A Framework for Drawing the Architecture of Buildings Using OpenGL and VC++

ShiQing Zhao

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

August 2002

Acknowledgements

I would like to take this opportunity to express my sincere thanks to my supervisor, Dr. Peter Grogono, for his conscientious supervision, his creative ideas, his constructive advice, and his erudite knowledge in Computer Science that sustained me throughout the development of this major report.

Many thanks to Dr. Adam Crzyzak, who kindly took time to review the report.

I also wish to thank Dr. Clement Lam and Dr. Sudhir P Mudur, in whose classes I acquired broad and detailed knowledge in Computer Graphics and OpenGL.

Furthermore, thanks to all the professors and staff in the department of computer science, especially Halina Monkiewicz, for their excellent support during my studies.

Finally, I would like to thank my parents, grandparents, relatives and good friends for their love, encouragement, and support.

Abstract

A Framework for Drawing the Architecture of Buildings

Using OpenGL and VC++

ShiQing Zhao

This report presents the design and implementation of a Framework for drawing the Architecture of Buildings. as well as a brief introduction to computer graphics. OpenGL. framework. and the advent of framework.

In the design phase. we discourse on how we design each class in the framework.

In the implementation phase. we demonstrate the programming process with an emphasis on the methods that will be overridden.

We also implemented two sample applications based on the framework to illustrate how application developers implement new applications by using the framework. and consequentially to testify the advent of the framework.

The conclusion is that successful framework is a reuse technique for developing new applications. and has potentiality to improve the efficiency and quality of application software development.

Contents

List of Figures

1. Introduction

The development of computer software proceeded from an initial concern with programming alone, through increasing interest in design, to a concern with analysis methods. The potentiality of reuse is always an important issue at different times.

At a time when the sole concern was programming alone, people created libraries that consisted of commonly used functions or procedures, so that software developers could call the functions in libraries as they developed their software. This required the reuse of the code.

At the time of OOAD, people created commonly used classes, so that software developers could implement their application through inheritance and composition of classes. This entailed the reuse of classes.

Now, people are demanding a higher degree of reuse, including the reuse of analysis, design, classes and code. Frameworks are regarded as a means of satisfying such requirements. Frameworks are generally targeted for a particular application domain, such as user interface, business data processing systems, telecommunications, or multimedia collaborative work environments. By using the OOAD method, software developers analyze, design and program for one typical application, and abstract the framework from the development of this typical application. Mature frameworks can be reused as the basis for many other applications.

My major report, as outlined in this paper, will develop a framework for drawing the architecture of buildings using OpenGL and VC++, and it will implement two sample applications based on the framework.

2. Background

2.1. Computer Graphics

The terminology of computer graphics can acquire different meanings in different contexts.

- Simply stated, computer graphics are pictures generated by computers.
- Computer graphics also refer to the computer tools used to generate such pictures, including both hardware and software tools.
  - Hardware tools include video monitors and printers that display graphics, as well as input devices, such as a mouse, that enable users to point to items and draw figures
  - As for software tools for graphics, there must be a collection of graphics routines that produce the pictures themselves. For example, all graphics libraries have functions to draw a simple line or circle. Some go beyond this, containing functions to draw and manage windows with pull-down menus and dialog boxes.
- Computer graphics may also refer the discipline of the study related to the depiction of pictures by using computers.

2.2. Open GL

Not too long ago, programmers were compelled to use highly device-dependent libraries designed for use on one specific computer system with one specific type of display device. This made it very difficult to port a program to another system or to use the program with another device: usually the programmer had to make substantial changes to the program to get it to work, and the process itself was time consuming and highly prone to errors. However, device-independent graphics libraries are now available, which allow the programmer to use a common set of functions within an application and to run the same application on a variety of systems and displays. Open GL constitutes such a library, and the Open GL way of creating graphics is widely used in industry and at universities.

2.3. Framework

As a domain of engineering, software engineering strives to establish standard, well understood building blocks, which are expected to be developed and stored in libraries for common use and considered as a basis for extensions and modifications in the spirit of the reusability concept. Although software reuse had existed in software development processes since software engineering began as a research field in 1969, it was not until 1978 that the concept of reusability was clear in the minds of people as a solution to the software crisis. Currently, the issue of software reuse is getting a great deal of attention in regard to its potentiality in increasing productivity, reducing costs, and improving software quality.

The advent of object-oriented techniques makes the reuse technology even more powerful. Framework, as an object-oriented reuse technique, is playing an increasingly important role in contemporary software development. Although people are developing and utilizing frameworks, the definitions of frameworks vary. Johnson [9] lists two common definitions of frameworks as follows:

- A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact.
- A framework is a skeleton of an application that can be customized by an application developer.

Although it is hard to define frameworks formally and clearly, there are several common features of frameworks:

- A framework is for reuse in the development of application in some specific problem domain.
- A framework is a skeleton program that can be extended into a complete application by being filled with necessary components at hot spots, the implementations of which vary from application to application of the framework.
- A framework is customizable to fit the applications being developed.

- A framework predefines some interfaces of the filling components as the framework contract so that the components fit each other and fit into the framework.
- A set of abstract classes is usually used to describe the behaviors of frameworks.
- A corresponding concrete class should be implemented as the filling component when extending the framework to application.

Given these features, we can see that framework is a design reuse technique for developing new application. There are a lot of advantages in using frameworks in application development. Rather than developing an application from scratch, extending and customizing an appropriate framework to cater to the needs of the application will save much effort and therefore speed up the development process. Another important advantage is that the quality guarantee of the framework as a product also assures the quality of its application.

Fayad-Schmidt [10] categorizes the existing frameworks into three scopes: system infrastructure framework, middle ware integration frameworks, and enterprise application frameworks. Frameworks can also be classified as white box and black box frameworks by the techniques used to extend them.

White box frameworks are extended to applications by inheriting from some base classes in the frameworks and overriding their interface methods, whereas black box frameworks are extended by customizing new components according to particular interfaces defined by the frameworks and plugging them into frameworks.

However, the white box and black box reuse techniques frequently coexist in one framework, because some common abstractions among many black box components (object classes) within the same problem domain may also be expressed and implemented using inheritance and overriding.

2.4. A Sample Framework for Graphics Recognition Applications

The framework for graphics recognition applications is an enterprise application

framework in the problem domain of graphics recognition. The recognition of graphic objects from files of scanned paper drawings, known as graphics recognition, which is a part of engineering drawing interpretation and document analysis and recognition (DAR) – is a topic of increasing interest in the field of pattern recognition and computer vision.

Although many algorithms and systems have been developed, the results have not been satisfactory due to the complex syntax and semantics these drawings convey, and more powerful algorithms and system are therefore strongly needed. To reduce the effort involved in developing basic algorithms for such systems, the graphics recognition framework was developed. After comprehensive domain analysis, the design of the graphics recognition framework was inspired by the observation that all classes of graphic objects consist of several or many primitive components, and the algorithms for recognizing them can employ an incremental, stepwise recovery of the primitive components.

The framework is implemented using a C++ template class with the graphic class used as a parameter. It is a black box framework, whose application can be easily assimilated to the framework contract. Some common graphic classes are also defined in a preliminary inheritance hierarchy so that new concrete graphic classes can also inherit from some appropriate class in the hierarchy.

The framework described above has been successfully used in developing the application of arc detection, leader and dimension set detection, and general line detection.

3. Design and Implementation of a Framework for Drawing the Architecture of Buildings

3.1. Polygonal Mesh

A polygonal mesh is a collection of polygons along with a normal vector associated with each vertex of each polygon.

Figure-1 illustrates a simple shape which can be called a barn. It has seven polygonal faces and a total of 10 vertices each of which is shared by three faces. Because the barn is associated with flat walls, there are only seven distinct normal vectors involved: the normal to each face.

Figure- 1 a sample polygonal mesh

3.1.1 Data Structure of Polygonal Mesh

The Polygonal Mesh itself plays a role in the drawing of a building, and it is also the building block of other shapes, such as extruded shapes, shapes of revolution, and shapes of revolution based on Bezier Curves. Therefore it is important to make the data structure for a Polygonal Mesh efficient and to let application developers know the structure clearly.

The simplest way to represent the data structure of the polygonal mesh is to use a list to describe all the faces, and each face consists of a list wherein its vertices are located and where the normal for each of its vertices is pointing. This kind of data structure for the barn is shown in figure-2. There are a total of 30 vertices and 30 normals. But this structure would be quite redundant, and its chief defect concerns the error of inconsistency resulting from redundancy. As in the case of barn, there are only 10 distinct vertices and seven distinct normals. The error of inconsistency is possible in that different values may be given for the same vertex shared by different faces.

Faces	Vertex	Vertex	Vertex	Vertex	Vertex
0	(0,0,0)	(0,0,1)	(0,1,1)	(0,1,0)	Null
1	(.5,1.5,0)	(0,1,0)	(0,1,1)	(.5,1.5,1)	Null
2	(1,1,0)	(.5,1.5,0)	(.5,1.5,1)	(1,1,1)	Null
3	(1,0,0)	(1,1,0)	(1,1,1)	(1,0,1)	Null
4	(0,0,0)	(1,0,0)	(1,0,1)	(0,0,1)	Null
5	(0,0,1)	(1,0,1)	(1,1,1)	(.5,1.5,1)	(0,1,1)
6	(0,0,0)	(0,1,0)	(.5,1.5,0)	(1,1,0)	(1,0,0)

Figure- 2 Face list available through using the simplest data structure

An improved way would be to use another two lists to store the vertices and normals of the polygonal mesh, and only indices of vertices and indices of normals would be used in the list of each face. The redundancy and thus the error of inconsistency can thus be avoided.

- A vertex list contains locational or geometric information.
- A normal list contains orientation information.
- A face list contains connectivity or topological information.

The face list simply indexes into the other two lists. Figure-3 shows the corresponding data structures used in the program.

List for vertices

Vertex	(x.y.z)
0	(0.0.0)
1	(1.0.0)
2	(1.1.0)
3	(.5.1.5.0)
4	(0.1.0)
5	(0.0.1)
6	(1.0.1)
7	(1.1.1)
8	(.5.1.5.1)
9	(0.1.1)

List for normals

normal	(Nx.Ny.Nz)
0	(-1.0.0)
1	(-0.7.0.7.0)
2	(0.7.0.7.0)
3	(1.0.0)
4	(0.-1.0)
5	(0.0.1)
6	(0.0.-1)

List for faces indexing into the other two lists

face	vertices	normal
0	0.5.9.4	0.0.0.0
1	3.4.9.8	1.1.1.1
2	2.3.8.7	2.2.2.2
3	1.2.7.6	3.3.3.3
4	0.1.6.5	4.4.4.4
5	5.6.7.8.9	5.5.5.5.5
6	0.4.3.2.1	6.6.6.6.6

Figure- 3 Data structure to describe a polygonal mesh without redundancy

3.1.1 The class diagram for a Polygonal Mesh

The following is the declaration of the class Mesh, along with those two other classes: VertexID and Face, which are used in the Mesh class. Point3 is declared as a structure, because it will be used not only in the Mesh class but also here and there in the application program. A mesh object has a vertex list, a normal list, and a face list, represented simply by the arrays: pt, norm and face, respectively.

These arrays are allocated dynamically at run time, after it is known how large they are. Their lengths are stored in numVerts, numNormals, and numFaces, respectively.

The face data type is an array of the faces associated with the Mesh, and each face is composed of an array of a list of vertices and the normal vectors, which is represented by an array of Class VertexID. It is organized as an array of index pairs: the v-th vertex in the f-th face has position pt[face[f].vert[v].vertIndex] and normal vector norm[face[f].vert[v].normIndex].

#define mesh

```
const float PI = 3.1415926f;
struct point3
{
    double x;
    double y;
    double z;
};
struct vector3
{
    double x;
    double y;
    double z;
};
typedef point3 Point3;
typedef vector3 Vector3;
class VertexID
{
public:
    int vertIndex;
    int normIndex;
};
class Face
{
public:
    int nVerts;
    VertexID * vert;
    Color  color;
    Vector3 averageNormal;
    Face();
    ~Face();
};
class Mesh
{
protected:
    int numVerts;
    Point3* pt;
    int numNormals;
    Vector3* norm;
    int numFaces;
    Face* face;
public:
```

```
    Mesh();
    ~Mesh();
    void draw();
    void computeNormal();
};
```

3.1.2 The implementation of Mesh::draw()

The first method is to draw such a mesh object after the structure has been designed. It is a matter of drawing each of its faces, so an Mesh::draw() will tranverse the array of faces in the mesh object and, for each face, send the list of vertices and their normals down the graphics pipeline. So the basic flow of Mesh::draw is as follows:

```
void Mesh::draw()
{
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    for (int f = 0; f < numFaces; f++)
    {
        glBegin(GL_POLYGON);
        for (int v = 0; v < face[f].nVerts; v++)
        {
            int iv = face[f].vert[v].vertIndex;
            glVertex3d(pt[iv].x, pt[iv].y, pt[iv].z);
        }
        glEnd();
    }
}
```

3.2. Extruded shapes

Many kinds of shapes can be generated by extruding, or sweeping, a 2D shape through space. The prism shown in Figure-5 is an example of a shape produced by linear sweeping.

Figure- 4 2D polygon on X-Y plane Figure- 5 2D polygon swept along Z

3.2.1 Creating Prisms

A prism is the shape formed by sweeping a polygon in a straight line. Figure-4 shows a prism based on a polygon P lying in the xy-plane. P is swept through a distance H along the z-axis, forming the prism shown in Figure-5. As P is swept along, an edge in the z-direction is created for each vertex of P. This generates another 6 vertices, so the prism has 12 vertices in all. They appear in pairs: if (Xi, Yi, 0) is one of the vertices of the base P, then the prism also contains the vertex (Xi, Yi, H).

More generally, the sweep could be along any vector d.

3.2.1.1 Data Structure of Prism

A prism is actually a polygonal mesh in which one bottom face is the base polygon, another face is the top face, and other faces are side faces which are actually quadrilaterals each of which is composed of two consecutive pairs of vertices: (Xi, Yi, 0), (Xi, Yi, H), (Xi+1, Yi+1, H), and (Xi+1, Yi+1, 0).

3.2.1.2 Class Diagram of Prism

Since a prism can be gotten by sweeping along any vector d. and such a vector is usually not known until an application program is developed. and even in the same application different prisms will be generated by sweeping along different vectors. it is important that the application developers possess the flexibility to set such vectors in their programs.

In accordance with the above ideas and requirements. we can design a new class inheriting from the Class of Mesh by merely adding three methods to draw prism and application developers have the desired flexibility to set the vector along which the prism is swept. One method would be Prism::drawPrism(...). and another two comprise virtual method Prism::setVector(...) and Prism::setFunctionForPrism(...).

The vector d along which the prism is generated can be set by two virtual methods: Prism::setVector(...) and Prism::setFunctionForPrism(). In this way. application developers can set any vector d by overriding the virtual function setVector(...) and setFunctionForPrism(...) in their new sub class inheriting from the class Prism. This is the reason why we add virtual before the declaration of these two methods. so that they may be then overridden in the subclass of the class Prism.

The virtual keyword before "public Mesh" is used to tell the C++ compiler that the base class Mesh may be indirectly inherited multiple times and that it should be included only a single instance of the base class.

```
#ifndef mesh
#define mesh
#include "mesh.h"
#endif
class Prism : virtual public Mesh
{
public:
    int drawPrism(Point3 p[], int numPts, int numSlices, double length, char xyz);
virtual void setFunctionForPrism(Point3 p1, Point3& p2, Vector3 v, double alpha);
virtual void setVector(Vector3& v, char xyz);
};
```

3.2.1.3 The implementation of drawing a prism: Prism::drawPrism(...)

In order to draw the prism, the base polygon must be passed to the method by parameter p[] and numPts, while p[] contains the vertices composing the polygon and numPts is the number of vertices of the polygon. Parameter numSlices, length and d will prescribe the shape of the prism. Parameter numSlices will be used to determine how many segments the prism will have, and parameter length will be used to describe how long the prism will be, while the vector d will be the direction along which the prism is swept.

Suppose we have m vertices of a base polygon and have n segments, and if (Xi,Yi,0) is one of the vertices at the bottom, then its pair is the vertex (Xi,Yi,H). while the consecutive pair are (Xi+1, Yi+1, 0) and (Xi+1,Yi+1,H).

The prism is composed of segments, and each segment is composed of a bottom face, a top face, and side faces, each of which is composed of two consecutive pairs of vertices: (Xi, Yi, 0), (Xi, Yi, H), (Xi+1, Yi+1, H), and (Xi+1, Yi+1, 0). Thus the prism consists of faces composed of vertices and generated vertices.

We can also determine that there will be (m-1) faces for each segment and there will be a total of ((m-1)*n +2) faces in the prism, so that the total number of vertices is m*(n+1). After determining the infrastructure of the prism and setting the corresponding variables and lists (arrays) in the class of Mesh, the Mesh::draw() will be called to draw the whole prism as a polygon mesh.

Because we allocate memory dynamically for arrays of face, norm, pt, etc., we have to release the dynamic memory after the polygonal mesh has been drawn, otherwise the memory will run out very easily due to the fact that there may be hundreds or thousands of faces in the application and the display() is called very frequently each time the window is refreshed or redrawn.

```
#include <afxwin.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <fstream.h>
#include <math.h>
```

```
#include "prism.h"
int Prism::drawPrism(Point3 p[], int numPts, int numSlices, double length, char xyz)
{
    double alpha;
    int whichFace;
    Vector3 v;
    numVerts = numPts * (numSlices + 1);
    numNormals = numVerts;
    numFaces = (numPts) * (numSlices);
    pt = new Point3[numVerts];
    face = new Face[numFaces];
    norm = new Vector3[numNormals];
    alpha = length / (numSlices);
    setVector(v,xyz);
    for (int i = 0; i < numSlices + 1 ; i++)
    {
        for (int j = 0; j < numPts; j++)
        {
        setFunctionForPrism(p[j].pt[i * numPts + j],v,alpha*i);
        }
        for (j = 0; j < numPts; j++)
        {
            if (i < numSlices )
            {
                whichFace = (i) * (numPts) + j;
                face[whichFace].nVerts = 4;
                face[whichFace].vert = new VertexID[4];
                face[whichFace].vert[0].vertIndex = (i) * (numPts) + j;
            face[whichFace].vert[1].vertIndex = (i) * (numPts) + (j + 1) % numPts;
                face[whichFace].vert[2].vertIndex = (i+1) * (numPts) +
                                                    (j + 1) % numPts;
                face[whichFace].vert[3].vertIndex = (i+1) * (numPts) + j;
            }
        }
    }
    draw();
    delete [] pt;
    delete [] face;
    delete [] norm;
    return 0;
}
```

In setFunctionForPrism(...), p1 stores points of the bottom face, p2 stores the points of the top face which is generated by sweeping p1 along the vector d for a length of len, and p2 is passed by reference as it has to change the actual parameter.

```
void Prism::setFunctionForPrism(Point3 p1, Point3& p2, Vector3 v, double alpha)
{
    p2.x = p1.x + alpha * v.x;
    p2.y = p1.y + alpha * v.y;
    p2.z = p1.z + alpha * v.z;
}
```

The setVector(Vector3& d) in the base class is virtual, so that application developers can override the method by defining their own special vector d, and the prism can then be along any vector they require.

```
void Prism::setVector(Vector3& v, char xyz)
{
  if (xyz=='x' || xyz == 'X')
   {
    v.x = -1;
    v.y = 0;
    v.z =0;
   }
  if (xyz=='y' || xyz =='Y')
   {
        v.x = 0;
        v.y = -1;
        v.z =0;
   }
   if (xyz=='z' || xyz == 'Z')
   {
        v.x =0;
        v.y =0;
        v.z = -1;
   }
}
```

3.2.2 Building segmented extrusions: Tubes or Pipes based on 3D curves

In section 3.2.1 the base polygon is swept linearly in order to get a prism. What

happens if we expand the idea of sweeping by allowing the base polygon to sweep non-linearly. For example, a base polygon can be modeled by employing a sequence of extrusions, each with its own transformation, and thereby laying them end to end to form a tube or pipe. The various transformed polygons are called the waist of the tube.

It is much easier to think of the tube as polygons wrapped around a curve, which is called the spine of the tube, that undulates through space in some organized fashion. The curve can be represented parametrically. For example, the helix shown in Figure-6 has the parametric representation:

$C(t) = (\cos(t), \sin(t), bt)$, for some constant b

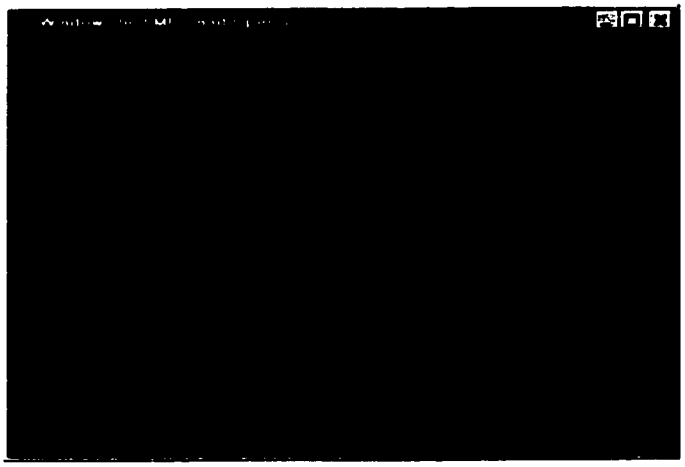

Figure- 6 a 3D helix

3.2.2.1 How to generate each waist of the TubePipe

What kind of transformation should be implemented to get each waist polygon from the base polygon. To form the various waist polygons of the tube, we sample C(t) at a set of t-values, $\{t_0, t_1, t_2 \ldots\}$, and build a transformed polygon in the plane perpendicular to the curve at each point $C(t_i)$, as shown in Figure-7 and Figure-8. It is convenient to think of erecting a local coordinate system at each chosen point along the spine: the local "z-axis" points along the curve, and the local "x- and y- axes" point in directions, both normal to the

z-axis and normal to each other. The waist polygon is set to lie in the local xy-plane.

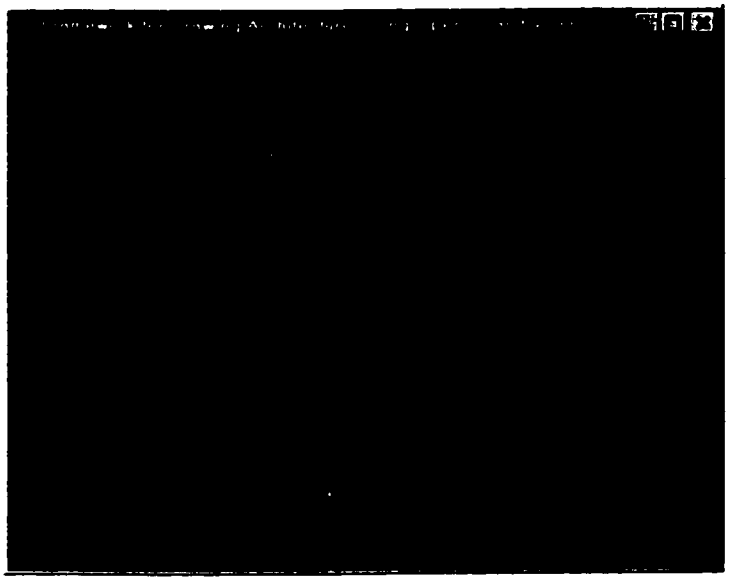

Figure- 7 a quadrilateral wrapping around a helix

Figure- 8 How a point moves along a curve

At each value t_i of interest, a vector $T(t_i)$ that is tangential to the curve can be computed. Consequently, two vectors, $N(t_i)$ and $B(t_i)$, which are perpendicular to $T(t_i)$ and to each other, can be computed. These three vectors constitute the coordinate frame at

t[i].

Once the new coordinate frame is computed, it is easy to find the transformation matrix M that transforms the base polygon of the tube to its position and orientation in that frame. It is the transformation matrix that carries the world coordinate system into the new coordinate frame. So the matrix M_i must carry X, Y and Z into $N(t_i)$, $B(t_i)$ and $C(t_i)$, respectively, and must carry the origin of the world into the spine point $C(t_i)$. Thus, the matrix has columns consisting directly of $N(t_i)$, $B(t_i)$, $T(t_i)$ and $C(t_i)$ that are expressed in homogeneous coordinates:

$$M_i = (N(t_i) \mid B(t_i) \mid T(t_i) \mid C(t_i))$$

$$M_i = \begin{vmatrix} N(t).x & B(t).x & T(t).x & C(t).x \\ N(t).y & B(t).y & T(t).y & C(t).y \\ N(t).z & B(t).z & T(t).z & C(t).z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$C(t) = (\cos(t), \sin(t), bt)$, for some constant b

Let C1(t) represent the first derivative of C(t)

Let C2(t) represent the second derivative of C(t)

$T(t) = C1(t)$

$$B(t) = \frac{C1(t) \times C2(t)}{|C1(t) \times C2(t)|}$$

$$T(t) = (-\sin(t), \cos(t), b) / \sqrt{1+b^2}$$

$$B(t) = (b\sin(t), -b\cos(t), 1) / \sqrt{1+b^2}$$

$N(t) = B(t) \times T(t)$

$N(t) = (-\cos(t), -\sin(t), 0)$

$$M_i = \begin{vmatrix} -\cos(t) & b\sin(t)/\sqrt{1+b^2} & -\sin(t)/\sqrt{1+b^2} & \cos(t) \\ -\sin(t) & -b\cos(t)/\sqrt{1+b^2} & \cos(t)/\sqrt{1+b^2} & \sin(t) \\ 0 & 1/\sqrt{1+b^2} & b/\sqrt{1+b^2} & bt \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

3.2.2.2 Class Diagram of TubePipe

We can design a new class TubePipe to draw a tube and pipe. The class TubePipe inherits from the base class Mesh and we then add two methods and one data type into the new class. One method is TubePipe::drawTubePipe(...) and another is TubePipe::setMatrix(...). The new data type is Matrix which is used to represent the transformation matrix by which the waist polygon can be gotten with the Transformation Matrix multiplied by the base polygon.

The application class may multi-inherits from Prsim and TubePipe, both of which inherit from the same base class Mesh, so that the base class may be indirectly inherited multiple times. In such a case, the base class Mesh has to be declared virtual in the place where it is inherited, so that only one instance of the base class Mesh is included; otherwise the base class Mesh will be included multiple times and yield a compile error of ambiguity. The same is true for Class Prism: virtual public Mesh.

Application developers can make the base polygon wrap about other kinds of 3D curves with formulae by changing the content of the transformation matrix M. However, according to the OOAD principle of information hiding and encapsulation, the apllication developers may not change anything in the class TubePipe. They can elaborate a new class inheriting from class TubePipe and write their own special setMatrix(...) in the sub class of TubePipe to override the method in the parent class. Therefore we add virtual before setMatrix(...) in the class of TubePipe to make such overriding possible.

```
#ifndef mesh
#define mesh
#include "mesh.h"
```

```
#endif
typedef double Matrix[4][4];
class TubePipe : virtual public Mesh
{
public:
    int  drawTubePipe(Point3 p[], int numPts, double t[], int numTimes);
    virtual void setMatrix(Matrix m, double t, double b);
};
```

3.2.2.3 The implementation of drawing TubePipe: TubePipe::drawTubePipe(...)

The main idea used for drawing TubePipe is similar to that used for drawing prism.

```
#include <afxwin.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <fstream.h>
#include <math.h>
#include "tube&pipe.h"

int TubePipe::drawTubePipe(Point3 p[], int numPts, double t[], int numTimes)
{
float b = 1.0;
int whichFace;
Matrix m;
numVerts = numPts * numTimes;
numNormals = numVerts;
numFaces = numPts * (numTimes-1);
pt = new Point3[numPts*numTimes];
norm = new Vector3[numPts*numTimes];
face = new Face[numPts*(numTimes-1)];
for (int i = 0; i < numPts; i++)
{
    pt[i].x = p[i].x;
    pt[i].y = p[i].y;
    pt[i].z = p[i].z;
}
for (i = 0; i < numTimes; i++)
{
    setMatrix(m, t[i], b);
    for ( int j = 0; j < numPts; j++)
    {
            pt[i*numPts + j].x = float((p[j].x)*m[0][0] + (p[j].y)*m[0][1] +
```

```
                                        (p[j].z)*m[0][2]+ m[0][3]);
            pt[i*numPts + j].y = float((p[j].x)*m[1][0] + (p[j].y)*m[1][1] +
                                        (p[j].z)*m[1][2]+ m[1][3]);
            pt[i*numPts + j].z = float((p[j].x)*m[2][0] + (p[j].y)*m[2][1] +
                                        (p[j].z)*m[2][2]+ m[2][3]);
        }
    for ( j = 0 ; j < numPts; j++)
    {
        if ( i > 0)
        {
        whichFace = (i-1)*numPts + (j);
        face[whichFace].nVerts = 4;
        face[whichFace].vert = new VertexID[4];
        face[whichFace].vert[0].vertIndex = (i-1) * numPts + j;
        face[whichFace].vert[1].vertIndex = (i-1) * numPts + (j + 1) % numPts;
        face[whichFace].vert[2].vertIndex = (i-1) * numPts + (j + 1) % numPts +
                                            numPts;
        face[whichFace].vert[3].vertIndex = (i-1) * numPts + j +numPts;
        }
    }
}
draw();
delete [] pt;
delete [] face;
delete [] norm;
return 0;
}
```

The transformation matrix can be set in the method of setMatrix(...) by the application developers according to the kind of spine they have chosen for their application. The default matrix is for helix. Application developers have to decide all the elements of the transformation matrix if the spine is not helix.

```
void TubePipe::setMatrix(Matrix m, double t, double b)
{
    m[0][0] = -cos(t);
    m[1][0] = -sin(t);
    m[2][0] = 0;
    m[3][0] = 0;
```

```
    m[0][1] = b*sin(t)/sqrt(1+b*b);
    m[1][1] = -b*cos(t)/sqrt(1+b*b);
    m[2][1] = 1/sqrt(1+b*b);
    m[3][1] = 0;
    m[0][2] = -sin(t)/sqrt(1+b*b);
    m[1][2] = cos(t) / sqrt(1+b*b);
    m[2][2] = b/sqrt(1+b*b);
    m[3][2] = 0;
    m[0][3] = 10*cos(t);
    m[1][3] = 10*sin(t);
    m[2][3] = b*t;
    m[3][3] = 1;
}
```

3.3. Shapes of Revolution

In the building domain, there are many shapes for which we may not find suitable parametric formulas, but they are frequently used for classical buildings, such as Renaissance-style domes (a famous one in Montreal is the dome of the Marche Bonsecours (Figure-9)), neoclassical domes (such as the dome of the Bank of Montreal at Notre-Dame across Place d'Armes), Indian style domes (a famous one is the dome of the Taj Mahal), Islamic style domes, and so on. However, most of them can be considered as a curve rotated around an axis. For such curves, we can not find parametric formulae, but we can use a profile consisting of a collection of discrete yet important data points along the curve to describe the curve, and then rotate the curve consecutively around a certain axis to form the dome we want. In this way, complex shapes of revolution can be easily drawn.

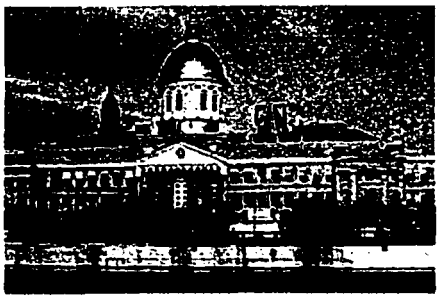

Figure- 9 Marche Bonsecours with its Renaissance-style dome

3.3.1 How to generate the shape of revolution

We only consider the case of the curve rotating around y-axis because we can transpose the former one in any location and orientation by implementing transformation afterwards.

Suppose the base curve is based on a set of Points Pi(xi, yi, 0), and the curve is rotated about y-axis consecutively, and there are a total number of K-1 generated curves equally spaced at angle θ about y-axis. We call the polygonal mesh between Ki-1 curve and Ki curve a slice of the shape of revolution, and call the base curve and generated curve the waist of the shape of revolution. There are k waists and k-1 slices.

θ = angle / (k-1), the angle is the degree of the revolution from 0 to angle.

So the basic transformation Matrix Mi of rotation about y-axis can be used to determine the generated vertices from the vertices of the base curve, since the base curve is used to obtain the generated vertices for every waist, the rotation angle should be i*θ

$$Mi = \begin{vmatrix} \cos(i\theta) & 0 & \sin(i\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(i\theta) & 0 & \cos(i\theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

3.3.2 The class diagram of the shape of revolution

We can design a new class Revolution to draw shapes of revolution. Class Revolution inherits from the base class Mesh, and we can add two methods and one data type into the new class. One method is Revolution::drawRevolution(...) and another is Revolution::getProfile(...).

Application developers can override getProfile(...) by writing their own version of getProfile(...) in their inheriting sub class. This is why we add virtual before the declaration of the method. The base curve can be described by sampling the discrete yet important data points along the curve, and getProfile(...) can be designed to ask user to use the mouse to input the points or to store the coordinates of points in a file and to let getProfile(...) read

from the designated file.

```
#ifndef mesh
#define mesh
#include "mesh.h"
#endif
typedef double Matrix[4][4];
class Revolution : virtual public Mesh
{
public:
    int  drawRevolution(Point3* p, Point3 cPoints[].int numPts, int numSlices,
                                    double angle);
    virtual void getProfile(Point3*& p, Point3 cPoints[], int& numPts);
};
```

3.3.3 The implementation of drawing the shape of revolution

The main idea of drawing the shape of revolution is consequently the same as that of drawing a prism and a TubePipe.

```
#include <afxwin.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <fstream.h>
#include <math.h>
#include "revolution.h"
int Revolution::drawRevolution(Point3* p, Point3 cPoints[], int numPts,
                                    int numSlices, double angle)
{
double alpha;
int whichFace;
numVerts = numPts * (numSlices);
numNormals = numVerts;
numFaces = (numPts - 1) * (numSlices);
pt = new Point3[numVerts];
face = new Face[numFaces];
norm = new Vector3[numNormals];
alpha = (angle) / (numSlices-1);
getProfile(p, cPoints, numPts);
for (int i = 0; i < numSlices; i++)
{
    for (int j = 0; j < numPts; j++)
    {
```

```
            pt[i * numPts + j].x = (p[j].x) * cos(alpha * i) + (p[j].z) * sin(alpha * i);
            pt[i * numPts + j].y = p[j].y;
            pt[i * numPts + j].z = -(p[j].x) * sin(alpha * i) + (p[j].z) * cos(alpha * i);
        }
        for (j = 0; j < numPts; j++)
        {
            if (j < numPts - 1)
            {
                whichFace = (i) * (numPts - 1) + j;
                face[whichFace].nVerts = 4;
                face[whichFace].vert = new VertexID[4];
                face[whichFace].vert[0].vertIndex = (i) * (numPts) + j;
                face[whichFace].vert[1].vertIndex = (i) * (numPts) + (j + 1);
                if ( i == numSlices -1 && (2*PI - angle) >= 0.1)
                {
                    face[whichFace].vert[2].vertIndex = (i) * (numPts) + (j + 1);
                    face[whichFace].vert[3].vertIndex = (i) * (numPts) + j;
                }
                else
                {
                    face[whichFace].vert[2].vertIndex = ((i+1) % numSlices) *
                                                        (numPts) + (j + 1);
                    face[whichFace].vert[3].vertIndex = ((i+1) % numSlices) *
                                                        (numPts) + j;
                }
            }
        }
    }
    draw();
    delete [] p;
    delete [] pt;
    delete [] face;
    delete [] norm;
    return 0;
    }
```

It is possible that the number of points of the curve is derived from getProfile, while its new value has to be returned to drawRevolution(...), and this is why parametric numPts is declared to be passed by reference; it is also the reason why p is declared as a dynamic

array. Since new operation may change the value of p. and we must pass its new value back. pointer p is also declared to be passed by reference.

```
void Revolution::getProfile(Point3*& p. Point3 cPoints[].int& numPts)
{
p = new Point3[numPts];
for (int i = 0; i < numPts; i++)
{
    p[i].x = cPoints[i].x;
    p[i].y = cPoints[i].y;
    p[i].z = cPoints[i].z;
}
}
```

3.4. 3D Surface with formulae

In section 3.3. we outlined how to draw shapes of revolution without formulae. we use a profile to record the important points to simulate the curve in 2D dimension. then rotate the 2D curve to get the 3D shape. However. sometimes we know the formulae for the 2D curve. and we can also determine the formulae for its 3D surface after revolution or through other kinds of movement if and only if we also know the formulae of the movement. In such cases. we can draw 3D surface according to its 3D formulae. Theoretically speaking. we can draw any 3D surface if and only if we know its formulae. This kind of drawing is therefore much smoother and much more realistic than the method described in section 3.3. It is more suitable to draw pillars or columns. such as corinthian. ionic and doric structures. which are used for ancient Greek style modes of architecture. Pillars are also popularly used in other styles of building. Examples in Montreal include the Bank of Montreal building (1847)(Figure-10) across the Place d'Armes offering an ornate facade that boasts six Corinthian columns with pediment sculptures.

Figure- 10 Bank of Montreal with six Corinthian columns and a neoclassical dome

3.4.1 How to generate 3D surface with formulae

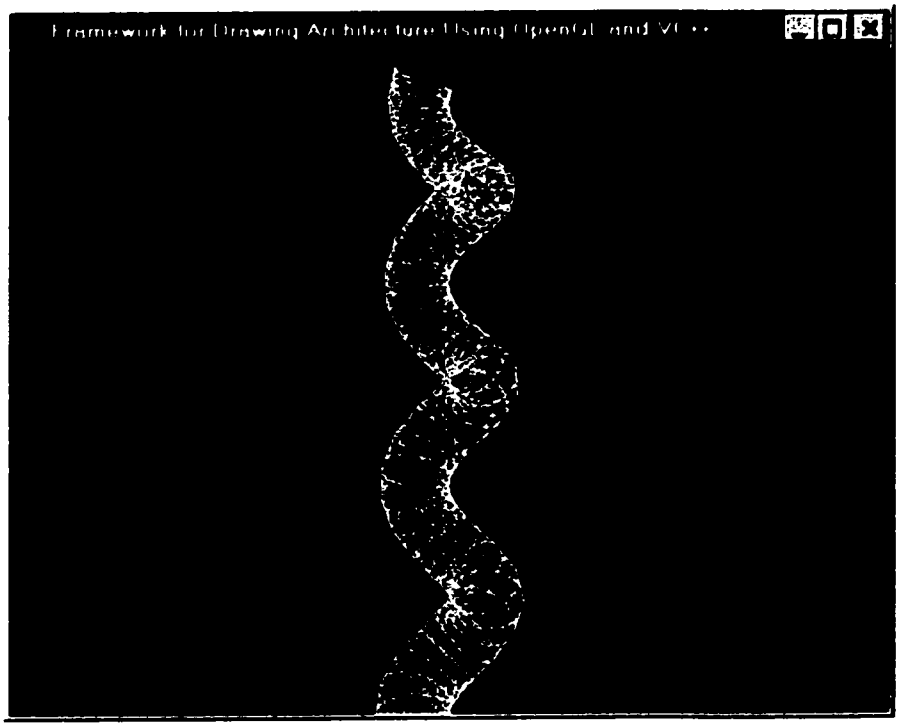

Figure- 11 a Circle wrapping around a helix

Suppose the parametric form for the surface is given by P(u.v) = (X(u.v). Y(u.v). z(u.v)). We can subdivide the whole surface into small patches. supposing. for instance. that there are m pieces equally spaced in the direction of u between uMin and uMax. and there are n pieces equally spaced in the direction of v between vMin and vMax. Consequently. the whole surface is composed of a face list consisting of m×n patches which are actually quadrilaterals. If we send each quadrilateral to the OpenGL pipe line to draw each patch one by one. then the whole surface can be drawn.

For example:

2D curve of circle can be represented by (cos(u). sin(u). 0)

if the circle wraps about the helix C(v) = (cos(v). sin(v). bv) . b is constant.

then the parametric formula for the 3D surface is

P(u. v) = C(v) + cos(u) N(v) + sin(u) B(v)

N(v) = (-cos(v). -sin(v). 0)

$B(v) = (b\sin(v). -b\cos(v). 1)/\sqrt{1+b^2}$

Therefore any point P on the surface can be determined by:

$P.x = \cos(v) + (\cos(u)(-\cos(v)) + \sin(u)b\sin(v)/\sqrt{1+b^2}$

$P.y = \sin(v) + \cos(u)(-\sin(v)) + \sin(u)(-b\cos(v))/\sqrt{1+b^2}$

$P.z = bv + \cos(u)(0) + \sin(u)/\sqrt{1+b^2}$

3.4.2 The class diagram of 3D surface with formulae

We can design a new class CurveSurface inheriting from class Mesh to draw 3D curved surface. There are two methods employed in the class. one of which is CurveSurface::drawCurveSurface() which is called by application program to draw 3D surface.

Another one is CurveSurface::setFunctionForCurveSurface(...) designed to enable the application developers to obtain the necessary flexibility to set different 3D surfaces with different formulas by writing their special setFunctionForCurveSurface(...) in their inheriting class so as to override this function. so this method has to be declared as virtual.

Because setFunctionForCurveSurface(...) will change the value of variable Point3 p and this value has to be returned to the caller. the parameter p has to be declared to be passed by reference.

```
#ifndef mesh
#define mesh
#include "mesh.h"
#endif
class CurveSurface : virtual public Mesh
{
public:
    int  drawCurveSurface(int numValsU, int numValsv, float uMin, float uMax,
                          float vMin, float vMax);
    virtual void setFunctionForCurveSurface(Point3& p, double u, double v);
};
```

3.4.3 The implementation of drawing 3D surface with formulae

Since we subdivide the whole 3D surface into numValsU × numValsV patches which are actually quadrilaterals. The main goal of the function of drawCurveSurface(...) is to put all the vertices into the vertices list. describe each quadrilateral along with its proper vertices index, put each of these quadrilaterals into the face list. then call method draw() in the parent class Mesh to draw all the patches or quadrilaterals. so that. in this way. the whole 3D surface is drawn. The degree of smoothness and reality can be gotten by changing the value of numValsU and numValsV. wherein the larger values result in a higher degree of smoothness and reality.

uMin. uMax and vMin.vMax are used to determine which part of the 3D surface is drawn.

```
#include <afxwin.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <fstream.h>
#include <math.h>
#include "curveSurface.h"
int CurveSurface::drawCurveSurface(int numValsU, int numValsV, float uMin,float
uMax, float vMin, float vMax)
{
```

```
int i, j;
double u, v;
double delU = (uMax - uMin) / (numValsU -1);
double delV = (vMax - vMin) / (numValsV -1);
numVerts = numValsU * numValsV + 1;
numFaces = (numValsU - 1) * (numValsV - 1);
numNormals = numVerts;
pt = new Point3[numVerts];
face = new Face[numFaces];
norm = new Vector3[numNormals];
for (i = 0, u = uMin; i < numValsU; i++, u += delU)
    for (j = 0, v = vMin; j < numValsV; j++, v += delV)
    {
    int whichVert = i * numValsV + j;
        setFunctionForCurveSurface(pt[whichVert], u, v);
            if (i > 0 && j >0)
            {
                int whichFace = (i-1) * (numValsV - 1) + (j-1);
                face[whichFace].vert = new VertexID[4];
                face[whichFace].nVerts = 4;
                face[whichFace].vert[0].vertIndex = whichVert;
                face[whichFace].vert[1].vertIndex = whichVert - 1;
                face[whichFace].vert[2].vertIndex = whichVert - numValsV -1;
                face[whichFace].vert[3].vertIndex = whichVert - numValsV;
            }
    }
    draw();
    delete [] pt;
    delete [] face;
    delete [] norm;
    return 0;
}
```

The default formulae of setFunctionForCurveSurface is:

$P(u,v) = C(v) + \cos(u)\, N(v) + \sin(u)\, B(v)$

which is a circle $(\cos(u),\sin(u),0)$ wrapping about a helix :

$C(v) = (\cos(v), \sin(v), bv)$

$N(v) = (-\cos(v), -\sin(v), 0)$

$B(v) = (b\sin(v),-b\cos(v),1)/\sqrt{1+b^2}$

$P.x = \cos(v) + \cos(u)(-\cos(v)) + \sin(u)(b\sin(v))/\sqrt{1+b^2}$

$$P.y = \sin(v) + \cos(u)(-\sin(v)) + \sin(u)(-b\cos(v))/\sqrt{1+b^2}$$

$$P.z = bv + \cos(u)(0) + \sin(u)(1)/\sqrt{1+b^2}$$

Application developers have to find the formulae if the 3D surface is not generated by wrapping a circle about a helix, then decide the elements for P(u,v), C(v), N(v) and B(v) accordingly, finally override the method setFunctionForCurveSurface(...).

```
void CurveSurface::setFunctionForCurveSurface(Point3& p, double u, double v)
{
float b;
    b = 1;
    p.x = cos(v) + cos(u)*(-cos(v)) + sin(u)*sin(v)*b/sqrt(1+b*b);
    p.y = sin(v) + cos(u)*(-sin(v)) + sin(u)*(-cos(v))*b/sqrt(1+b*b);
    p.z = b*v + cos(u)*0 + sin(u) * 1/sqrt(1+b*b);
}
```

3.5. 3D Surface based on Bezier Curves

In reality, it often happens that we can not find formulas for some natural and beautiful 3D shapes or surfaces, however we still want to render them while importing a high degree of smoothness and reality. What should we do for such kinds of surface? We need another method to deal with such kinds of surface.

Since Bezier Curves are widely used to simulate the real curves for which we can not find formulae, we can also use them to simulate real 3D surfaces. Theoretically speaking, we can render any 3D surface very well as long as we can set the control points properly.

Figure- 12 a Bezier curve rotating about y-axis

3.5.1 How to generate 3D surfaces based on Bezier Curves

Suppose one Bezier curve rotates about axis-Z, which means that a profile is swept about z-axis. Such a profile can be represented by:

$C(v) = (X(v), Z(v))$

The resulting surface has the parametric form:

$P(u,v) = (X(v)\cos(u), X(v)\sin(u), Z(v))$

Since the profile represents a Bezier curve, we can express the profile by the following formulae:

$$(X(v), Z(v)) = \sum_{k=0}^{3} (X_k , Z_k) B_k (v)$$

$(X_0 ,Z_0), (X_1 ,Z_1) , (X_2 ,Z_2), (X_3 ,Z_3)$ are control points of the curve

$B_0 (v) = (1-v)^3$

$B_1 (v) = 3(1-v)^2 v$

$B_2 (v) = 3(1-v)v^2$

$B_3 (v) = v^3$

Therefore, any point on the surface can be determined by:

$P.x = \cos(u) (X0(1-v)^3 + X1(3(1-v)^2 v) + X2(3(1-v)v^2) + X3(v^3))$

$P.y = \sin(u) (X0(1-v)^3 + X1(3(1-v)^2 v) + X2(3(1-v)v^2) + X3(v^3))$

$P.z =(Z0(1-v)^3 + Z1(3(1-v)^2 v) + Z2(3(1-v)v^2) + Z3(v^3))$

3.5.2 The class diagram of 3D surfaces based on Bezier Curves

According to the above analysis, we can design a new class to draw the 3D surfaces based on Bezier Curves. Application developers can write their own special method in their new class inheriting from the class BezierCurveSurface to override the virtual method setFunctionForBezierCurveSurface(...), so that they can obtain different degree of Bezier, B-Spline, Nurbs curves.

```
#ifndef mesh
#define mesh
#include "mesh.h"
#endif

class BezierCurveSurface : virtual public Mesh
{
public:
    int  drawBezierCurveSurface(Point3 cPoints[], int numPts, int numValsU,
            int numValsV, float uMin, float uMax, float vMin, float vMax);
    virtual void setFunctionForBezierCurveSurface(Point3& p, Point3 cPoints[],
            int numPts,double u, double v);
};
```

3.5.3 The implementation of drawing 3D surfaces based on Bezier Curves

The control points of the bezier curve are passed by the caller of the method drawBezierCurveSurface(...). and the control points are also passed to method setFunctionForBezierCurveSurface(...) to calculate the value of the point at (u, v).

The new value of the point p has to be returned to the caller, so it is passed by reference, and this is the reason why it is declared as "Point3& p".

```
#include <afxwin.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <fstream.h>
#include <math.h>
#include "bezierCurveSurface.h"
int BezierCurveSurface::drawBezierCurveSurface(Point3 cPoints[], int numPts,int
numValsU, int numValsV, float uMin, float uMax, float vMin, float vMax)
{
int i, j;
double u, v;
double delU = (uMax - uMin) / (numValsU -1);
double delV = (vMax - vMin) / (numValsV -1);
double b = 1.;
numVerts = numValsU * numValsV + 1;
numFaces = (numValsU - 1) * (numValsV - 1);
numNormals = numVerts;
pt = new Point3[numVerts];
face = new Face[numFaces];
```

```
norm = new Vector3[numNormals];
for (i = 0, u = uMin; i < numValsU; i++, u += delU)
    for (j = 0, v = vMin; j < numValsV; j++, v += delV)
    {
    int whichVert = i * numValsV + j;
      setFunctionForBezierCurveSurface(pt[whichVert],cPoints,numPts,u,v);
            if (i > 0 && j >0)
            {
                int whichFace = (i-1) * (numValsV - 1) + (j-1);
                face[whichFace].vert = new VertexID[4];
                face[whichFace].nVerts = 4;
                face[whichFace].vert[0].vertIndex = whichVert;
                face[whichFace].vert[1].vertIndex = whichVert - 1;
                face[whichFace].vert[2].vertIndex = whichVert - numValsV -1;
                face[whichFace].vert[3].vertIndex = whichVert - numValsV;
            }
    }
    draw();
    delete [] pt;
    delete [] face;
    delete [] norm;
    return 0;
}
```

The default 3D surface is generated by a Bezier Curve rotating about axis Y, and its formulae are listed as follows:

$P(u,v) = (X(v)*\cos(u), X(v)*\sin(u), Z(v))$

$C(v) = (X(v), Z(v))$

$$(X(v), Z(v)) = \sum_{k=0}^{3} (X[k], Z[k])\,(B[k](v))$$

P[0],P[1],P[2],P[3] are control points of the curve

$B_0(v) = (1-v)^3$

$B_1(v) = 3(1-v)^2 v$

$B_2(v) = 3(1-v)v^2$

$B_3(v) = v^3$

If 3D surface is not generated by a Bezier curve rotating about one of the axis, then

application developers have to deduce the proper formulae for P(u.v). X(v). Z(v) respectively according to what type of the curve rotating about which vector.

```
void BezierCurveSurface::setFunctionForBezierCurveSurface(Point3& p, Point3
cPoints[], int numPts, double u, double v)
{
p.x = cos(u)*((cPoints[0].x) * pow((1-v),3)+ (cPoints[1].x) * 3 * pow((1-v),2) * v +
        (cPoints[2].x) * 3 * (1-v) * v * v + (cPoints[3].x) * pow(v,3));
p.y = sin(u)*((cPoints[0].x) * pow((1-v),3)+ (cPoints[1].x) * 3 * pow((1-v),2) * v +
        (cPoints[2].x) * 3 * (1-v) * v * v + (cPoints[3].x) * pow(v,3));
p.z = (cPoints[0].z) * pow((1-v),3)+ (cPoints[1].z) * 3 * pow((1-v),2) * v +
        (cPoints[2].z) * 3 * (1-v) * v * v + (cPoints[3].z) * pow(v,3);
}
```

3.6. Auxiliary class CopyTools

3.6.1 The importance of class CopyTools

Symmetry is the most significant property of most forms of architecture, and the same kind of compartments may appear here and there in the same building, and so it is important for us to have tools to duplicate or copy the same kind of compartments into different locations. The methods employed in the class of CopyTools are used for such a purpose.

We can also use the OpenGL transformation function to arrive at the above purpose without using the methods in class CopyTools. In order to illustrate the chief differences, we can use methods in the class CopyTools for help in drawing Taj Mahal, while we do not use them at all for drawing Tian Tan.

- MirrorCopy(...) is used to get new points after known points mirrored about x-axis, y-axis or z-axis;
- linearCopy(...) is used to get new points after known points swept along x-axis, y-axis or z-axis;
- scaleCopy(...) is used to get new points after known points scaled in x, y, z dimension;
- rotateCopy(...) is used to get new points after known points rotated about x-axis, y-axis or z-axis.

3.6.2 The declaration of class CopyTools

Variable oldPts[] is used to store the known points. and variable newPts[] is used to store the new points after known points are copied in one way of linear. scale. rotate copy. Variable numPts is used to specify how many points there are in the arrays of oldPts[] and NewPts[].

In method mirrorCopy(...). variable xyz is used to specify which axis is used as the mirror axis.

In method linearCopy(...). variable dist is used to specify the length of movement. and variable xyz is used to specify which axis is emploied as the vector along which the point is moving.

In method scaleCopy(...). variables x. y. z are used to specify the scale factors in each direction of X. Y. Z.

In method rotateCopy(...). variable alpha is used to specify the angle of rotation. and variable xyz is used to specify the axis about which the point is rotated.

```
class CopyTools
{
public:
void mirrorCopy(Point3 oldPts[], Point3 newPts[], int numPts, double a, char xyz);
void linearCopy(Point3 oldPts[], Point3 newPts[], int numPts, double dist, char xyz);
void scaleCopy(Point3 oldPts[], Point3 newPts[], int numPts, double x,double y,
double z);
void rotateCopy(Point3 oldPts[], Point3 newPts[], int numPts, double alpha, char
xyz);
};
```

3.7. Class diagram of the framework for drawing architecture of buidling

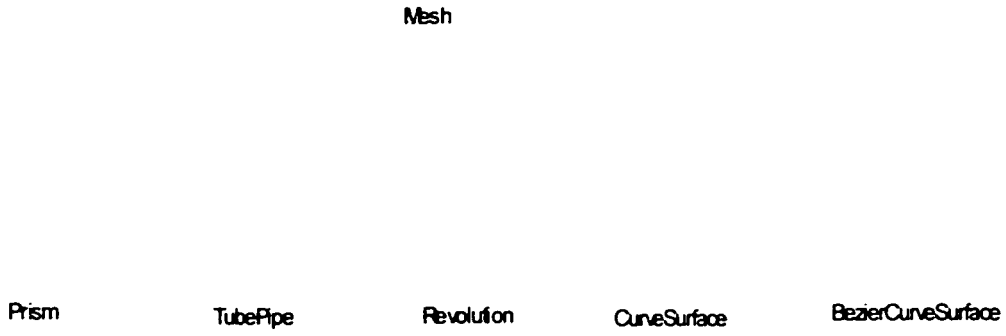

Figure- 13 Class diagram of the classes in the framework

4. Applications

4.1. Taj Mahal

Figure- 14 A photo of the Taj Mahal

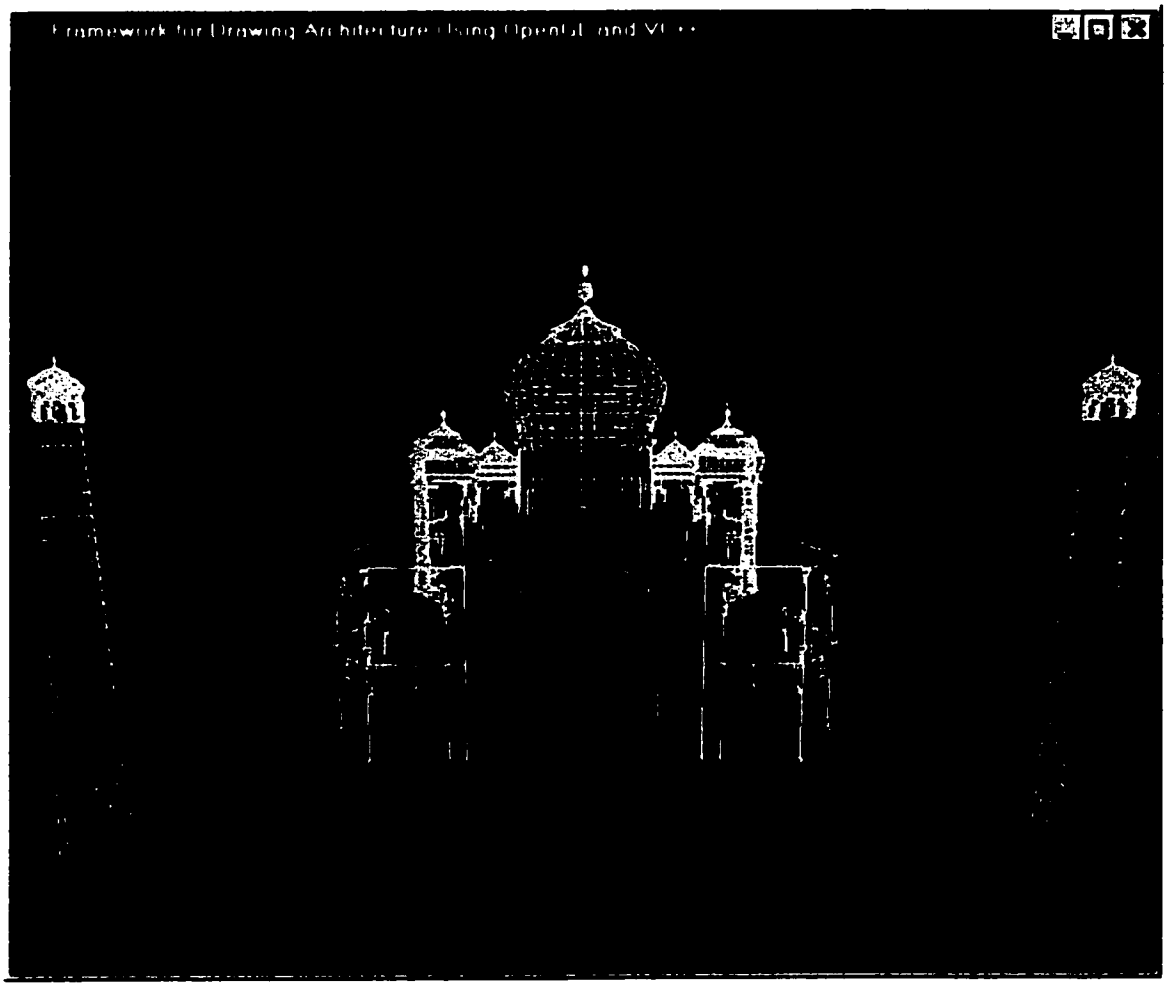

Figure- 15 The picture drawn by drawTaj() in the class of TajMahal

When the program is running and other picture is displayed, user can press F2 to switch to the picture of Taj Mahal, press F1 to switch to Tian Tan, F3 to switch to Sphere.

4.1.1 How to generate 3D Model of Taj Mahal

After analyzing the architecture of the Taj Mahal, we found it is mainly composed of three totally different kinds of compartments, comprising the dome, the wall and the pillar, all of which can be drawn by the methods in our framework.

Domes can be regarded as a 2D curve on x-y plane rotating about y-axis, so we can draw them using the method drawRevolution(...) in the class of Revolution, and we have to write our own getProfile(...) to override the default method.

Pillars can be regarded as circles swept along the direction of the positive direction of y-axis, and so we can draw them with the method draw3DcurveSurface(...) in the class of CurveSurface, and we have to write our own special setFunctionForCurveSurface(...) to override the default method.

Walls can be regarded as polygons swept along a certain direction of x-axis, y-axis or z-axis. Therefore we can draw them with the method drawPrism(...) in the class of Prism.

After drawing the compartments, we can place them in the proper location by using the transformation functions provided by OpenGL, such as glTranslatef(...), glRotatef(...), and we can get the proper size by using glScalef(...) provided by OpenGL. glPushMatrix(...) and glPopMatrix(...) are used to make the transformation affecting only certain parts of the building.

Because most buildings are based on design symmetry, methods in the Auxiliary class CopyTools can be used to determine the values of points from known points. In this way we only sample points for one kind of compartment, and as for the points for each instance of such compartments in the building, we can derive their points using the methods of class CopyTools.

4.1.2 The class diagram of 3D model of Taj Mahal

Class TajMahal multiply inherits from classes Prism, TubePipe, Revolution, CurveSurface and BezierCurveSurface, and we have our own setFunctionForPrism(...), setFunctionForCurveSurface(...) and getProfile(...). Therefore we must rewrite them in

TajMahal class to override these methods.

```
#ifndef Application
#define Application
#include "mesh.h"
#include "prism.h"
#include "tube&pipe.h"
#include "revolution.h"
#include "curveSurface.h"
#include "bezierCurveSurface.h"
#endif
class TajMahal : public Prism, public TubePipe, public Revolution, public
CurveSurface, public BezierCurveSurface
{
public:
void drawTaj();
void setFunctionForPrism(Point3 p1, Point3& p2, Vector3 v, double alpha);
void setFunctionForCurveSurface(Point3& p, double u,double v);
void getProfile(Point3*& p, Point3 domePoints[], int& numPts);
};
```

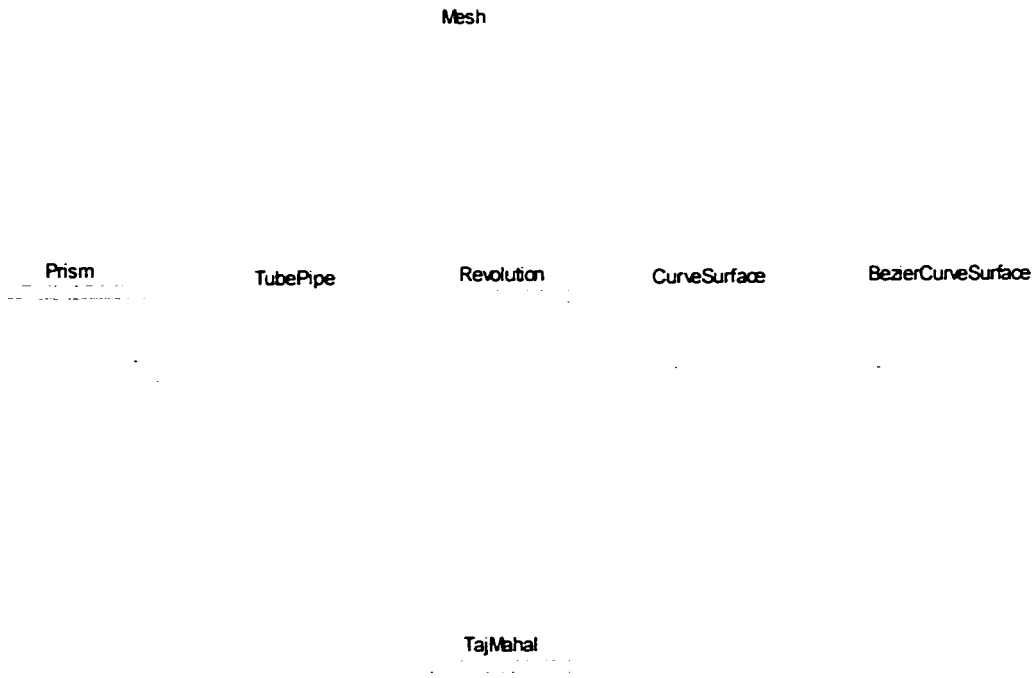

Figure- 16 The class diagram of the class of TajMahal

4.1.3 The implementation of drawing 3D model of Taj Mahal

```
#include <afxwin.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <math.h>
#include "TajMahal.h"
void TajMahal::drawTaj()
{
CopyTools copyTools;
Point3* revolutionPoints = NULL;
Point3 domePoints[31] =
{{0.0,172.0,0.0},{3.0,169.0,0.0},{2.5,167.5,0.0},{2.5,165.0,0.0},
{0.0,160.0,0.0},{5.0,158.0,0.0},{6.0,149.5,0.0},{0.0,145.0,0.0},{3.0,143.0,0.0},
{5.0,140.0,0.0},{9.0,135.0,0.0},{16.5,130.0,0.0},{26.5,125.0,0.0},{23.5,122.5,0.0},
{34.0,115.0,0.0},{39.5,110.0,0.0},{46.0,105.0,0.0},{50.0,100.0,0.0},{53.0,95.0,0.0},
{56.0,90.0,0.0},{57.5,85.0,0.0},{56.5,80.0,0.0},{55.5,75.0,0.0},{53.5,70.0,0.0},
{50.5,65.0,0.0},{48.0,62.5,0.0},{48.0,55.0,0.0},{51.0,50.0,0.0},{48.0,45.0,0.0},
{48.0,45.0,0.0},{48.0,0.0,0.0}};
Point3 cPoints[9] = {{0.0,28.0,0.0},{6.0,25.0,0.0},{7.0,24.0,0.0},{9.0,21.5,0.0},
{9.5,18.5,0.0},{9.5,0.0,0.0},{15.5,0.0,0.0},{15.5,36.0,0.0},{0.0,36.0,0.0}};
Point3 myPoints[10][9];
// declarations of other variables
...
```

// Drawing the biggest dome

/* Domes can be regarded as a 2D curve on x-y plane rotating about y-axis, so we can draw them using the method drawRevolution(...) in the class of Revolution, and we have to write our own getProfile(...) to override the default method. */

```
drawRevolution(revolutionPoints,domePoints,31,36,2*PI);
```

/* Walls can be regarded as polygons swept along a certain direction of x-axis, y-axis or z-axis. Therefore we can draw them with the method drawPrism(...) in the class of Prism. In drawTaj(), we use methods in the class copyTools to transpose walls into proper location and to scale them into proper size. */

// Drawing arched walls

```
for (int i = 0; i < 4; i++)
{
.....
glPushMatrix();
.....
if ( i % 2 == 0)
copyTools.scaleCopy(myPoints[0],myPoints[2],9,3.548*xScale,3.548*yScale,1.0);
else
copyTools.scaleCopy(myPoints[0],myPoints[2],9,1.0,3.548*yScale,3.548*xScale);
copyTools.linearCopy(myPoints[2], myPoints[2],9,55.0 + 15.5*xScale*3.548,sign1);
drawPrism(myPoints[2],9,2,thickness,sign2);
copyTools.mirrorCopy(myPoints[2], myPoints[3],9, 55.0 + 15.5 * xScale*3.548, sign1);
drawPrism(myPoints[3],9,2,thickness,sign2);
copyTools.linearCopy(myPoints[2], myPoints[4],9,3.548*yScale*36,'y');
drawPrism(myPoints[4],9,2,thickness,sign2);
copyTools.mirrorCopy(myPoints[4],myPoints[5],9,55.0+15.5*xScale*3.548,sign1);
drawPrism(myPoints[5],9,2,thickness,sign2);
copyTools.mirrorCopy(myPoints[2],myPoints[6],9,0,sign1);
copyTools.mirrorCopy(myPoints[3],myPoints[7],9,0,sign1);
copyTools.mirrorCopy(myPoints[4],myPoints[8],9,0,sign1);
copyTools.mirrorCopy(myPoints[5],myPoints[9],9,0,sign1);
drawPrism(myPoints[6],9,2,thickness,sign2);
drawPrism(myPoints[7],9,2,thickness,sign2);
drawPrism(myPoints[8],9,2,thickness,sign2);
drawPrism(myPoints[9],9,2,thickness,sign2);
glPopMatrix();
}
// Drawing four polars around temple
/* The pillars around Taj can be regarded as circles swept along the positive direction
of y-axis, and these pillars become thinner gradually at one end, so we can draw them with
the method draw3DcurveSurface(...) in the class of CurveSurface, and we have to write
our own special setFunctionForCurveSurface(...) to override the default method. */

for (i = 0; i < 4; i++)
{
float sign1,sign2;
glPushMatrix();
if (i < 2)
    sign1 = +1.0;
```

```
    else
        sign1 = -1.0;
    if ( i == 1 || i ==2)
        sign2 = -1.0;
    else
        sign2 = +1.0;
    glTranslated(sign1*180.0,-30*3.5480,sign2*180.0);
    glScaled(2.0,2.6,2.0);
    glRotatef(90.0,1.0,0.0,0.0);
    glTranslated(0.0,0.0,-132.567);
    drawCurveSurface(20, 20, 0., 2*PI, 73.540f, 132.567f);
    glPopMatrix();
    }
    //Drawing four smaller domes
/* It is similar to the drawing of the biggest dome, so the codes for drawing them are
omitted. */
    .....

    }
    void TajMahal::setFunctionForPrism(Point3 p1, Point3& p2, Vector3 v, double
alpha)
    {
    p2.x = p1.x + alpha * v.x;
    p2.y = p1.y + alpha * v.y;
    p2.z = p1.z + alpha * v.z;
    }
/* The pillars around Taj can be regarded as circles swept along the direction of the positive
direction of y-axis, these pillars become thinner gradually at one end. They are different
from the default curve surfaces generated by wrapping a circle about helix. Therefore Class
TajMahal has its own setFunctionForCurveSurface(...) to override the method in its parent
class. */
    void TajMahal::setFunctionForCurveSurface(Point3& p, double u, double v)
    {
        float b = 1;
        p.x = v/tan(PI/2-0.01745*2.16)*cos(u);
        p.y = v/tan(PI/2-0.01745*2.16)*sin(u);
        p.z = b*v;
    }
```

/* Each point in the curve profile for drawing revolution surface has to be passed into Point3 p, regardless how application developers get the points for its curve profile. In the case of drawing Taj Mahal, we set curve profile points into array variable domePoints, then getProfile(...) will duplicate domePoints into array variable Point3 p. */

```
void TajMahal::getProfile(Point3*& p,  Point3 domePoints[], int& numPts)
{
    p = new Point3[numPts];
    for (int i = 0; i < numPts; i++)
    {
        p[i].x = domePoints[i].x;
        p[i].y = domePoints[i].y;
        p[i].z = domePoints[i].z;
    }
}
```

4.2. Tian Tan

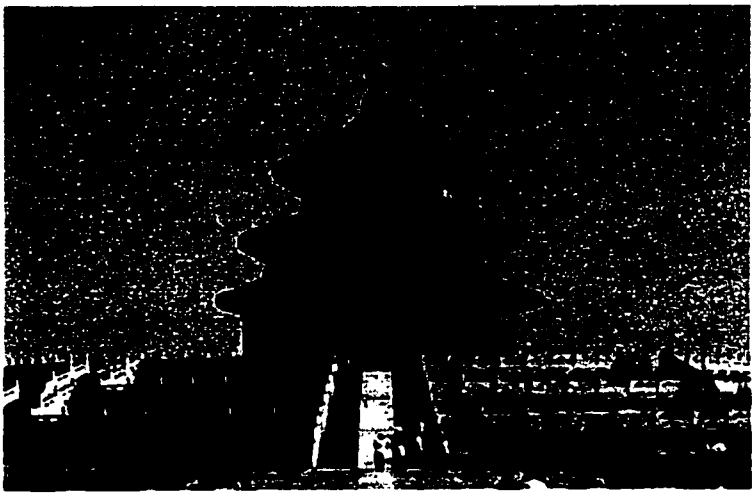

Figure- 17 The photo of TianTan

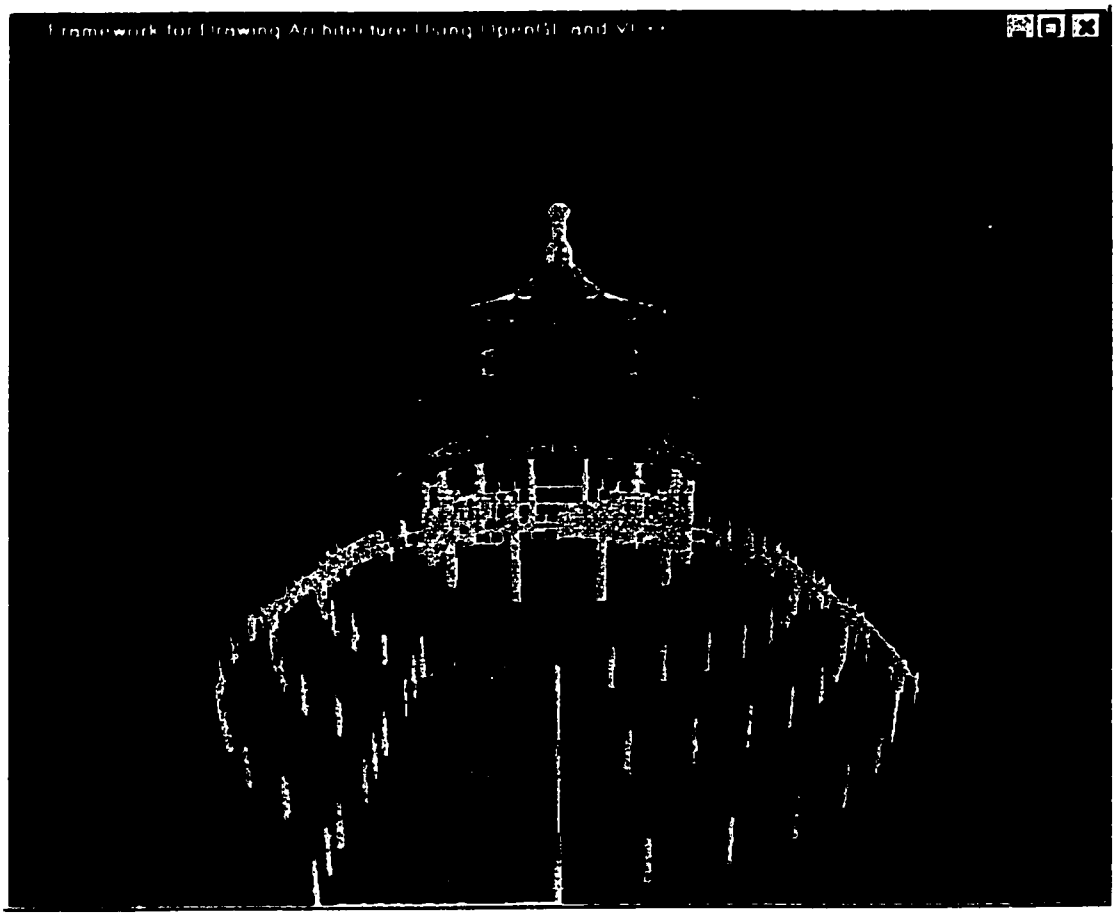

Figure- 18 The picture drawn by drawTianTan() in the class of TianTan

4.2.1 How to generate 3D Model of Tian Tan

The most significant characteristic of the architecture of Tian Tan is that most of its compartments are round, in order to inform people quietly that it is a temple of heaven, since the ancient Chinese people considered heaven as round. While another temple of the earth displays the characteristics of squares, as in this instance they thought of the earth as being square shape.

The roof of Tian Tan can be regarded as a 2D curve rotating about axis-y, and since this 2D curve has no formulae, we can use a profile to simulate the 2D curve, so that it can be drawn using the method in class Revolution.

The pillars between the walls, the round bases and the small pillars between the railings along the round bases can be regarded as circles swept along axis-y. Since we can determine the formulae for 2D circles and thus 3D formulae for pillars, we can use the method in the class CurveSurface to draw these artifacts.

Stairs between the round bases can be regarded as 2D zigzag curve rotating about axis-y from a certain distance, we use a profile to describe the 2D zigzag curve, so it can be drawn with the method in the class of Revolution.

4.2.2 The class diagram of 3D model of Tian Tan

Class TiaTan multiply inherits from classes Prism, TubePipe, Revolution, CurveSurface and BezierCurveSurface, since we have our own setFunctionForPrism(...), setFunctionForCurveSurface(...) and getProfile(...), we have to rewrite them in TianTan class to override these methods.

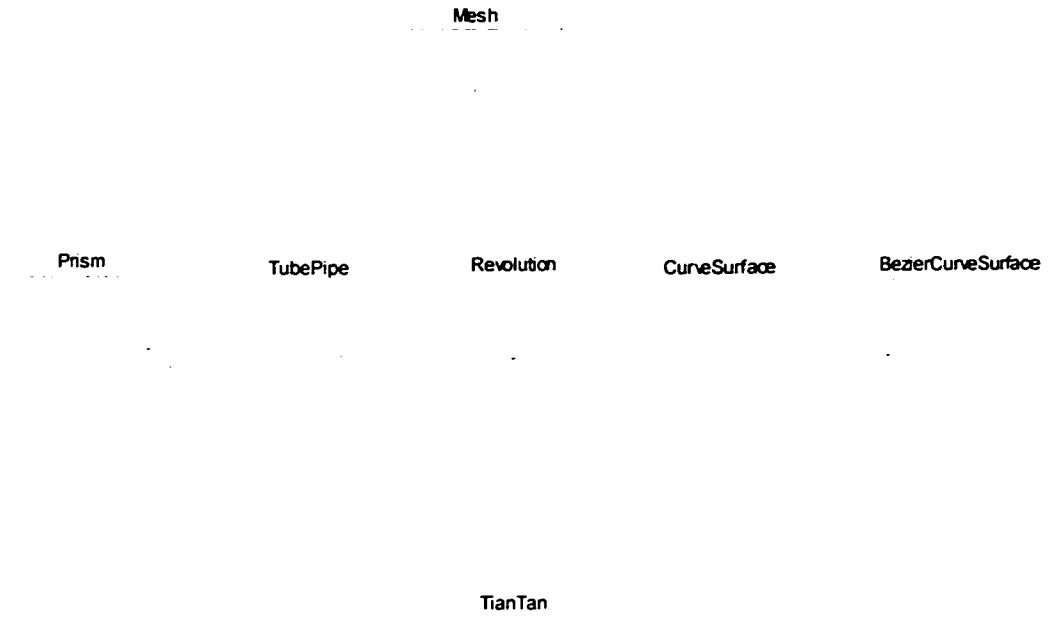

Figure- 19 The class diagram of the class of TianTan

```
#ifndef Application
#define Application
#include "mesh.h"
#include "prism.h"
#include "tube&pipe.h"
#include "revolution.h"
#include "curveSurface.h"
#include "bezierCurveSurface.h"
#endif
class TianTan : public Prism, public TubePipe, public Revolution, public
CurveSurface, public BezierCurveSurface
{
public:
void drawTianTan();
void setFunctionForPrism(Point3 p1, Point3& p2, Vector3 v, double alpha);
void setFunctionForCurveSurface(Point3& p, double u,double v);
void getProfile(Point3*& p, Point3 domePoints[], int& numPts);
};
```

4.2.3 The implementation of drawing 3D model of Tian Tan

```
#include <afxwin.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <math.h>
#include "TianTan.h"
void TianTan::drawTianTan()
{

Point3* revolutionPoints = NULL;
Point3 domePoints[30] =
{{0.0,171.0,0.0},{4.0,170.0,0.0},{7.5,167.,0.0},{7,163.5,0.0},
{5.0,160.0,0.0},{5.0,157.0,0.0},{6.0,156.0,0.0},{5.0,155.0,0.0},{5.0,150.0,0.0},
{7.0,146.5,0.0},{9.0,142.5,0.0},{9.0,136.5,0.0},{20.,124.0,0.0},{30.0,116.5,0.0},
{48.5,109.0,0.0},{56.5,106.0,0.0},{65.0,105.0,0.0},{65.0,95.,0.0},{53.5,81.0,0.0},
{48.5,80.0,0.0},{48.5,67.0,0.0},{88.5,51.5,0.0},{88.5,45.5,0.0},{70.0,33.,0.0},
{70.0,27.0,0.0},{72.0,25.0,0.0},{72.0,21.0,0.0},{102.0,8.0,0.0},{102.0,3.5,0.0},
{86.0,10.,0.0}};
Point3 stairPoints1[20] = {{0.,4*3.,0.},{4*.5,4*3.,0.},{4*.5,4*2.5,0.},
{4*1.,4*2.5,0.},{4*1.,4*2.,0.},{4*1.5,4*2.,0.},{4*1.5,4*1.5,0.},
{4*2.,4*1.5,0.},{4*2.,4*1.0,0.},{4*2.5,4*1.0,0.},{4*2.5,4*0.5,0.},
{4*3.0,4*0.5,0.0},{4*3.0,0.0,0.0},{4*3.0,-4*0.5,0.},{4*3.5,-4*0.5,0.0},
{4*3.5,-4*1.0,0.0},{4*4.0,-4*1.0,0.0},{4*4.0,-4*1.5,0.0},{4*4.5,-4*1.5,0.0},
{4*5.0,-4*1.5,0.0}};
Point3 stairPoints2[20];
double basis,thickness,xScale,yScale;
double alpha;
basis = 32.526;
thickness = basis/5;
xScale = 0.419;
yScale = 0.361;

//drawing the roof
/* The roof of Tian Tan can be regarded as a 2D curve rotating about axis-y, and since
this 2D curve has no formulae, we use a profile to simulate the 2D curve, so that it can be
drawn with the method drawRevolution(...) in class Revolution. */

drawRevolution(revolutionPoints, domePoints,30,36, 2*PI);

//drawing pillars between the walls
```

/* The pillars between the walls, the round bases and the small pillars between the railings along the round bases can be regarded as circles swept along axis-y, since we can determine the formulas for 2D circles and thus 3D formulas for pillars, we can use method drawCurveSurface(...) in class CurveSurface to draw these artifacts. We only use OpenGL transformation functions to transpose objects into their proper location and direction and to scale them into proper size. */

```
alpha = 360.0/12.0;
for (int i = 0; i < 12; i++)
{
glPushMatrix();
glRotated(i * alpha + alpha/2,0.0,1.0,0.0);
glTranslatef(83.5,0.0,0.0);
glRotatef(90.0,1.,0.,0.);
glScaled(2.5,2.5,0.55);
drawCurveSurface(20,20,0.0,2*PI,0,100);
glPopMatrix();
}
```

//drawing the walls between pillars

/* Walls can be regarded as polygons swept along a certain direction of x-axis, y-axis or z-axis, therefore we can draw them with the method drawPrism(...) in the class of Prism.*/

```
alpha = 360.0/12.0;
for ( i = 0; i < 12; i++)
{
glPushMatrix();
glRotated(i * alpha,0.0,1.0,0.0);
glTranslatef(0.0,-55.,83.5);
drawPrism(cPoints,28,2,thickness,'z');
glPopMatrix();
}
```

// drawing the round bases

// drawing the balcony

/* It is similar to the drawing of pillars, so the codes for drawing them are omitted. */

.....

// drawing stairs

/*Stairs between the round bases can be regarded as 2D zigzag curve rotating about axis-y from a certain distance, and we use a profile to describe the 2D zigzag curve, so it can be drawn with method drawRevolution(...) in the class of Revolution. */

```
for ( i = 0; i < 4; i++)
{
    glPushMatrix();
    for (int j = 0; j < 3; j++)
    {
        for ( m = 0; m < 20; m++)
        {
           stairPoints2[m].x = stairPoints1[m].x + 148 + j * 25;
            stairPoints2[m].y = stairPoints1[m].y;
            stairPoints2[m].z = stairPoints1[m].z;
        }
        glPushMatrix();
        glTranslated(0.0,-(65+j*20),0.0);
        glRotated(90*i + (3*360.0/16),0.0,1.0,0.0);
        drawRevolution(revolutionPoints, stairPoints2, 16, 8, 2*PI/16);
        glPopMatrix();
    }
    glPopMatrix();
}
```

// drawing arms

/* It is similar to the drawing of pillars, so the codes for drawing them are omitted. */

.....

```
}

void TianTan::setFunctionForPrism(Point3 p1, Point3& p2, Vector3 v, double alpha)
{
p2.x = p1.x + alpha * v.x;
p2.y = p1.y + alpha * v.y;
p2.z = p1.z + alpha * v.z;
}
```

/* The pillar can be regarded as a 3D surface generated by sweeping a circle along one of the axis. It is different from the default curve surface generated by wrapping a circle about helix, therefore Tian Tan has its own setFunctionForCurveSurface(...) to override the method in its parent class. */

```
void TianTan::setFunctionForCurveSurface(Point3& p, double u, double v)
{
    float b = 1;
    p.x = cos(u);
    p.y = sin(u);
    p.z = b*v;

}
```

/* Each point in the curve profile for drawing revolution surface has to be passed into array variable Point3 p, regardless how application developers get the points for its curve profile. In the case of drawing Tian Tan, we set curve profile points into array variable domePoints, then getProfile(...) will duplicate domePoints into array variable Point3 p. */

```
void TianTan::getProfile(Point3*& p, Point3 domePoints[], int& numPts)
{
    p = new Point3[numPts];
    for (int i = 0; i < numPts; i++)
    {
        p[i].x = domePoints[i].x;
        p[i].y = domePoints[i].y;
        p[i].z = domePoints[i].z;
    }
}
```

5. Conclusion

While we developed the typical application for drawing Taj Mahal, we designed and implemented certain classes and methods for drawing different kinds of compartments in this building. After analyzing the possible use of each kind of compartment in the architecture domain, we can then improve the corresponding methods and classes, so that they can be reused in other applications. In this way, we can derive a framework for drawing the architecture of the building.

By using this framework, we can implement the second application for drawing Tian Tan very easily, while we can also improve certain methods and classes in the framework to enable them to develop more flexibility in drawing the architecture of building.

From the above introduction of the design and the implementation of the framework, as well as two sample applications, we can conclude that it is feasible to develop a practical OpenGL framework to help application developers in drawing the architectures of buildings. We can also conclude that successful frameworks will permit the reuse of design, classes and codes, and thus improve the efficiency and quality of application software development.

Besides drawing the architecture of buildings, people can also design OpenGL frameworks for other domains, as with rendering Piping in Refinery and Chemical works.

6. Readme file for running applications

All the files are stored in the directory of frameOne. Users can use VC++6.0 to open the workspace file: frameOne. and they can then use build to compile and link the files to generate executable file frameOne.exe.

While the application program is running. users can press F1 to switch to TianTan. F2 to Taj Mahal. F3 to Sphere. F4 to the Tube as Prism swept non-linearly. F5 to Pipe as 3D curve with formulae. and finally F6 to 3D curve based on Bezier Curve.

References

[1] Getting Started with OpenGL. Peter Grogono. Concordia University. 1998.

[2] Computer Graphics using OpenGL. F.S.Hill. JR. Prentice Hall 2001

[3] OpenGL Programming Guide. Mason Woo. Jackie Neider. Tom Davis. Dave Shreiner. Addison Wesley. 1999

[4] Interactive Computer Graphics. A top-down approach with OpenGL. Edward Angel. Addison Wesley. 1997

[5] Implementing Application Frameworks. Object-Oriented Frameworks at Work. Mohamed E.Fayad. Douglas C.Schmidt. Ralph E. Johnson. Addison Wesley. 1999

[6] Object Oriented Methods. Principles & Practice. Ian Graham.Addison Wesley. 2001

[7] Programming with C++. John Hubbard. McGraw Hill. 1996

[8] Unified Objects. Object-Oriented Programming Using C++. Babak Sadr. IEEE Computer Society. 1998

[9] Frameworks = (components + patterns). Johnson R.E.. Communications of the ACM. October 1997

[10]Object Oriented application frameworks. Fayad M.E. and D.C. Schmidt. Communications of the ACM. October 1997