APPENDIX

Background Information and Techniques

B

B.1 Vectors and matrices

A *vector* is a one-dimensional list of values. This list can be shown as a row vector or a column vector (e.g., Eq. B.1). In general, a matrix is an n-dimensional array of values. For purposes of this book, a matrix is two-dimensional (e.g., Eq. B.2).

$$\begin{bmatrix} a & b & c \end{bmatrix} \quad \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{B.1}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{B.2}$$

Matrices are multiplied together by taking the ith row of the first matrix and multiplying each element by the corresponding element of the jth column of the second matrix and summing all the products to produce the i,jth element. When computing $C = AB$, where A has v elements in each row and B has v elements in each column, an element C_{ij} is computed according to Equation B.3.

$$\begin{aligned} C_{ij} &= A_{i1}B_{1j} + A_{i2}B_{2j} + A_{i3}B_{3j} + \ldots + A_{iv}B_{vj} \\ &= \sum_{k=1}^{v} A_{ik}B_{kj} \end{aligned} \tag{B.3}$$

The "inside" dimension of the matrices must match in order for the matrices to be multiplied together. That is, if A and B are multiplied and A is a matrix with U rows and V columns (a $U \times V$ matrix), then B must be a $V \times W$ matrix; the result will be a $U \times W$ matrix. In other words, the number of columns (the number of elements in a row) of A must be equal to the number of rows (the number of elements in a column) of B. As a more concrete example, consider multiplying two 3×3 matrices. Equation B.4 shows the computation for the first element.

$$
\begin{aligned}
AB &= \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} \\
&= \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \end{bmatrix}
\end{aligned} \tag{B.4}
$$

The *transpose* of a vector or matrix is the original vector or matrix with its rows and columns exchanged (e.g., Eq. B.5). The *identity matrix* is a square matrix with ones along its diagonal and zeros elsewhere (e.g., Eq. B.6). The *inverse* of a square matrix when multiplied by the original matrix produces the identity matrix (e.g., Eq. B.7). The *determinant* of a 3×3 matrix is formed as shown in Equation B.8. The determinant of matrices greater than 3×3 can be defined recursively. First, define an element's *submatrix* as the matrix formed when removing the element's row and column from the original matrix. The determinant is formed by considering any row, element by element. The determinant is the first element of the row times the determinant of its submatrix, minus the second element of the row times the determinant of its submatrix, plus the third element of the row times the determinant of its submatrix, and so on. The sum is formed for the entire row, alternating additions and subtractions.

$$
\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}^T = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix} \tag{B.5}
$$

$$
I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{B.6}
$$

$$
MM^{-1} = M^{-1}M = I \tag{B.7}
$$

$$
\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a(ei - fh) - b(di - fg) + c(dh - eg) \tag{B.8}
$$

B.1.1 Inverse matrix and solving linear systems

The inverse of a matrix is useful in computer graphics to represent the inverse of a transformation and in computer animation to solve a set of linear equations. There are various ways to compute the inverse. One common method, which is also useful for solving sets of linear equations, is LU decomposition. The basic idea is that a square matrix, for example, a 4×4, can be decomposed into a lower triangular matrix times an upper triangular matrix. How this is done is discussed later in this section. For now, it is assumed that the LU decomposition is available (Eq. B.9).

$$
\begin{aligned}
A &= \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \\
&= LU = \begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix}
\end{aligned} \tag{B.9}
$$

The decomposition of a matrix A into the L and U matrices can be used to easily solve a system of linear equations. For example, consider the case of four unknowns (x) and four equations shown in Equation B.10. Use of the decomposition permits the system of equations to be solved by forming two systems of equations using triangular matrices (Eq. B.11).

$$
\begin{aligned}
A_{11}x_1 + A_{12}x_2 + A_{13}x_3 + A_{14}x_4 &= b_1 \\
A_{21}x_1 + A_{22}x_2 + A_{23}x_3 + A_{24}x_4 &= b_2 \\
A_{31}x_1 + A_{32}x_2 + A_{33}x_3 + A_{34}x_4 &= b_3 \\
A_{41}x_1 + A_{42}x_2 + A_{43}x_3 + A_{44}x_4 &= b_4 \\
\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} &= \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}
\end{aligned} \tag{B.10}
$$

$$
\begin{aligned}
Ax &= b \\
Ax &= b \\
(LU)x &= b \\
L(Ux) &= b \\
Ux &= y \\
Ly &= b
\end{aligned} \tag{B.11}
$$

This solves the original set of equations. The advantage of doing it this way is that both of the last two equations resulting from the decomposition involve triangular matrices and therefore can be solved trivially with simple substitution methods. For example, Equation B.12 shows the solution to $Ly = b$. Notice that by solving the equations in a top to bottom fashion, the results from the equations of previous rows are used so that there is only one unknown in any equation being considered. Once the solution for y has been determined, it can be used to solve for x in $Ux = y$ using a similar approach. Once the LU decomposition of A is formed, it can be used repeatedly to solve sets of linear equations that differ only in right-hand sides, such as those for computing the inverse of a matrix. This is one of the advantages of LU decomposition over methods such as Gauss-Jordan elimination.

$$Ly = b$$

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$\begin{aligned} &\text{First row:} && L_{11}y_1 = b_1 \\ & && y_1 = b_1/L_{11} \\ &\text{Second row:} && L_{21}y_1 + L_{22}y_2 = b_2 \\ & && y_2 = (b2 - (L_{21}y_1))/L_{22} \\ &\text{Third row:} && L_{31}y_1 + L_{32}y_2 + L_{33}y_3 = b_3 \\ & && y_3 = (b_3 - (L_{31}y_1 - L_{32}y_2))/L_{33} \\ &\text{Fourth row:} && L_{41}y_1 + L_{42}y_2 + L_{43}y_3 + L_{44}y_4 = b_4 \\ & && y_4 = (b_4 - (L_{41}y_1) - (L_{42}y_2) - (L_{43}y_3))/L_{44} \end{aligned} \tag{B.12}$$

The decomposition procedure sets up equations and orders them so that each is solved simply. Given the matrix equation for the decomposition relationship, one can construct equations on a term-by-term basis for the A matrix. This results in N^2 equations with $N^2 + N$ unknowns. As there are more unknowns than equations, N elements are set to some arbitrary value. A particularly useful set of values is $L_{ii} = 1.0$. Once this is done, the simplest equations (for A_{11}, A_{12}, etc.) are used to establish values for some of the L and U elements. These values are then used in the more complicated equations. In this way the equations can be ordered so there is only one unknown in any single equation by the time it is evaluated. Consider the case of a 4×4 matrix. Equation B.13 repeats the original matrix equation for reference and Equation B.14 shows the resulting sequence of equations in which the underlined variable is the only unknown.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ L_{21} & 1 & 0 & 0 \\ L_{31} & L_{32} & 1 & 0 \\ L_{41} & L_{42} & L_{43} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \tag{B.13}$$

$$\begin{aligned} &\text{For the first column of } A \\ &\quad U_{11} = A_{11} \\ &\underline{L_{21}}U_{11} = A_{21} \\ &\underline{L_{31}}U_{11} = A_{31} \\ &\underline{L_{41}}U_{11} = A_{41} \\ &\\ &\text{For the second column of } A \\ &\underline{U_{12}} = A_{12} \\ &L_{21}U_{12} + \underline{U_{22}} = A_{22} \\ &L_{31}U_{12} + \underline{L_{32}}U_{22} = A_{32} \\ &L_{41}U_{12} + \underline{L_{42}}U_{22} = A_{42} \end{aligned} \tag{B.14}$$

$$
\begin{aligned}
&\text{For the third column of } A\\
&\underline{U_{13}} = A_{13}\\
&L_{21}U_{13} + \underline{U_{23}} = A_{23}\\
&L_{31}U_{13} + L_{32}U_{23} + \underline{U_{33}} = A_{33}\\
&L_{41}U_{13} + L_{42}U_{23} + \underline{L_{43}}U_{33} = A_{43}\\
&\text{For the fourth column of } A\\
&\underline{U_{14}} = A_{14}\\
&L_{21}U_{14} + \underline{U_{24}} = A_{24}\\
&L_{31}U_{14} + L_{32}U_{24} + \underline{U_{34}} = A_{34}\\
&L_{41}U_{14} + L_{42}U_{24} + L_{43}U_{34} + \underline{U_{44}} = A_{44}
\end{aligned}
\tag{B.14}
$$

Notice a two-phase pattern in each column in which terms from the U matrix from the first row to the diagonal are determined first, followed by terms from the L matrix below the diagonal. This pattern can be generalized easily to matrices of arbitrary size [16], as shown in Equation B.15. The computations for each column j must be completed before proceeding to the next column.

$$
\begin{aligned}
U_{ij} &= A_{ij} - \sum_{K=1} L_{ik}U_{kj} && \text{for } i = 1, \ldots, j\\
L_{ij} &= \frac{1}{U_{ij}}\left(Aij - \sum_{k=1}^{j-1} L_{ik}U_{kj}\right) && \text{for } i = j+1, \ldots, n
\end{aligned}
\tag{B.15}
$$

So far this is fairly simple and easy to follow. Now comes the complication—*partial pivoting*. Notice that some of the equations require a division to compute the value for the unknown. For this computation to be numerically stable (i.e., the numerical error less sensitive to particular input values), this division should be by a relatively large number. By reordering the rows, one can exert some control over what divisor is used. Reordering rows does not affect the computation if the matrices are viewed as a system of linear equations; reordering obviously matters if the inverse of a matrix is being computed. However, as long as the reordering is recorded, it can easily be undone when the inverse matrix is formed from the L and U matrices.

Consider the first column of the 4×4 matrix. The divisor used in the last three equations is U_{11}, which is equal to A_{11}. However, if the rows are reordered, then a different value might end up as A_{11}. So the objective is to swap rows so that the largest value (in the absolute value sense) of $A_{11}, A_{21}, A_{31}, A_{41}$ ends up at A_{11}, which makes the computation more stable. Similarly, in the second column, the divisor is U_{22}, which is equal to $A_{22} - (L_{21}U_{12})$. As in the case of the first column, the rows below this are checked to see if a row swap might make this value larger. The row above is not checked because that row was needed to calculate U_{12}, which is needed in the computations of the rows below it. For each successive column, there are fewer choices because the only rows that are checked are the ones below the topmost row that requires the divisor.

There is one other modification to partial pivoting. Because any linear equation has the same solution under a scale factor, an arbitrary scale factor applied to a linear equation could bias the comparisons made in the partial pivoting. To factor out the effect of an arbitrary scale factor when comparing values for the partial pivoting, one scales the coefficients for that row, just for the purposes of the comparison, so that the largest coefficient of a particular row is equal to one. This is referred to as *implicit pivoting*.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ L_{21} & U_{22} & U_{23} & U_{24} \\ L_{31} & L_{32} & U_{33} & U_{34} \\ L_{41} & L_{42} & L_{43} & U_{44} \end{bmatrix}$$

FIGURE B.1

In-place computation of the L and U values assuming no row exchanges.

When performing the decomposition, the values of the input matrix can be replaced with the values of L and U (Figure B.1). Notice that the diagonal elements of the L matrix do not have to be stored because they are routinely set to one. If row exchanges take place, then the rows of the matrices in Figure B.1 will be mixed up. For solving the linear system of equations, the row exchanges have no effect on the computed answers. However, the row exchanges are recorded in a separate array so that they can be undone for the formation of the inverse matrix. In the code that follows (Figure B.2), the LU decomposition approach is used to solve a system of linear equations and is broken down into several procedures. These procedures follow those found in *Numerical Recipes* [16].

After the execution of **LUdecomp**, the A matrix contains the elements of the L and U matrices. This matrix can then be used either for solving a linear system of equations or for computing the inverse of a matrix. The previously discussed simple substitution methods can be used to solve the equations that involve triangular matrices. In the code that follows, the subroutine **LUsubstitute** is called with the newly computed A matrix, the dimension of A, the vector of row swaps, and the right-hand vector (i.e., the b in $Ax = b$).

One of the advantages of the LU decomposition is that the decomposition matrices can be reused if the only change in a system of linear equations is the right-hand vector of values. In such a case, the routine **LUdecomp** only needs to be called once to form the LU matrices. The substitution routine, **LUsubstitute**, needs to be called with each new vector of values (remember to reuse the A matrix that holds the LU decomposition).

To perform matrix inversion, use the L and U matrices repeatedly with the b matrix holding column-by-column values of the identity matrix. In this way the inverse matrix is built up column by column.

B.1.2 Singular value decomposition

Singular value decomposition (SVD) is a popular method used for solving linear least-squares problems ($Ax = b$, where the number of rows of A is greater than the number of columns). It gives the user information about how ill conditioned the set of equations is and allows the user to step in and remove sources of numerical inaccuracy.

As with LU decomposition, the first step decomposes the A matrix into more than one matrix (Eq. B.16). In this case, an $M \times N$ A matrix is decomposed into a column-orthogonal $M \times N$ U matrix, a diagonal $N \times M$ W matrix, and an $N \times N$ orthogonal V matrix.

$$A = UWV^T \tag{B.16}$$

The magnitude of the elements in the W matrix indicates the potential for numerical problems. Zeros on the diagonal indicate singularities. Small values (where *small* is user defined) indicate the

```
/*LU Decomposition
* with partial implicit pivoting
* partial means that the pivoting only happens by row
* implicit means that the pivots are scaled by the maximum value in the row
*/

/*=============================================================================*/
/*LUdecomp
* inputs: A matrix of coefficients
*   n – dimension of A
* outputs: A matrix replaced with L and U diagonal matrices
  (diagonal   values of L = 1)
*   Rowswaps – vector to keep track of row swaps
*   Val – indicator of odd/even number of row swaps
*/
int LUdecomp(float **A,int n,int *rowswaps,int *val)
{
  float epsilon,*rowscale, temp;
  float sum;
  float pvt;
  int   ipvt;
  int   i,j,k;
  rowscale = (float*)malloc(sizeof(float)*n);

  epsilon = 0.00000000001;  /* small value to avoid division by zero */
  *val = 1;                 /* even/odd indicator (valence) */

  /* initialize the rowswap vector to indicate no swaps */
  for (i=0; i<n; i++) rowswaps[i] = i;

  /* for each row, find largest (in absolute value sense) element and
     record in rowscale */
  for (i=0; i<n; i++) {
    temp = fabs(A[i][0]);
    for (j=1; j<n; j++)
     if (fabs(A[i][j]) > temp) temp = fabs(A[i][j]);
    if (temp == 0) return(-1); /* got a row of all zeros – can't deal
                                  with that */
    rowscale[i] = 1/temp; /* later we need to divide by largest
                             element */
  }

  /* loop through the columns of A (and U) */
  for (j=0; j<n; j++) {
```

FIGURE B.2

LU decomposition code.

Continued

```
    /* do the rows down to the diagonal — these don't need a
       division       so no swap */
    for (i=0; i<j; i++) {
      sum = A[i][j];
      for (k=0; k<i; k++) sum = sum - A[i][k]*A[k][j];
      A[i][j] = sum;
    }
    /* do the rows from the diagonal down */
    pvt = 0.0;
    ipvt = -1;
    for (i=j; i<n; i++) {
      sum = A[i][j];
      for (k=0; k<j; k++) sum = sum - A[i][k]*A[k][j];
      A[i][j] = sum;
      /* calculate the scaled value for pivoting consideration */
         temp = rowscale[i]*fabs(sum);
         if (temp >= pvt) {ipvt = i; pvt = temp;}
    }

    /* if a better pivot value is found, interchange the rows */
       if (j != ipvt) {
          for (k=0; k<n; k++) {
            temp = A[ipvt][k];
            A[ipvt][k] = A[j][k];
            A[j][k] = temp;
          }
          *val = -(*val);  /* keep track of even/odd number
                              interchanges */
          rowscale[ipvt] = rowscale[j];  /* and record which
                                            was     swapped */
       }
      rowswaps[j] = ipvt;

      if (A[j][j] == 0.0) A[j][j] = epsilon; /* to guard against
                                                divisions by zero */
      /* now the row is ready for division */
      for (i=j+1; i<n; i++) A[i][j] = A[i][j]/A[j][j];
   }
   return 1;
}

/* ========================================================================= */
/* LUsubstitute
```

FIGURE B.2—CONT'D

```
* inputs: A – matrix holding the L and U matrix values as a result
  of    LUdecomp
*     n – dimension of A
*     Rowswaps – vector holding a record of the row swaps
      performed       in LUdecomp
*     b – vector of right-hand values as in Ax = b
*/
void LUsubstitute(float **A,int n,int *rowswaps,float *b)
{
  int  i,j,ib;
  float  sum;
  int  m;

   /* row swap version */
   ib = -1;
   for (i=0; i<n; i++) {
     m = rowswaps[i];
     sum = b[m];
     b[m] = b[i];
     if (ib != -1) {
        for (j=ib; j<i; j++) sum = sum-A[i][j]*b[j];
     }
     else {
       if (sum != 0.0) ib = i;
     }
     b[i] = sum;
   }

   for (i=n-1; i>=0; i--) {
     sum = b[i];
     for (j=i+1; j<n; j++) sum = sum - A[i][j]*b[j];
       b[i] = sum/A[i][i];
     }
   return;
}
```

FIGURE B.2—CONT'D

potential of numerical instability. It is the user's responsibility to inspect the *W* matrix and zero out values that are small enough to present numerical problems. Once this is done, the matrices can be used to solve the least-squares problem using a back substitution method similar to that used in *LU* decomposition. The code for SVD is available in various software packages and can be found in *Numerical Recipes* [16].

B.2 Geometric computations

A vector (a one-dimensional list of numbers) is often used to represent a point in space or a direction and magnitude in space (e.g., Figure B.3). A slight complication in terminology results because a direction and magnitude in space is also referred to as a *vector*. As a practical matter, this distinction is usually not important. A vector in space has no position, only magnitude and direction. For geometric computations, a matrix usually represents a transformation (e.g., Figure B.4).

B.2.1 Components of a vector

A vector, A, with coordinates (A_x, A_y, A_z) can be written as a sum of vectors, as shown in Equation B.17, in which i, j, k are unit vectors along the principal axes, x, y, and z, respectively.

$$A = A_x i + A_y j + A_z k \tag{B.17}$$

B.2.2 Length of a vector

The *length* of a vector is computed as in Equation B.18. If a vector is of unit length, then $|A| = 1.0$, and it is said to be *normalized*. Dividing a vector by its length, forming a unit-length vector, is said to be *normalizing* the vector.

$$|A| = \sqrt{A_x^2 + A_y^2 + A_z^2} \tag{B.18}$$

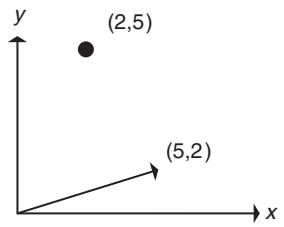


FIGURE B.3

A point and vector in two-space.

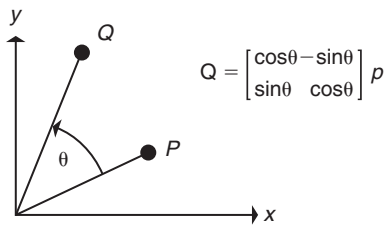


FIGURE B.4

A matrix representing a rotation.

B.2.3 Dot product of two vectors

The *dot product*, or *inner product*, of two vectors is computed as in Equation B.19. The computation is commutative (Eq. B.20) and associative (Eq. B.21). The dot product of a vector with itself results in the square of its length (Eq. B.22). The dot product of two vectors, A and B, is equal to the lengths of the vectors times the cosine of the angle between them (Eq. B.23, Figure B.5). As a result, the angle between two vectors can be determined by taking the arccosine of the dot product of the two normalized vectors (or, if they are not normalized, by taking the arccosine of the dot product divided by the lengths of the two vectors). The dot product of two vectors is equal to zero if the vectors are perpendicular to each other, as in Figure B.6 (or if one or both of the vectors are zero vectors). The dot product can also be used to compute the projection of one vector onto another vector (Figure B.7). This is useful in cases in which the coordinates of a vector are needed in an auxiliary coordinate system (Figure B.8).

$$A \bullet B = A_x B_x + A_y B_y + A_z B_z \tag{B.19}$$

$$A \bullet B = B \bullet A \tag{B.20}$$

$$(A \bullet B) \bullet C = A \bullet (B \bullet C) \tag{B.21}$$

$$A \bullet A = |A|^2 \tag{B.22}$$

$$A \bullet B = |A||B| cos\theta \tag{B.23}$$

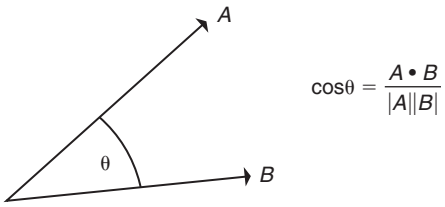


FIGURE B.5

Using the dot product to compute the cosine of the angle between two vectors.

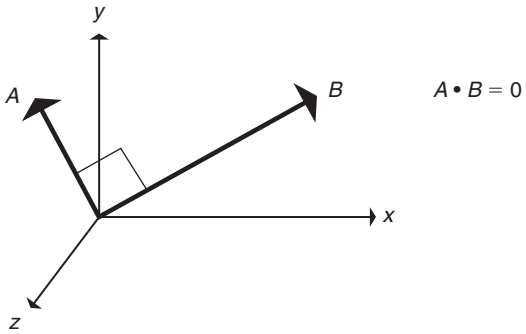


FIGURE B.6

The dot product of perpendicular vectors.

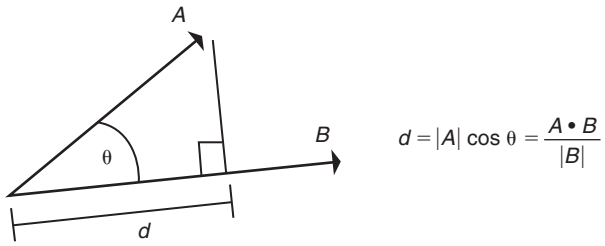


FIGURE B.7

Computing the length of the projection of vector *A* onto vector *B*.

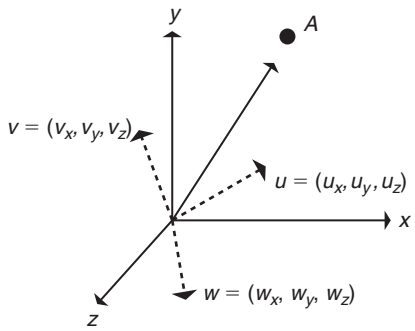


FIGURE B.8

Computing the coordinates of a vector in an auxiliary coordinate system.

B.2.4 Cross-product of two vectors

The *cross-product*, or *outer product*, of two vectors can be defined using the determinant of a 3×3 matrix as shown in Equation B.24, where i, j, and k are unit vectors in the directions of the principal axes. Equation B.25 shows the definition as an explicit equation. The cross-product is not commutative (Eq. B.26), but it is associative (Eq. B.27).

$$A \times B = \begin{vmatrix} i & j & k \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix} \tag{B.24}$$

$$A \times B = (A_y B_z - A_z B_y) i + (A_z B_x - A_x B_z) j + (A_x B_y - A_y B_x) k \tag{B.25}$$

$$A \times B = -(B \times A) = (-A) \times (-B) \tag{B.26}$$

$$A \times (B \times C) = (A \times B) \times C \tag{B.27}$$

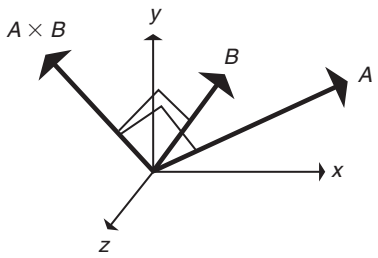


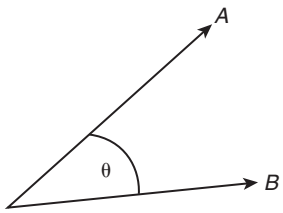
FIGURE B.9

Vector formed by the cross product of two vectors.

The direction of $A \times B$ is perpendicular to both A and B (Figure B.9), and the direction is determined by the right-hand rule (if A and B are in right-hand space). If the thumb of the right hand is put in the direction of the first vector (A) and the index finger is put in the direction of the second vector (B), the cross-product of the two vectors will be in the direction of the middle finger when it is held perpendicular to the first two fingers.

The magnitude of the cross-product is the length of one vector times the length of the other vector times the sine of the angle between them (Eq. B.28). A zero vector will result if the two vectors are colinear or if either vector is a zero vector (Eq. B.29). This relationship is useful for determining the sine of the angle between two vectors (Figure B.10) and for computing the perpendicular distance from a point to a line (Figure B.11).

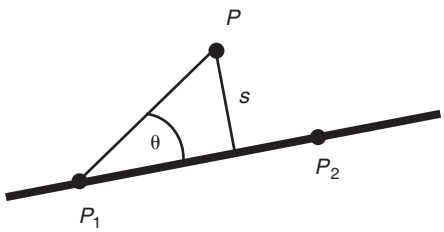
$$|A \times B| = |A||B|\sin\theta \tag{B.28}$$



$$\sin\theta = \frac{|A \times B|}{|A||B|}$$

FIGURE B.10

Using the cross-product to compute the sine of the angle between two vectors.



$$s = |P - P_1|\sin\theta = \frac{|(P - P_1) \times (P_2 - P_1)|}{|P_2 - P_1|}$$

FIGURE B.11

Computing the perpendicular distance from a point to a line.

where θ is the angle from A to B, $0 < \theta < 180$

$$|A \times B| = 0 \text{ if and only if } A \text{ and } B \text{ are colinear,} \tag{B.29}$$

i.e., $\sin\theta = \sin 0 = 0$, or $A = 0$ or $B = 0$.

B.2.5 Vector and matrix routines

Simple vector and matrix routines are given here (Figure B.12). These are used in some of the routines presented elsewhere in this appendix.

Vector routines

Some vector routines can be implemented just as easily as in-line code using C's *#define* construction.

B.2.6 Closest point between two lines in three-space

The intersection of two lines in three-space often needs to be calculated. Because of numerical imprecision, the lines rarely, if ever, actually intersect in three-space. As a result, the computation that needs to be performed is to find the two points, one from each line, at which the lines are closest to each other.

```
/* Vector.c */
typedef   struct xyz_struct {
float     x,y,z;
xyz_td;

}
xyz_td     p;

/* ================================================================== */
/* compute the cross product of two vectors */
xyz_td crossProduct(xyz_td v1,xyz_td v2)
{
  xyz_td     p;

  p.x = v1.y*v2.z - v1.z*v2.y;
  p.y = v1.z*v2.x - v1.x*v2.z;
  p.z = v1.x*v2.y - v1.y*v2.x;
  return     p;
}

/* ================================================================== */
/* compute the dot product of two vectors */
float dotProduct(xyz_td v1,xyz_td v2)
```

FIGURE B.12

Vector routines.

Continued

```
{
  return v1.x*v2.x=v1.y*v2.y+v1.z*v2.z;
}

/* ================================================================== */
/* normalize a vector */
void normalizeVector(xyz_td *v)
{
  float  len;

  len = sqrt(v->x*v->x + v->y*v->y + v->z*v->z);
  v->x /= len;
  v->y /= len;
  v->z /= len;
}

/* ================================================================== */
/* form the vector from the first point to the second */
xyz_td formVector(xyz_td p1, xyz_td p2)
{
  xyz_td    p;

  p.x = p2.x-p1.x;
  p.y = p2.y-p1.y;
  p.z = p2.z-p1.z;
  return p;
}

/* ================================================================== */
/* compute the length of a vector */
float length(xyz_td v)
{
  return sqrt(v.x*v.x+v.y*v.y+v.z*v.z);
}
```

Matrix Routines

```
/* Matrix.c */

/* ================================================================== */
/* Matrix multiplication */
/* C x B = A */
void Matrix4x4MatrixMult (float **C,float **B,float **A)
{
  int  i,j;
```

FIGURE B.12—CONT'D

```
  for (i=0; i<4; i++) {
    for (j=0; j<4; j++) {
      A[i][j] = C[i][0]*B[0][j]+ C[i][1]*B[1][j]+
            C[i][2]*B[2][j]+ C[i][3]*B[3][j];
    }
  }
}

/* ========================================================================= */
/* matrix-vector multiplication */
/* N = M x V */
void Matrix4 x 4Vector4Mult (float **M,float *V,float *N)
{
  N[0] = M[0][0]*V[0]+M[0][1]*V[1]+M[0][2]*V[2]+M[0][3]*V[3];
  N[1] = M[1][0]*V[0]+M[1][1]*V[1]+M[1][2]*V[2]+M[1][3]*V[3];
  N[2] = M[2][0]*V[0]+M[2][1]*V[1]+M[2][2]*V[2]+M[2][3]*V[3];
  N[3] = M[3][0]*V[0]+M[3][1]*V[1]+M[3][2]*V[2]+M[3][3]*V[3];
}

/* ========================================================================= */
/* vector-matrix multiplication */
/* N = V x M */
void Vector4Matrix4 x 4Mult (float *V,float **M,float *N)
{
  N[0] = M[0][0]*V[0]+M[1][0]*V[1]+M[2][0]*V[2]+M[3][0]*V[3];
  N[1] = M[0][1]*V[0]+M[1][1]*V[1]+M[2][1]*V[2]+M[3][1]*V[3];
  N[2] = M[0][2]*V[0]+M[1][2]*V[1]+M[2][2]*V[2]+M[3][2]*V[3];
  N[3] = M[0][3]*V[0]+M[1][3]*V[1]+M[2][3]*V[2]+M[3][3]*V[3];
}

/* ========================================================================= */
/* compute the inverse of a matrix */
void ComputeInverse4 x 4(float **M,float **Minv)
{
  int      rowswaps[4];
  int      val;
  float    b[4];
  int      i,j;
  float**A;

  A = (float **)malloc(sizeof(float *)*4);
  for (i=0; i<4; i++) {
```

FIGURE B.12—CONT'D

```
    A[i] = (float *)malloc(sizeof(float)*4);
  }
  for (i=0; i<4; i++) {
    for (j=0; j<4; j++) {
      A[i][j] = M[i][j];
    }
  }

  LUdecomp(A,4,rowswaps,&val);

  for (i=0; i<4; i++) {
    for (j=0; j<4; j++) b[j] = (i==j) ? 1:0;
    LUsubstitute(A,4,rowswaps,b);
    for (j=0; j<4; j++) Minv[j][i] = b[j];
  }

}
```

FIGURE B.12—CONT'D

The points P_1 and P_2 at which the lines are closest form a line segment perpendicular to both lines (Figure B.13). They can be represented parametrically as points along the lines, and then the parametric interpolants can be solved for by satisfying the equations that state the requirement for perpendicularity (Eq. B.30).

$$
\begin{aligned}
&(P_2 - P_1) \bullet V = 0 \\
&(P_2 - P_1) \bullet W = 0 \\
&(B + tW - (A + sV)) \bullet V = 0 \\
&(B + tW - (A + sV)) \bullet W = 0 \\
&t = \frac{-B \bullet V + A \bullet V + sV \bullet V}{W \bullet V} \\
&t = \frac{-B \bullet W + A \bullet W + sV \bullet W}{W \bullet W} \\
&\frac{-B \bullet V + A \bullet V + sV \bullet V}{W \bullet V} = \frac{-B \bullet W + A \bullet W + sV \bullet W}{W \bullet W} \\
&(-B \bullet V + A \bullet V + sV \bullet V)(W \bullet W) = (-B \bullet W + A \bullet W + sV \bullet W) \cdot (W \bullet V) \\
&s = \frac{(A \bullet W - B \bullet W)(W \bullet V) + (B \bullet V - A \bullet V) \cdot (W \bullet W)}{(V \bullet V)(W \bullet W) - (V \bullet W)^2} \\
&t = \frac{(B \bullet V - A \bullet V)(W \bullet V) + (A \bullet W - B \bullet W)(V \bullet V)}{(W \bullet W)(V \bullet V) - (V \bullet W)^2}
\end{aligned}
\tag{B.30}
$$

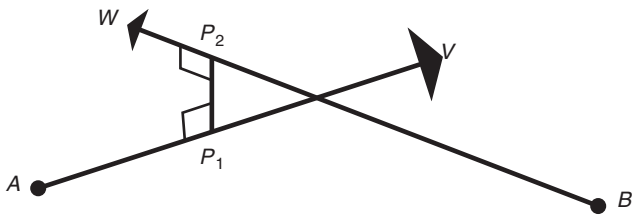


FIGURE B.13

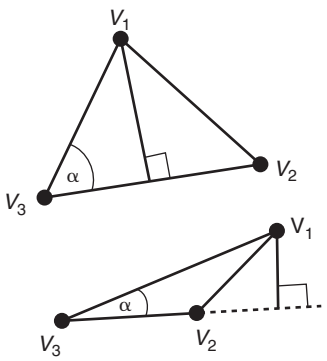
Two lines are closest to each other at points P_1 and P_2.

B.2.7 Area calculations

Area of a triangle

The area of a triangle consisting of vertices V_1, V_2, V_3 is one-half times the length of one edge times the perpendicular distance from that edge to the other vertex. The perpendicular distance from the edge to a vertex can be computed using the cross-product (see Figure B.14). For triangles in two dimensions, the z-coordinates are essentially considered zero and the cross-product computation is simplified accordingly (only the z-coordinate of the cross-product is non-zero).

The signed area of the triangle is required for computing the area of a polygon. In the two-dimensional case, this is done simply by not taking the absolute value of the z-coordinate of the cross-product. In the three-dimensional case, a vector normal to the polygon can be used to indicate the positive direction. The direction of the vector produced by the cross-product can be compared to the normal (using the dot product) to determine whether it is in the positive or negative direction. The length of the cross-product vector can then be computed and the appropriate sign applied to it.



$$\begin{aligned}\text{Area}(V_1, V_2, V_3) &= \frac{1}{2}|V_2 - V_3|\,|V_1 - V_3| \sin\alpha \\ &= \frac{1}{2}|V_2 - V_3|\frac{|(V_1 - V_3) \times (V_2 - V_3)|}{|V_2 - V_3|} \\ &= \frac{1}{2}|(V_1 - V_3) \times (V_2 - V_3)|\end{aligned}$$

FIGURE B.14

Area of a triangle.

Area of a polygon

The area of a polygon can be computed as a sum of the signed areas of simple elements. In the two-dimensional case, the signed area under each edge of the polygon can be summed to form the area (Figure B.15). The area under an edge is the average height of the edge times its width (Eq. B.31, where subscripts are computed modulo $n + 1$).

$$\begin{aligned} \text{Area} &= \sum_{i=1}^{n} \frac{(y_i + y_{i+1})}{2} (x_{i+1} - x_i) \\ &= \frac{1}{2} \sum_{i=1}^{n} (y_i x_{i+1} - y_{i+1} x_i) \end{aligned} \tag{B.31}$$

The area of a polygon can also be computed by using each edge of the polygon to construct a triangle with the origin (Figure B.16). The signed area of the triangle must be used so that edges directed

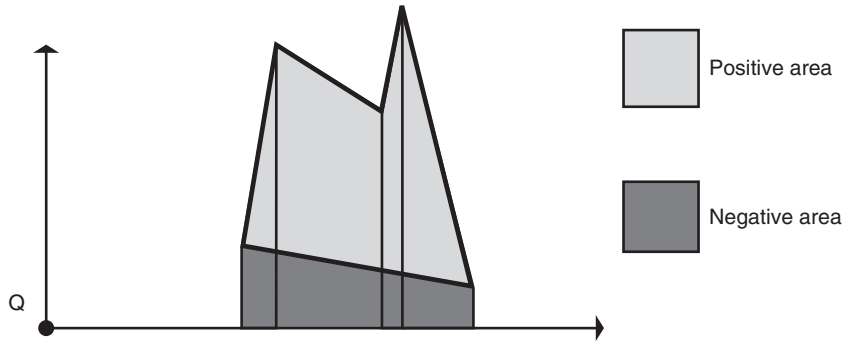
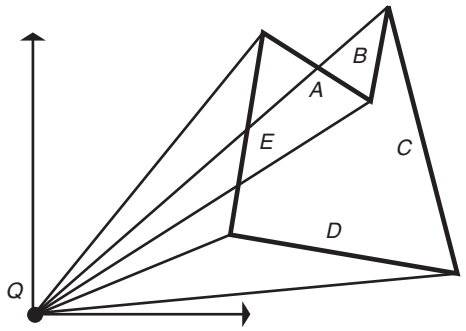


FIGURE B.15

Computing the area of a polygon.



Area of polygon (*A*, *B*, *C*, *D*, *E*) = Area of Triangle (*Q*, *A*)
+ Area of Triangle(*Q*, *B*)
+ Area of Triangle(*Q*, *C*)
+ Area of Triangle(*Q*, *D*)
+ Area of Triangle(*Q*, *E*)

FIGURE B.16

The area of a two-dimensional polygon: the edges are labeled with letters, triangles are constructed from each edge to the origin, and the areas of the triangles are signed according to the direction of the edge with respect to the origin. Area of polygon (*A*, *B*, *C*, *D*, *E*) = Area of Triangle (*Q*, *A*)

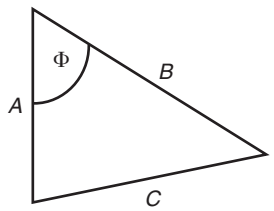
clockwise with respect to the origin cancel out edges directed counterclockwise with respect to the origin. Although this is more computationally expensive than summing the areas under the edges, it suggests a way to compute the area of a polygon in three-space. In the three-dimensional case, one of the vertices of the polygon can be used to construct a triangle with each edge, and the three-dimensional version of the vector equations of Figure B.14 can be used.

B.2.8 The cosine rule

The cosine rule states the relationship between the lengths of the edges of a triangle and the cosine of an interior angle (Figure B.17). It is useful for determining the interior angle of a triangle when the locations of the vertices are known.

B.2.9 Barycentric coordinates

Barycentric coordinates are the coordinates of a point in terms of weights associated with other points. Most commonly used are the barycentric coordinates of a point with respect to vertices of a triangle. The barycentric coordinates (u_1, u_2, u_3) of a point, P, with respect to a triangle with vertices V_1, V_2, V_3 are shown in Figure B.18. Notice that for a point inside the triangle, the coordinates always sum to one. This can be extended easily to any convex polygon by a direct generalization of the equations. However, it cannot be extended to concave polygons.

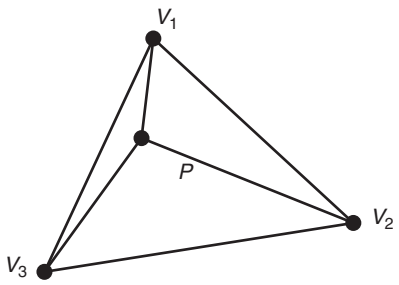


$$|C|^2 = |A|^2 + |B|^2 - 2|A||B|\cos\Phi$$

$$\cos\Phi = \frac{|A|^2 + |B|^2 - |C|^2}{2|A||B|}$$

FIGURE B.17

The cosine rule.



$$u_1 = \frac{\text{Area}(V_1, V_2, V_3)}{\text{Area}(P, V_2, V_3)}$$

$$u_2 = \frac{\text{Area}(P, V_3, V_1)}{\text{Area}(V_1, V_2, V_3)}$$

$$u_3 = \frac{\text{Area}(P, V_1, V_2)}{\text{Area}(V_1, V_2, V_3)}$$

$$P = u_1 V_1 + u_2 V_2 + u_3 V_3$$

FIGURE B.18

The barycentric coordinates of a point with respect to vertices of a triangle.

B.2.10 Computing bounding shapes

Bounding volumes are useful as approximate extents of more complex objects. Often, simpler tests can be used to determine the general position of an object by using bounding volumes, thus saving computation. In computer animation, the most obvious example occurs in testing for object collisions. If the bounding volumes of objects are not overlapping, then the objects themselves must not be penetrating each other. Planar polyhedra are considered here because the bounding volumes can be determined by inspecting the vertices of the polyhedra. Nonplanar objects require more sophisticated techniques. *Axis-aligned bounding boxes* and bounding spheres are relatively easy to calculate but can be poor approximations of the object's shape. Slabs and *oriented bounding boxes* (OBBs) can provide a much better fit. OBBs are rectangular bounding boxes at an arbitrary orientation [7]. For collision detection, bounding shapes are often hierarchically organized into tighter and tighter approximations of the object's space. A *convex hull*, the smallest convex shape bounding the object, provides an even tighter approximation but requires more computation.

Bounding boxes

Bounding box typically refers to a boundary cuboid (or rectangular solid) whose sides are aligned with the principal axes. A bounding box for a collection of points is easily computed by searching for minimum and maximum values for the x-, y-, and z-coordinates. A point is inside the bounding box if its coordinates are between min/max values for each of the three coordinate pairs. While the bounding box may be a good fit for some objects, it may not be a good fit for others (Figure B.19). How well the bounding box approximates the shape of the object is rotationally variant, as shown in Figures B.19b and B.19c.

Bounding slabs

Bounding slabs are a generalization of bounding boxes. A pair of arbitrarily oriented planes are used to bound the object. The orientation of the pair of planes is specified by a user-supplied normal vector.

The normal defines a family of planes that vary according to perpendicular distance to the origin. The planar equation $a\,x + b\,y + c\,z = d$, (a, b, c) represents a vector normal to the plane. If this vector has unit length, then d is the perpendicular distance to the plane. If the length of (a, b, c) is not one, then d is the perpendicular distance scaled by the vector's length. Notice that d is equal to the dot product of the vector (a, b, c) and a point on the plane.

Given a user-supplied normal vector, the user computes the dot product of that vector and each vertex of the object and records the minimum and maximum values (Eq. B.32). The normal vector

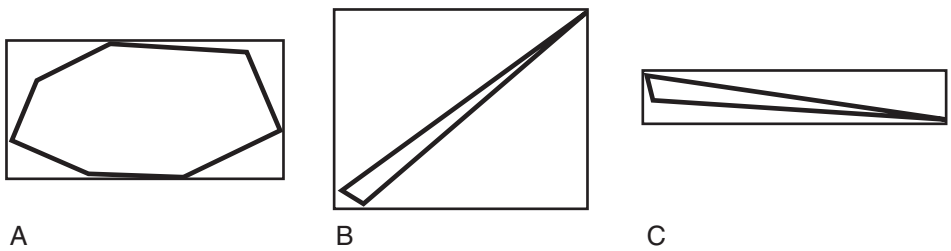


FIGURE B.19

Sample objects and their bounding boxes (in two-dimensional).

and these min/max values define the bounding slab. See Figure B.20 for a two-dimensional diagram illustrating the idea. Multiple slabs can be used to form an arbitrarily tight bounding volume of the convex hull of the polyhedron. A point is inside this bounding volume if the result of the dot product of it and the normal vector is between the corresponding min/max values for each slab (see Figure B.21 for a two-dimensional example).

$$\begin{array}{ll} p = (x, y, z) & \text{vertex} \\ N = (a, b, c) & \text{normal} \\ PgN = d & \text{computing the planar equation constant} \end{array} \tag{B.32}$$

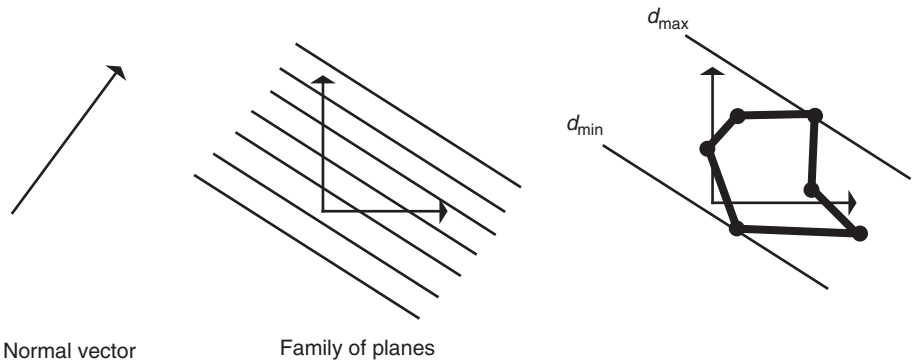


FIGURE B.20

Computing a boundary slab for a polyhedron.

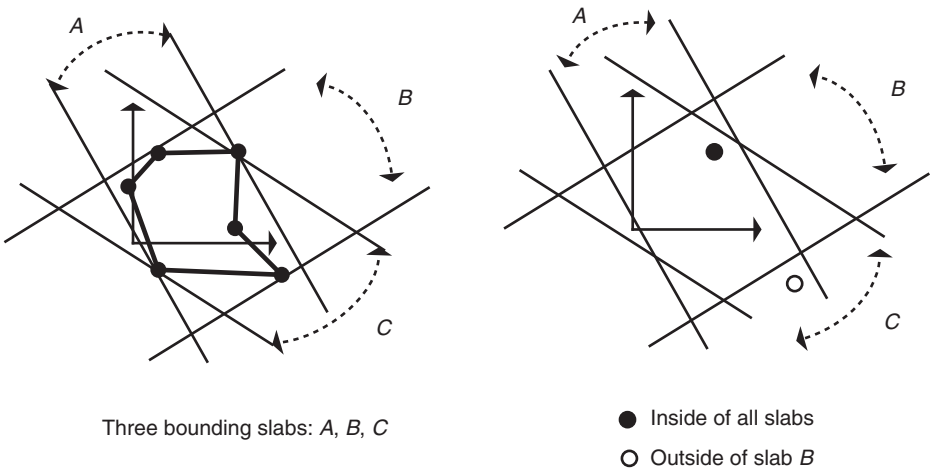


FIGURE B.21

Multiple bounding slabs.

Bounding spheres

Computing the optimal bounding sphere for a set of points can be expensive. However, more tractable approximate methods exist. A quick and fairly accurate method of computing an approximate bounding sphere for a collection of points is to make an initial guess at the bounding sphere and then incrementally enlarge the sphere as necessary by inspecting each point in the set. The description here follows Ritter [17].

The first step is to loop through the points and record the minimum and maximum points in each of the three principal directions. The second step is to use the maximally separated pair of points from the three pairs of recorded points and create an initial approximation of the bounding sphere. The third step is, for each point in the original set, to adjust the bounding sphere as necessary to include the point. Once all the points have been processed, a near-optimal bounding sphere has been computed (Figure B.22). This method is fast and it is easy to implement (Figure B.23).

Convex hull

The convex hull of a set of points is the smallest convex polyhedron that contains all the points. A simple algorithm for computing the complex hull is given here, although more efficient techniques exist. Since this is usually a one-time code for an object (unless the object is deforming), this is a case where efficiency can be traded for ease of implementation. Refer to Figure B.24.

1. Find a point on the convex hull by finding the point with the largest y-coordinate. Refer to the point found in this step as P_1.
2. Construct an edge on the convex hull by using the following method. Find the point that, when connected with P_1, makes the smallest angle with the horizontal plane passing through P_1. Use L to refer to the line from P_1 to the point. Finding the smallest sine of the angle is equivalent to finding the smallest angle. The sine of the angle between L and the horizontal plane passing through P_1 is equal to the cosine of the angle between L and the vector $(0, -1, 0)$. The dot product of these two vectors is used to compute the cosine of the angle between them. Refer to the point found in this step as P_2, and refer to the line from P_1 to P_2 as L.
3. Construct a triangle on the convex hull by the following method. First, construct the plane defined by L and a horizontal line perpendicular to L at P_1. The horizontal line is constructed according to $(L \times (0, -1, 0))$. All of the points are below this plane. Find the point that, when connected with P_1, makes the smallest angle with this plane. Use K to refer to the line from P_1 to the point. The sine of the angle between K and the plane is equal to the cosine of the angle between K and the

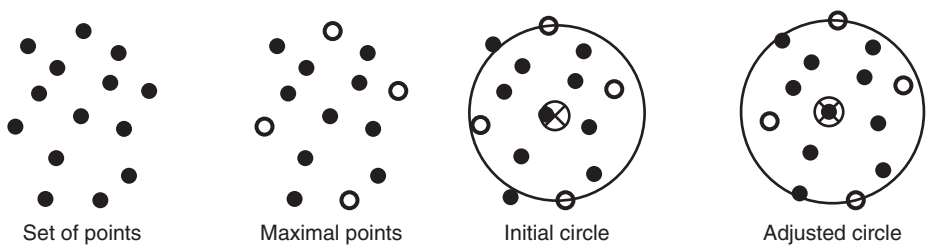


FIGURE B.22

Computing a bounding circle for a set of points.

downward-pointing normal vector of the plane. This normal can be computed as $N = (L \times (0, -1, 0)) \times L$ (in right-hand space). Refer to the point found in this step as P_3. The triangle on the convex hull is defined by these three points. A consistent ordering of the points should be used so that they, for example, compute an outward-pointing normal vector ($N = (P_3 - P_1) \times (P_2 - P_1), N_y > 0.0$) in right-hand space. Initialize the list of convex hull triangles with this triangle and its outward-pointing normal vector. Mark each of its three edges as *unmatched* to indicate that the triangle that shares the edge has not been found yet.

4. Search the current list of convex hull triangles and find an *unmatched* edge. Construct the triangle of the convex hull that shares this edge by the following method. Find the point that, when connected by a line from a point on the edge, makes the smallest angle with the plane of the triangle while creating a dihedral angle (interior angle between two faces measured at a shared edge) greater than 90 degrees. The dihedral angle can be computed using the angle between the normals of the two triangles. When the point has been found, add the triangle defined by this point and the marked edge to the list of convex hull triangles and the unmarked edge. The rest of the unmarked edges in the list

```
/*
** Bounding Sphere Computation
*/
void boundingSphere(xyz_td *pnts,int n, xyz_td *cntr, float *radius)
{
  int   i,minxi,maxxi,minyi,maxyi,minzi,maxzi,p1i,p2i;
  float  minx,maxx,miny,maxy,minz,maxz;
  float  diam2,diam2x,diam2y,diam2z,rad,rad2;
  float  dx,dy,dz;
  float  cntrx,cntry,cntrz;
  float  delta;
  float  dist,dist2;
  float  newrad,newrad2;
  float  newcntrx,newcntry,newcntrz;

  /* step one: find minimal and maximal points in each of 3
     principal    directions */
  minxi = 0; minx = pnts[0].x; maxxi = 0; maxx = pnts[0].x;
  minyi = 0; miny = pnts[0].y; maxyi = 0; maxy = pnts[0].y;
  minzi = 0; minz = pnts[0].z; maxzi = 0; maxz = pnts[0].z;
  for (i=1; i<n; i++) {
    if (pnts[i].x < minx) { minx = pnts[i].x; minxi = i; }
    if (pnts[i].x > maxx) { maxx = pnts[i].x; maxxi = i; }
    if (pnts[i].y < miny) { miny = pnts[i].y; minyi = i; }
```

FIGURE B.23

Bounding sphere code.

Continued

```
 if (pnts[i].y > maxy) { maxy = pnts[i].y; maxyi = i; }
 if (pnts[i].z < minz) { minz = pnts[i].z; minzi = i; }
 if (pnts[i].z > maxz) { maxz = pnts[i].z; maxzi = i; }
}

/* step two: find maximally separated points from the 3 pairs; use
   to initialize sphere */
/* find maximally separated points by comparing the distance squared
   between points */
dx = pnts[minxi].x - pnts[maxxi].x;
dy = pnts[minxi].y - pnts[maxxi].y;
dz = pnts[minxi].z - pnts[maxxi].z;
diam2x = dx*dx + dy*dy + dz*dz;
dx = pnts[minyi].x - pnts[maxyi].x;
dy = pnts[minyi].y - pnts[maxyi].y;
dz = pnts[minyi].z - pnts[maxyi].z;
diam2y = dx*dx + dy*dy + dz*dz;
dx = pnts[minzi].x - pnts[maxzi].x;
dy = pnts[minzi].y - pnts[maxzi].y;
dz = pnts[minzi].z - pnts[maxzi].z;
diam2z = dx*dx + dy*dy + dz*dz;
diam2 = diam2x; p1i = minxi; p2i = maxxi;
if (diam2y>diam2) { diam2 = diam2y; p1i=minyi; p2i=maxyi; }
if (diam2z>diam2) { diam2 = diam2z; p1i=minzi; p2i=maxzi;}
/* center of initial sphere is average of two points */
cntrx = (pnts[p1i].x+pnts[p2i].x)/2;
cntry = (pnts[p1i].y+pnts[p2i].y)/2;
cntrz = (pnts[p1i].z+pnts[p2i].z)/2;
/* calculate radius and radius squared of initial sphere - from
   diameter squared*/
rad2 = diam2/4;
rad = sqrt(rad2);
printf("maximally separated pair: (%f,%f,%f):(%f,%f,%f),%f\n",
  pnts[p1i].x,pnts[p1i].y,pnts[p1i].z,
  pnts[p2i].x,pnts[p2i].y,pnts[p2i].z,diam2);
printf("initial center: (%f,%f,%f)\n",cntrx,cntry,cntrz);
printf("initial diam2: %f\n",diam2);
printf("initial radius, radius2 = %f,%f\n",rad,rad2);

/* third step: now step through the set of points and adjust
   bounding    sphere as necessary */
for (i=0; i<n; i++) {
  dx = pnts[i].x - cntrx;
```

FIGURE B.23—CONT'D

```
      dy = pnts[i].y - cntry;
      dz = pnts[i].z - cntrz;
      dist2 = dx*dx + dy*dy + dz*dz;     /* distance squared of old
                                            center to pnt */
      if (dist2 > rad2) {                /* need to update sphere if this
                                            point is outside old radius*/
         dist = sqrt(dist2);
         /* new radius is average of current radius and distance from
            center to pnt */
         newrad = (rad + dist)/2;
         newrad2 = newrad*newrad;
         printf("new radius = %f\n",newrad);
         delta = dist - newrad;      /* distance from old center to new
                                        center */
         /* delta/dist and rad/dist are weights of pnt and old center to
            compute new center */
         newcntrx = (newrad*cntrx+delta*pnts[i].x)/dist;
         newcntry = (newrad*cntry+delta*pnts[i].y)/dist;
         newcntrz = (newrad*cntrz+delta*pnts[i].z)/dist;

         /* test to see if new radius and center contain the point */
         /* this test should only fail by an epsilon due to numeric
            imprecision */
         dx = pnts[i].x - newcntrx;
         dy = pnts[i].y - newcntry;
         dz = pnts[i].z - newcntrz;
         dist2 = dx*dx + dy*dy + dz*dz;
         if (dist2 > newrad2) {
            printf("ERROR by %lf\n",((double)(dist2))-newrad2);
            printf("  center - radius: (%f,%f,%f) -  %f\n",cntrx,cntry,
                      cntrz,rad);
            printf(" New center - radius: (%f,%f,%f) - %f\n",
             newcntrx,newcntry,newcntrz,newrad);

         }
         cntrx = newcntrx;
         cntry = newcntry;
         cntrz = newcntrz;
         rad = newrad;
         rad2 = rad*rad;

       }
   }
```

FIGURE B.23—CONT'D

```
    *radius = rad;
    cntr->x = cntrx;
    cntr->y = cntry;
    cntr->z = cntrz;
    return;
}
```

FIGURE B.23—CONT'D

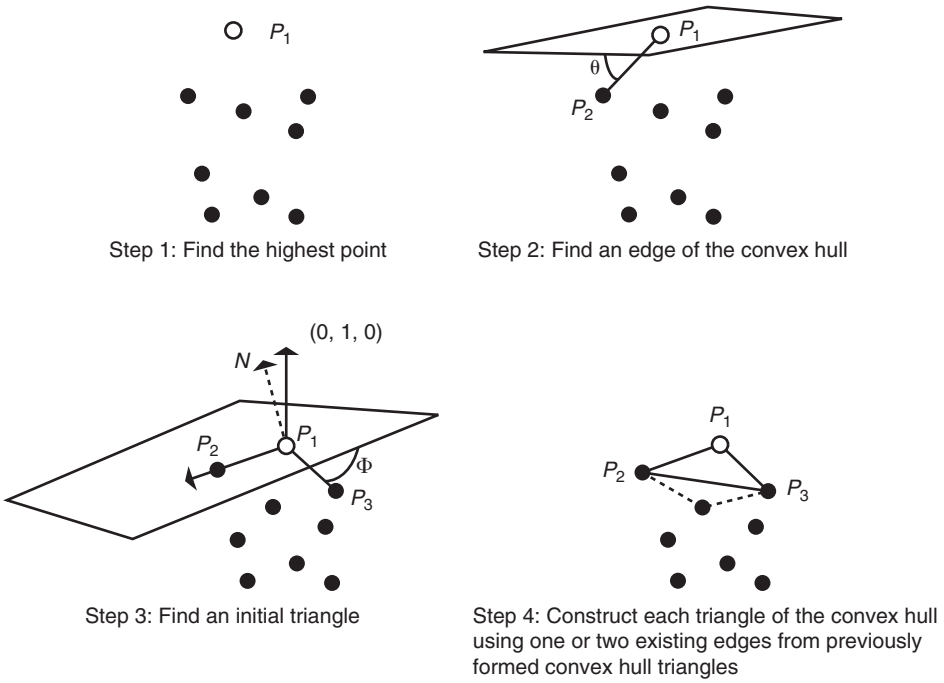


FIGURE B.24

Computing the convex hull.

must be searched to see if the two other edges of the new triangle already occur in the list. If either of the new edges does not have a match in the list, then it should be marked as unmatched. Otherwise, mark the edge as *matched*. Now, go back through the list of convex hull triangles and look for another *unmatched* edge and repeat the procedure to construct a new triangle that shares that edge. When there are no more *unmatched* edges in the list of convex hull triangles, the hull has been constructed.

Step 4 in the previous algorithm does not handle the case in which there are more than three coplanar vertices. To handle these cases, instead of forming a triangle, form a convex hull polygon for the coplanar points. This is done similarly in two and three dimensions. First, collect all of the coplanar points

(all of the points within some epsilon of being coplanar) into a set. Second, initialize the current edge to be the unmatched edge of step 4 and add the first point of the current edge to the set of coplanar points. Third, iteratively find the point in the set of coplanar points that makes the smallest angle with the current edge as measured from the first point of the edge to the second point of the edge to the candidate point. When the point is found, remove it from the set of coplanar points and make the new current edge the edge from the second point of the old current edge to the newly found point. Continue iterating until the first point of the original unmatched edge is found. This will complete the convex hull polygon, which is then added to the list of convex hull polygons; its newly formed edges are processed as in step 4 (see Figure B.25).

```
/* ConvexHull.c
 * This code uses a brute force algorithm to construct the convex hull
 * and does not handle more than three coplanar points
 */

#include "Vector.h"

typedef struct conHullTri_struct {
  int   pi[3];
  int   matched[3];
  xyz_td  normal;
  struct conHullTri_struct *next;
} conHullTri_td;

conHullTri_td       *chtList;

/* ========================================================================== */
/* CONVEX HULL */
int ConvexHull(xyz_td *pntList,int num,int **triangleList,int *numTriangles)
{
  int      i;
  int      p1i,p2i,p3i,pi;
  xyz_td   yaxis;
  xyz_td   v,n,nn,nnn,v1,v2;
  float    t,t1,t2;
  int      count;
  conHullTri_td *chtPtr,*chtPtrTail,*chtPtrNew,*chtPtrA;
  int        done;
  int        *triList;
  int      dummy,notError;
```

FIGURE B.25

Convex hull code.

Continued

```
yaxis.x = 0; yaxis.y = 1; yaxis.z = 0;

/* find the highest point */
p1i = 0; t = pntList[0].y;
for (i=1; i<num; i++) {
    if (pntList[i].y > t) { p1i=i; t=pntList[i].y;}
 }

 /* find point that makes minimum angle with horizontal plane */
 p2i = (p1i==0) ? 1:0;
 v = formVector(pntList[p1i],pntList[p2i]);
 normalizeVector(&v);
 t = v.y;
 if (t>0.0) {
    printf(" ERROR - found higher point\n");
    scanf("%d",&dummy);
    return 1;
 }
 for (i=p2i+1; i<num; i++) {
    if (i!=p1i) {
       v = formVector(pntList[p1i],pntList[i]);
       normalizeVector(&v);
       t1 = v.y;
       if (t1 > t) {p2i = i; t = t1;}
    }
 }

 /* find point that makes triangle with minimum angle with horizontal
    plane through edge */
 v1 = formVector(pntList[p1i],pntList[p2i]);

 if ((p1i!=0) && (p2i!=0)) p3i=0;
 else if ((p1i!=1) && (p2i!=1)) p3i=1;
 else p3i=2;
 v2 = formVector(pntList[p2i],pntList[i]);
 n = crossProduct(v2,v1);
 normalizeVector(&n);
 if (n.y < 0) {
    n.x = -n.x; n.y = -n.y; n.z = -n.z;
 }
 for (i=p3i+1; i<num; i++) {
  if ((i!=p1i)&&(i!=p2i)) {
      v = formVector(pntList[p2i],pntList[i]);
```

FIGURE B.25—CONT'D

```
    nn = crossProduct(v1,v);
    normalizeVector(&nn);
    if (nn.y < 0) { nn.x = -nn.x; nn.y = -nn.y; nn.z = -nn.z; }
    if (nn.y>n.y) {
    p3i=i;
    n.x = nn.x; n.y = nn.y; n.z = n.z;
  }
 }
}

/* compute outward-pointing normal vector in right-hand space for
   clockwise triangle */
/* recalculate the normal vector */
v1 = formVector(pntList[p1i],pntList[p2i]);
v2 = formVector(pntList[p2i],pntList[p3i]);
n = crossProduct(v2,v1);
normalizeVector(&n);
if (n.y<0) {
   n.x = -n.x; n.y = -n.y; n.z = -n.z;
   pi = p1i; p1i = p2i; p2i = pi;
}

/* make a convex hull entry */
count = 1;
chtPtr = (conHullTri_td *)malloc(sizeof(conHullTri_td));
if (chtPtr == NULL) {
   printf("unsuccessful memory allocation 1\n");
   scanf("%d",&dummy);
   return 1;
}
chtPtr->pi[0] = p1i;
chtPtr->pi[1] = p2i;
chtPtr->pi[2] = p3i;
chtPtr->matched[0] = FALSE;
chtPtr->matched[1] = FALSE;
chtPtr->matched[2] = FALSE;
chtPtr->normal = n;

/* initialize the convex hull triangle list with the triangle */
chtList = chtPtr;
chtPtr->next = NULL;
chtPtrTail = chtPtr;
```

FIGURE B.25—CONT'D

```
/* check and make sure all vertices are 'underneath' the initial
   triangle */
for (i=0; i<num; i++) {
  if ((i!=chtPtr->pi[0]) &&
     (i!=chtPtr->pi[1]) &&
     (i!=chtPtr->pi[2]) ) {
     v = formVector(pntList[chtPtr->pi[0]],pntList[i]);
     t = dotProduct(v,n);
  if (t>0.0) {
     /* ERROR - point above initial triangle */
     printf("ERROR - found a point above initial triangle  (%d)\n",i);
     return 1;
  }
 }
}

/* now loop through the convex hull triangle list and process
   unmatched edges */
done = FALSE;
chtPtr = chtList;
while (chtPtr!=NULL) {
 /* look for first unmatched edge */
 if ((!(chtPtr->matched[0])) ||
    (!(chtPtr->matched[1])) ||
    (!(chtPtr->matched[2])) ) {

    /* set it now as matched, and record 3 points with unmatched as
       first two */
    if (!(chtPtr->matched[0])) {
       p1i=chtPtr->pi[0]; p2i=chtPtr->pi[1]; p3i=chtPtr->pi[2];
       chtPtr->matched[0] = TRUE;
    }
    else if (!(chtPtr->matched[1])) {
     p1i=chtPtr->pi[1]; p2i=chtPtr->pi[2]; p3i=chtPtr->pi[0];
     chtPtr->matched[1] = TRUE;
    }
    else if (!(chtPtr->matched[2])) {
     p1i=chtPtr->pi[2]; p2i=chtPtr->pi[0]; p3i=chtPtr->pi[1];
     chtPtr->matched[2] = TRUE;
    }

    /* get info of triangle of unmatched edge */
    n.x = chtPtr->normal.x;
```

FIGURE B.25—CONT'D

```
    n.y = chtPtr->normal.y;
    n.z = chtPtr->normal.z;
    v1=formVector(pntList[p2i],pntList[p1i]);

    /* find new vertex which, with unmatched edge, makes    triangle */
    /* whose normal is closest to normal of triangle of unmatched
       edge */
    pi = -1;
    for (i=0; i<num; i++) {
     if ((i!=p1i)&&(i!=p2i)&&(i!=p3i)) {
        v=formVector(pntList[p1i],pntList[i]);
    /* test to see if point is above triangle */
    t1 = dotProduct(v,n);
    if (t1>0) {
       /* ERROR - point above initial triangle */
      printf("ERROR - found a point above initial triangle   (%d)\n",i);
      return 1;
    }
    /* compute normal of proposed new triangle */
    nn = crossProduct(v,v1);
    normalizeVector(&nn);
    /* test for concave corner */
    nnn = crossProduct(n,nn);
    t2 = dotProduct(nnn,v1);
    if (t2<0.0) {
       printf("ERROR - concave corner found\n");
       return 1;
    }
    /* compute angle made by faces (=angle made by normals)    */
    t1 = dotProduct(n,nn);
    /* printf(" %d: dot product of normals: %f\n",i,t1); */
    /* printf(" normal for comparison: %f %f    %f\n",n.x,n.y,n.z); */
    /* printf(" normal: %f %f %f\n",nn.x,nn.y,nn.z); */
    /* save smallest angle (largest cosine) */
    if (pi==-1) {pi=i; t=t1;}
    else if (t1>t) {pi=i; t=t1;}
   }
  }

  /* check and make sure all vertices are 'underneath' this    triangle */
  v=formVector(pntList[p1i],pntList[pi]);
  nn = crossProduct(v,v1);
  normalizeVector(&nn);
```

FIGURE B.25—CONT'D

```
        for (i=0; i<num; i++) {
         if ((i!=p2i) &&
            (i!=p1i) &&
            (i!=pi) ) {
            v = formVector(pntList[p1i],pntList[i]);
            t = dotProduct(v,nn);
            if (t>0.0) {
             /* ERROR - point above new triangle */
             printf("ERROR - found a point above new triangle    (%d)\n",i);
             return 1;
        }
      }
    }

  /* search for p2i-pi or pi-pi1 already in database - error    condition */
   chtPtrA = chtList; notError = TRUE;
   while ((chtPtrA!=NULL)&&notError) {
    if((chtPtrA->pi[0]==p1i)&&(chtPtrA->pi[1]==pi)) notError =    FALSE;
    else if((chtPtrA->pi[1]==p1i)&&(chtPtrA->pi[2]==pi))  notError = FALSE;
    else if((chtPtrA->pi[2]==p1i)&&(chtPtrA->pi[0]==pi))  notError = FALSE;
    else if((chtPtrA->pi[0]==pi)&&(chtPtrA->pi[1]==p2i))  notError = FALSE;
    else if((chtPtrA->pi[1]==pi)&&(chtPtrA->pi[2]==p2i))  notError = FALSE;
    else if((chtPtrA->pi[2]==pi)&&(chtPtrA->pi[0]==p2i))  notError = FALSE;
    else chtPtrA = chtPtrA->next;
    /* end while */
   if (!notError) {
     printf("ERROR - duplicating edge    (%d,%d,%d)\n",p1i,p2i,pi);
     return 1;
   }

   /* add p1i, p2i, pi */
   count++;
   chtPtrNew = (conHullTri_td *)malloc(sizeof(conHullTri_td));
   if (chtPtrNew == NULL) {
      printf(" unsuccessful memory allocation 2\n");
      return 1;
   }

   chtPtrTail->next = chtPtrNew;
   chtPtrNew->pi[0] = p2i;
   chtPtrNew->pi[1] = p1i;
   chtPtrNew->pi[2] = pi;
```

FIGURE B.25—CONT'D

```
chtPtrNew->matched[0] = TRUE;
chtPtrNew->matched[1] = FALSE;
chtPtrNew->matched[2] = FALSE;
chtPtrNew->normal.x = nn.x;
chtPtrNew->normal.y = nn.y;
chtPtrNew->normal.z = nn.z;
chtPtrNew->next = NULL;
chtPtrTail = chtPtrNew;

/* search for p2i-pi or pi-p1i already in database in reverse   order */
chtPtrA = chtList;
while (chtPtrA!=NULL) {
 if (!chtPtrA->matched[0]&&(chtPtrA-  >pi[0]==pi)&&(chtPtrA->
    pi[1]==p1i)) {
    chtPtrA->matched[0] = TRUE;
    chtPtrNew->matched[1] = TRUE;
 }
 else if (!chtPtrA->matched[1]&&(chtPtrA-  > pi[1]==pi)&&
 (chtPtrA->pi[2]==p1i)) {
   chtPtrA->matched[1] = TRUE;
   chtPtrNew->matched[1] = TRUE;
 }
 else if (!chtPtrA->matched[2]&&(chtPtrA-  > pi[2]==pi)&&
 (chtPtrA->pi[0]==p1i)) {
   chtPtrA->matched[2] = TRUE;
   chtPtrNew->matched[1] = TRUE;
  }
  else if (!chtPtrA->matched[0]&&(chtPtrA-  > pi[0]==p2i)&&
  (chtPtrA->pi[1]==pi)) {
   chtPtrA->matched[0] = TRUE;
   chtPtrNew->matched[2] = TRUE;
  }
  else if (!chtPtrA->matched[1]&&(chtPtrA-  > pi[1]==p2i)&&
  (chtPtrA->pi[2]==pi)) {
   chtPtrA->matched[1] = TRUE;
   chtPtrNew->matched[2] = TRUE;
  }
  else if (!chtPtrA->matched[2]&&(chtPtrA-  >pi[2]==p2i)&&
  (chtPtrA->pi[0]==pi)) {
   chtPtrA->matched[2] = TRUE;
```

FIGURE B.25—CONT'D

```
        vchtPtrNew->matched[2] = TRUE;
       }
       else {
         chtPtrA = chtPtrA->next;
        }
    } /* end while */

 } /* end endif */
 else {
   chtPtr = chtPtr->next;
  }
 }

triList = (int *)malloc(sizeof(int)*count*3);
chtPtr = chtList;
for (i=0; i<count; i++) {
 if (chtPtr==NULL) {
    printf("ERROR: count %d doesn't match data structure\n",count);
    return 1;
   }
   triList[3*i] = chtPtr->pi[0];
   triList[3*i+1] = chtPtr->pi[1];
   triList[3*i+2] = chtPtr->pi[2];
   chtPtr=chtPtr->next;
  }
  *numTriangles = count;
  *triangleList = triList;
 return 0;
}

void printCHTlist(conHullTri_td *chtPtr)
{
printf("CHT list\n");
while (chtPtr!=NULL) {
 printf("%d:%d:%d ; %d:%d:%d ; %f,%f,%f\n",
  chtPtr->pi[0],chtPtr->pi[1],chtPtr->pi[2],
  chtPtr->matched[0],chtPtr->matched[1],chtPtr->matched[2],
  chtPtr->normal.x,chtPtr->normal.y,chtPtr->normal.z);
  chtPtr = chtPtr->next;
 }
}
```

FIGURE B.25—CONT'D

B.3 Transformations

B.3.1 Transforming a point using vector-matrix multiplication

Vector-matrix multiplication is usually how the transformation of a point is represented. Because a vector is just an $N \times 1$ matrix, vector-matrix multiplication is actually a special case of matrix-matrix multiplication. Vector-matrix multiplication is usually performed by premultiplying a column vector by a matrix. This is equivalent to post-multiplying a row vector by the transpose of that same matrix. Both notations are encountered in the graphics literature, but use of the column vector is more common. The examples in Equations B.33 and B.34 use a 4×4 matrix and a point in three-space using homogeneous coordinates, consistent with what is typically encountered in graphics applications.

$$\begin{bmatrix} Q_x \\ Q_y \\ Q_z \\ Q_w \end{bmatrix} = Q = MP = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{B.33}$$

$$\begin{aligned} \begin{bmatrix} Q_x & Q_y & Q_z & Q_w \end{bmatrix} = Q^T &= P^T M^T \\ &= \begin{bmatrix} P_x & P_y & P_z & 1 \end{bmatrix} \begin{bmatrix} M_{11} & M_{21} & M_{31} & M_{41} \\ M_{12} & M_{22} & M_{32} & M_{42} \\ M_{13} & M_{23} & M_{33} & M_{43} \\ M_{14} & M_{24} & M_{34} & M_{44} \end{bmatrix} \end{aligned} \tag{B.34}$$

B.3.2 Transforming a vector using vector-matrix multiplication

In addition to transforming points, it is also often useful to transform vectors, such as normal vectors, from one space to another. However, the computations used to transform a vector are different from those used to transform a point. Vectors have direction and magnitude but do not have a position in space. Thus, for example, a pure translation has no effect on a vector. If the transformation of one space to another is a pure rotation and uniform scale, then those transformations can be applied directly to the vector. However, it is not so obvious how to apply transformations that incorporate nonuniform scale.

The transformation of a vector can be demonstrated by considering a point, P, which satisfies a planar equation (Eq. B.35). Note that (a, b, c) represents a vector normal to the plane. Showing how to transform a planar equation will, in effect, show how to transform a vector. The point is transformed by a matrix, M (Eq. B.36). Because the transformations of rotation, translation, and scale preserve planarity, the transformed point, P', will satisfy some new planar equation, N', in the transformed space (Eq. B.37). Substituting the definition of the transformed point, Equation B.36, into Equation B.37 produces Equation B.38. If the transformed planar equation is equal to the original normal postmultiplied by the inverse of the transformation matrix (Eq. B.39), then Equation B.37 is satisfied, as shown by Equation B.40. The transformed normal vector is, therefore, (a', b', c').

$$ax + by + cz + d = 0$$

$$\begin{bmatrix} a & b & c & d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0 \tag{B.35}$$

$$N^T P = 0$$

$$P' = MP \tag{B.36}$$

$$N'^T P' = 0 \tag{B.37}$$

$$N'^T MP = 0 \tag{B.38}$$

$$N'^T = N^T M^{-1} \tag{B.39}$$

$$N^T M^{-1} MP = N^T P = 0 \tag{B.40}$$

In order to transform a vector (a, b, c), treat it as a normal vector for a plane passing through the origin $[a, b, c, 0]$ and post-multiply it by the inverse of the transformation matrix (Eq. B.39). If it is desirable to keep all vectors as column vectors, then Equation B.41 can be used.

$$N' = \left(N'^T\right)^T = \left(N^T M^{-1}\right)^T = \left(M^{-1}\right)^T N \tag{B.41}$$

B.3.3 Axis-angle rotations

Given an axis of rotation $A = [\, a_x \;\; a_y \;\; a_z]$ of unit length and an angle θ to rotate by (Figure B.26), the rotation matrix M can be formed by Equation B.42. This is a more direct way to rotate a point around an axis, as opposed to implementing the rotation as a series of rotations about the global axes.

$$\hat{A} = \begin{bmatrix} a_x a_x & a_x a_y & a_x a_z \\ a_y a_x & a_y a_y & a_y a_z \\ a_z a_x & a_z a_x & a_z a_z \end{bmatrix}$$

$$A^* = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_z & a_x & 0 \end{bmatrix} \tag{B.42}$$

$$M = \hat{A} + \cos\theta\left(I - \hat{A}\right) + \sin\theta A^*$$

$$p' = MP$$

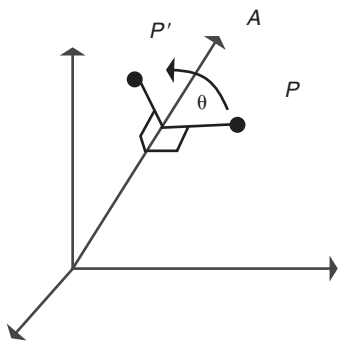


FIGURE B.26

Axis-angle rotation.

B.3.4 Quaternions

Quaternions are discussed in Chapter 2.4 and Chapter 3.1. The equations from those chapters, along with additional equations, are collected here to facilitate the discussion.

Quaternion arithmetic

Quaternions are four-tuples and can be considered as a scalar combined with a vector (Eq. B.43). Addition and multiplication are defined for quaternions by Equations B.44 and B.45, respectively. Quaternion multiplication is associative (Eq. B.46), but it is not commutative (Eq. B.47). The magnitude of a quaternion is computed as the square root of the sum of the squares of its four components (Eq. B.48). Quaternion multiplication has an identity (Eq. B.49) and an inverse (Eq. B.50). The inverse distributes over quaternion multiplication similarly to how the inverse distributes over matrix multiplication (Eq. B.51). A quaternion is normalized by dividing it by its magnitude (Eq. B.52). Equation B.52a is used to compute the quaternion that represents the rotation r such that q represents the half-way rotation between p and r.

$$q = [s, x, y, z] = [s, v] \tag{B.43}$$

$$[s_1, v_1] + [s_2, v_2] = [s_1 + s_2, v_1 + v_2] \tag{B.44}$$

$$[s_1, v_1][s_2, v_2] = [s_1\, s_2 - v_1 \,.\, v_2, s_1 v_2 + s_2 v_1 + v_1 \times v_2] \tag{B.45}$$

$$(q_1 q_2) q_3 = q_1 (q_2 q_3) \tag{B.46}$$

$$q_1 q_2 \neq q_2 q_1 \tag{B.47}$$

$$\|q\| = \sqrt{s^2 + x^2 + y^2 + z^2} \tag{B.48}$$

$$[s, v][1, (0, 0, 0)] = [s, v] \tag{B.49}$$

$$\begin{aligned} q^{-1} &= (1/\|q\|)^2 [s, -v] \\ q^{-1} q &= q q^{-1} = [1, (0, 0, 0)] \end{aligned} \tag{B.50}$$

$$(pq)^{-1} = q^{-1} p^{-1} \tag{B.51}$$

$$qunit = q/(\|q\|) \tag{B.52}$$

$$r = 2(p \cdot q)q - p \tag{B.52a}$$

Rotations by quaternions

A point in space is represented by a vector quantity in quaternion form by using a zero scalar value (Eq. B.53). A quaternion can be used to rotate a vector using quaternion multiplication (Eq. B.54). Compound rotations can be implemented by premultiplying the corresponding quaternions (Eq. B.55), similar to what is routinely done when rotation matrices are used. As should be expected, compounding a rotation with its inverse produces the identity transformation for vectors (Eq. B.56). An axis-angle rotation is represented by a unit quaternion, as shown in Equation B.57. Any scalar multiple of a quaternion represents the same rotation. In particular, the negation of a quaternion (negating each of its four components, $-q = [-s, -x, y, -z]$) represents the same rotation that the original quaternion represents (Eq. B.58).

$$v = [0, x, y, z] \tag{B.53}$$

$$v' = \mathrm{Rot}(v) = qvq^{-1} \tag{B.54}$$

$$\begin{aligned}\mathrm{Rot}_q\big(\mathrm{Rot}_p(v)\big) &= q(pvp^{-1})q^{-1} \\ &= ((qp)v(p^{-1}q^{-1})) \\ &= ((qp)v(qp^{-1})) \\ &= \mathrm{Rot}_{qp}(v)\end{aligned} \tag{B.55}$$

$$\begin{aligned}\mathrm{Rot}^{-1}(\mathrm{Rot}(v)) &= q^{-1}(qvq^{-1})q \\ &= (q^{-1}q)v(q^{-1}q) = v\end{aligned} \tag{B.56}$$

$$\mathrm{Rot}_{[\theta,x,y,z]} \equiv [\cos(\theta/2),\ \sin(\theta/2)(x,y,z)] \tag{B.57}$$

$$\begin{aligned}-q &\equiv \mathrm{Rot}_{[-\theta,-x,-y,-z]} \\ &= [\cos(-\theta/2),\ \sin(-\theta/2)(-(x,y,z))] \\ &= [\cos(\theta/2), -\sin(\theta/2)(-(x,y,z))] \\ &= [\cos(\theta/2),\ \sin(\theta/2)(x,y,z)] \\ &\equiv \mathrm{Rot}_{[\theta,x,y,z]} \\ &\equiv q\end{aligned} \tag{B.58}$$

Conversions

It is often useful to convert back and forth between rotation matrices and quaternions. Often, quaternions are used to interpolate between orientations, and the result is converted to a rotation matrix so as to combine it with other matrices in the display pipeline.

Given a unit quaternion ($q = [s, x, y, z]$, $s^2 + x^2 + y^2 + z^2 = 1$), one can easily determine the corresponding rotation matrix by rotating the three unit vectors that correspond to the principal axes. The rotated vectors are the columns of the equivalent rotation matrix (Eq. B.59).

$$M_q = \begin{bmatrix} 1-2y^2-2z^2 & 2xy-2sz & 2xz+2sy \\ 2xy+2sz & 1-2x^2-2z^2 & 2yz-2sx \\ 2xy-2sy & 2yz+2sx & 1-2x^2-2y^2 \end{bmatrix} \tag{B.59}$$

Given a rotation matrix, one can use the definitions for the terms of the matrix in Equation B.59 to solve for the elements of the equivalent unit quaternion. The fact that the unit quaternion has a magnitude of one ($s^2 + x^2 + y^2 + z^2 = 1$) makes it easy to see that the diagonal elements sum to $4 \cdot s^2 - 1$. Summing the diagonal elements of the matrix in Equation B.60 results in Equation B.61. The diagonal elements can also be used to solve for the remaining terms (Eq. B.62). The square roots of these last equations can be avoided if the off-diagonal elements are used to solve for x, y, and z at the expense of testing for a divide by an s that is equal to zero (in which case Eq. B.62 can be used).

$$\begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & m_{2,2} \end{bmatrix} \tag{B.60}$$

$$s = \frac{\sqrt{m_{0,0} + m_{1,1} + m_{2,2} + 1}}{2} \tag{B.61}$$

$$\begin{aligned}m_{0,0} &= 1 - 2y^2 - 2z^2 \\ &= 1 - 2(y^2 + z^2) \\ &= 1 - 2(1 - x^2 - s^2) \\ &= -1 + 2x^2 + 2s\end{aligned} \tag{B.62}$$

$$x = \frac{\sqrt{m_{0,0} + 1 - 2s^2}}{2}$$
$$y = \frac{\sqrt{m_{1,1} + 1 - 2s^2}}{2} \tag{B.62a}$$
$$z = \frac{\sqrt{m_{2,2} + 1 - 2s^2}}{2}$$

B.4 Denevit and Hartenberg representation for linked appendages

B.4.1 Denavit-Hartenberg notation

The Denavit-Hartenberg (DH) notation is a particular way of describing the relationship of a parent coordinate frame to a child coordinate frame. This convention is commonly used in robotics and often adopted for use in computer animation. Each frame is described relative to an adjacent frame by four parameters that describe the position and orientation of a child frame in relation to its parent's frame.

For revolute joints, the z-axis of the joint's frame corresponds to the axis of rotation (prismatic joints are discussed in the following paragraph). The link associated with the joint extends down the x-axis of the frame. First, consider a simple configuration in which the joints and the axes of rotation are coplanar. The distance down the x-axis from one joint to the next is the *link length*, a_i. The *joint angle*, θ_{i+1}, is specified by the rotation of the $i + 1$ joint's x-axis, x_{i+1}, about its z-axis relative to the ith frame's x-axis direction, x_i (see Figure B.27).

Nonplanar configurations can be represented by including the two other DH parameters. For this general case, the x-axis of the ith joint is defined as the line segment perpendicular to the z-axes of the ith and $i + 1$ frames. The *link twist* parameter, α_i, describes the rotation of the $i + 1$ frame's z-axis about this perpendicular relative to the z-axis of the ith frame. The *link offset* parameter, d_{i+1}, specifies the distance along the z-axis (rotated by α_i) of the $i + 1$ frame from the ith x-axis to the $i + 1$ x-axis (see Figure B.28).

Notice that the parameters associated with the ith joint do not all relate the ith frame to the $i + 1$ frame. The link length and link twist relate the ith and $i + 1$ frames; the link offset and joint rotation relate the $i - 1$ and ith frames (see Table B.1).

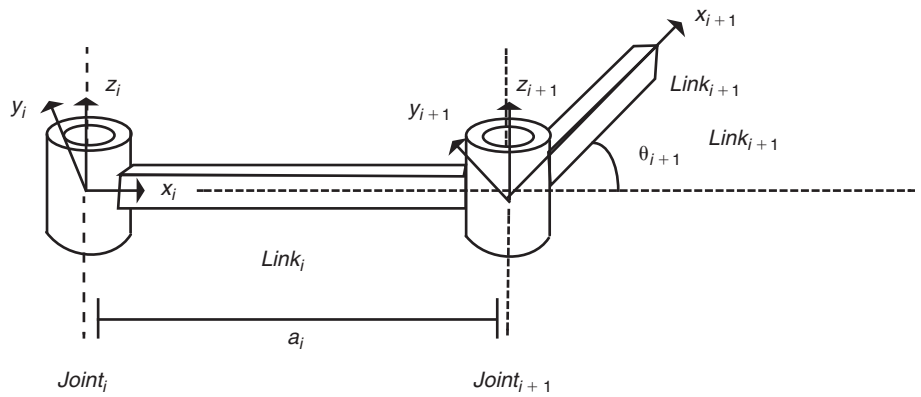


FIGURE B.27

DH parameters for planar joints.

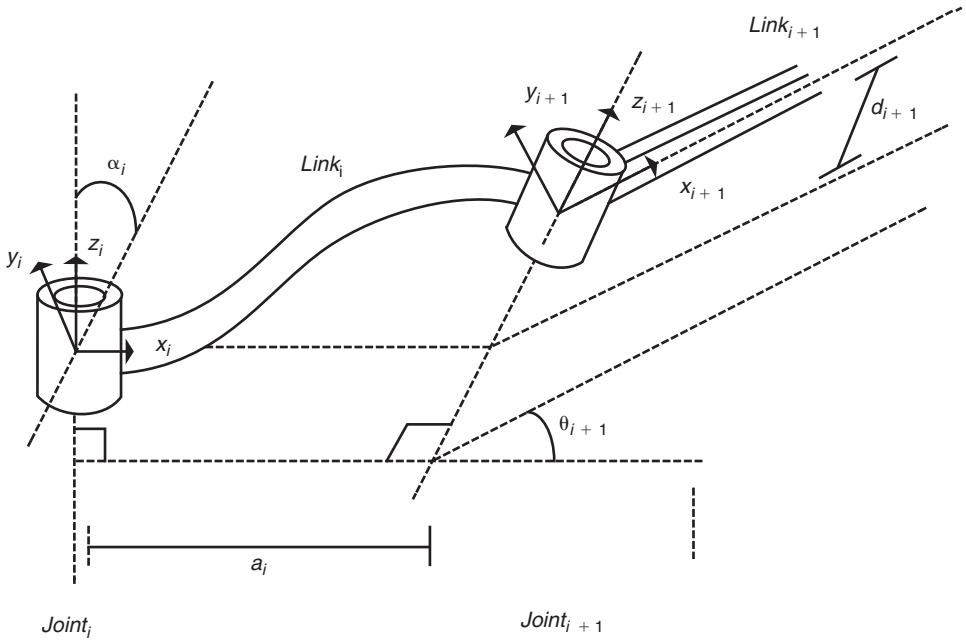


FIGURE B.28

DH parameters.

Table B.1 DH Joint Parameters for Joint i

Name	Symbol	Description
Link offset	d_i	Distance from x_{i-1} to x_i along z_i
Joint angle	θ_i	Angle between x_{i-1} and x_i about z_i
Link length	a_i	Distance from z_i to z_{i+1} along x_i
Link twist	α_i	Angle between z_i and z_{i+1} about x_i

Stated another way, the parameters that describe the relationship of the $i+1$ frame to the ith frame are a combination of ith joint parameters and $i+1$ joint parameters. The parameters can be paired off to define two *screw transformations*, each of which consists of a translation and rotation relative to a single axis. The offset (d_{i+1}) and angle (θ_i+1) are the translation and rotation of the $i+1$ joint relative to the ith joint with respect to the ith joint's z-axis. The length (a_i) and twist (α_i) are the translation and rotation of the $i+1$ joint with respect to the ith joint's x-axis (see Table B.2). The transformation of the $i+1$ joint's frame from the ith frame can be constructed from a series of transformations, each of which corresponds to one of the DH parameters. As an example, consider a point, V_{i+1}, whose coordinates are given in the coordinate system of joint $i+1$. To determine the point's coordinates in terms of the coordinate system of joint i, the transformation shown in Equation B.63 is applied.

In Equation B.63, T and R represent translation and rotation transformation matrices respectively; the parameter specifies the amount of rotation or translation, and the subscript specifies the axis involved. The matrix M maps a point defined in the $i+1$ frame into a point in the ith frame. By forming the M matrix and its inverse associated with each pair of joints, one can convert points from one frame to another, up and down the hierarchy.

Table B.2 Parameters That Relate the *i*th Frame and the $i + 1$ Frame

Name	Symbol	Description	Screw Transformation
Link offset	d_{i+1}	Distance from x_i to x_{i+1} along z_{i+1}	Relative to z_{i+1}
Joint angle	θ_{i+1}	Angle between x_i and x_{i+1} about z_{i+1}	Relative to z_{i+1}
Link length	a_i	Distance from z_i to z_{i+1} along x_i	Relative to x_i
Link twist	α_i	Angle between z_i and z_{i+1} about x_i	Relative to x_i

B.4.2 A simple example

Consider the simple three-joint manipulator of Figure B.29. The DH parameters are given in Table B.3. The linkage is planar, so there are no displacement parameters and no twist parameters. Each successive frame is described by the joint angle and the length of the link.

$$V_i = T_x(\alpha_i)R_X(\alpha_i)T_Z(d_i{+}_1)R_Z(\theta_i{+}_1)V_{i+1}$$

$$R_Z(\theta_i{+}_1) = \begin{bmatrix} \cos(\theta_i{+}_1) & (-\sin(\theta_i{+}_1)) & 0 & 0 \\ \sin(\theta_i{+}_1) & \cos(\theta_i{+}_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_Z(d_{i+1}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_{i+1} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_X(\alpha_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_i) & -\sin(\alpha_i) & 0 \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.63}$$

$$T_X(ai) = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$V_i = M_i^{i+1}V_i + 1$$

$$M_i^{i+1} = \begin{bmatrix} \cos(\theta_{i+1}) & -\sin(\theta_{i+1}) & 0 & a_i \\ \cos(\alpha_i)\sin(\theta_{i+1}) & \cos(\alpha_i)\cos(\theta_{i+1}) & -\sin(\alpha_i) & (-d_{i+1})\sin(\alpha_i) \\ \sin(\alpha_i)\sin(\theta_{i+1}) & \sin(\alpha_i)\cos(\theta_{i+1}) & \cos(\alpha_i) & d_{i+1}\cos(\alpha_i) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

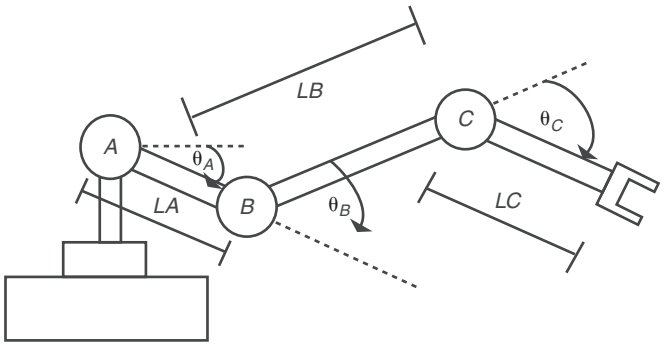


FIGURE B.29

Simple manipulator using three revolute joints.

Table B.3 Parameters for Three-revolute Joint Armature

Joint/Parameter	Link Displacement	Joint Angle	Link Length	Link Twist
A	0	θ_A	0	0
B	0	θ_B	*LA*	0
C	0	θ_C	*LB*	0

B.4.3 Including a ball-and-socket joint

Some human joints are conveniently modeled using a ball-and-socket joint. Consider an armature with a hinge joint, followed by a ball-and-socket joint followed by another hinge joint, as shown in Figure B.30.

The DH notation can represent the ball-and-socket joint by three single degree of freedom (DOF) joints with zero-length links between them (see Figure B.31).

Notice that in a default configuration with joint angles set to zero, the DH model of the ball-and-socket joint is in a gimbal lock position (incrementally changing two of the parameters results in rotation about the same axis). The first and third DOFs of that joint are aligned. The *z*-axes of these joints are colinear because the links between them are zero length and the two link twist parameters relating them are 90 degrees. This results in a total of 180 degrees and thus aligns the axes. As a consequence, the representation of the ball-and-socket joint is usually initialized with the middle of the three joint angles set to 90 degrees (see Table B.4).

B.4.4 Constructing the frame description

Because each frame's displacement and joint angle are defined relative to the previous frame, a Frame_0 is defined so that the Frame_1 displacement and angle can be defined relative to it. Frame_0 is typically defined so that it coincides with Frame_1 with zero displacement and zero joint angle. Similarly, because the link of the last frame does not connect to anything, the *x*-axis of the last frame is chosen so that it coincides with the *x*-axis of the previous frame when the joint angle is zero; the origin of the *n*th frame is

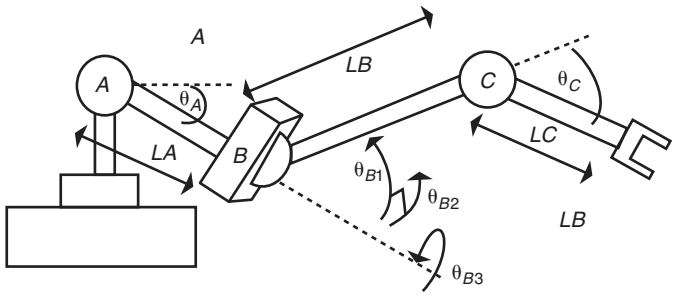


FIGURE B.30

Incorporating a ball-and-socket joint.

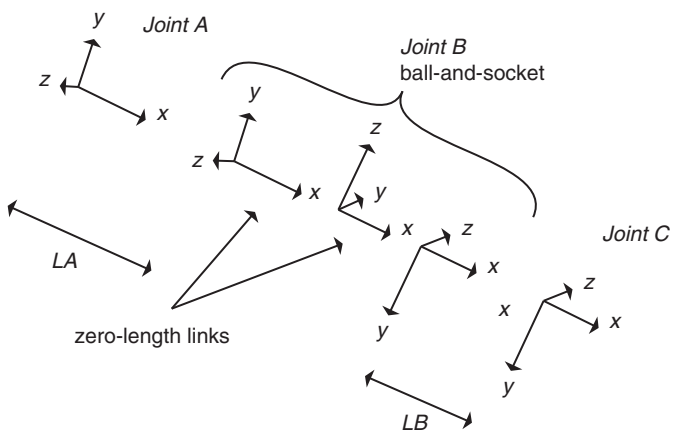


FIGURE B.31

Coordinate axes induced by the DH representation of a ball-and-socket joint.

Table B.4 Joint Parameters for Ball-and-socket Joint

Joint/Parameter	Link Displacement	Joint Angle	Link Length	Link Twist
A	0	θ_A	0	0
*B*1	0	θ_{B1}	*LA*	90
*B*2	0	$90 + \theta_{B2}$	0	90
*B*3	0	θ_{B3}	0	0
C	0	θ_C	*LB*	0

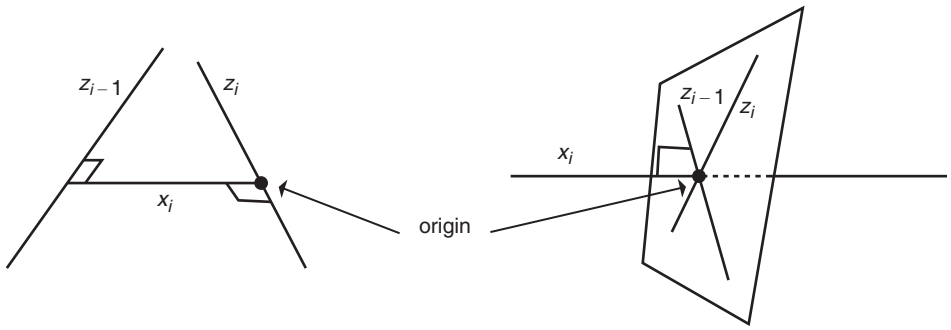


FIGURE B.32

Determining the origin and *x*-axis of the *i*th frame.

chosen as the intersection of the *x*-axis of the previous frame and the joint axis when the displacement is zero.

The following procedure can be used to construct the frames for intermediate joints.

1. For each joint, identify the axis of rotation for revolute joints and the axis of displacement for prismatic joints. Refer to this axis as the *z*-axis of the joint's frame.
2. For each adjacent pair of joints, the *i*th − 1 and *i*th for *i* from 1 to *n*, construct the common perpendicular between the *z*-axes or, if they intersect, the perpendicular to the plane that contains them. Refer to the intersection of the perpendicular and the *i*th frame's *z*-axis (or the point of intersection of the two axes) as the origin of the *i*th frame. Refer to the perpendicular as the *x*-axis of the *i*th frame (see Figure B.32).
3. Construct the *y*-axis of each frame to be consistent with the right-hand rule (assuming right-hand space).

B.5 Interpolating and approximating curves

This section covers many of the basic terms and concepts needed to interpolate values in computer animation. It is not a complete treatise of curves but an overview of the important ones. While many of the terms and concepts discussed are applicable to functions in general, they are presented as they relate to functions having to do with the practical interpolation of points in Euclidean space as typically used in computer animation applications. For more complete discussions of the topics contained here, see, for example, Mortenson [14], Rogers and Adams [18], Farin [4], and Bartels, Beatty, and Barsky [1].

B.5.1 Equations: some basic terms

For present purposes, there are three types of equations: *explicit*, *implicit*, and *parametric*. *Explicit equations* are of the form $y = f(x)$. The explicit form is good for generating points because it generates a value of y for any value of x put into the function. The drawback of the explicit form is that it is dependent on the choice of coordinate axes, and it is ambiguous if there is more than one y for a given

x (such as $y = \sqrt{x}$, in which an input value of 4 would generate values of 2 or -2). *Implicit equations* are of the form $f(x, y) = 0$. The implicit form is good for testing to see if a point is on a curve because the coordinates of the point can easily be put into the equation for the curve and checked to see if the equation is satisfied. The drawback of the implicit form is that generating a series of points along a curve is often desired, and implicit forms are not generative. *Parametric equations* are of the form $x = f(t)$, $y = g(t)$. For any given value of t, a point (x, y) is generated. This form is good for generating a sequence of points as ordered values of t are given. The parametric form is also useful because it can be used for multivalued functions of x, which are problematic for explicit equations.

Functions can be classified according to the terms contained in them. Functions that contain only variables raised to a power are *polynomial* functions. If the highest power is one, then the function is *linear*. If the highest power is two, then the function is *quadratic*. If the highest power is three, then it is *cubic*. The highest power of a polynomial function of a single variable is referred to as the *degree* of the polynomial. If the function is not a simple polynomial but rather contains sines, cosines, log, or a variety of other functions, then it is called *transcendental*. In computer graphics, the most commonly encountered type of function is the cubic polynomial.

Continuity refers to how well behaved the curve is in a mathematical sense. For a value arbitrarily close to an x_0 if the function is arbitrarily close to $f(x_0)$, then it has *positional*, or *zeroth-order*, continuity (C^0) at that point. If the slope of the curve (or the first derivative of the function) is continuous, then the function has *tangential*, or *first-order*, continuity (C^1). This is extended to all of the function's derivatives, although for purposes of computer animation the concern is with first-order continuity or, possibly, *second-order*, or *curvature*, continuity (C^2). Polynomials are infinitely continuous.

If a curve is pieced together from individual curve segments, one can speak of *piecewise properties*—the properties of the individual pieces. For example, a sequence of straight line segments, sometimes called a *polyline* or a *wire*, is piecewise linear. A major concern regarding piecewise curves is the continuity conditions at the junctions of the curve segments. A junction has C^0 *continuity*, or is C^0 *continuous*, if one curve segment begins where the previous segment ends. This is referred to as *zeroth-order*, or *positional*, continuity at the junction. If the beginning tangent of one curve segment is the same as the ending tangent of the previous curve segment, then there is *first-order*, or *tangential*, continuity at the junction. The junction is C^1 continuous if it has tangential and positional continuity. If the beginning curvature of one curve segment is the same as the ending curvature of the previous curve segment, then there is *second-order*, or *curvature*, continuity at the junction. The junction is C^2 continuous if it has curvature, tangential, and positional continuity. Typically, computer animation is not concerned with continuity beyond second order.

Sometimes in discussions of the continuity at segment junctions, a distinction is made between *parametric continuity* and *geometric continuity* (e.g., [14]). So far the discussion has concerned parametric continuity. Geometric continuity is less restrictive. First-order parametric continuity, for example, requires that the ending tangent vector of the first segment be the same as the beginning tangent vector of the second. First-order geometric continuity, on the other hand, requires that only the direction of the tangents be the same, and it allows the magnitudes of the tangents to be different. Similar definitions exist for higher order geometric continuity. One distinction worth mentioning is that parametric continuity is sensitive to the rate at which the parameter varies relative to the length of the curve traced out. Geometric continuity is not sensitive to this rate.

When a curve is constructed from a set of points and the curve passes through the points, it is said to *interpolate* the points. However, if the points are used to control the general shape of the curve, with the curve not necessarily passing through them, then the curve is said to *approximate the points*. *Interpolation* is also used generally to refer to all approaches for constructing a curve from a set of points. For a given interpolation technique, if the resulting curve is guaranteed to lie within the convex hull of the set of points, then it is said to have the *convex hull property*.

B.5.2 Simple linear interpolation: geometric and algebraic forms

Simple linear interpolation is given by Equation B.64 and shown in Figure B.33. Notice that the interpolants, $1 - u$ and u, sum to one. This property ensures that the interpolating curve (in this case a straight line) falls within the convex hull of the geometric entities being interpolated (in this simple case the convex hull is the straight line itself).

$$P(u) = (1 - u)P_0 + uP_1 \tag{B.64}$$

Using more general notation, one can rewrite Equation 64 as in Equation B.65. Here, F_0 and F_1 are called blending functions. This is referred to as the geometric form because the geometric information, in this case P_0 and P_1, is explicit in the equation.

$$P(u) = F_0(u)P_0 + F_1(u)P_1 \tag{B.65}$$

The linear interpolation equation can also be rewritten as in Equation B.66. This form is typical of polynomial equations in which the terms are collected according to coefficients of the variable raised to a power. It is more generally written as Equation B.67. In this case there are only linear terms. This way of expressing the equation is referred to as the *algebraic* form.

$$P(u) = (P_1 - P_0)u + P_0 \tag{B.66}$$

$$P(u) = a_1u + a_0 \tag{B.67}$$

Alternatively, both of these forms can be put in a *matrix representation*. The geometric form becomes Equation B.68 and the algebraic form becomes Equation B.69. The geometric form is useful in situations in which the geometric information (the points defining the curve) needs to be frequently updated or replaced. The algebraic form is useful for repeated evaluation of a single curve for different values of the parameter. The fully expanded form is shown in Equation B.70. The curves discussed next can all be written in this form. Of course, depending on the actual curve type, the U (variable), M (coefficient), and B (geometric information) matrices will contain different values.

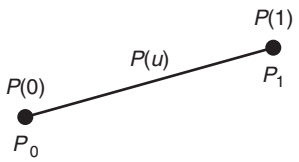


FIGURE B.33

Linear interpolation.

$$P(u) = \begin{bmatrix} F_0(u) \\ F_1(u) \end{bmatrix} [P_0 P_1] = FB^T \tag{B.68}$$

$$P(u) = [\, u \quad 1 \,] \begin{bmatrix} a_1 \\ a_0 \end{bmatrix} = U^T A \tag{B.69}$$

$$P(u) = [\, u \quad 1 \,] \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \end{bmatrix} = U^T MB = FB = U^T A \tag{B.70}$$

B.5.3 Parameterization by arc length

It should be noted that in general there is not a linear relationship between changes in the parameter u and the distance traveled along a curve (its *arc length*). It happens to be true in the previous example concerning a straight line and the parameter u. However, as Mortenson [14] points out, there are other equations that trace out a straight line in space that are fairly convoluted in their relationship between changes in the parameter and distance traveled. For example, consider Equation B.71, which is linear in P_0 and P_1. That is, it traces out a straight line in space between P_0 and P_1. However, it is nonlinear in u. As a result, the curve is not traced out in a nice monotonic, constant-velocity manner. The nonlinear relationship is evident in most parameterized curves unless special care is taken to ensure constant velocity.

$$P(u) = P_0 + ((1-u)u + u)(P_1 - P_0) \tag{B.71}$$

B.5.4 Computing derivatives

One of the matrix forms for parametric curves, as shown in Equation B.70 for linear interpolation, is $U^T MB$. Parametric curves of any polynomial order can be put into this matrix form. Often, it is useful to compute the derivatives of a parametric curve. This can be done easily by taking the derivative of the U vector. For example, the first two derivatives of a cubic curve, shown in Equation B.72, are easily evaluated for any value of u.

$$\begin{aligned} P(u) &= U^T MB = |u^3\ u^2\ u\ \ 1\,|MB \\ P'(u) &= U'^T MB = |3u^3\ 2u\ 1\ \ 0\,|MB \\ P''(u) &= U''^T MB = |6u\ 2\ \ 0\ \ 0\,|MB \end{aligned} \tag{B.72}$$

B.5.5 Hermite interpolation

Hermite interpolation generates a cubic polynomial from one point to another. In addition to specifying the beginning and ending points (P_i, P_{i+1}), the user needs to supply beginning and ending tangent vectors (P'_i,P'_{i+1}) as well (Figure B.34). The general matrix form for a curve is repeated in Equation B.73, and the Hermite matrices are given in Equation B.74.

$$P(u) = U^T MB \tag{B.73}$$

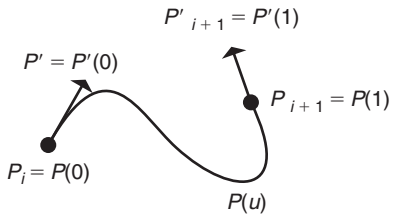


FIGURE B.34

Hermite interpolation.

$$
\begin{aligned}
U^T &= \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \\
M &= \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\
B &= \begin{bmatrix} P_i \\ P_{i+1} \\ P_{i+1} \\ P'_i \\ P'_{i+1} \end{bmatrix}
\end{aligned}
\tag{B.74}
$$

Continuity between beginning and ending tangent vectors of connected segments is ensured by merely using the ending tangent vector of one segment as the beginning tangent vector of the next. A composite Hermite curve (piecewise cubic with first-order continuity at the junctions) is shown in Figure B.35.

Trying to put a Hermite curve through a large number of points, which requires the user to specify all of the needed tangent vectors, can be a burden. There are several techniques to get around this. One is to enforce second-degree continuity. This requirement provides enough constraints so that the user does not have to provide interior tangent vectors; they can be calculated automatically. See Rogers and Adams [18] or Mortenson [14] for alternative formulations. A more common technique is the Catmull-Rom spline.

B.5.6 Catmull-Rom spline

The Catmull-Rom curve can be viewed as a Hermite curve in which the tangents at the interior control points are automatically generated according to a relatively simple geometric procedure (as opposed to the more involved numerical techniques referred to above). For each interior point, P_i, the tangent at

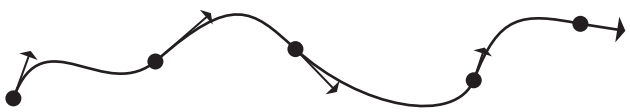


FIGURE B.35

Composite Hermite curve.

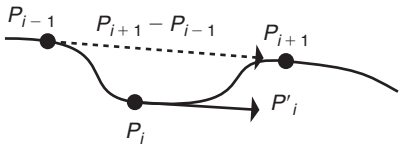


FIGURE B.36

Catmull-Rom spline.

that point, P'_i, is computed as one-half the vector from the previous control point, P_{i-1}, to the following control point, P_{i+1} (Eq. B.75), as shown in Figure B.36.[1] The matrices for the Catmull-Rom curve in general matrix form are given in Equation B.76. A Catmull-Rom spline is a specific type of cardinal spline.

$$P'_i = (1/2)(P_{i+1} - P_{i-1}) \tag{B.75}$$

$$U^T = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

$$M = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \tag{B.76}$$

$$B = \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

For the end conditions, the user can provide tangent vectors at the very beginning and at the very end of the cubic curve. Alternatively, various automatic techniques can be used. For example, the beginning tangent vector can be defined as follows. The vector from the second point (P_1) to the third point (P_2) is subtracted from the second point and used as a virtual point to which the initial tangent is directed. This tangent is computed by Equation B.77. Figure B.37 shows the formation of the initial tangent curve according to the equation, and Figure B.38 shows a curve that uses this technique.

$$P'(0.0) = \frac{1}{2}(P_1 - (P_2 - P_1) - P_0) = \frac{1}{2}(2P_1 - P_2 - P_0) \tag{B.77}$$

A drawback of the Catmull-Rom formulation is that an internal tangent vector is not dependent on the position of the internal point relative to its two neighbors. In Figure B.39, all three positions (Q_i, P_i, R_i) for the ith point would have the same tangent vector.

[1]Farin [4] describes the Catmull-Rom spline curve in terms of a cubic Bezier curve by defining interior control points. Placement of the interior control points is determined by use of an auxiliary knot vector. With a uniform distance between knot values, the control points are displaced from the point to be interpolated by one-sixth of the vector from the previous interpolated point to the following interpolated point. Tangent vectors are three times the vector from an interior control point to the interpolated point. This results in the Catmull-Rom tangent vector described here.

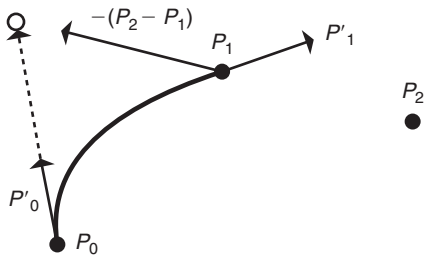


FIGURE B.37

Automatically forming the initial tangent of a Catmull-Rom spline.

FIGURE B.38

Catmull-Rom spline with end conditions using Equation B.77.

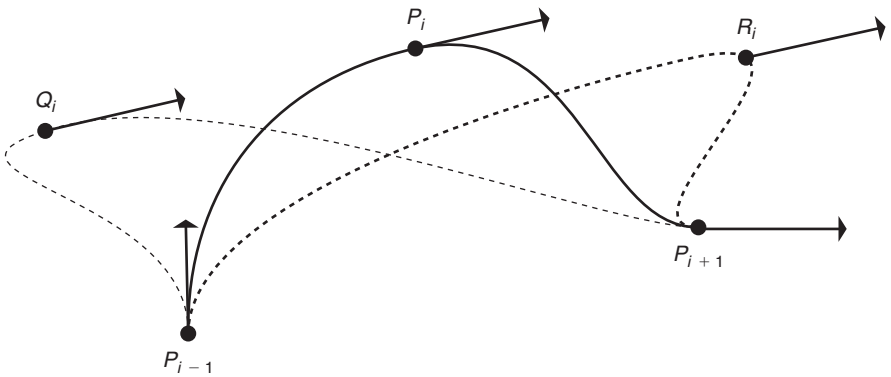


FIGURE B.39

Three curve segments, (P_{i-1}, P_i, P_{i+1}), (P_{i-1}, Q_i, P_{i+1}), and (P_{i-1}, R_i, P_{i+1}), using the standard Catmull-Rom form for computing the internal tangent.

An advantage of Catmull-Rom is that the calculation to compute the internal tangent vectors is extremely simple and fast. However, for each segment the tangent computation is a one-time-only cost. It is then used repeatedly in the computation for each new point in that segment. Therefore, it often makes sense to spend a little more time computing more appropriate internal tangent vectors to obtain a better set of points along the segment. One alternative is to use a vector perpendicular to the plane that bisects the angle made by $P_{i-1} - P_i$ and $P_{i+1} - P_i$ (Figure B.40). This can be computed easily by adding the normalized vector from P_{i-1} to P_i with the normalized vector from P_i to P_{i+1}.

Another modification, which can be used with the original Catmull-Rom tangent computation or with the previously mentioned bisector technique, is to use the relative position of the internal point (P_i)

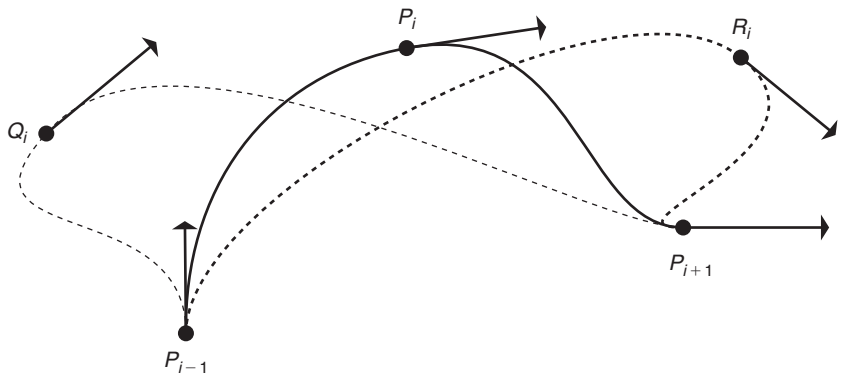


FIGURE B.40

Three curve segments, (P_{i-1}, P_i, P_{i+1}), (P_{i-1}, Q_i, P_{i+1}), and (P_{i-1}, R_i, P_{i+1}), using the perpendicular to the angle bisector for computing the internal tangent.

to independently determine the length of the tangent vector for each segment it is associated with. Thus, a point P_i has an ending tangent vector associated with it for the segment from P_{i-1} to P_i as well as a beginning tangent vector associated with it for the segment P_i to P_{i+1}. These tangents have the same direction but different lengths. This relaxes the C^1 continuity of the Catmull-Rom spline and uses G^1 continuity instead. For example, an initial tangent vector at an interior point is determined as the vector from P_{i-1} to P_{i+1}. The ending tangent vector for the segment P_{i-1} to P_i is computed by scaling this initial tangent vector by the ratio of the distance between the points P_i and P_{i-1} to the distance between points P_{i-1} and P_{i+1}. Referring to the segment between P_{i-1} and P_i as $P_{i-1}(u)$ results in Equation B.78. A similar calculation for the beginning tangent vector of the segment between P_i and P_{i+1} results in Equation B.79. These tangents can be seen in Figure B.41.The computational cost of this approach is only a little more than the standard Catmull-Rom spline and seems to give more intuitive results.

$$P'_{i-1}(1.0) = \frac{|P_i - P_{i-1}|}{|P_{i+1} - P_{i-1}|}(P_{i+1} - P_{i-1}) \tag{B.78}$$

$$P'_{i}(0.0) = \frac{|P_{i+1} - P_i|}{|P_{i+1} - P_{i-1}|}(P_{i+1} - P_{i-1}) \tag{B.79}$$

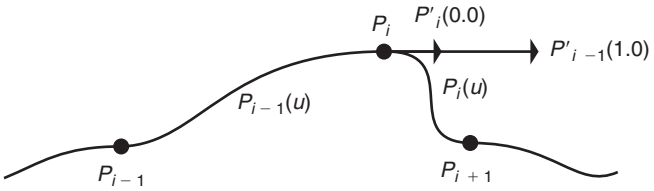


FIGURE B.41

Interior tangents based on relative segment lengths.

B.5.7 Four-point form

Fitting a cubic segment to four points (P_0, P_1, P_2, P_3) assigned to user-specified parametric values (u_0, u_1, u_2, u_3) can be accomplished by setting up the linear system of equations for the points (Eq. B.80) and solving for the unknown coefficient matrix. In the case of parametric values of 0, 1/3, 2/3, and 1, the matrix is given by Equation B.81. However, with this form it is difficult to join segments with C^1 continuity.

$$P(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$\begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} u_0^3 & u_0^2 & u_0 & 1 \\ u_1^3 & u_1^2 & u_1 & 1 \\ u_2^3 & u_2^2 & u_2 & 1 \\ u_3^3 & u_3^2 & u_3 & 1 \end{bmatrix} \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{B.80}$$

$$M = \frac{1}{2} \begin{bmatrix} -9 & 27 & -27 & - \\ 18 & -45 & 36 & -9 \\ -11 & 18 & -9 & 2 \\ 2 & 0 & 0 & 0 \end{bmatrix} \tag{B.81}$$

B.5.8 Blended parabolas

Blending overlapping parabolas to define a cubic segment is another approach to interpolating a curve through a set of points. In addition, the end conditions are handled by parabolic segments, which is consistent with how the interior segments are defined. Blending parabolas results in a formulation that is very similar to Catmull-Rom in that each segment is defined by four points, it is an interpolating curve, and local control is provided. Under the assumptions used here for Catmull-Rom and the blended parabolas, the interpolating matrices are identical.

For each overlapping triple of points, a parabolic curve is defined by the three points. A cubic curve segment is created by linearly interpolating between the two overlapping parabolic segments. More specifically, take the first three points, P_0, P_1, and P_2, and fit a parabola, $P(u)$, through them using the following constraints: $P(0.0) = P_0$, $P(0.5) = P_1$, $P(1.0) = P_2$. Take the next group of three points, P_1, P_2, P_3, which partially overlap the first set of three points, and fit a parabola, $R(u)$, through them using similar constraints: $R(0.0) = P_1$, $R(0.5) = P_2$, $R(1.0) = P_3$. Between points P_1 and P_2 the two parabolas overlap. Reparameterize this region into the range [0.0, 1.0] and linearly blend the two parabolic segments (Figure B.42). The result can be put in matrix form for a cubic curve using the four points as the geometric information together with the coefficient matrix shown in Equation B.82. To interpolate a list of points, this matrix is used by varying U as in previous examples.

$$M = \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \tag{B.82}$$

FIGURE B.42

Parabolic blend segment.

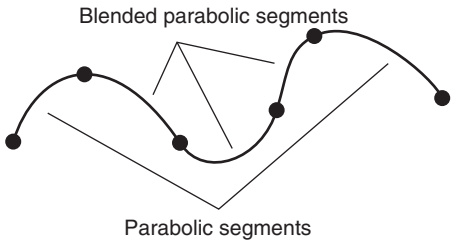


FIGURE B.43

Multiple parabolic blend segments.

End conditions can be handled by constructing parabolic arcs at the very beginning and very end (Figure B.43). For example, referring to the first three points as p_0, p_1, and p_2, Equation B.83 can be used to solve for the constants of the parabolic equation $P(u) = au^2 + bu + c$.

$$\begin{aligned} P(0) &= c = p_0 \\ P(0.5) &= a(0.5)^2 + b(0.5) + c = p_1 \\ P(1.0) &= a + b + c = p_2 \end{aligned} \tag{B.83}$$

This form assumes that all points are equally spaced in parametric space. Often it is the case that even spacing is not present. In such cases, relative chord length can be used to estimate parametric values. The derivation is a bit more involved [18], but the final result can still be formed into a 4×4 matrix and used to produce a cubic polynomial in the interior segments.

B.5.9 Bezier interpolation/approximation

A cubic Bezier curve is defined by the beginning point and the ending point, which are interpolated, and two interior points, which control the shape of the curve. The cubic Bezier curve is similar to the Hermite form. The Hermite form uses beginning and ending tangent vectors to control the shape of the curve; the Bezier form uses auxiliary control points to define tangent vectors. A cubic curve is defined by four points: P_0, P_1, P_2, and P_3. The beginning and ending points of the curve are P_0 and P_3, respectively. The interior control points used to control the shape of the curve and define the beginning and ending tangent vectors are P_1 and P_2 (see Figure B.44). The coefficient matrix for a single cubic Bezier curve is shown in Equation B.84. In the cubic case, $P'(0) = 3(P_1 - P_0)$ and $P'(1) = 3(P_3 - P_2)$.

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{B.84}$$

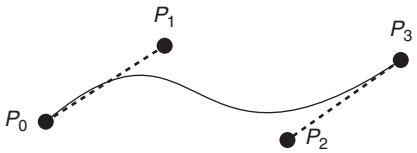


FIGURE B.44

Cubic Bezier curve segment.

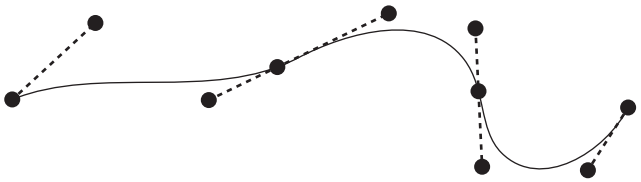


FIGURE B.45

Composite cubic Bezier curve showing tangents and colinear control points.

Continuity between adjacent Bezier segments can be controlled by colinearity of the control points on either side of the shared beginning/ending point of the two curve segments where they join (Figure B.45). In addition, the Bezier curve form allows one to define a curve of arbitrary order. If three interior control points are used, then the resulting curve will be quartic; if four interior control points are used, then the resulting curve will be quintic. See Mortenson [14] for a more complete discussion.

B.5.10 De Casteljau construction of Bezier curves

The de Casteljau method is a way to geometrically construct a Bezier curve. Figure B.46 shows the construction of a point at $u = 1/3$. This method constructs a point u along the way between paired control points (identified by a "1" in Figure B.46). Then points are constructed u along the way between points just previously constructed. These new points are marked "2" in Figure B.46. In the cubic case, in which there were four initial points, there are two newly constructed points. The point on the curve is constructed by going u along the way between these two points. This can be done for any values of u and for any order of curve. Higher order Bezier curves require more iterations to produce the final point on the curve.

B.5.11 Tension, continuity, and bias control

Often an animator wants better control over the interpolation of key frames than the standard interpolating splines provide. For better control of the shape of an interpolating curve, Kochanek [11] suggests a parameterization of the internal tangent vectors based on the three values: tension, continuity, and bias. The three parameters are explained by decomposing each internal tangent vector into an incoming part and an outgoing part. These tangents are referred to as the left and right parts, respectively, and are notated by T_i^L and T_i^R for the tangents at P_i.

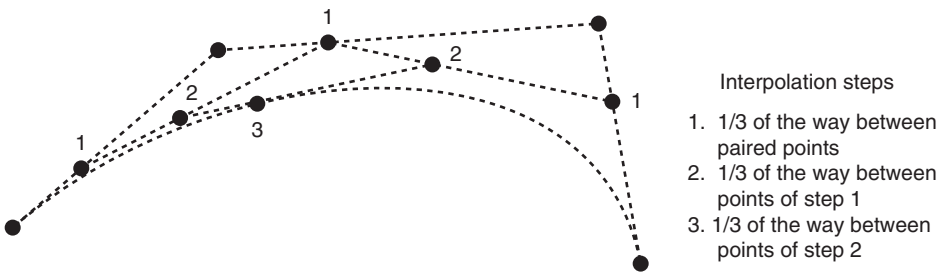


FIGURE B.46

De Casteljau construction of a point on a cubic Bezier curve.

Tension controls the sharpness of the bend of the curve at P_i. It does this by means of a scale factor that changes the length of both the incoming and outgoing tangents at the control point (Eq. B.85). In the default case, $t = 0$ and the tangent vector is the average of the two adjacent chords or, equivalently, half of the chord between the two adjacent points, as in the Catmull-Rom spline. As the tension parameter, t, goes to one, the tangents become shorter until they reach zero. Shorter tangents at the control point mean that the curve is pulled closer to a straight line in the neighborhood of the control point (see Figure B.47).

$$T_i^L = T_i^R = (1-t)\frac{1}{2}((P_{i+1} - P_i) + (P_i - P_{i-1})) \tag{B.85}$$

The continuity parameter, c, gives the user control over the continuity of the curve at the control point where the two curve segments join. The incoming (left) and outgoing (right) tangents at a control point are defined symmetrically with respect to the chords on either side of the control point. Assuming default tension, c blends the adjacent chords to form the two tangents, as shown in Equation B.86.

$$\begin{aligned} T_i^L &= \frac{1-c}{2}(P_i - P_{i-1}) + \frac{1+c}{2}(P_{i+1} - P_i) \\ T_i^R &= \frac{1+c}{2}(P_i - P_{i-1}) + \frac{1-c}{2}(P_{i+1} - P_i) \end{aligned} \tag{B.86}$$

The default value for continuity is $c = 0$, which produces equal left and right tangent vectors, resulting in continuity at the joint. As c approaches –1, the left tangent approaches equality with the chord to the left of the control point and the right tangent approaches equality with the chord to the right of the control point. As c approaches $+1$, the definitions of the tangents reverse themselves, and the left tangent approaches the right chord and the right tangent approaches the left chord (see Figure B.48).

Bias, b, defines a common tangent vector, which is a blend between the chord left of the control point and the chord right of the control point (Eq. B.87). At the default value ($b = 0$), the tangent is an even blend of these two, resulting in a Catmull-Rom type of internal tangent vector. Values of b approaching -1 bias the tangent toward the chord to the left of the control point, while values of b approaching $+1$ bias the tangent toward the chord to the right (see Figure B.49).

$$T_i^R = T_i^L = \frac{1+b}{2}(P_i - P_{i-1}) + \frac{1-b}{2}(P_{i+1} - P_i) \tag{B.87}$$

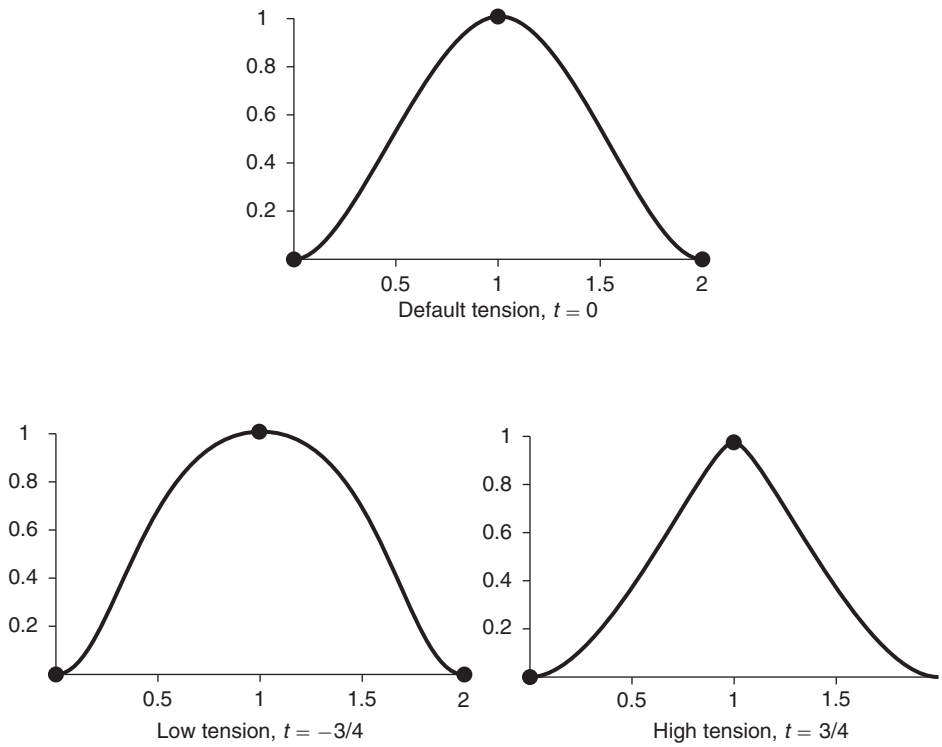


FIGURE B.47

The effect of varying the tension parameter.

The three parameters, tension, continuity, and bias, are combined in Equation B.88.

$$T_i^R = \frac{((1-t)(1+c)(1+b))}{2}(P_i - P_{i-1}) + \frac{((1-t)(1-c)(1-b))}{2}(P_{i+1} - P_i)$$
$$T_i^L = \frac{((1-t)(1-c)(1+b))}{2}(P_i - P_{i-1}) + \frac{((1-t)(1+c)(1-b))}{2}(P_{i+1} - P_i) \quad \text{(B.88)}$$

B.5.12 B-splines

B-splines are the most flexible and useful type of curve, but they are also more difficult to grasp intuitively. The formulation includes Bezier curves as a special case. The formulation for B-spline curves decouples the number of control points from the degree of the resulting polynomial. It accomplishes this with additional information contained in the *knot vector*. An example of a *uniform knot vector* is $[0, 1, 2, 3, 4, 5, 6, \ldots, n + k - 1]$, in which the knot values are uniformly spaced apart. In this knot vector, n is the number of control points and k is the degree of the B-spline curve. The parametric value

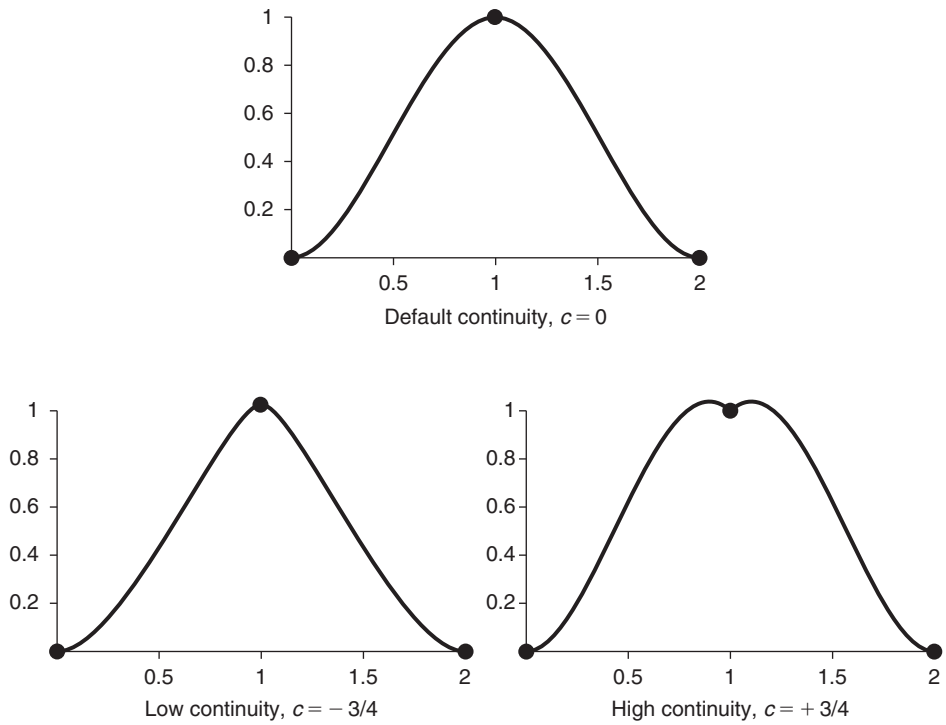


FIGURE B.48

The effect of varying the continuity parameter (with default tension).

varies between the first and last values of the knot vector. The knot vector establishes a relationship between the parametric value and the control points. With replication of values in the knot vector, the curve can be drawn closer to a particular control point up to the point where the curve actually passes through the control point.

A particularly simple, yet useful, type of B-spline curve is a uniform cubic B-spline curve. It is defined using four control points over the interval zero to one (Eq. B.89). A compound curve is generated from an arbitrary number of control points by constructing a curve segment from each four-tuple of adjacent control points: $(P_i, P_{i+1}, P_{i+2}, P_{i+3})$ for $i = 1, 2, \ldots, n-3$, where n is the total number of control points (Figure B.50). Each section of the curve is generated by multiplying the same 4×4 matrix by four adjacent control points with an interpolating parameter between zero and one. In this case, none of the control points is interpolated.

$$P(u) = \frac{1}{6}\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}\begin{bmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{bmatrix} \tag{B.89}$$

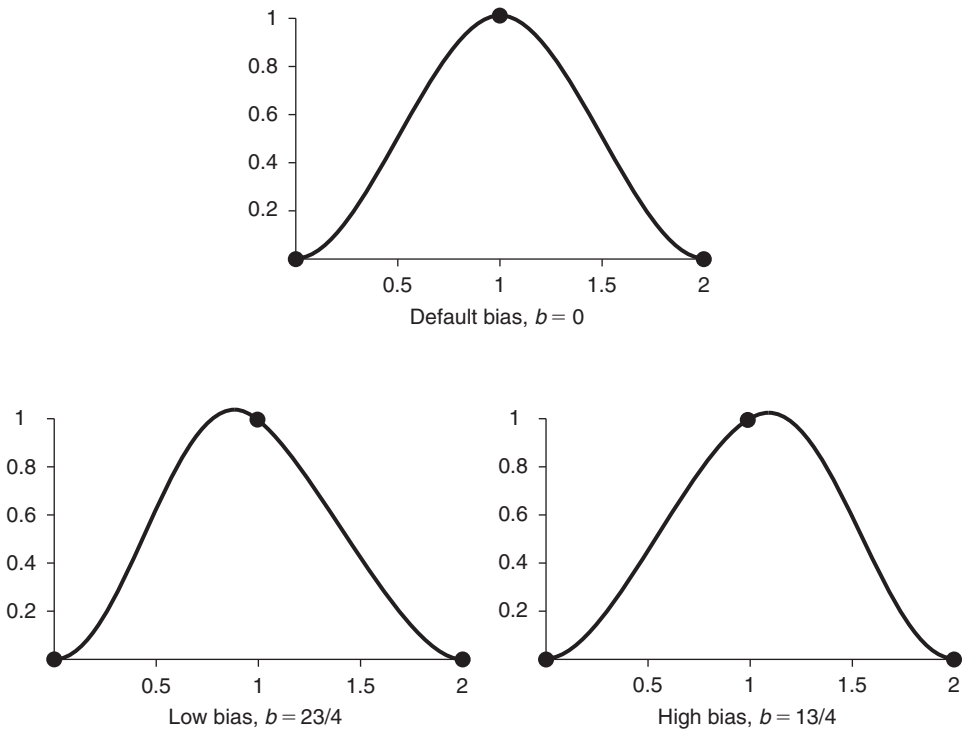


FIGURE B.49

The effect of varying the bias parameter (with default tension and continuity).

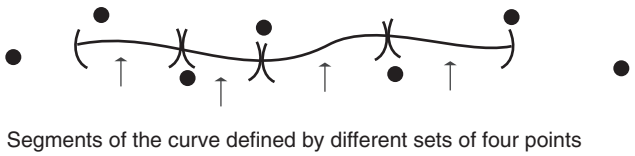


FIGURE B.50

Compound cubic B-spline curve.

Nonuniform rational B-splines (NURBS), are even more flexible than basic B-splines. NURBS allow for exact representation of circular arcs, whereas Bezier and nonrational B-splines do not. This is often important in modeling, but for purposes of animation, the basic uniform cubic B-spline is usually sufficient.

B.5.13 Fitting curves to a given set of points

Sometimes it is desirable to interpolate a set of points using a Bezier formulation. The points to be interpolated can be designated as the endpoints of the Bezier curve segments, and the interior control points can be constructed by forming tangent vectors at the vertices, as with the Catmull-Rom

formulation. The interior control points can be constructed by displacing the control points along the tangent lines just formed. For example, for the segment between given points b_i and b_{i11}, the first control point for the segment, c_i^1, can be positioned at $b_i + (1/3)\,(b_{i+1} - b_{i-1})$. The second control point for the segment, c_i^2, can be positioned at $b_{i+1} - (1/3) \cdot (b_{i+2} - b_i)$ (see Figure B.51).

Other methods exist. Farin [4] presents a more general method of constructing the Bezier curve and, from that, constructing the B-spline control points. Both Farin [4] and Rogers and Adams [18] present a method of constructing a composite Hermite curve through a set of points that automatically calculates internal tangent vectors by assuming second-order continuity at the segment joints.

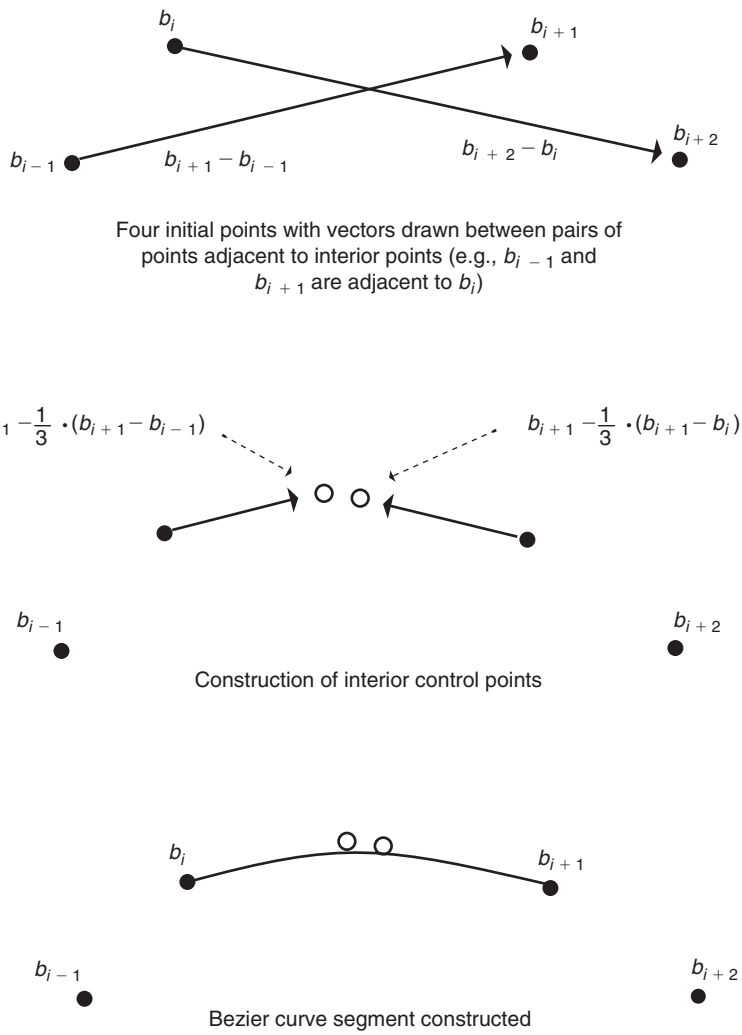


FIGURE B.51

Constructing a Bezier segment that interpolates points.

B.6 Randomness

Introducing controlled randomness in both modeling and animation can often produce more interesting, realistic, natural-looking imagery. Noise and turbulence functions are often used in textures but also can be used in modeling natural phenomena such as smoke and clouds. The code for noise and turbulence that follows is from Peachey's chapter in Ebert [3]. Random perturbations are also useful in human-figure animation to make the motion less "robotic" looking. There are various algorithms proposed in the literature for generating random numbers; Gasch's [6] is presented at the end of this section.

B.6.1 Noise

The *noise* function uses a table of pseudorandom numbers between -1 and $+1$ to represent the integer lattice values. The table is created by *valueTableInit* the first time that *noise* is called. Lattice coordinates are used to index into a table of pseudorandom numbers. A simple function of the coordinates, such as their sum, is used to compute the index. However, this can result in unwanted patterns. To help avoid these artifacts, a table of random permutation values is used to modify the index before it is used. A four-point spline is used to interpolate among the lattice pseudorandom numbers (*FPspline*).

```
#define TABSIZE                 256
#define TABMASK                 (TABSIZE-1)
#define PERM(x)                 perm[(x)&TABMASK]
#define INDEX(ix,iy,iz)         PERM((ix)+PERM((iy)+PERM(iz)))
#define FLOOR(x)                (int)(x)
/* PERMUTATION TABLE */
static unsigned char perm[TABSIZE] = {
225, 155, 210, 108, 175, 199, 221, 144, 203, 116, 70, 213, 69, 158, 33, 252, 5, 82, 173,
133, 222, 139, 174, 27, 9, 71, 90, 246, 75, 130, 91, 191, 169, 138, 2, 151, 194, 235, 81, 7,
25, 113, 228, 159, 205, 253, 134, 142, 248, 65, 224, 217, 22, 121, 229, 63, 89, 103, 96,
104, 156, 17, 201, 129, 36, 8, 165, 110, 237, 117, 231, 56, 132, 211, 152, 20, 181, 111,
239, 218, 170, 163, 51, 172, 157, 47, 80, 212, 176, 250, 87, 49, 99, 242, 136, 189, 162,
115, 44, 43, 124, 94, 150, 16, 141, 247, 32, 10, 198, 223, 255, 72, 53, 131, 84, 57, 220,
197, 58, 50, 208, 11, 241, 28, 3, 192, 62, 202, 18, 215, 153, 24, 76, 41, 15, 179, 39, 46,
55, 6, 128, 167, 23, 188, 106, 34, 187, 140, 164, 73, 112, 182, 244, 195, 227, 13, 35, 77,
196, 185, 26, 200, 226, 119, 31, 123, 168, 125, 249, 68, 183, 230, 177, 135, 160, 180, 12,
1, 243, 148, 102, 166, 38, 238, 251, 37, 240, 126, 64, 74, 161, 40, 184, 149, 171, 178, 101,
66, 29, 59, 146, 61, 254, 107, 42, 86, 154, 4, 236, 232, 120, 21, 233, 209, 45, 98, 193, 114,
78, 19, 206, 14, 118, 127, 48, 79, 147, 85, 30, 207, 219, 54, 88, 234, 190, 122, 95, 67, 143,
109, 137, 214, 145, 93, 92, 100, 245, 0, 216, 186, 60, 83, 105, 97, 204, 52
};
#define RANDNBR ww (((float)rand())/RAND_MASK)
float valueTab[TABSIZE];
/* ========================================================== */
/* VALUE TABLE INIT */
/* initialize the table of pseudorandom numbers */
void valueTableInit(int seed)
```

```
{
  float *table = valueTab;
  int i;
  srand(seed);
  for (i=0; i<TABSIZE; i++)
    *(table++) = 1.0 -2.0*RANDNBR;
}
/* ================================================================ */
/* LATTICE function */
/* returns a value corresponding to the lattice point */
float lattice(int ix, int iy, int iz)
{
  return valueTab[INDEX(ix,iy,iz)];
}
/* ================================================================ */
/* NOISE function */
float noise(float x, float y, float z)
{
  int ix,iy,iz;
  int i,j,k;
  float fx,fy,fz;
  float xknots[4],yknots[4],zknots[4];
  static int initialized = 0;
  if (!initialized) {
    valueTableInit(665);
    initialized = 1;
  }
  ix = FLOOR(x);
  fx = x - ix;
  iy = FLOOR(y);
  fy = y - iy;
  iz = FLOOR(z);
  fz = z-iz;
  for (k=-1; k<=2; k++) {
    for (j=-1; j<=2; j++) {
      for (i=-1; i<=2; i++) [i+1] = lattice(ix+i,iy+j,iz+k);
      yknots[j+1] = spline(fx,xknots);
    }
    zknots[k+1] = spline(fy,yknots);
  }
  return spline(fz,zknots);
}
#define FP00 -0.5
#define FP01 1.5
#define FP02 -1.5
#define FP03 0.5
#define FP10 1.0
```

```
#define FP11 -2.5
#define FP12 2.0
#define FP13 -0.5
#define FP20 -0.5
#define FP21 0.0
#define FP22 0.5
#define FP23 0.0
#define FP30 0.0
#define FP31 1.0
#define FP32 0.0
#define FP33 0.0
/* ========================================================= */
float spline(float u,float *knots)
{
  float c3,c2,c1,c0;
  c3 = FP00*knots[0] + FP01*knots[1] + FP02*knots[2] + FP03*knots[3];
  c2 = FP10*knots[0] + FP11*knots[1] + FP12*knots[2] + FP13*knots[3];
  c1 = FP20*knots[0] + FP21*knots[1] + FP22*knots[2] + FP23*knots[3];
  c0 = FP30*knots[0] + FP31*knots[1] + FP32*knots[2] + FP33*knots[3];
  return ((c3*u + c2)*u + c1)*u + c0;
}
```

B.6.2 Turbulence

Turbulence is a stochastic function with a "fractal" power spectrum [3]. The function is a sum of amplitude-varying frequencies. As frequency increases, the amplitude decreases.

```
/* TURBULENCE */
float turbulence (float x, float y, float z)
{
  float f;
  float value = 0;
  for (f = MINFREQ; f < MAXFREQ; f *= 2) value += fabs(noise(x*f, y*f, z*f))/f;
  return value;
}
```

B.6.3 Random number generator

This random number generator returns a random number in the range 0 to 999999999. An auxiliary routine would map this number into an arbitrary range of integers.

```
int r[100]; /* "global" pseudorandom table -- */
/* must be visible to rand and init_rand */
/* ========================================================= */
/* RAND */
/* return a random number in the range 0 to 999999999 */
int rand (void)
{
```

```
  int i = r[98];
  int j = r[99];
  int k;
  int t;
  if ((t = r[i] 2 r[j]) < 0) t += 1000000000 L;
  r[i] = t;
  r[98]--; r[99]--;
  if (r[98] == 0) r[98] = 55;
  if (r[99] == 0) r[99] = 55;
  k = r[100] % 42 + 56;
  r[100] = r[k];
  r[k] = t;
  return(r[100]);
}
/* ========================================================== */
/* INIT RAND */
/* seed the random number table */
void init_rand (char *seed)
{
  char buf[101];
  int i, j, k;
  if (strlen(seed) > 85) return(0);
  sprintf(buf, "aEbFcGdHeI%s", seed);
  while (strlen(buf) < 98) strcat(buf, "Q");
  for (i = 1; i < 98; i++) r[i] = buf[i] * 8171717 + i * 997;
  i = 97; j = 12;
  for (k = 1; k < 998; k++) {
    r[i] -= r[j];
    if (r[i] < 0) r[i] += 1000000000;
    i--; j--;
    if (i == 0) i=97;
    if (j == 0) j=97;
  }
  r[98] = 55;
  r[99] = 24;
  r[100] = 77;
}
/* ========================================================== */
/* RAND INT */
/* return a random int between a and b */
/* assumes init_rand already called. */
int rand_int(int a, int b)
{
  return (a + rand() % (b - a + 1));
}
```

B.7 Physics primer

Physically based motion is a limited simulation of physical reality. This can be as simple or as complex as the implementation requires. In the following sections are some of the equations of importance in simple physics simulation that can be found in any one of several standard texts. *The Mechanical Universe* [5] is used as the source for the brief discussion that follows.

B.7.1 Position, velocity, and acceleration

The fundamental equation relating position, distance, and speed is shown in Equation B.90. This can be used to control the positioning of an object for any particular frame of the animation because the frame number is tied directly to time (Eq. B.91). The average velocity of a body is the distance moved divided by the time it took to move, as stated by Equation B.92. Notice that the unit of velocity is distance per time, for example, feet/second.

$$\text{distance} = \text{speed} \times \text{time} \tag{B.90}$$

$$\text{time} = \text{frameNumber} \times \text{timePerFrame} \tag{B.91}$$

$$\text{average Velocity} = \text{distanceTraveled}/\text{time} \tag{B.92}$$

For this discussion, distance as a function of time is $s(t)$; the average velocity from time $t1$ to $t2$ is $(s(t2) - s(t1))/(t2 - t1)$. The instantaneous velocity is determined by moving $t2$ closer and closer to $t1$. In the limiting case this becomes the derivative of the distance function with respect to time. Similarly, the average acceleration of an object is the change in velocity divided by the time it took to effect the change. This is presented in Equation B.93, where $v(t)$ is a function that gives the velocity of the object at time t. Notice that the unit of acceleration is velocity per time or distance per time, for example, feet/second^2. In the same way, instantaneous acceleration is the derivative of $v(t)$ with respect to time (Eq. B.94). In the case of motion due to gravity, g is the acceleration due to gravity—a constant that has been measured to be 32 feet/second^2 or 9.8 meters/second^2 (Eq. B.95).

$$\mathrm{a}_{\mathrm{ave}} = (v(t_2) - v(t_1))/(t_2 - t_1) \tag{B.93}$$

$$a(t) = v'(t) = s''(t) \tag{B.94}$$

$$\begin{aligned} a(t) &= g \\ v(t) &= gt \\ s(t) &= (1/2)gt^2 \end{aligned} \tag{B.95}$$

B.7.2 Circular motion

Circular motion is important in physics and arises for a variety of phenomena, including the movement of planets and robotic armatures. Circular motion is easily specified by using polar coordinates. The position of a particle orbiting the origin at a distance r can be described using Equation B.96. Here, i and j are orthonormal unit vectors (at right angles to each other and unit length) and $p(t)$ is the positional vector of the particle. In a constant radius circular orbit, $\theta(t)$ varies as a function of time, and the distance r is constant. During uniform circular motion, $\theta(t)$ changes at a constant rate, and *angular velocity*

is said to be constant. Angular velocity is referred to here as $\omega(t)$ (Eq. B.97). As for constant-velocity linear motion, in which the distance equals speed multiplied by time, for constant angular velocity the angle equals angular velocity multiplied by time (Eq. B.98). If $\theta(t)$ is measured in radians and time in seconds, then $\omega(t)$ is measured in radians per second. To simplify the following equations, the functional dependence on time will often be omitted when the dependence is obvious from the context.

$$p(t) = (r\cos(\theta(t)))i + (r\sin(\theta(t)))j \tag{B.96}$$

$$\frac{d}{dt}\theta(t) = \omega(t) \tag{B.97}$$

$$\theta(t) = \omega t \tag{B.98}$$

Taking the derivative of Equation B.96 with respect to time gives the instantaneous velocity (Eq. B.99). Notice that the velocity vector, $v(t)$, is perpendicular to the position vector, $p(t)$. This can be demonstrated by taking the dot product of the two vectors $v(t)$ and $p(t)$ and showing that it is identically zero.

$$v(t) = \frac{dp}{dt} = ((-r)\omega\sin(\omega t))i + (-r\omega\cos(\omega t))j \tag{B.99}$$

Computing the length of $v(t)$ shows that $|v| = r\,\omega$ and, therefore, that the velocity is independent of t (i.e., constant). Notice, however, that a constant circular motion still gives rise to an acceleration. Taking the derivative of Equation B.99 produces Equation B.100, which is called the centripetal acceleration. The centripetal acceleration resulting from uniform circular motion is directed radially inward and has constant magnitude. With the equation for the length of $v(t)$ from earlier, the magnitude of the acceleration can be written using Equation B.101.

$$\begin{aligned} a(t) = \frac{dv}{dt} &= (-\omega^2)((r\cos(\omega t))i + (r\sin(\omega t))j) \\ &= (-\omega^2)p(t) \end{aligned} \tag{B.100}$$

$$a = v^2/r \tag{B.101}$$

For any particle in a rigid mass undergoing a rotation, that particle is undergoing the same rotation about its own center. In addition, if the particle is displaced from the center of rotation, then it is also undergoing an instantaneous positional translation as a result of its circular motion (Figure B.52).

B.7.3 Newton's laws of motion

It is useful to review Newton's laws of motion. The first law is the principle of inertia. The second law relates force to the acceleration of a mass (Eq. B.102). In another form, this law relates force to change in momentum (Eq. B.103), where momentum is mass times velocity ($m \bullet v$). The third law states that when an object pushes with a force on another object, the second object pushes back with an equal but opposite force. It is important to note that the force F used here is considered to be the sum of all external forces acting on an object. Force is a vector quantity, and these equations really represent three sets of equations, one for each coordinate (Eq. B.104). Newton's laws of motion are stated as follows:

First Law: If no force is acting on an object, then it will not accelerate. It will maintain a constant velocity.

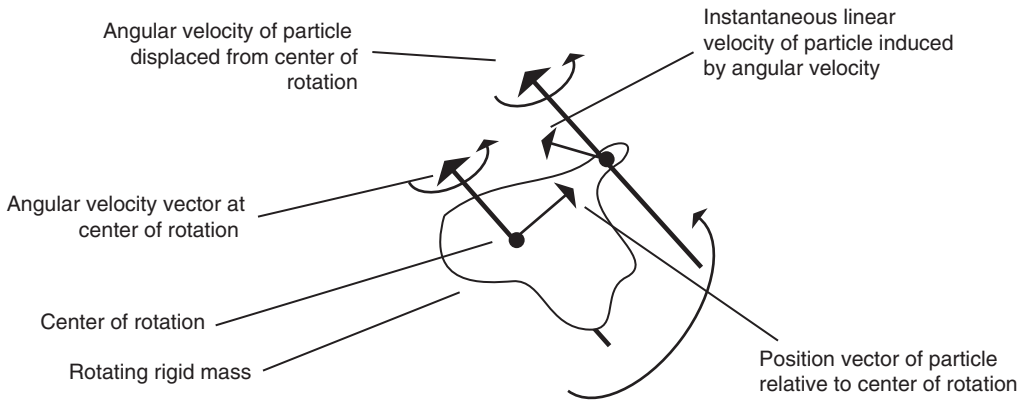


FIGURE B.52

Motion of particle in a rotating rigid mass.

Second Law: The change of motion of an object is proportional to the forces applied to it.
Third Law: To every action there is always opposed an equal and opposite reaction.

$$F = ma \tag{B.102}$$

$$F = \frac{d}{dt}(mv) \tag{B.103}$$

$$F_x = ma_x$$
$$F_x = ma_y$$

$$F_x = ma_z \tag{B.104}$$

B.7.4 Inertia and inertial reference frames

An *inertial frame* is a frame of reference (e.g., a local coordinate system of an object) in which the principle of inertia holds. Any frame that is not accelerating is an inertial frame. In an inertial frame one observes the laws of motion and has no way of determining whether one is at rest or moving in an "absolute" sense. (But then, what is "absolute"?)

B.7.5 Center of mass

The center of mass of an object is that point at which the object is balanced in all directions. If an external force is applied in line with the center of mass of an object, then the object moves as if all the mass were concentrated at the center ("c" in Figure B.53).

B.7.6 Torque

The tendency of a force to produce circular motion is called *torque*. Torque is produced by a force applied off-center from the center of mass of an object (Figure B.54) and is computed by Equation B.105.

$$\tau = r \times F \tag{B.105}$$

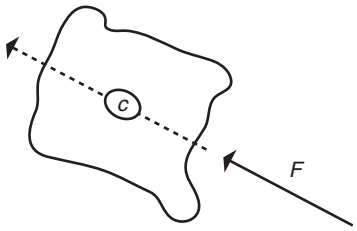


FIGURE B.53

Center of mass.

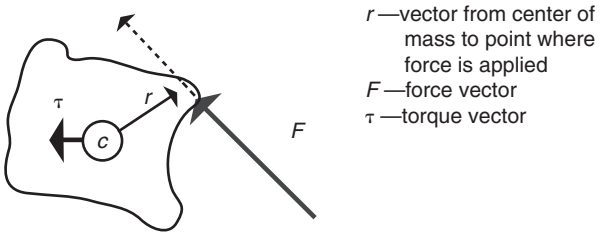


FIGURE B.54

Applying a force off-center induces a torque.

It is important to note that *torque* refers to a specific axis of rotation and is perpendicular to both r and F in Figure B.54. The same force can exert different torques about different axes of rotation if it is applied at different locations on an object. A given torque vector for a rigid body is position independent. That is, for any particle in a rigid mass, the particle is undergoing that same torque.

B.7.7 Equilibrium: balancing forces

In the absence of acceleration, there is no change in the motion of an object (Newton's first law). In such cases, the net force and the torque acting on a body vanish (Eqs. B.106 and B.107). There may be forces present, but the vector sum is equal to zero.

$$\Sigma F = 0 \tag{B.106}$$

$$\Sigma \tau = 0 \tag{B.107}$$

B.7.8 Gravity

Equation B.108 is Newton's law of universal gravitation. It calculates the force of gravity between two masses (m_1 and m_2) whose centers of mass are a distance r apart. G is the universal gravitational constant equal to 6.67×10^{-11} Newton meter^2/kilogram^2. When two objects are not touching, each acts on the other as if all its mass were concentrated at its center of mass. When an object is on (or near) the earth's surface, the distance of the object to the center of mass of the earth, the mass of the earth, and the

gravitational constant of Equation B.108 can all be combined to produce the object's acceleration (Eq. B.109). This acceleration is usually denoted as g.

$$F = G\frac{m_1 m_2}{r^2} \tag{B.108}$$

$$a = \frac{F}{m_{\text{object}}} = G\frac{m_{\text{earth}}}{(\text{radius}_{\text{earth}})^2} = 9.8\text{m/s}^2 = 32\text{f/s}^2 = g \tag{B.109}$$

B.7.9 Centripetal force

Centripetal force is any force that is directed inward, toward the center of an object's motion (centripetal means *center seeking*). In the case of a body in orbit, gravity is the centripetal force that holds the body in the orbit. Consider a body, such as the moon, with mass M_m and a distance r away from the earth. The unit vector from the earth to the moon is R. The earth has a mass M_e. Equation B.110 calculates the force vector that results from gravity. The acceleration induced by a circular orbit always points toward the center (centripetal), as in Equation B.111. As previously shown, the acceleration due to circular motion has magnitude v^2/r (Eq. B.100). This can be used to solve for velocity (Eq. B.112).

$$F = -G\frac{M_m M_e}{r^2}R \tag{B.110}$$

$$\begin{aligned} a &= F/M_m \\ a &= ((-GM_e)/r^2)R \\ a &= (-v^2/r)R \\ a &= (-GM_e/r^2)R \end{aligned} \tag{B.111}$$

$$v = \sqrt{(GM_e)/r} \tag{B.112}$$

B.7.10 Contact forces

Gravity is one of the four fundamental forces (gravity, electromagnetic, strong, and weak) that act at a distance. Another important category is *contact forces*. The tension in a wire or rope is an example of a contact force, as is the compression in a rigid rod. These forces arise from the complex interactions of electric forces that tend to keep atoms a certain distance apart. The empirical law that governs such forces, and that is familiar when dealing with springs, is *Hooke's law* (Eq. B.113). Variable x is the change from the equilibrium length of the spring, k is the spring constant, and F is the restoring force of the spring to return to rest length. The constant k is a measure of the stiffness of the spring; the larger k is, the more sensitive the spring is to motion away from the rest position.

$$F = -kx \tag{B.113}$$

Another example of contact force, known as the *normal force*, is the result of repulsion between any two objects pressed against each other. It arises from the repulsion of the atoms of the two objects. It is always perpendicular to the surfaces, and its magnitude is proportional to how hard the two objects are pressed against each other. When objects in motion come in contact, an *impulse force* due to collision is

produced. The impulse force is a short-duration force that is applied normal to the surface of contact on each of the two objects. Calculation of the impulse force due to collision is discussed in Chapter 7, Section 4.2. Other important examples of contact forces are friction and viscosity.

Friction

Friction arises from the interaction of surfaces in contact. It can be thought of as resulting from the multitude of collisions of molecules caused by the unevenness of the surfaces at the microscopic level.

The frictional force works against the relative motion of the two objects. Frictional forces from a surface do not exceed an amount proportional to the normal force exerted by the surface on the object. This is stated in Equation B.114, where s is the coefficient of static friction and f_N is the normal force (component of the force that is perpendicular to the surface). Variable s varies according to the two materials that are in contact.

$$f_s = sf_N \tag{B.114}$$

The frictional forces acting between surfaces at rest with respect to each other are called *forces of static friction*. For a force applied to an object sitting on another object that is parallel to the surfaces in contact, there will be a specific force at which the block starts to slip. As the force increases from zero up to that threshold force, the lateral force is counteracted by an equal force of friction in the opposite direction. Once the object begins to move, *kinetic friction* acts on the object and approximately obeys the empirical law of Equation B.115, where k is the coefficient of kinetic friction and f_N is the force normal to the surface. Kinetic friction is typically less than static friction. The force of kinetic friction is always opposite to the velocity of the object.

$$f_k = kf_N \tag{B.115}$$

Viscosity

The resistive force of an object moving in a medium is *viscosity*. It is another contact force and is extremely difficult to model accurately. When an object moves at low velocity through a fluid, the viscosity is approximately proportional to the velocity (Eq. B.116); K is the constant of proportionality, which depends on the size and the shape of the object, and n is the coefficient of viscosity, which depends on the properties of the fluid. The coefficient of viscosity, n, decreases with increasing temperature for liquids and increases with temperature for gases. Stokes's law for a sphere of radius R is given in Equation B.117.

$$F_{\text{vis}} = -Knv \tag{B.116}$$

$$K = 6\pi R \tag{B.117}$$

An object dropping through a liquid attains a constant speed, called the limiting or terminal velocity, at which gravity, acting downward, and the viscous force, acting upward, balance each other and there is no acceleration (e.g., Eq. B.118 for a sphere). Terminal velocity is given by Equation B.119. In a viscous medium, heavier bodies fall faster than lighter bodies. For spherical objects falling at a low velocity in a viscous medium, not necessarily at terminal velocity, change in momentum is given by Equation B.120.

$$mg = 6\pi Rnv \tag{B.118}$$

$$v = (mg)/(6\pi Rn) \tag{B.119}$$

$$m\frac{dv}{dt} = mg - 6\pi Rnv \tag{B.120}$$

B.7.11 Centrifugal force

Consider an object (a frame of reference) that rotates in uniform circular motion with respect to a post (an inertial frame) because it is held at a constant distance by a rope. Relative to the inertial frame, each point in the uniformly rotating frame has centripetal acceleration expressed by Equation B.121; r is the distance of the point from the axis of rotation, R is the unit vector from the inertial frame to the rotating frame, and v is the speed of the point. The tension in the rope supplies the force necessary to produce the centripetal acceleration. Relative to the rotating frame (not an inertial frame), the frame itself does not move and therefore the *centrifugal* force necessary to counteract the force supplied by the rope is calculated by using Equation B.122.

$$a = -\frac{v^2}{r}R \tag{B.121}$$

$$F_c = -(ma) = -\frac{mv^2R}{r} \tag{B.122}$$

B.7.12 Work and potential energy

For a constant force of magnitude F moving an object a distance h parallel to the force, the work W performed by the force is shown in Equation B.123. If a mass m is lifted up so that it does not accelerate, then the lifting force is equal to the weight (mass $\times$ gravitational acceleration) of the object. Since the weight is constant, the work done to raise the object up to a height h is presented in Equation B.124. Energy that a body has by virtue of its location is called *potential energy*. The work in this case is converted into potential energy.

$$W = Fh \tag{B.123}$$

$$W = mgh \tag{B.124}$$

B.7.13 Kinetic energy

Energy of motion is called *kinetic energy* and is shown in Equation B.125. The velocity of a falling body that started at a height h is calculated by Equation B.126. Its kinetic energy is therefore calculated by Equation B.127.

$$K = \frac{1}{2}mv^2 \tag{B.125}$$

$$v^2 = 2gh \tag{B.126}$$

$$K = \frac{1}{2}mv^2 = mgh \tag{B.127}$$

B.7.14 Conservation of energy

Potential plus kinetic energy of a closed system is conserved (Eq. B.128). This is useful, for example, when solving for an object's current height (current potential energy) when its initial height (initial potential energy), initial velocity (initial kinetic energy), and current velocity (current kinetic energy) are known.

$$U_a + K_a = U_b + K_b \tag{B.128}$$

B.7.15 Conservation of momentum

In a closed system, total momentum (mass times velocity) is conserved. This means that it does not change (Eq. B.129) and that the momenta of all objects in a closed system always sum to the same amount (Eq. B.130). This is useful, for example, when solving for velocities after a collision when the velocities before the collision are known.

$$\frac{d}{dt}((m_1v_1 + m_2v_2 + \ldots + m_nv_n)) = 0 \tag{B.129}$$

$$m_1v_1 + m_2v_2 + \ldots + m_nv_n = \text{constant} \tag{B.130}$$

B.7.16 Oscillatory motion

In some systems, the stability of an object is subject to a linear restoring force, F. The force is linearly proportional (using k as the constant of proportionality) to the distance x the object has been displaced from its equilibrium position (Eq. B.131). Associated with this force is the potential energy (Eq. B.132) that results from the object's displacement. These systems have the property that, if they are disturbed from equilibrium, the restoring force that acts on them tends to move them back into equilibrium. When disturbed from equilibrium, they tend to overshoot that point when they return, due to inertia. Then the restoring force acts in the opposite direction, trying to return the system to equilibrium. The result is that the system oscillates back and forth like a mass on the end of a spring or the weight at the end of a pendulum.

$$F = -kx \tag{B.131}$$

$$U = \frac{1}{2}kx^2 \tag{B.132}$$

Combining the basic equations of motion (force equals mass times acceleration) with Equation B.131, one can derive the differential equation for oscillatory motion. This equation is satisfied by the displacement function $x(t)$ (Eq. B.133).

$$\begin{aligned} F &= ma \\ F &= -kx \\ a &= \frac{d^2x}{dt} \\ m\frac{d^2x}{dt} &= -kx \\ \frac{d^2x}{dt} &= -kx/m \end{aligned} \tag{B.133}$$

B.7.17 Damping

The damping force can be modeled after Stokes's law, in which resistance is assumed to be linearly proportional to velocity (Eq. B.134). This is usually valid for oscillations of sufficiently small amplitude. The damping force opposes the motion. The constant, k_d, is called the *damping coefficient*. Adding the damping force to the spring force produces Equation B.135. Dividing through by m and collecting terms results in Equation B.136, where $b = k_d/m$ and $a^2 = k/m$.

$$F_d = -k_d \frac{dx}{dt} \tag{B.134}$$

$$m\frac{d^2x}{dt} = -kx - k_d \frac{dx}{dt} \tag{B.135}$$

$$\frac{d^2x}{dt} + b\frac{dx}{dt} + a^2x = 0 \tag{B.136}$$

If there is a spring force but no damping, the general solution can be written as $x = C\cos(a\,t + \theta_0)$. If there is damping but no spring force, the general solution turns out to be $x = C\,e^{-bt} + D$, with C and D constant. If both the spring force and the damping force are present, the solution takes the form shown in Equation B.137.

$$\begin{aligned} &x = Ce^{-bt}\cos(dt + \theta_0) \\ \text{where } &d = \sqrt{a^2 - \frac{b^2}{4}} \\ \text{for } &b < 2a \end{aligned} \tag{B.137}$$

B.7.18 Angular momentum

Angular momentum is the rotational equivalent of linear momentum and can be computed by Equation B.138, where r is the vector from the center of rotation and p is momentum (mass × velocity). The temporal rate of change of angular momentum is equal to torque (Eq. B.139). Angular momentum, like linear momentum, is conserved in a closed system (Eq. B.140).

$$L = r \times p \tag{B.138}$$

$$\tau = \frac{dL}{dt} \tag{B.139}$$

$$\Sigma L_i = \text{constant} \tag{B.140}$$

B.7.19 Inertia tensors

An *inertia tensor*, or *angular mass*, describes the resistance of an object to a change in its angular momentum [5] [13]. It is represented as a matrix when the angular mass is related to the principal axes of the object (Eq. B.141). The terms of the matrix describe the distribution of the mass of the object relative to a local coordinate system (Eq. B.142). For objects that are symmetrical with respect to the local axes, the off-diagonal elements are zero (Eq. B.143). For a rectangular solid with mass M and dimensions a, b, and c along its local axes, the inertia tensor at the center of mass is given by Equation B.144. For a sphere with

radius R and mass M, the inertia tensor is given by Equation B.145. If the inertia tensor is known at one position by I, then the inertia tensor I' for parallel axes at a new position (X, Y, Z) relative to the original position is calculated by Equation B.146. If the inertia tensor for one set of axes is known by I, then the inertia tensor for a rotated frame is calculated by Equation B.147, where R is the rotation matrix describing the rotated frame relative to the original frame.

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix} \tag{B.141}$$

$$\begin{aligned} I_{xx} &= \int (y^2 + z^2) dm \\ I_{yy} &= \int (x^2 + z^2) dm \\ I_{zz} &= \int (x^2 + y^2) dm \\ I_{xy} &= \int xy dm \\ I_{xz} &= \int xz dm \\ I_{yz} &= \int yz dm \end{aligned} \tag{B.142}$$

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{B.143}$$

$$I = \frac{1}{12} \begin{bmatrix} M(b^2 + c^2) & 0 & 0 \\ 0 & M(a^2 + c^2) & 0 \\ 0 & 0 & M(a^2 + b^2) \end{bmatrix} \tag{B.144}$$

$$I = \frac{2MR^2}{5} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{B.145}$$

$$I_{\text{translated}} = \begin{bmatrix} I_{xx} + M(Y^2 + Z^2) & -I_{xy} - MXY & -I_{xz} - MXZ \\ -I_{xy} - MXY & I_{yy} + M(X^2 + Z^2) & -I_{yz} - MXYZ \\ -I_{xz} - MXZ & -I_{yz} - MZYX & I_{zz} + MX^2 + Y^2 \end{bmatrix} \tag{B.146}$$

$$I_{\text{rotated}} = R I_{\text{object}} R^{-1} \tag{B.147}$$

B.8 Numerical integration techniques

Numerical integration is useful for finding the arc length of the curve, updating arbitrary function values using derivative information, and specifically updating the position of an object over time. A useful technique for arc length computation is Gaussian quadrature. With regard to general function value updating, Runge-Kutta, explicit Euler integration, and implicit Euler integration are discussed. Huen, Verlet, and Leapfrog integration are covered specifically in the context of position update using a known acceleration. As with many numerical techniques, Press et al. [16] is an extremely valuable reference.

B.8.1 Function integration for arc length computation

Given a function $f(x)$, Gaussian quadrature can be used to produce an arbitrarily close approximation to the integral of the function between two values, $\int_a^b f(x)$, if the function is sufficiently well behaved [16]. Gaussian quadrature approximates the integral as a sum of weighted evaluations of the function at specific values (abscissas). The number of evaluations controls the error in the approximation. In its general form, Gaussian quadrature incorporates a multiplicative function, $W(x)$, which can condition some functions for the approximation (Eq. B.148). Gauss-Legendre integration is a special case in which $W(x) = 1.0$, and it results in Equation B.149. The code in Figure B.55 for $n = 10$ duplicates that used in Chapter 3, Section 2.1 for computing arc length. Figure B.56 gives the code for computing Gauss-Legendre weights and abscissas for arbitrary n.

$$\int_a^b (W(x)f(x))dx \cong \sum_{i=1}^{n} w_i f(x_i) \tag{B.148}$$

$$\int_a^b f(x)dx \cong \sum_{i=1}^{n} w f(x_i) \tag{B.149}$$

```
/* ---------------------------------------------------------------------
INTEGRATE FUNCTION
use gaussian quadrature to integrate square root of given function in
the given interval */
double integrate_func(polynomial_td *func,interval_td *interval)
{
  double x[5]={.1488743389,.4333953941,.6794095682,.8650633666,.9739065285};
  double w[5]={.2966242247,.2692667193,.2190863625,.1494513491,.0666713443};
  double length, midu, dx, diff;
  int i;
  double evaluate_polynomial();
  double u1,u2;

  u1 = interval->u1;
  u2 = interval->u2;

  midu = (u1+u2)/2.0;
  diff = (u2-u1)/2.0;
  length = 0.0;
  for (i=0; i<5; i++) {
    dx = diff*x[i];
    length += w[i]*(sqrt(evaluate_polynomial(func,midu+dx)) +
    sqrt(evaluate_polynomial(func,midu-dx)));
  }
  length *= diff;
  return (length);
}
```

FIGURE B.55

Gauss-Legendre integration for $n = 10$.

```
/* GAUSS-LEGENDRE */

#define EPSILON 0.00000000001
/* calculate the weights and abscissas of the Gauss-Legendre n-point form */
void gaussWeights(float a, float b, float *x, float *w, int n)
{
  int i,j,m;
  float p1,p2,p3,p;
  float z,z1;
  float xave,xdiff;

  m = (n+1)/2;
  xave = (b+a)/2;
  xdiff = (b-a)/2;
  for (i = 0; i<m; i++) {
    z = cos(PI*((i+1)-0.25)/(n+0.5));
    do {
      p1 = 1.0;
      p2 = 0.0;
      for (j=0; j<n; j++) {
        p3 = p2;
        p2 = p1;
        p1 = ((2*(j+1) - 1.0)*z*p2-j*p3)/(j+1);
      }
      pp = n*(z*p1-p2)/(z*z-1);
      z1 = z;
      z = z1-p1/pp;
    } while (fabs(z-z1) > EPSILON);
    x[i] = xave - xdiff*z;
    x[n-1-i] = xave + xdiff*z;
    w[i] = 2.0*xdiff/((1.0-z*z)*pp*pp);
    w[n-1-i] = w[i];
  }
}
```

FIGURE B.56

Computing Gauss-Legendre weights and abscissas.

B.8.2 Updating function values

Integrating ordinary differential equations (ODEs) in computer animation typically means that the derivative function f' is available and that a numerical approximation to the function f is desired. For example, in a physically based simulation, the time-varying acceleration of an object is computed from the object's mass and the forces acting on the object in the environment. From the acceleration (the derivative function), the velocity (the function) is numerically calculated over time. Similarly,

once the time-based velocity function is known (the derivative function), the time-varying position of the object (the function) can be calculated numerically.

The simple form of an ordinary differential equation involves a first-order derivative of a function of a single variable. In addition, it is usually the case that conditions at an initial point in time are known and that the numerical integration is used in a simulation of a system as time moves forward. Such problems are referred to as *initial value problems*.

The (explicit) Euler method

The *Euler method* is the most basic technique used for solving such simple ODE initial value problems. It is shown in Equation B.150, where h is the time step such that $x_{n+1} = h + x_n$. This method is not symmetrical in that it uses information at the beginning of the time step to advance to the end of the time step. The derivative at the beginning of the time step produces the vector, which is tangent to the curve representing the function at that point in time. The tangent at the beginning of the interval is used as a linear approximation to the behavior of the function over the entire interval (Figure B.57). The Euler method is neither stable nor very accurate. Other methods, such as Runge-Kutta, are more accurate for equivalent computational cost.

$$y_{n+1} = y_n + hf'(x_n, y_n) \tag{B.150}$$

Runge-Kutta

Runge-Kutta is a family of methods that is symmetrical with respect to the interval. The *second-order Runge-Kutta*, or *midpoint method*, is mentioned in Chapter 7, Section 4.1 in the discussion of physically based simulations. A half-step, using the explicit Euler method, is taken and the derivative is evaluated. This derivative is then used to update the original value (see Eq. B.151).

$$y_{n+1} = y_n + f'\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}f'(x_n, y_n)\right) \tag{B.151}$$

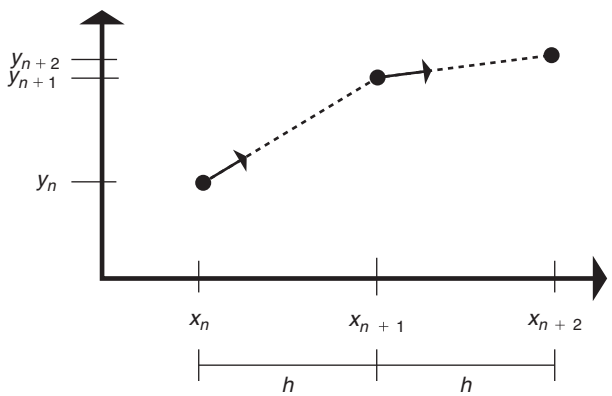


FIGURE B.57

The Euler method.

One of the most useful methods is *fourth-order Runge-Kutta* [16], shown in Equation B.152 and Figure B.58. It is referred to as a fourth-order method because its error term is on the order of the interval h to the fifth power. The advantage of using the method is that although each step requires more computation, larger step sizes can be used, resulting in an overall computational savings.

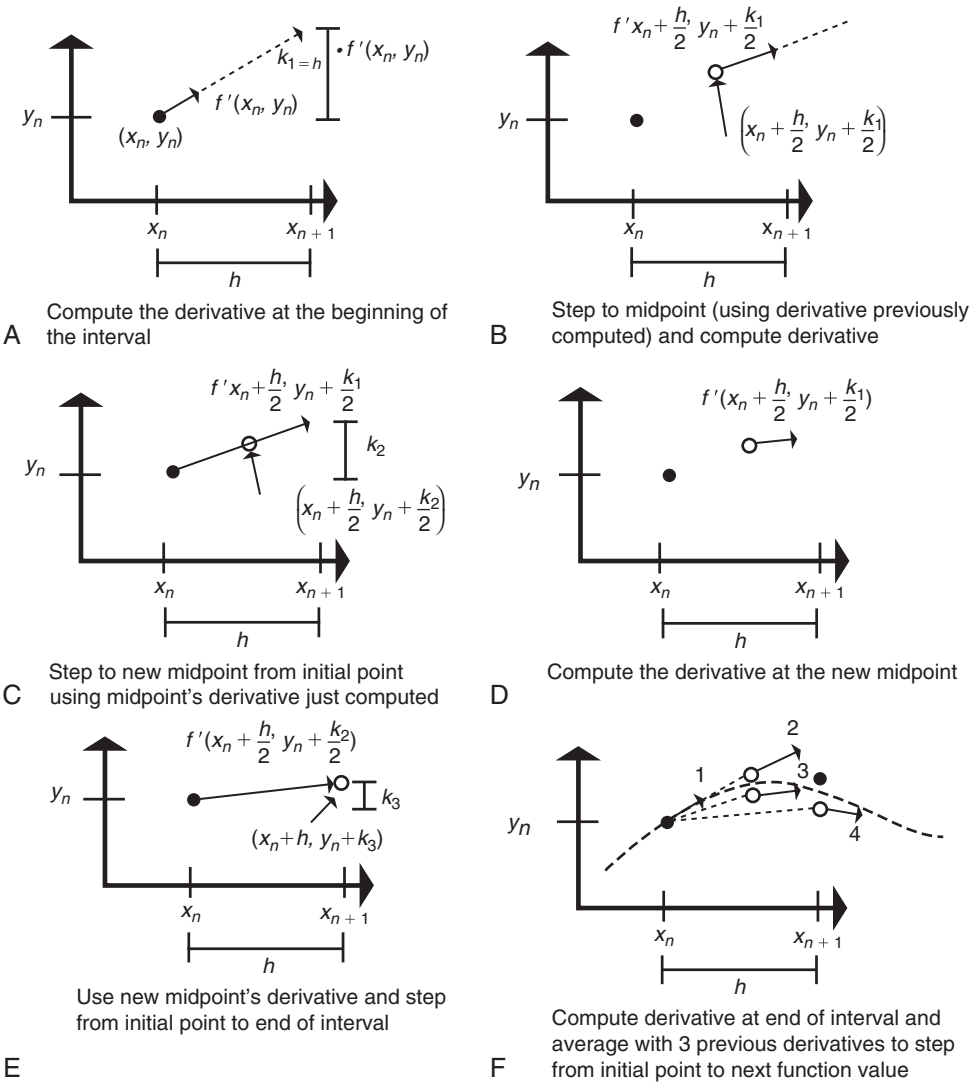


FIGURE B.58

The steps in computing fourth-order Runge-Kutta.

$$
\begin{aligned}
k_1 &= hf'(x_n, y_n) \\
k_2 &= hf'\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\
k_3 &= hf'\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \\
k_4 &= hf'(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{aligned}
\tag{B.152}
$$

The implicit Euler method

The explicit Euler method updates the function value using the derivative at the current value. The danger with the implicit method is that if the time step is too large, the updated function values can be a poor approximation to the actual function. The implicit method, for comparison, finds a new position whose derivative can update the current value to the new value (Eq. B.153).

$$y_{n+1} = y_n + hf'(x_{n+1}, y_{n+1}) \tag{B.153}$$

This requires solving an equation to find such a y_{n+1} using, for example, a Newton-Raphson method. It takes more time to take a step using the implicit Euler method but because this technique is more stable than explicit Euler, large time steps can be taken and, thus, an overall computational savings can often be achieved.

The semi-implicit Euler method

The *semi-implicit Euler method* is derived from the implicit Euler method by approximating the y_{n+1} term used in evaluating the derivative by an explicit Euler step, thus resulting in Equation B.154.

$$y_{n+1} = y_n + hf'(x_{n+1}, y_n + hf'(x_n, y_n)) \tag{B.154}$$

The semi-implicit Euler method offers more stability than explicit Euler and is computationally much less expensive than implicit Euler.

B.8.3 Updating position

The next three methods are discussed specifically in the context of updating position from velocity and/or acceleration over a time periodΔt.

The Heun method

The *Heun method* of integration, also called the *improved Euler method*, augments the basic Euler method. The Euler method takes the derivative at the current position to update to the next position. The Heun method takes the derivative at the position computed by the basic Euler method and averages it with the first-computed derivative and uses the average to update the position (Eq. B.155). This is a common integration method used in physics texts.

$$\begin{aligned} v(t_{i+1}) &= v(t_i) + a(t_i)\Delta t \\ x(t_{i+1}) &= x(t_i) + \frac{1}{2}(v(t_i) + v(t_{i+1}))\Delta t \\ x(t_{i+1}) &= x(t_i) + v(t_i)\Delta t + \frac{1}{2}a(t_i)\Delta t^2 \end{aligned} \tag{B.155}$$

The Verlet method

Verlet integration is used to update the position directly from acceleration without explicitly calculating velocity. Essentially, the difference between the current position and the previous position is used as velocity. This vector is updated by acceleration and added to the current position to generate the next position (see Eq. B.156).

$$x(t_{i+1}) = 2x(t_i) - x(t_{i-1}) + a(t_i)\Delta t^2 \tag{B.156}$$

The Leapfrog method

The *Leapfrog method* is unique in its temporal symmetry. It can be run forward in time or backward in time and it will trace out the same path. Position and acceleration are evaluated on the interval while velocity is evaluated on the half interval relative to the position (see Eq. B.157).

$$\begin{aligned} x(t_{i+1}) &= x(t_i) + v(t_{i+\frac{1}{2}})\Delta t \\ v(t_{i+\frac{3}{2}}) &= v(t_{i+\frac{1}{2}}) + a(t_{i+1})\Delta t \end{aligned} \tag{B.157}$$

B.9 Optimization

As the field of computer graphics and computer animation matures, more complex and mathematically sophisticated techniques are being used. An increasingly important mathematical tool is *optimization*, also know as *mathematical programming*.[2] This discussion is intended for the reader who is well versed in computer graphics and familiar with the basics of calculus, but who has not had much, if any, exposure to mathematical techniques involving optimization and who might be a bit intimidated whenever discussion turns that direction. Optimization is a complex and expansive topic, and finds application in control theory, chemistry, meteorology, VLSI CAD, geophysics, power grid design, and many others. A complete treatment is beyond the scope of the current discussion and the interested reader should follow up with material from the graphics literature (e.g., [27] [28]) or the mass of material on optimization in general (e.g., [29]) or in the literature concerning a specific application of interest.

In computer graphics, optimization finds application in areas such as surface fitting, decimation, inverse kinematics, motion capture data editing, physically based motion, and path construction. These tasks have certain features in common. For example, problems in these areas usually have a large

[2]The term "programming" refers to the U.S. military's use of the term program as applied to logistics, the problem domain for the first early optimization techniques, referring to a well-defined procedure for doing something (i.e., linear programming), and has nothing to do with computer programming.

parameter space. Geometric problems that define vertex placement or selection can have tens of thousands of vertices and, of course, triple the number in coordinate values. Animation problems might have a relatively small set of parameters, such as velocity or torque, but require the values for these parameters over thousands of frames of animation. The resulting high-dimensional parameter space can be too complex for an animator to easily set the parameters and produce the desired motion. The other common feature is that the function that evaluates the parameters to find the "best" values is not an analytic function; rather, it is a rather computationally expensive procedure. Thus, techniques that require fewer function evaluations are preferred.

Optimization is finding the optimal, or best, value among a set of alternatives. For the current discussion, the minimum function value is sought over the space spanned by the function's parameters. For example, in Equation B.158, the minimum value is 1.

$$\min_{x \in \Re^n} \quad f(x) \tag{B.158}$$

Alternatively, optimization can also be expressed in terms of finding the values for a set of input parameters that minimize (or maximize) a function's single output value. For example, in Equation B.159, the input value that corresponds to the function's minimum is 0.

$$\arg\min_{x \in \Re^n} \quad f(x) \tag{B.159}$$

The number of parameters and their possible values define the search space. The input parameters may be *unconstrained*, as in "x in reals," or the optimization problem may be *constrainted*, as in "x greater than 0." The constraints can include equality as well as inequality constraints (Eq. B.160).

$$\begin{aligned} &\text{minimize } f(\mathrm{X}) \\ &\text{Subject to } c_i(\mathrm{X}) = 0 \quad i = 1, 2, \ldots, k_{eq} \\ &\text{Subject to } \hat{c}_j(\mathrm{X}) = 0 \quad j = 1, 2, \ldots, k_{le} \end{aligned} \tag{B.160}$$

The function, f, can be anything that is able to be evaluated given values for its input parameters. As mentioned, in computer graphics applications, the function is often not analytically defined because the function is actually an arbitrarily complex procedure that produces a value such as the time it takes to get somewhere, or a motion that minimizes the maximum torque during some task.

This discussion of optimization will often consider minimizing a function, although it obviously does not matter whether we are trying to minimize or maximize something—a function to be maximized uses the same principles as a function to be minimized, or, if desired, a function to be maximized can be negated and the resulting function considered for minimization. Numerical optimization is often used in situations in which the function is not analytic and an exhaustive search over the parameter space is not feasible because the function is very expensive to evaluate, or because the parameter space is very large, or both.

Optimization techniques can be characterized by the type of function, the type of input parameters, the number and type of constraints, the type of the output value(s), and so forth. The function can be analytic, comprised of mathematical operations that can be evaluated given the input values. The function might, or might not, be continuous and differentiable. The input parameters and/or function value might be reals or could be restricted to integer values (*integer programming*) or might be a mixture of integers and reals (*mixed programming*). The function can be relatively simple to compute or it can be computationally

expensive to evaluate. The derivative might have similar variations. There might be constraints on the values that the input parameters can have (*constrained optimization*), and the function might have multiple output values. Because optimization can be applied in a variety of situations under a variety of assumptions, strategies come in many forms. As a result, optimization does lend itself to a nice single taxonomy.

B.9.1 Analytic Solution

Many approaches to optimization are merely extensions of basic calculus taught in high school physics or calculus class. A basic analytic approach usually involves finding the optimal input x^* that minimizes (or maximizes) a function $f(x)$ by computing a solution to $f'(x^*) = 0$ where the second derivative at x indicates a minimum ($f''(x^*) > 0$) or a maximum ($f'(x^*) < 0$). In this case, optimization needs to find the roots of the first derivative.

If $f''(x) > 0$ for all x, then the function is considered *convex*. Convex means that the function's slope is always increasing. This is important because it means that, assuming $f'(x)$ is negative at negative infinity, it will increase until it reaches zero and that, once $f'(x) = 0$, it will continue to increase as x increases; that is, it will never go back down to zero. This means that $f(x^*)$ is a minimum ($f'(x^*) = 0$ and $f''(x^*) > 0$) and there is no other minimum because $f'(x)$ will never get back down to zero; that is, x^* is a *global* minimum of the function. Similarly, if $f''(x) < 0$ for all x, then the function is *concave* and x^* is a global maximum. It should be pointed out that if the derivative function is not easily available or computable, then approximations to the derivative by finite differencing can be used with a corresponding degree of inaccuracy. Functions that are not convex do not necessarily have a global minimum. In such cases, the initial guess is important as this will dictate which, if any, minimum is found.

B.9.2 Numerical methods

At the risk of oversimplifying the area, there are two basical approaches to numerical optimization. The first simply randomly samples values of the function and adaptively samples more often in areas of the parameter space where lower values are found. This approach is usually used for functions in a high dimensional space where there are many local optima, so the random sampling hopes to uncover the global optimum or at least an acceptable near-global optimum. This is guaranteed to succeed only in the limit (i.e., taking an infinite number of samples). The second approach to numerical optimization uses knowledge about local trends of the function value to search in a particular direction. The direction is based on derivatives of the function or finite difference approximations to the derivative. The derivative indicates which direction to follow in order to reduce (or increase) the function value. The local minimum (maximum) is found when the derivative is equal to zero and the second derivative is positive (negative).

Gradient information can be used to direct the search of parameter space to drive the function to a minimum or drive the derivative to zero. In cases where $f'(x) = 0$ cannot be solved analytically, numerical techniques can be used to find its zero values. The Taylor series expansion of a function $g(x + dx)$ is shown in Equation B.161 using the big O notation as a function of x^2 to represent the error term when using the first two terms as an approximation to $g(x + dx)$.

$$g(x + dx) = g(x) + g'(x)dx + O(dx^2) \tag{B.161}$$

If $x0$ is an initial guess, $x1 = x0 - a^*g'(x0)$ can be computed if $g'(x)$ can be computed or is estimated. With the objective of driving the value of $g()$ to zero with successive guesses, the Taylor approximation

(without the error term) is set to zero and rearranged to get Equation B.162. Thus, a succession of guesses are generated using Equation B.163.

$$\begin{aligned} 0 &= g(x) + g'(x)dx \\ dx &= x_1 - x_0 = -\frac{g(x)}{g'(x)} \end{aligned} \tag{B.162}$$

$$x_n = x_{n-1} - \frac{g(x)}{g'(x)} \tag{B.163}$$

In the current case, $f'(x) = g(x)$. If x is multidimensional, as it usually is in computer graphics applications, then $g(x)$ is the gradient of f(x) (Eq. B.164) and $-g'(x)$ is the inverse of the Hessian (Eq. B.165), but the same strategy holds and becomes Equation B.166.

$$\nabla f(\mathrm{x}) = \left[\frac{\partial f}{\partial x_1}(\mathrm{x}), \frac{\partial f}{\partial x_2}(\mathrm{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathrm{x})\right] \tag{B.164}$$

$$H(x) = \left[\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathrm{x})\right] \tag{B.165}$$

$$\mathrm{x}_n = \mathrm{x}_{n-1} - [Hg(\mathrm{x}_{n-1})]^{-1} \nabla g(\mathrm{x}_{n-1}) \tag{B.166}$$

The negative of gradient is used because the gradient points in the direction of the greatest increase. This is called the *gradient descent method*.

It has been shown that always taking steps in the direction of the gradient can lead to situations that converge very slowly. Therefore, taking steps in directions orthogonal to previous steps helps the convergence. This is called the *conjugate gradient method*.

An alternative way to approach the minimization problem is to fit a quadratic approximation to the function f at x, and then solve the quadratic equation for its minimum point and use that in the next iteration. The Taylor series expansion of G is shown in Equation B.167.

$$f(x + dx) = f(x) + f'(x)(dx) + f''(x)\frac{(dx)^2}{2} + O((dx)^3) \tag{B.167}$$

The quadratic approximation can be easily solved for the minimum value. This is used as the next guess for the minimum and the procedure iterates. In this way, steps are taken toward the minimum of the function using the first and second derivative of the objective function.

So far, constraints on the parameter space have not been considered, and this is called unconstrained optimization. Often, there are constraints on valid input values, such as greater than zero.

Hard and soft constraints, equality constraints, and inequality constraints

The simplest constrained optimization is when the function is linear and the constrains are linear, which is referred to as linear programming. Think of a panel that is tilted representing the linear objective function, and a frame consisting of 2×4s, the constraints, bounding a region of the board. A well-known method of solving a linear programming problem is the simplex method. Essentially, this method says that the constrained minimum must occur at a corner point of the frame. The simplex method progresses from corner to corner, always going in a direction that reduces the objective function.

Nonlinear, constrained optimization is where things get interesting—and complicated. One strategy is to take a step in parameter space to reduce the objective function regardless of the constraints, followed by a step to restore constraint satisfaction.

A different common method for considering constraints is to incorporate constraints in the objective function. This adds a dimension to the parameter space. Assume $f(x)$ is the objective function and $g(x) - c = 0$ is a constraint. The new objective function is shown in Equation B.168, where λ is an unknown constant called the *Lagrange multiplier*. Differentiating produces Equation B.169. To deal with more complex constraint situations, *sequential quadratic programming* can be used.

$$\text{minimize } f(\mathrm{X}) + \lambda(c(\mathrm{X}) - d) \tag{B.168}$$

$$\begin{aligned} \frac{\partial F(\mathrm{X})}{\partial \mathrm{X}} &= \nabla f(\mathrm{X}) + \lambda \nabla c(\mathrm{X}) = 0 \\ \frac{\partial F(\mathrm{X})}{\partial \lambda} &= c(\mathrm{X}) - d = 0 \end{aligned} \tag{B.169}$$

If evaluation of the derivative of the function is expensive or unavailable, or if the parameter space is expansive with many local minima, the parameter space can be randomly sampled.

Completely random sampling is blended with increased sampling in areas where low values are found. Early sampling favors randomness while later sampling favors sampling in low value areas. Maintaining some randomness ensures that the global minimum will be found in the limit.

This is the basis for the approaches such as *genetic algorithms* and *genetic programming*. They are generally simple to program, but with finite resources do not guarantee convergence on the optimum. They are very useful when finding "good" values are sufficient.

B.10 Standards for moving pictures

When producing computer animation, one must decide what format to use for storing the sequence of images. Years ago the images were captured on film by taking pictures of refresh vector screens or by plotting the images directly onto the film. This was a long and expensive process requiring single-frame film cameras and the developing of nonreusable film stock. The advent of frame buffers and video encoders made recording on videotape a convenient alternative. With today's cheap disks, memory, and CPU cycles, all-digital desktop video production is a reality. While the entertainment industry is still based on film, most of the rest of computer animation is produced using digital images intended to be displayed on a raster-scan device in such forms as broadcast video, DVD video, animated Web banners, and streaming video. This section is intended to give the reader some idea of the various standards used for recording moving pictures. Standards related to the digital image are emphasized.

B.10.1 In the beginning, there was analog

Before the rise of digital video, the two common formats for moving pictures were film and (analog) video. Over the years there have been almost a hundred film formats. The formats differ in the size of the frame, in the placement, size, and number of perforations, and in the placement and type of audio tracks [15]. Silent film is played at 16 frames per second (fps). Some sound film is played at 18 fps, but 24 fps is more common. Film played at 24 fps is typically doubly projected; that is, each frame is displayed twice to reduce the effects of flicker.

Table B.5 Film Formats

Film Width	Notes
8 mm	An old format, introduced in 1932, 8 mm is used for inexpensive home movies. Cameras for regular 8 mm are no longer manufactured. Regular 8 mm uses 16 mm stock, which is recorded on both sides after flipping the film in the camera. This allows the 16 mm film stock to be split down the middle to produce two 8 mm reels. The frame is 0.192 × 0.145″. Super-8 was introduced in 1965 as an improvement over regular 8 mm. The perforations (the holes in the film stock used to advance and register the film) were made smaller and the frame size was increased to 0.224 × 0.163″. The film was placed into cassettes instead of on the reels of regular 8 mm film.
16 mm	16 mm is used for television and low-budget theatrical productions. It was introduced in 1923 and has a frame size of 0.404 × 0.295″.
35 mm	35 mm has been a standard film size since the turn of the twentieth century [19]. It first became popular because it could be derived from the original 70 mm film made by Kodak. It is the standard for theatrical work as well as television. The *standard academy frame*, the most popular of several 35 mm formats, is 0.864 × 0.630″.
65 mm	65 mm was the standard format for large-format cinematography. It is now gaining in popularity in special-venue and "ride" films.
70 mm	70 mm film is often a blow-up print of 35 mm film, produced for improved audio, better registration, and less grain of the release print. With better sound technology (e.g., digital) and the advent of multiplex theaters with smaller screen sizes, there is less demand for this type of 70 mm film. However, IMAX uses 70 mm film (69.6 × 48.5 mm) that is run at 24 fps.

Some of the film sizes (widths) are listed in Table B.5. Note that there are often several formats for each film size. Only the most popular film formats are listed here. See the Web pages of Norwood [15] and Rogge [19] for more information. With the rise of desktop video production, film is less of an issue for home-brew computer animation, although it remains useful for conventional animation and, of course, is still the standard medium for display of feature-length films in theaters, although even this is starting to change.

Broadcast video standard

In 1941, the National Television Standards Committee (NTSC) established 525-line, 60.00 Hz field rate, 2:1 interlaced monochrome television in the United States. In 1953, 525-line, 59.94 Hz field rate, 2:1 interlaced, composite color television signals were established as a standard. The image is displayed top to bottom, with each scanline displayed left to right. *2:1 interlaced* refers to the scanning pattern, with the information on the odd scanlines followed by the information on the even scanlines. Each set of scanlines is referred to as a *field*; there are two fields per *frame*. This standard is typically referred to by the initials of the committee—NTSC. Broadcast video in the United States must correspond to this standard. The standard sets a specific duration for a horizontal scanline, a frame time, the amplitude and duration of the various sync pulses, and so on. Home video recording units typically generate much sloppier signals and would not qualify for broadcast. There are encoders that can strip old sync signals off a video signal and re-encode it so that it conforms to broadcast quality standards.

There are a total of 525 scanline times per frame time in the NTSC format. The number of frames transmitted per second is 29.97. There is a 2:1 interlace of the scanlines in alternate fields. Of the 525 total scanline times, approximately 480 contain picture information. The remainder of the scanline times are occupied by the overhead involved in the scanning pattern: the time it takes the beam to

Table B.6 Video Format Comparison [10]

Standard	Lines	Scan Pattern	Field Rate (Hz)	Aspect Ratio
NTSC	525	2:1 interlaced	59.94	4:3
PAL	625	2:1 interlaced	50	4:3
SECAM	625	2:1 interlaced	50	4:3

go from the end of one scanline to the beginning of the next and the time it takes for the beam to go from the bottom of the image to the top. The aspect ratio of a 525-line television picture is 4:3, so equal vertical and horizontal resolutions are obtained at a horizontal resolution of 480 times 4/3, or 640 pixels per scanline. PAL and SECAM are the other two standards in use around the world (Table B.6). They differ from NTSC in specifics like the number of scanlines per frame, the field rate, and the frequency of the color subcarrier, but both are interlaced raster formats. One of the reasons that television technology uses interlaced scanning is that, when a camera is providing the image, the motion is updated every field, thus producing smoother motion.

Black-and-white signal

A black-and-white video signal is basically a single line that has the sync information and intensity signal (luminance) superimposed on one signal. The vertical and horizontal sync pulses are negative with respect to a reference level, with vertical sync a much longer pulse than horizontal sync. On either side of the sync pulses are reference levels called the front porch and the back porch. The active scanline interval is the period between horizontal sync pulses. During the active scanline interval, the intensity of the signal controls the intensity of the electron beam of the monitor as it scans out the image.

Incorporating color into the black-and-white signal

When color came on the scene in broadcast television, engineers were faced with incorporating the color information in such a way that black-and-white television sets could still display a color signal and color sets could still display black-and-white signals. The solution is to encode color into a high-frequency component that is superimposed on the intensity signal of the black-and-white video.

A reference signal for the color component, called the *color burst*, is added to the back porch of each horizontal sync pulse, with a frequency of 3.58 MHz. The color is encoded as an amplitude and phase shift with respect to this reference signal. A signal that has separate lines for the color signals is referred to as a *component* signal. A signal such as the color TV signal with all of the information superimposed on one line is referred to as a *composite* signal.

Because of the limited room for information in the color signal of the composite signal, the TV engineers optimized the color information for a particular hue they considered most important: Caucasian skin tone. Because of that, the RGB information has to be converted into a different color space: YUV. Refer to Figure B.59. Y is luminance and is essentially the intensity information found in the black-and-white signal. It is computed by Equation B.170.

$$Y = 0.299R + 0.587G + 0.114B \tag{B.170}$$

The U and V of television are scaled and filtered versions of B-Y and R-Y, respectively. U and V are used to modulate the amplitude and phase shift of the 3.58 MHz color frequency reference signal. The phase of this chroma signal, C, conveys a quantity related to hue, and its amplitude conveys a quantity related to color saturation. In fact, the I and Q stand for "in phase" and "quadrature," respectively. The

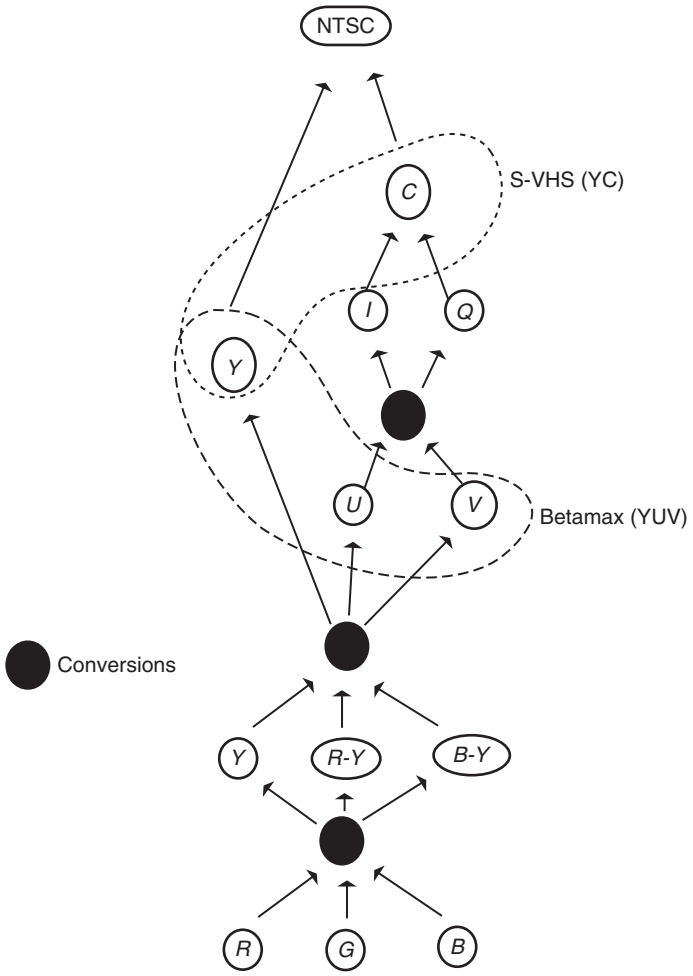


FIGURE B.59

Video signal.

NTSC system mixes *Y* and *C* together and conveys the result on one piece of wire. The result of this addition operation is not theoretically reversible; the process of separating luminance and color often confuses one for the other (e.g., the appearance of color patterns seen on TV shots of people wearing black-and-white seersucker suits).

Videotape formats

The size and speed of the tape and the encoding format contribute to the quality that can be supported by a particular video format. Common tape sizes are 1/2, 3/4, 1, and 2″. The first two sizes are of cassette tapes, and the other two are of open reel tapes. One-half inch supports consumer-grade formats (e.g., VHS, S-VHS, and the less popular Betamax), 3/4″ is industrial strength (U-Matic), and 1 and 2″ are professional broadcast-quality formats. One inch is the newer technology and has replaced some 2″ systems.

The common 1/2″ video formats are VHS and S-VHS. S-VHS is a format in which the *Y* and *C* signals are kept separate when played back, thus avoiding the problems created when the signals are superimposed. All video equipment actually records signals this way, but S-VHS allows the *Y* signal (luminance) to be recorded at a higher resolution than color. The color information is recorded with the same fidelity as on VHS. In addition, the sound is encoded differently than on regular VHS, also resulting in greater fidelity. The advantages of S-VHS are especially pronounced when it is played back on an S-VHS-compatible television.

B.10.2 In the digital world

Full-color (24 bits per pixel), video resolution (640 × 480), video rate (30 fps) animation requires approximately 1.6 gigabytes of information per minute of animation when stored as uncompressed RGB images. The problem is how to store and play back the animation. Various trade-offs must be considered when choosing a format, including the amount of compression, the time it takes for the compression/decompression, and the color and spatial resolution supported. The objective here is to provide an overview of the terminology, the important issues, and the most popular standards in recording animation. The discussion is at the level of consumer-grade technology and is not intended for professionals involved in the production of high-quality digital material.

Digital video sometimes refers specifically to digital versions of video to be broadcast for reception by television sets. In some literature this is also referred to as *digital TV* (DTV), which is the term used here. On the other hand, *digital video* (DV) is also used in the sense of computer-generated moving images intended to be played back on an RGB computer monitor. DV is used here to specifically denote material intended to be displayed by a computer. These two categories of digital representations, DTV and DV, have much in common. One of the most important common issues is the use of compression/decompression (codec) technology described in the following paragraphs.

Standards related to DTV have two additional concerns. First, the standards are concerned with images that will be encoded for broadcast. As a result, they rarely deal directly with RGB images. When destined for television, digital images are at least partially encoded in a format related to the broadcast video standard (NTSC) soon after they leave the camera (e.g., YUV). DTV has the advantage of encoding a better image (480p) in less bandwidth than its analog counterpart. As a result, DTV can broadcast additional information in the bandwidth given a channel.

Digital images synthesized for playback on a computer are typically generated as RGB images, compressed for storage and transmission, and then decompressed for playback on RGB monitors. Second, DTV standards are concerned with an associated recording format of the images and audio on tape; tape is the common storage medium for broadcast video studios. Because images for computer playback are typically stored digitally on a hard disk or removable disk, DV standards do not cover tape formats. However, there are DV standards for file formats and for digital movies. Digital movie formats organize the image and audio data into tracks with associated timing information.

Compression/decompression

The recent explosion in multimedia applications, especially as a result of the Web, has led to the development of a variety of compression/decompression schemes [25]. After the frames of an animation are computed, they are compressed and stored on a hard disk or CD-ROM. When playback of the animation is requested, the compressed animation is sent to the compute box, using disk I/O or over the Web, where the frames are decompressed and displayed. The decompression and playback can take place in real time

as the data is transmitted. In this case, it is referred to as *streaming video*. If the decompression is not fast enough to support streaming video, the compressed animation is transmitted in its entirety to the compute box, decompressed into the complete animation, and then played back. In either case, the compression not only saves space on the storage device but also allows animation to be transferred to the computer much faster. Several of the codecs are proprietary and are used in workstation-based video products. The different schemes have various strengths and weaknesses and thus involve trade-offs, the most important of which are compression level, quality of video, and compression/decompression speed [21].

The amount of compression is usually traded off for image quality. With some codecs, the amount of compression can be set by a user-supplied parameter so that, with a trial-and-error process, the best compression level for the particular images at hand can be selected. Codecs with greater compression levels usually incorporate interframe compression as well as intraframe compression. *Intraframe compression* means that each frame is compressed individually. *Interframe compression* refers to the temporal compression possible when one processes a series of similar still images using techniques such as image differencing. However, when one edits a sequence, interframe compression means that more frames must be decompressed and recompressed to perform the edits.

The quality of the video after decompression is, of course, a big concern. The most fundamental feature of a compression scheme is whether it is *lossless* or *lossy*. With lossless compression, in which the final image is identical to the original, only nominal amounts of compression can be realized, usually in the range of 2:1. The codecs commonly used for video are lossy in order to attain the 500:1 compression levels necessary to realize the transmission speeds for pumping animations over the Web or from a CD-ROM. To get these levels of compression, the quality of the images must be compromised. Various compression schemes might do better than others with such image features as edges, large monochrome areas, or complex outdoor scenes. The amount of color resolution supported by the compression is also a concern. Some, such as animated GIF format, support only 8-bit color.

The compression and decompression speed is a concern for obvious reasons, but decompression speed is especially important in some applications, for example, streaming video. On the other hand, real-time compression is useful for applications that store compressed images as they are captured. If compression and decompression take the same amount of time, the codec is said to be *symmetric*. In some applications such as streaming video, it is acceptable to take a relatively long time for compression as long as the resulting decompression is very fast. In the case of unequal times for compression and decompression, the codec is said to be *asymmetric*. To attain acceptable decompression speeds on typical compute boxes, some codecs require hardware support.

A variety of compression techniques form the basis of the codec products. *Run-length encoding* is one of the oldest and most primitive schemes that have been applied to computer-generated images. Whenever a value repeats itself in the input stream, the string of repeating values is replaced by a single occurrence of the value along with a count of how many times it occurred. This was sufficient for early graphic images, which were simple and contained large areas of uniform color. This technique does not perform well with today's complex imagery. The Lempel-Ziv-Welch (LZW) technique was developed for compressing text. As the input is read in, a dictionary of strings and associated codes is generated and then used as the rest of the input is read in. *Vector quantization* simply refers to any scheme that uses a sample value to approximate a range of values. *YUV-9* is a technique in which the color is undersampled, so that, for example, a single color value is recorded for each 4×4 block of pixels. Discrete cosine transform (DCT) is a very popular technique that breaks a signal into a sum of cosine functions of various frequencies at specific amplitudes. The signal can be compressed by throwing away low-amplitude and/or high-frequency components. DCT is an example of the more general *wavelet compression* in which the form of the base

component function can be selected according to its properties. *Fractal compression* is based on the fact that many signals are self-similar under scale. A section of the signal can be composed of transformed copies of other sections of the signal. This is applied to rectangular blocks of the image. In fact, most of the compression schemes break the image into blocks and then compress the blocks.

Codecs employ one or more of the techniques mentioned earlier. The names of some of the more popular codecs are Video I, RLE, GIF, Motion JPEG, MPEG, Cinepak, Sorenson, and Indeo 3.2. Microsoft products are Video I and RLE. Video I employs DCT compression, and RLE uses run-length encoding. GIF is an 8-bit color image compression scheme based on the LZW compression. JPEG uses DCT compression.

Motion JPEG (MJPEG) is simply the application of JPEG for still images applied to a series of images. The compression/decompression is symmetric and is done in 1/30 of a second. JPEG compression can introduce some artifacts into some images with hard edges. The Moving Pictures Expert Group (MPEG) standards were designed specifically for digital video. MPEG uses the same algorithms as JPEG for intraframe compression of individual frames, called I-frames. MPEG then uses interframe compression to create B (bidirectional) and P (predicted) frames. MPEG allows quality settings to specify the amount of compression to use [24] and can be set individually for each frame type (I, P, B). MPEG-1 is designed for high-quality CD video, MPEG-2 is designed for high-quality DVD video, and MPEG-4 is designed for low-bandwidth Web applications [22].

Cinepak and Sorenson are products initially targeted for the MAC world, although Cinepak is now available for the PC. Cinepak uses block-oriented vector quantization. Sorenson uses YUV compression with 4×4 blocks and employs interframe compression [20]. Indeo 3.2 is an Intel product that also uses block-oriented vector quantization. Cinepak and Indeo are highly asymmetric, requiring on the order of 300 times longer for compression than for their efficient decompression. Indeo also incorporates color blending and run-length encoding into its scheme.

Digital video formats

The codec products (as opposed to the underlying codec techniques) used for DV have an associated file format. Some formats also include timing information, the ability to animate overlay images (sprites), and the ability to loop over a series of frames. MPEG and MJPEG are both common DV formats. GIF89a-based animation (animated GIF) is basically a number of GIF images stored in one file with interframe timing information but no interframe compression. Compressing continuous-tone images can result in color banding. The compression does well on line drawings but not on complex outdoor scenes. GIF animations can use delta frames, which, for example, overlay images on a previously transmitted background. This saves retransmitting static information for some animations. As used here, *movie format* refers to a format that is codec independent and that can handle audio as well as imagery. Both Quicktime (MOV) and Video for Windows (AVI) [2] are movie formats designed as open codec architectures. Any codec can be used with these standards as long as a compatible plug-in is available. Cinepak has been a standard codec used with Quicktime, but Quicktime can also accommodate other codecs such as Sorenson and JPEG [24]. Quicktime organizes data into tracks and includes timing information. Video for Windows from Microsoft uses Video I and RLE as standard codecs but can also accommodate others. Video for Windows allows interleaving of image and audio information.

Digital television formats

Most of the DTV formats are based on sampling scaled versions of the color difference signals (B-Y, R-Y). Luminance and the scaled color difference signals are referred to as YUV. Common formats are D1, D2, D3, D5, D6, Digital Betacam, Ampex DCT, Digital8, DV, DVCam, and DVCPRO. When

digitally sampled, the YUV signals are referred to as YCrCb. A typical sampling scheme is 4:2:2, in which the color difference signals (Cr, Cb) are sampled at half the sampling rate of luminance (Y) in the horizontal direction. 4:1:1 sampling means that the color difference signals are also sampled at half the rate in the vertical direction, resulting in one-quarter of the luminance sampling.

The D1 standard was developed when the broadcast television industry thought it would make the composite-analog-to-component-digital transition in one fell swoop. But that did not happen because the cost was prohibitive. D1 uses YUV coding, so-called 4:2:2, which means that the U and V components are horizontally subsampled 2:1. Luminance is sampled at 13.5 MHz, resulting in 720 samples per picture width. There is no compression other than undersampling the chroma information. Aggregate data rate is roughly 27 megabytes per second (MB/s). The D2 standard was developed as a low-cost alternative to D1. D2 is a composite NTSC digital format (i.e., digitized NTSC). The composite signal is sampled at four-times-color-subcarrier, about 14.318 MHz at one byte per sample (aggregate data rate, of course, 14.318 MB/s). It has all the impairment of NTSC but the reliability and performance of digital. It uses the same 3/4″ cassette as D1. As with D1, D2 uses no compression other than undersampling the chroma information. Other uncompressed digital formats include D3, D5, and D6. D3 is a composite format that uses 1/2″ tape. D5 is a component format that uses 1/2″ tape. D6 is a component HDTV format.

Common compressed DTV formats include Digital Betacam, Ampex DCT, and Digital8 [9]. Digital Betacam uses 1/2″ tapes similar to the Betacam SP format with 2:1 compression based on DCT. Ampex DCT is a proprietary format; the *DCT* in its name stands for Digital Component Technology and not the compression scheme. The trio of DV, DVCam, and DVCPRO are similar formats using DCT compression. Depending on image content, the encoder decides whether to compress two fields separately or as a unit. Digital8 is a consumer-grade version of the DV format but uses cheaper Hi8 tapes. Newer formats include W-VHS, Digital S, Betacam SX, Sony HDD-1000, and D-VHS.

High-definition television and wide-screen format

In the United States the Grand Alliance was created by the FCC to develop the American HDTV specification. It is most commonly based on the MPEG-2 codec [8]. The basic idea behind high-definition television (HDTV) is to increase the percentage of the visual field occupied by the image [12]. HDTV is a type of DTV. Currently, there are two popular wide-screen HDTV formats: 720p and 1080i. The number refers to how many scanlines there are in an image. The "p" in 720p refers to progressive scan; the "i" in 1080i refers to interlaced scan. NTSC-format television has an aspect ratio (ratio of width to height) of 4:3, whereas the widescreen format more closely matches that found in movie theaters and is 16:9.

B.11 Camera calibration

For digitally capturing motion from a camera image, one must have a transformation from the image coordinate system to the global coordinate system. This requires knowledge of the camera's intrinsic parameters (focal length and aspect ratio) and extrinsic parameters (position and orientation in space). In the capture of an image, a point in global space is projected onto the camera's local coordinate system and then mapped to pixel coordinates. To establish the camera's parameters, one uses several points whose coordinates are known in the global coordinate system and whose corresponding pixel

coordinates are also known. By setting up a system of equations that relates these coordinates through the camera's parameters, one can form a least-squares solution of the parameters [23].

Calibration is performed by imaging a set of points whose global coordinates are known and identifying the image coordinates of the points and the correspondence between the two sets of points. This results in a series of five-tuples, $(x_i, y_i, z_i, c_i, r_i)$ consisting of the three-dimensional global coordinates and two-dimensional image coordinates for each point. The two-dimensional image coordinates are a mapping of the local two-dimensional image plane of the camera located a distance f in front of the camera (Eq. B.171). The image plane is located relative to the three-dimensional local coordinate system (u, v, w) of the camera (Figure B.60). The imaging of a three-dimensional point is approximated using a pinhole camera model. The three-dimensional local coordinate system of the camera is related to the three-dimensional global coordinate system by a rotation and translation (Eq. B.172); the origin of the camera's local coordinate system is assumed to be at the focal point of the camera. The three-dimensional coordinates are related to the two-dimensional coordinates by the transformation to be determined. Equation B.173 expresses the relationship between a pixel's column and row number and the global coordinates of the point. These equations are rearranged and set equal to zero in Equation B.174. They can be put in the form of a system of linear equations (Eq. B.175) so that the unknowns are isolated (Eq. B.176) by using substitutions common in camera calibration (Eqs. B.176 and B.177). Temporarily dividing through by t_3 ensures that $t_3 \neq 0.0$ and therefore that the global origin is in front of the camera. This step results in Equation B.178, where A' is the first 11 columns of A; B' is the last column of A; and W' is the first 11 rows of W. Typically, enough points are captured to ensure an overdetermined system. Then a least-squares method, such as singular value decomposition, can be used to find the W' that satisfies Equation B.179. W' is related to W by Equation B.180, and the camera parameters can be recovered by undoing the substitutions made in Equation B.176 by Equation B.181. Because of numerical imprecision, the rotation matrix recovered may not be orthonormal, so it is best to reconstruct the rotation matrix first (Eq. B.182), massage it into orthonormality, and then use the new rotation matrix to generate the rest of the parameters (Eq. B.183).

$$\begin{aligned} c_i - c_0 &= s_u u_i \\ r - r_0 &= s_v v_i \end{aligned} \tag{B.171}$$

$$\begin{bmatrix} u_i \\ v_i \\ f \end{bmatrix} = R \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} + T$$

$$\begin{aligned} R &= \begin{bmatrix} R_0 \\ R_1 \\ R_2 \end{bmatrix} = \begin{bmatrix} r_{0,0} & r_{0,1} & r_{0,2} \\ r_{1,0} & r_{1,1} & r_{1,2} \\ r_{2,0} & r_{2,1} & r_{2,2} \end{bmatrix} \\ T &= \begin{bmatrix} t_0 \\ t_1 \\ t_2 \end{bmatrix} \end{aligned} \tag{B.172}$$

$$\begin{aligned} \frac{u_i}{f} &= \frac{c_i - c_0}{s_u f} = \frac{c_i - c_0}{f_u} = \frac{R_0 \cdot [x_i y_i z_i] + t_0}{R_2 \cdot [x_i y_i z_i] + t_2} \\ \frac{v_i}{f} &= \frac{r_i - r_0}{s_v f} = \frac{c_i - c_0}{fv} = \frac{R_1 \cdot [x_i y_i z_i] + t_1}{R_2 \cdot [x_i y_i z_i] + t_2} \end{aligned} \tag{B.173}$$

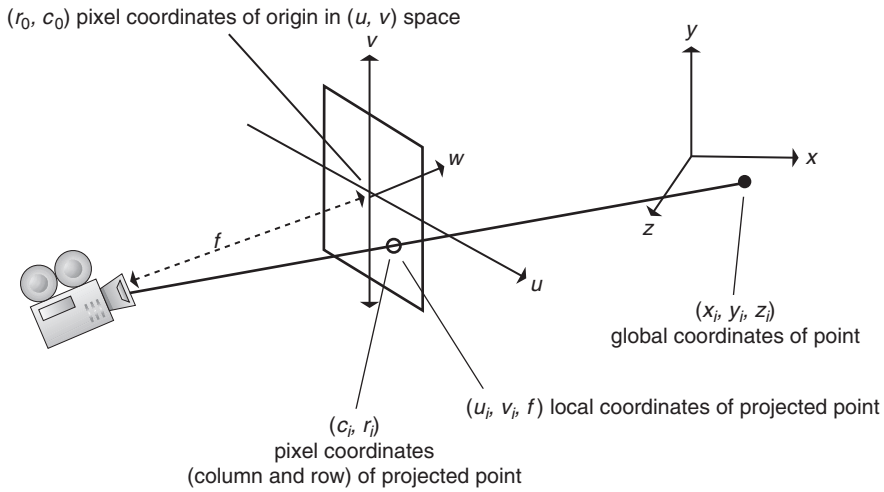


FIGURE B.60

Coordinate systems used in the projection of a global point to pixel coordinates.

$$\begin{aligned}(c_i - c_0)(R_2 \cdot [x_i y_i z_i] + t_2) - f_u(R_0 \cdot [x_i y_i z_i] + t_0) = 0\\ (r_i - r_0)(R_2 \cdot [x_i y_i z_i] + t_2) - f_v(R_1 \cdot [x_i y_i z_i] + t_1) = 0\end{aligned} \tag{B.174}$$

$$AW = 0 \tag{B.175}$$

$$\begin{bmatrix} -x_1 & -y_1 & -z_1 & 0 & 0 & 0 & r_1x_1 & r_1y_1 & r_1z_1 & -1 & 0 & r_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -z_1 & c_1x_1 & c_1y_1 & c_1z_1 & 0 & -1 & c_1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -x_1 & -y_n & -z_n & 0 & 0 & 0 & r_nx_n & r_ny_n & r_nz_n & -1 & 0 & r_n \\ 0 & 0 & 0 & -x_n & -y_n & -z_n & c_nx_n & c_ny_n & c_nz_n & 0 & -1 & c_n \end{bmatrix}$$

$$\begin{aligned}W_0 &= f_u R_0 + c_0 R_2 = [w_0 w_1 w_2]^T \\ W_3 &= f_v R_1 + r_0 R_2 = [w_3 w_4 w_5]^T \\ W_6 &= R_2 = [w_6 w_7 w_8]^T \\ w_9 &= f_u t_0 + c_0 t_2 \\ w_{10} &= f_v f_1 + r_0 t_2 \\ w_{11} &= t_2\end{aligned} \tag{B.176}$$

$$W = [w_0\ w_1\ w_2\ w_3\ w_4\ w_5\ w_6\ w_7\ w_8\ w_9\ w_{10}\ w_{11}]^T \tag{B.177}$$

$$A'W' + B' = 0 \tag{B.178}$$

$$\min_w \|A'W' + B'\| \tag{B.179}$$

$$W = \begin{bmatrix} W_0 \\ W_3 \\ W_6 \\ W_9 \\ W_{10} \\ W_{11} \end{bmatrix} = \frac{1}{\|W_6^{'}\|} \times \begin{bmatrix} W_0^{'} \\ W_3^{'} \\ W_6^{'} \\ W_9^{'} \\ W_{10}^{'} \\ 1 \end{bmatrix} \tag{B.180}$$

$$\begin{aligned} c_0 &= W_0^T W_6 \\ r_0 &= W_1^T W_6 \\ f_u &= -\|W_0 - c_0 W_6\| \\ t_0 &= (w_9 - c_0)/f_u \\ t_1 &= (w_{10} - r_0)/f_v \\ t_2 &= w_{11} \\ R_0 &= (W_0 - c_0 W_6)/f_u \\ R_1 &= (W_3 - r_0 W_6)/f_v \\ R_2 &= W_6 \end{aligned} \tag{B.181}$$

$$\begin{aligned} R_0 &= (W_0^{'} - c_0 W_6^{'})/f_u \\ R_1 &= (W_3^{'} - r_0 W_6^{'})/f_v \\ R_2 &= W_6^{'} \end{aligned} \tag{B.182}$$

$$\begin{aligned} c_0 &= W_0^T R_2 \\ r_0 &= W_1^T R_2 \\ f_u &= -\|W_0 - c_0 R_2\| \\ f_v &= \|W_3 - r_0 R_2\| \\ t_0 &= (W_9 - c_0)/f_u \\ t_1 &= (W_{10} - r_0)/f_v \\ t_2 &= w_{11} \end{aligned} \tag{B.183}$$

Given an approximation to a rotation matrix, $\tilde{\mathrm{R}}$, the objective is to find the closest valid rotation matrix to the given matrix (Eq. B.184). This is of the form shown in Equation B.185, where the matrices C and D are notated as shown in Equation B.186. To solve this, define a matrix B as in Equation B.187. If $q = (q_0, q_1, q_2, q_3)^T$ is the eigenvector of B associated with the smallest eigenvalue, R is defined by Equation B.188 [26].

$$\min\|\tilde{R} - R\| \tag{B.184}$$

$$\|R \cdot C - D\| \tag{B.185}$$

$$\begin{aligned} C &= [C_1, C_2, C_3] \\ D &= [D_1, D_2, D_3] \end{aligned} \tag{B.186}$$

$$
\begin{aligned}
B &= \sum_{i=1}^{3} B_i^T B_i \\
B_i &= \begin{bmatrix} 0 & (C_i - D_i)^T \\ D_i - C_i & [D_i - C_i]_x \end{bmatrix} \\
[(x, y, z)]x^* &= \begin{bmatrix} 0 & -z & y \\ -z & 0 & -x \\ -y & x & 0 \end{bmatrix}
\end{aligned}
\tag{B.187}
$$

$$
R = \begin{bmatrix} q_0 - q_1 + q_2 - q_3 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & q_0 - q_1 + q_2 - q_3 & 2(q_3 q_2 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_3 q_1 + q_0 q_2) & q_0 - q_1 - q_2 + q_3 \end{bmatrix} \tag{B.188}
$$

References

[1] Bartels R, Beatty J, Barsky B. An Introduction to Splines for Use in Computer Graphics and Geometric Modeling. San Francisco: Morgan-Kaufmann; 1987.

[2] Dixon D. AVI Formats, http://www.manifest-tech.com/pc_video/avi_formats/avi_formats. htm**; January 2001.**

[3] Ebert D, editor. Texturing and Modeling: A Procedural Approach. 2nd ed. Boston: AP Professional; 1998.

[4] Farin G. Curves and Surfaces for Computer Aided Design. Orlando, Fla.: Academic Press; 1990.

[5] Frautsch S, Olenick R, Apostol T, Goodstein D. The Mechanical Universe: Mechanics and Heat, the Advanced Edition. Cambridge: Cambridge University Press; 1986.

[6] Gasch S. 0.5.12.0 Source Code, http://wannabe.guru.org/alg/node136.html**; January 2001.**

[7] Gottschalk S, Lin M, Manocha D. OBB Tree: A Hierarchical Structure for Rapid Interference Detection. In: Rushmeier H, editor. Computer Graphics. Proceedings of SIGGRAPH 96, Annual Conference Series. New Orleans, La.: Addison-Wesley; August 1996. p. 171–80. ISBN 0-201-94800-1.

[8] Hromas D. Digital Television: The Television System of the Future . . ., http://www.cgg. cvut.cz/~xhromas/dtv/**; January 2001.**

[9] Iisakkila M. Video Recording Formats, http://www.hut.fi/u/iisakkil/videoformats.html**; January 2001.**

[10] King B. TV Systems: A Comparison, http://www.ee.surrey.ac.uk/Contrib/WorldTV/-compare.html**; January 2001.**

[11] Kochanek D. Interpolating Splines with Local Tension, Continuity and Bias Control. In: Computer Graphics. Proceedings of SIGGRAPH 84, vol. 18(3). Minneapolis, Minn.; July 1984. p. 33–41.

[12] Kuhn K. HDTV Television—an Introduction: EE 498, http://www.ee.washington.edu/conselec/CE/kuhn/hdtv/95x5.htm**; January 2001.**

[13] Madsen N. Three Dimensional Dynamics of Rigid Bodies, http://www.eng.auburn.edu/users/gflowers/me232/class_problems/angmom.html**; January 2001.**

[14] Mortenson M. Geometric Modeling. New York: John Wiley & Sons; 1985.

[15] Norwood S. rec.arts.movies.tech: Frequently Asked Questions (FAQ) Version 2.00, http://www.redballoon.net/~snorwood/faq2.html**; January 2001.**

[16] Press W, Flannery B, Teukolsky S, Vetterling W. Numerical Recipes: The Art of Scientific Computing. Cambridge: Cambridge University Press; 1986.
[17] Ritter J. An Efficient Bounding Sphere. In: Glassner A, editor. Graphics Gems. New York: Academic Press; 1990. p. 301–3.
[18] Rogers D, Adams J. Mathematical Elements for Computer Graphics. New York: McGraw-Hill; 1976.
[19] Rogge M. More Than One Hundred Years of Film Sizes, http://www.xs4all.nl/∼wichm/filmsize.html**; January 2001.**
[20] Sorenson Media. Sorenson Video Tutorial: Sorenson Video Compression, http://www. sorenson.com/Sorenson-Video/tutorial/page03.html**; January 2001.**
[21] Speights II L. Video Compression Methods (Codecs), http://home.earthlink.net/∼radse/Page9.html**; January 2001.**
[22] Terrain Interactive, Index of Codecs for Video, CD, DVD, and Audio, http://www. terran.com/CodecCentral/Codecs/index.html**; January 2001.**
[23] Tuceryan M, Greer G, Whitaker R, Breen D, Crampton C, Rose E, et al. Calibration Requirements and Procedures for a Monitor-Based Augmented Reality System. IEEE Trans Vis Comput Graph September 1995;1(3):255–73.
[24] Vahlenkamp H. GFDL Visualization Guide: Animation Formats, http://www.manifest-tech.com/pc_video/avi_formats/avi_formats.htm**; January 2001.**
[25] Waggoner B. Technology: A Web Video Guide to Codecs, http://www.dv.com/webvideo/ 2000/0900/waggoner0900.html**; January 2001.**
[26] Weng J, Cohen P, Herniou M. Camera Calibration with Distortion Models and Accuracy Evaluation. IEEE Trans Pattern Anal Mach Intell October 1992;14(10):965–80.
[27] Zordan V. Solving Computer Animation Problems with Numeric Optimization. GVU Technical Report GIT-GVU-02-13. Georgia Institute of Technology; 2002.
[28] Velho L, Carvalho P, Gomes J, de Figueiredo L. Mathematical Optimization in Computer Graphics and Vision. New York: Morgan Kaufmann; 2008.
[29] Bradley S, Hax A, Magnanti T. Applied mathematical Programming. New York: Addison Wesley; 1977.