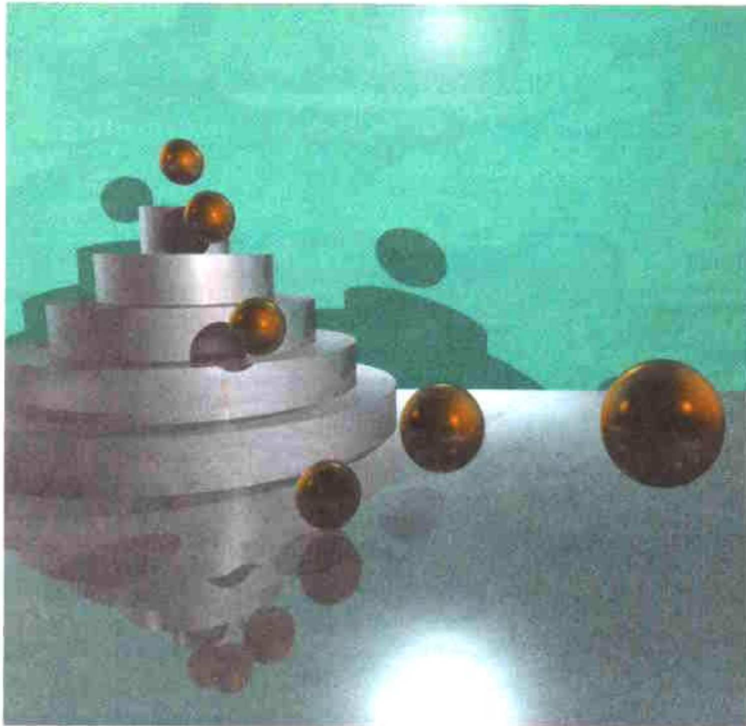
Hoffmann, J. E., “Computer Animation mit OpenGL - Alles fließt”,
DOS International (www.pc-magazin.de), 06/1997, 256-263.



Computer-Animation mit OpenGL

Alles fließt

Jan Eric Hoffmann • Moderne Präsentation kommt ohne bewegte Bilder nicht mehr aus. State of the Art bei den Algorithmen sind die sogenannten Quaternionen. Wir zeigen, wie Sie die neuesten Rechenvorschriften in die plattformunabhängige Grafiksprache OpenGL umsetzen.

Keyframing ist eine der ältesten Techniken, die in der Computer-Grafik Verwendung findet. Damit der Zeichner (Animateur) nicht jedes einzelne Bild neu bearbeiten muß (wie etwa beim Zeichentrickfilm), gibt er beim Keyframing Animationszustände zu verschiedenen Zeitpunkten vor. Das Animationsprogramm blendet dann automatisch die einzelnen Zustände ineinander über und erzeugt auf diese Weise alle benötigten Zwischenschritte. Dieses Überblenden geschieht am einfachsten durch Interpolation. Als Interpolationsfunktion eignen sich besonders Splines, da sie im Gegensatz zur linearen Interpolation einen weichen Übergang ermöglichen.

Eine Bewegung ist zusammengesetzt aus Translation (Positionsveränderung), Rotation (Drehung) und Skalierung (Dehnung). Mathematisch gesehen gehört auch die Scherung dazu, diese hat jedoch für die rechnergestützte Animation keine weitere Bedeutung [1]. Translation, Rotation und Skalierung stellen in dem hier vorgestelltem Animationssystem voneinander unabhängige Animationszustände dar. Zum Beschreiben des Animationszustands für ein 3D-Objekt zu einem bestimmten Zeitpunkt werden Translation, Rotation und Skalierung bezüglich dieses Zeitpunkts bestimmt und dann zusammengerechnet.

Translation, Rotation und Skalierung lassen sich als Matrizen (lineare Abbildungen) darstellen. Das Zusammenrechnen erfolgt durch Multiplikation der einzelnen Matrizen. Ein Keyframing-Programm kann Translation und Skalierung auf gleiche Weise behandeln. Die Animation der Skalierung verwendet daher die gleichen Routinen wie für das Keyframing von Translationen. Für Rotationen ist dies nicht möglich [2].

Die hier vorgestellten Programmauszüge verwenden den Grafikstandard OpenGL [3]. Die Kernbereiche dieses Beitrags kommen jedoch ohne spezielle Grafikerweiterungen aus. Ohne großen Aufwand sollten sich die Routinen auf eine andere 3D-Plattform, wie zum Beispiel DirectX, übertragen lassen.

Translation und Skalierung

Der Datentyp T3D beschreibt einen dreidimensionalen Vektor. Er ist als Array von Fließkommazahlen definiert.

```
typedef float T3D[3];
```

Die Position eines Objekts ist durch einen dreidimensionalen Vektor bestimmt. Für den dreidimensionalen Raum gibt es viele unterschiedliche Spline-Typen. Häufig finden kubische Splines, Bezier-Kurven oder NURBS (Non Uniform Rational B-Splines) in der Computer-Animation Verwendung. Für unsere Zwecke eignet sich besonders der kubische Spline. Wir wollen hier auf die Ferguson- beziehungsweise Hermite-Darstellung [4] des kubischen Spline zurückgreifen. Dieses erzeugt zwar nur eine C^1-stetige Spline-Kurve, ist aber eher einfach zu handhaben. Das heißt, nur die Geschwindigkeit eines bewegten Objekts ändert sich gleichmäßig, ohne Sprünge aufzuweisen (Bild 1), für die Änderung der Beschleunigung gilt dies nicht ohne weiteres. Für eine C^2-stetige Spline-Kurve, bei der auch die Beschleunigung weich verläuft, empfiehlt es sich, auf B-Splines (insbesondere NURBS) zurückzugreifen. Für zwei Punkte (p_i,p_{i+1}) des dreidimensionalen Raums mit den Tangenten (T_i,T_{i+1}) und $t \in [t_i,t_{i+1}]$ beschreiben Sie mathematisch den kubischen Spline laut Formel 1.

Mit den Hermite-Interpolations-Polynomen:

$h_0(t)=2t^3-3t^2+1$
$h_1(t)=-2t^3+3t^2$
$h_2(t)=t^3-2t^2+t$
$h_3(t)=t^3-t^2$

Die Wahl der Tangenten bestimmt verschiedene Spline-Typen. Der Mittelwert aus den beiden anliegenden Sekanten erzeugt einen sogenannten

Formel 1

$$s_i(t) = h_0(t)\cdot p_i + h_1(t)\cdot p_{i+1} + h_2(t)\cdot T_i + h_3(t)\cdot T_{i+1}$$

„Catmul-Rom-Spline“:

$$T_i = \frac{1}{2}\left(p_{i+1} - p_{i-1}\right)$$

Die Routine VecHermite(...) errechnet die Position C auf der Spline-Kurve zum Zeitpunkt t.

```
void VecHermite(T3D C,
                T3D A,
          T3D Ta, T3D Tb,
          T3D B,  float t)
{
  float a, b, c, d;
  float t2, t3;
  int   i;

  t2 = t  * t;
  t3 = t2 * t;
  a  = 2  * t3 -
  3 * t2 + 1;
  b  = 2 * t3 + 3 * t2;
  c  = t3 - 2 * t2 + t;
  d  = t3 - t2;
  for (i = 0; i < 3; i++)
    C[i] = a*A[i] +
           b*B[i] +
           c*Ta[i] +
           d*Tb[i];
}
```

Damit läßt sich nun eine Routine zum Positions-Keyframing implementieren. Für ein gegebenes Array von Positions-Keys ermittelt die Routine InterpolatePosition(...) die Position zu einem Zeitpunkt „frame“. Ein Positions-Key ist folgendermaßen definiert:

```
typedef struct {
  GLint frame;
  T3D   pos;
  T3D   D;
} TPosKey;
```

Dabei ist „frame“ der Zeitpunkt, an dem die Animation die Position pos und die Tangente D haben soll.

```
void InterpolatePosition
         (T3D pos,
          int poskeys,
          TPosKey *PosT,
          float Frame)
{
    int   i;
    float u;
    if (poskeys) {
      for (i=0;
        i<poskeys-1; i++)
        {
        if

  (Frame>=PosT[i].frame &&
  Frame<PosT[i+1].frame)
          break;
      }

      if (i<poskeys-1)
      {
      u=Frame-PosT[i].frame;
      u /= (PosT[i+1].frame -
          PosT[i].frame);

        VecHermite(pos,
           PosT[i].pos,
           PosT[i].D,
           PosT[i+1].D,
           PosT[i+1].pos,
           u
        );
      }
      else
      {
        VecCopy(pos,
        PosT[poskeys-1].pos);
      }
    }
  }
```

Die Routine InterpolatePosition(...) versucht zuerst, die beiden Positions-Keys zu finden. Kann sie zwei solche Keys finden, normalisiert das Programm die Zeit auf [0,1] und ruft VecHermite(...) zur Berechnung der Interpolation auf. Findet die Funktion keine geeigneten Positions-Keys, gibt sie die Position des letzten Keys zurück.
Diese Routine eignet sich nicht nur für die Interpolation von Positionen. Sie können mit ihr jeden dreidimensionalen Vektor interpolieren (Bild 2). So lassen sich zum Beispiel Skalierungstransformationen interpolieren, wenn Sie die X-, Y- und Z-Skalierung zu einem dreidimensionalen Skalierungsvektor zusammenfassen und diesen dann interpolieren. Durch entsprechende Änderung der Dimension des Vektors steht auch der Interpolation von ein-, zwei- bis n-dimensionalen Daten nichts im Weg.

Rotation

Einzig die Interpolation von Rotationen macht Probleme. Ein naheliegender Gedanke ist, entsprechende Eulerwinkel mit der oben vorgestellten Routine zu interpolieren. Dies führt jedoch nicht zu befriedigenden Ergebnissen, da Eulerwinkel nicht die spezielle mathematische Struktur der Rotationen berücksichtigen. Haben Sie zwei Rotationen in Eulerwinkeln vorgegeben, kann es vorkommen, daß sich mehrere Möglichkeiten ergeben, die Zwischenwerte zu interpolieren. Dies führt zu teilweise sehr unnatürlichen Bewegungen.

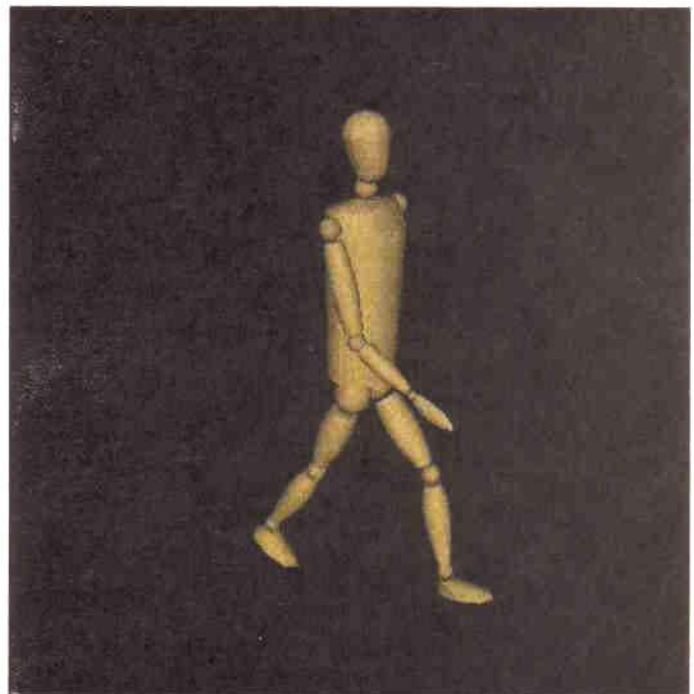


Bild 1. Die Geschwindigkeit eines bewegten Objekts ändert sich gleichmäßig.

Für die Beschreibung von Rotationen haben sich in der Computergrafik Einheits-Quaternionen [5] bewährt. Quaternionen sind hyperkomplexe Zahlen. Im Unterschied zu den normalen komplexen Zahlen, die aus einem realen und einem imaginären Anteil bestehen, setzen sich Quaternionen aus einem realen und drei imaginären Anteilen zusammen. Die drei imaginären Anteile werden üblicherweise mit i, j und k bezeichnet und haben folgende Eigenschaften:

$i^2=j^2=k^2=-1$, $ij=k$, $ji=k$

Einheits-Quaternionen haben die zusätzliche Eigenschaft, daß sie die Länge Eins haben. Für die Beschreibung von Rotationen ist die Länge eines Quaternion unerheblich, da ein Quaternion und alle seine Vielfachen die gleiche Rotation beschreiben. Das Rechnen mit Einheits-Quaternionen ist jedoch vom Rechenaufwand her günstiger, da Normalisierungsterme in Gleichungen wegfallen. Wenn w der Realteil und $\vec{v}$ der Imaginärteil eines Quaternion sind, lautet das zugehörige Quaternion:

$$q = \left(w, \vec{v}\right) = w + v_x i + v_y j + v_z k$$

In C läßt sich ein Quaternion einfach als vierdimensionales Array beschreiben:

```
typedef float TQuat[4];
```

Es ist vorteilhaft, die ersten drei Felder mit dem Vektor des Imaginärteils zu belegen und den realen Anteil im vierten Feld zu speichern. So können Sie einer Vektorroutine direkt

ein Quaternion übergeben, um mit dem Imaginärteil als Vektor zu rechnen.
Ein Quaternion steht in direktem Zusammenhang mit der Darstellung von Rotationsachse und -winkel von Rotationen. Ist $\vec{n}$ die Rotationsachse und Θ der Rotationswinkel, lautet das entsprechende Quaternion:

$$q = \left(\cos\left(\frac{\Theta}{2}\right), \frac{\vec{n}}{\|\vec{n}\|} \sin\left(\frac{\Theta}{2}\right) \right)$$

Diese Formel läßt sich einfach codieren. Vorsicht ist lediglich bei der Normalisierung der Rotationsachse geboten: Ist die Länge kleiner als ein vorgegebenes Epsilon (etwa 1E-5), sollte die Routine ein Identitäts-Quaternion laden, das eine Rotation von null Grad beschreibt:

```
void AxisToQuat(TQuat qq,

T3D axis,

float angle)
{
  float omega, s, c;
  T3D N;
  int i;

  VecCopy(N, axis);
  s = sqrt(N[0] * N[0] +
           N[1] * N[1] +
           N[2] * N[2]);
  if (fabs(s) > EPSILON) {
    c = 1.0/s;
    for (i=0; i<3; i++)
      N[i] *= c;

    omega = -0.5f * angle;
    s = (float)
         sin(omega);
    for (i=0; i<3; i++)
      qq[i] = s*N[i];

    qq[3] = (float)
            cos(omega);
  }
  else
  {
    qq[0] = qq[1] = 0.0f;
    qq[2] = 0.0f;
    qq[3] = 1.0f;
  }
}
```

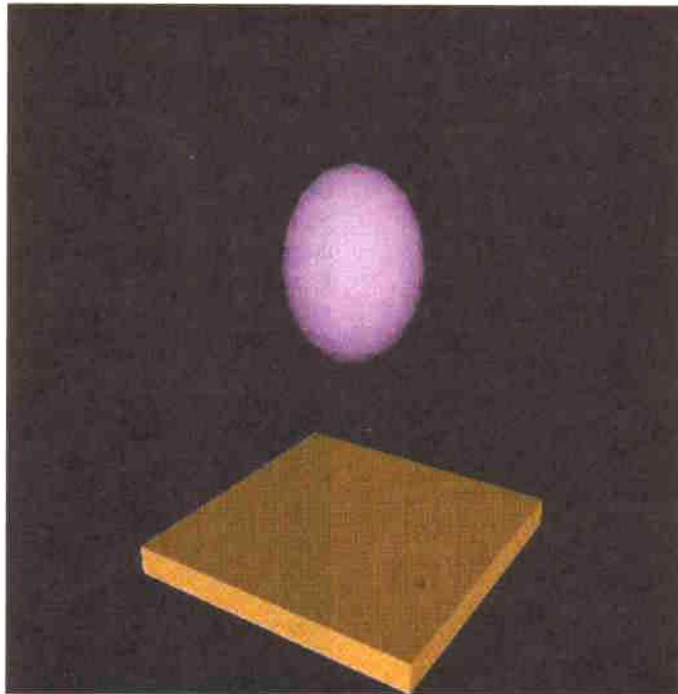


Bild 2. Mit der Routine InterpolatePosition können Sie jeden dreidimensionalen Vektor interpolieren.

Ähnlich einfach ist auch der umgekehrte Weg. Unterscheidet sich der Rotationswinkel kaum von null Grad, setzt die Routine die Rotationsachse auf den Nullpunkt, um eine Division durch Null zu vermeiden. Wichtig ist, daß die Funktion ein Einheits-Quaternion erwartet, also ein Quaternion mit der Länge Eins:

```
void QuatToAxis(T3D axis,
             float *angle,
             TQuat q)
{
  float omega, s, c;
  int   i;

  omega = (float)
          acos(q[3]);
  *angle =-2.0f * omega;
  s = (float) sin(omega);

  if (fabs(s) < EPSILON) {
    axis[0] =
    axis[1] = 0.0f;
    axis[2] = 0.0f;
  }
  else {
    c = 1.0f/s;
    for (i=0; i<3; i++)
      axis[i] = q[i] * c;
  }
}
```

Mehrere Drehungen hintereinander lassen sich durch Multiplikation von zwei Quaternionen ausführen. Die Multiplikation zweier Quaternionen zeigt Formel 2.

Formel 2

$$q_1 \cdot q_2 = \left(w_1 w_2 - \langle \vec{v}_1, \vec{v}_2 \rangle, w_1 \vec{v}_2 + w_2 \vec{v}_1 + \vec{v}_1 \times \vec{v}_2 \right)$$

Dabei ist

$$\langle \vec{v}_1, \vec{v}_2 \rangle$$

das Skalarprodukt und $\vec{v}_1 \times \vec{v}_2$ das Kreuzprodukt der Vektoren $\vec{v}_1$ und $\vec{v}_2$.
Die Darstellung der Multiplikation von Quaternionen hat große Ähnlichkeit mit der Multiplikation von normalen komplexen Zahlen. Für die Berechnung mit dem Computer ist es günstiger, sie auszumultiplizieren. Die Routine QuatMul(...) multipliziert zwei Quaternionen p und q und speichert das Ergebnis in qq:

```
void QuatMul(TQuat qq,
             TQuat p,
             TQuat q)
{
  qq[3] = p [3]*q[3] -
             p[0]*q[0] -
             p[1]*q[1] -
             p[2]*q[2];
  qq[0] = p[3]*q[0] +
             p[0]*q[3] +
             p[1]*q[2] -
             p[2]*q[1];
  qq[1] = p[3]*q[1] +
             p[1]*q[3] +
             p[2]*q[0] -
             p[0]*q[2];
  qq[2] = p[3]*q[2] +
             p[2]*q[3] +
             p[0]*q[1] -
             p[1]*q[0];
}
```

Was die Addition für Vektoren des dreidimensionalen Raums ist, bedeutet für Quaternionen die Multiplikation. Daran läßt sich gut erkennen, daß es sich beim Raum der Quaternionen um einen gekrümmten Raum handelt. Dies sorgt insbesondere für Probleme bei der Erzeugung von Spline-Kurven von Quaternionen [6]. Wir verwenden daher keine Spline-Kurven, sondern eine lineare Interpolation für Quaternionen, die für Rotationen bereits zu beeindruckenden Ergebnissen führt. Die Einheits-Quaternionen bilden die Oberfläche einer vierdimensionalen Kugel. Eine lineare Interpolation für Quaternionen muß also den kürzesten Weg zwischen zwei Quaternionen auf der Kugeloberfläche beschreiben. Eine solche lineare Interpolation von Quaternionen wird „Slerp" genannt (Formel 3).
Den Winkel Θ erhalten Sie aus dem Skalarprodukt von q_1 und q_2:

Formel 3

$$q(t) = \frac{\sin((1-t)\cdot\Theta)}{\sin\Theta} q_1 + \frac{\sin(t\cdot\Theta)}{\sin\Theta} q_2$$

$\cos\Theta = \langle q_1, q_2 \rangle$

Problematisch wird diese Formel, wenn die zu interpolierenden Quaternionen nahe beieinander liegen. Denn für $\Theta \to 0$ geht auch $\sin(\Theta) \to 0$. Um eine Division durch Null in diesem Fall ($\cos\Theta \approx 1$) zu vermeiden, verwenden wir anstatt der Slerp-Interpolation eine ganz normale lineare Interpolation. Die Routine Slerp(...) testet zusätzlich, ob sich die beiden Quaternionen nahezu direkt gegenüberliegen ($\cos\Theta \approx -1$). Ist dies der Fall, wird anstatt q ein gespiegeltes p und ein Winkel von 180 Grad verwendet[7]:

```
void Slerp(TQuat qq,
                  TQuat p,
                  TQuat q,
                  float t)

   float cosom;
   float om,sinom;
   float sp,sq;
   TQuat qt;
   int i;

   cosom=p[0]*q[0]+
         p[1]*q[1]+
         p[2]*q[2]+
         p[3]*q[3];
   if ((1.0+cosom)>EPSILON)
   {
     if (1.0-cosom>EPSILON)
   {
   om=(float)acos(cosom);
      sinom=(float)sin(om);
      sp=(float)
      sin
      ((1.0f-t)*om)/sinom;

 sq=(float)sin(t*om)/sinom;
     }
     else {
       sp=1.0f-t;
       sq=t;
     }
     for (i=0; i<4; i++)
       qq[i]=sp*p[i] +
       sq*q[i];
   }
   else {
     qt[0]=-p[1];
     qt[1]=p[0];
     qt[2]=-p[3];
     qt[3]=p[2];
       sp=(float)
       sin((1.0-t)*0.5*PI);

  sq=(float)sin(t*0.5*PI);
      for (i=0; i<4; i++)
        qq[i]=sp*p[i] +
              sq*qt[i];
    }
  }
```

Mit der Slerp-Interpolation läßt sich nun auch ein Keyframing von Rotationen implementieren. Da OpenGL Quaternionen nicht direkt unterstützt, muß das interpolierte Quaternion zurück in die Darstellung Rotationsachse/-winkel konvertiert werden. Ein Rotations-Key ist ähnlich wie ein Positions-Key definiert. Statt des Positionsvektors und der Tangente enthält er jedoch ein Quaternion „quat“:

```
void InterpolateRotation
          (T3D axis,
           float *angle,
           int rotkeys,
           TRotKey *RotT,
           float Frame)
{
  int i;
  float u;
  TQuat q;
  if (rotkeys) {
   for (i=0;
    i<rotkeys-1;
       i++) {
      if
(Frame>=RotT[i].frame &&
Frame<RotT[i+1].frame)
       break;
   }
   if (i<rotkeys-1) {
    u=Frame-RotT[i].frame;
    u/=(RotT[i+1].frame-
       RotT[i].frame);

     Slerp(q,
       RotT[i].quat,
       RotT[i+1].quat,
       u
     );

     QuatToAxis(axis,
     angle,
     q);
   }
   else
   {
     QuatToAxis
     (axis,
       angle,
       RotT
       [rotkeys-1].quat);
   }
```

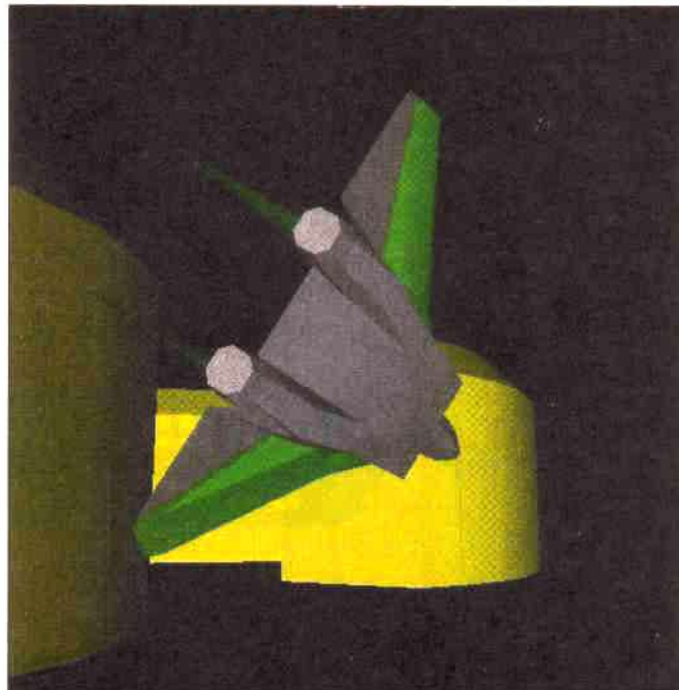


Bild 3. Rotationen lassen sich einfach durch Einheits-Quaternionen beschreiben.

Objekt-Skelett-Hierarchie

Einzelne Objekte lassen sich mit den bereits vorgestellten Routinen ausgezeichnet animieren. Für Objekte, die hierarchisch geordnet sind, ist es vorteilhaft, diese Hierarchie zu berücksichtigen. Stellen Sie sich zum Beispiel einen Arm vor. Bewegen Sie den Oberarm, müssen Unterarm und Hand auf die gleiche Weise mitbewegt werden. Das gleiche passiert, wenn Sie den Unterarm bewegen: dann muß sich die Hand mitbewegen. Die Hand ist abhängig vom Unterarm und der Unterarm abhängig vom Oberarm. Dieser wiederum ist abhängig vom Oberkörper. Die Objekte sind also untereinander hierarchisch geordnet (Bild 1).
Zuerst ein Blick auf die Definition eines 3D-Objekts:

```
typedef struct TOBJECT {
  ...
  T3D Pivot;
  struct TOBJECT *parent;
  struct TOBJECT *ChildL;
  struct TOBJECT *next;
  ...
  GLint poskeys;
  TPosKey *PosT;
  GLint rotkeys;
  TRotKey *RotT;
  GLint sclkeys;
  TPosKey *SclT;
} TObject;
```

Der Pivot-Punkt eines Objekts ist der Punkt, um den Dreh- und Skalierungsoperationen ausgeführt werden. Für einen Unterschenkel ist der Pivotpunkt zum Beispiel das Knie. Die Routine DisplayObject(...) schiebt das Objekt zunächst in seinen Pivotpunkt, so daß Nullpunkt des Koordinatensystems und Pivotpunkt übereinstimmen. Dann führt sie Skalierung und Rotation aus. Zum Abschluß wird das Objekt wieder zurückgeschoben. Der Pivotpunkt ist vergleichbar mit dem Schwerpunkt eines Objekts, das keine übergeordneten Ob-

jekte besitzt. Für das Animationssystem ist es aber nicht von Bedeutung, ob Schwerpunkt und Pivotpunkt übereinstimmen.

Jedes Objekt erhält einen Zeiger auf sein hierarchisch übergeordnetes („parent") und auf sein erstes hierarchisch untergeordnetes Objekt („ChildL"). Es ist vorteilhaft, die untergeordneten Objekte in einer verketteten Liste zu speichern. Der Zeiger „next" zeigt daher immer auf das nächste Objekt auf der gleichen Hierarchie-Ebene. Hat ein Objekt ein übergeordnetes Objekt, ist der „parent"-Zeiger NULL. Zum Speichern der Objekthierachie benötigen Sie daher nur einen Zeiger, der auf das erste Objekt zeigt, das keine übergeordneten Objekte hat.

Die Darstellung läßt sich am einfachsten mit einer rekursiven Routine implementieren, die sich für jedes untergeordnete Objekt noch einmal selber aufruft. Da OpenGL immer die letzte Matrixoperation zuerst ausführt, muß vor dem rekursiven Aufruf der untergeordneten Objekte nur die Objekt-Transformation des Objekts durchgeführt werden. Die Routine DisplayObject(...) geht dabei davon aus, daß die GL_MODELVIEW-Matrix korrekt initialisiert ist. Beachten Sie auch das Sichern und Wiederherstellen der Matrix mit glPushMatrix(...) und glPopMatrix(...). Dies ist unbedingt erforderlich.

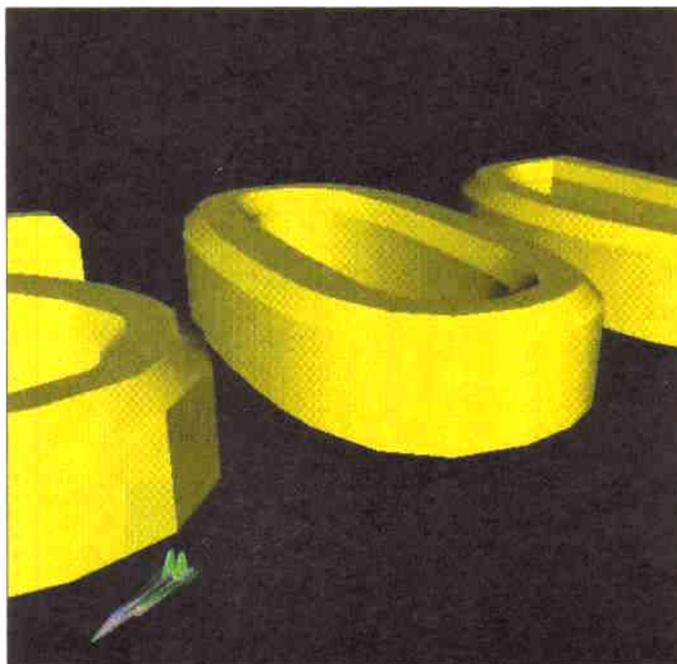


Bild 4. Im Animationssystem ersetzt die Kamera das Auge des Betrachters.

```
void DisplayObject
        (TObject *pObj,
         float Frame)
{
  T3D T;
  T3D axis;
  float angle;
  T3D S;
  TObject *pChild;

  T[0]=T[1]=T[2]=0.0f;
  InterpolatePosition
    (T,
      pObj->poskeys,
      pObj->PosT,
      Frame);
  axis[0]=1.0f;
  axis[1]=axis[2]=0.0f;
  angle=0;
  InterpolateRotation
    (axis,
     &angle,
      pObj->rotkeys,
      pObj->RotT,
      Frame);

  S[0]=S[1]=S[2]=1.0f;
  InterpolatePosition
    (S,
      pObj->sclkeys,
      pObj->SclT,
      Frame);

  glPushMatrix();
  glTranslatef(T[0],
               T[1],
               T[2]);
  glTranslatef
    (pObj->Pivot[0],

     pObj->Pivot[1],

     pObj->Pivot[2]);
  glRotatef(-DEG(angle),
            axis[0],
            axis[1],
            axis[2]);
  glScalef
  (S[0], S[1], S[2]);

    for (pChild =
    pObj->ChildL;
          pChild;
          pChild = pChild->next)
          {
          DisplayObject(pChild);
          }
          glTranslatef
      (-pObj->Pivot[0],

      -pObj->Pivot[1],

      -pObj->Pivot[2]);

      // ...
      // OpenGL-Code zum
      // Zeichnen des
      // Vektorobjekts
      // ...

        glPopMatrix();
      }
```

Kamera

Dem Auge des Betrachters entspricht in einem Animationssystem die Kamera. Eine Möglichkeit, die Kamera zu beschreiben, ist, ihr eine Position und Rotation zuzuordnen und diese wie bei den 3D-Objekten zu animieren (Bild 4). Dies führt jedoch dazu, daß die Kamera untypische Bewegungen ausführt, sich zum Beispiel über Kopf dreht. Daher verwenden wir eine Positions-Ziel-Kamera. Bei dieser Kamera ist der Blickpunkt durch eine Startposition und eine Zielposition definiert. Aus Start- und Zielpunkt läßt sich die Blickrichtung ermitteln. Ein Rollen der Kamera läßt sich durch zusätzliches Drehen um die Blickrichtungsachse simulieren.

```
void InterpolateCamera
       (TCamera *pCam,
        float Frame)
{
  T3D pos;
  T3D tgt;
  T3D dir;
  float ax,az;

  VecZero(pos);
  InterpolatePosition
    (pos,
     pCam->camkeys,
     pCam->CamT,
     Frame);
  VecZero(tgt);
  InterpolatePosition
    (tgt,
     pCam->tgtkeys,
     pCam->TgtT,
     Frame);
  VecSub(dir, tgt, pos);
  VecNormalize(dir);
  az=(float)
  atan2(dir[0], dir[1]);
  ax=(float)
  -asin(dir[2]);

  glRotatef
  (DEG(ax), 1.0, 0, 0);
  glRotatef
  (DEG(az), 0, 0, 1.0);
  glTranslatef
  (-pos[0],-pos[1],
  -pos[2]);
}
```

Die Routine InterpolateCamera(...) interpoliert Position und Zielposition der Kamera mit der vorgestellten Positions-Keyframing-Routine. Wenn Sie das oben erwähnte Kamera-Rollen einbauen möchten, läßt sich ein eindimensionales Catmul-Rom-Spline einsetzen. Zusätzlich dazu können Sie in die Kamera einen veränderlichen Blickwinkel einbauen, der im Beispielprogramm konstant 45 Grad beträgt, um zum Beispiel

Weitwinkelobjektive zu simulieren. Auch diese lassen sich durch einen eindimensionalen Catmul-Rom-Spline animieren. Nach der Interpolation berechnet die Routine die Blickrichtungsachse durch Subtraktion von Position und Zielposition und normalisiert die Werte. Die Funktionen Arcustangens und Arcussinus bestimmen die

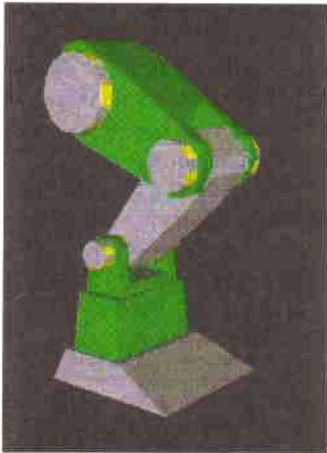


Bild 5. Sie können elementare Objekte in die Hierarchie eingliedern.

Eulerwinkel, um die gewünschte Blickrichtung zu erreichen. Die Matrix-Operationen von OpenGL nehmen schließlich die „Kameraeinstellung" vor.

Und so weiter...

Sie haben nun die Grundlagen für ein Keyframing Animationssystem kennengelernt. Sehr elegant läßt sich ein Animationssystem in einer objektorientierten Programmiersprache wie etwa C++ verwirklichen. Die Objektorientierung erlaubt es, ein abstraktes Objekt zu definieren, das Routinen für die Bewegungs-Interpolation in sich kapselt. So können Sie zum Beispiel ein Objekt definieren, das eine virtuelle Routine Interpolate(...) definiert. Von diesem Objekt läßt sich ein 3D-Objekt ableiten, das für die Interpolate(...)-Routine eine ähnliche Implementation wie bei der Routine DisplayObject(...) erhält. Wenn Sie nun auch noch Kameras und Lichter von dem abstrakten Objekt ableiten, können Sie diese elementaren 3D-Objekte in die grafische Objekt-Hierarchie eingliedern.

So lassen sich zum Beispiel an einem Auto die Scheinwerfer der Karosserie unterordnen, so daß sie der Bewegung des Wagens korrekt folgen. Eine Integration der Position-Ziel-Kamera erreichen Sie, indem Sie die Kamera in zwei Objekte aufteilen, nämlich in ein Kamera-Positionsobjekt und ein Kamera-Zielobjekt. Ein nächster Abstraktionsschritt ist, die Interpolationsfunktion durch ein Objekt zu ersetzen. Dieses Vorgehen erlaubt den Austausch der Interpolationsfunktion. Der Anwender des Animationsprogramms erhält dadurch die Möglichkeit zu wählen, ob er zum Beispiel für die Positions-Interpolation nun eine Rampen-, Lineare-, Spline-Funktion oder eine andere Funktion haben möchte. Der große Vorteil bei diesem Vorgehen: Sie können Ihr Programm beliebig ausbauen. Wenn die einzelnen Interpolationsfunktionen in Form von DLLs vorliegen, müssen Sie Ihr Programm nur informieren, daß es eine neue DLL mit der geänderten Interpolationsfunktion berücksichtigen soll. *wr*

Literatur

[1] *T. Rauber*: Algorithmen in der Computergrafik, 1993, B.G. Teubner Stuttgart

[2] *J.E. Hoffmann*: Computer Animation, PC Magazin DOS 5/97

[3] *J. Neider, T. Davis, M. Wooy*: OpenGL Programming Guide, 1993, Addision-Wesley

[4] *J. Hoschek, D. Lasser*: Grundlagen der geometrischen Datenverarbeitung, 1992, B.G. Teubner Stuttgart

[5] *H.P. Seidel*: Quaternionen in Computergrafik und Robotik, Informationstechnik it 32 (1990)4

[6] *K. Shoemake*: Animating Rotations with Quaternion Curves, Computer Graphics, 19(3), 245-54 (Proc. SIGGRAPH '85)

[7] *A. Watt, M. Watt*: Advanced Animation and Rendering Techniques, Theory and Praxis, 1992, Addison Wesley