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

Fore June

Chapter 12 Normal Vectors and Polygon Mesh

A polygon mesh (or mesh) is a collection of polygons which share edges and vertices. We can construct any 3D graphical object using polygon meshes. In general, we can use different representations of polygon meshes for different applications and goals. However, triangles are the most commonly used polygons to form polygon meshes because a triangle is the simplest polygon, having three sides and three angles, and is always coplanar. Any other simple polygon can be decomposed into triangles. The process of decomposing a polygon into a set of triangles is referred to as **triangulation**.

Depending on the number of polygons used, a mesh can represent an object with various degrees of resolution, from a very coarse representation to a very fine-detailed description. A mesh can be used for graphics rendering or for object recognition. There are many ways to represent a mesh. A simple way is to use the *wire-frame* representation where the model consists of only the vertices and edges of the object model. Figure 3-5 of Chapter 3 shows the wire-frame representation of a teapot and its mirror image. In general the wire-frame model assumes that the polygons are planar, consisting of straight edges. A popular generalization of the wire-frame representation is the *face-edge-vertex* representation where we specify an object by the faces, edges and vertices. The information can be saved in different lists as discussed below. To specify the face of a surface, besides the vertices, we also need to calculate the normal to it.

A normal to a plane is a vector perpendicular to the plane. If a surface is curved, we have to specify a normal at each vertex and a normal is a vector perpendicular to the tangential plane at the vertex of the surface. Normals of a surface are important in calculating the correct amount of light that it can receive from a light source. In this chapter we discuss how to construct simple objects using polygon meshes and how to import these objects from an external application to an OpenGL program and export them from an OpenGL program to another application.

12.1 3D Vectors

A 3D vector is the difference between two points in 3D space, possessing a magnitude (or length) and a direction. It is usually represented by bold face. If P_1 and P_2 are two points, then $\mathbf{A} = P_2 - P_1$ is a vector directing from P_1 to P_2. The three standard unit vectors in Euclidean Space along the x, y, and z directions are denoted by **i**, **j**, and **k**, and represented as

$$\mathbf{i} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad \mathbf{j} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad \mathbf{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{12.1}$$

A vector $\mathbf{A}$ can be expressed as $\mathbf{A} = A_x\mathbf{i} + A_y\mathbf{j} + A_z\mathbf{k}$, or as

$$\mathbf{A} = \begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \tag{12.2}$$

The magnitude of $\mathbf{A}$ is $|\mathbf{A}| = \sqrt{A_x^2 + A_y^2 + A_z^2}$.

The dot product of two vectors **A** and **B** is a scalar and is given by

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}| cos\ \theta \tag{12.3}$$

where θ is the angle between **A** and **B** as shown in the following figure.

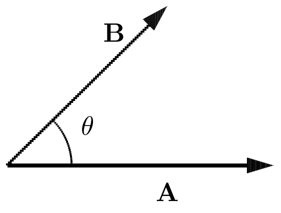

Figure 12-1 $\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}|cos\theta$

The cross product of two vectors **A** and **B** is another vector **C**, which is perpendicular to both **A** and **B** and is denoted by

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} \tag{12.4}$$

The magnitude of the cross product is given by

$$|\mathbf{C}| = |\mathbf{A}||\mathbf{B}| sin\ \theta \tag{12.5}$$

where θ is the angle between the two vectors as shown in Figure 12-2.

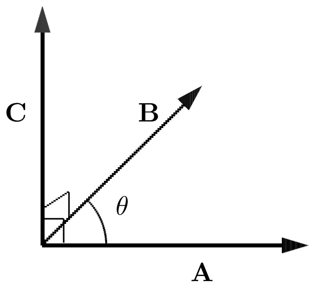

Figure 12-2 $\mathbf{C} = \mathbf{A} \times \mathbf{B}$

The direction of the cross-product **C** is determined by the right-hand rule; when we roll our fingers from **A** to **B** with our right hand, our thumb points in the direction of **C**. The cross product can be also calculated by the following formula.

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = (A_yB_z - A_zB_y)\mathbf{i} + (A_zB_x - A_xB_z)\mathbf{j} + (A_xB_y - A_yB_x)\mathbf{k} \tag{12.6}$$

The components of the cross product can be obtained in the way we calculate the determinant of a 3×3 matrix as shown in the following figure; the x, y, and z components of the product are obtained by collecting the **i**, **j**, and **k** terms respectively.

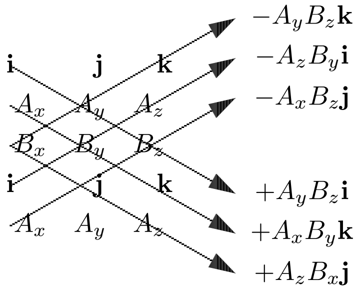

Figure 12-3 Calculating Cross Product of Two Vectors

The following are some basic properties of 3D vectors. Suppose a and b are scalars, and **A**, **B**, and **C** are 3D vectors. We let $A = |\mathbf{A}|$. Then

1. $\mathbf{A} \times \mathbf{B} = -(\mathbf{B} \times \mathbf{A})$
2. $(a\mathbf{A}) \times \mathbf{B} = a(\mathbf{A} \times \mathbf{B})$
3. $\mathbf{A} \times (\mathbf{B} + \mathbf{C}) = \mathbf{A} \times \mathbf{B} + \mathbf{A} \times \mathbf{C}$
4. $\mathbf{A} \times \mathbf{A} = \mathbf{0} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$
5. $(\mathbf{A} \times \mathbf{B}) \cdot \mathbf{C} = (\mathbf{C} \times \mathbf{A}) \cdot \mathbf{B} = (\mathbf{B} \times \mathbf{C}) \cdot \mathbf{A}$
6. $\mathbf{A} \times (\mathbf{B} \times \mathbf{A}) = \mathbf{A} \times \mathbf{B} \times \mathbf{A} = A^2\mathbf{B} - (\mathbf{A} \cdot \mathbf{B})\mathbf{A}$
7. $a(\mathbf{A} + \mathbf{B}) = a\mathbf{A} + a\mathbf{B}$
8. $(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C})$

Implementations

We have discussed the difference between points and vectors in Chapter 3. Since both a point and a vector are specified by three coordinates, it is convenient to define a class called *XYZ* that has the common properties of points and vectors; we define a *Vector3* class (for 3D vectors) and a *Point3* class (for 3D points) that will inherit the properties of *XYZ* as shown in Figure 12-4. The empty-head arrow in the figure denotes the inheritance relation; it points toward the parent class *XYZ*; the child classes are at the tail.

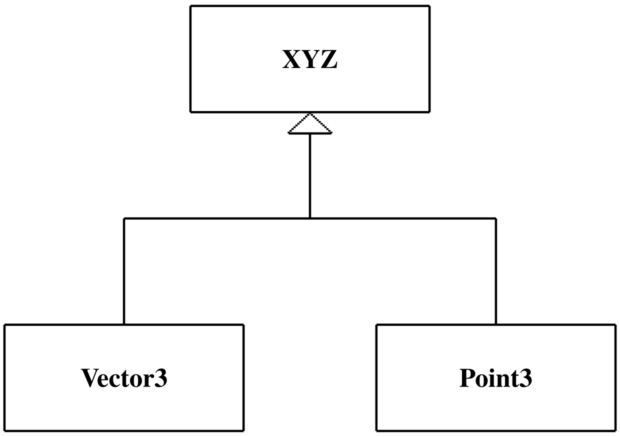

Figure 12-4. Relations between vector and point classes

We shall overload several operators for the two child classes. (In C++, *overload an operator* means *redefine a built-in operator*. An overloaded operator can be implemented as a global function or as a member function of a class.) The overloaded operators for **Vector3** and **Point3** include the following:

$-$ vector or point subtraction

$+$ vector addition or vector-point addition

$\wedge$ vector cross product operation

$*$ vector dot product operation

$*$ multiplication of a scalar and a vector

Listing 12-1 below shows the declaration of the class *XYZ*.

```
class XYZ {
  public:
    double x;
    double y;
    double z;
    XYZ ();
    XYZ ( double x0, double y0, double z0 );
    void set ( double x0, double y0, double z0 );
    XYZ getXYZ();
    void getXYZ( double a[]  );
    void print();
};
```

Listing 12-1   Class XYZ

The functions **getXYZ**() return the information of the triple (x, y, z) of an *XYZ* object. They are implemented as follows:

```
XYZ XYZ::getXYZ()
{
  XYZ a( x, y, z );

  return a;
}

void XYZ::getXYZ( double a[] )
{
  a[0] = x;  a[1] = y;  a[2] = z;
}
```

Listing 12-2 shows the declaration of the class *Point3*, which inherits all the data and function members (except constructors) of *XYZ*.

```
class Vector3;          //forward declaration

class Point3: public XYZ
{
  public:
    Point3();
    Point3( double x0, double y0, double z0 );
    Point3( const Point3 &p);
    Vector3 operator-(const Point3 &p); //point-point-> vector
    Point3  operator+(const Vector3 &v);//point+vector-> point
};
```

Listing 12-2 Class *Point3*

Implementation of the overloaded operators '-' and '+' are straightforward. For example, the subtraction operator '-' which calculates the difference between two points and returns the difference as a vector can be implemented as follows:

```
Vector3 Point3::operator - ( const Point3 &p )
{
  Vector3 v1;

  v1.x = x - p.x;
  v1.y = y - p.y;
  v1.z = z - p.z;

  return v1;
}
```

In this implementation, the argument of the operator function is a *Point3* object. The '&' before *p* in the argument means that we are using pass-by-reference to pass the object to the function; this is a more efficient way of passing an object to a function than using "pass-by-value", in which the passing object is copied and pushed onto the stack, and the called function has to pop it from the stack, which could be time-consuming when the object is large. Pass-by-reference simply passes the address of the object. To inform the user that our purpose of using "pass-by-reference" is for efficiency but not because we want to alter the content of the object, we put the keyword **const** in front of the data type. In general, "pass-by-reference" is more efficient but "pass-by-value" is more robust as it cleanly separates the calling and called functions.

Listing 12-3 below shows the declaration of the class *Vector3* and the overloaded operator '*'.

```
class Point3;            //forward declaration

class Vector3: public XYZ
{
  public:
    Vector3();
    Vector3( double x0, double y0, double z0 );
    Vector3( const Vector3 &v );
    Vector3 operator + (const Vector3 &v);//vec + vec->vector
    Vector3 operator - (const Vector3 &v);//vec - vec->vector
    Vector3 operator ^ (const Vector3 &v);//cross product
    double  operator * (const Vector3 &v);//dot product
    Point3  operator + (const Point3 &p); //vec+point->point
    double magnitude();
    void  normalize();                //make it a unit vector
};

Vector3 operator * ( double a, const Vector3 &v );
Vector3 operator * ( const Vector3 &v, double a );
```

Listing 12-3 Class *Vector3*

Listing 12-4 shows the implementations of this class and the overloaded operators.

Program Listing 12-4: *Vector3* class implementation

```
Vector3::Vector3():XYZ()
{ }

Vector3::Vector3( double x0, double y0, double z0 ): XYZ( x0, y0, z0 )
{ }

Vector3::Vector3( const Vector3 &v )
{
  x = v.x;
  y = v.y;
  z = v.z;
}

Vector3 Vector3::operator + ( const Vector3 &v )
{
  Vector3 v1;
  v1.x = x + v.x;
  v1.y = y + v.y;
  v1.z = z + v.z;

  return v1;
}

Vector3 Vector3::operator - ( const Vector3 &v )
{
  Vector3 v1;
  v1.x = x - v.x;
  v1.y = y - v.y;
  v1.z = z - v.z;

  return v1;
```

```
}

//cross product
Vector3 Vector3::operator ^ ( const Vector3 &v )
{
  Vector3 v1;
  v1.x = y * v.z - z * v.y;
  v1.y = z * v.x - x * v.z;
  v1.z = x * v.y - y * v.x;

  return v1;
}

//dot product
double Vector3::operator * ( const Vector3 &v )
{
  double d;
  d = x * v.x + y * v.y + z * v.z;

  return d;
}

//vector + point --> point
Point3 Vector3::operator + ( const Point3 &p )
{
  Point3 p1;

  p1.x = x + p.x;
  p1.y = y + p.y;
  p1.z = z + p.z;

  return p1;
}

double Vector3::magnitude()
{
  return sqrt(x * x + y * y + z * z );
}

void Vector3::normalize()
{
  double d = x*x + y*y + z*z;

  if ( d > 0 ) {
    d = sqrt ( d );
    x /= d;
    y /= d;
    z /= d;
  }
}
//------------------- external functions ---------------
//scalar times vector
Vector3 operator * ( double a, const Vector3 &v )
{
  Vector3 v1;
  v1.x = a * v.x;
  v1.y = a * v.y;
  v1.z = a * v.z;

  return v1;
```

```
}

Vector3 operator * ( const Vector3 &v, double a )
{
  return a * v;
}
```

Listing 12-5 below is a sample program that demonstrates the usage of the *Vector3* and *Point3* classes operators.

Program Listing 12-5:   Example of Using *Vector3* and Point3 classes

```
//vpdemo.cpp
int main()
{
  Vector3 v1 ( 1.0, 2.0, 4.0 );  //a vector
  XYZ a; //an XYZ object
  a = v1.getXYZ (); //get XYZ object of v1
  cout << "v1 = ";
  v1.print(); //prints  (x, y, z) values
  Vector3 v2 ( 2.0, 4.0,  6.0 );
  cout << "v2 = ";
  v2.print(); //prints (x, y, z) values of v2
  Vector3 v3 ( v2 ); //construct v2 from v2
  v3.normalize(); //normalize v3
  cout << "v3 = ";
  v3.print(); //now v3 is a unit vector
  cout << "magnitude of v3 is ";
  cout << v3.magnitude() << endl; //magnitude should be 1

  Vector3 v4 = v2 + v1; //addition of two vectors
  cout << "v2 + v1 = ";
  v4.print();
  Vector3 v5 = v2 - v1; //subtraction of two vectors
  cout << "v2 - v1 = ";
  v5.print();
  Vector3 v6 = v1 ^ v2;
  cout << "v1 X v2 = ";
  v6.print();
  double d = v1 * v2;
  cout << "Dot product of v1 and v2 is " << d << endl;
  Vector3 v7 = 3 * v1;
  cout << " 3 * v1 = ";
  v7.print();

  Point3 p1 ( 0.0, 1.0, 2.0 );
  cout << "Point p1 = ";
  p1.print();
  Point3 p2 = v1 + p1;
  cout << "v1 + p1 is the point ";
  p2.print();

  return 0;
}
```

When executed, the program of Listing 12-5 generates the following outputs.

```
v1 = (1, 2, 4)
v2 = (2, 4, 6)
v3 = (0.267261, 0.534522, 0.801784)
magnitude of v3 is 1
v2 + v1 = (3, 6, 10)
v2 - v1 = (1, 2, 2)
v1 X v2 = (-4, 2, 0)
Dot product of v1 and v2 is 34
3 * v1 = (3, 6, 12)
Point p1 = (0, 1, 2)
v1 + p1 is the point (1, 3, 6)
```

12.2 Normal to a Surface

12.2.1 Normal Vector

A **normal vector** (or normal for short) to a surface at a point is a vector pointing in a direction which is perpendicular to the surface at that point. For a plane (flat surface), one perpendicular direction is the same for every point on the surface. A normal to a surface at a point is the same as a normal to the tangent plane to that surface at that point. We know that any three points P_1, P_2, P_3 determine a unique plane. To find a normal to the plane, we build two vectors

$$\begin{aligned}\mathbf{A} &= P_2 - P_1 \\ \mathbf{B} &= P_3 - P_1\end{aligned} \tag{12.7}$$

In forming the vectors, the points P_1, P_2, P_3 should appear counter-clockwise when we look at the plane (i.e. front face) formed by the three points. The normal to the plane is given by

$$\mathbf{N} = \mathbf{A} \times \mathbf{B} \tag{12.8}$$

The corresponding unit normal is given by

$$\mathbf{n} = \frac{\mathbf{N}}{N} \tag{12.9}$$

where $N = |\mathbf{N}|$ is the magnitude of normal $\mathbf{N}$, and $|\mathbf{n}| = 1$.

12.2.2 Equation of a Plane

Suppose $P = (x, y, z)$ is an arbitrary point in the plane formed by the three points P_1, P_2, P_3 discussed above. Then $\mathbf{V} = P - P_1$ is a vector directing from P_1 to P. That is,

$$\mathbf{V} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \\ y_1 \end{pmatrix} \tag{12.10}$$

Since $\mathbf{V}$ is on the plane, it is perpendicular to the normal. Therefore, $\mathbf{V} \cdot \mathbf{N} = 0$. As a result,

$$xN_x + yN_y + zN_z - (x_1N_x + y_1N_y + z_1N_z) = \mathbf{V} \cdot \mathbf{N} = 0 \tag{12.11}$$

Equation (12.11) can be expressed in the form,

$$ax + by + cz - d = 0 \tag{12.12}$$

where $(a, b, c) = (N_x, N_y, N_z)$, and $d = (x_1N_x + y_1N_y + z_1N_z)$; it is the general equation of a plane. Conversely, if we are given the equation of a plane represented by

$$Ax + By + Cz + D = 0 \tag{12.13}$$

we immediately know its normal, which is

$$\mathbf{N} = \begin{pmatrix} A \\ B \\ C \end{pmatrix} \tag{12.14}$$

Also, suppose $O = (0, 0, 0)$ is the origin of the coordinate system of the 3D space. Then

$$\mathbf{v_1} = P_1 - O = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \tag{12.15}$$

is the vector directing from the origin to the point P_1, and $\mathbf{v_1} \cdot \mathbf{n}$ is the perpendicular distance from the origin to the plane containing the point P_1. This is equal to the value of d of equation (12.12) divided by N (i.e. $\frac{d}{N}$); this is shown in Figure 12-5 below.

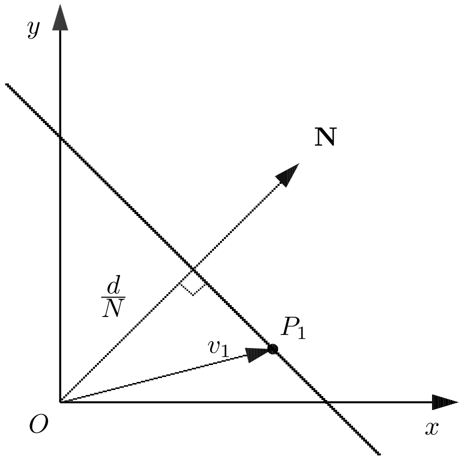

Figure 12-5 Distance to a Plane

If we want to find the distance d_0 from a point $P_0 = (x_0, y_0, z_0)$ to the plane rather than from the origin, we can first calculate v_1 by

$$\mathbf{v_1} = P_1 - P_0 = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} - \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} \tag{12.17}$$

Then the distance is gvien by

$$\begin{aligned} d_0 = & \mathbf{v_1} \cdot n \\ = & x_1 n_x + y_1 n_y + z_1 n_z - (x_0 n_x + y_0 n_y + z_0 n_z) \\ = & \frac{d - (ax_0 + by_0 + cz_0)}{|\mathbf{N}|} \\ = & -\frac{ax_0 + by_0 + cz_0 - d}{\sqrt{a^2 + b^2 + c^2}} \end{aligned} \tag{12.18}$$

Example

Find the unit normal to the plane described by the equaiton $x + 2y + z - 6 = 0$ and the distance from the point $(1, 0, 1)$ to the plane.

Solution

1. A normal to the plane is given by

$$\mathbf{N} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

2. The unit normal to the plane is

$$\mathbf{n} = \frac{\mathbf{N}}{N} = \frac{\mathbf{N}}{\sqrt{1^2 + 2^2 + 1^2}} = \frac{1}{\sqrt{6}} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

3. The distance from $(1, 0, 1)$ to the plane is given by

$$d_0 = -\frac{1 \times 1 + 2 \times 0 + 1 \times 1 - 6}{\sqrt{6}} = \frac{2\sqrt{2}}{\sqrt{3}}$$

Example

Find the unit normal to the plane containing the three points, $(1, 2, 0)$, $(1, 1, 1)$, and $(2, 0, -1)$.

Solution

Let $P_1 = (1, 2, 0)$, $P_2 = (1, 1, 1)$, and $P_3 = (2, 0, -1)$. When looking at the plane formed by the three points, P_1, P_2, P_3 appear anti-clockwise. Therefore, we form the vectors

$$\mathbf{A} = P_2 - P_1 = \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix}$$

$$\mathbf{B} = P_3 - P_1 = \begin{pmatrix} 1 \\ -2 \\ -1 \end{pmatrix}$$

A normal to the plane is given by

$$\mathbf{N} = \mathbf{A} \times \mathbf{B} = \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}$$

The unit normal is

$$\mathbf{n} = \frac{\mathbf{N}}{N} = \frac{1}{3.32}\begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.90 \\ 0.30 \\ 0.30 \end{pmatrix}$$

12.3 Polygon Mesh Modeling

As we mentioned above, a polygon mesh is a collection of polygons and we can construct any graphics object using a polygon mesh. In order for a mesh to be lit with appropriate light, we need to know the normal to each polygon of the mesh. There are many ways to model and render a polygon mesh. One simple way is to specify the mesh using a vertex list, a normal list, and a face list. Lets consider the following example, which is taken from the popular graphics textbook, “Computer Graphics Using OpenGL” by Hill and Kelly, to illustrate this method. In this example, a barn is represented by seven polygons as shown in Figure 12-6. The mesh consists of 10 vertices numbered from 0 to 9 and 7 polygons; the arrows denote the 7 normals of the 7 polygons.

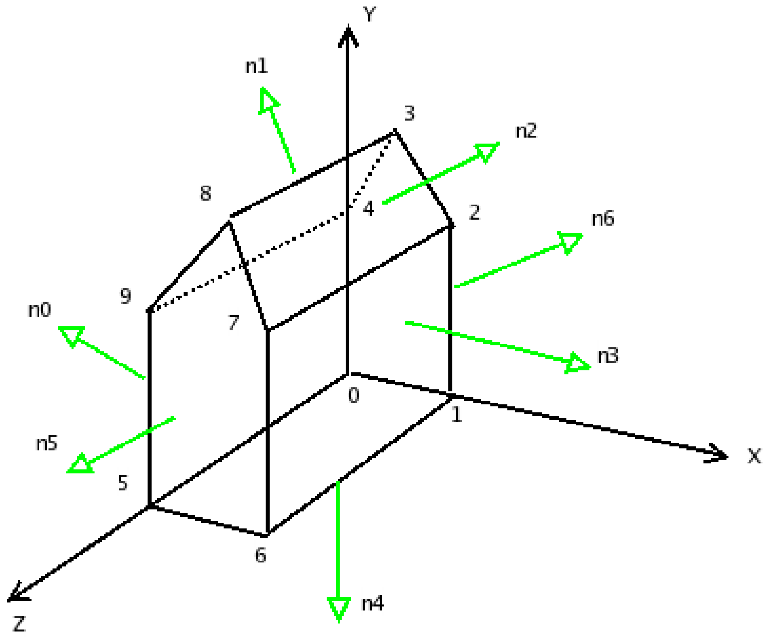

Figure 12-6 Polygon Mesh Representing a Barn

Tables 12-1 to 12-3 show the list of normals, the list of polygons and the associated vertices of each polygon, and the list of vertices of the mesh respectively.

Table 12-1 Normal List

Normal	n_x	n_y	n_z
0	-1	0	0
1	-0.707	0.707	0
2	-0.707	0.707	0
3	1	0	0
4	0	-1	0
5	0	0	1
6	0	0	-1

Table 12-2 Polygon List

Polygon	Vertices	Normals
0 (left)	0,5,9,4	0,0,0,0
1 (roof left)	3, 4, 9, 8	1,1,1,1
2 (roof right)	2, 3, 8, 7	2,2,2,2
3 (right)	1, 2, 7, 6	3,3,3,3
4 (bottom)	0, 1, 6, 5	4,4,4,4
5 (front)	5, 6, 7, 8, 9	5,5,5,5,5
6 (back)	0, 4, 3, 2, 1	6,6,6,6,6

Table 12-3 Vertex List

Vertex	Coordinates (x, y, z)
0	$(0, 0, 0)$
1	$(1, 0, 0)$
2	$(1, 1, 0)$
3	$(0.5, 1.5, 0)$
4	$(0, 1, 0)$
5	$(0, 0, 1)$
6	$(1, 0, 1)$
7	$(1, 1, 1)$
8	$(0.5, 1.5, 1)$
9	$(0, 1, 1)$

Implementation

To simplify the implementation of the mesh model, we make use of the C++ standard template library (STL) class **vector** to implement the vertex, normal, and polygon lists. Of course, the C++ STL **vector** class has a total different meaning from the **vector3** class we have presented above. The STL **vector** class is essentially a dynamic array that can work with different data types. To use this class, we have to include the following header line in our program:

```
#include <vector>
```

To specify a face (a polygon), we just need to specify the vertices of the polygon and the corresponding normals; actually, we just need to specify the indices of the coordinates of the vertices and normals as shown in Table 12-2. So we define the *Polygon* class to represent a face, which consists of a vector to hold the vertex indices and a vector to hold the normal indices:

```
class Polygon {
public:
  int n;                   //n sides
  vector <int> vertices;   //vertex indices of vertexList;
  vector <int> normals;    //indices of normals at vertices
};
```

For those who are not familiar with C++ STL syntax, the statement **vector** $<$**int**$>$ **vertices** means we declare the variable *vertices* as a **vector** of data type **int**.

Now we can define the class *Mesh* to represent the polygon mesh, which has a vertex list, a normal list and a face (polygon) list. Each of these lists is declared as a **vector**. The vertex list is a vector of *Point3*; the normal list is a vector of *Vector3*, and the face list is a vector of *Polygon* as shown Listing 12-6 below:

Program Listing 12-6: *Mesh* class

```
#include <vector>
#include <fstream>
#include <GL/glut.h>
#include "Point3.h"
#include "Vector3.h"

using namespace std;

class Polygon {
public:
  int n;                  //n sides
  vector <int> vertices; //vertex indices of vertexList
  vector <int> normals;  //indices of normals at vertices
};

class Mesh {
public:
  int nVertices;          //number of vertices
  int nNormals;           //number of normals
  int nFaces;             //number of polygons
  vector<Point3> vertexList;
  vector<Vector3> normalList;
  vector <Polygon> faceList; //each face is a polygon
  Mesh();
  bool readData( char fileName[] );
  void renderMesh();      //render the mesh
};
```

Listing 12-7 below shows the implementation of the two member functions, **readData**() and **renderMesh**() of the class *Mesh*. The function **readData**() reads the data from a file. It first reads the number of vertices, the number of normals and the number of faces (polygons) of the mesh. Secondly, it reads all the 3D coordinates of the vertices; the coordinates of each vertex are read as a *Point3* object, which is pushed onto the back of the vertex list, *vertexList*, by the STL vector function **push_back**(). (More precisely, the **push_back**() function creates a copy of the object and it inserts the copy into the list.) It then reads the coordinates of the normals, each of which is read as a *Vector3* object and is pushed onto the back of the normal list, *normalList*. At the end, the number of vertices, the indices of the vertices and normals of each face are read as a *Polygon* object and pushed onto the back of the face list, *faceList*. The data of the file should be organized in the order that the function reads them. Table 12-4 below shows the data and its organization of the data file of the Barn example of Figure 12-6 and Tables 12-1 to 12-3.

Table 12-4 Data of Barn Example

10 7 7	number of vertices, normals, faces
0 0 0 1 0 0 1 1 0 0.5 1.5 0 0 1 0	vertice coordinates (x, y, z)
0 0 1 1 0 1 1 1 1 0.5 1.5 1 0 1 1	
-1 0 0 -0.707 0.707 0 0.707 0.707 0	normal coordinates (x, y, z)
1 0 0 0 -1 0 0 0 1 0 0 -1	
4 0 5 9 4 0 0 0 0	a face with 4 vertices and normals
4 3 4 9 8 1 1 1 1	a face with 4 vertices and normals
4 2 3 8 7 2 2 2 2	a face with 4 vertices and normals
4 1 2 7 6 3 3 3 3	a face with 4 vertices and normals
4 0 1 6 5 4 4 4 4	a face with 4 vertices and normals
5 5 6 7 8 9 5 5 5 5 5 5	a face with 5 vertices and normals
5 0 4 3 2 1 6 6 6 6 6 6	a face with 5 vertices and normals

Program Listing 12-7: Implementation of *Mesh* Functions

```
Mesh::Mesh()
{
  nVertices = nNormals = nFaces = 0;
}

//Read mesh data from file
bool Mesh::readData ( char fName[] )
{
  fstream ins;                        //file input stream
  ins.open ( fName, ios::in );
  if( ins.fail() ) return false;      // error - can't open file
  if( ins.eof() )  return false;      // error - empty file
  // read in number of vertices, normals, and faces
  ins >> nVertices >> nNormals >> nFaces;
  for (int i = 0; i < nVertices; i++){ //read vertices
    Point3 p;
    ins >> p.x >> p.y >> p.z;
    vertexList.push_back ( p );  //insert into list at the tail
  }
  for (int i = 0; i < nNormals; i++){  //read normals
    Vector3 v;
    ins >> v.x >> v.y >> v.z;
    normalList.push_back ( v );  //insert into list at tail
  }
  for ( int i = 0; i < nFaces; i++ ) {
    Polygon p;
    ins >> p.n;
    for ( int j = 0; j < p.n; j++ ) {
      int vertexIndex;
      ins >> vertexIndex;        //read vertice index
      p.vertices.push_back ( vertexIndex );
    }
    for ( int j = 0; j < p.n; j++ ) {
      int normalIndex;
      ins >> normalIndex;        //read normal index
      p.normals.push_back ( normalIndex );
    }
    faceList.push_back ( p );
  }
```

```
    return true;
  }

  //render the mesh
  void Mesh::renderMesh()
  {
    //Draw each polygon of the mesh
    glEnable( GL_CULL_FACE );
    glCullFace ( GL_BACK );  //do not render back faces

    //draw one polygon at a time
    for ( int i = 0; i < nFaces; i++ ) {
      glBegin ( GL_POLYGON );
        //specifying vertices of the polygon
        for (int j = 0; j<faceList[i].n; j++){ //traverse the list
          int vi = faceList[i].vertices[j];    //vertex index
          int ni = faceList[i].normals[j];     //normal index
          glNormal3f ( normalList[ni].x, normalList[ni].y,
                                          normalList[ni].z );
          glVertex3f ( vertexList[vi].x, vertexList[vi].y,
                                           vertexList[vi].z );
        } //for j
      glEnd();
    }  //for i
  }
```

The function **renderMesh**() renders the polygon mesh. We use the function **glCullFace**() to cull the back faces of the mesh. That is, we won't display any face that is facing inside the mesh (back face). Since each polygon of the face is saved in the face list, to render the mesh, all we need to do is to traverse this list and render one polygon at a time as shown in the code above. Listing 12-8 shows a code section that makes use the *Mesh* functions to render a barn; the file name of the data is hard-coded to be *data.txt*. The rendered barn is shown in Figure 12-7.

Program Listing 12-8: Sample Code for Rendering Barn of Figure 12-6

```
void display(void)
{
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  glOrtho(-2.0, 2.0, -2.0, 2.0, 0.1, 100 );
  glMatrixMode(GL_MODELVIEW); // position and aim the camera
  glLoadIdentity();
  gluLookAt(8.0, 8.0, 8.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f( 0, 0, 0 );
  barn.renderMesh();
  glFlush();
}

int main( int argc, char *argv[] )
{
  if ( !barn.readData ( "data.txt" ) ) { //hard-coded filename
     cout << "Error opening file" << endl;
     return 1;
  }

  glutInit( &argc, argv );
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH );
  glutInitWindowSize( 500, 500 );
  glutInitWindowPosition( 100, 100 );
  glutCreateWindow("Barn");
  init();
  glutDisplayFunc(display);
  glutKeyboardFunc( keyboard );
  glClearColor( 1.0f, 1.0f, 1.0f, 0.0f ); //white background
  glViewport ( 0, 0, 500, 500 );

  glutMainLoop();

  return 0;
}
```

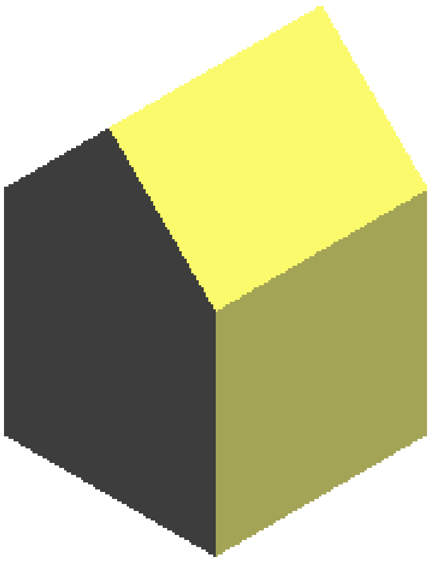

Figure 12-7 Rendered Barn of Figure 12-6

12.4 COLLADA

12.4.1 3D Graphics Formats

We mentioned that we can represent any 3D graphical object by a polygon mesh. Very often a graphical object is created by an artist using a graphics application package such as Blender or Maya; the object is represented by a mesh, which will be saved in a file along with other attributes of the object such as lighting, field of view, camera position and texture. A programmer can parse the file, transform and render the object using OpenGL. In order that the programmer can parse the object effectively, the data of object created by the artist need to be saved in an agreed upon format. There are quite a lot of popular graphics file format in the market. The following table lists a few most commonly used formats and their host organizations.

Table 12-5 Graphics Formats

format	**affiliation**
3DS	3D Studio
BLEN	BLENDER
DAE	COLLADA
DXF	AutoCAD
LWO	Lightwave
OBJ	Wavefront Technologies
SKP	Google sketchup
WRL	VRML

Each of the formats listed in Table 12-5 has its special characteristics, goals, virtues and shortcomings. Among all of these formats, we are most interested in COLLADA which is the only format we shall use and discuss in this chapter (*http://collada.org*). COLLADA, short for COLLAborative Design Activity, is an open-source 3D graphics format managed by the Khronos Group (*http://www.khronos.org/*), which is a nonprofit industry consortium whose main tasks are to create open standards and royalty-free APIs to enhance the authoring and acceleration of parallel computing, graphics and dynamic media on a wide variety of platforms and devices. COLLADA is royalty-free and is an XML-based schema. XML, short for Extensible Markup Language, is a metalanguage using extra "markup" (information enclosed between angle brackets) for defining a set of rules to encode documents in a format that can be understood by humans and machines easily. It is produced and maintained by the World Wide Web Consortium (W3C).

COLLADA defines an XML database schema that helps 3-D authoring applications to freely exchange digital assets without loss of information, and enables multiple software packages to be combined into powerful tool chains. Many large technology companies or organizations support the work of COLLADA. These include Sony Computer Entertainment, NVIDIA, ATI, Softimage, Autodesk, Google, and Intel. Moreover, some major game engines such as OGRE, C4 Engine, AgentFX, Multiverse, PhyreEngine, and CryEngine also support COLLADA.

12.4.2 COLLADA Features

One main feature of COLLADA is that it is an open-source standard. Its schema and specifications are freely available from The Khronos Group. The standard allows graphics software packages interchange data in an effective way. One can import art assets created with other software packages or to export them to other packages. An **art asset** or sometimes referred to as media asset in computer graphics is an individual piece of digital media used in the creation of a larger graphics scene. Art assets include synthetic and photographic bitmaps, 3D models of meshes, shaders, motion captured or artificial animation data, and video samples. The term "art" here refers to a piece of work created from a piece of software; its data are not necessarily represent anything artistic. COLLADA supports a wide variety graphics and related features, which include the following:

1. Mesh geometry

2. Transform hierarchy (rotation, translation, shear, scale, matrix)
3. Effects
4. Shaders (Cg, GLSL, GLES)
5. Materials
6. Textures
7. Lights
8. Cameras
9. Skinning
10. Animation
11. Physics (rigid bodies, constraints, rag dolls, collision, volumes)
12. Instantiation
13. Techniques
14. Multirepresentations
15. Assets
16. User data
17. Kinematics
18. Boundary representations (B-reps)
19. Geographic coverage
20. Custom image initialization
21. Math formulas

Since a COLLADA file is a text file consisting of data with XML tags, all COLLADA documents can be edited by an ordinary text editor such as **vi**, **notepad** and **emacs**. After we have edited a COLLADA document, we should always validate it manually to ensure that the document is correctly formatted. A COLLADA file ends with extension ".dae" and there are a couple of ways to validate a COLLADA document.

Validating COLLADA Document

We can use the XML tool, **xmllint** of the **libxml** package, which we will discuss below, to validate a COLLADA document against the COLLADA schema. Suppose we have a COLLADA file called "cube.dae". We check whether this file is correctly formatted using a command like the following:

```
$xmllint --noout --schema  \
 http://www.khronos.org/files/collada_schema_1_4 cube.dae
```

It may be more convenient if we first download the schema and save it as a local file; this can be done by the command,

```
$wget http://www.khronos.org/files/collada_schema_1_4
```

and then rename the downloaded file to "colladaSchema.xsd":

```
$mv collada_schema_1_4 colladaSchema.xsd
```

Then we can validate the document using the command

```
$xmllint --noout --schema colladaSchema.xsd cube.dae
```

The option "–noout" is used to prevent superfluous output. If the file is valid, the following message is displayed:

```
      cube.dae validates
```

If the document is invalid, we may see a message similar the following:

```
cube.dae:9: element source_data: Schemas validity error : Element
'{http://www.collada.org/2005/11/COLLADASchema}source_data':
'file://' is not a valid value of the atomic type 'xs:anyURI'.
cube.dae:196: element instance_rigid_body: Schemas validity error :
Element
'{http://www.collada.org/2005/11/COLLADASchema}instance_rigid_body':
Missing  child element(s). Expected is
({http://www.collada.org/2005/11/COLLADASchema}technique_common ).
cube.dae:194: element physics_scene: Schemas validity error : Element
'{http://www.collada.org/2005/11/COLLADASchema}physics_scene':
Missing child element(s). Expected is one of (
{http://www.collada.org/2005/11/COLLADASchema}instance_physics_model,
{http://www.collada.org/2005/11/COLLADASchema}technique_common ).
cube.dae fails to validate
```

12.4.3 COLLADA Format

In the subsequent discussions of COLLADA, we will refer to a COLLADA file called “cube.dae” to explain the basic format of COLLADA. The file “cube.dae” is exported from the free open-source 3D content creation suite, **Blender** (*http://www.blender.org/*), version 2.61. This file is part of the resource distribution at this book’s web site at

http://www.forejune.com/stereo/.

Actually, you can easily obtain a similar file from Blender which always starts with a default cube object similar to the one we have used. We will discuss briefly the Blender suite later in this chapter.

Like viewing any COLLADA file, we can view “cube.dae” with a text editor such as **vi** and **emacs**, or we can view it using a browser such as Firefox that supports XML. If we view it with a browser, we should see the structure of the file which looks like the following:

```
-<COLLADA version="1.4.1">
  -<asset>
    -<contributor>
       <author>Blender User</author>
       <authoring_tool>Blender 2.61.0 r42614</authoring_tool>
     </contributor>
     <created>2012-01-04T10:45:03</created>
     <modified>2012-01-04T10:45:03</modified>
     <unit name="meter" meter="1"/>
     <up_axis>Z_UP</up_axis>
   </asset>
-<library_cameras>
 -<camera id="Camera-camera" name="Camera">
   -<optics>
     -<technique_common>
       -<perspective>
          <xfov sid="xfov">49.13434</xfov>
          <aspect_ratio>1.777778</aspect_ratio>
          <znear sid="znear">0.1</znear>
          <zfar sid="zfar">100</zfar>
          </perspective>
        </technique_common>
      </optics>
    </camera>
  </library_cameras>
............
  -<scene>
     <instance_visual_scene url="#Scene"/>
   </scene>
</COLLADA>
```

The '-' sign in front of a tag indicates the beginning of a structure (node). We can click on the '-' sign to collapse the structure, which also changes the '-' sign to '+'. For example, when we click on the '-' sign in front of the <COLLADA> tag, the whole document will collapse to a single line like the following:

```
+<COLLADA version="1.4.1"> </COLLADA>
```

Clicking on the '+' sign will expand the document.

XML organizes data in a tree structure and refers to each structure enclosed by a beginning tag and ending tag as a node. For example, the <mesh> node starts with <mesh> and ends with </mesh>. The <COLLADA> node is is the root of a COLLADA document. COLLADA defines a lot of nodes to describe various graphics object attributes. As an introduction, we only consider a few simple nodes. We list some of the COLLADA nodes below.

Reading Geometry Data

1. <**library_geometries**>:
   This node is a library that contains geometry type nodes that define geometries in the scene.
2. <**mesh**>:
   This node contains the geometry data of the mesh. It usually contains a few <source> child nodes that define the data of vertices, normals and texture.
3. <**source**>:
   This node contains child nodes such as <float_array> and <technique_common> that define the geometry data.
4. <**float_array**>:
   This node contains floating point numbers for defining various attributes, which are described by a sibling node of type <technique_common>.
5. <**technique_common**>:
   This node's <accessor> child node specifies the data usage for the arrays defined in <float_array> or <Name_array>. (<Name_array> is similar to <float_array> except that it specifies strings instead of floating point numbers.)

In the file "cube.dae", you will find one <library_geometries> node, which has one <geometry> child node. The <geometry> node has one <mesh> child node that defines the polygon mesh of the cube object. The following is a <source> node example taken from "cube.dae", which defines the data of the vertices of a cube. As you can imagine, a cube has 8 vertices, and each vertex has 3 coordinate values. So there are totally 24 data values.

```
<source id="Cube-mesh-positions">
  <float_array id="Cube-mesh-positions-array" count="24">1 1 -1 1 -1 -1
       -1 -0.98 -1 -0.97 1 -1 1 0.95 1 0.94 -1.01 1 -1 -0.97 1 -1 1 1
  </float_array>
  <technique_common>
    <accessor source="#Cube-mesh-positions-array" count="8" stride="3">
      <param name="X" type="float"/>
      <param name="Y" type="float"/>
      <param name="Z" type="float"/>
    </accessor>
  </technique_common>
</source>
```

In this example, the <float_array> node defines 24 floats (i.e. count="24"). The <accessor> node tells us how to interpret the data; it has three <param> child nodes describing the (x, y, z) coordinates of a vertex. The attribute stride="3" means the next vertex is 3 floats away from the current one; the count="8" attribute indicates that there are 8 vertices. In summary, the <source> node describes that there are 8 vertices, each with 3 components, which are saved in <float_array> as 24 float values; the components are called "X", "Y" and "Z". (If the <source> contains texture coordinates, then the components would be called "S", "T" and "P".) Actually, such a node could also describe the (x, y, z) coordinates of a normal vector. To distinguish whether we are processing a vertice or a normal vector, we have to read another child node of <mesh> called <vertices> to find the vertices source; this node contains a child node named <input> with a semantic attribute of "POSITION" value.

```
<vertices id="Cube-mesh-vertices">
  <input semantic="POSITION" source="#Cube-mesh-positions"/>
</vertices>
```

If you navigate through the file "cube.dae", you will find 2 <source> nodes (they are children of <mesh>). One of them defines the vertices as discussed above. You might have guessed that the other <source> node defines the normal coordinates. Indeed, your guess is right; the normal <source> structure is very similar to that of the vertex <source>. The following is the corresponding normal segment taken from the file "cube.dae":

```
<source id="Cube-mesh-normals">
   <float_array id="Cube-mesh-normals-array" count="18">0 0 -1  0 0 1
     1 -2.83e-7 0  -2.83e-7 -1 0  -1 2.23e-7 -1.34e-7 2.38e-7 1 2.08e-7
   </float_array>
   <technique_common>
     <accessor source="#Cube-mesh-normals-array" count="6" stride="3">
        <param name="X" type="float"/>
        <param name="Y" type="float"/>
        <param name="Z" type="float"/>
     </accessor>
   </technique_common>
</source>
```

As you can see, a cube has 6 faces and thus we have 6 normals, one for each face. Therefore, the accessor source count is "6". Since each normal is specified by 3 numbers, we need a total of $6 \times 3 = 18$ data values. So the float_array count is "18".

In the file "cube.dae", another child of <source> is the <polylist> node. You might have guessed that this node describes the list of polygons of the mesh. Again, your guess is right. This node defines the polygon (face) list of the mesh just as we discuss in Section 12.3. The following shows the xml code of this node taken from the file "cube.dae".

```
<polylist material="Material1" count="6">
   <input semantic="VERTEX" source="#Cube-mesh-vertices" offset="0"/>
   <input semantic="NORMAL" source="#Cube-mesh-normals" offset="1"/>
   <vcount>4 4 4 4 4 4 </vcount>
   <p>0 0  1 0  2 0  3 0   4 1  7 1  6 1  5 1
      0 2  4 2  5 2  1 2   1 3  5 3  6 3  2 3
      2 4  6 4  7 4  3 4   4 5  0 5  3 5  7 5</p>
</polylist>
```

In the node <polylist>, we see that count="6" indicating that the list has 6 polygons. Inside the <polylist> node, the child node **vcount** indicates "vertex count", the number of vertices a polygon (face) has. Obviously, each face of a cube has 4 vertices. That is why we see six 4's for the six faces in the <vcount> node. The <p> node consists of the indices of the vertex and the normal coordinates of each of the six faces. The two <input> nodes tell us which is which. For each polygon, the first index is for the vertex tuple (offsets="0") and the second index is for the normal tuple. So the first polygon is specified by

```
0 0  1 0  2 0  3 0
```

meaning that the polygon consists of vertices 0, 1, 2, and 3, and the normal at each of the vertex is normal 0.

From these data, we can reconstruct tables similar to those of Table 12-1 to 12-3, specifying a normal list, a polygon list and a vertex list. The following are the tables thus constructed.

Table 12-6 Vertex List

Vertex	Coordinates (x, y, z)
0	$(1, 1, -1)$
1	$(1, -1, -1)$
2	$(-1, -0.98, -1)$
3	$(-0.97, 1, -1)$
4	$(1, 0.95, 1)$
5	$(0.94, -1.01, 1)$
6	$(-1, -0.97, 1)$
7	$(-1, 1, 1)$

Table 12-7 Normal List

Normal	(n_x, n_y, n_z)
0	$(0, 0, -1)$
1	$(0, 0, 1)$
2	$(1, -2.83 \times 10^{-7}, 0)$
3	$(-2.83 \times 10^{-7}, -1, 0)$
4	$(-1, -2.23 \times 10^{-7}, -1.34 \times 10^{-7})$
5	$(2.38 \times 10^{-7}, 1, 2.08 \times 10^{-7})$

Table 12-8 Polygon List

Polygon	vcount	Vertices	Normals
0	4	0, 1, 2, 3	0,0,0,0
1	4	4, 7 , 6, 5	1,1,1,1
2	4	0, 4, 5, 1	2,2,2,2
3	4	1, 5, 6, 2	3,3,3,3
4	4	2, 6, 7, 3	4,4,4,4
5	4	4, 5, 3, 7	5,5,5,5

For example, face (polygon) 2 has 4 vertices, which are vertex 0, 4, 5, and 1 and the coordinates of these vertices are shown in Table 12-6 as

$$(1, 1, -1), (1, 0.95, 1), (0.94, -1.01, 1), (1, -1, -1)$$

The normal to this face is normal 2 with components shown in Table 12-7 as

$$(1, -2.83 \times 10^{-7}, 0)$$

12.4.3 Parsing COLLADA Files

After understanding the basic features of a COLLADA file, the next thing we want to do is to extract the mesh data and process or render them using our OpenGL programs. The collada.org provides a package called **COLLADA Document Object Model (DOM)** for loading, saving, and parsing COLLADA files. The package is a C++ library which provides rich features to process COLLADA data. However, this package is huge and require special libraries such as the *Boost Filesystem* library to build; it is fairly difficult to compile. For beginners who are interested to work in the open-source environment, we want something simpler to accomplish our tasks of studying, understanding, and developing graphics applications with COLLADA files. Since a COLLADA file is basically an xml file, to parse it, all we need is a simple xml parser. There are quite a few C/C++ open-source xml parsers. The one we have chosen to study and use is the *libxml* library maintained by Daniel Veillard (*http://xmlsoft.org/*). Actually, the one we are going to use is *libxml2*, which is a newer version of the library. Interestingly, the DOM package is also built on top of libxml.

12.5 The libxml Library

12.5.1 Introduction

The *libxml2* library, a newer version of *libxml*, is an XML parser and toolkit developed for the Gnome project. It is written in C and is free software available under the MIT License. It is known to be fast and very portable, working effectively on a variety of systems, including Linux, Unix, Windows, CygWin, MacOS, MacOS X, RISC Os, OS/2, VMS, QNX, MVS, and VxWorks.

According to its offical website at *http://xmlsoft.org/*, *libxml2* implements a number of existing standards related to markup languages:

1. the XML standard: http://www.w3.org/TR/REC-xml
2. Namespaces in XML: http://www.w3.org/TR/REC-xml-names/
3. XML Base: http://www.w3.org/TR/xmlbase/
4. RFC 2396 : Uniform Resource Identifiers http://www.ietf.org/rfc/rfc2396.txt
5. XML Path Language (XPath) 1.0: http://www.w3.org/TR/xpath
6. HTML4 parser: http://www.w3.org/TR/html401/
7. XML Pointer Language (XPointer) Version 1.0: http://www.w3.org/TR/xptr
8. XML Inclusions (XInclude) Version 1.0: http://www.w3.org/TR/xinclude/
9. ISO-8859-x encodings, as well as rfc2044 [UTF-8] and rfc2781 [UTF-16] Unicode encodings, and more if using iconv support
10. part of SGML Open Technical Resolution TR9401:1997
11. XML Catalogs Working Draft 06 August 2001: http://www.oasis-open.org/committees/entity/spec-2001-08-06.html
12. Canonical XML Version 1.0: http://www.w3.org/TR/xml-c14n and the Exclusive XML Canonicalization CR draft http://www.w3.org/TR/xml-exc-c14n
13. Relax NG, ISO/IEC 19757-2:2003, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html
14. W3C XML Schemas Part 2: Datatypes REC 02 May 2001
15. W3C xml:id Working Draft 7 April 2004

The *libxml* web site provides coding examples and details of the API's of the library. There are 3 main API modules; the *Parser API* provides interfaces, constants and types related to the XML parser; the *Tree API* allows users to access and process the tree structures of an XML or HTML document; and the Reader API provides functions to read, write, and validate XML files, and allows users to extract the data and attributes recursively.

Moreover, the web site also provides some information about XML, detailed enough to understand and use *libxml*.

12.5.2 Reading and Parsing an XML File

We can find practical coding examples of libxml2 at

http://xmlsoft.org/examples

We will first discuss how to read and parse an XML file. We present a program called "readxml.cpp" to illustrate the concept; most of the code we are presenting is from the above site's program "reader1.c", which uses the function **xmlReaderForFile**() to parse an XML file and print out the information about the nodes found in the process. To use the function, we have to include the header statement,

#include <libxml/xmlreader.h>

For simplicity, we hard-code the file name for testing to be "cube.dae"; this is the COLLADA file we have used in the above section. In this program, the **main**() function will open the file "cube.dae" for parsing:

```
int main()
{
  // Initializes library and check whether correct version is used.
  LIBXML_TEST_VERSION

  const char filename[] = "cube.dae";
  xmlTextReaderPtr reader = xmlReaderForFile( filename, NULL, 0 );
  if ( reader == NULL ) {
      fprintf(stderr, "Unable to open %s\n", filename);
      return 1;
  }

  int ret = xmlTextReaderRead(reader);
  while (ret == 1) {
      processNode(reader);
      ret = xmlTextReaderRead(reader);
  }
  xmlFreeTextReader(reader);
  if (ret != 0) {
    fprintf(stderr, "%s : failed to parse\n", filename);
  }

  //Cleanup function for the XML library.
  xmlCleanupParser();

  return 0;
}
```

The function **xmlReaderForFile**() is used to open the file; this function can parse an XML file from the filesystem or the network (the first input parameter can be an URL); it returns a pointer *reader*, pointing to the new **xmlTextReader** or NULL if an error has occurred. This returned *reader* will be used as a handle for further processing of the document.

Next, the function **xmlTextReaderRead**() is used to move the position of the current node pointer to the next node in the stream, exposing its properties; it returns 1 if the node was read successfully, and 0 if there are no more nodes to read, or -1 in case of error. Therefore, we setup a while loop so that as long as the returned value is 1 (node read successfully), we call the function **processNode**() discussed below to process the information contained in the node. When finished reading the file, the function **xmlFreeTextReader**() is called to deallocate all the resources associated with the reader. At the end, the function **xmlCleanupParser**() is called to clean up the memory allocated by the library; this function name is somewhat misleading as it does not clean up parser state. It is a cleanup function for the XML library. It tries to reclaim all related global memory allocated for the library processing but it does not deallocate any memory associated with the document.

One can write the function **processNode**( ) to process the information of the current node in any way the application requires. In our example, we just print out the attributes and the value of the node. We used the following libxml2 functions to obtain the information:

1. **xmlTextReaderConstValue**() reads the text value of the current node; the function returns the string of the value or NULL if the value is not available. The result will be deallocated on the next read operation.
2. **xmlTextReaderDepth**() reads the depth of the node in the tree; it returns the depth or -1 in case of error.
3. **xmlTextReaderNodeType**() gets the node type of the current node; it returns the xmlNodeType of the current node or -1 in case of error. The node type is defined at the link,

   http://www.gnu.org/software/dotgnu/pnetlib-doc/System/Xml/XmlNodeType.html

4. **xmlTextReaderIsEmptyElement**() checks whether the current node is empty; it returns 1 if empty, 0 if not and -1 in case of error.
5. **xmlTextReaderHasValue**() is used to check whether the node can have a text value; it returns 1 if true, 0 if false, and -1 in case or error.

When we compile the program, we need to link it with the libxml2 library. We may use a command similar to the following to generate the executable:

```
g++  -o readxml readxml.cpp  -lxml2 -L/{\it libxml2\_dir}/libxml2/lib \
     -I/{\it libxml2\_dir}/libxml2/include/libxml2
```

When "readxml" is executed, it will read the COLLADA file "cube.dae" and prints out the information of each node, which looks like the following output:

```
 1: 0 1 COLLADA 0 0
 2: 1 14 #text 0 1    value=

 3: 1 1 asset 0 0
 4: 2 14 #text 0 1    value=

 5: 2 1 contributor 0 0
 6: 3 14 #text 0 1    value=

 7: 3 1 author 0 0
 8: 4 3 #text 0 1    value= Blender User
 9: 3 15 author 0 0
10: 3 14 #text 0 1    value=
 11: 3 1 authoring_tool 0 0
 12: 4 3 #text 0 1    value= Blender 2.61.0 r42614
 13: 3 15 authoring_tool 0 0
 14: 3 14 #text 0 1    value=

 15: 2 15 contributor 0 0
 16: 2 14 #text 0 1    value=

 17: 2 1 created 0 0
 18: 3 3 #text 0 1    value= 2012-01-04T10:45:03
 19: 2 15 created 0 0
 20: 2 14 #text 0 1    value=
 .....
 39: 4 1 technique_common 0 0
 40: 5 14 #text 0 1    value=

 41: 5 1 perspective 0 0
 42: 6 14 #text 0 1    value=

 43: 6 1 xfov 0 0
 44: 7 3 #text 0 1    value= 49.13434
 45: 6 15 xfov 0 0
 46: 6 14 #text 0 1    value=

 47: 6 1 aspect_ratio 0 0
 48: 7 3 #text 0 1    value= 1.777778
 49: 6 15 aspect_ratio 0 0
 50: 6 14 #text 0 1    value=

 51: 6 1 znear 0 0
 52: 7 3 #text 0 1    value= 0.1
 53: 6 15 znear 0 0
 54: 6 14 #text 0 1    value=
 .....
609: 1 15 library_visual_scenes 0 0
610: 1 14 #text 0 1    value=

611: 1 1 scene 0 0
612: 2 14 #text 0 1    value=

613: 2 1 instance_visual_scene 1 0
614: 2 14 #text 0 1    value=
615: 1 15 scene 0 0
616: 1 14 #text 0 1    value=
617: 0 15 COLLADA 0 0
```

12.5.3 Parsing a File To a Tree

The example here, “treexml.cpp” is based on the program “tree1.c” example the libxml2 web site. The program parses an xml file to a tree. It first uses **xmlDocGetRootElement**() to get the root element. Then it scans the document and prints all the element names in document order. In the program, we have to include the following header statements:

```
#include <libxml/parser.h>
#include <libxml/tree.h>
#include <libxml/xmlreader.h>
```

For simplicity, we again hard-code “cube.dae” as our file name. The **main**() of the program uses **xmlReadFile**(), which belongs to the parser API of libxml2, to open “cube.dae”; this function parses an XML file from the filesystem or the network; it returns a pointer pointing to the document tree and NULL on failure. The **main**() function then uses **xmlDocGetRootElement**() to obtain the pointer pointing to the root of the document tree and calls **print_element_names**() recursively to print out all the element names of the nodes in the tree:

```
int main()
{
  xmlDoc *doc = NULL;
  xmlNode *root = NULL;
  const char filename[] = "cube.dae";

  // Initialize library and check whether correct version is used.
  LIBXML_TEST_VERSION
  // parse the file and get the DOM
  doc = xmlReadFile( filename, NULL, 0);

  if ( doc == NULL ) {
    printf( "error: could not parse file %s\n", filename );
    return 1;
  }

  // Get the root element node
  root = xmlDocGetRootElement( doc );

  //starts printing from the root at level -1
  print_element_names( root, -1 );

  xmlFreeDoc(doc);
  xmlCleanupParser();

  return 0;
}
```

The function **print_element_names**() takes in two input arguments; the first is a pointer pointing to an **xmlDoc** node, a root of a subtree of the document tree. The second argument is the relative level of the node. each time the pointer moves a level deeper, the value of level is incremented by one; a ‘+’ is printed for the advance of a level along with the node element name:

```
void print_element_names(xmlNode *root, int level )
{
  xmlNode *cur_node = NULL;
  ++level;   //one level deeper in next call

  for (cur_node = root; cur_node; cur_node = cur_node->next) {
    if (cur_node->type == XML_ELEMENT_NODE) {
      for ( int i = 0; i < level; i++ )
        printf(" +");  //signifies level of node
      printf(" %s\n", cur_node->name);
    }
    print_element_names( cur_node->children, level );
  }
}
```

Again, this program can be easily compiled and linked with a command like “g++ -o treexml treexml.cpp -lxml2”. When we run the executable “treexml”, outputs similar to the following will be generated:

```
COLLADA
```

```
+ asset
+ + contributor
+ + + author
+ + + authoring_tool
+ + created
+ + modified
+ + unit
+ + up_axis
+ library_cameras
+ + camera
+ + + optics
+ + + + technique_common
+ + + + + perspective
+ + + + + + xfov
+ + + + + + aspect_ratio
+ + + + + + znear
+ + + + + + zfar
+ library_lights
+ + light
+ + + technique_common
+ + + + point
+ + + + + color
+ + + + + constant_attenuation
+ + + + + linear_attenuation
+ + + + + quadratic_attenuation
+ + + extra
+ + + + technique
............
+ library_visual_scenes
+ + visual_scene
+ + + node
+ + + + translate
+ + + + rotate
+ + + + rotate
+ + + + rotate
+ + + + scale
+ + + + instance_geometry
+ + + + + bind_material
+ + + + + + technique_common
+ + + + + + + instance_material
+ + + node
+ + + + translate
+ + + + rotate
+ + + + rotate
+ + + + rotate
+ + + + scale
+ + + + instance_camera
+ scene
+ + instance_visual_scene
```

12.5.4 Searching For a Node

A COLLADA file is an XML file organizing information in a tree structure. The previous section shows how to parse such a file into a tree. Given the tree, we can search for a particular node and extract the information from the node or the subtree rooted at the searched node. Here, we discuss how to search for a node and a few ways to extract the information of the node. Again, we use the COLLADA file "cube.dae" generated by Blender in our example.

Given the root of a subtree of the document, we can search a target node specified by a key string. We first search the root, its siblings and all the children of them and then we search recursively the subtrees of each child until the target is found. This can be accomplished by the following function **searchNode**(), which returns a pointer to the node associated with the target key or NULL if the target is not found:

```
xmlNode *searchNode ( xmlNode *a_node, char target[] )
{
  xmlNode *nodeFound = NULL;

  for ( xmlNode *cur = a_node; cur; cur= cur->next) {
    if (cur->type == XML_ELEMENT_NODE) {
      if ( !xmlStrcmp (  cur->name, (const xmlChar* )target ) ) {
         printf("Found %s \n", cur->name );
         nodeFound = cur;
         break;
      }
    }
    //search recursively until node is found.
    if ( nodeFound == NULL && cur != NULL )
      nodeFound = searchNode ( cur->children, target );
  }

  return nodeFound;
}
```

In practice, there could be more than one node that has the same target key. If we need all the targeted nodes, we can save the node in a vector rather than breaking out of the **for** loop of **searchNode**() after one node has been found. The following main code first searches for nodes associated with key strings "source" and print out the subtrees rooted at the "source" nodes. It then searches for "float_array" from the first "source" subtree. If the node is found, it calls **parseNode**() to print out the data of the node. The function **parseNode**() makes use of **xmlNodeListGetString**() to get the content of the node; this function builds the string equivalent to the text contained in the node list made of TEXTs and ENTITY_REFs; it returns a pointer to the string copy, which must be freed by the caller with **xmlFree**():

```
void parseNode ( xmlDocPtr doc, xmlNodePtr cur )
{
  xmlChar *key;
  cur = cur->xmlChildrenNode;
  while (cur != NULL) {
      key = xmlNodeListGetString(doc, cur, 1);
      printf(" %s\n", key);
      xmlFree(key);
      cur = cur->next;
  }
  return;
}

int main(int argc, char **argv)
{
  xmlDoc *doc = NULL;
  xmlNode *root_element = NULL;
  const char filename[] = "cube.dae";

  // Initialize library and check whether correct version is used.
  LIBXML_TEST_VERSION

  // parse the file and get the DOM
```

```
    doc = xmlReadFile( filename, NULL, 0);

    if ( doc == NULL ) {
        printf( "error: could not parse file %s\n", filename );
    }
    // Get the root element node
    root_element = xmlDocGetRootElement( doc );

    xmlNode *nodeFound;

    nodeFound = searchNode ( root_element, "source" );
    //print subtrees rooted at "source"
    print_element_names(nodeFound, -1);
    //find "float_array" within first "source" subtree
    if ( nodeFound != NULL ){
       nodeFound = searchNode ( nodeFound, "float_array" );
    }
    //print out data of node found
    if ( nodeFound != NULL )
      parseNode ( doc, nodeFound );
     printf("--------------------------------------------------------\n");
     ......
  }
```

When this code is executed, it will generate an output similar to the following:

```
Found source
 source
 + float_array
 + technique_common
 + + accessor
 + + + param
 + + + param
 + + + param
 source
 + float_array
 + technique_common
 + + accessor
 + + + param
 + + + param
 + + + param
 vertices
 + input
 polylist
 + input
 + input
 + vcount
 + p
Found float_array
 1 1 -1 1 -1 -1 -1 -0.9999998 -1 -0.9999997 1 -1 1 0.9999995 1
 0.9999994 -1.000001 1 -1 -0.9999997 1 -1 1 1
---------------------------------------------------------
```

If you use **xmlNodeListGetString**() to find and print the "content" of a node, very often you may find that an empty line is printed. According to the documentation of libxml2, the empty lines of the elements that do not have text content but have child elements in them are actually part of the document, even if they are supposedly only for formatting; libxml2 could not determine whether a text node with blank content is for formatting or data for the user. The application programmer has to know whether the text node is part of the application data or not and has to do the appropriate filtering. On the other hand, libxml2 provides a function called **xmlElemDump**() that allows a user to dump the subtree rooted at the specified node to a file; actually, we can use the function to dump to a pipe and use some popular text processing utilites such as **sed** and **awk** to further parse it, or we can direct it to a buffer using the C function **setvbuf** for further processing in our program. The following code segment of **main**() shows how this is done. (For simplicity, we have omitted some error-checking code.)

```
  int main(int argc, char **argv)
  {
    xmlDoc *doc = NULL;
    doc = xmlReadFile( filename, NULL, 0);
    ......
    printf("---------------------------------------------------------\n");

    nodeFound = searchNode ( root_element, "library_geometries" );
    nodeFound = searchNode ( nodeFound, "vertices" );
    if ( nodeFound == NULL ) {
      printf("\nvertices Node not found!\n");
      return 1;
    }
    //send information to screen
    xmlElemDump ( stdout, doc, nodeFound );
    printf("\nparsing the node:\n");
    //open a pipe, which will execute the awk command when writing
    FILE *fp = popen ( "awk 'BEGIN {} {for(i=NF;i > 0;i--)
                            printf(\"%s\\n\",$i); } END {} '", "w" );
    //print the parsed text to screen
    xmlElemDump ( fp, doc, nodeFound );
    pclose ( fp );  //close the pipe

    printf("---------------------------------------------------------\n");

    //search a new node
    nodeFound = searchNode ( nodeFound, "polylist" );
    if ( nodeFound == NULL ) return 1;
    //open a temporary file
    FILE *f = fopen ("temp.$$$", "w" );
    const int bufsize = 20000;
    char buf[bufsize];       //create a buffer
    bzero ( buf, bufsize );  //set buffer to zeros
    //send the output stream to buf before writing to "temp.$$$"
    setvbuf ( f, buf, _IOFBF, bufsize );

    //dump the element of node to the buffer
    xmlElemDump ( f, doc, nodeFound );
    printf("%s\n", buf ); //print content of node

    /*free the document */
    xmlFreeDoc(doc);

    xmlCleanupParser();

    return 0;
}
```

In the code, we first search for the "library_geometries" node and then search this subtree to find the "vertices" node. Then we use the statement "xmlElemDump (stdout, doc, nodeFound);" to dump the subtree rooted at "vertices" to screen. To demonstrate how to parse the information, we open a pipe using **popen**(), which has the prototype,

FILE *popen(const char **command*, const char **type*);

where *command* is the command to be issued and *type* is a string indicating the operation mode with “r” indicating reading data from the pipe obtained from output of *command*, and “w” indicating writing data to the pipe which sends them to the command. In our example the *command* is to use awk to parse the text.

The last part of the code demonstrates the dumping of a subtree to a buffer. We first search for the “polylist” node. After we have found the node, we open a temporary file named “temp.$$$” for writing information. We declare a buffer named *buf* and use the **setvbuf**() to buffer the file output so that the node information will be dumped to *buf* when the function **xmlElemDump**() is called. Then we print out the text stored in *buf*.

When this code is executed, outputs similar to the following will be generated:

```
Found library_geometries
Found vertices
<vertices id="Cube-mesh-vertices">
          <input semantic="POSITION" source="#Cube-mesh-positions"/>
        </vertices>
parsing the node:
id="Cube-mesh-vertices">
<vertices
source="#Cube-mesh-positions"/>
semantic="POSITION"
<input
</vertices>
---------------------------------------------------------
Found polylist
<polylist material="Material1" count="6">
        <input semantic="VERTEX" source="#Cube-mesh-vertices" offset="0"/>
        <input semantic="NORMAL" source="#Cube-mesh-normals" offset="1"/>
        <vcount>4 4 4 4 4 4 </vcount>
        <p>0 0 1 0 2 0 3 0 4 1 7 1 6 1 5 1 0 2 4 2 5 2 1 2 1 3 5 3 6 3 2
                         3 2 4 6 4 7 4 3 4 4 5 0 5 3 5 7 5</p>
      </polylist>
```

12.6 Blender

Blender is an open-source 3D content creation suite, free and available for all major operating systems including Linux, Windows and Mac OS/X under the GNU General Public License (*http://www.blender.org/*). One can do and create a lot of amazing graphics using Blender. To name a few, we can use Blender to model a human head basemesh that can easily be used as a starting point for sculpting, to create a modern wind turbine with lighting, to model an entire building exterior from photo references, to use simple shapes and alpha masked leaves to create a tree with realistic look, or to create an animated movie. There are a lot of tutorials on the Web about using Blender. For example, besides the official site (www.blender.org), the site of **BlenderArt Magazine** (*http://blenderart.org/*) also offers a few good tutorials. Our COLLADA file “cube.dae” used in the examples of the previous sections is created using Blender. Figure 12-8 shows a screen capture of the Blender interface when creating the cube.

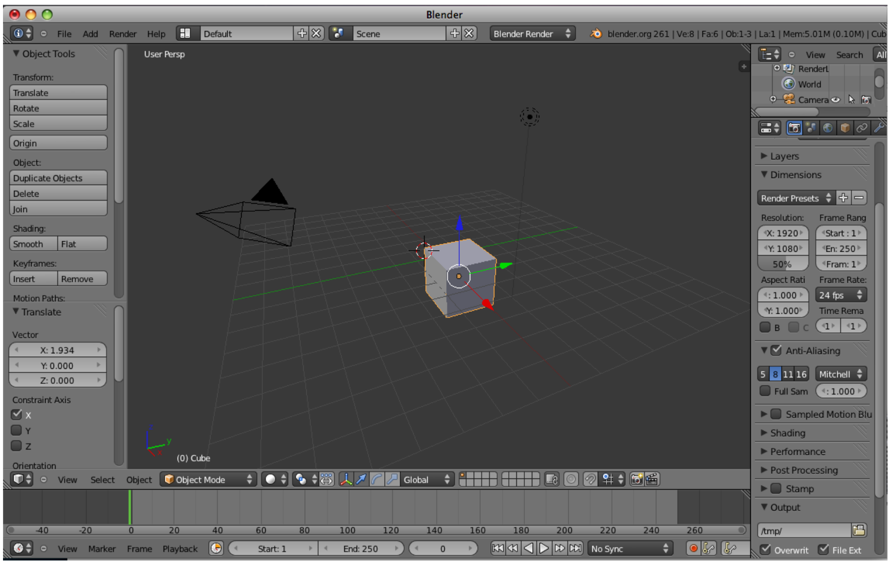

Figure 12-8 Blender Interface

With the help of Blender, we can largely extend our capabilities of creating graphics using OpenGL and C/C++. Very often, our application may need a graphics object that is difficult to create from sketch using OpenGL; in this case, we can use Blender to create the object and save it as a COLLADA file. Our C/C++ OpenGL program can parse the COLLADA file using libxml2 discussed above, and further fine-tune or process it with OpenGL commands. Or sometimes we can use Blender to import a graphics object from the Internet saved in another 3D file format, and use Blender to export the file to the COLLADA format, which can then be parsed and rendered by our application. (The site at *http://www.hongkiat.com/blog/60-excellent-free-3d-model-websites/* lists 60 excellent free 3D model Websites, which have free 3D models in various formats available for download.) The import and export procedures are straightforward. Using a 3DS file as an example, to import the 3DS object, we can issue the following sequence of "commands" (clicking menu or entering file name) in the Blender IDE:

File > **Import** > **3D Studio (3ds)** > *Select 3DS file* > **Import 3DS**

The imported object will appear in the Blender IDE.

To export a graphics object from the Blender IDE to a COLLADA file, we can issue the following sequence of "commands":

File > **Export** > **COLLADA (.dae)** > *Enter File name* > **Export COLLADA**

After obtaining the exported COLLADA file, we can parse it and incorporate it in our OpenGL application.

In conclusion, the availability of open-source software packages has created unlimited opportunities for every body. It brings democracy to the world, enriches our life, and makes big positive impact to the planet. As long as one pays effort to acquire the knowledge of the modern world, he or she will find that the world is abundant and beautiful.