An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 14 3D Objects from Planar Contours

In previous chapters, we have discussed how to construct a polygon mesh by specifying vertices and faces and how to use subdivision to obtain a finer mesh. Another common and simple way to create a mesh is by extrusion, which extrudes or sweeps a 2D face or a group of faces through space. By moving a uniform cross-section along a path in space, we obtain a 3D object. For example, extruding a square upward would create a cube connecting the bottom and the top faces. Thus, extrusion involves a 2D shape or a group of 2D shapes and straight line-segments in space. The cross-section of the extruded 3D object is determined by the 2D shape. In the extrusion process, we connect a starting point and the corresponding ending point of the extruded form in space by a line segment. We first duplicate the cross-section to create the top and bottom faces. Then we join the top and bottom faces together with parallel sides, which would generate outward-facing faces.

Extrusion is one of the general methods of constructing 3D surface from planar contours that represent cross sections of the objects. Besides 3D modeling, this field has applications in many scientific and engineering areas, including bio-medical applications, geology, archeology, and oceanogragphy. We may consider a contour as a continuous curve, which is the intersection of a plane and a surface.

14.1 Prisms

We first consider a simple example to illustrate the extrusion technique. We will sweep a cross-section along a straight line to obtain a prism. The cross-section we will use is composed of two triangles and a regular hexagon connected together as shown in (a) of Figure 14-1, where we assume that the cross-section lies in the xy-plane. To create a prism, we start with a **base** B consisting of the three polygons (two triangles and a hexagon) with a total of 12 vertices. We label each vertex of a polygon as $0, 1, ..., n-1$, where n is the number of vertices in the polygon as shown in (b) of Figure 14-1. The coordinates of each vertex of the base is $(x, y, 0)$. We can sweep B in any direction $\mathbf{D}$ but for simplicity, we sweep B along the z-direction; the sweep creates an edge and another vertex for each vertex of the base as shown in (c) of Figure 14-1. The new vertices form the **cap** of the prism. The coordinates of each vertex of the cap is (x, y, H) where H is the height of the prism. To draw the prism, we simply use an array, say, $base[][3]$ to hold the coordinates of the N vertices of the base and a corresponding array, say, $cap[][3]$ to hold the vertices of the cap. If $base[][3]$ has been specified, the following code segment gives the cap:

```
for ( int i = 0; i < N; i++ ){
  cap[i][0] = base[i][0];
  cap[i][1] = base[i][1];
  cap[i][2] = H;
}
```

Polygons are formed from subsets of the base or cap vertices. We can use indices to specify a polygon of the base. For example, a polygon with n vertices can be specified by the indices $i_0, i_1, ..., i_{n-1}$ that refer to the coordinates at $base[i_0], base[i_1], ..., base[i_{n-1}]$. We can easily draw the polygons of the base and the cap. A sweeping edge joins the vertices $base[i_j]$ with $cap[i_j]$. The construction of a side face or a wall is straightforward.

For the j-th wall ($j = 0, 1, ..., n-1$), we construct a face from two vertices of the base and two vertices of the cap having indices i_j and i_{j+1}, where the operation $j+1$ is taken $mod\ n$. (i.e., if the value of $j+1$ is equal to n, we set it to 0.) The normal to each face can be calculated by finding the cross product of two vectors pointing in the directions of two adjacent edges. The following code segment shows the implementation of this concept for our example cross-section:

```
void drawWalls ()
{
  int i, j, k, n;
  int sides[3] = {3, 6, 3};
  for ( i = 0; i < 3; i++ ) {
    n = sides[i];
    glBegin(GL_POLYGON);
     for ( j = 0; j < n; j++ ) {
        k = indexp[i][j];
        glVertex3dv ( cap[k] );
        glVertex3dv ( base[k] );
        int next = (j + 1) % n;
        k = indexp[i][next];
        glVertex3dv ( base[k] );
        glVertex3dv ( cap[k] );
     }
   glEnd();
  }
}
```

In this code, the array *sides* contains the number of sides each polygon has. (In our example, there are three polygons in the base or the cap as shown in Figure 14-1.) The array *indexp* is a global variable that holds the indices to the vertices of the polygons; *indexp*[i][j] refers to the j-th vertex of the i-th polygon of the base or the cap.

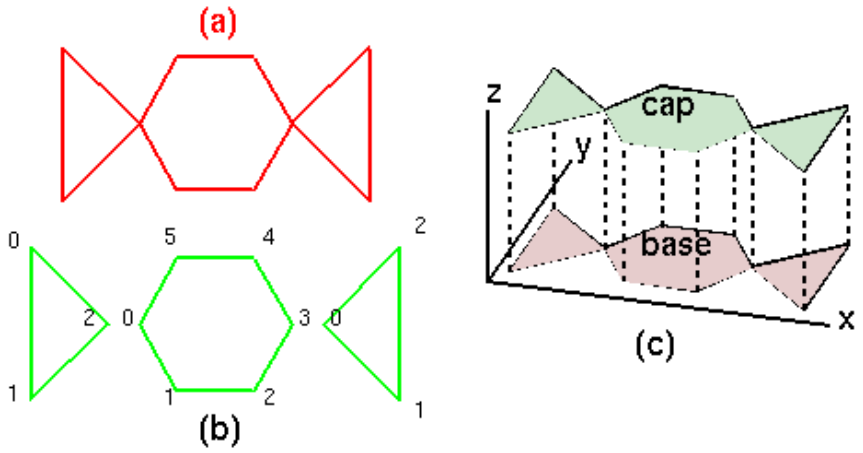


Figure 14-1 Cross-section and Walls of a Prism

Figure 14-2 below shows the prism created by the above code segment using the cross-section of Figure 14-1. Different colors have been used to render the faces of the prism. The complete code can be found in the source-code package of this book.

We should note that OpenGL can render only convex polygons reliably. If we draw the base of Figure 14-1 as one single polygon, OpenGL will fail to draw it properly as it is not a convex polygon. In this situation, we have to decompose the polygon into a set of convex polygons like what we did in our example, and extrude each one independently. Also, the windings of the base and the cap should be different. If we draw the vertices of both the base and the cap in the same order, the front face (outward face) is defined in different winding directions for the base and the cap. The front face winding direction can be changed by the command **glFrontFace**(). The following sample code will ensure that every face drawn is an outward face:

```
glFrontFace( GL_CW );      //clockwise is front facing
draw( base );      //draw base
glFrontFace( GL_CCW );   //anitclockwise is front facing
draw( cap );       //draw cap
drawWalls();
```

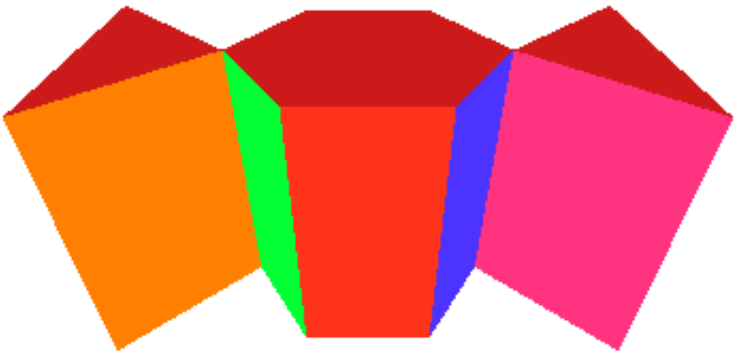


Figure 14-2 Prism by Extrusion

In the above example, the cap is just a translation of the base along the z-axis; the cap and the base have the same shape and size. We can easily generalize this by making the cap an affine transformation of the base. Suppose $b_i = (x_i, y_i, z_i)$ is the i-th vertex of the base. Then the corresponding vertex c_i of the cap is obtained by

$$c_i = Mb_i \tag{14.1}$$

where M is an affine transformation matrix and homegeneous coordinates are used.

For example, if we want the cap to have a smaller size we can choose M to have the following form:

$$M = \begin{pmatrix} 0.8 & 0 & 0 & 0 \\ 0 & 0.8 & 0 & 0 \\ 0 & 0 & 1 & H \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{14.2}$$

The transformation given by (14.2) is a scaling in the x and y directions with scaling factor 0.8 plus a translation of H in the z-direction. Figure 14-3(a) below shows an example of this case.

If we want to 'twist' the prism, we can first rotate the base through an angle θ about the

z-axis before translation; the transformation matrix will have the form,

$$M = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & H \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{14.3}$$

Figure 14-3(b) shows an example of such a prism. The transformation involves matrix multiplications, which will be also used in subsequent sections of this chapter and later chapters. We therefore define a few simple matrix classes to handle the special transformations. We discuss the matrix classes in the next section.

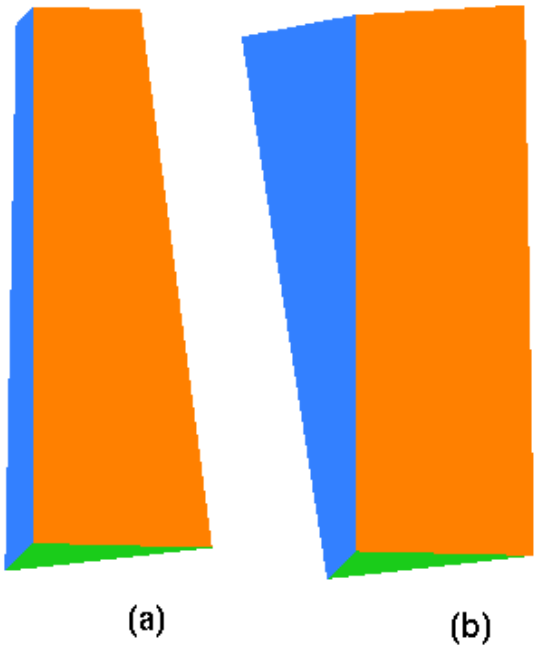


Figure 14-3 Extruded Prisms: Caps by Affine Transformation of Base

14.2 Matrix Classes

In this section, we define a few simple matrix classes that will help implement extrusion programs more effectively and systematically. For simplicity, we only consider 1×4, 4×1, and 4×4 matrices. We first define the base class *Matrix4* that has the common properties of 1×4 and 4×1 matrices. We then define the subclasses *Matrix14* for 1×4 matrix operations and *Matrix41* for 4×1 matrix operations. Their inheritance relations are shown in Figure 14-4 below. The class *Matrix44* is another independent class that can be constructed from *Matrix41* or *Matrix14*.

Both *Matrix14* and *Matrix41* have four components x, y, z, and w, which are inherited from their parent, *Matrix4*. In general, if $w = 0$, it represents a 3D vector. If $w \neq 0$, it represents a point; the position of the corresponding 3D point is specified by $(x/w, y/w, z/w)$. Conversely, if the class is constructed from a 3D vector (*Vector3*), its fourth component w

is set to 0; if it is constructed from a 3D point (*Point3*), w is set to 1. In the implementation, we use the array $m[4]$ to represent x, y, z, and w with $m[0]$ representing x, $m[1]$ representing y, and so on. The following header shows the constituents of the *Matrix4* class:

```
//Matrix4.h : Base class for 1 X 4 and 4 x 1 matrix operations
class Matrix4
{
  public:
    double m[4];    //row or column elements (x, y, z, w)
    //constructors
    Matrix4();
    Matrix4 ( double x, double y, double z, double w );
    Matrix4 ( const double a[3], double w );
    Matrix4 ( const double a[4] );
    Matrix4 ( const Point3 &p );
    Matrix4 ( const Vector3 &v );
    //member functions
    void set ( double x, double y, double z, double w );
    void set ( const double a[3], double w );
    void set ( const double a[4] );
    void get3 ( double a[3] );
    void get4 ( double a[4] );
    XYZ getXYZ();
    bool isVector3();  //does it represent a 3D vector or a point?
    Point3 getPoint3();
    Matrix4 operator + ( const Matrix4 &mat );
    Matrix4 operator - ( const Matrix4 &mat );
    Matrix4 operator * ( const double a );
};
```

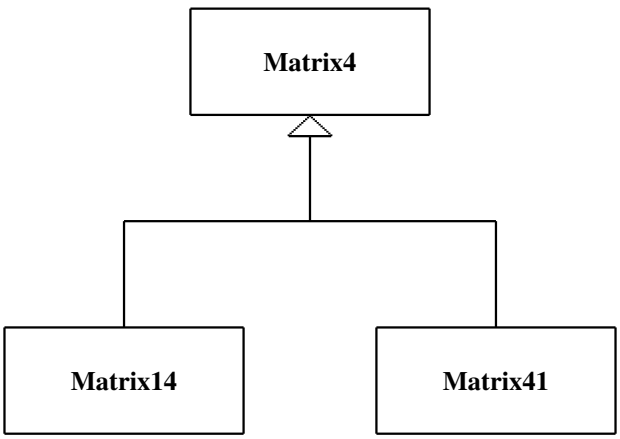


Figure 14-4. Class *Matrix4* and subclasses *Matrix14* and *Matrix41*

The following code shows the subclass *Matrix14* declaration:

```
//Matrix14.h : Class defines 1 X 4 matrix operations
#include "Matrix4.h"
```

```
class Matrix41;  //forward declaration
class Matrix44;  //forward declaration

class Matrix14: public Matrix4
{
  public:
    Matrix14();
    Matrix14 ( double x, double y, double z, double w );
    Matrix14 ( const Point3 &p );
    Matrix14 ( const Vector3 &v );
    Matrix41 transpose();
    double operator * (const Matrix41 &mat);  //(1x4)(4x1)-->(1x1)
    Matrix14 operator * (const Matrix44 &mat);//(1x4)(4x4)-->(1x4)
    void print();   //print row matrix
};
```

We have similar code for the class *Matrix41*. From the header code, we see that the operator * represents matrix multiplications. The multiplication of a 1×4 matrix and a 4×1 matrix yields a 1×1 matrix (a scalar). Multiplying a 1×4 matrix to a 4×4 matrix gives a 1×4 matrix, and the product of a 4×1 matrix and a 1×4 matrix is a 4×4 matrix. In general,

$$(n \times m)(m \times r) \rightarrow (n \times r)$$

In homogeneous coordinates, points and vectors are 4×1 matrices.

The other class, *Matrix44* handles 4×4 matrix operations such as affine transformations. The following code shows the *Matrix44* class declaration:

```
//Matrix44.h : Class defines 4 X 4 matrix operations

#include "Matrix4.h"
class Matrix14;  //forward declarations
class Matrix41;

class Matrix44
{
  public:
    double m[4][4];    //4 x 4 array
    Matrix44();
    Matrix44 ( const double mat44[4][4] );
    Matrix44 ( const double *a );
    Matrix44 ( const Matrix14 &r0, const Matrix14 &r1, const Matrix14 &r2,
               const Matrix14 &r3 );
    Matrix44 ( const Matrix41 &c0, const Matrix41 &c1, const Matrix41 &c2,
               const Matrix41 &c3 );
    void set ( const double *a );
    void get ( double *a );
    bool setCol ( int nCol, const double a[] );  //set one column
    bool setCol ( int nCol, double a0, double a1, double a2, double a3 );
    Matrix44 transpose();
    Matrix44 operator + ( const Matrix44 &mat );
    Matrix44 operator - ( const Matrix44 &mat );
    Matrix44 operator * ( const double d );
    Matrix41 operator * ( const Matrix41 &mat );  //(4x4)(4x1) --> (4x1)
    Matrix44 operator * ( const Matrix44 &mat );  //(4x4)(4x4) --> (4x4)
    double determinant();
    Matrix44 inverse();
};
```

There are many ways we can construct a 4×4 matrix. For example, we can construct it from a 1D array of 16 elements, or from a 2D array of 4×4 elements, or from 4 *Matrix14*

objects. The member functions **transpose**(), **determinant**(), and **inverse**() calculate the transpose, determinant and inverse of the 4×4 matrix respectively. Listing 14-1 below presents some sample implementations of some of the functions or operators of the matrix classes.

Program Listing 14-1 Sample Implementations of Matrix Classes

```
//------------------- Matrix4  --------------------
Matrix4::Matrix4 ( const Point3 &p )
{
  m[0] = p.x;  m[1] = p.y;  m[2] = p.z;
  m[3] = 1;
}

Matrix4::Matrix4 ( const Vector3 &v )
{
  m[0] = v.x;  m[1] = v.y;  m[2] = v.z;
  m[3] = 0;
}

void Matrix4::get4 ( double a[4] )
{
  for ( int i = 0; i < 4; i++ )
    a[i] = m[i];
}

XYZ Matrix4::getXYZ()
{
   XYZ xyz (m[0], m[1], m[2]);

   return xyz;
}

//------------------- Matrix14 --------------------
Matrix41 Matrix14::transpose()
{
  Matrix41 mat41;

  for ( int i = 0; i < 4; i++ )
    mat41.m[i] = m[i];

  return mat41;
}

//(1x4)(4x4) --> (1x4)
Matrix14 Matrix14::operator * ( const Matrix44 &mat )
{
  Matrix14 mat14;    //elements initialize to 0

  for ( int i = 0; i < 4; i++ )
    for ( int j = 0; j < 4; j++ )
      mat14.m[i] += m[j] * mat.m[j][i];

  return mat14;
}

//------------------- Matrix41 --------------------
Matrix44 Matrix41::operator * ( const Matrix14 &mat14 ) //(4x1)(1x4)-->(4x4)
{
```

```
  Matrix44 mat44;
  for ( int i = 0; i < 4; i++ )
    for ( int j = 0; j < 4; j++ )
      mat44.m[i][j] = m[i] * mat14.m[j];

  return mat44;
}

//------------------- Matrix44 --------------------
Matrix44::Matrix44 ( const Matrix41 &c0, const Matrix41 &c1,
                     const Matrix41 &c2, const Matrix41 &c3 )
{
  for ( int i = 0; i < 4; i++ )
    m[i][0] = c0.m[i];
  for ( int i = 0; i < 4; i++ )
    m[i][1] = c1.m[i];
  for ( int i = 0; i < 4; i++ )
    m[i][2] = c2.m[i];
  for ( int i = 0; i < 4; i++ )
    m[i][3] = c3.m[i];
}

void Matrix44::set ( const double *a )
{
  int k = 0;
  for ( int i = 0; i < 4; i++ )
    for ( int j = 0; j < 4; j++ )
      m[i][j] = a[k++];
}

//(4x4)(4x1) --> (4x1)
Matrix41 Matrix44::operator * ( const Matrix41 &mat )
{
  Matrix41 mat41;

  for ( int i = 0; i < 4; i++ )
    for ( int j = 0; j < 4; j++ )
      mat41.m[i] += m[i][j] * mat.m[j];

  return mat41;
}

//(4x4)(4x4) --> (4x4)
Matrix44 Matrix44::operator * ( const Matrix44 &mat )
{
  Matrix44 mat44;    //elements initialize to 0

  for ( int i = 0; i < 4; i++ )
    for ( int j = 0; j < 4; j++ )
      for ( int k = 0; k < 4; k++ )
        mat44.m[i][j] += m[i][k] * mat.m[k][j];

  return mat44;
}
```

After defining the matrix classes, we can now present the code for transforming a base to a cap through an affine transformation given by (14.2) or (14.3). The following code segment shows how we obtain the cap from a base by a rotation-translation transformation

defined by (14.3).

```
const double H = 3;    //height
double theta = 3.1415926 / 6;

//define transformation matrix elements
double a[4][4] = { { cos (theta), -sin (theta), 0 , 0 },
                   { -sin (theta), cos (theta), 0, 0 },
                   { 0, 0, 1, H },
                   { 0, 0, 0, 1 } };

Matrix44 Rz ( a );   //Construct a Matrix44 object from a[4][4]

//base and cap have 3 vertices
double base[3][3] = {
   {0, 1.0, 0}, {0, 0, 0}, {1, 0.0, 0}
};
double cap[3][3];

Matrix41 Base[3];  //3 vertices, a Matrix41 object for each vertex
for ( int i = 0; i < 3; i++ ){
  Base[i].set ( base[i], 1 );    //set x, y, z, w value
  Matrix41 temp = Rz * Base[i];  //multiply matrix by a vertex
  temp.get3 ( cap[i] );          //get coordinates of transformed
                                 //  vertice; save them in cap[i]
}
```

14.3 Tubes and Snakes

14.3.1 Segmented Tubes

We have discussed that we can form a simple tube by sweeping a cross-section in space and we obtain a special shape of the tube by performing an affine transformation of the cross section. If we create various tube segments each with a particular transformation and join the segments end to end, we can obtain a long tube with a specific shape. Figure 14-5 shows a tube created in this way; we extrude a triangular base B_0 three times, in different directions with different scaling. The first cap C_0, which becomes the next base B_1 is given by $C_0 = B_1 = M_0 B_0$, where M_0 is the first transformation matrix employed to make the first segment S_0; in the notation, the matrix operates on each vertex of the base. For example, the j-th vertex of the cap is $C_{0j} = M_0 B_{0j}$. We can generalize this to the creation of segment i from base B_i with matrix M_i as

$$B_{i+1} = C_i = M_i B_0 \tag{14.4}$$

Again, it is understood that matrix M_i operates on each vertex of B_0. The various intermediate transformed bases are referred to as **waists** of the tube. In the example, the waists are given by $M_0 B_0, M_1 B_0, M_2 B_0$. Program Listing 14-2 below shows an implementation that generates that tube of Figure 14-5.

Program Listing 14-2 Implementation of Building a Segmented Tube

```
const int N0 = 3;      //number of vertices in base
double base[N0][3] = {
```

```
   {0, 0.5, 0}, {0, 0, 0}, {0.5, 0.0, 0}
};
double cap[3][N0][3];
//Three Transformation matrices
double a0[4][4] = { {0.5, 0.0, 0.0, 1.0},
                    {0.0, 0.5, 0.0, -0.2},
                    { 0, 0, 0.5, 0.5 },
                    { 0, 0, 0, 1 } };
double a1[4][4] = { {0.8, 0.0, 0.0, 2.0},
                    {0.0, 0.8, 0.0, 0.2},
                    { 0, 0, 0.8, 0.7 },
                    { 0, 0, 0, 1 } };
double a2[4][4] = { {1.5, 0.0, 0.0, 3.0},
                    {0.0, 1.5, 0.0, 1.0},
                    { 0, 0, 1.5, 1.0 },
                    { 0, 0, 0, 1 } };
Matrix44 M[3];
void init(void)
{
   //setup transformation matrices
   M[0].set ( a0 );  M[1].set ( a1 );  M[2].set ( a2 );
   //calculate caps
   for ( int k = 0; k < 3; k++ ) {
     Matrix41 Base[N0];
     for ( int i = 0; i < N0; i++ ){
       Base[i].set ( base[i], 1 );
       Matrix41 temp = M[k] * Base[i];
       temp.get3 ( cap[k][i] );   //put values in cap array
     }
  }
}
void drawWalls ( double base[3][3], double cap[3][3] )
{
  int j, k, n = 0;
  for ( j = 0; j < 3; j++ ) {
    setColor ( n++ );           //use different colors
    glBegin(GL_POLYGON);
      k = j;
      glVertex3dv ( cap[k] );
      glVertex3dv ( base[k] );
      int next = (j + 1) % 3;
      k = next;
      glVertex3dv ( base[k] );
      glVertex3dv ( cap[k] );
    glEnd();
  }
}
void display(void)
{
  draw( base );     //draw base
  for ( int i = 0; i < 3; i++ ) {
    draw( cap[i] );      //draw cap
    //draw walls formed between a base and a cap
    if ( i == 0 )
      drawWalls( base, cap[0] );
    else
      drawWalls( cap[i-1], cap[i] );
  }
}
```

We can also make use of the transformation matrices to grow and shrink the cross-section of the tube to create a snake-shaped tube like the one shown in Figure 14-6.

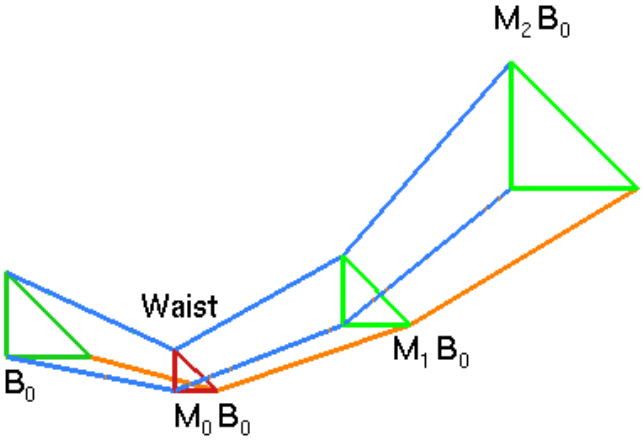


Figure 14-5 A Tube Composed of Different Segments

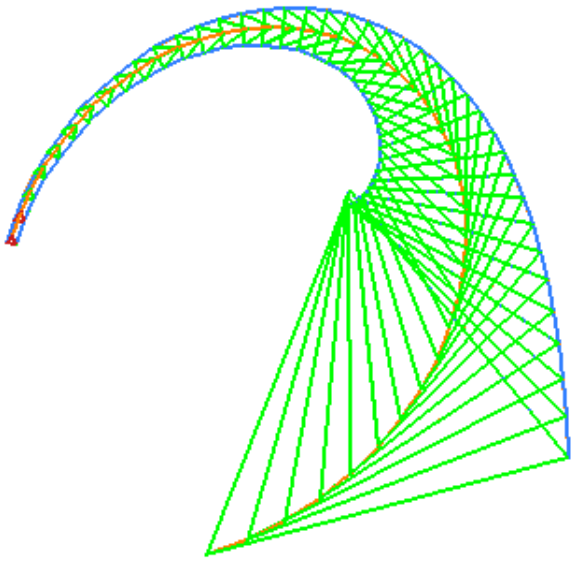


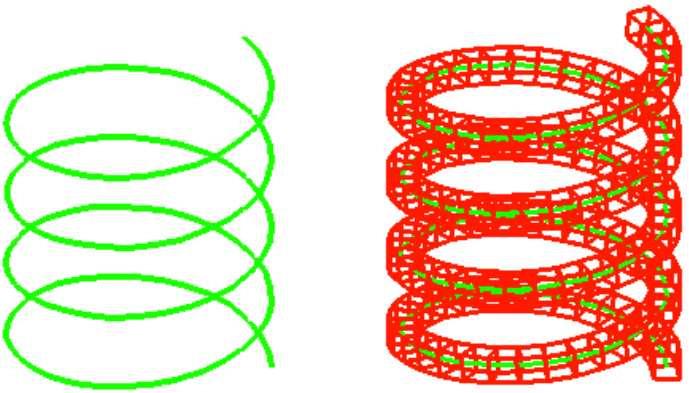
Figure 14-6 A Snake by Growing or Shrinking Caps

14.3.2 Frenet Frame

We can naturally construct a tube by wrapping a 3D shape around a curve C, which can be considered as the spine of the extrusion. In general we represent the curve parametrically by $C(u) = (x(u), y(u), z(u))$. For example, a helix curve shown in Figure 14-7 (a) can be

described parametrically by by

$$\begin{aligned} x(u) &= cos(u) \\ y(u) &= sin(u) \\ z(u) &= bu \qquad \text{for some constant b} \end{aligned} \tag{14.5}$$



(a) (b)

Figure 14-7 Helix Curve and Tube

Figure 14-7 (b) shows a tube obtained by sweeping a square along the helix curve of (a).

A common technique to construct a tube along a curve is to first obtain the curve values at various parametric value u_i and then build a polygon perpendicular to the curve at the point $C(u_i)$ using a Frenet frame, which is one of the most important tools used to analyze a curve. A Frenet frame is a moving frame that provides a local coordinate system at each point of the curve that adapts to the changes of the curvature of the curve. A Frenet frame along a curve C is created in the following steps:

1. We consider a point $C(u_i)$ on the curve in homogeneous coordinates:

$$C(u_i) = \begin{pmatrix} x(u_i) \\ y(u_i) \\ z(u_i) \\ 1 \end{pmatrix} \tag{14.6}$$

2. At each parametric value u_i, we calculate a normalized vector $\mathbf{T}(u_i)$, which is the unit tangent to the curve at u_i. We can find $\mathbf{T}(u)$ from the derivative $C'(u)$ of $C(u)$:

$$C'(u) = \frac{dC(u)}{du} = \begin{pmatrix} \frac{dx}{du} \\ \frac{dy}{du} \\ \frac{dz}{du} \\ 0 \end{pmatrix}, \qquad \mathbf{T}(u) = \frac{C'(u)}{|C'(u)|} \tag{14.7}$$

3. We calculate the unit normal $\mathbf{N}(u)$ at u_i from the derivative of the unit tangent $\mathbf{T}(u)$ at u_i:

$$\mathbf{T}'(u) = \frac{d\mathbf{T}(\mathbf{u})}{du}, \qquad \mathbf{N}(u) = \frac{\mathbf{T}'(u)}{|\mathbf{T}'(u)|} \tag{14.8}$$

The two unit vectors $\mathbf{T}$ and $\mathbf{N}$ are perpendicular to each other.

4. We then find a third unit vector, binormal $\mathbf{B}$, which is perpendicular to both $\mathbf{T}$ and $\mathbf{N}$. This unit binormal vector can be found by taking the cross product of $\mathbf{T}$ and $\mathbf{N}$:

$$\mathbf{B}(u) = \mathbf{T}(u) \times \mathbf{N}(u) \tag{14.9}$$

5. The three orthonormal vectors $\mathbf{T}(u_i), \mathbf{N}(u_i)$, and $\mathbf{B}(u_i)$ constitute the Frenet frame at u_i.

6. In summary, the basis of a Frenet frame is given by

$$\begin{aligned} \mathbf{T}(u) &= \frac{C'(u)}{|C'(u)|} && \text{the unit tangent} \\ \mathbf{N}(u) &= \frac{\mathbf{T}'(u)}{|\mathbf{T}'(u)|} && \text{the unit principal normal} \\ \mathbf{B}(u) &= \mathbf{T}(u) \times \mathbf{N}(u) && \text{the unit binormal} \end{aligned} \tag{14.10}$$

If we want to reference an object in the Frenet frame at u_i, we can translate the origin $O = (0, 0, 0)$ of our world coordinate system to the spine point $C(u_i)$ and transform the basis vectors $(\mathbf{i}, \mathbf{j}, \mathbf{k})$ to the Frenet frame basis $(\mathbf{T}(u_i), \mathbf{N}(u_i), \mathbf{B}(u_i))$. The 4×4 transformation matrix M that does this is

$$M = (\mathbf{T}(u_i), \mathbf{N}(u_i), \mathbf{B}(u_i), C(u_i)) \tag{14.11}$$

This is the transformation matrix that transforms a base polygon of the tube to its position and orientation in the Frenet frame as shown in Figure 14-8 below.

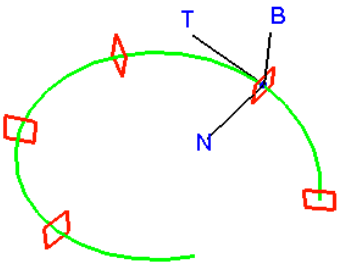


Figure 14-8 Frenet Frame

Very often people define the curvature κ and torsion τ of a curve $C(u)$ as:

$$\begin{aligned} \kappa = \kappa(u) &= \left| \frac{d\mathbf{T}(u)}{du} \right| \\ \tau = \tau(u) &= -\frac{d\mathbf{B}(u)}{du} \cdot \mathbf{N}(u) \end{aligned} \tag{14.12}$$

Note that $\frac{d\mathbf{B}}{du}$ is parallel to $\mathbf{N}$. Therefore, $|\tau| = |\frac{d\mathbf{B}}{du}|$. With these definitions, we can derive the Frenet-Serret formulas:

$$\frac{d}{du} \begin{pmatrix} \mathbf{T} \\ \mathbf{N} \\ \mathbf{B} \end{pmatrix} = \begin{pmatrix} 0 & \kappa & 0 \\ -\kappa & 0 & \tau \\ 0 & -\tau & 0 \end{pmatrix} \begin{pmatrix} \mathbf{T} \\ \mathbf{N} \\ \mathbf{B} \end{pmatrix} \tag{14.13}$$

Example 14-1:
Consider the helix curve described by Equation (14.5). Find a Frenet frame at u_i and the transformation matrix M that transforms objects to the new coordinate system.

Solution:
In this example, the spine point on the curve at u_i is:

$$C(u_i) = \begin{pmatrix} cos(u_i) \\ sin(u_i) \\ bu_i \\ 1 \end{pmatrix} \tag{14.14}$$

A tangent vector to the curve is

$$C'(u) = \frac{dC(u)}{du} = \begin{pmatrix} -sin(u) \\ cos(u) \\ b \\ 0 \end{pmatrix}$$

Therefore the unit tangent at u_i is:

$$\mathbf{T}(u_i) = \frac{C'(u_i)}{|C'(u_i)|} = \frac{1}{\sqrt{1+b^2}} \begin{pmatrix} -sin(u_i) \\ cos(u_i) \\ b \\ 0 \end{pmatrix} \tag{14.15}$$

where we have used the trigonometry identity $sin^2\theta + cos^2\theta = 1$.
The unit normal at u_i is:

$$\mathbf{N}(u_i) = \frac{T'(u_i)}{|T'(u_i)|} = \begin{pmatrix} -cos(u_i) \\ -sin(u_i) \\ 0 \\ 0 \end{pmatrix} \tag{14.16}$$

The unit binormal at u_i is:

$$\mathbf{B}(u_i) = \mathbf{T}(u_i) \times \mathbf{N}(u_i) = \frac{1}{\sqrt{1+b^2}} \begin{pmatrix} b \times sin(u_i) \\ -b \times cos(u_i) \\ 1 \\ 0 \end{pmatrix} \tag{14.17}$$

The three unit vectors $(\mathbf{T}(u_i), \mathbf{N}(u_i), \mathbf{B}(u_i))$ form an orthonormal basis, which is a Frenet frame.
Let $c = \frac{1}{\sqrt{1+b^2}}$. The transformation matrix $M = (\mathbf{T}(u_i), \mathbf{N}(u_i), \mathbf{B}(u_i), C(u_i))$ is:

$$M = \begin{pmatrix} -c \times \sin(u_i) & -\cos(u_i) & c \times b \times \sin(u_i) & \cos(u_i) \\ -c \times \cos(u_i) & -\sin(u_i) & -c \times b \times \cos(u_i) & \sin(u_i) \\ c \times b & 0 & c & b \times u_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{14.18}$$

The implementation of transforming objects to a Frenet frame of the above example is straight forward. All we need to do is to calculate the transformation matrix of (14.18) and then multiply the vertex coordinates of any object by this matrix. The following code segment presents an example implementation.

Program Listing 14-3 Example Code for Frenet Frame Transformation

```
//Matrix for transforming to Frenet frame of Helix Curve
void setM( Matrix44 &M, float u, float b )
{
  float c = 1.0 / sqrt ( 1 + b*b );
  M.setCol( 0, -c * sin(u), c * cos(u), b * c, 0);      //Tangent   T(u)
  M.setCol( 1,  -cos(u), -sin(u), 0, 0 );               //Normal    N(u)
  M.setCol( 2, c * b * sin(u), -c * b * cos(u), c, 0 ); //Binormal  B(u)
  M.setCol( 3, cos(u), sin(u), b * u, 1 );              //The curve C(u)
}

//An array is not copyable; so we define this class mainly for copying
class Cdouble3 {
  public:
     double p3[3];
};

void display(void)
{
  const float b = 0.1;       //constant of Helix curve
  Matrix44 M44;              //Transformation matrix
  const int N = 4;           //number of vertices in base

  //two STL vectors to hold base and cap
  vector<Cdouble3>vp0(N), vp1(N);
  Matrix41 p_1;              //transformed point

  //4 vertices of a quad
  //homogeneous coordinates of the four vertices of a quad
  Matrix41 points[4];        //define four points
  points[0] =  Matrix41 ( 0, -0.1, -0.1, 1 );  //x, y, z, w
  points[1] =  Matrix41 ( 0, -0.1,  0.1, 1 );  //x, y, z, w
  points[2] =  Matrix41 ( 0,  0.1,  0.1, 1 );  //x, y, z, w
  points[3] =  Matrix41 ( 0,  0.1, -0.1, 1 );  //x, y, z, w

  float p3[3];               //3-D point, (x, y, z)
  //starting
  setM ( M44, 0, b );        //u = 0
  for ( int i = 0; i < 4; ++i ) {
    p_1 = M44 * points[i];   //transform the point
    p_1.get3( vp0[i].p3 );   //put (x, y, z) in vp0[i].p3[]
  }
  glBegin( GL_QUADS );       //a side has four points
  for ( float u = 0.2; u <=  26; u += 0.2 ) {
    setM ( M44, u, b );
    for ( int i = 0; i < N; ++i ) {
      p_1 = M44 * points[i]; //transform to Frenet Frame system
      p_1.get3( vp1[i].p3 ); //put (x, y, z) in vp1[i].p3[]
    }
    //draw the N sides of tube between 'base' and 'cap'
    for ( int i = 0; i < N; ++i ) {
      int j = (i+1) % N;
      glVertex3dv( vp0[i].p3 );
```

```
      glVertex3dv( vp0[j].p3 );
      glVertex3dv( vp1[j].p3 );
      glVertex3dv( vp1[i].p3 );
    }
    copy ( vp1.begin(), vp1.end(), vp0.begin() ); //copy vp1 to vp0
  }  //for u
  glEnd();
}
```

In Listing 14-3, we have made use of the C++ Standard Template Library (STL) to simplify our implementation. In the program, we have included two STL headers:

```
#include <vector>
#include <algorithm>
```

These allow us to use the C++ *vector* class and the function **copy**(). As we can see from the Listing, the function **setM**() sets up the Frenet frame transformation matrix M at a certain value of u of a helix curve. The class *Cdouble3* only has one data member, *p3*, which is an array of 3 doubles. We define this class because we cannot copy an array directly using the C/C++ assignment operator '=' with one simple statement. However, we can perform this copy operation for class objects. The statement

```
vector<Cdouble3>vp0(N), vp1(N);
```

declares two vectors, *vp0*, and *vp1* each with capacity *N* and of data type *Cdouble3*. They save the vertices of the base and the cap of a segment respectively. When we sweep the cross section, the cap of the previous segment becomes the current base. So we copy the previous cap vector to the current base vector. This is done by the statement

```
copy ( vp1.begin(), vp1.end(), vp0.begin() ); //copy vp1 to vp0
```

The new cap is obtained from transformation. Note that the x-axis **i** maps to the unit tangent **T** of the Frenet frame and we sweep the cross section, which is a square here, along the curve. To construct the tube, we want the square to be perpendicular to the curve with its center on the curve. In other words, the square is centered around the **T** axis. Therefore, we define our cross section (the square) to lie on the y-z plane and centered around the x-axis like the following:

$$\begin{aligned} &\text{vertex } 0 : (0, \quad -0.1, \quad -0.1) \\ &\text{vertex } 1 : (0, \quad -0.1, \quad 0.1) \\ &\text{vertex } 2 : (0, \quad 0.1, \quad 0.1) \\ &\text{vertex } 3 : (0, \quad 0.1, \quad -0.1) \end{aligned}$$

In the above code, the vertices coordinates are stored in the array *points*; each *points* object is a 4×1 matrix.

The variable *p_1* defines a point (a 4×1 matrix in homogeneous coordinates). This variable is used to calculate the new position of a point transformed by the transformation matrix such as

```
for ( int i = 0; i < N; ++i ) {
   p_1 = M44 * points[i]; //transform to Frenet Frame system
   p_1.get3( vp1[i].p3 ); //put (x, y, z) in vp1[i].p3[]
}
```

where the member function **get3**(double a[]) of *p_1* extracts the x, y, z values of *p_1* and puts them in the array a[], which is a function argument.

Figure 14-7 (b) above shows an output of this program and Figure 14-9 below shows an output with lighting effect and tube segments drawn as solid polygons.

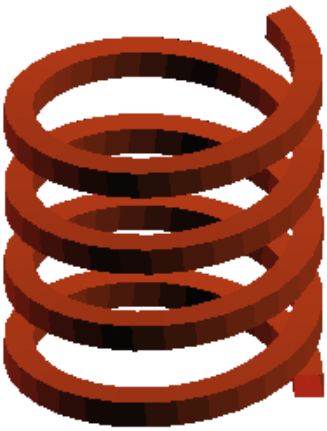


Figure 14-9 Solid Tube Created using Frenet Frame

In some cases, the curve $C(u)$ may have a complicated form and it is difficult to find the derivatives of it. In these situations, we can approximate the derivatives using numerical techniques like the following:

$$\mathbf{T}(u) \sim \frac{dC(u)}{du} = \frac{C(u+\epsilon) - C(u-\epsilon)}{2\epsilon} \tag{14.19a}$$

$$\mathbf{N}(u) \sim \frac{d^2C(u)}{du^2} = \frac{C(u+\epsilon) - 2C(u) + C(u-\epsilon)}{\epsilon^2} \tag{14.19b}$$

The numerical differentiations usually produce good approximations for the Frenet frame basis vectors. However, the method may sometimes give unstable solutions.

14.3.3 Toroidal Spirals and Other Examples

We may use toroidal spirals to create interesting tubes. Toroidal spiral curves can be described by the equations,

$$C(u) = \begin{pmatrix} x(u) \\ y(u) \\ z(u) \\ 1 \end{pmatrix} = \begin{pmatrix} (R + r\cos(qu))\cos(pu) \\ (R + r\cos(qu))\sin(pu) \\ r\sin(qu) \\ 1 \end{pmatrix}, \qquad 0 \le u \le 2\pi \tag{14.20}$$

where R, r, p, and q are some constants. Such a curve wraps around the surface of a torus centered at the origin (see Figure 13-2); the constants R and r are the large and small radii of the torus as shown in Figure 13-2 of Chapter 13. The parameter q determines the number of times the curve wraps around the torus, and the parameter p defines the number of times the curve winds around the origin.

In this case the tangent at u is $\mathbf{T}(u) \sim \frac{dC(u)}{du}$ with

$$\begin{aligned} \frac{dx(u)}{du} &= -p(R + r\cos(qu))\sin(pu) - rq\sin(qu)\cos(pu) \\ &= -py(u) - rq\sin(qu)\cos(pu) \end{aligned} \tag{14.21a}$$

and

$$\begin{aligned}\frac{dy(u)}{du} &= p(R + r\cos(qu))\cos(pu) - rq\sin(qu)\sin(pu)\\ &= -px(u) - rq\sin(qu)\sin(pu)\\ \frac{dz(u)}{du} &= rq\cos(qu)\end{aligned} \tag{14.21b}$$

The normal vector **N**(u) is obtained from the second derivatives of $C(u)$:

$$\begin{aligned}\frac{d^2x(u)}{du} &= -p\frac{dy}{du} + rq(p\sin(qu)\sin(pu) - q\cos(qu)\cos(pu))\\ \frac{d^2y(u)}{du} &= p\frac{dx}{du} - rq(p\sin(qu)\cos(pu) + q\cos(qu)\sin(pu))\\ \frac{d^2z(u)}{du} &= -rq^2\sin(qu)\end{aligned} \tag{14.22}$$

The binormal vector **B**(u) is obtained by $\mathbf{B} = \mathbf{T} \times \mathbf{N}$.

The following code segment shows an implementation of calculating the transformation matrix M for a toroidal spiral:

```
//toroidal-spiral
void get_C ( double C[4], double u, double R, double r,
                                          double p, double q )
{
  double t1 = q * u, t2 = p * u;

  C[0] = (  R + r * cos ( t1 )) * cos ( t2 );
  C[1] = ( R + r * cos ( t1 ) ) * sin ( t2 );
  C[2] = r * sin( t1 );
  C[3] = 1;
}
//Tangent
void get_T ( double T[4], double C[4], double  u, double r,
                                          double p, double q )
{
  double  t1, t2;
  t1 = q * u;
  t2 = p * u;
  T[0] = -p * C[1] - r * q * sin ( t1 ) * cos ( t2 );
  T[1] = p * C[0] - r * q * sin ( t1 ) * sin ( t2 );
  T[2] = r * q * cos ( t1 );
  T[3] = 0;
}

//Normal
void get_N ( double N[4],  double C[4], double T[4], double u,
                                 double r, double p, double q )
{
  float t1, t2;
  t1 = q * u;
  t2 = p * u;

  N[0] = -p*T[1] + r*q*(p * sin(t1) * sin(t2) - q * cos(t1) * cos(t2));
  N[1] = p * T[0] - r * q*(p * sin(t1) * cos(t2) + q* cos(t1)*sin(t2));
  N[2] = -q * q * r * sin ( t1 );
  N[3] = 0;
}

//Matrix for transforming to Frenet frame
void setM( Matrix44 &M, double u, double R, double r, double p, double q )
```

```
{
  double C[4], T[4],  N[4], B[4];
  get_C ( C, u, R, r, p, q );
  get_T ( T, C, u, r, p, q );
  get_N ( N, T, C, u, r, p, q );
  vector_normalize (  T );   //normalize the vector
  vector_normalize (  N );   //normalize the vector
  vector_cross ( B, T, N );  //B = T X N

  M.setCol( 0, T );     //Tangent   T
  M.setCol( 1, N );     //Normal    N
  M.setCol( 2, B );     //Binormal  B
  M.setCol( 3, C );     //The curve C
}
```

If we want a tube with circular cross section, we can approximate the base with an n-sided regular polygon as shown in Figure 14-23 below.

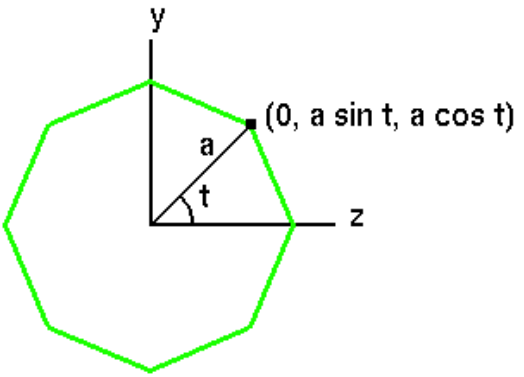
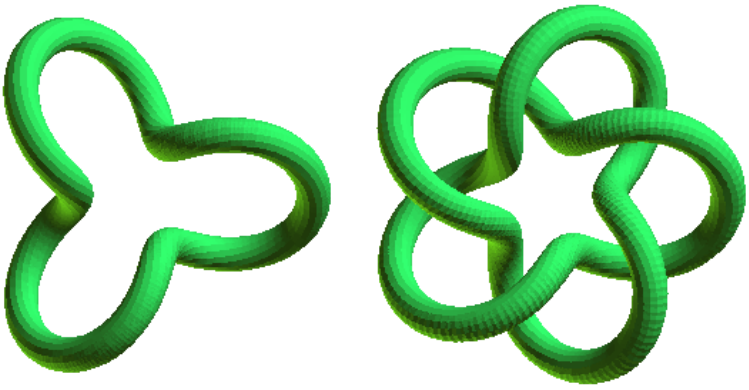


Figure 14-23 Approximating a Circle with an N-sided Polygon

The following code segment shows an implementation of approximating a circular base with a 16-sided polygon:

```
const int N = 16;             //number of vertices in base
Matrix41 points[N];           //define N points
double a = 0.15;
double x, y, z;
double dtheta = 2 * 3.1415926/ N;
double theta = 0;
for ( int i = 0; i < N; i++ ) {
  z = a *   cos ( theta );
  y = a *  sin ( theta );
  x = 0;
  points[i] = Matrix41( x, y, z, 1 );
  theta += dtheta;
}
```

Figure 14-24 below shows a couple of outputs of the implementations.



(a) (b)

Figure 14-24 Toroidal Spiral Tubes (a) p = 1.0, q = 3.0 (b) p = 2.0, q = 5.0

There are a lot of variations of using Frenet frames to construct tubes. We can multiply the Frenet frame transformation matrix by another matrix to obtain tubes with special shapes. For example, we can construct a seashell by wrapping a growing radius about a helix. This can be achieved by multiplying the matrix M of Equation (14.18) by a scaling matrix, where the scaling factors may depend on the parameter u:

$$M' = M \times \begin{pmatrix} g_1(u) & 0 & 0 & 0 \\ 0 & g_2(u) & 0 & 0 \\ 0 & 0 & g_3(u) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{14.23}$$

Figure 14-25 below shows an example of a seashell, where we have set $g_1(u) = 1, g_2(u) = g_3(u) = u/10$.

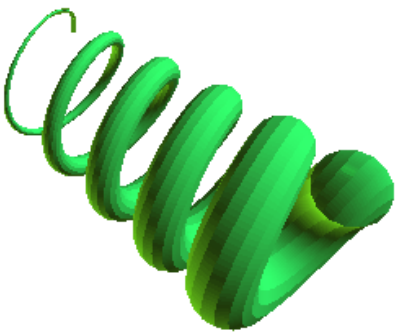
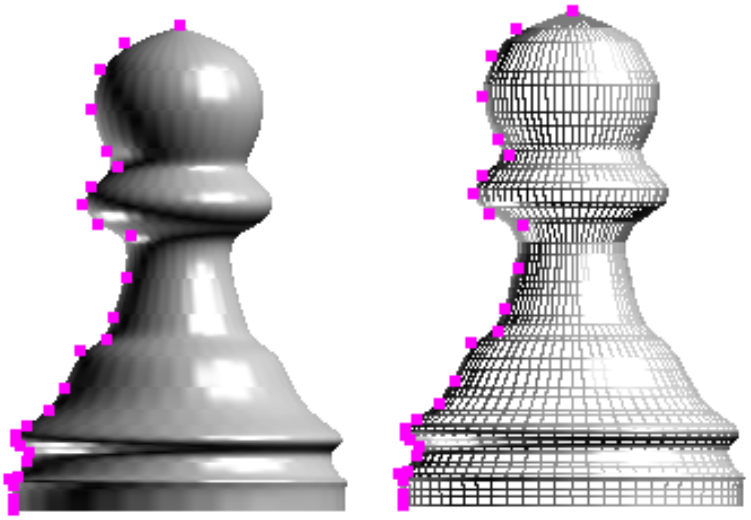


Figure 14-25 Seashell Produced by Matrix of (14.23)

14.4 Surface of Revolution

14.4.1 Revolving a Profile

Another simple and popular way to create the surface of a 3D object is to rotate a two-dimensional curve around an axis. A surface thus obtained is referred to as a **surface of revolution**, and the 2D curve is also called a **profile**. The surface always has azimuthal symmetry. For example, the walls of a cone, a conical, a cylinder, a hyperboloid, a lemon, a sphere and a spheroid can all be created by revolving a line or a curve around a vertical axis. Figure 14-26(a) below shows the surface of a pawn obtained by revolving a profile around the vertical axis; the profile is a spline curve defined by the control points shown in the figure. Figure 14-26(b) shows the corresponding wireframe of the surface.



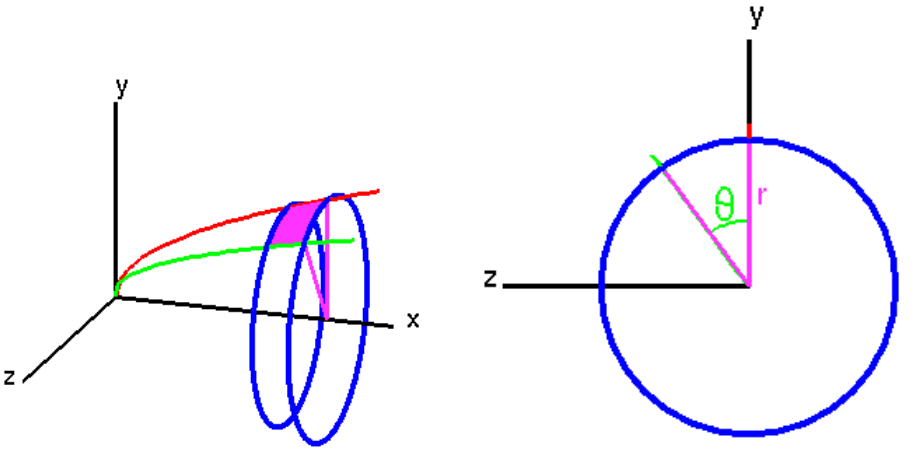
(a) (b)

Figure 14-26 Pawn Created by Revolving a Spline Curve

14.4.2 Area of a Surface of Revolution

In many situations, we are interested to find the area of a surface of revolution. Understanding how we calculate the area would also give us insight on how to use OpenGL to construct the surface.

Suppose we rotate the curve $y = f(x) > 0$ from $x = a$ to $x = b$ and we assume that the second derivatives of $f(x)$ exist for $a \leq x \leq b$. Figure 14-27 (a) shows an area element of the surface of revolution thus obtained, and Figure 14-27 (b) shows the cross-section of revolution when viewed along the negative x-axis. Note that the value of y considered here is evaluated at a point on the surface at $z = 0$. For points not at $z = 0$, the y value is given by $rcos\theta$ and $z = rsin\theta$ as shown in Figure 14-27 (b), where r is the y value at $z = 0$.



(a) (b)

Figure 14-27 Area Element and Cross Section of Surface of Revolution

Suppose Δs is the segment length of the area element along the curve. Then the area ΔA_x of the element (shaded region of Figure 14-27(a)) is given by

$$\Delta A_x = 2\pi y \Delta s$$

but

$$(\Delta s) = \sqrt{(\Delta x)^2 + (\Delta y)^2} = \Delta x \sqrt{1 + \frac{(\Delta y)^2}{(\Delta x)^2}}$$

When $\Delta x \to 0$, we have $\Delta A_x \to dA_x$, and

$$ds = dx \sqrt{1 + \left(\frac{dy}{dx}\right)^2}$$

Therefore,

$$dA_x = 2\pi y \sqrt{1 + \left(\frac{dy}{dx}\right)^2} dx$$

So the surface area is

$$A_x = 2\pi \int_a^b y \sqrt{1 + \left(\frac{dy}{dx}\right)^2} dx \tag{14.24}$$

Similarly, if we rotate the curve $x = g(y) > 0$ from $y = c$ to $y = d$ about the y-axis, the area of the surface of revolution is given by

$$A_y = 2\pi \int_c^d x \sqrt{1 + \left(\frac{dx}{dy}\right)^2} dy \tag{14.25}$$

If we instead specify the curve parametrically by $(x(u), y(u))$, and we rotate the curve about the x-axis for $u \in [a, b]$ with $x(u) > 0$ in the interval, then the surface area obtained is given by

$$A_x = 2\pi \int_a^b y(u) \sqrt{\left(\frac{dx}{du}\right)^2 + \left(\frac{dy}{du}\right)^2} du \tag{14.26}$$

The corresponding equation for y-axis rotation is

$$A_y = 2\pi \int_c^d x(u)\sqrt{\left(\frac{dx}{du}\right)^2 + \left(\frac{dy}{du}\right)^2}\,du \tag{14.27}$$

Example 14-2:
Suppose $y = f(x) = \sqrt{x}$, what is A_x for $1 \le x \le 4$?

Solution:
Here, $y = \sqrt{x}$. So

$$\left(\frac{dy}{dx}\right)^2 = \left(\frac{1}{2\sqrt{x}}\right)^2 = \frac{1}{4x}$$

Substituting this into Equation (14.24), we have

$$\begin{aligned} A_x &= 2\pi \textstyle\int_1^4 \sqrt{x}\sqrt{1+\frac{1}{4x}}dx \\ &= \pi \textstyle\int_1^4 \sqrt{4x+1}dx \\ &= \frac{\pi}{6}\ (4x+1)^{3/2}\big|_1^4 \\ &= \frac{\pi}{6}(17\sqrt{17} - 5\sqrt{5}) \\ &= 30.85 \end{aligned}$$

14.4.3 Volume of a Surface of Revolution

The volume enclosed by a surface of revolution can be found by first dividing the volume into thin discs; the total volume is the sum (integration) of the volumes of all the discs. If we rotate the curve around the x-axis like the one shown in Figure 14-27, the volume of each disc is

$$dV_x = \pi y^2 dx$$

Therefore, the total volume for $a \le x \le b$ is

$$V_x = \int_a^b dV_x = \pi \int_a^b y^2 dx \tag{14.28}$$

Similarly, the volume enclosed by the surface of revolution by rotating a curve $x = g(y) > 0$ for $c \le y \le d$ is given by

$$V_y = \pi \int_c^d x^2 dy \tag{14.29}$$

Example 14-3:
Find the enclosed volume V_x for surface shown in Figure 14-28 below, which is obtained by revolving $y = f(x) = 2 + sin(x), 0 \le x \le 2\pi$ about the x-axis.

Solution:
From Equation (14.28), we have

$$\begin{aligned} V_y &= \pi \int_0^{2\pi} (2 + sin(x))^2 dx \\ &= \pi \int_0^{2\pi} (4 + 4sin(x) + sin^2 x) dx \\ &= \pi \left[4x - 4cos(x) + \frac{x}{2} - \frac{sin(2x)}{4} \right]_0^{2\pi} \\ &= \pi[8\pi - 4 + \pi - 0 - 0 + 4 - 0 + 0] \\ &= 9\pi^2 \end{aligned}$$

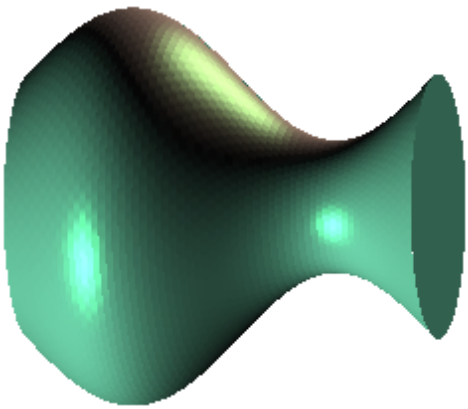


Figure 14-28 Revolving about x-axis for $f(x) = 2 + sin(x), \quad 0 \leq x \leq 2\pi$

14.4.4 Computing a Surface of Revolution

The procedure of computing a surface of revolution is like finding its area by integration. We decompose the surface into thin disks and compute an area element, which is a quad on the disk like that shown in Figure 14-27. The following steps show the details of constructing a surface of revolution:

1. Define a curve $y = f(x)$ in the xy plane and determine a set of discrete points, say $(x_i, y_i, 0)$ for $i = 0$ to $N - 1$, on the curve.
2. For each pair of two adjacent points, rotate them about the x-axis by a small angle $\Delta\theta$.
3. The rotation generates a polygon (quad) consisting of four points, the original two points and the points after they have been rotated.
4. Calculate the normal to the quad by finding the cross product of two vectors formed by the differences among three points of the polygon.
5. Draw the quad.
6. Rotate the two new points about the x-axis by another $\Delta\theta$ to create two newer points.
7. Form the quad between the new points and the newer points; calculate its normal and draw it.
8. Continue the process until a rotation of 360^o has been made.

9. Move to the next two points in the 2D curve (corresponding to a new disc) and repeat the steps above until all of the points have been used.

Program Listing 14-4 below shows the implementation of these steps, which are used to generate the vase image shown in Figure 14-28 above.

Program Listing 14-4 Computing Surface of Revolution

```
//function defining profile of revolution
double f1 ( double x )
{
  double y = 2 + sin ( x );

  return y;
}

void vase(int nx, int ntheta, float startx, float endx )
{
   const float dx = (endx - startx)/nx; //x step size
   const float dtheta = 2*PI / ntheta;  //angular step size
   float theta = PI/2.0;                //from pi/2 to3pi/2

   int i, j;
   float x, y, z, r;                    //current coordinates
   float x1, y1, z1, r1;                //next coordinates
   float t, v[3];
   float va[3], vb[3], vc[3], normal[3];
   int nturn = 0;
   x = startx;
   r = f1 ( x );
   bool first_point = true;
   for ( int si = 1; si <= nx; si++ ) {
      theta = 0;
      int start=0, nn=60, end=nn;
      x1 = x + dx;
      r1 = f1 ( x1 );
      //draw the surface composed of quadrilaterals by sweeping theta
      glBegin( GL_QUAD_STRIP );
      for ( j = 0; j <= ntheta; ++j ) {
        theta += dtheta;
        double cosa = cos( theta );
        double sina = sin ( theta );
        y = r * cosa;  y1 = r1 * cosa;  //current and next y
        z = r * sina;    z1 = r1 * sina;        //current and next z
        if ( nturn == 0 ) {
          va[0] = x;   va[1] = y;     va[2] = z;
          vb[0] = x1;  vb[1] = y1;    vb[2] = z1;
          nturn++;
        } else {
          nturn = 0;
          vc[0] = x;    vc[1] = y;    vc[2] = z;
          //calculates normal of surface
          plane_normal ( normal, va, vb, vc );
          glNormal3f ( normal[0], normal[1], normal[2] );
        }
        //edge from point at x to point at next x
        glVertex3f (x, y, z);
        glVertex3f (x1, y1, z1);
        //forms quad with next pair of points with incremented theta value
```

```
      }
      glEnd();
      x = x1;
      r = r1;
   } //for k
}
```

We can call the function **vase**() directly to render the graphics object. For example, the function

```
vase(64, 128, 0, 2 * 3.1415926 ); //x: 0 to 2pi
```

generates the vase image of Figure 14-28. Alternatively, one can make use of the OpenGL command **glCallList**() to make the rendering process more convenient and sometimes more efficient. This command takes the integer handle of a display list as the input parameter and executes the display list, which is a group of OpenGL commands that have been saved in memory for later execution. When we invoke a display list, the OpenGL functions in the list are executed in the order in which they were issued. In other words, the commands saved in the list are executed as if they were called without using a display list. In an OpenGL program, we can mix the rendering in the usual immediate mode and the display list mode. In our example here, we create the display list as follows:

```
GLuint theVase;
theVase = glGenLists (1);
glNewList(theVase, GL_COMPILE);
vase(64, 128, 0, 2 * 3.1415926 );   //x: 0 -> 2pi
glEndList();
```

The variable *theVase* in the above code is an integer handle, which is used to refer to the display list. To execute the commands saved in the list, we simply issue the command in the program:

```
glCallList ( theVase );
```

14.5 Anaglyphs by Extrusion and Revolution

14.5.1 Stereo Images with Black Background

We have discussed the general process of creating anaglyphs in Chapter 10. The main idea of the method as shown in Program Listing 10-2, is to 'render' the image with the camera at the left-eye position with a red-filter and then 'render' the image again with the camera at the right-eye position with a blue-filter; the accumulation buffer is used to superimpose the left and right images and render the composite image. The process of creating an image and the process of rendering it as a red-blue composite image is independent of each other. Therefore, the code for generating anaglyphs with images created by extrusion or surface of revolution is the same as that presented in Program Listing 10-2. All we need to do is to call the functions that create the extrusion or revolution images in the function **drawScene**(). Of course, we can always make changes in the **init**() function to have different graphical environments. For example, we can include lighting in the scene by declaring white light sources and enabling lighting in OpenGL. Figure 14-29 shows an

anaglyph version of the seashell of Figure 14-25, which is created using the technique of time-variant extrusion. Figure 14-30 shows an anaglyph of three pawns, each of which can be created independently like the one shown in Figure 14-26, which is constructed using the technique of surface-of-revolution with use of a cubic spline.

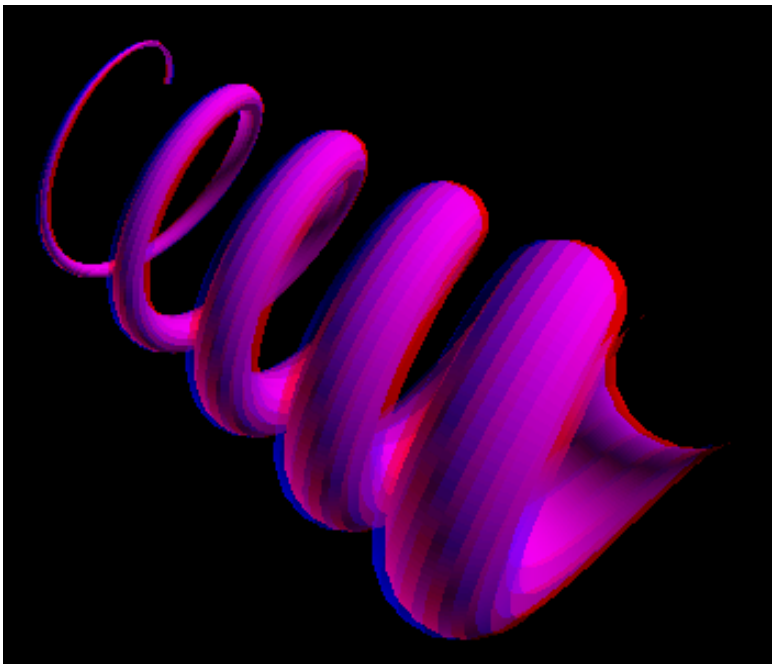


Figure 14-29 Stereoscopic Seashell

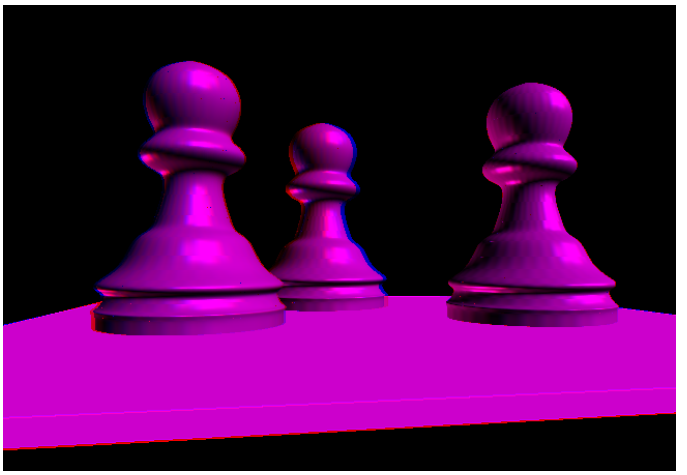


Figure 14-30 Stereoscopic Pawns

14.5.2 Stencil Buffer

We have discussed in Chapter 10 that an anaglyph should have a black background. Any background that is not totally black will somewhat compromise the 3D effect. If in some

cases, we do need a background, we can create the composite image with the help of the stencil buffer, which has been discussed briefly in Chapter 6. The main purpose of using the stencil buffer is to create a mask so that certain regions of an image can be masked off and not rendered. That is, we can use the stencil buffer to restrict drawing to a certain portion of the screen.

The stencil buffer can be turned on or off. If it is on, a pixel value is rendered only if it passes the stencil test. The stencil test at each rendering pixel position is performed in the following way:

1. It takes place only if the there is a stencil buffer and stencil test has been enabled.
2. It compares a reference value with the stencil buffer value at the pixel position.
3. The value of the stencil buffer at the pixel position may be modified, depending on the result of the test.

OpenGl provides the functions **glStencilFunc**() and **glStencilOp**() to specify the details of a stencil test. The following are their prototypes:

void **glStencilFunc** (GLenum *func*, GLint *ref*, GLuint *mask*);
This function sets the comparison function (*func*), the reference value (*ref*), and a mask (*mask*) which are used in the stencil test. The reference value is compared with the value in the stencil buffer at each rendering pixel position in a way specified by the comparison function. However, only bits with corresponding bits of *mask* having values of 1 are compared. The following table lists the possible values of the comparison function *func*, where the symbol & denotes the bitwise-and operation:

func	comparison between (*ref* & *mask*) and stencil value
GL_NEVER	always fails
GL_ALWAYS	always passes
GL_LESS	passes if *ref* $<$ stencil value
GL_LEQUAL	passes if *ref* $\leq$ stencil value
GL_EQUAL	passes if *ref* $=$ stencil value
GL_GEQUAL	passes if *ref* $\geq$ stencil value
GL_GREATER	passes if *ref* $>$ stencil value
GL_NOTEQUAL	passes if *ref* $\neq$ stencil value

void **glStencilOp** (GLenum *fail*, GLenum *zfail*, GLenum *zpass*);
This function specifies how the value in the stencil buffer at each rendering pixel is modified when the pixel passes or fails a stencil test:

fail specifies what to do to the stencil value at the pixel location when the stencil test fails,
zfail specifies what to do when the stencil test passes, but the depth test fails, and
zpass specifies what to do when both the stencil test and the depth test pass, or when the stencil test passes and either there is no depth buffer or depth testing is not enabled.

The following table lists the possible stencil operations and the resulted action of each operation on the stencil value:

Stencil operation	Resulted action on stencil value
GL_KEEP	stencil value unchanged
GL_ZERO	stencil value set to 0
GL_REPLACE	stencil value replaced by reference value
GL_INCR	stencil value incremented
GL_DECR	stencil value decremented
GL_INVERT	stencil value bitwise-inverted

Program Listing 14-5 below shows an example of using the stencil buffer to restrict drawing of certain portions of the screen. In the callback function **display**(), the statement "$glClear(GL\_STENCIL\_BUFFER\_BIT)$;" clears the stencil buffer (in this case, it fills the whole buffer with zeros). The call to the function **drawTriangle**() fills the the triangle region of the stencil buffer with ones. Before calling the first **drawSquare**() function, it executes the statement "*glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);*" to ensure that only regions outside the triangle will be rendered because the reference value is 0x1 and inside the triangle, the stencil values are also 1, which fail the stencil test of "GL_NOTEQUAL". Therefore, a red square region minus a triangle is rendered. Before calling the second **drawSquare**() function, it executes *glStencilFunc (GL_EQUAL, 0x1, 0x1);*, which has the opposite effect. Because now the comparison function is "GL_EQUAL" and the reference value is 0x1, only pixels inside the triangle will pass the stencil test. Therefore, a green triangle is rendered even though we draw a square. Figure 14-31 shows the output of this example.

Program Listing 14-5 Use of Stencil Buffer to Mask Regions

```
/*
 *  stencil-demo.cpp
 *  This program demonstrates the use of the stencil buffer to
 *  mask  pixels for rendering.
 *  Whenever the window is redrawn, a value of 1 is drawn
 *  into a triangular-region in the stencil buffer and  0 elsewhere.
 */
void init (void)
{
   glEnable ( GL_DEPTH_TEST );
   glEnable ( GL_STENCIL_TEST );
   glClearColor (1.0f, 1.0f, 1.0f, 0.0f);
   glClearStencil ( 0x0 );
}

void drawSquare( float z )
{
  glBegin ( GL_POLYGON );
      glVertex3f (-1.0, 1.0, z);
      glVertex3f (1.0, 1.0, z);
      glVertex3f (1.0, -1.0, z);
      glVertex3f (-1.0, -1.0, z);
   glEnd();
}

void drawTriangle( float z )
{
  glBegin ( GL_POLYGON );
      glVertex3f ( 0.0,  0.0, z );
```

```
      glVertex3f (-1.0, -1.0, z );
      glVertex3f ( 1.0, -1.0, z );
   glEnd();
}

//green triangle inside red square
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   glClear(GL_STENCIL_BUFFER_BIT);  //fill stencil buffer with 0s
   glStencilFunc (GL_ALWAYS, 0x1, 0x1);
   glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
   drawTriangle( -0.5 ); //stencil buffer: triangle region filled with 1s
                         //  but outside triangle, stencil values  are 0

   //fail inside triangle, pass outside; so triangle region not rendered
   glStencilFunc ( GL_NOTEQUAL, 0x1, 0x1 );
   glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP); //no change in stencil values

   glColor3f ( 1, 0, 0 );                //red
   drawSquare( -0.5 );                   //z = -0.5,
   glStencilFunc ( GL_EQUAL, 0x1, 0x1 ); //pass in triangle region,fail outside
   glStencilOp ( GL_KEEP, GL_KEEP, GL_KEEP );//no change in stencil buffer values
   glColor3f ( 0, 1, 0 );                //green
   drawSquare ( 0 );                     //z = 0, so in front of red region
   glFlush();
}
void reshape ( int w, int h )
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      gluOrtho2D(-3.0, 3.0, -3.0*(GLfloat)h/(GLfloat)w,
                 3.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(-3.0*(GLfloat)w/(GLfloat)h,
                 3.0*(GLfloat)w/(GLfloat)h, -3.0, 3.0);

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

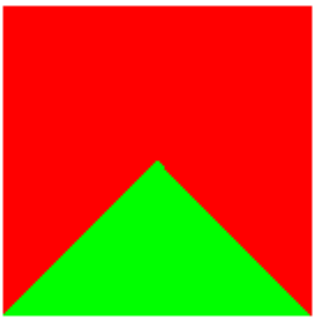


Figure 14-31 Triangular Mask with Stencil Buffer

14.5.3 Stereo Images with Background

After we have learned the usage of the stencil buffer, we can use it to add a background to a stereo composite image. The background could be a texture or simply a grey area. The background should be defined by a rectangular region that covers the entire screen minus the regions taken up by the composite image. In the two-center method of rendering an anaglyph, the top and bottom values of the rectangular region are the same for the left and right eyes. The left and right boundary values can be obtained by the left and right boundary values of the viewing frustum. Suppose we use the right eye frustum parameters to set the left and right boundary values for the rectangular background. Then the left boundary is the same as that of the frustum. The right boundary value is that of the frustum plus half of the eye separation. The background should be at the near plane. Thus its z value is equal to the difference between the focal length and the near distance that defines the frustum (i.e. $z = f - N$). In summary, using the notations of Equation (10.11) of Chapter 10, the rectangular region is defined by the parameters,

$$\begin{aligned} top &= T_r \\ bottom &= B_r \\ left &= L_r \\ right &= R_r + \frac{e}{2} \\ z &= f - N \end{aligned} \tag{14.30}$$

Program Listing 14-6 below shows a sample code segment that renders an anaglyph with a background using the stencil buffer. Figure 14-32 shows the image of the seashell of Figure 14-29 with a grey background produced by this code segment.

Program Listing 14-6 Rendering Anaglyphs With a Background

```
//rectangular region
void background ( double L, double R, double B, double T, double N )
{
  glDisable ( GL_DEPTH_TEST );
  glDisable( GL_CULL_FACE );
  glDisable ( GL_LIGHTING );
  glColor3f (0.2, 0.2, 0.2 );  //grey background
  glBegin( GL_POLYGON );
    glVertex3f ( L, B, N );
    glVertex3f ( R, B, N );
    glVertex3f ( R, T, N );
    glVertex3f ( L, T, N );
  glEnd();
  glEnable ( GL_LIGHTING );
  glEnable( GL_CULL_FACE );
  glEnable ( GL_DEPTH_TEST );
}

void display(void)
{
   glClearAccum ( 0, 0, 0, 0 );      // The default
   glClear( GL_DEPTH_BUFFER_BIT );
   glClear(GL_STENCIL_BUFFER_BIT);  //fill stencil buffer with 0s

   double theta2, near,far;
   double L, R, T, B, N, F, f, e, a, b, c, ratio;
```

```
glClearColor(0, 0, 0, 0.0);      //black background
glClear ( GL_COLOR_BUFFER_BIT );

near = camera.f / 5;
far = 1000.0;
f = camera.f;
glDrawBuffer ( GL_BACK );
glReadBuffer ( GL_BACK );
glClear(GL_ACCUM_BUFFER_BIT);

// Left eye filter (red)
glColorMask ( GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE );

// Create the projection
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
theta2 = ( 3.1415926 / 180 ) * camera.fov / 2; //theta/2 in radians
ratio  = camera.w / (double)camera.h;
a = f * ratio * tan ( theta2 );
b = a - camera.es / 2.0;
c = a + camera.es / 2.0;
N = near;
F = far;
T = N * tan ( theta2 );
B = -T;
L = -b * N / f;
R =  c * N / f;
glFrustum( L, R, B, T, N, F );
// Create the model for left eye
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
camera.p = Point3( -camera.es/2, 0, f );       //change camera viewpoint
camera.focus = Point3 ( -camera.es/2, 0, 0 );
camera.lookAt();

glDrawBuffer ( GL_BACK );
glReadBuffer ( GL_BACK );
glClearColor(0, 0, 0, 0.0);     //black background
glClear ( GL_COLOR_BUFFER_BIT );
glEnable( GL_STENCIL_TEST );    //enable stencil test

//Fill scene region of stencil buffer with 1s
glStencilFunc ( GL_ALWAYS, 0x1, 0x1);
glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);

drawScene();
glFlush();
// Write over the accumulation buffer
glAccum ( GL_LOAD, 1.0 );

glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

//now handle the right eye
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// Obtain right-eye parameters from left-eye, no change in T and B
double temp;
temp = R;
R = -L;
```

```
    L = -temp;
    glFrustum( L, R, B, T, N, F );

    // Right eye filter  (blue)
    glColorMask ( GL_FALSE, GL_FALSE, GL_TRUE, GL_TRUE );
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    camera.p = Point3( camera.es/2, 0, f );   //change camera viewpoint
    camera.focus = Point3 ( camera.es/2, 0, 0 );
    camera.lookAt();

    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    drawScene();    //scene region of stencil buffer filled with 1s
    glFlush();
    // Add  the new image
    glAccum ( GL_ACCUM, 1.0 );

    //disable stencil test to render composite scene
    glDisable ( GL_STENCIL_TEST);

    // Allow all color components
    glColorMask ( GL_TRUE, GL_TRUE, GL_TRUE,GL_TRUE );
    glAccum ( GL_RETURN, 1.0 );

    //enable stencil test to add background minus scene
    glEnable ( GL_STENCIL_TEST );
    //only pixels not occupied by scene will be rendered
    glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
    glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);   //no change in stencil values
    background ( L, R + camera.es/2 , B, T, f-N );
    glFlush();
    glutSwapBuffers();
}
```

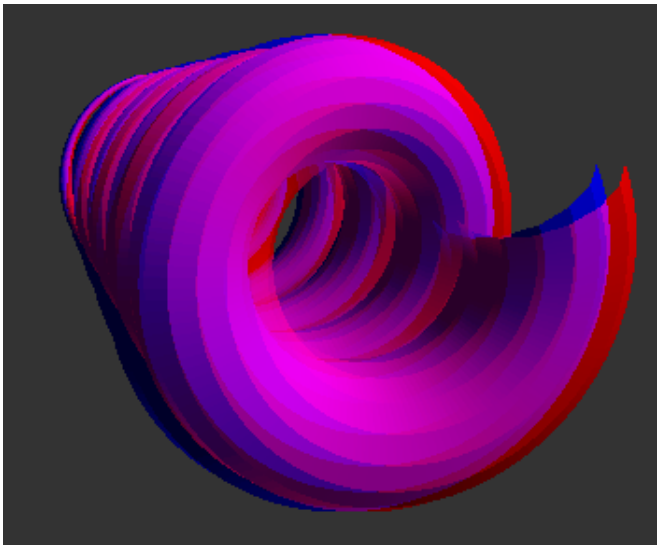


Figure 14-32 Anaglyph with Grey Background

Other books by the same author

Windows Fan, Linux Fan

by *Fore June*

Windws Fan, Linux Fan describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider (ISP) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

http://www.forejune.com/

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273