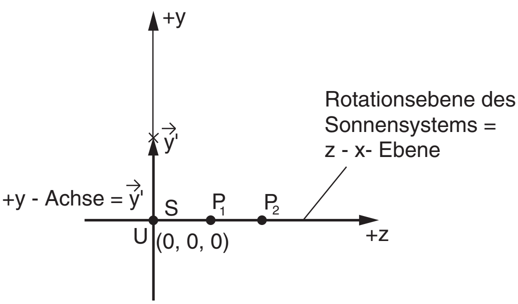
1. Wie bereits beschrieben, durchläuft die Ebene, in welcher sich die Planeten befinden, den Punkt **S** und bildet einen **30** -Winkel mit der x-z-Ebene. Die *lokale* y-Achse, um welche die Rotation erfolgt, ist mit **y'** gekennzeichnet:



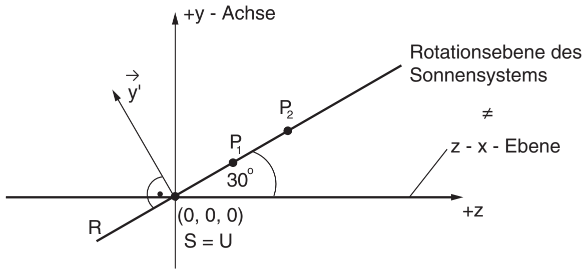
2. Für die Durchführung der Rotation um **y'** muss **S**, das Rotationszentrum, zunächst in Richtung des globalen Ursprunges **(0, 0, 0)** verschoben werden:



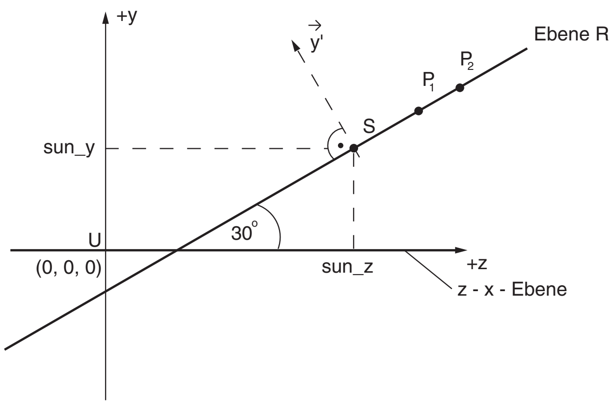
3. Anschließend ist eine Rotation der Ebene **E** um die x-Achse erforderlich, um die Deckungsgleichheit mit der x-z-Ebene zu erreichen. Um eine beliebige Ebene **E** drehen zu können, rotiert man sämtliche Punkte, welche sich in **E** befinden:



4. An diesem Zeitpunkt ist die lokale y-Achse **y'** mit der globalen Achse deckungsgleich; die Rotation kann somit wie gehabt durchgeführt werden. Nach der Rotation jedes Planeten unter Verwendung der entsprechenden Matrix muss das Sonnensystem seine ursprüngliche Position und Ausrichtung annehmen. Wie man leicht nachvollziehen kann, sind hierzu zunächst eine Rotation und anschließend eine Rotation erforderlich:



a) Rotation von R um die x - Achse



b) Translation von R, sodass S = (0, sun_y, sun_z)

Diese Rotationsform kann unter Verwendung einer konstanten Matrix durchgeführt werden. Ein **30** -Winkel entspricht einem Winkel von:

system_angle = $(0.5 * \pi) / 3$

bzw.

system_angle ≈ 0.5236

Einheiten Bogenmaß. Wenn der Rotationswinkel eines Planeten **angle** Radiant lautet, muss die dazugehörige Matrix folgendermaßen aufgebaut werden:

```
matrix m;

m.translate( -sun.wx, -sun.wy, -sun.wz );
m.rotate_x( system_angle );
m.rotate_y( angle );
m.rotate_x( -system_angle );
m.translate( sun.wx, sun.wy, sun.wz );
```

Hierbei handelt es sich bei `sun` um eine Variable vom Typ `vertex`, welche die Koordinaten des Mittelpunkts der Sonne enthält. Bevor die Planeten jedoch mithilfe einer auf dieser Art aufgebauten Matrix rotiert werden können, müssen diese zunächst auf ihre Grundposition, welche am Anfang des 1. Schritts dargestellt ist, gebracht werden. Zunächst gehen wir davon aus dass die Sonne sich im globalen Ursprung befindet. Nach der Initialisierung einer Variablen vom Typ `vertex` unter Verwendung der Datei `"sphere.tg1"`:

```
thing t;

t.load( "sphere.tg1" );

matrix m;
m.scale( scale_factor, scale_factor, scale_factor );
```

befindet sich der Mittelpunkt `vertex::wpos` an der Position (0, 0, 0). In der grafischen Darstellung des 3. Schritts liegt zwischen jedem Planeten und der Sonne eine gewissen Entfernung vor; um diese Vorgabe einzuhalten, muss jeder Planet um `planet::dist` Einheiten vom globalen Ursprung versetzt werden:

```
m.translate( 0, 0, dist );
```

Im 4. Schritt wird die Ebene des Sonnensystems um eine Winkel von –`system_angle` Radiant um die x-Achse rotiert, und anschließend verschoben. Durch diese Vorgehensweise wird das System in seine Ausgangslage gebracht, welche die Grundlage für die anschließend erfolgende Rotation darstellt:

```
m.rotate_x( -system_angle );
m.translate( sun.wx, sun.wy, sun.wz );
```

Die vollständige Definition der Funktion `planet::load()` kann somit wie folgt aufgebaut werden:

```
void planet::load( double dist, double angle,
                   double scale_factor, vertex sun )
{
  t.load( "sphere.tg1" );

  m.scale( scale_factor, scale_factor, scale_factor );
  m.translate( 0, 0, dist );
  m.rotate_x( -system_angle );
  m.translate( sun.wx, sun.wy, sun.wz );
  t.update_pos( m );
  m.clear();

  m.translate( -sun.wx, -sun.wy, -sun.wz );
  m.rotate_x( system_angle );
  m.rotate_y( angle );
  m.rotate_x( -system_angle );
  m.translate( sun.wx, sun.wy, sun.wz );
}
```

Der vollständige Quelltext des Programms `4_16` befindet sich auf der CD im gleichnamigen Verzeichnis.

Unterstützung von Eingabegeräten

5.3 Besprechung der Übungsaufgaben

Übungsaufgabe 5.1:

1. Das Problem der flachen und tiefen Objektkopien tritt nur bei Klassen mit Arrays auf, deren Speicherplatz **dynamisch**, das heißt mit Hilfe des Operators `new` reserviert wird. Das Array der Klasse `matrix` ist hingegen **statisch**:

   ```
   double mx[ 4 ][ 4 ];
   ```

   Der Speicherplatz eines statischen Arrays wird durch die Angabe einer konstanten Anzahl an Elementen reserviert. In diesem Fall ist die Definition des Arrays `mx[][]` identisch mit der Definition von sechzehn **gewöhnlichen** Variablen desselben Datentyps:

   ```
   double a0, a1, a2, /* ... */ a14, a15;
   ```

   Übergibt man eine Variable vom Typ `matrix` einer Funktion *by value*, wird intern eine Kopie generiert mit sechzehn neuen Array-Elementen, die automatisch mit dem richtigen Inhalt initialisiert werden. Zur Freigabe des Speicherplatzes gewöhnlicher Variablen wird kein Destruktor benötigt; das statische Array `mx[][]` der Originalmatrix bleibt unbeeinträchtigt, wenn am Ende der Funktion der Speicherplatz der sechzehn neuen Variablen wieder freigegeben wird.

   Definiert man das Array `matrix::mx[][]`hingegen dynamisch,

   ```
   double **mx;

   mx = new (double *)[ 4 ];
   for( char i=0 ; i<4 ; i++ ) mx[ i ] = new double[ 4 ];
   ```

   tritt das Problem der flachen Objektkopien auf.

2. Um diese Übungsaufgabe zu lösen, muss `handle_input()` lediglich um folgende Anweisungen erweitert werden:

```
if( input.key_pressed( 'H' ) ) m.translate(-1, 0, 0 );
if( input.key_pressed( 'K' ) ) m.translate( 1, 0, 0 );
if( input.key_pressed( 'Z' ) ) m.translate( 0,-1, 0 );
if( input.key_pressed( 'I' ) ) m.translate( 0, 1, 0 );
if( input.key_pressed( 'J' ) ) m.translate( 0, 0,-1 );
if( input.key_pressed( 'U' ) ) m.translate( 0, 0, 1 );
```

Übungsaufgabe 5.2:

Eine mögliche Lösung dieser Aufgabe sieht wie folgt aus:

```
void draw_control_points( long pc, svertex *points )
{
  glPointSize( 9 );

  glBegin( GL_POINTS );

    for( long x=0 ; x<pc ; x++ )
    {
      if( moving_point == x ) continue;

      glColor3ub( 0, 0, 200 );

      if
      (
        moving_point == -1 &&

        dist( points[ x ], input.mouse_act_pos ) <=
        max_dist
      )
      glColor3ub( 255, 255, 255 );

      glVertex2d( points[ x ].sx, points[ x ].sy );
    }

    if( moving_point >= 0 )
    {
```

```
      glColor3ub( 255, 255, 255 );
      glVertex2d
      (
        points[ moving_point ].sx,
        points[ moving_point ].sy
      );
    }

  glEnd();
}
```

Einfache Polygonschattierung

6.12 Besprechung der Übungsaufgaben

Übungsaufgabe 6.1:

Die Erstellung des gesuchten Programms sollte kein Problem darstellen, sie sieht lediglich den Einbau der neuen Farbgebung in die gegebene Definition von a6_1 vor. Der vollständige Quelltext von a6_4 befindet sich auf der CD im gleichnamigen Verzeichnis in dreimaliger Ausführung.

Das Polyeder, das aus der Vereinigung von fünf Tetraedern entsteht, wird von dem Programm a6_4_2 dargestellt. Hierbei handelt es sich eindeutig um einen konkaven Polyeder, mit vielen sich schneidenden Polygonen. Bei sehr genauem Hinsehen fällt während der Rotation des Polyeders eine geringfügige Ungenauigkeit auf: Die von zwei sich schneidenden Polygonen gebildete Linie ist zwar sehr scharf, scheint sich jedoch während der Drehung zu verschieben.

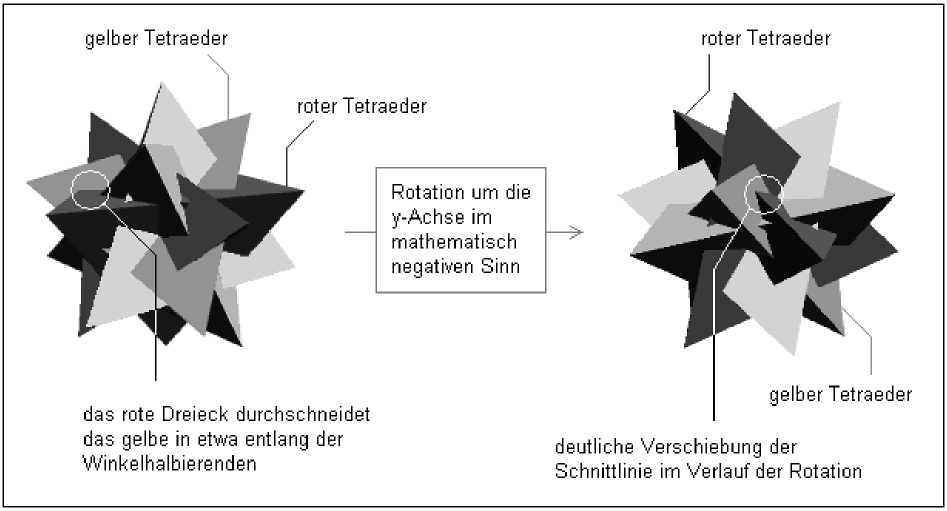


Abb. 6.1: Die geringfügige Ungenauigkeit des linearen Z-Speicher-Algorithmus fällt nur bei genauem Hinsehen auf

Auf der linken Seite der Abbildung 6.1 im Buch wird das hellere Dreieck vom dunkleren direkt in der Mitte durchgeschnitten; diese Schnittlinie wird nach einer Rotation um die y-Achse eindeutig verschoben.

Der Grund für dieses Phänomen ist die im vierten Kapitel beschriebene Ungenauigkeit des linearen Z-Speicher-Algorithmus – die lineare Interpolation unveränderter z-Koordinaten ist mit einem Fehler behaftet. Es ist dennoch bemerkenswert, dass die trotz dieser eindeutigen Ungenauigkeit gezeugten Grafiken diesen hohen Grad an Klarheit und perspektivischer Korrektheit erreichen.

Das beschriebene Problem tritt bei der Verwendung des im 8. Kapitel beschriebenen, perspektivischen Z-Speicher-Algorithmus nicht mehr auf.

Übungsaufgaben 6.2 und 6.3:

Die vollständigen Quelltexte der Programme `a6_8`, `a6_13` und `a6_14` befinden sich auf der CD in den gleichnamigen Verzeichnisen.

6.13 Besprechung der Projekte

Projekt a6_7, erster Schritt:

Eine mögliche Implementierung des gesuchten Programms ist die Erweiterung der Klasse `polyhedron` um eine neue Komponente `polyhedron::detail_level`. Diese wird mit dem Wert `(dl+1)` initialisiert – wenn die quadratische Punktwolke die Seitenlänge `dl` haben soll, dann muss jede Seite `(dl+1)` Punkte enthalten:

```
void polyhedron::initialise_vertices( long dl )
{
 detail_level = dl+1;
  vertex_count = detail_level * detail_level;

  if( (vs = new vertex[ vertex_count ]) == NULL )
exit_error( "*vs: Fehler bei der Reservierung von \
                    Arbeitsspeicher.\n" );

  for( long x=0 ; x<vertex_count ; x++ )
     vs[ x ] = vertex( 0, 0, 0 );

  pos = vertex( 0, 0, 0 );
}
```

```
void polyhedron::plane( long dl )
{
 initialise_vertices( dl );

  long x, y;

  for( y=0 ; y<detail_level ; y++ )
     for( x=0 ; x<detail_level ; x++ )
     {
   vs[ y * detail_level + x ].wx =  x;
       vs[ y * detail_level + x ].wy =  0;
       vs[ y * detail_level + x ].wz = -y;
     }

  matrix m;
  m.translate
  ( -0.5*(detail_level-1), 0, 0.5*(detail_level-1) );
 for( x=0 ; x<vertex_count ; x++ ) vs[x] = m * vs[x];
}
```

Nach der Ausführung der beiden Schleifen in `polyhedrno::plane()` befindet sich der erste Vertex `vs[ 0 ]` des Polyeders im Ursprung der Welt, während der Mittelpunkt der Punktwolke die Koordinaten `( 0.5*dl, 0, 0.5*dl )` besitzt; dieser Mittelpunkt wird durch die Anwendung der Matrix `m` am Ende der Funktion auf alle Vertices des Polyeders in Richtung des globalen Ursprunges verschoben.

Projekt a6_7, zweiter Schritt:

Die gesuchte Definition von `polygon::load()` besitzt große Ähnlichkeit mit der bereits bekannten Version der Ladefunktion:

```
void polygon::load
( long a, long b, long c, long d, vertex *vs )
{
 point_count = 4;
  if( (points = new long[ point_count ]) == NULL )
    exit_error( "*points: Fehler bei der Reservierung \
                  von Arbeitsspeicher.\n" );
```

```
 points[ 0 ] = a;  points[ 1 ] = b;
  points[ 2 ] = c;  points[ 3 ] = d;

  color_offset = rand() % 2;

  vector r = vs[ points[ 1 ] ] - vs[ points[ 0 ] ];
  vector s = vs[ points[ 3 ] ] - vs[ points[ 0 ] ];
  normal = cross( r, s );
}
```

Diese Funktion wird von der erweiterten Version der Funktion `polyhedron::intialise_polygons()` aufgerufen:

```
void polyhedron::initialise_polygons( void )
{
  polygon_count = (detail_level-1)*(detail_level-1);

  if( (ps = new polygon[ polygon_count ]) == NULL )
    exit_error( "*ps: Fehler bei der Reservierung von \
                 Arbeitsspeicher.\n" );

  long z=0;
  for( long y=0 ; y<detail_level-1 ; y++ )
     for( long x=0 ; x<detail_level-1 ; x++ )
     {
       long a = y * detail_level + x;

   ps[ z++ ].load
      ( a, a+1, a+detail_level+1, a+detail_level, vs );
   }
}

void polyhedron::plane( long dl )
{
  initialise_vertices( dl );

  long x, y;
```

```
  for( y=0 ; y<detail_level ; y++ )
     for( x=0 ; x<detail_level ; x++ )
     {
       vs[ y * detail_level + x ].wx =  x;
       vs[ y * detail_level + x ].wy =  0;
       vs[ y * detail_level + x ].wz = -y;
     }

  matrix m;
  m.translate
  ( -0.5*(detail_level-1), 0, 0.5*(detail_level-1) );
  for( x=0 ; x<vertex_count ; x++ ) vs[x] = m * vs[x];

 initialise_polygons();
}
```

Projekt a6_7, dritter Schritt:

Die gesuchte Funktion `polyhedron::sphere()` kann wie folgt definiert werden:

```
void polyhedron::sphere( long dl )
{
  initialise_vertices( dl );

 const double pi = 3.1415926535;
  matrix m;
  m.rotate_z( pi/(detail_level-1) );

  vs[ 0 ] = vertex( 0, 1, 0 );
  for( long y=1 ; y<detail_level ; y++ )
  {
    vs[ y*detail_level ] = vs[ (y-1)*detail_level ];

    vs[ y*detail_level ] = m * vs[ y*detail_level ];
  }

  rotation_symmetry();
```

```
  initialise_polygons();
}
```

Die hier angegebene Definition enthält eine geringfügige Optimierung im Vergleich zu der im ersten Punkt beschriebenen Vorgehensweise: Nur die Vertices der *ersten Spalte* werden in regelmäßigen Abständen auf der Oberfläche eines Halbkreises mit Radius 1.0 angeordnet. Diese Initialisierung erfolgt rekursiv, das heißt, dass der aktuelle Vertex innerhalb der Spalte eine um den konstanten Winkel `pi/(detail_level-1)` rotierte Version seines Vorgängers darstellt. Wie in der Abbildung vorgestellt, erfolgt diese Rotation um die z-Achse im mathematisch positiven Sinn.

Die im zweiten Punkt beschriebene Rotation um die y-Achse erfolgt nach demselben Prinzip während der Ausführung von `rotation_symmetry()`:

```
void polyhedron::rotation_symmetry( void )
{
  const double pi = 3.1415926535;
  matrix m;
  m.rotate_y( (2*pi)/(detail_level-1) );

  for( long x=1 ; x<detail_level ; x++ )
     for( long y=0 ; y<detail_level ; y++ )
     {
       vs[y*detail_level+x] = vs[ y*detail_level+x-1 ];

       vs[y*detail_level+x] = m * vs[y*detail_level+x];
     }
}
```

Projekt a6_7, vierter Schritt:

```
void polyhedron::cylinder( long dl, double radius )
{
  initialise_vertices( dl );

 for( long y=0 ; y<detail_level ; y++ )
    vs[ y*detail_level ] = vertex( -radius, -y, 0 );
```

```
  rotation_symmetry();

  matrix m;
  m.translate( 0, detail_level*0.5, 0 );
  for( long x=0 ; x<vertex_count ; x++ )
     vs[ x ] = m * vs[ x ];

  initialise_polygons();
}

void polyhedron :: cone
( long dl, double radius, double height )
{
  initialise_vertices( dl );

 for( long y=0 ; y<detail_level ; y++ )
  {
    vector dir = vertex( -radius, -height*0.5, 0 ) -
               vertex( 0, height*0.5, 0 );
    dir.set_length( double(y) / double(detail_level) );

    vs[y*detail_level] = vertex(0,height*0.5,0) + dir;
  }

  rotation_symmetry();

  initialise_polygons();
}

void polyhedron::torus( long dl, double radius )
{
  initialise_vertices( dl );

 const double pi = 3.1415926535;
  matrix m;
```

```
  m.rotate_z( (2*pi)/(detail_level-1) );

  long y;

  vs[ 0 ] = vertex( -1, 0, 0 );
  for( y=1 ; y<detail_level ; y++ )
  {
    vs[ y*detail_level ] = vs[ (y-1)*detail_level ];

    vs[ y*detail_level ] = m * vs[ y*detail_level ];
  }

  m.clear();
  m.translate( -radius, 0, 0 );
  for( y=0 ; y<detail_level ; y++ )
     vs[ y*detail_level ] = m * vs[ y*detail_level ];

 rotation_symmetry();

  initialise_polygons();
}
```

Bei der Konstruktion des Kegels werden folgende überladene Operatoren eingesetzt:

```
vector operator - ( vertex v1, vertex v2 )
{
  return vector
  ( (v1.wx - v2.wx), (v1.wy - v2.wy), (v1.wz - v2.wz) );
}

vector operator * ( double t, vector a )
{
  return vector( t*a.x, t*a.y, t*a.z );
}

vertex operator + ( vertex p, vector dir )
{
```

```
  return vertex
  ( (p.wx + dir.x), (p.wy + dir.y), (p.wz + dir.z) );
}
```

Projekt a6_7, fünfter Schritt:

Die vollständigen Quelltexte der Programme a6_7_4 und a6_7_5 befinden sich auf der CD in den gleichnamigen Verzeichnissen.

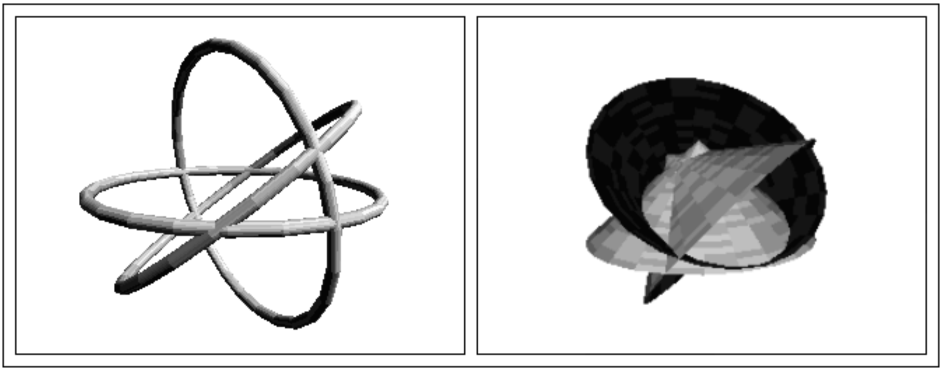


Abb. 6.2: Ausgaben der Programme *a6_7_4* und *a6_7_5*

Projekt a6_9, erster Schritt:

```
void floating_polyhedron :: load
( polyhedron *begin, polyhedron *end )
{
 primary.load( begin );

  {
    if( begin->vertex_count != end->vertex_count )
      exit_error("Anfangs- und Endpolyeder besitzen \
        eine unterschiedliche Anzahl an Eckpunkten.\n");

    if
    ( (ds = new vector[primary.vertex_count]) == NULL )
    exit_error( "*ds: Fehler bei der Reservierung von \
                 Arbeitsspeicher.\n" );

    step_count = 1000;
```

```
    for( long x=0 ; x<primary.vertex_count ; x++ )
   ds[ x ] = (1.0 / step_count) *
              (end->vs[ x ] - begin->vs[ x ]);
  }
}

void floating_polyhedron::update_pos( matrix m )
{
  pos = m * pos;

  primary.update_pos( m );

 for( long x=0 ; x<primary.vertex_count ; x++ )
    ds[ x ] = m * ds[ x ];
}

void floating_polyhedron::display( void )
{
 if( act_step < step_count )
  {
   for( long x=0 ; x<primary.vertex_count ; x++ )
      primary.vs[ x ] = primary.vs[ x ] + ds[ x ];

   act_step++;
  }

  primary.display_edges();
}
```

Durch die Veränderung der Elemente des Arrays `ds[]` während der Ausführung von `update_pos()` wird gewährleistet, dass die Umwandlung auch während einer Bewegung des Polyeders problemlos durchgeführt wird.

Würde der Aufruf von `polyhedron::display_polygons()` am Ende der Funktion `display()` erfolgen, würde bei der gegebenen Definition der Klasse `polyhedron` eine fehlerhafte Ausgabe angezeigt werden. Damit die Schattierung der

Polygone wie beabsichtigt erfolgen kann, müssen nach jeder Veränderung des Arrays `primary.vs[]` zusätzlich noch die Normalenvektoren der Polygone `primary.ps[]` neu berechnet werden. Die Darstellung des Drahtgittermodells erfolgt mit Hilfe der folgenden Funktion:

```
void polyhedron::display_edges( void )
{
  glColor3d( 1, 1, 1 );

  long x, y;

  glBegin( GL_LINES );
    for( y=0 ; y<detail_level ; y++ )
       for( x=y*detail_level ;
            x<(y+1)*detail_level ;
            x++ )
       {
         if( x < (y+1)*detail_level-1 )
         {
           glVertex3d( vs[x].wx, vs[x].wy, vs[x].wz );
           glVertex3d
           ( vs[ x+1 ].wx, vs[ x+1 ].wy, vs[ x+1 ].wz );
         }

         if( y < detail_level-1 )
         {
           glVertex3d( vs[x].wx, vs[x].wy, vs[x].wz );
           glVertex3d( vs[ x+detail_level ].wx,
                       vs[ x+detail_level ].wy,
                       vs[ x+detail_level ].wz );
         }
       }
  glEnd();
}
```

Projekt a6_9, zweiter Schritt:

Der vollständige Quelltext des Programms `a6_9_2` befindet sich auf der CD im gleichnamigen Verzeichnis.

Bitmaps

7.12 Besprechung der Übungsaufgaben

Übungsaufgabe 7.1:

Im ersten Schritt der Definition der gesuchten Funktion wird ein neues, lokales Array definiert, das die neue Version der Bitmap enthalten soll:

```
pixel_32 *new_picture = new pixel_32[ new_xs * new_ys ];
```

Anschließend werden die Elemente dieses Arrays nach dem bekannten Prinzip mit Hilfe von zwei Schleifen initialisiert:

```
for( long y=0 ; y<new_ys ; y++ )
   for( long x=0 ; x<new_xs ; x++ )
   {
     new_picture[ y * new_xs + x ] =
       picture[ get_yt( y ) * xscale + get_xt( x ) ];
   }
```

Der letzte Schritt ist die Definition der Funktionen `get_xt()` und `get_yt()`, mit deren Hilfe die Koordinaten des gesuchten Pixels innerhalb der ursprünglichen `bitmap_32::picture[]` in Abhängigkeit von der aktuellen Position `(x, y)` innerhalb der neuen, skalierten Bitmap berechnet werden.

Diese Skalierung soll *gleichmäßig* sein: Bewegt man sich um `new_xs` Pixel innerhalb der neuen Bitmap, soll man sich gleichzeitig um `xscale` Pixel innerhalb der ursprünglichen bewegen. Gesucht ist die Länge der Strecke `u`, die man in `bitmap::picture[]` zurücklegt, wenn man sich wie in der inneren `for()`-Schleife um **einen** Pixel in +x-Richtung in `new_picture[]` bewegt. Hierbei gilt, dass je größer `xscale` ist, desto größer muss auch `u` sein. Es handelt sich demnach um eine **proportionale Zuordnung**, aus der folgt:

```
u = double( xscale ) / double( new_xs )
v = double( yscale ) / double( new_ys )
```

Die Variable v gibt die entsprechende Länge in vertikaler Richtung. Die vollständige Definition der gesuchten Funktion ist somit:

```
void bitmap_32::resize( long new_xs, long new_ys )
{
  pixel_32 *new_picture;

  if((new_picture=new pixel_32[new_xs*new_ys]) == NULL)
    exit_error("*new_picture: Fehler bei der \
                 Reservierung von Arbeitsspeicher.\n");

  double u = double( xscale ) / double( new_xs );
  double v = double( yscale ) / double( new_ys );

  for( long y=0 ; y<new_ys ; y++ )
     for( long x=0 ; x<new_xs ; x++ )
     {
       long tx = long( x * u );
       long ty = long( y * v );

       new_picture[ y * new_xs + x ] =
         picture[ ty * xscale + tx ];
     }

  xscale = new_xs;  yscale = new_ys;
  delete [] picture;
  picture = new_picture;
}
```

Übungsaufgabe 7.2:

1. Die einfachste Möglichkeit, die Flammenhöhe zu regulieren besteht in der Veränderung der Konstanten `component_count`, welche die Anzahl der Farben der eingesetzten Palette angibt. Verdoppelt man diese Anzahl von 256 auf 512, werden die Flammen doppelt so hoch, denn es dauert doppelt so lange, bis die bekannte Anweisung

   ```
   if( sbuffer[ x-x_res ] > 0 ) sbuffer[ x-x_res ]--;
   ```

   die aktuelle Farbe auf 0, die Position der Hintergrundfarbe, heruntergezählt hat.

2. Der vollständige Quelltext befindet sich auf der CD im Verzeichnis `a7_8`.
3. Um diese Aufgabe zu lösen, muss lediglich die Klasse `pixel_8` geringfügig verändert werden. Der zweite Schritt der Konstruktion des Feuereffektes kann in einer Funktion `bitmap_8::update()` kopiert werden, die in jedem Frame aufgerufen werden muss. Die vollständige Lösung befindet sich in demselben Verzeichnis.

Übungsaufgabe 7.3:

Die gesuchte Geschwindigkeitsoptimierung sieht vor, die `yf( y )` am Anfang jedes Frames in einem Array `y_values[]` einzutragen. Während der Ausführung der inneren `for()`-Schleife lassen sich die gesuchten Werte einfach aus diesem Array herauslesen. Diese Optimierung bewirkt eine sehr starke Erhöhung der Ausführungsgeschwindigkeit des Programms.

```
const long xscale = 600, yscale = 400;
double y_values[ y_scale ];

while( 1 )
{
  if( input.check() == 1 ) break;
  if( input.event_key != 0 ) break;

  for( long y=0 ; z<yscale ; y++ )
     y_values[ y ] = yf( y );

  for( long x=0 ; x<xscale ; x++ )
  {
    double x_value = xf( x );

    long offset = x;
    for( long y=0 ; y<yscale ; y++ )
    {
      long color_offset =
        uchar( x_value + y_values[ y ] );

      screen[ offset ] =
        palette.components[ color_offset ];

      offset += x_res;
    }
  }
}
```

Übungsaufgabe 7.4:

1. Bei der Definition von `xh()` und `yh()` werden zum ersten Mal Sinusfunktionen miteinander multipliziert. Das Ergebnis sind Verzerrungen des Plasmaeffektes, die sich mit den anderen Operationen nicht erzielen lassen:

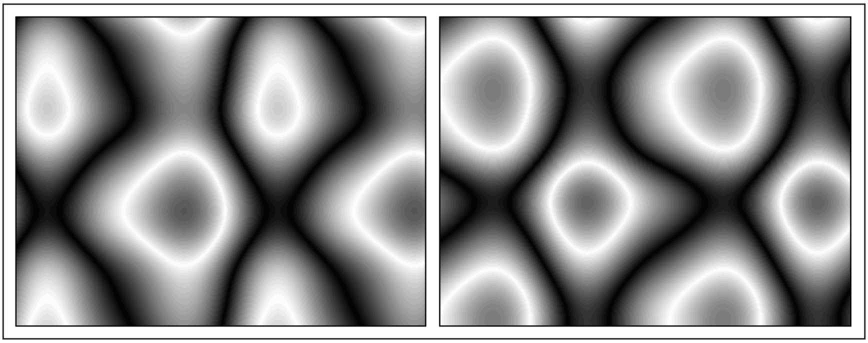


Abb. 7.1: Animiertes Plasmaeffekt mit Verzerrungen, die durch Multiplikation von Sinusfunktionen hervorgerufen werden

2. Die dritte Operation, die Multiplikation des Parameters mit einer Konstanten, bewirkt eine Stauchung des Funktionsgraphen. Wie wir wissen, werden die Laufvariablen `x` und `y` bereits mit der Konstanten `t` multipliziert, um die Gradangabe in Bogenmaß umzurechnen. Vergrößert man den Wert von `t`, erhöht sich die Anzahl der Extrempunkte der Funktionen pro Längeneinheit auf der x-Achse, wodurch die Anzahl der sichtbaren Zellen vergrößert wird:

```
double t = 2 * 3.1415926535 / 180.0;
```

Übungsaufgabe 7.5:

Wenn `grad` einen größeren Wert besitzt, kann es durchaus vorkommen, dass der Farbwert `act_h` gleich um ein Vielfaches höher als der Maximalwert 255 wird; genauer ausgedrückt, gibt es in diesem Fall zwei natürliche Zahlen **u** und **t**, für die gilt:

```
act_h == u * 255 + t
```

Ein Farbwert kann nicht höher werden als 255; die Strecke `t`, die hinzuaddiert werden müsste, kann demnach nur noch von 255 subtrahiert werden; die vollständige Anweisung sieht demnach wie folgt aus:

```
if( act_h > 255 )
  {  act_h = act_h % 256;  act_h = 255 - act_h;  }
```

Negative Höhen werden nach demselben Prinzip behandelt, das heißt der Minimalhöhe 1 hinzuaddiert:

```
if( act_h < 1 )
  {  act_h = act_h % 256;  act_h = -act_h + 1;  }
```

Übungsaufgabe 7.6:

Die Grundidee der gesuchten Optimierung besteht in der Definition einer Variablen `offset`, die mit der Position des obersten Punktes der Linse initialisiert wird. Dieser Punkt mit den Bildschirmkoordinaten `(lens_pos.sy - lens_radius, lens_pos.sx)` ist auch in Abbildung 7.44 im Buch dargestellt. Da `x` den horizontalen Abstand zwischen dem aktuellen Pixel und dem Mittelpunkt der Linse angibt, lässt sich die entsprechende Position im Videospeicher durch `sbuffer[ offset+x ]` angeben:

```
void draw_lens( pixel_32 *sbuffer )
{
  short r = lens_radius;

  long offset = (lens_pos.sy - r) * x_res + lens_pos.sx;

  for( short y = -r ; y <= r ; y++ )
  {
    short b = short( sqrt( r*r - y*y ) );

    for( short x = -b ; x <= b ; x++ )
       if( sqrt( x*x + y*y ) > inner_radius )
         sbuffer[ offset + x ] = pixel_32(100,100,100);

    offset += x_res;
  }
}
```

Übungsaufgabe 7.7:

1. Anhand Abbildung 7.47 im Buch lässt sich sehr gut erkennen, dass die gesuchte Funktion **f()** eine skalierte und um **0.5** Einheiten in Richtung der +y-Achse verschobene Version der Funktion **g()** ist. Somit gilt:

1. Schritt:

f(x) = f_1(x) + 0.5

Aus den gegebenen Werten folgt: **f_1(0) = 0** und **f_1(100) = 1.5**

$$f_1(x) \;=\; t * g(x) \;\Leftrightarrow\; f_1(x) \;=\; t * x^2 \;\; und \;\; f_1(100) \;=\; 1.5 \;=\; t * 100^2$$
$$\Rightarrow \quad t = 0.00015 \quad \Rightarrow \quad f_1(x) \;=\; 0.00015 * x^2$$

Die Definition der gesuchten Funktion lautet daher:

$$f(x) \;=\; 0.00015 * x^2 + 0.5$$

2. Die Anfang des Abschnittes aufgeführte Beschreibung des Aussehens des Graphen von **f()** lässt eine Vielzahl von verschiedenen, gültigen Definitionen zu. Legt man der Herleitung die Funktion **u(x) = x^n** zugrunde, ist das Ergebnis:

$$f(x) \;=\; \frac{1.5}{100^n} * x^n + 0.5 \qquad\qquad für\; n > 0$$

Je größer der Exponent **n > 1** ist, desto flacher verläuft die Funktion zu Beginn des Intervalls [**0 .. lens_radius**], und umso später erfolgt der Anstieg bis zum vorgegebenen **f(lens_radius) = 2**. Die anfangs beschriebene Kurvenform bleibt aber nicht für alle positiven Exponenten n erhalten: Ist **n < 1**, erfolgt der starke Anstieg der Funktion bereits zu Beginn des Intervalls, wodurch die Linse ein anderes Aussehen annimmt.

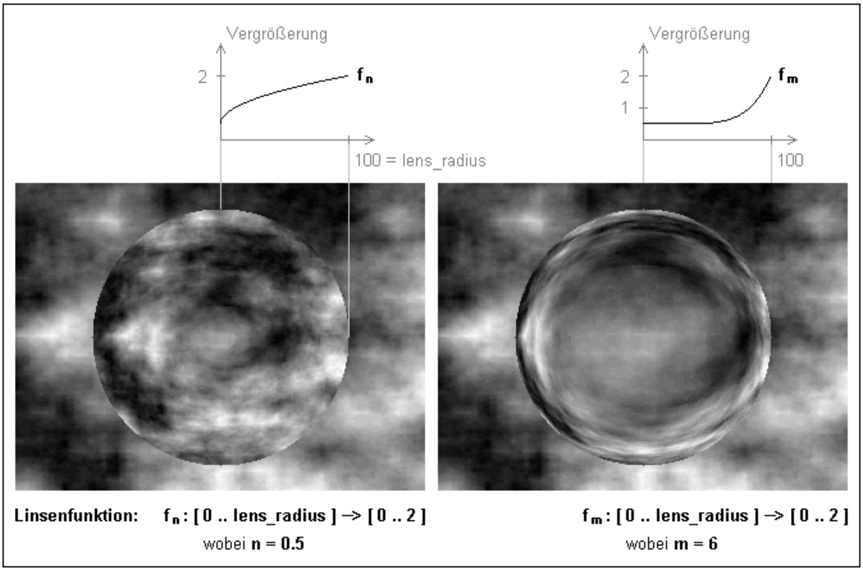


Abb. 7.2: Verläufe der zweiten Version der Funktion *f()* sowie die dazugehörigen Linsen

Übungsaufgabe 7.8:

Wir legen fest: **lens_radius = 100**. Die Funktion **get_angle()** wird zunächst an der y-Achse gespiegelt durch Multiplikation des Parameters mit der Konstante **–1**; es gilt somit:

$$get\_angle(x) = get\_angle_1(-x) \quad wobei$$
$$get\_angle_1(-100) = 0 \quad und \quad get\_angle_1(0) = 5$$

Anschließend folgt eine Verschiebung in Richtung der +x-Achse, um eine Funktion **get_angle**$_2$**()** zu erhalten, die **g()** ähnlich ist:

$$get\_angle_1(z) = get\_angle_2(z+100) \quad wobei$$
$$get\_angle_2(0) = 0 \quad und \quad get\_angle_2(100) = 5$$

Da **get_angle**$_2$**(z) = t * s(z) = t * x**^{3}, folgt:

$$get\_angle_2(z) = \frac{5}{100^3} * z^3 \quad \Rightarrow \quad get\_angle_1(z) = \frac{5}{100^3} * (z+100)^3$$
$$\Rightarrow \quad get\_angle(x) = \frac{5}{100^3} * (100-x)^3$$

7.13 Besprechung des Projektes

Erster Schritt:

Die Ausführung der beiden Schleifen lässt sich in die Funktion `mandelbrot_set()` einbauen, die einmal vor der Hauptschleife des Programms aufgerufen wird:

```
void mandelbrot_set
( long mx, long my, pixel_32 *pal, pixel_32 *screen )
{
  for( long sy=0 ; sy<yscale ; sy++ )
  {
    for( long sx=0 ; sx<xscale ; sx++ )
    {
      long double cx = (end_cx - begin_cx) /
        double( xscale ) * sx + begin_cx;
      long double cy = (begin_cy - end_cy) /
        double( yscale ) * sy + end_cy;
```

```
      uchar color =
        uchar( mandelbrot( complex( cx, cy ) ) );

      fractal[ sy * xscale + sx ] = color;

      screen[ (my+sy) * x_res + (mx+sx) ] =
        pal[ color ];
    }
  }
}
```

Interessant an dieser Funktion ist die Tatsache, dass sie zusätzlich zu der Beschriftung von `fractal[]` die Mandelbrotmenge auch auf dem Bildschirm darstellt. Auf diese Weise kann der Benutzer die Entstehung des Fraktals in Echtzeit verfolgen und muss nicht vor einem schwarzen Bildschirm warten.

Zweiter Schritt:

Die Farbpalette **`palette`** wird wie gehabt unter Verwendung der Klasse `primary_color` erzeugt. Die Übertragung der Farbwerte von `fractal[]` nach **`buffer`** kann entweder direkt oder durch Definition einer neuen Funktion `bitmap_32::copy()` erfolgen:

```
bitmap_32 buffer( xscale, yscale );

long mx = (x_res - xscale) / 2;
long my = (y_res - yscale) / 2;

mandelbrot_set( mx, my, palette.components, screen );

while( 1 )
{
  if( input.check() == 1 ) break;
  if( input.event_key != 0 ) break;

  long pixel_count = xscale * xscale;

  for( long x=0 ; x<pixel_count ; x++ )
```

```
    buffer.picture[ x ] =
      palette.components[ fractal[ x ] ];

  buffer.display( mx, my, screen );
}
```

Dritter Schritt:

Die wichtigsten Informationen über den Umriss des Quadrates, die Position `pos` seiner oberen linken Ecke innerhalb der Bitmap `buffer` sowie seine Seitenlänge `size` in Pixel, lassen sich zu einer Struktur `square` zusammenfassen:

```
struct square
{
  long size;
  svertex pos;
  pixel_32 color;

  void display( bitmap_32 *bmp );

  square( long s, svertex p, pixel_32 c )
   {  size = s;  pos = p;  color = c;  }
};
```

Die Darstellung der vier Linien seines Umrisses innerhalb der Bitmap lässt sich wie folgt durchführen:

```
void square::display( bitmap_32 *bmp )
{
  for( long z=0 ; z<=size ; z++ )
  {
    bmp->picture
      [pos.sy * bmp->xscale + (pos.sx+z)]=color;

    bmp->picture
      [(pos.sy+size)*bmp->xscale+(pos.sx+z)]=color;

    bmp->picture
      [(pos.sy+z) * bmp->xscale + pos.sx]=color;
```

```
    bmp->picture
      [(pos.sy+z)*bmp->xscale+(pos.sx+size)]=color;
  }
}
```

Wir definieren eine einzige Variable `lens` vom Typ `square`, die anfangs in der Mitte der Bitmap eingezeichnet werden soll:

```
const long scale = 75;
square lens
(
  scale,
  svertex( (xscale - scale) / 2, (yscale - scale) / 2 ),
  pixel_32( 255, 255, 255 )
);
```

Die Verschiebung des Quadrates innerhalb der Bitmap erfolgt während der Ausführung von `handle_input()`:

```
uchar handle_input( square *lens )
{
  if( input.check() == 1 ) return 1;
  if( input.event_key == VK_ESCAPE ) return 1;

  if( input.key_pressed( VK_LEFT ) )
    {  lens->pos.sx--;
       if( lens->pos.sx < 0 ) lens->pos.sx = 0;  }

  if( input.key_pressed( VK_RIGHT ) )
    {  lens->pos.sx++;
       if( lens->pos.sx+lens->size >= xscale )
         lens->pos.sx = xscale-1-lens->size;  }

  if( input.key_pressed( VK_UP ) )
    {  lens->pos.sy--;
       if( lens->pos.sy < 0 ) lens->pos.sy = 0;  }
```

```
  if( input.key_pressed( VK_DOWN ) )
    {  lens->pos.sy++;
       if( lens->pos.sy+lens->size >= yscale )
         lens->pos.sy = yscale-1-lens->size;  }

  return 0;
}
```

Vierter Schritt:

Die Neuberechnung des Fraktals wird über die globale Variable `redraw_fractal` gesteuert, deren Initialisierungswert 0 beträgt:

```
if( input.event_key == VK_RETURN ) redraw_fractal = 1;
```

Die eigentliche Durchführung dieser Neuberechnung erfolgt während der Ausführung der Hauptschleife des Programms:

```
if( redraw_fractal == 1 )
{
  long double bcx = (end_cx - begin_cx) /
    double(xscale) * lens.pos.sx + begin_cx;

  long double ecx = (end_cx - begin_cx) /
    double(xscale) * (lens.pos.sx+lens.size) + begin_cx;

  long double bcy = (begin_cy - end_cy) /
    double(yscale) * (lens.pos.sy+lens.size) + end_cy;

  long double ecy = (begin_cy - end_cy) /
    double( yscale ) * lens.pos.sy + end_cy;

  begin_cx = bcx;  end_cx = ecx;
  begin_cy = bcy;  end_cy = ecy;
```

```
  mandelbrot_set( mx, my, palette.components, screen );
  redraw_fractal = 0;
}
```

Texturprojektion

8.12 Besprechung der Übungsaufgaben

Übungsaufgabe 8.1:

Das Prinzip des Clear Reduction Algorithmus bleibt auch bei der Verwendung inverser z-Koordinaten unverändert: *Sämtliche* z-Koordinaten **sz**, die während der Darstellung eines Frames den gesetzten Pixeln zugeordnet werden, befinden sich zwischen zwei eindeutigen Grenzen **sz_min** und **sz_max**: **sz_min** ≤sz ≤ **sz_max**.

Dabei ist zu beachten, dass **sz_min** durchaus größer **1.0** sein kann – da wir inverse z-Koordinaten verwenden, gilt: **sz_min = 1.0 / z_min**. Die minimale Welt-z-Koordinate **z_min** darf positive Zahlen kleiner **1.0** annehmen, **0.5** beispielsweise. Daraus folgt: **sz_min = 1.0 / 0.5 = 2 > 1**.

Nach demselben Prinzip gilt, dass **sz_max** im Allgemeinen größer **0.0** ist – **z_max** ist in vielen Fällen beschränkt, das heißt es existiert eine reelle Zahl **border > 0** mit **border > z_max**. Demnach gilt: **sz_max = (1.0 / z_max) > (1.0 / border) > 0.0**.

Daraus folgt, dass der globalen Variable `clear_translation`, die mit `0.0` initialisiert wird, in jedem Frame der Wert `(1.0 / z_min)` hinzuaddiert werden muss:

```
if( clear_translation > max_clear_translation )
  clear_translation += (1.0 / z_min);

else
{
  for( long x=0 ; x<x_res*y_res ; x++ ) zbuffer[x] = 0;
  clear_translation = 0;
}
```

Bei einer Bildwiederholfrequenz von 24 Bildern pro Sekunde werden in einer Stunde 24 * 60 * 60 = 86400 Bilder angezeigt. Demnach gilt in diesem Zeitpunkt:

```
clear_translation == 86400 * (1.0 / z_min) == (86400 / z_min)
```

Der Z-Buffer darf nach einer Stunde gelöscht werden, `max_clear_translation` wird demnach mit diesem Wert initialisiert:

```
const double max_clear_translation = 86400 / z_min;
```

Übungsaufgabe 8.2.1:

Die Elemente des Arrays `polygon::points[]` werden während der Ausführung von `polygon::load()` mit verallgemeinerten Texturkoordinaten initialisiert. Diese werden genau wie die speziellen Texturkoordinaten entlang der Polygonseiten und Rasterzeilen interpoliert; in der innersten Schleife des Programms werden diese schließlich durch Multiplikation mit der Breite und Höhe der Textur in gültige Texturkoordinaten umgewandelt.

```
while( length-- > 0 )
{
  if( act_z > zbuffer[ offset ] )
  {
    double z = 1.0 / (act_z - clear_translation);

    long tx = long( act_tx * (surface.xscale-1) * z ) %
              surface.xscale;
    if( tx < 0 ) tx += surface.xscale;

    long ty = long( act_ty * (surface.yscale-1) * z ) %
              surface.yscale;
    if( ty < 0 ) ty += surface.yscale;

    sbuffer[ offset ] =
      surface.picture[ ty * surface.xscale + tx ];
    zbuffer[ offset ] = act_z;
  }

    offset++;
    act_z += z_step;
    act_tx += tx_step;  act_ty += ty_step;
}
```

Übungsaufgabe 8.2.2:

Die Umwandlung der allgemeinen in gültige Texturkoordinaten erfolgt nach demselben Prinzip wie in Übungsaufgabe 8.2.1; neu ist, dass die Textur, auf die sich diese verallgemeinerten Koordinaten beziehen, in jedem Pixel auf der Grundlage der aktuellen z-Koordinate ermittelt wird. Der vollständige Quelltext des Programms a8_11_4 befindet sich auf der CD im gleichnamigen Verzeichnis.

Übungsaufgabe 8.3, OpenGL:

```
double tx_scale = 2.5, ty_scale = 2.5;
double tx_translation = 0, ty_translation = 0;

void polygon::display( vertex *vs )
{
  surface.activate();

  glBegin( GL_POLYGON );

    for( long x=0 ; x<point_count ; x++ )
    {
     tx_translation += 0.001;

     glTexCoord2d
     (
       points[ x ].tx*tx_scale + tx_translation,
       points[ x ].ty*ty_scale + ty_translation
     );

      vertex pos = vs[ points[ x ].vertex_offset ];

      glVertex3d( pos.wx, pos.wy, pos.wz );
    }

  glEnd();
}
```

Übungsaufgabe 8.3, DirectX:

```
float tx_scale = 2.5, ty_scale = 2.5;
float tx_translation = 0, ty_translation = 0;

void polygon::display( vertex *vs )
{
  tx_translation += 0.005f;

   for( long x=0 ; x<point_count ; x++ )
      wpoint[ x ] = hvertex
      (
        vs[ points[ x ].offset ],
       points[ x ].tx*tx_scale + tx_translation,
       points[ x ].ty*ty_scale + ty_translation
      );

   screen_interface.draw_polygon
   ( point_count, wpoint, tx_offset );
}
```

Übungsaufgabe 8.4:

Die Verarbeitung der Benutzereingaben erfolgt beim vorgegebenen Programm während der Ausführung der Funktion `handle_input()` – um die Aufgabe zu lösen, müssen lediglich alle Rotationswinkel und Versetzungswerte negiert werden. Der vollständige Quelltext des Programms `a8_18_2` befindet sich auf der CD im gleichnamigen Verzeichnis.

Übungsaufgabe 8.5:

Die Lösung dieser Aufgabe sieht lediglich das Einfügen der neuen Versionen der Klassen `wireframe` und `matrix` zusammen mit dem Datentyp `local_system` und den Funktionen `move_ls()` und `roate()` in den Quelltext des vorgegebenen Programms `a8_18` vor. Der vollständige Quelltext des gesuchten Programms befindet sich auf der CD im Verzeichnis `a8_18_3_4`.

Übungsaufgaben 8.6 und 8.7:

Die vollständigen Quelltexte der Programme `a8_19_3` und `a8_19_4` befinden sich auf der CD in den gleichnamigen Verzeichnissen.

8.13 Besprechung der Projekte

Projekt a8_10, erster Schritt:

Das Programm a8_10_1 ist lediglich eine geringfügig veränderte Version von a6_12: Der Lichtvektor light besitzt die Komponenten (0.2, -0.2, 1), bei der Berechnung der Farbintensität mit Hilfe von mt() wird der Exponent 4.0 eingesetzt, die eingesetzte blaue Primärfarbe lässt sich mit Hilfe des folgenden Ausdrucks definieren:

```
pixel_32 borders[ 5 ] =
{
  pixel_32(   0,   0,  80 ), pixel_32(   0,   0, 180 ),
  pixel_32(   0,   0, 240 ), pixel_32( 128, 128, 255 ),
  pixel_32( 255, 255, 255 ),
};
primary_color blue;  blue.load( 256, 5, borders );
```

Bei dem dargestellten Polyeder "sphere.tg2" handelt es sich um eine Sphärenapproximation des Programms a6_7_3 mit dl = 64.

Projekt a8_10, zweiter Schritt:

```
void bitmap_32::display( long sx, long sy )
{
  long z = 0;

 glBegin( GL_POINTS );

    for( long y=0 ; y<yscale ; y++ )
       for( long x=0 ; x<xscale ; x++ )
       {
       glColor3ub( picture[z].red, picture[z].green,
                 picture[z].blue );

       glVertex2d( sx+x, sy+y );

         z++;
       }
```

```
  glEnd();
}
```

Projekt a8_10, dritter Schritt:

```
////////////////         a8_10_3         ///////////////
//                                                     //
//  Projektion einer Textur auf der Oberfläche        //
//  eines zweidimensionalen Polygons                   //
//  Darstellungsart: Hardwarebeschleunigt, OpenGL      //
//                                                     //
/////////////////////////////////////////////////////////

int APIENTRY WinMain
(
  HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int iCmdShow
)
{
  screen_interface.open_window(hInstance, 800, 600, 32);
  glMatrixMode( GL_PROJECTION );  glLoadIdentity();
  gluOrtho2D( 0, x_res-1, y_res-1, 0 );

  glTexEnvf
  ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
  glEnable( GL_TEXTURE_2D );

  glClearColor( 1, 1, 1, 0 );

  texture blue( "blue.bmp" );

  long top_x = long( (x_res - blue.xscale) / 2 );
  long top_y = long( (y_res - blue.yscale) / 2 );

  while( 1 )
  {
```

```
    if( input.check() == 1 ) break;
    if( input.event_key != 0 ) break;

    glClear( GL_COLOR_BUFFER_BIT );

   blue.activate();

   glBegin( GL_POLYGON );

     glTexCoord2d( 0, 0 );
     glVertex2d( top_x, top_y );

     glTexCoord2d( 1, 0 );
     glVertex2d( top_x + blue.xscale, top_y );

     glTexCoord2d( 1, 1 );
     glVertex2d(top_x+blue.xscale, top_y+blue.yscale);

     glTexCoord2d( 0, 1 );
     glVertex2d( top_x, top_y + blue.yscale );

   glEnd();

    screen_interface.swap_buffers();
  }

  return input.msg.wParam;
}

//////////////          Ende a8_10_3          /////////////
```

Projekt a8_10, vierter Schritt:

Die Länge und Breite des Rechtecks, in dem die Bitmap zu sehen ist, werden mit Hilfe von zwei globalen Variablen `xscale` und `yscale` angegeben, deren Werte sich in Abhängigkeit von den Tastatureingaben verändern. Der vollständige Quelltext des Programms `a8_10_4` befindet sich auf der CD im gleichnamigen Verzeichnis.

Projekt a8_10, fünfter Schritt:

Die globale Variable `window_pos` enthält die Koordinaten der oberen linke Ecke des Rahmens, während `window_size` die Seitenlänge des Quadrates angibt. Die Darstellung dieses Rahmens erfolgt mit Hilfe der folgenden Funktion:

```
long window_size = 500;
svertex window_pos = (150, 50);

void display_window( void )
{
  glColor3ub( 0, 0, 0 );
  glBegin( GL_LINE_LOOP );
    glVertex2d( window_pos.sx, window_pos.sy );
    glVertex2d( window_pos.sx + window_size,
                window_pos.sy );
    glVertex2d( window_pos.sx + window_size,
                window_pos.sy + window_size );
    glVertex2d( window_pos.sx,
                window_pos.sy + window_size );
  glEnd();
}
```

Die hier vorgestellte Definition der Klasse `marble` erfolgt im Hinblick auf die in den nächsten Schritten erfolgenden Erweiterungen; Radius und Geschwindigkeit der Kugel werden von zwei *globalen* Variablen `blue_radius` und `blue_speed` angegeben, deren Definition in "`global_definitions.h`" erfolgt:

```
double blue_radius = 27;
double blue_speed = 5;

marble::marble( texture *t )
{
  surface = t;

  pos.sx = (rand() % long(window_size-2*blue_radius)) +
            window_pos.sx + blue_radius;
  pos.sy = (rand() % long(window_size-2*blue_radius)) +
            window_pos.sy + blue_radius;
```

```
  while( dir.length() == 0 )
       dir = vector( (rand() % 100) - 50,
                     (rand() % 100) - 50 );

  dir.set_length( blue_speed );
}

void marble::update_pos( void )
{
  pos = pos + dir;

  adjust_pos( blue_radius, &pos, &dir );
}
```

Der Konstruktor initialisiert die Position des Mittelpunktes mit zufälligen Koordinaten, die innerhalb des Rahmens liegen. Durch die `while()`-Schleife wird sichergestellt, dass die Richtung des Vektors `marble::dir` zufällig und ungleich **0** ist – ansonsten würde sich die Kugel nicht bewegen. Die Länge dieses Vektors stellt zugleich die Geschwindigkeit der Kugel dar; diese ist durch die globale Variable `blue_speed` festgelegt.

Die Bereichskorrektur erfolgt während der Ausführung von `adjust_pos()`:

```
void adjust_pos
( double radius, svertex *pos, vector *dir )
{
  double min_sx = window_pos.sx + radius;
  double max_sx = window_pos.sx + window_size - radius;
  double min_sy = window_pos.sy + radius;
  double max_sy = window_pos.sy + window_size - radius;

  if( pos->sx < min_sx )
  {  pos->sx = min_sx;  dir->x = -dir->x;  }
  if( pos->sx > max_sx )
  {  pos->sx = max_sx;  dir->x = -dir->x;  }
  if( pos->sy < min_sy )
  {  pos->sy = min_sy;  dir->y = -dir->y;  }
```

```
  if( pos->sy > max_sy )
  {  pos->sy = max_sy;  dir->y = -dir->y;  }
}
```

Projekt a8_10, sechster Schritt:

Die Grundidee hierbei ist einfach: Wenn ein Pixel die transparente Farbe `tr` besitzt, wird seiner Alphakomponente der Wert `0` zugewiesen; ansonsten ist seine Farbe zu 100% sichtbar:

```
void bitmap_32::load( char *filename, pixel_32 tr )
{
  load( filename );

  for( long x=0 ; x<xscale*yscale ; x++ )
  {
   if( picture[ x ] == tr )
     picture[ x ].alpha = 0;

    else picture[ x ].alpha = 255;
  }
}
```

Die als transparent definierten Pixel werden bei der Darstellung der Textur mit `glTexCoord2d()` automatisch ausgeblendet.

Projekt a8_10, siebter Schritt:

Von großer Wichtigkeit ist, dass die Funktionen sämtlicher Schaltflächen weder Parameter entgegennehmen noch Ausgabewerte besitzen dürfen, damit diese dem Zeiger `button :: *function` zugewiesen werden können. Der vollständige Quelltext des Programms `a8_10_7` befindet sich auf der CD im gleichnamigen Verzeichnis.

```
double blue_radius = 27;
double blue_speed = 5;

void increase_blue_radius( void )
{  if( (blue_radius += 0.5) > 80 ) blue_radius = 80;  }
```

```
void decrease_blue_radius( void )
{  if( (blue_radius -= 0.5) <  8 ) blue_radius =  8;  }

void increase_speed( void )
{  if( (blue_speed += 0.02) > 20 ) blue_speed = 20;  }

void decrease_speed( void )
{  if( (blue_speed -= 0.02) < 0.1 ) blue_speed = 0.1;  }
```

Der virtuelle Test zur Erhöhung der Geschwindigkeit verändert lediglich die globale Variable `blue_speed`; damit diese Veränderung auch auf dem Bildschirm sichtbar wird, muss die Länge von `marble::dir` in jedem Frame entsprechend angepasst werden:

```
void marble :: update_pos( void )
{
  dir.set_length( blue_speed );

  pos = pos + dir;

  adjust_pos( blue_radius, &pos, &dir );
}
```

Projekt a8_10, achter Schritt:

Der vollständige Quelltext des Programms `a8_10_7` befindet sich auf der CD im gleichnamigen Verzeichnis.

Projekt a8_10, neunter Schritt:

Um die Abstufungen der Farbe **Blau** in rote Farbabstufungen zu ändern, wird die Farbe jedes Pixels von **(c, c, b)** nach **(b, c, c)** umgewandelt – auf diese Weise ändert sich zwar die Farbe, die Helligkeit bleibt jedoch erhalten:

```
void bitmap_32 :: load
( char *filename, pixel_32 tr, long c )
{
  load( filename );

  for( long x=0 ; x<xscale*yscale ; x++ )
```

```
  {
    if( picture[ x ] == tr )
      picture[ x ].alpha = 0;

    uchar red_component;

    switch( c )
    {
     case 0 : picture[ x ].red = picture[ x ].blue;
              picture[ x ].blue = picture[ x ].green;
              break;

      case 1 : picture[ x ].green = picture[ x ].blue;
               picture[ x ].blue = picture[ x ].red;
               break;

      case 3 : red_component = picture[ x ].red;
               picture[ x ].red = picture[ x ].green =
               picture[ x ].blue;
               picture[ x ].blue = red_component;
               break;

      case 4 : picture[ x ].red = picture[ x ].blue;
               break;

      case 5 : picture[ x ].green = picture[ x ].blue;
               break;

      case 6 : picture[ x ].red =
               picture[ x ].green = picture[ x ].blue;
    }
  }
}
```

Das Programm a8_10_9 definiert ein globales Array aus sieben Texturen, die jeweils eine der festgelegten Farbabstufungen besitzen.

```
texture colors[ 7 ];
long act_color = 0;

// Aufbau des Arrays in initialise_world():

for( long x=0 ; x<7 ; x++ )
   colors[x].load("blue.bmp", pixel_32(255,255,254), x);
```

Die beiden neuen Schaltflächen de- bzw. inkrementieren den Wert der globalen Variable `act_color`; in diesem Zusammenhang ist die ursprüngliche Art der Verarbeitung von Eingaben am besten geeignet – der Farbwechsel darf nur dann erfolgen, wenn der Cursor sich sowohl beim Drücken als auch beim Loslassen der linken Maustaste über der Schaltfläche befindet.

Projekt a8_10, zehnter Schritt:

Aufgrund der geringen Voraussetzungen braucht die gelbe Kugel nicht in Form eines Elementes vom Typ `marble` definiert zu werden; selbst im Hinblick auf die weiteren Erweiterungen sind lediglich eine globale Variable `yellow_radius` und eine zweidimensionale Position `yellow_pos` erforderlich. Die Darstellung der Kugel erfolgt mit Hilfe der Funktion `display_texture()`.

Die Voraussetzung an die Anfangsposition der blauen Kugeln ist einfach zu erfüllen: Es wird so lange eine neue zufällige Position berechnet, bis der Abstand zur gelben Kugel groß genug ist:

```
void marble::load( texture *t )
{
  surface = t;

  vector v;

  do
  {
    pos.sx =
      (rand() % long( window_size - 2 * blue_radius )) +
       window_pos.sx + blue_radius;

    pos.sy =
      (rand() % long( window_size - 2 * blue_radius )) +
```

```
      window_pos.sy + blue_radius;

   v = pos -
     svertex( window_pos.sx + window_size/2,
              window_pos.sy + window_size/2 );

  } while( v.length() < (blue_radius + yellow_radius) );

  while( dir.length() == 0 )
    dir = vector( (rand() % 100) - 50,
                  (rand() % 100) - 50 );
}
```

Kollisionserkennung, Berechnung des Punktes p

Grundlage: $p = np + t * \overrightarrow{dir}$

Gesucht: $t \in R \quad mit \quad s(\ yp - p,\ \overrightarrow{dir}\ ) = 0$

Berechnung:

$$s(\overrightarrow{yp} - \vec{p}, \overrightarrow{dir}) = 0 \overset{(1)}{\Leftrightarrow} s(\overrightarrow{yp} - \overrightarrow{np} - t * \overrightarrow{dir}, \overrightarrow{dir}) = 0 \overset{(2)}{\Leftrightarrow}$$

$$s(\overrightarrow{yp} - \overrightarrow{np}, \overrightarrow{dir}) - t * s(\overrightarrow{dir}, \overrightarrow{dir}) = 0 \overset{(3)}{\Leftrightarrow} t = \frac{s(\overrightarrow{yp} - \overrightarrow{np}, \overrightarrow{dir})}{s(\overrightarrow{dir}, \overrightarrow{dir})}$$

$$\overset{(1)}{\Leftrightarrow} \quad p = np + \frac{s(\overrightarrow{yp} - \overrightarrow{np}, \overrightarrow{dir})}{s(\overrightarrow{dir}, \overrightarrow{dir})} * \overrightarrow{dir}$$

(1): Definition von $\vec{p}$

(2): Additivität und Homogenität des Skalarproduktes **s()**

(3): Einfache algebraische Umformung

Kollisionserkennung, Berechnung des Punktes tp

Grundlage: $tp = p + \vec{v}'$

Berechnung:

$$dist(\ yp, p\ )^2 + dist(\ p, tp\ )^2 = dist(\ tp, yp\ )^2 \quad \Leftrightarrow$$

$$dist(\ p, tp\ ) \ = \ \sqrt{dist(\ tp, yp\ )^2 - dist(\ yp, p\ )^2}$$

$$wobei \quad dist(\ tp, yp\ ) \ = \ (blue\_radius + yellow\_radius)$$

$$dist(\ yp, p\ ) \ = \ |\ yp - p\ |$$

Schließlich gilt:

$$\vec{v}' = t*(-1)*\overrightarrow{dir} \quad mit\ |\vec{v}'| = dist(p, tp) \quad \overset{(1)}{\Leftrightarrow}$$

$$\vec{v}' = -\frac{dist(p, tp)}{|\overrightarrow{dir}|}*\overrightarrow{dir} \quad \overset{(2)}{\Leftrightarrow}$$

$$tp = p - \frac{dist(p, tp)}{|\overrightarrow{dir}|} * \overrightarrow{dir}$$

(1): Formel aus dem ersten Kapitel, die einem Vektor eine bestimmte Länge zuweist

(2): Definition von *tp* und $\vec{v}'$

Kollisionserkennung, Berechnung des neuen Richtungsvektors

Grundlage: $q = tp - \lambda * (yp - tp)$

Gesucht: $\lambda \in R\ mit\ s(-1*\overrightarrow{dir} + \lambda * \vec{w}, \vec{w}) = 0$

wobei $\vec{w} = yp - tp$

Berechnung:

$$s(-1*\overrightarrow{dir} + \lambda * \vec{w}, \vec{w}) = 0 \quad \Leftrightarrow \quad -s(\overrightarrow{dir}, \vec{w}) + \lambda * s(\vec{w}, \vec{w}) = 0 \quad \Leftrightarrow$$

$$\lambda = \frac{s(\overrightarrow{dir}, \vec{w})}{s(\vec{w}, \vec{w})} \quad \Leftrightarrow \quad q = tp - \frac{s(\overrightarrow{dir}, \vec{w})}{s(\vec{w}, \vec{w})} * \vec{w}$$

Bei dieser Umformung werden dieselben Regeln wie bei der Berechnung des Punktes **p** eingesetzt. Für den neuen Richtungsvektor gilt schließlich:

$$c = q + (q - (tp - \overrightarrow{dir}))$$

$$\overrightarrow{dir}' = \tau * (c - tp) \quad \Leftrightarrow \quad \overrightarrow{dir}' = \frac{blue\_speed}{|c - tp|} * (c - tp)$$

Die Kollisionserkennung zwischen der aktuellen blauen und der gelben Kugel kann wie folgt während der Ausführung von `marble::update_pos()` implementiert werden:

```
void marble::update_pos( svertex yp )
{
  dir.set_length( blue_speed );

  pos = pos + dir;
```

```
  adjust_pos( blue_radius, &pos, &dir );

  vector v = yp - pos;

  if( v.length() < blue_radius + yellow_radius )
  {
    svertex p = pos + (dot( yp - pos, dir ) /
                       dot( dir, dir )) * dir;
    v = yp - p;
    double t = sqrt( (blue_radius+yellow_radius) *
                     (blue_radius+yellow_radius) -
                      v.length()*v.length() );
    vector u = (-1)*dir;  u.set_length(t);  pos = p + u;

    vector w = yp - pos;
    double lambda = dot( dir, w ) / dot( w, w );
    svertex q = pos + (-1)*lambda * w;
    svertex c = q + (q - (pos + (-1) * dir));

    dir = c - pos;  dir.set_length( blue_speed );
  }
}
```

Projekt a8_10, elfter Schritt:

```
vector marble::update_pos( svertex yp )
{
  if( dir.length() > 0 ) dir.set_length( blue_speed );

  pos = pos + dir;

  adjust_pos( blue_radius, &pos, &dir );

  vector v = yp - pos;
  vector prev_v = yp - (pos + (-1) * dir);

  if( v.length() < blue_radius + yellow_radius &&
```

```
    v.length() > prev_v.length() )
  {
   v.set_length( blue_radius + yellow_radius );
   pos = yp + (-1) * v;
  }

  else if( v.length() < blue_radius + yellow_radius )
  {
    svertex p = pos + (dot( yp - pos, dir ) /
                       dot( dir, dir )) * dir;
    v = yp - p;
    double t = sqrt( (blue_radius+yellow_radius) *
                     (blue_radius+yellow_radius) -
                      v.length()*v.length() );
    v = (-1) * dir;  v.set_length( t );  pos = p + v;

    v = yp - pos;
    double lambda = dot( -1*dir, v ) / dot( v, v );
    svertex q = pos + lambda * v;
    svertex c = q + (q - (pos + (-1) * dir));

    dir = c - pos;  dir.set_length( blue_speed );

   v.set_length((blue_mass/yellow_mass) * blue_speed);
   return v;
  }

  return vector( 0, 0 );
}
```

In der neuen Version der Funktion `marble::update_pos()` wird vor der bereits bekannten eine neue `if()`-Abfrage ausgeführt, deren Zweck nicht auf den ersten Blick erkennbar ist. Diese Abfrage beseitigt einen Darstellungsfehler, der im Programm `a8_10_10` unter anderem im folgenden Zusammenhang auftritt: Die Geschwindigkeit `blue_speed` der blauen Kugeln besitzt den Minimalwert 0.1, es befinden sich mehrere blaue Kugel in unmittelbarer Nähe der gelben und der Radius `yellow_radius` wird durch langes Drücken der linken Maustaste über der entsprechenden Schaltfläche erhöht.

Der Darstellungsfehler besteht in einer schnellen, unrealistisch aussehenden Drehung der blauen Kugeln um die gelbe. Der Grund für diesen Fehler ist die ungewollte Auslösung der Kollisionsverarbeitung durch die Erfüllung der bekannten Ungleichung: **| yellow_pos – blue_pos | < (yellow_radius + blue_radius)**.

Um diesen Darstellungsfehler zu beheben, muss die Kollisionserkennung erweitert werden: Wir legen fest, dass die im 10. Schritt beschriebene Kollisionsverarbeitung nur dann ausgeführt werden darf, wenn die beschriebene Ungleichung erfüllt **und** die blaue Kugel sich der gelben *nähert*.

Entfernt sich die blaue Kugel jedoch von der gelben und die Ungleichung wird erfüllt, dann soll die blaue Kugel lediglich in Richtung des Vektors **(tp – yp)** verschoben werden, so dass der Abstand zwischen den Mittelpunkten den Wert der Summe beider Radien annimmt.

Projekt a8_13, erster Schritt:

```
for( x=0 ; x<detail_level ; x++ )
{
  vs[ y * detail_level + x ].wx =  x;
  vs[ y * detail_level + x ].wy =  0;
  vs[ y * detail_level + x ].wz = -y;

  vs[ y * detail_level + x ].tx =
    x * (4.0*(surface.xscale-1)/(detail_level-1));
  vs[ y * detail_level + x ].ty =
    y * (4.0*(surface.yscale-1)/(detail_level-1));
}
```

Bei der Texturierung des Polyeders werden **skalierte Texturkoordinaten** eingesetzt: Wenn die letzte Pixelspalte erreicht ist, nimmt die Laufvariable `x` ihren Maximalwert (`detail_level-1`) an, und dem entsprechenden Vertex wird die Texturkoordinate `4.0*(surface.xscale-1)` zugewiesen. Hierbei handelt es sich um eine spezielle Texturkoordinate, nicht um eine verallgemeinerte.

Diese skalierten Texturkoordinaten werden während der Ausführung von `polygon::rasterize()` wie gehabt in gültige Texturkoordinaten umgewandelt:

```
if( act_z > zbuffer[ offset ] )
{
  double inv_z = act_z - clear_translation;
  long tx = long( act_tx / inv_z ) % surface->xscale;
```

```
 long ty = long( act_ty / inv_z ) % surface->yscale;

  sbuffer[ offset ] =
    surface->picture[ ty * surface->xscale + tx ];
  zbuffer[ offset ] = act_z;
}
```

Projekt a8_13, zweiter Schritt:

```
void polyhedron::torus
( long dl, double radius, char *filename )
{
  initialise_vertices( dl );

  const double pi = 3.1415926535;
  matrix m;
  m.rotate_z( (2*pi)/(detail_level-1) );

  long x, y;

  vs[ 0 ] = vertex( -1, 0, 0 );
  for( y=1 ; y<detail_level ; y++ )
  {
    vs[ y*detail_level ] = vs[ (y-1)*detail_level ];

    vs[ y*detail_level ] = m * vs[ y*detail_level ];
  }

  m.clear();
  m.translate( -radius, 0, 0 );
  for( y=0 ; y<detail_level ; y++ )
     vs[ y*detail_level ] = m * vs[ y*detail_level ];

  rotation_symmetry();

 surface.load( filename );
```

```
  for( y=0 ; y<detail_level ; y++ )
    for( x=0 ; x<detail_level ; x++ )
    {
      vs[ y * detail_level + x ].tx =
        x * (16.0*(surface.xscale-1)/(detail_level-1));

      vs[ y * detail_level + x ].ty =
        y * ( 4.0*(surface.yscale-1)/(detail_level-1));
    }

  initialise_polygons();
}
```

Projekt a8_16, erster Schritt:

Während `polyhedron::rectangle()` der bekannten Funktion `polyhedron::plane()` sehr ähnlich ist, wird die Softwaredarstellung eines Drahtgittermodells zum ersten Mal vorgestellt – die zugrundeliegende Technik ist bereits ausführlich im Zusammenhang mit der Visualisierung von Polygonen besprochen worden:

```
uchar project_point( vertex w, svertex *s )
{
  if( w.wz <= z_min ) return 0;

  double inv_z = 1.0 / w.wz;

  s->sx = long( w.wx * inv_z *  pr_cnst + x_res / 2 );
  s->sy = long( w.wy * inv_z * -pr_cnst + y_res / 2 );
  s->sz = 1.0 / w.wz;

  if( s->sx < 0 || s->sx >= x_max ||
      s->sy < 0 || s->sy >= y_max ) return 0;

  return 1;
}

void draw_line
( vertex w_begin, vertex w_end, pixel_32 *sbuffer )
```

```
{
  svertex s_begin, s_end;

  if( project_point( w_begin, &s_begin ) &&
      project_point( w_end, &s_end ) )
    draw_line( s_begin, s_end,
               pixel_32( 255, 255, 255 ), sbuffer );
}

void polyhedron::display_edges( pixel_32 *sbuffer )
{
  draw_line( vs[ 0 ], vs[ detail_level ], sbuffer );

  for( long x=1 ; x<detail_level ; x++ )
  {
    vertex a = vs[ x-1 ];
    vertex b = vs[ x ];
    vertex c = vs[ x + detail_level ];
    vertex d = vs[ x-1 + detail_level ];

    draw_line( a, b, sbuffer );
    draw_line( b, c, sbuffer );
    draw_line( c, d, sbuffer );
  }
}
```

Projekt a8_16, zweiter Schritt:

Um die Funktion `f()` auf die Vertices des Polyeders anwenden zu können, benötigt man die Ausrichtung der Achse `f(t)`, die in der Abbildung mit der Ausgabe des Programms `a8_16_2` dargestellt ist. Diese Ausrichtung wird in eine neue Variable `polyhedron::normal` vom Typ `vector` gespeichert. Nach der Initialisierung des Polyeders liegen sämtliche Vertices innerhalb der xz-Ebene – die Komponenten dieses Vektors besitzen demnach die Anfangswerte `(0, 1, 0)`.

Diese Ausrichtung ändert sich mit jeder Bewegung des Polyeders – während der Ausführung von `polyhedron::update_pos()` muss demnach auch dieser Vektor mit der entsprechenden Matrix multipliziert werden.

Die Anwendung der Funktion `f()` auf die Vertices des Polyeders erfolgt in `polyhedron::display()` – zu diesem Zweck werden jedoch lokale Variablen vom Typ `vertex` eingesetzt, die dreidimensionalen Positionen der Vertices innerhalb des Arrays `polyhedron::vs[]` bleiben unverändert:

```
double f( double x, long detail_level )
{
  const double pi = 3.1415926535;
  double t = x * 2 * pi / (detail_level - 1);

  return 3 * sin( t*2 ) + 2 * sin( t );
}

void polyhedron::display_edges( pixel_32 *sbuffer )
{
  draw_line
  (
    vs[ 0 ] + f( 0, detail_level ) * normal,
    vs[ detail_level ] + f( 0, detail_level ) * normal,
    sbuffer
  );

  for( long x=1 ; x<detail_level ; x++ )
  {
   vertex a = vs[x-1] + f(x-1, detail_level) * normal;

   vertex b = vs[x] + f(x, detail_level) * normal;

   vertex c = vs[x + detail_level] +
            f( x, detail_level ) * normal;

   vertex d = vs[ x-1 + detail_level ] +
            f( x-1, detail_level ) * normal;

    draw_line( a, b, sbuffer );
    draw_line( b, c, sbuffer );
```

```
    draw_line( c, d, sbuffer );
  }
}
```

Projekt a8_16, dritter Schritt:

```
double add = 0;
const double pi = 3.1415926535;

double f( double x, long detail_level )
{
  double t = x * 2 * pi / (detail_level - 1);

  return 3 * sin( (t-add)*2 ) + 2 * sin( t-add );
}

void polyhedron::display_edges( pixel_32 *sbuffer )
{
  draw_line
  (
     // ...
  );

  for( long x=1 ; x<detail_level ; x++ )
  {
     // ...
  }

  add += 0.1;  if( add > 2*pi ) add -= 2*pi;
}
```

Projekt a8_16, vierter Schritt:

Die vollständigen Quelltexte der Programme a8_16_4 und a8_16_5 befinden sich auf der CD in den gleichnamigen Verzeichnissen.

Aufbau und effiziente Darstellung dreidimensionaler Landschaften

9.6 Besprechung der Übungsaufgaben

Übungsaufgabe 9.1:

```
void landscape :: display( viewport *user )
{
  vertex points[ 8 ];
  vertex hull[ 4 ];

  user->get_points( points );
  rectangle( points, hull );

  double world_size = vs[detail_level-1].wx - vs[0].wx;
  double square_size = vs[ 1 ].wx - vs[ 0 ].wx;

  svertex spoint[ 4 ];
  for( long x=0 ; x<4 ; x++ )
  {
    spoint[ x ].sx = long
    ( ( hull[ x ].wx + 0.5*world_size) / square_size );

    spoint[ x ].sz = long
    ( (-hull[ x ].wz + 0.5*world_size) / square_size );
  }

  // ...
}
```

Übungsaufgabe 9.2:

Die Programme a9_12 und a9_13 arbeiten nach demselben Prinzip wie ihre Vorgänger a9_7 und a9_8: Die konvexe Hülle wird mit Hilfe der Funktion convex_hull() berechnet und in Form eines Linienzuges auf dem Bildschirm ausgegeben. Auch die Bestimmung und Ausgabe der Landschaftsquadrate sind in beiden Fällen gleich.

9.7 Besprechung des Projektes

Erster Schritt:

Die Konstanten, die bei der Definition der Variable user eingesetzt werden, sollten im Hinblick auf die späteren Programme möglichst realistische Werte erhalten:

```
viewport user( x_res, y_res, pr_cnst, z_min, local_z_max );
```

Hierbei werden dieselbe Auflösung, Projektionskonstante und minimale z-Koordinate wie im übergeordneten Programm eingesetzt. Aus Übersichtlichkeitsgründen kann z_max nicht in diesem Zusammenhang verwendet werden, stattdessen wird auf eine neue globale Konstante local_z_max mit dem Wert 10.0 zurückgegriffen.

Für die Bewegung der dreidimensionalen Welt wird ein lokales Koordinatensystem namens global_view definiert, der Position und Ausrichtung der Landschaft anzeigt. Dieses wird am Anfang des Programms an die Position (0, 0, 50) versetzt, und beim Drücken der Pfeiltasten um lokale Achsen rotiert, die parallel zu den Weltachsen verlaufen.

Die im letzten Kapitel in Zusammenhang mit der Navigation im dreidimensionalen Raum kennen gelernten Anweisungen übertragen die Bewegungen in die Modelview Matrix, die erforderlich sind, um die Landschaft entsprechend zu versetzen und auszurichten:

```
matrix m;

// ...

m.clear();
m.columns
( global_view.x_axis, global_view.y_axis,
  global_view.z_axis);
```

```
m.translate
( global_view.pos.wx, global_view.pos.wy,
  global_view.pos.wz );

m.adjust_hardware();


planet.display_points();
user.display();
```

Diese Anweisungen sind Teil der äußersten Programmschleife in `WinMain()` der ersten Version des Programms `a9_1`.

Der Aufruf von `glVertex3d()` während der Ausführung der Darstellungsfunktionen von `planet` und `user` bewegt die Punkte automatisch wie von der internen Modelview Matrix vorgeschrieben.

Die letzte Variable besitzt den Typ `viewport`, der wiederum eine `public`-Ableitung der Klasse `wireframe` ist. Die hierbei zum Einsatz kommende Funktion `wireframe::display()` hat sich seit dem letzten Kapitel nicht verändert, und setzt weiterhin eine Softwarematrix ein, um die mathematisch korrekten Vertices aus `user.vs[]` mit Hilfe des Orthonormalsystems `user.ls` auszurichten.

Die beschriebene Vorgehensweise lässt sich noch weiter optimieren – die Grundidee der zweiten, optimierten Version des Programms `a9_1` ist, dass jeder Gegenstand vor seiner Darstellung *eine eigene Version* der Modelview Matrix in die Hardware überträgt; die oben beschriebenen Anweisungen werden von `WinMain()` in die jeweilige Darstellungsfunktion verschoben:

```
void wireframe :: display( void )
{
  matrix m;

  m.columns( ls.x_axis, ls.y_axis, ls.z_axis );
  m.translate( ls.pos.wx, ls.pos.wy, ls.pos.wz );

  m.columns
  ( global_view.x_axis, global_view.y_axis,
    global_view.z_axis );
  m.translate
  ( global_view.pos.wx, global_view.pos.wy,
    global_view.pos.wz );
```

```
  m.adjust_hardware();

  glBegin( GL_LINES );

    for( long x=0 ; x<line_count ; x++ )
    {
      glColor3ub
      ( es[ x ].color.red, es[ x ].color.green,
        es[ x ].color.blue );

      vertex begin = vs[ es[ x ].begin ];
      vertex end = vs[ es[ x ].end ];

      glVertex3d( begin.wx, begin.wy, begin.wz );
      glVertex3d( end.wx, end.wy, end.wz );
    }

  glEnd();
}
```

Hierbei wird zunächst eine Softwarematrix definiert, in der *zuerst* die Anweisungen gespeichert werden, welche die mathematisch korrekten Positionen aus `wireframe::vs[]` in gültige Weltpositionen umwandeln. Erst anschließend erfolgt die Ausrichtung und Verschiebung, die von `global_view` vorgegeben sind.

Die Ende des letzten Kapitels beschriebene Umkehrung der Reihenfolge der Matrizenmultiplikation beim Aufbau der Modelview Matrix bezieht sich lediglich auf den Aufruf der Funktionen `glTranslated()`, `glScaled()` und `glRotated()`.

Wird der Grafikhardware hingegen eine bereits vollständige Matrix `m` durch `matrix::adjust_hardware()` bzw. `glMultMatrixd()` zugewiesen, darf während des Aufbaus von `m` mit Hilfe der Softwarefunktionen `matrix::translate()`, `matrix::rotate_x()` usw. durchaus die gewohnte Reihenfolge der Matrizenmultiplikation verwendet werden.

Zweiter Schritt:

```
void landscape :: load_heights( void )
{
  long grad = 350;
```

```
  heightfield terrain( detail_level, detail_level );
  terrain.plasma_fractal( grad );

  for( long x=0 ; x<vertex_count ; x++ )
     vs[ x ].wy = 0.05 * terrain.heights[ x ];
}
```

Die Funktion `load_heights()` wird am Ende der Funktion `landscape::load()` ausgeführt; die Klasse `landscape` ist nichts anderes als eine Version der Klasse `polyhedron`, mit der wir im 6. Kapitel die zweidimensionale Punktwolke der rotationssymmetrischen Polyeder generiert haben.

Dritter Schritt:

```
void landscape::load( long dl )
{
  detail_level = dl+1;
  vertex_count = detail_level * detail_level;

  if( (vs = new vertex[ vertex_count ]) == NULL )
    exit_error( "*vs: Fehler bei der Reservierung von \
                 Arbeitsspeicher.\n" );

  long x, y;

  for( y=0 ; y<detail_level ; y++ )
     for( x=0 ; x<detail_level ; x++ )
     {
       vs[ y * detail_level + x ].wx =  x;
       vs[ y * detail_level + x ].wy =  0;
       vs[ y * detail_level + x ].wz = -y;

       vs[ y * detail_level + x ].tx =
         x * (8.0 / (detail_level-1));

       vs[ y * detail_level + x ].ty =
         y * (8.0 / (detail_level-1));
     }
```

```
  matrix m;
  m.translate
  ( -0.5*(detail_level-1), 0, 0.5*(detail_level-1) );
  for( x=0 ; x<vertex_count ; x++ )
     vs[ x ] = m * vs[ x ];

  tx.load( "tx.bmp" );

  load_heights();
}
```

Vierter Schritt:

Die Farbkomponenten der Elemente des Arrays `landscape::vs[]` werden während der Ausführung von `landscape::load_heights()` initialisiert:

```
void landscape::load_heights( void )
{
  pixel_32 borders[ 5 ] =
  {
    pixel_32( 10, 60, 0 ), pixel_32( 80, 130, 50 ),
    pixel_32( 140, 170, 90 ), pixel_32( 160, 160, 190 ),
    pixel_32( 255, 255, 255 )
  };
  primary_color hc;  hc.load( 256, 5, borders );

  long grad = 350;

  heightfield terrain( detail_level, detail_level );
  terrain.plasma_fractal( grad );

  for( long x=0 ; x<vertex_count ; x++ )
  {
    vs[x].wy = 0.05 * terrain.heights[ x ];
    vs[x].color = hc.components[ terrain.heights[ x ] ];
  }
}
```

Die Verschiebung der Eckpunkte der Texturen in Richtung dieser Farbwerte erfolgt während der Ausführung von `landscape::draw_square()`; diese Funktion zeichnet das Landschaftsquadrat auf dem Bildschirm, dessen obere linke Ecke durch den Vertex an der Position `offset` innerhalb von `landscape::vs[]` festgelegt wird.

Die übrigen drei Vertices befinden sich, im Uhrzeigersinn angegeben, an den Positionen `(offset + 1)`, `(offset + detail_level + 1)` und `(offset+ detail_level)`. Um die Darstellung möglichst effizient zu gestalten, wird `glBegin()` mit dem Parameter `GL_TRIANGLE_STRIP` aufgerufen, anstelle des bisher üblichen `GL_POLYGON`. Die Konstante `GL_TRIANGLE_STRIP` gibt an, dass der `glBegin()`, `glEnd()`-Block die Punkte von Dreiecken enthält, die jeweils eine gemeinsame Seite besitzen:

```
void landscape :: draw_square( long offset )
{
  long pos;

  glBegin( GL_TRIANGLE_STRIP );

    pos = offset + detail_level;
    glColor3ub( vs[ pos ].color.red,
      vs[ pos ].color.green, vs[ pos ].color.blue );
    glTexCoord2d( vs[ pos ].tx, vs[ pos ].ty );
    glVertex3d( vs[pos].wx, vs[pos].wy, vs[pos].wz );

    pos = offset;
    glColor3ub( vs[ pos ].color.red,
      vs[ pos ].color.green, vs[ pos ].color.blue );
    glTexCoord2d( vs[ pos ].tx, vs[ pos ].ty );
    glVertex3d( vs[pos].wx, vs[pos].wy, vs[pos].wz );

    pos = offset + detail_level + 1;
    glColor3ub( vs[ pos ].color.red,
      vs[ pos ].color.green, vs[ pos ].color.blue );
    glTexCoord2d( vs[ pos ].tx, vs[ pos ].ty );
    glVertex3d( vs[pos].wx, vs[pos].wy, vs[pos].wz );

    pos = offset + 1;
    glColor3ub( vs[ pos ].color.red,
      vs[ pos ].color.green, vs[ pos ].color.blue );
    glTexCoord2d( vs[ pos ].tx, vs[ pos ].ty );
```

```
    glVertex3d( vs[pos].wx, vs[pos].wy, vs[pos].wz );

  glEnd();
}
```

Fünfter Schritt:

Der vollständige Quelltext des Programms `a9_5` befindet sich auf der CD im gleichnamigen Verzeichnis.