Chapter 13

ANIMATION

Animation introduces time-varying features into computer graphics, most importantly the motion of objects and the camera. Theoretically, all the parameters defining a scene in the virtual world model can be functions of time, including color, size, shape, optical coefficients, etc., but these are rarely animated, thus we shall mainly concentrate on motion and camera animation. In order to illustrate motion and other time-varying phenomena, animation systems generate not a single image of a virtual world, but a sequence of them, where each image represents a different point of time. The images are shown one by one on the screen allowing a short time for the user to have a look at each one before the next is displayed.

Supposing that the objects are defined in their respective local coordinate system, the position and orientation of the particular object is given by its modeling transformation. Recall that this modeling transformation places the objects in the global world coordinate system determining the relative position and orientation from other objects and the camera.

The camera parameters, on the other hand, which include the position and orientation of the 3D window and the relative location of the camera, are given in the global coordinate system thus defining the viewing transformation which takes us from the world to the screen coordinate system.

Both transformations can be characterized by 4×4 homogeneous matrices. Let the time-varying modeling transformation of object o be $\mathbf{T}_{\mathbf{M},\mathbf{o}}(t)$ and the viewing transformation be $\mathbf{T}_{\mathbf{V}}(t)$.

A simplistic algorithm of the generation of an animation sequence, assuming a built-in timer, is:

```
Initialize Timer( t_start );
do
    t = Read Timer;
    for each object o do Set modeling transformation: T_M,o = T_M,o(t);
    Set viewing transformation: T_V = T_V(t);
    Generate Image;
while t < t_end;
```

In order to provide the effect of continuous motion, a new static image should be generated at least every 60 msec. If the computer is capable of producing the sequence at such a speed, we call this **real-time animation**, since now the timer can provide real time values. With less powerful computers we are still able to generate continuous looking sequences by storing the computed image sequence on mass storage, such as a video recorder, and replaying them later. This technique, called **non-real-time animation**, requires the calculation of subsequent images in uniformly distributed time samples. The time gap between the samples has to exceed the load time of the image from the mass storage, and should meet the requirements of continuous motion as well. The general sequence of this type of animation is:

```
t = t_start;                                  // preprocessing phase: recording
do
    for each object o do Set modeling transformation: T_M,o = T_M,o(t);
    Set viewing transformation: T_V = T_V(t);
    Generate Image;
    Store Image;
    t += Δt;
while t < t_end ;
Initialize Timer( t_start) ;                  // animation phase: replay
do
    t = Read Timer;
    Load next image;
    t += Δt;
    while (t > Read Timer) Wait;
while t < t_end;
```

Note that the only responsibility of the animation phase is the loading and the display of the subsequent images at each time interval Δt. The simplest way to do this is to use a commercial VCR and television set, having recorded the images computed for arbitrary time on a video tape frame-by-frame in analog signal form. In this way computers are used only for the preprocessing phase, the real-time display of the animation sequence is generated by other equipments developed for this purpose.

As in traditional computer graphics, the objective of animation is to generate realistic motion. Motion is realistic if it is similar to what observers are used to in their everyday lives. The motion of natural objects obeys the laws of physics, specifically, **Newton's law** stating that the acceleration of masses is proportional to the resultant driving force. Let a point of object mass have positional vector $\vec{r}(t)$ at time t, and assume that the resultant driving force on this mass is $\vec{D}$. The position vector can be expressed by the modeling transformation and the position in the local coordinates ($\vec{r}_L$):

$$\vec{r}(t) = \vec{r}_L \cdot \mathbf{T_M}(t). \tag{13.1}$$

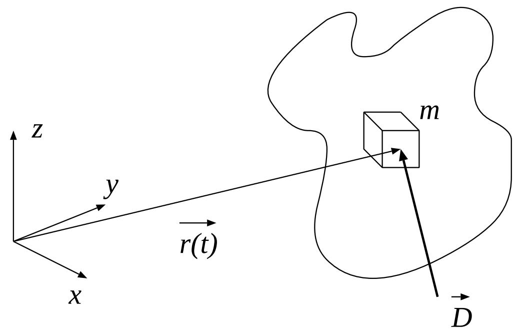

Figure 13.1: Dynamics of a single point of mass in an object

Newton's law expresses the second derivative (acceleration) of $\vec{r}(t)$ using the driving force $\vec{D}$ and the mass m:

$$\frac{\vec{D}}{m} = \frac{d^2\vec{r}(t)}{dt^2} = \vec{r}_L \cdot \frac{d^2\mathbf{T_M}(t)}{dt^2}. \tag{13.2}$$

Since there are only finite forces in nature, the second derivative of all elements of the transformation matrix must be finite. More precisely, the second derivative has to be continuous, since mechanical forces cannot change abruptly, because they act on some elastic mechanism, making $\vec{D}(t)$ continuous. To summarize, the illusion of realistic motion requires the elements of the transformation matrices to have finite and continuous second derivatives. Functions meeting these requirements belong to the C^2 family (C stands for parametric continuity, and superscript 2 denotes that the second derivatives are regarded). The C^2 property implies that the function is also of type C^1 and C^0.

The crucial problem of animation is the definition of the appropriate matrices $\mathbf{T_M}(t)$ and $\mathbf{T_V}(t)$ to reflect user intention and also to give the illusion of realistic motion. This task is called **motion control**.

To allow maximum flexibility, interpolation and approximation techniques applied in the design of free form curves are recommended here. The designer of the motion defines the position and the orientation of the objects just at a few knot points of time, and the computer interpolates or approximates a function depending on these knot points taking into account the requirements of realistic motion. The interpolated function is then sampled in points required by the animation loop.

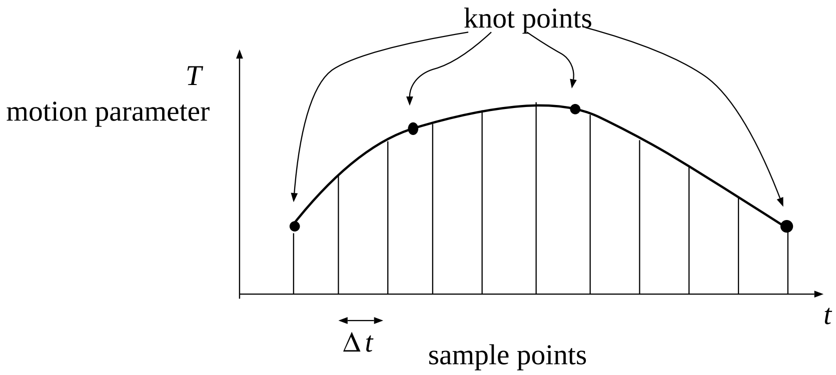

Figure 13.2: Motion control by interpolation

13.1 Interpolation of position-orientation matrices

As we know, arbitrary position and orientation can be defined by a matrix of the following form:

$$\begin{bmatrix} & 0 \\ \mathbf{A}_{3\times 3} & 0 \\ & 0 \\ \mathbf{q}^{\mathbf{T}} & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ q_x & q_y & q_z & 1 \end{bmatrix}. \tag{13.3}$$

Vector $\mathbf{q}^{\mathbf{T}}$ sets the position and $\mathbf{A}_{3\times 3}$ is responsible for defining the orientation of the object. The elements of $\mathbf{q}^{\mathbf{T}}$ can be controlled independently, adjusting the x, y and z coordinates of the position. Matrix elements $a_{11}, \ldots a_{33}$, however, are not independent, since the degree of freedom in orientation is 3, not 9, the number of elements in the matrix. In fact, a matrix representing a valid orientation must not change the size and the shape of the object, thus requiring the row vectors of $\mathbf{A}$ to be unit vectors forming a perpendicular triple. Matrices having this property are called **orthonormal**.

Concerning the interpolation, the elements of the position vector can be interpolated independently, but independent interpolation is not permitted for the elements of the orientation matrix, since the interpolated elements would make non-orthonormal matrices. A possible solution to this problem is to interpolate in the space of the roll/pitch/yaw (α, β, γ) angles (see section 5.1.1), since they form a basis in the space of the orientations, that is, any roll-pitch-yaw triple represents an orientation, and all orientations can be expressed in this way. Consequently, the time functions describing the motion are:

$$\mathbf{p}(t) = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)] \tag{13.4}$$

($\mathbf{p}(t)$ is called parameter vector).

In image generation the homogeneous matrix form of transformation is needed, and thus, having calculated the position value and orientation angles, the transformation matrix has to be expressed.

Using the definitions of the roll, pitch and yaw angles:

$$\mathbf{A} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & \sin\gamma \\ 0 & -\sin\gamma & \cos\gamma \end{bmatrix}. \tag{13.5}$$

The position vector is obviously:

$$\mathbf{q}^{\mathbf{T}} = [x, y, z]. \tag{13.6}$$

We concluded that in order to generate realistic motion, **T** should have continuous second derivatives. The interpolation is executed, however, in the space of the parameter vectors, meaning that this requirement must be replaced by another one concerning the position values and orientation angles.

The modeling transformation depends on time indirectly, through the parameter vector:

$$\mathbf{T} = \mathbf{T}(\mathbf{p}(t)). \tag{13.7}$$

Expressing the second derivative of a matrix element T_{ij}:

$$\frac{dT_{ij}}{dt} = grad_{\mathbf{p}} T_{ij} \cdot \dot{\mathbf{p}}, \tag{13.8}$$

$$\frac{d^2 T_{ij}}{dt^2} = (\frac{d}{dt}\mathrm{grad}_{\mathbf{p}} T_{ij}) \cdot \dot{\mathbf{p}} + \mathrm{grad}_{\mathbf{p}} T_{ij} \cdot \ddot{\mathbf{p}} = h(\mathbf{p}, \dot{\mathbf{p}}) + H(\mathbf{p}) \cdot \ddot{\mathbf{p}}. \tag{13.9}$$

In order to guarantee that $d^2 T_{ij}/dt^2$ is finite and continuous, $\ddot{\mathbf{p}}$ should also be finite and continuous. This means that the interpolation method used for the elements of the parameter vector has to generate functions which have continuous second derivatives, or in other words, has to provide C^2 continuity.

There are several alternative interpolation methods which satisfy the above criterion. We shall consider a very popular method which is based on cubic B-splines.

13.2 Interpolation of the camera parameters

Interpolation along the path of the camera is a little bit more difficult, since a complete camera setting contains more parameters than a position and orientation. Recall that the setting has been defined by the following independent values:

1. $v\vec{r}p$, view reference point, defining the center of the window,
2. $v\vec{p}n$, view plane normal, concerning the normal of the plane of the window,
3. $v\vec{u}p$, view up vector, showing the directions of the edges of the window,
4. $wheight, wwidth$, horizontal and vertical sizes of the window,
5. $e\vec{y}e$, the location of the camera in the u, v, w coordinate system fixed to the center of the window,
6. fp, bp, front and back clipping plane,
7. Type of projection.

Theoretically all of these parameters can be interpolated, except for the type of projection. The position of clipping planes, however, is not often varied with time, since clipping planes do not correspond to any natural phenomena, but are used to avoid overflows in the computations and to simplify the projection.

Some of the above parameters are not completely independent. Vectors $v\vec{p}n$ and $v\vec{u}p$ ought to be perpendicular unit vectors. Fortunately, the algorithm generating the viewing transformation matrix $\mathbf{T_V}$ takes care of these requirements, and should the two vectors not be perpendicular or of unit length, it adjusts them. Consequently, an appropriate space of independently adjustable parameters is:

$$\mathbf{p_{cam}}(t) = [v\vec{r}p, v\vec{p}n, v\vec{u}p, wheight, wwidth, e\vec{y}e, fp, bp]. \tag{13.10}$$

As has been discussed in chapter 5 (on transformations, clipping and projection), these parameters define a viewing transformation matrix $\mathbf{T_V}$ if the following conditions hold:

$$\vec{vpn} \times \vec{vup} \neq 0, \quad wheight \geq 0, \quad wwidth \geq 0, \quad eye_w < 0, \quad eye_w < fp < bp. \tag{13.11}$$

This means that the interpolation method has to take account not only of the existence of continuous derivatives of these parameters, but also of the requirements of the above inequalities for any possible point of time. In practical cases, the above conditions are checked in the knot points (keyframes) only, and then animation is attempted. Should it be that the path fails to provide these conditions, the animation system will then require the designer to modify the keyframes accordingly.

13.3 Motion design

The design of the animation sequence starts by specifying the knot points of the interpolation. The designer sets several time points, say $t_1, t_2 \ldots t_n$, and defines the position and orientation of objects and the camera in these points of time. This information could be expressed in the form of the parameter vector, or of the transformation matrix. In the latter case, the parameter vector should be derived from the transformation matrix for the interpolation. This task, called the **inverse geometric problem**, involves the solution of several trigonometric equations and is quite straightforward due to the formulation of the orientation matrix based on the roll, pitch and yaw angles.

Arranging the objects in t_i, we define a knot point for a parameter vector $\mathbf{p_o}(t_i)$ for each object o. These knot points will be used to interpolate a C^2 function (e.g. a B-spline) for each parameter of each object, completing the design phase.

In the animation phase, the parameter functions are sampled in the actual time instant, and the respective transformation matrices are set. Then the image is generated.

These steps are summarized in the following algorithm:

```
Define the time of knot points: t_1, ..., t_n;                    // design phase
for each knot point k do
    for each object o do
        Arrange object o:
            p_o(t_k) = [x(t_k), y(t_k), z(t_k), α(t_k), β(t_k), γ(t_k)]_o;
    endfor
    Set camera parameters: p_cam(t_k);
endfor
for each object o do
    Interpolate a C^2 function for:
            p_o(t) = [x(t), y(t), z(t), α(t), β(t), γ(t)]_o;
endfor
Interpolate a C^2 function for: p_cam(t);

Initialize Timer(t_start);                                         // animation phase
do
    t = Read Timer;
    for each object o do
        Sample parameters of object o:
            p_o = [x(t), y(t), z(t), α(t), β(t), γ(t)]_o;
        T_M,o = T_M,o(p_o);
    endfor
    Sample parameters of camera: p_cam = p_cam(t);
    T_V = T_V(p_cam);
    Generate Image;
while t < t_end ;
```

This approach has several disadvantages. Suppose that having designed and animated a sequence, we are not satisfied with the result, because we find that a particular part of the film is too slow, and we want to speed it up. The only thing we can do is to re-start the motion design from scratch and re-define all the knot points. This seems unreasonable, since it was not our aim to change the path of the objects, but only to modify the **kinematics** of the motion. Unfortunately, in the above approach both the geometry of the trajectories and the kinematics (that is the speed and acceleration along the trajectories) are specified by the same transformation matrices. That is why this approach does not allow for simple kinematic modification.

This problem is not a new one for actors and actresses in theaters, since a performance is very similar to an animation sequence, where the objects are the actors themselves. Assume a performance were directed in the same fashion as the above algorithm. The actors need to know the exact time when they are supposed to come on the stage. What would happen if the schedule were slightly modified, because of a small delay in the first part of the performance? Every single actor would have to be given a new schedule. That would be a nightmare, would it not. Fortunately, theaters do not work that way. Dramas are broken down into scenes. Actors are aware of the scene when they have to come on, not the time, and there is a special person, called a stage manager, who keeps an eye on the performance and informs the actors when they are due on (all of them at once). If there is a delay, or the timing has to be modified, only the stage manager's schedule has to change, the actors' schedules are unaffected. The geometry of the trajectory (movement of actors) and the kinematics (timing) has been successfully separated.

The very same approach can be applied to computer animation as well. Now the sequence is broken down into **frames** (this is the analogy of scenes), and the geometry of the trajectories is defined in terms of frames ($\mathcal{F}$), not in terms of time. The knot points of frames are called **keyframes**, and conveniently the first keyframe defines the arrangement at $\mathcal{F} = 1$, the second at $\mathcal{F} = 2$ etc. The kinematics (stage manager) is introduced to the system by defining the sequence of frames in terms of time, resulting in a function $\mathcal{F}(t)$.

Concerning the animation phase, in order to generate an image in time t, first $\mathcal{F}(t)$ is evaluated, then the result is used to calculate $\mathbf{T}(\mathbf{p}(\mathcal{F}))$ matrices. Tasks, such as modifying the timing of the sequence, or even reversing the whole animation, can be accomplished by the proper modification of the frame function $\mathcal{F}(t)$ without affecting the transformation matrices.

Now the transformation matrices are defined indirectly, through $\mathcal{F}$. Thus special attention is needed to guarantee the continuity of second derivatives

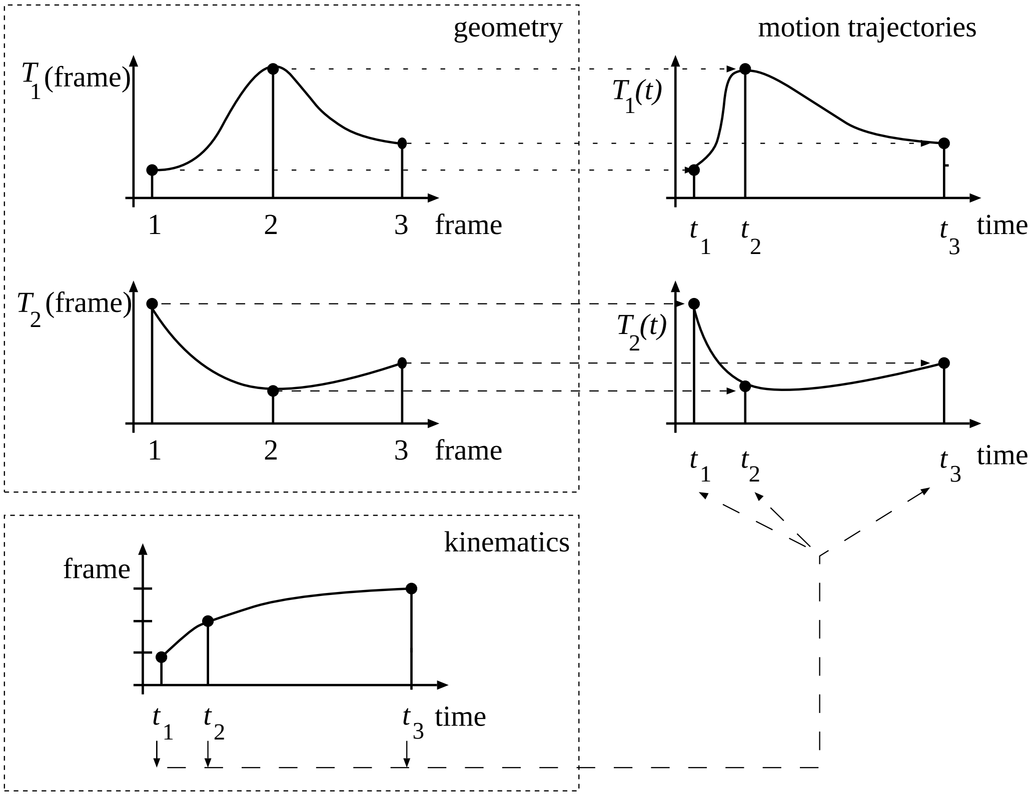

Figure 13.3: Keyframe animation

of the complete function. Expressing the second derivatives of T_{ij}, we get:

$$\frac{d^2T_{ij}}{dt^2} = \frac{d^2T_{ij}}{d\mathcal{F}^2} \cdot (\dot{\mathcal{F}})^2 + \frac{dT_{ij}}{d\mathcal{F}} \cdot \ddot{\mathcal{F}}. \tag{13.12}$$

A sufficient condition for the continuous second derivatives of $T_{ij}(t)$ is that both $\mathcal{F}(t)$ and $T_{ij}(\mathcal{F})$ are C^2 functions. The latter condition requires that the parameter vector $\mathbf{p}$ is a C^2 type function of the frame variable $\mathcal{F}$.

The concept of **keyframe animation** is summarized in the following algorithm:

// Design of the geometry
for each keyframe kf **do**
  for each object o **do**
    Arrange object o:
      $\mathbf{p_o}(kf) = [x(kf), y(kf), z(kf), \alpha(kf), \beta(kf), \gamma(kf)]_o$;
  endfor
  Set camera parameters: $\mathbf{p_{cam}}(kf)$;
endfor
for each object o **do**
  Interpolate a C^2 function for:
  $\mathbf{p_o}(f) = [x(f), y(f), z(f), \alpha(f), \beta(f), \gamma(f)]_o$;
endfor
Interpolate a C^2 function for: $\mathbf{p_{cam}}(f)$;

// Design of the kinematics
for each keyframe kf **do** Define t_{kf}, when $\mathcal{F}(t_{kf}) = kf$;
Interpolate a C^2 function for $\mathcal{F}(t)$;

// Animation phase
Initialize Timer(t_{start});
do
  t = Read Timer;
  $f = \mathcal{F}(t)$;
  for each object o **do**
    Sample parameters of object o:
      $\mathbf{p_o} = [x(f), y(f), z(f), \alpha(f), \beta(f), \gamma(f)]_o$;
    $\mathbf{T_{M,o}} = \mathbf{T_{M,o}}(\mathbf{p_o})$;
  endfor
  Sample parameters of camera: $\mathbf{p_{cam}}(f)$;
  $\mathbf{T_V} = \mathbf{T_V}(\mathbf{p_{cam}})$;
  Generate Image;
while $t < t_{\text{end}}$;

13.4 Parameter trajectory calculation

We concluded in the previous sections that the modeling ($\mathbf{T_M}$) and viewing ($\mathbf{T_V}$) transformation matrices should be controlled or animated via parameter vectors. Additionally, the elements of these parameter vectors and the frame function for keyframe animation must be C^2 functions — that is, they must have continuous second derivatives — to satisfy Newton's law which is essential if the motion is to be realistic. In fact, the second derivatives of these parameters are proportional to the force components, which must be continuous in real-life situations, and which in turn require the parameters to have this C^2 property.

We also stated that in order to allow for easy and flexible trajectory design, the designer of the motion defines the position and the orientation of the objects — that is indirectly the values of the motion parameters — in just a few knot points of time, and lets the computer interpolate a function relying on these knot points taking into account the requirements of C^2 continuity. The interpolated function is then sampled in points required by the animation loop.

This section focuses on this **interpolation** process which can be formulated for a single parameter as follows:

Suppose that points $[p_0, p_1, \ldots, p_n]$ are given with their respective time values $[t_0, t_1, \ldots, t_n]$, and we must find a function $p(t)$ that satisfies:

$$p(t_i) = p_i, \qquad i = 0, \ldots, n, \tag{13.13}$$

and that the second derivative of $p(t)$ is continuous (C^2 type). Function $p(t)$ must be easily represented and computed numerically, thus it is usually selected from the family of polynomials or the piecewise combination of polynomials. Whatever family is used, the number of its independently controllable parameters, called the degree of freedom, must be at least n, to allow the function to satisfy n number of independent equations.

Function $p(t)$ that satisfies equation 13.13 is called the **interpolation function** or curve of **knot** or **control points** $[t_0, p_0; t_1, p_1; \ldots; t_n, p_n]$. Interpolation functions are thus required to pass through its knot points. In many cases this is not an essential requirement, however, and it seems advantageous that a function should just follow the general directions provided by the knot points — that is, that it should only approximate the knot points — if by thus eliminating the "passing through" constraint we improve other, more important, properties of the generated function. This latter function type is called the **approximation function**. This section considers interpolation functions first, then discusses the possibilities of approximation functions in computer animation.

A possible approach to interpolation can take a single, minimal order polynomial which satisfies the above criterion. The degree of the polynomial should be at least $n-1$ to have n independent coefficients, thus the interpolation polynomial is:

$$p(t) = \sum_{i=0}^{n-1} a_i \cdot t^i. \tag{13.14}$$

The method that makes use of this approach is called **Lagrange interpolation**. It can be easily shown that the polynomial which is incident to the given knot points can be expressed as:

$$p(t) = \sum_{i=0}^{n} p_i \cdot L_i^{(n)}(t), \tag{13.15}$$

where $L_i^{(n)}(t)$, called the Lagrange base polynomial, is:

$$L_i^{(n)}(t) = \frac{\prod\limits_{\substack{j=0\\ j\neq i}}^{n} (t - t_j)}{\prod\limits_{\substack{j=0\\ j\neq i}}^{n} (t_i - t_j)}. \tag{13.16}$$

Equation 13.15 gives an interesting geometric interpretation to this schema. The Lagrange base polynomials are in fact **weight functions** which give a certain weight to the knot points in the linear combination defining $p(t)$. Thus, the value of $p(t)$ comes from the time-varying blend of the control points. The roots of blending functions $L_i^{(n)}(t)$ are $t_0, \ldots, t_{i-1}, t_{i+1}, \ldots t_n$, and the function gives positive or negative values in the subsequent ranges $[t_j, t_{j+1}]$, resulting in an oscillating shape. Due to the oscillating blending functions, the interpolated polynomial also tends to oscillate between even reasonably arranged knot points, thus the motion exhibits wild wiggles that are not inherent in the definition data. The greater the number of knot points, the more noticeable the oscillations become since the degree of the base polynomials is one less that the number of knot points. Thus, although single polynomial based interpolation meets the requirement of continuous second derivatives and easy definition and calculation, it is acceptable for animation only if the degree, that is the number of knot points, is small.

A possible and promising direction of the refinement of the polynomial interpolation is the application of several polynomials of low degree instead of a single high-degree polynomial. This means that in order to interpolate through knot points $[t_0, p_0; t_1, p_1; \ldots; t_n, p_n]$, a different polynomial $p_i(t)$ is found for each range $[t_i, t_{i+1}]$ between the subsequent knot points. The complete interpolated function $p(t)$ will be the composition of the segment polynomials responsible for defining it in the different $[t_i, t_{i+1}]$ intervals, that is:

$$p(t) = \begin{cases} p_0(t) \text{ if } t_0 \le t < t_1 \\ \vdots \\ p_i(t) \text{ if } t_i \le t < t_{i+1} \\ \vdots \\ p_{n-1}(t) \text{ if } t_{n-1} \le t \le t_n \end{cases} \tag{13.17}$$

In order to guarantee that $p(t)$ is a C^2 function, the segments must be carefully connected to provide C^2 continuity at the joints. Since this may mean different second derivatives required at the two endpoints of the segment, the polynomial must not have a constant second derivative, that is, at least cubic (3-degree) polynomials should be used for these segments. A composite function of different segments connected together to guarantee C^2 continuity is called a **spline** [RA89]. The simplest, but practically the most important, spline consists of 3-degree polynomials, and is therefore called the **cubic spline**. A cubic spline segment valid in $[t_i, t_{i+1}]$ can be written as:

$$p_i(t) = a_3 \cdot (t - t_i)^3 + a_2 \cdot (t - t_i)^2 + a_1 \cdot (t - t_i) + a_0. \tag{13.18}$$

The coefficients (a_3, a_2, a_1, a_0) define the function unambiguously, but they cannot be given a direct geometrical interpretation. Thus an alternative representation is selected, which defines the values and the derivatives of the segment at the two endpoints, forming a quadruple $(p_i, p_{i+1}, p'_i, p'_{i+1})$. The correspondence between the coefficients of the polynomial can be established by calculating the values and the derivatives of equation 13.18.

Using the simplifying notation $T_i = t_{i+1} - t_i$, we get:

$$\begin{aligned} p_i &= p_i(t_i) = a_0, \\ p_{i+1} &= p_i(t_{i+1}) = a_3 \cdot T_i^3 + a_2 \cdot T_i^2 + a_1 \cdot T_i + a_0, \\ p_i' &= p_i'(t_i) = a_1, \\ p_{i+1}' &= p_i'(t_{i+1}) = 3a_3 \cdot T_i^2 + 2a_2 \cdot T_i + a_1. \end{aligned} \tag{13.19}$$

These equations can be used to express $p_i(t)$ by the endpoint values and derivatives, proving that this is also an unambiguous representation:

$$p_i(t) = [2(p_i - p_{i+1}) + (p_i' + p_{i+1}')T_i] \cdot (\frac{t - t_i}{T_i})^3 +$$
$$[3(p_{i+1} - p_i) - (2p_i' + p_{i+1}')T_i] \cdot (\frac{t - t_i}{T_i})^2 + p_i' \cdot (t - t_i) + p_i. \tag{13.20}$$

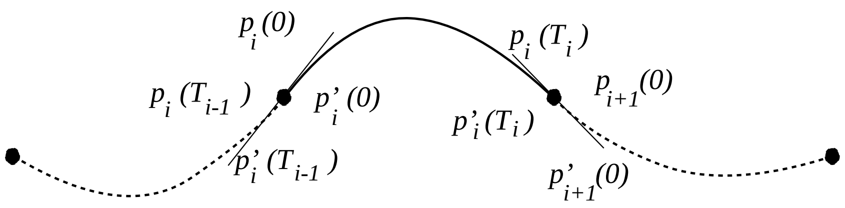

Figure 13.4: Cubic B-spline interpolation

The continuous connection of consecutive cubic segments expressed in this way is easy because C^0 and C^1 continuity is automatically provided if the value and the derivative of the endpoint of one segment is used as the starting point of the next segment. Only the continuity of the second derivative ($p''(t)$) must somehow be obtained. The first two elements in the quadruple $(p_i, p_{i+1}, p_i', p_{i+1}')$ are the knot points which are known before the calculation. The derivatives at these points, however, are usually not available, thus they must be determined from the requirement of C^2 continuous connection. Expressing the second derivative of the function defined by equation 13.20 for any k and $k+1$, and requiring $p_k''(t_{k+1}) = p_{k+1}''(t_{k+1})$, we get:

$$T_{k+1}p_k' + 2(T_k + T_{k+1})p_{k+1}' + T_k p_{k+2}' = 3[\frac{T_k}{T_{k+1}}(p_{k+2} - p_{k+1}) + \frac{T_{k+1}}{T_k}(p_{k+1} - p_k)]. \tag{13.21}$$

Applying this equation for each joint ($k = 1, 2, \ldots, n-2$) of the composite curve yields $n-2$ equations, which is less than the number of unknown derivatives (n). By specifying the derivatives — that is the "speed" — at the endpoints of the composite curve by assigning $p_0'(t_0) = v_{\text{start}}$ and $p_{n-1}'(t_n) = v_{\text{end}}$ (we usually expect objects to be at a standstill before their movement and to stop after accomplishing it, which requires $v_{\text{start}}, v_{\text{end}} = 0$), however, the problem becomes determinant. The linear equation in matrix form is:

$$\begin{bmatrix} 1 & 0 & \ldots & & & & & \\ T_1 & 2(T_0+T_1) & T_0 & 0 & \ldots & & & \\ 0 & T_2 & 2(T_1+T_2) & T_1 & 0 & & \ldots & \\ \vdots & & & & & & & \\ & & \ldots & 0 & T_{n-1} & 2(T_{n-1}+T_{n-2}) & T_{n-2} \\ & & & & \ldots & 0 & 1 \end{bmatrix} \begin{bmatrix} p_0' \\ p_1' \\ p_2' \\ \vdots \\ p_{n-1}' \\ p_n' \end{bmatrix}$$

$$= \begin{bmatrix} v_{\text{start}} \\ 3[T_0/T_1(p_2-p_1) + T_1/T_0(p_1-p_0)] \\ 3[T_1/T_2(p_3-p_2) + T_2/T_1(p_2-p_1)] \\ \vdots \\ 3[T_{n-2}/T_{n-1}(p_n-p_{n-1}) + T_{n-1}/T_{n-2}(p_{n-1}-p_{n-2})] \\ v_{\text{end}} \end{bmatrix}. \qquad (13.22)$$

By solving this linear equation, the unknown derivatives $[p_0', \ldots, p_n']$ can be determined, which in turn can be substituted into equation 13.20 to define the segments and consequently the composite function $p(t)$.

Cubic spline interpolation produces a C^2 curve from piecewise 3-degree polynomials, thus neither the complexity of the calculations nor the tendency to oscillate increases as the number of knot points increases. The result is a smooth curve exhibiting no variations that are not inherent in the series of knot points, and therefore it can provide realistic animation sequences. This method, however, still has a drawback which appears during the design phase, namely the lack of **local control**. When the animator desires to change a small part of the animation sequence, he will modify a knot point nearest to the timing parameter of the given part. The modification of a single point (either its value or its derivative), however, affects the whole trajectory, since in order to guarantee second order continuity, the derivatives at the knot points must be recalculated by solving equation 13.22. This may lead to unwanted changes in a part of the trajectory far from the modified knot point, which makes the design process difficult to execute in cases where very fine control is needed. This is why we prefer methods which have this "local control" property, where the modification of a knot point alters only a limited part of the function.

Recall that the representation of cubic polynomials was changed from the set of coefficients to the values and the derivatives at the endpoints when the cubic spline interpolation was introduced. This representation change had a significant advantage in that by forcing two consecutive segments to share two parameters from the four (namely the value and derivative at one endpoint), C^0 and C^1 continuity was automatically guaranteed, and only the continuity of the second derivative had to be taken care of by additional equations. We might ask whether there is another representation of cubic segments which guarantees even C^2 continuity by simply sharing 3 control values from the possible four. There is, namely, the **cubic B-spline**.

The cubic B-spline is a member of a more general family of **k-order B-splines** which are based on a set of k-order (degree $k-1$) blending functions that can be used to define a $p(t)$ function by the linear combination of its knot points $[t_0, p_0; t_1, p_1; \ldots; t_n, p_n]$:

$$p(t) = \sum_{i=0}^{n} p_i \cdot N_{i,k}(t), \qquad k = 2, \ldots n, \tag{13.23}$$

where the blending functions $N_{i,k}$ are usually defined by the Cox-deBoor recursion formulae:

$$N_{i,1}(t) = \begin{cases} 1 \text{ if } t_i \leq t < t_{i+1} \\ \\ 0 \text{ otherwise} \end{cases} \tag{13.24}$$

$$N_{i,k}(t) = \frac{(t - t_i)N_{i,k-1}(t)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - t)N_{i+1,k-1}(t)}{t_{i+k} - t_{i+1}}, \qquad \text{if } k > 1. \tag{13.25}$$

The construction of these blending functions can be interpreted geometrically. At each level of the recursion two subsequent blending functions are taken and they are blended together by linear weighting (see figure 13.5).

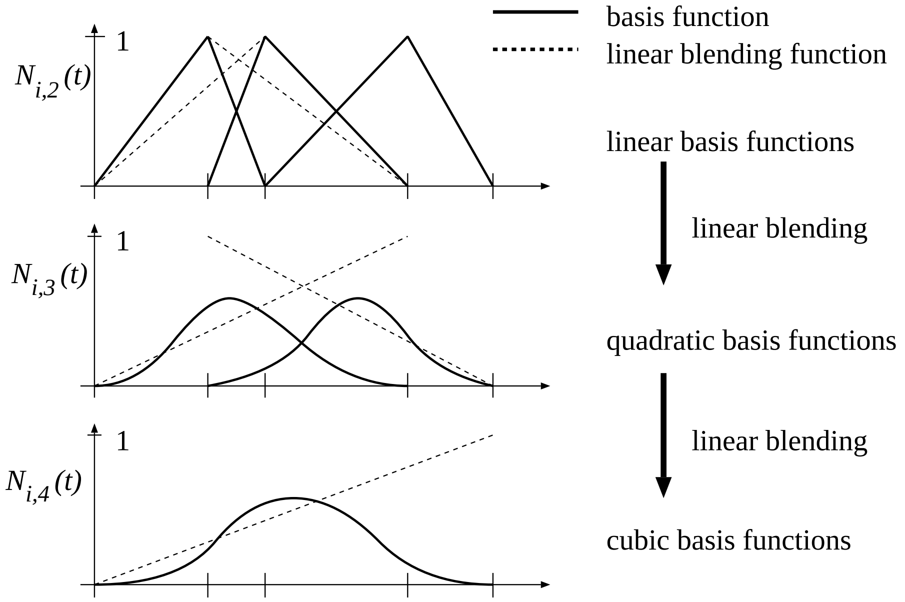

Figure 13.5: Construction of B-spline blending functions

It is obvious from the construction process that $N_{i,k}(t)$ is non-zero only in $[t_i, t_{i+k}]$. Thus a control point p_i can affect the generated

$$p(t) = \sum_{i=0}^{n} p_i \cdot N_{i,k}(t)$$

only in $[t_i, t_{i+k}]$, and therefore the B-spline method has the local control property. The function $p(t)$ is a piecewise polynomial of degree $k-1$, and it can be proven that its derivatives of order $0, 1, \ldots, k-2$ are all continuous at the joints. For animation purposes C^2 continuity is required, thus 4-order (degree 3, that is cubic) B-splines are used.

Examining the cubic B-spline method more carefully, we can see that the interpolation requirement, that is $p(t_i) = p_i$, is not provided, because at t_i more than one blending functions are different from zero. Thus the cubic B-spline offers an **approximation method**. The four blending functions affecting a single point, however, are all positive and their sum is 1, that is, the point is always in the **convex hull** of the 4 nearest control points, and thus the resulting function will follow the polygon of the control points quite reasonably.

The fact that the B-splines offer an approximation method does not mean that they cannot be used for interpolation. If a B-spline which passes through points $[t_0, p_0; t_1, p_1; \ldots ; t_n, p_n]$ is needed, then a set of control points $[c_{-1}, c_0, c_1 \ldots c_{n+1}]$ must be found so that:

$$p(t_j) = \sum_{i=-1}^{n+1} c_i \cdot N_{i,k}(t_j) = p_j. \tag{13.26}$$

This is a linear system of n equations which have $n+2$ unknown variables. To make the problem determinant the derivatives at the beginning and at the end of the function must be defined. The resulting linear equation can be solved for the unknown control points.

13.5 Interpolation with quaternions

The previous section discussed several trajectory interpolation techniques which determined a time function for each independently controllable motion parameter. These parameters can be used later on to derive the transformation matrix. This two-step method guarantees that the *"inbetweened"* samples are really valid transformations which do not destroy the shape of the animated objects.

Interpolating in the motion parameter space, however, generates new problems which need to be addressed in animation. Suppose, for the sake of simplicity, that an object is to be animated between two different positions and orientations with uniform speed. In parameter space straight line segments are the shortest paths between the two knot points. Unfortunately these line segments do not necessarily correspond to the shortest "natural" path in the space of orientations, only in the space of positions. The core of the problem is the selection of the orientation parameters, that is the roll-pitch-yaw angles, since real objects rotate around a single (time-varying) direction instead of around three superficial coordinate axes, and the dependence of the angle of the single rotation on the roll-pitch-yaw angles is not linear (compare equations 5.26 and 5.30). When rotating by angle α around a given direction in time t, for instance, the linearly interpolated roll-pitch-yaw angles will not necessarily correspond to a rotation by $\lambda \cdot \alpha$ in $\lambda \cdot t$ ($\lambda \in [0..1]$), which inevitably results in uneven and "non-natural" motion. In order to demonstrate this problem, suppose that an object located in [1,0,0] has to be rotated around vector [1,1,1] by 240 degrees and the motion is defined by three knot points representing rotation by 0, 120 and 240 degrees respectively (figure 13.6). Rotation by 120 degrees moves the x axis to the z axis and rotation by 240 degrees transforms the x axis to y axis. These transformations, however, are realized by 90 degree rotations around the y axis then around the x axis if the roll-pitch-yaw representation is used. Thus the interpolation in roll-pitch-yaw angles forces the object to rotate first around the y axis by 90 degrees then around the x axis instead of rotating continuously around [1,1,1]. This obviously results in uneven and unrealistic motion even if this effect is decreased by a C^2 interpolation.

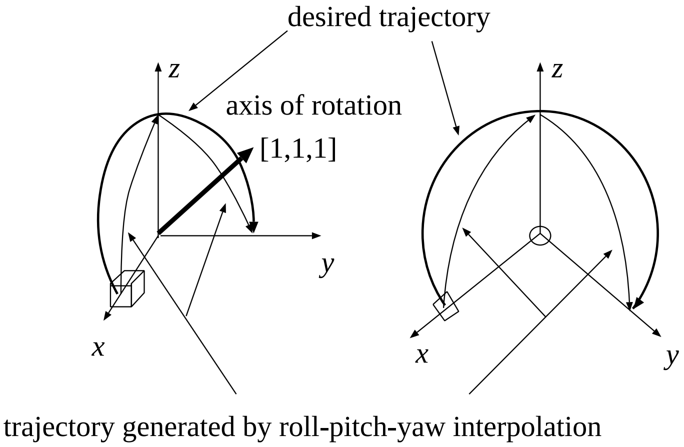

Figure 13.6: Problems of interpolation in roll-pitch-yaw angles

This means that a certain orientation change cannot be inbetweened by independently interpolating the roll-pitch-yaw angles in cases when these effects are not tolerable. Rather the axis of the final rotation is required, and the 2D rotation around this single axis must be interpolated and sampled in the different frames. Unfortunately, neither the roll-pitch-yaw parameters nor the transformation matrix supply this required information including the axis and the angle of rotation. Another representation is needed which explicitly refers to the axis and the angle of rotation.

In the mid-eighties several publications appeared promoting **quaternions** as a mathematical tool to describe and handle rotations and orientations in graphics and robotics [Bra82]. Not only did quaternions solve the problem of natural rotation interpolation, but they also simplified the calculations and out-performed the standard roll-pitch-yaw angle based matrix operations.

Like a matrix, a quaternion q can be regarded as a tool for changing one vector $\vec{u}$ into another $\vec{v}$:

$$\vec{u} \stackrel{q}{\Longrightarrow} \vec{v}. \tag{13.27}$$

Matrices do this change with a certain element of redundancy, that is, there is an infinite number of matrices which can transform one vector to another given vector. For 3D vectors, the matrices have 9 elements, although 4 real numbers can define this change unambiguously, namely:

1. The change of the length of the vector.
2. The plane of rotation, which can be defined by 2 angles from two given axes.
3. The angle of rotation.

A quaternion q, on the other hand, consists only of the necessary 4 numbers, which are usually partitioned into a pair consisting of a scalar element and a vector of 3 scalars, that is:

$$q = [s, x, y, z] = [s, \vec{w}]. \tag{13.28}$$

Quaternions are four-vectors (this is why they were given this name), and inherit vector operations including addition, scalar multiplication, dot product and norm, but their multiplication is defined specially, in a way somehow similar to the arithmetic of complex numbers, because quaternions can also be interpreted as a generalization of the complex numbers with s as the real part and x, y, z as the imaginary part. Denoting the imaginary axes by $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ yields:

$$q = s + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}. \tag{13.29}$$

In fact, Sir Hamilton introduced the quaternions more than a hundred years ago to generalize complex numbers, which can be regarded as *pairs* with special algebraic rules. He failed to find the rules for *triples*, but realized that the generalization is possible for *quadruples* with the rules:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1, \quad \mathbf{ij} = \mathbf{k}, \text{ etc.}$$

To summarize, the definitions of the operations on quaternions are:

$$q_1 + q_2 = [s_1, \vec{w}_1] + [s_2, \vec{w}_2] = [s_1 + s_2, \vec{w}_1 + \vec{w}_2],$$

$$\lambda q = \lambda[s, \vec{w}] = [\lambda s, \lambda \vec{w}],$$

$$q_1 \cdot q_2 = [s_1, \vec{w}_1] \cdot [s_2, \vec{w}_2] = [s_1 s_2 - \vec{w}_1 \cdot \vec{w}_2, s_1 \vec{w}_2 + s_2 \vec{w}_1 + \vec{w}_1 \times \vec{w}_2],$$

$$\langle q_1, q_2 \rangle = \langle [s_1, x_1, y_1, z_1], [s_2, x_2, y_2, z_2] \rangle = s_1 s_2 + x_1 x_2 + y_1 y_2 + z_1 z_2,$$

$$||q|| = ||[s, x, y, z]|| = \sqrt{\langle q, q \rangle} = \sqrt{s^2 + x^2 + y^2 + z^2}. \tag{13.30}$$

Quaternion multiplication and addition satisfies the distributive law. Addition is commutative and associative. Multiplication is associative but is not commutative. It can easily be shown that the multiplicative identity is $I = [1, \vec{0}]$. With respect to quaternion multiplication the inverse quaternion is:

$$q^{-1} = \frac{[s, -\vec{w}]}{||q||^2} \tag{13.31}$$

since

$$[s, \vec{w}] \cdot [s, -\vec{w}] = [s^2 + |\vec{w}|^2, \vec{0}] = ||q||^2 \cdot [1, \vec{0}]. \tag{13.32}$$

As for matrices, the inverse reverses the order of multiplication, that is:

$$(q_2 \cdot q_1)^{-1} = q_1^{-1} \cdot q_2^{-1}. \tag{13.33}$$

Our original goal, the rotation of 3D vectors using quaternions, can be achieved relying on quaternion multiplication by having extended the 3D vector by an $s = 0$ fourth parameter to make it, too, a quaternion:

$$\vec{u} \stackrel{q}{\Longrightarrow} \vec{v}: \qquad [0, \vec{v}] = q \cdot [0, \vec{u}] \cdot q^{-1} =$$

$$\frac{[0, s^2\vec{u} + 2s(\vec{w} \times \vec{u}) + (\vec{w} \cdot \vec{u})\vec{w} + \vec{w} \times (\vec{w} \times \vec{u})]}{||q||^2}. \tag{13.34}$$

Note that a scaling in quaternion $q = [s, \vec{w}]$ makes no difference to the resulting vector v, since scaling of $[s, \vec{w}]$ and $[s, -\vec{w}]$ in q^{-1} is compensated for by the attenuation of $||q||^2$. Thus, without the loss of generality, we assume that q is a unit quaternion, that is

$$||q||^2 = s^2 + |\vec{w}|^2 = 1 \tag{13.35}$$

For unit quaternions, equation 13.34 can also be written as:

$$[0, \vec{v}] = q \cdot [0, \vec{u}] \cdot q^{-1} = [0, \vec{u} + 2s(\vec{w} \times \vec{u}) + 2\vec{w} \times (\vec{w} \times \vec{u})] \qquad (13.36)$$

since

$$s^2\vec{u} = \vec{u} - |\vec{w}|^2\vec{u} \quad \text{and} \quad (\vec{w} \cdot \vec{u})\vec{w} - |\vec{w}|^2\vec{u} = \vec{w} \times (\vec{w} \times \vec{u}). \qquad (13.37)$$

In order to examine the effects of the above defined transformation, vector $\vec{u}$ is first supposed to be perpendicular to vector $\vec{w}$, then the parallel case will be analyzed.

If vector $\vec{u}$ is perpendicular to quaternion element $\vec{w}$, then for unit quaternions equation 13.36 yields:

$$q \cdot [0, \vec{u}] \cdot q^{-1} = [0, \vec{u}(1 - 2|\vec{w}|^2) + 2s(\vec{w} \times \vec{u})] = [0, \vec{v}]. \qquad (13.38)$$

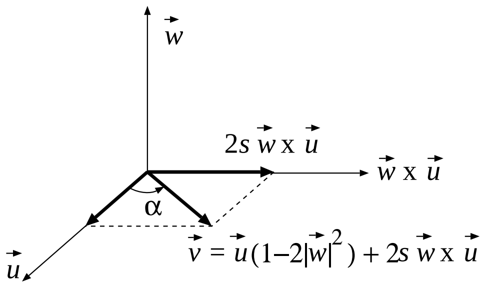

Figure 13.7: Geometry of quaternion rotation for the perpendicular case

That is, $\vec{v}$ is a linear combination of perpendicular vectors $\vec{u}$ and $\vec{w} \times \vec{u}$ (figure 13.7), it thus lies in the plane of $\vec{u}$ and $\vec{w} \times \vec{u}$, and its length is:

$$|\vec{v}| = |\vec{u}|\sqrt{(1 - 2|\vec{w}|^2)^2 + (2s|\vec{w}|)^2} = |\vec{u}|\sqrt{(1 + 4|\vec{w}|^2(s^2 + |\vec{w}|^2 - 1)} = |\vec{u}|. \qquad (13.39)$$

Since $\vec{w}$ is perpendicular to the plane of $\vec{u}$ and the resulting vector $\vec{v}$, and the transformation does not alter the length of the vector, vector $\vec{v}$ is, in fact, a rotation of $\vec{u}$ around $\vec{w}$. The cosine of the rotation angle (α) can be expressed by the dot product of $\vec{u}$ and $\vec{v}$, that is:

$$\cos\alpha = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \cdot |\vec{v}|} = \frac{(\vec{u} \cdot \vec{u})(1 - 2|\vec{w}|^2) + 2s\vec{u} \cdot (\vec{w} \times \vec{u})}{|\vec{u}|^2} = 1 - 2|\vec{w}|^2. \qquad (13.40)$$

If vector $\vec{u}$ is parallel to quaternion element $\vec{w}$, then for unit quaternions equation 13.36 yields:

$$[0, \vec{v}] = q \cdot [0, \vec{u}] \cdot q^{-1} = [0, \vec{u}]. \tag{13.41}$$

Thus the parallel vectors are not affected by quaternion multiplication as rotation does not alter the axis parallel vectors.

General vectors can be broken down into a parallel and a perpendicular component with respect to $\vec{w}$ because of the distributive property. As has been demonstrated, the quaternion transformation rotates the perpendicular component by an angle that satisfies $\cos\alpha = 1 - 2|\vec{w}|^2$ and the parallel component is unaffected, thus the transformed components will define the rotated version of the original vector $\vec{u}$ by angle α around the vector part of the quaternion.

Let us apply this concept in the reverse direction and determine the rotating quaternion for a required rotation axis $\vec{d}$ and angle α. We concluded that the quaternion transformation rotates the vector around its vector part, thus a unit quaternion rotating around unit vector $\vec{d}$ has the following form:

$$q = [s, r \cdot \vec{d}], \qquad s^2 + r^2 = 1 \tag{13.42}$$

Parameters s and r have to be selected according to the requirement that quaternion q must rotate by angle α. Using equations 13.40 and 13.42, we get:

$$\cos\alpha = 1 - 2r^2, \qquad s = \sqrt{1 - r^2}. \tag{13.43}$$

Expressing parameters s and r, then quaternion q that represents a rotation by angle α around a unit vector $\vec{d}$, we get:

$$q = [\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2} \cdot \vec{d}]. \tag{13.44}$$

The special case when $\sin\alpha/2 = 0$, that is $\alpha = 2k\pi$ and $q = [\pm 1, \vec{0}]$, poses no problem, since a rotation of an even multiple of π does not affect the object, and the axis is irrelevant.

Composition of rotations is the "concatenation" of quaternions as in matrix representation since:

$$q_2 \cdot (q_1 \cdot [0, \vec{u}] \cdot q_1^{-1}) \cdot q_2^{-1} = (q_2 \cdot q_1) \cdot [0, \vec{u}] \cdot (q_2 \cdot q_1)^{-1}. \tag{13.45}$$

Let us focus on the interpolation of orientations between two knot points in the framework of quaternions. Suppose that the orientations are described in the two knot points by quaternions q_1 and q_2 respectively. For the sake of simplicity, we suppose first that q_1 and q_2 represent rotations around the same unit axis $\vec{d}$, that is:

$$q_1 = [\cos\frac{\alpha_1}{2}, \sin\frac{\alpha_1}{2}\cdot\vec{d}], \qquad q_2 = [\cos\frac{\alpha_2}{2}, \sin\frac{\alpha_2}{2}\cdot\vec{d}]. \tag{13.46}$$

Calculating the dot product of q_1 and q_2,

$$\langle q_1, q_2\rangle = \cos\frac{\alpha_1}{2}\cdot\cos\frac{\alpha_2}{2} + \sin\frac{\alpha_1}{2}\cdot\sin\frac{\alpha_2}{2} = \cos\frac{\alpha_2-\alpha_1}{2},$$

we come to the interesting conclusion that the angle of rotation between the two orientations represented by the quaternions is, in fact, half of the angle between the two quaternions in 4D space.

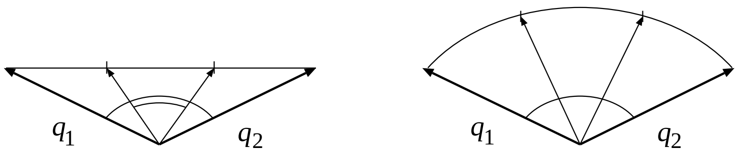

Figure 13.8: Linear versus spherical interpolation of orientations

Our ultimate objective is to move an object from an orientation represented by q_1 to a new orientation of q_2 by an even and uniform motion. If linear interpolation is used to generate the path of orientations between q_1 and q_2, then the angles of the subsequent quaternions will not be constant, as is demonstrated in figure 13.8. Thus the speed of the rotation will not be uniform, and the motion will give an effect of acceleration followed by deceleration, which is usually undesirable.

Instead of linear interpolation, a non-linear interpolation must be found that guarantees the constant angle between the subsequent interpolated quaternions. Spherical interpolation obviously meets this requirement, where the interpolated quaternions are selected uniformly from the arc between q_1 and q_2. If q_1 and q_2 are unit quaternions, then all the interpolated quaternions will also be of unit length. Unit-size quaternions can be regarded as unit-size four-vectors which correspond to a 4D unit-radius sphere. An appropriate interpolation method must generate the great arc between q_1 and q_2, and as can easily be shown, this great arc has the following form:

$$q(t) = \frac{\sin(1-t)\theta}{\sin\theta} \cdot q_1 + \frac{\sin t\theta}{\sin\theta} \cdot q_2, \tag{13.47}$$

where $\cos\theta = \langle q_1, q_2 \rangle$ (figure 13.9).

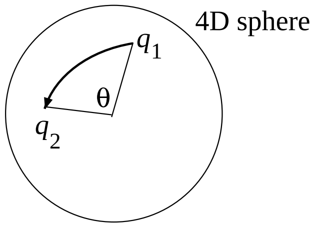

Figure 13.9: Interpolation of unit quaternions on a 4D unit sphere

In order to demonstrate that this really results in a uniform interpolation, the following equations must be proven for $q(t)$:

$$||q(t)|| = 1, \quad \langle q_1, q(t)\rangle = \cos(t\theta), \quad \langle q_2, q(t)\rangle = \cos((1-t)\theta). \tag{13.48}$$

That is, the interpolant is really on the surface of the sphere, and the angle of rotation is a linear function of the time t.

Let us first prove the second assertion (the third can be proven similarly):

$$\langle q_1, q(t)\rangle = \frac{\sin(1-t)\theta}{\sin\theta} + \frac{\sin t\theta}{\sin\theta} \cdot \cos\theta =$$

$$\frac{\sin\theta \cdot \cos t\theta}{\sin\theta} - \frac{\sin t\theta \cdot \cos\theta}{\sin\theta} + \frac{\sin t\theta}{\sin\theta} \cdot \cos\theta = \cos(t\theta). \tag{13.49}$$

Concerning the norm of the interpolant, we can use the definition of the norm and the previous results, thus:

$$||q(t)||^2 = \langle q(t), q(t)\rangle = \langle \frac{\sin(1-t)\theta}{\sin\theta} \cdot q_1 + \frac{\sin t\theta}{\sin\theta} \cdot q_2, q(t)\rangle =$$

$$\frac{\sin(1-t)\theta}{\sin\theta} \cdot \cos(t\theta) + \frac{\sin t\theta}{\sin\theta} \cdot \cos((1-t)\theta) = \frac{\sin((1-t)\theta + t\theta)}{\sin\theta} = 1 \tag{13.50}$$

If there is a series of consecutive quaternions $q_1, q_2, \ldots, q_n$ to follow during the animation, this interpolation can be executed in a similar way to that discussed in the previous section. The blending function approach can be used, but here the constraints are slightly modified. Supposing $b_1(t), b_2(t), \ldots, b_n(t)$ are blending functions, for any t, they must satisfy that:

$$||b_1(t)q_1 + b_2(t)q_2 + \ldots + b_n(t)q_n|| = 1 \tag{13.51}$$

which means that the curve must lie on the sphere. This is certainly more difficult than generating a curve in the plane of the control points, which was done in the previous section. Selecting $b_i(t)$s as piecewise curves defined by equation 13.47 would solve this problem, but the resulting curve would not be of C^1 and C^2 type.

Shoemake [Sho85] proposed a successive linear blending technique on the surface of the sphere to enforce the continuous derivatives. Suppose that a curve segment adjacent to $q_1, q_2, \ldots q_n$ is to be constructed. In the first phase, piecewise curves are generated between q_1 and q_2, q_2 and q_3, etc.:

$$q^{(1)}(t_1) = \frac{\sin(1-t_1)\theta_1}{\sin\theta_1} q_1 + \frac{\sin t_1\theta_1}{\sin\theta_1} q_2,$$

$$q^{(2)}(t_2) = \frac{\sin(1-t_2)\theta_2}{\sin\theta_2} q_2 + \frac{\sin t_2\theta_2}{\sin\theta_2} q_3,$$

$$\vdots$$

$$q^{(n-1)}(t_{n-1}) = \frac{\sin(1-t_{n-1})\theta_{n-1}}{\sin\theta_{n-1}} q_{n-1} + \frac{\sin t_{n-1}\theta_{n-1}}{\sin\theta_{n-1}} q_n. \tag{13.52}$$

In the second phase these piecewise segments are blended to provide a higher order continuity at the joints. Let us mirror q_{i-1} with respect to q_i on the sphere generating q^*_{i-1}, and determine the point a_i that bisects the great arc between q_{i+1} and q^*_{i-1} (see figure 13.10). Let us form another great arc by mirroring a_i with respect to q_i generating a^*_i as the other endpoint. Having done this, a C^2 approximating path $g(t)$ has been produced from the neighborhood of q_{i-1} (since $q_{i-1} \approx a^*_i$) through q_i to the neighborhood of q_{i+1} (since $q_{i+1} \approx a_i$). This great arc is subdivided into two segments producing $g^{(i-1)}(t)$ between a^*_i and q_i and $g^{(i)}(t)$ between q_i and a_i respectively.

In order to guarantee that the final curve goes through q_{i-1} and q_{i+1} without losing its smoothness, a linear blending is applied between the piecewise curves $q^{(i-1)}(t)$, $q^{(i)}(t)$ and the new approximation arcs $g^{(i-1)}(t)$, $g^{i}(t)$ in such a way that the blending gives weight 1 to $q^{(i-1)}(t)$ at $t_{i-1}=0$, to $q^{(i+1)}(t)$ at $t_i = 1$ and to the approximation arcs $g^{(i-1)}$ and $g^{(i)}$ at $t_{i-1}=1$ and $t_i = 0$ respectively, that is:

$$\begin{aligned} \hat{q}^{(i-1)}(t_{i-1}) &= (1-t_{i-1})\cdot q^{(i-1)}(t_{i-1}) + t_{i-1}\cdot g^{(i-1)}(t_{i-1}) \\ \hat{q}^{(i)}(t_i) &= (1-t_i)\cdot g^{(i)}(t_i) + t_i \cdot q^{(i)}(t_i). \end{aligned} \tag{13.53}$$

This method requires uniform timing between the successive knot points. By applying a linear transformation in the time domain, however, any kind of timing can be specified.

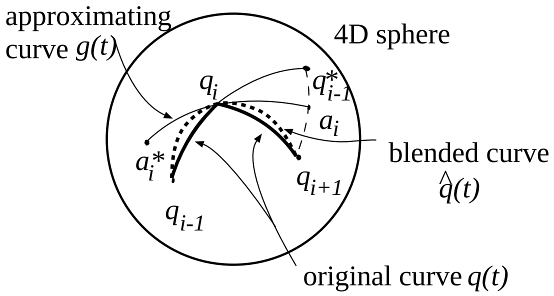

Figure 13.10: Shoemake's algorithm for interpolation of unit quaternions on a 4D unit sphere

Once the corresponding quaternion of the interpolated orientation is determined for a given time parameter t, it can be used for rotating the objects in the model. Comparing the number of instructions needed for (spherical) quaternion interpolation and rotation of the objects by quaternion multiplication, we can see that the method of quaternions not only provides more realistic motion but is slightly more effective computationally.

However, the traditional transformation method based on matrices can be combined with this new approach using quaternions. Using the interpolated quaternion a corresponding transformation matrix can be set up. More precisely this is the upper-left minor matrix of the transformation matrix, which is responsible for the rotation, and the last row is the position vector which is interpolated by the usual techniques. In order to identify the transformation matrix from a quaternion, the way the basis vectors are transformed when multiplied by the quaternion must be examined. By applying unit quaternion $q = [s, x, y, z]$ to the first, second and third standard basis vectors [1,0,0], [0,1,0] and [0,0,1], the first, second and the third rows of the matrix can be determined, thus:

$$\mathbf{A}_{3\times 3} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2sz & 2xz-2sy \\ 2xy-2sz & 1-2x^2-2z^2 & 2yz+2sx \\ 2xz+2sy & 2yz-2sx & 1-2x^2-2y^2 \end{bmatrix}. \tag{13.54}$$

During the interactive design phase of the animation sequence, we may need the inverse conversion which generates the quaternion from the (orthonormal) upper-left part of the transformation matrix or from the roll-pitch-yaw angles. Expressing $[s, x, y, z]$ from equation 13.54 we get:

$$s = \sqrt{1-(x^2+y^2+z^2)} = \frac{1}{2}\sqrt{a_{11}+a_{22}+a_{33}+1},$$

$$x = \frac{a_{23}-a_{32}}{4s}, \quad y = \frac{a_{31}-a_{13}}{4s}, \quad z = \frac{a_{12}-a_{21}}{4s}. \tag{13.55}$$

The roll-pitch-yaw (α, β, γ) description can also be easily transformed into a quaternion if the quaternions corresponding to the elementary rotations are combined:

$$q(\alpha,\beta,\gamma) = [\cos\frac{\alpha}{2}, (0,0,\sin\frac{\alpha}{2})] \cdot [\cos\frac{\beta}{2}, (0,\sin\frac{\beta}{2},0)] \cdot [\cos\frac{\gamma}{2}, (\sin\frac{\gamma}{2},0,0)]. \tag{13.56}$$

13.6 Hierarchical motion

So far we have been discussing the animation of individual rigid objects whose paths could be defined separately taking just several collision constraints into consideration. To avoid unexpected collisions in these cases, the animation sequence should be reviewed and the definition of the keyframes must be altered iteratively until the animation sequence is satisfactory.

Real objects usually consist of several **linked segments**, as for example a human body is composed of a trunk, a head, two arms and two legs. The arms can in turn be broken down into an upper arm, a lower arm, hand, fingers etc. A car, on the other hand, is an assembly of its body and the four wheels (figure 13.11). The segments of a composed object (an assembly) do not move independently, because they are linked together by joints which restrict the relative motion of linked segments. Revolute joints, such as human joints and the coupling between the wheel and the body of the car, allow for specific rotations about a fixed common point of the two linked segments. Prismatic joints, common in robots and in machines [Lan91], however, allow the parts to translate in a given direction. When these assembly structures are animated, the constraints generated by the features of the links must be satisfied in every single frame of the animation sequence. Unfortunately, it is not enough to meet this requirement in the keyframes only and animate the segments separately. A running human body, for instance, can result in frames when the trunk, legs, and the arms are separated even if they are properly connected in the keyframes. In order to avoid these annoying effects, the constraints and relationships of the various segments must continuously be taken into consideration during the interpolation, not just in the knot points. This can be achieved if the segments are not animated separately but their relative motion is specified.

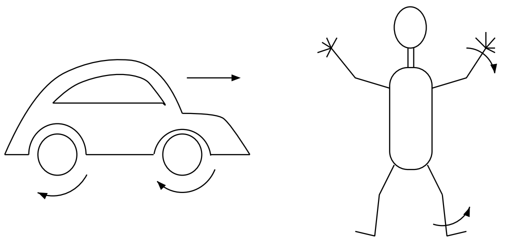

Figure 13.11: Examples of multi-segment objects

Recall that the motion of an individual object is defined by a time-varying modeling transformation matrix which places the object in the common world coordinate system. If the relative motion of object i must be defined with respect to object j, then the relative modeling transformation $\mathbf{T_{ij}}$ of object i must place it in the local coordinate system of object j. Since object j is fixed in its own modeling coordinate system, $\mathbf{T_{ij}}$ will determine the relative position and orientation of object i with respect to object j. A point $\vec{r}_i$ in object i's coordinate system will be transformed to point:

$$[\vec{r}_j, 1] = [\vec{r}_i, 1] \cdot \mathbf{T_{ij}} \qquad \Longrightarrow \qquad \vec{r}_j = \vec{r}_i \cdot \mathbf{A_{ij}} + \vec{p}_{ij} \tag{13.57}$$

in the local modeling system of object j if $\mathbf{A_{ij}}$ and $\vec{p}_{ij}$ are the orientation matrix and translation vector of matrix $\mathbf{T_{ij}}$ respectively. While animating this object, matrix $\mathbf{T_{ij}}$ is a function of time. If only orientation matrix $\mathbf{A_{ij}}$ varies with time, the relative position of object i and object j will be fixed, that is, the two objects will be linked together by a revolute joint at point $\vec{p}_{ij}$ of object j and at the center of its own local coordinate system of object i. Similarly, if the orientation matrix is constant in time, but the position vector is not, then a prismatic joint is simulated which allows object i to move anywhere but keeps its orientation constant with respect to object j.

Transformation $\mathbf{T_{ij}}$ places object i in the local modeling coordinate system of object j. Thus, the world coordinate points of object i can be generated if another transformation — object j's modeling transformation $\mathbf{T_j}$ which maps the local modeling space of object j onto world space — is applied:

$$[\vec{r}_w, 1] = [\vec{r}_j, 1] \cdot \mathbf{T_j} = [\vec{r}_i, 1] \cdot \mathbf{T_{ij}} \cdot \mathbf{T_j}. \tag{13.58}$$

In this way, whenever object j is moved, object i will follow it with a given relative orientation and position since object j's local modeling transformation will affect object i as well. Therefore, object j is usually called the **parent segment** of object i and object i is called the **child segment** of object j. A child segment can also be a parent of other segments. In a simulated human body, for instance, the upper arm is the child of the trunk, in turn is the parent of the lower arm (figure 13.12). The lower arm has a child, the hand, which is in turn the parent of the fingers. The parent-child relationships form a *hierarchy of segments* which is responsible for determining the types of motion the assembly structure can accomplish. This hierarchy usually corresponds to a *tree-structure* where a child has only one parent, as in the examples of the human body or the car. The motion of an assembly having a tree-like hierarchy can be controlled by defining the modeling transformation of the complete structure and the relative modeling transformation for every single parent-child pair (joints in the assembly). In order to set these transformations, the normal interactive techniques can be used. First we move the complete human body (including the trunk, arms, legs etc.), then we arrange the arms (including the lower arms, hand etc.) and legs, then the lower arms, hands, fingers etc. by interactive manipulation. In the animation design program, this interactive manipulation updates the modeling transformation of the body first, then the relative modeling trans-

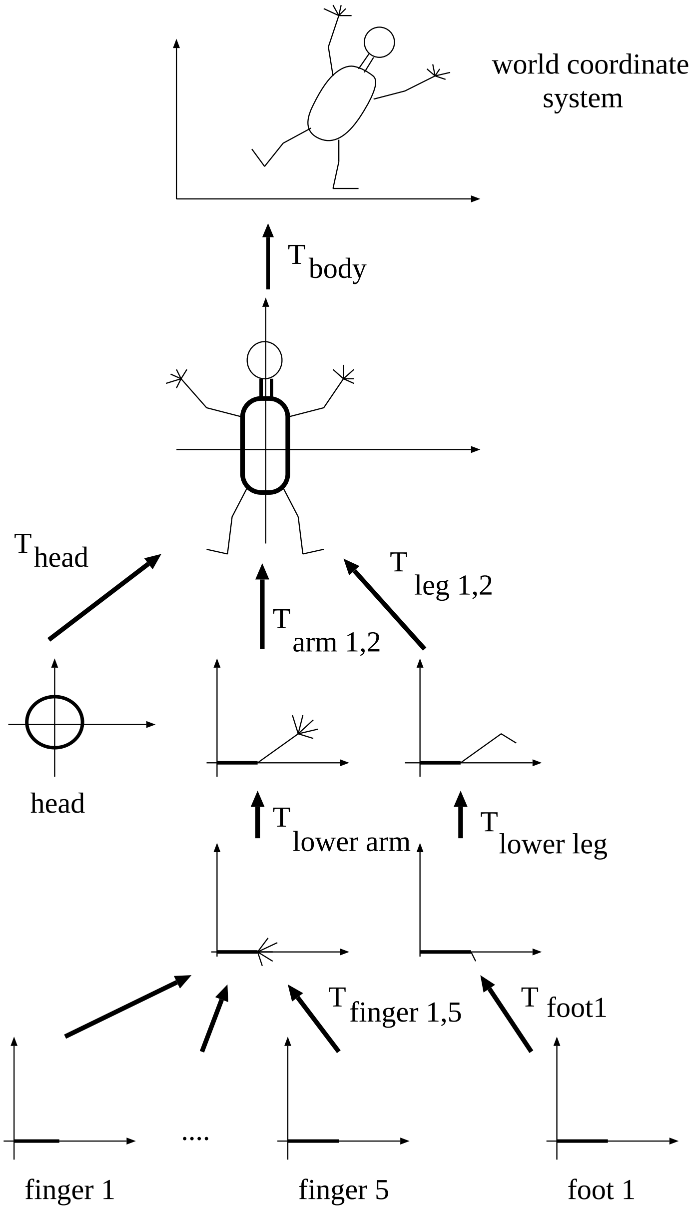

Figure 13.12: Transformation tree of the human body

formation of the arms and legs, then the relative transformation of the lower arms etc. Thus, in each keyframe, the modeling transformation in the joints of hierarchy can be defined. During interpolation, these transformations are interpolated independently and meeting the requirements of the individual joints (in a revolute joint the relative position is constant), but the overall transformation of a segment is generated by the concatenation of the relative transformations of its ancestors and the modeling transformation of the complete assembly structure. This will guarantee that the constraints imposed by the joints in the assembly will be satisfied.

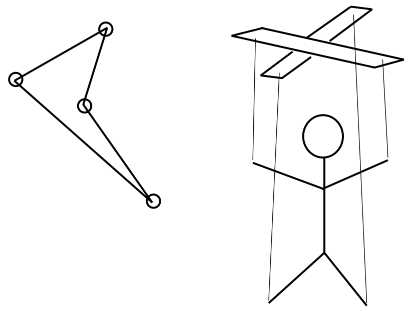

Figure 13.13: Assemblies having non tree-like segment structure

13.6.1 Constraint-based systems

In tree-like assemblies independent interpolation is made possible by the assumption that each joint enjoys independent degree(s) of freedom in its motion. Unfortunately, this assumption is not always correct, and this can cause, for example, the leg to go through the trunk which is certainly not "realistic". Most of these collision problems can be resolved by reviewing the animation sequence that was generated without taking care of the collisions and the interdependence of various joints, and modifying the keyframes until a satisfactory result is generated. This try-and-check method may still work for problems where there are several objects in the scene and their collision must be avoided, but this can be very tiresome, so we would prefer methods which resolve these collision and interdependence problems automatically. The application of these automatic constraint resolution methods is essential for non tree-like assemblies (figure 13.13), where the degree of freedom is less than the individually controllable parameters of the joints, because the independent interpolation of a subset of joint parameters may cause other joints to fall apart even if all requirements are met in the keyframes.

Such an automatic constraint resolution algorithm basically does the same as the user who interactively tries to modify the definition of the sequence and checks whether or not the result satisfied the constraints. The algorithm is controlled by an *error* function which is non-zero if a constraint is not satisfied and usually increases as we move away from the allowed arrangements. The motion algorithm tries to minimize this function by interpolating a C^2 function for each controllable parameter, calculating the maximum of this error function along the path and then modifying the knot points of the C^2 function around the parameters where the error value is large. Whenever a knot point is modified, the trajectory is evaluated again and a check is made to see the error value has decreased or not. If it has decreased, then the previous modification is repeated; if it has increased, the previous modification is inverted. The new parameter knot point should be randomly perturbed to avoid infinite oscillations and to reduce the probability of reaching a local minimum of the error function. The algorithm keeps repeating this step until either it can generate zero error or it decides that no convergence can be achieved possibly because of overconstraining the system. This method is also called the **relaxation technique**.

When animating complex structures, such as the model of the human body, producing the effect of realistic motion can be extremely difficult and can require a lot of expertise and experience of traditional cartoon designers. The C^2 interpolation of the parameters is a necessary but not a sufficient requirement for this. Generally, the real behavior and the internal structure of the simulated objects must be understood in order to imitate their motion. Fortunately, the most important rule governing the motion of animals and humans is very simple: *Living objects always try to minimize the energy needed for a given change of position and orientation and the motion must satisfy the geometric constraints of the body and the dynamic constraints of the muscles.* Thus, when the motion parameters are interpolated between the keyframe positions, the force needed in the different joints as well as the potential and kinetic energy must be calculated. This seems simple, but the actual calculation can be very complex. Fortunately, the same problems have arisen in the control of robots, and therefore the solution methods developed for robotics can also be used here [Lan91]. The previous relaxation technique must be extended to find not only a trajectory where the error including the geometric and dynamic constraints is zero, but one where the energy generated by the "muscles" in the joints is minimal.

Finally it must be mentioned that an important field of animation, called the **scientific visualization**, focuses on the behavior of systems that are described by a set of physical laws. The objective is to find an arrangement or movement that satisfies these laws.

13.7 Double buffering

Animation means the fast generation of images shown one after the other on the computer screen. If the display of these static images takes a very short time, the human eye is unable to identify them as separate pictures, but rather interprets them as a continuously changing sequence. This phenomenon is well known and is exploited in the motion picture industry.

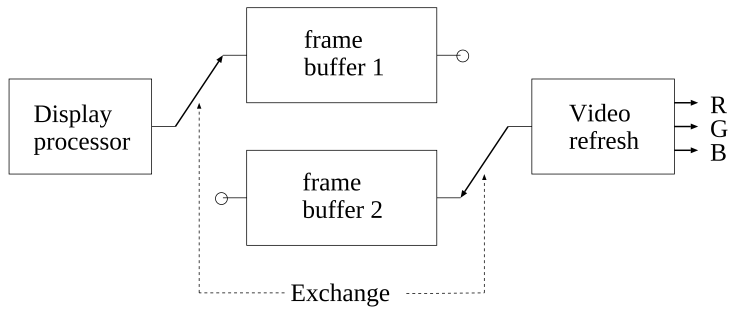

Figure 13.14: Double buffer animation systems

When an image is generated on the computer screen, it usually evolves gradually, depending on the actual visibility algorithm. Painter's algorithm, for example, draws the polygons in order of distance from the camera; thus even those polygons that turn out to be invisible later on will be seen on the screen for a very short time during the image generation. The z-buffer algorithm, on the other hand, draws a polygon point if the previously displayed polygons do not hide it, which can also cause the temporary display of invisible polygons. The evolution of the images, even if it takes a very short time, may cause noticeable and objectionable effects which need to be eliminated. We must prevent the observer from seeing the generation process of the images and present to him the final result only. This problem had to be solved in traditional motion pictures as well. The usual way of doing it is via the application of two frame buffers, which leads to a method of **double-buffer animation** (figure 13.14). In each frame of the animation sequence, the content of one of the frame buffers is displayed, while the other is being filled up by the image generation algorithm. Once the image is complete, the roles of the two frame buffers are exchanged. Since it takes practically no time — being only a switch of two multiplexers during the vertical retrace — only complete images are displayed on the screen.

13.8 Temporal aliasing

As has been mentioned, animation is a fast display of static image sequences providing the illusion of continuous motion. This means that the motion must be sampled in discrete time instances and then the "continuous" motion produced by showing these static images until the next sampling point. Thus, sampling artifacts, called **temporal aliasing**, can occur if the sampling frequency and the frequency range of the motion do not satisfy the sampling theorem. Well-known examples of temporal aliasing are backward rotating wheels and the jerky motion which can be seen in old movies. These kinds of temporal aliasing phenomena are usually called **strobing**. Since the core of the problem is the same as spatial aliasing due to the finite resolution raster grid, similar approaches can be applied to solve it, including either post-filtering with supersampling, which generates several images in each frame time and produces the final one as their average, or pre-filtering, which solves the visibility and shading problems as a function of time and calculates the convolution of the time-varying image with an appropriate filter function. The filtering process will produce **motion blur** for fast moving objects just as moving objects cause blur on normal films because of finite exposure time. Since visibility and shading algorithms have been developed to deal with static object spaces and images, and most of them are not appropriate for a generalization to take time-varying phenomena into account, temporal anti-aliasing methods usually use a combination of post-filtering and supersampling. (An exceptional case is a kind of ray tracing which allows for some degree of dynamic generalization as proposed by Cook [CPC84] creating a method called *distributed ray tracing*.)

Let ΔT be the interval during which the images, called **subframes**, are averaged. This time is analogous to the exposure time when the shutter is open in a normal camera. If n number of subframes are generated and box filtering is used, then the averaged color at some point of the image is:

$$I = \frac{1}{n}\sum_{i=0}^{n-1} I(t_0 + \frac{i \cdot \Delta T}{n}). \tag{13.59}$$

The averaging calculation can be executed in the frame buffer. Before writing a pixel value into the frame buffer, its red, green and blue components must be divided by n, and the actual *pixel operation* must be set to "arithmetic addition". The number of samples, n, must be determined to meet (at least approximately) the requirements of the sampling theorem, taking the temporal frequencies of the motion into consideration. Large n values, however, are disadvantageous because temporal supersampling increases the generation time of the animation sequence considerably. Fortunately, acceptable results can be generated with relatively small n if this method is combined with **stochastic sampling** (see section 11.4), that is, if the sample times of the subframes are selected randomly rather than uniformly in the frame interval. Stochastic sampling will transform temporal aliasing into noise appearing as motion blur. Let δ be a random variable distributed in [0,1] to perturb the uniform sample locations. The modified equation to calculate the averaged color is:

$$I = \frac{1}{n}\sum_{i=0}^{n-1} I(t_0 + \frac{(i+\delta) \cdot \Delta T}{n}). \tag{13.60}$$

Temporal filtering can be combined with spatial filtering used to eliminate the "jaggies" [SR92]. Now an image (frame) is averaged from n static images. If these static images are rendered assuming a slightly shifting pixel grid, then the averaging will effectively cause the static parts of the image to be box filtered. The shift of the pixel grid must be evenly distributed in $[(0,0)\ldots(1,1)]$ assuming pixel coordinates. This can be achieved by the proper control of the real to integer conversion during image generation. Recall that we used the Trunc function to produce this, having added 0.5 to the values in the initialization phase. By modifying this 0.5 value in the range of [0,1], the shift of the pixel grid can be simulated.